JOP Reference Handbook

JOP Reference Handbook

Building Embedded Systems with a Java Processor

Martin Schoeberl

Martin Schoeberl
Strausseng. 2-10/2/55
A-1050 Vienna, Austria

Email: `martin@jopdesign.com`
Visit the accompanying web site on `http://www.jopdesign.com/` and
the JOP Wiki at `http://www.jopwiki.com/`

Published 2009 by CreateSpace,
`http://www.createspace.com/`

Library of Congress Cataloging-in-Publication Data

Schoeberl, Martin

JOP Reference Handbook: Building Embedded Systems
with a Java Processor / Martin Schoeberl
Includes bibliographical references and index.
ISBN 978-1438239699

Manufactured in the United States of America.
Typeset in 11pt Times by Martin Schoeberl

Foreword

This book is about JOP, the Java Optimized Processor. JOP is an implementation of the Java virtual machine (JVM) in hardware. The main implementation platform is a field-programmable gate array (FPGA). JOP began as a research project for a PhD thesis. In the mean time, JOP has been used in several industrial applications and as a research platform. JOP is a time-predictable processor for hard real-time systems implemented in Java.

JOP is open-source under the GNU GPL and has a growing user base. This book is written for all of you who build this lively community. For a long time the PhD thesis, some research papers, and the web site have been the main documentation for JOP. A PhD thesis focus is on research results and implementation details are usually omitted. This book complements the thesis and provides insight into the implementation of JOP and the accompanying JVM. Furthermore, it gives you an idea how to build an embedded real-time system based on JOP.

Acknowledgements

Many users of JOP contributed to the design of JOP and to the tool chain. I also want to thank the students at the Vienna University of Technology during the four years of the course “The JVM in Hardware” and the students from CBS, Copenhagen at an embedded systems course in Java for the interesting questions and discussions. Furthermore, the questions and discussions in the Java processor mailing list provided valuable input for the documentation now available in form of this book. The following list of contributors to JOP is roughly in chronological order.

Ed Anuff wrote testmon.asm to perform a memory interface test and BlockGen.java to convert Altera .mif files to Xilinx memory blocks. Flavius Gruian wrote the initial version of JOPizer to generate the .jop file from the application classes. JOPizer is based on the open source library BCEL and is a substitute to the formerly used JavaCodeCompact from Sun. Peter Schrammel and Christof Pitter have implemented the first version of long bytecodes. Rasmus Pedersen based a class on very small information systems on JOP and invited my to co-teach this class in Copenhagen. During this time the first version of the WCET analysis tool was developed by Rasmus. Rasmus has also implemented an Eclipse plugin for the JOP design flow. Alexander Dejaco and Peter Hilber have developed the I/O interface board for the LEGO Mindstorms. Christof Pitter designed and implemented the chip-multiprocessor (CMP) version of JOP during his PhD thesis. Wolfgang Puffitsch first contribution to JOP was the finalization of the floating point library SoftFloat. Wolfgang, now an active developer of JOP, contributed several enhancements (e.g., exceptions, HW field access, data cache,...) and works towards real-time garbage collection for the CMP version of JOP. Alberto Andriotti contributed several JVM test cases. Stefan Hepp has implemented an optimizer at bytecode level during his Bachelor thesis work. Benedikt Huber has redesigned the WCET analysis tool for JOP during his Master’s thesis. Trevor Harmon, who implemented the WCET tool Volta for JOP during his PhD thesis, helped me with proofreading of the handbook.

Furthermore, I would like to thank Walter Wilhelm from EEG for taking the risk to accept a JOP based hardware for the *Kippfahrleitung* project at a very early development stage of JOP. The development of JOP has received funding from the Wiener Innovationsföderprogram (Call IKT 2004) and from the EU project JEOPARD.

Contents

1 Introduction

This handbook introduces a Java processor for embedded real-time systems, in particular the design of a small processor for resource-constrained devices with time-predictable execution of Java programs. This Java processor is called JOP – which stands for Java Optimized Processor –, based on the assumption that a full native implementation of all Java bytecode instructions is not a useful approach.

1.1 A Quick Tour on JOP

In the following section we will give a quick overview on JOP and a short description how to get JOP running within an FPGA. A detailed description of the build process can be found in Chapter 2. JOP is a soft-core written in VHDL plus tools in Java, a simplified Java library (JDK), and application examples. JOP is delivered in source only.

1.1.1 Building JOP and Running "Hello World"

To build JOP you first have to download the source tree. A *Makefile* (or an Ant file) contains all necessary steps to build the tools, the processor, and the application. Configuration of the FPGA and downloading the Java application is also part of the Makefile.

In this description we assume the FPGA board Cycore (see Appendix E.1). This board is the default target for the Makefile. The board has to be connected to the power supply and to the PC via a ByteBlaster download cable and a serial cable.

The FPGA is configured via the ByteBlaster cable. The Java application is downloaded after the FPGA configuration via the serial cable. Besides the download the serial cable is also used as a communication link between JOP and the PC. System.out and System.in represent this serial link on JOP.

In order to build the whole system you need a Java compiler[1] and an FPGA compiler. In our case we use the free web edition of Quartus from Altera.[2] As we use make and

[1]Download the Java SE Development Kit (JDK) from `http://java.sun.com/javase/downloads/index.jsp`.

[2]`http://www.altera.com/`

the preprocessor from the GNU compiler collection, Cygwin[3] should be installed under Windows.

When all tools are setup correctly[4] a simple make should build the tools, the processor, compile the “Hello World” example, configure the FPGA and download the application. The whole build process will take a few minutes. After typing

```
make
```

you see a lot of messages from the various tools. However, the last lines should be actual messages received from JOP. It should look similar to the following:

```
JOP start V 20080821
60 MHz, 1024 KB RAM, 1 CPUs
Hello World from JOP!

JVM exit!
```

Note that JOP prints some internal information, such as version and memory size, at startup. After that, the message “Hello World from JOP!” can be seen. Our first program runs on JOP!

As a next step, locate the Hello World example in the source tree[5] and change the output message. The tools and the processor have been built already. So we do not need to compile everything from scratch. Use the following make target to just compile the Java application and download the processor and the application:

```
make japp
```

The compile process should now be faster and the output similar to before.

The Hello World application is the default target in the Makefile. See Chapter 2 for a description how this target can be changed. In case you use a different FPGA board you can find information on how to change the build process also in Chapter 2.

1.1.2 The Design Structure

Browsing the source tree of JOP can give the impression that the design is complex. However, the basic structure is not that complex. The design consists of three entities:

1. The processor JOP

[3] http://www.cygwin.com/
[4] Check at the command prompt that javac is in the path.
[5] .../jop/java/target/src/test/test/HelloWorld.java

2. Supporting tools
3. The Java library and applications

The different entities are also reflected during the configuration and download process. The download is a two step process:

1. Configuration of the FPGA: JOP is downloaded via a FPGA download cable (e.g., ByteBlaster on the PCs parallel port). After FPGA configuration the processor automatically starts and listens to the second channel (the serial line) for the software download.
2. Java application download: the compiled and linked application is downloaded usually via a serial line. JOP stores the application in the main memory and starts execution at main() after the download.

Further details of the source structure can be found in Section 2.10.

1.2 A Short History

The first version of JOP was created in 2000 based on the adaptation of earlier processor designs created between 1995 and 2000. The first version was written in Altera's proprietary AHDL language. The first *program* (3 bytecode instructions) ran on JOP on October 2, 2000. The first approach was a general purpose accumulator/register machine with 16-bit instructions, 32-bit registers, and a pipeline length of 3. It used the on-chip block memory to implement (somehow unusual) 1024 registers.

The JVM was implemented in the assembler of that machine. That concept was similar to the microcode in the current JOP version. The decoding of the bytecode was performed by a long jump table. In the best case (assuming a local, single cycle memory) a simple bytecode (e.g. iadd) took 12 cycles for fetch and decode and additional 11 cycles for execution.

A redesign followed in April 2001, now coded in VHDL. The second version of JOP introduced features to speed up the implementation of the JVM with specific instructions for the stack access and a dedicated stack pointer. The register file was reduced to 16 entries and the instruction width reduced to 8 bits. The pipeline contained 5 stages and special support for decoding bytecode instructions was added – a first version of the dynamic bytecode to microcode address translation as it is used in the current version of JOP. The enhancements within JOP2 resulted in the reduction of the execution time for a simple bytecode to 3 cycles. A great enhancement, compared to the 23 cycles in JOP1.

The next redesign (JOP3) followed in June 2001. The challenge was to execute simple bytecodes fully pipelined in a single cycle. The microcode instruction set was changed to implement a stack machine and the execution stage combined with the on-chip stack cache. Microcode instructions where coded in 16 bit and the pipeline was reduced to four stages. JOP3 is the basis of JOP as it is described in this handbook. The later changes have not been so radical to call them a redesign.

The first real-world application of JOP was in the project *Kippfahrleitung* (see Section 11.4.1). At the start of the project (October 2001) JOP could only execute a single static method stored in the on-chip memory. The project greatly pushed the development of JOP. After successful deployment of the JOP-based control system in the field, several projects followed (TeleAlarm, Lift, the railway control system). The source of the commercial applications is part of the JOP distribution. Some of these applications are now used as a test bench for embedded Java performance and to benchmark WCET analysis tools.

More details and the source code of JOP1[6], JOP2[7] and the first JOP3[8] version are available on the web site.

1.3 JOP Features

This book presents a hardware implementation of the Java virtual machine (JVM), targeting small embedded systems with real-time constraints. JOP is designed from the ground up with time-predictable execution of Java bytecode as a major design goal. All functional units, and especially the interactions between them, are carefully designed to avoid any time dependency between bytecodes.

JOP is a stack computer with its own instruction set, called microcode in this book. Java bytecodes are translated into microcode instructions or sequences of microcode. The difference between the JVM and JOP is best described as the following:

> The JVM is a CISC stack architecture, whereas JOP is a RISC stack architecture.

The architectural features and highlights of JOP are:

- Dynamic translation of the CISC Java bytecodes to a RISC, stack based instruction set (the microcode) that can be executed in a 3 stage pipeline.

[6]`http://www.jopdesign.com/jop1.jsp`
[7]`http://www.jopdesign.com/jop2.jsp`
[8]`http://www.jopdesign.com/jop3.jsp`

- The translation takes exactly one cycle per bytecode and is therefore pipelined. Compared to other forms of dynamic code translation the translation does not add any variable latency to the execution time and is therefore time predictable.
- Interrupts are inserted in the translation stage as special bytecodes and are transparent to the microcode pipeline.
- The short pipeline (4 stages) results in short conditional branch delays and a hard to analyze branch prediction logic or branch target buffer can be avoided.
- Simple execution stage with the two topmost stack elements as discrete registers. No write back stage or forwarding logic is needed.
- Constant execution time (one cycle) for all microcode instructions. The microcode pipeline never stalls. Loads and stores of object fields are handled explicitly.
- No time dependencies between bytecodes result in a simple processor model for the low-level WCET analysis.
- Time predictable instruction cache that caches whole methods. Only invoke and return instruction can result in a cache miss. All other instructions are guaranteed cache hits.
- Time predictable data cache for local variables and the operand stack. Access to local variables is a guaranteed hit and no pipeline stall can happen. Stack cache fill and spill is under microcode control and analyzable.
- No prefetch buffers or store buffers that can introduce unbounded time dependencies of instructions. Even simple processors can contain an instruction prefetch buffer that prohibits exact WCET values. The design of the method cache and the translation unit avoids the variable latency of a prefetch buffer.
- Good average case performance compared with other non real-time Java processors.
- Avoidance of hard to analyze architectural features results in a very small design. Therefore an available real estate can be used for a chip multi-processor solution.
- JOP is the smallest hardware implementation of the JVM available to date. This fact enables usage of low-cost FPGAs in embedded systems. The resource usage of JOP can be configured to trade size against performance for different application domains.

- JOP is actually in use in several real-world applications showing that a Java based embedded system implemented in an FPGA is a viable option.

JOP is implemented as a soft-core in a field programmable gate array (FPGA) giving a lot of flexibility for the overall hardware design. The processor can easily be extended by peripheral components inside the same chip. Therefore, it is possible to customize the solution exactly to the needs of the system.

1.4 Is JOP the Solution for Your Problem?

I had a lot of fun, and still have, developing and using JOP. However, should you use JOP? JOP is a processor design intended as a time predictable solution for hard real-time systems. If your application or research focus is on those systems and you prefer Java as programming language, JOP is the right choice. If you are interested in larger, dynamic systems, JOP is the wrong choice. If average performance is important for you and you do not care about worst-case performance other solutions will probably do a better job.

1.5 Outline of the Book

Chapter 2 gives a detailed introduction into the design flow of JOP. It explains how the individual parts are compiled and which files have to be changed when you want to extend JOP or adapt it to a new hardware platform. The chapter is concluded by an exercise to explore the different steps in the design flow.

Chapter 3 provides background information on the Java programming language, the execution environment, and the Java virtual machine, for Java applications. If you are already familiar with Java and the JVM, feel free to skip this chapter.

Chapter 4 is the main chapter in which the architecture of JOP is described. The motivation behind different design decisions is given. A Java processor alone is not a complete JVM. Chapter 5 describes the runtime environment on top of JOP, including the definition of a real-time profile for Java and the description of the scheduler in Java.

In Chapter 6 worst-case execution time (WCET) analysis for JOP is presented. It is shown how the time-predictable bytecode instructions form the basis of WCET analysis of Java applications.

Garbage collection (GC) is an important part of the Java technology. Even in real-time systems new real-time garbage collectors emerge. In Chapter 7 the formulas to calculate the correct scheduling of the GC thread are given and the implementation of the real-time GC for JOP is explained.

JOP uses a simple and efficient system-on-chip interconnection called SimpCon to connect the memory controller and peripheral devices to the processor pipeline. The definition of SimpCon and the rationale behind the SimpCon specification is given in Chapter 9. Based on a SimpCon memory arbiter, chip-multiprocessor (CMP) versions of JOP can be configured. Chapter 10 gives some background information on the JOP CMP system.

In Chapter 11, JOP is evaluated with respect to size and performance. This is followed by a description of some commercial real-world applications of JOP. Other hardware implementations of the JVM are presented in Chapter 12. Different hardware solutions from both academia and industry for accelerating Java in embedded systems are analyzed.

Finally, in Chapter 13, the work is summarized and the major contributions are presented. This chapter concludes with directions for future work using JOP and real-time Java. A more theoretical treatment of the design of JOP can be found in the PhD thesis [123], which is also available as book [131].

2 The Design Flow

This chapter describes the design flow for JOP — how to build the Java processor and a Java application from scratch (the VHDL and Java sources) and download the processor to an FPGA and the Java application to the processor.

2.1 Introduction

JOP [123], the Java optimized processor, is an open-source development platform available for different targets (Altera and Xilinx FPGAs and various types of FPGA boards). To support several targets, the resulting design-flow is a little bit complicated. There is a Makefile available and when everything is set up correctly, a simple

```
make
```

should build everything from the sources and download a *Hello World* example. However, to customize the Makefile for a different target it is necessary to understand the complete design flow. It should be noted that an Ant[1] based build process is also available.

2.1.1 Tools

All needed tools are freely available.

- Java SE Development Kit (JDK) Java compiler and runtime
- Cygwin GNU tools for Windows. Packages cvs, gcc and make are needed
- Quarts II Web Edition VHDL synthesis, place and route for Altera FPGAs

The PATH variable should contain entries to the executables of all packages (java and javac, Cygwin bin, and Quartus executables). Check the PATH at the command prompt with:

[1] http://ant.apache.org/

```
javac
gcc
make
git
quartus_map
```

All the executables should be found and usually report their usage.

2.1.2 Getting Started

This section shows a quick step-by-step build of JOP for the Cyclone target in the minimal configuration. All directory paths are given relative to the JOP root directory jop. The build process is explained in more detail in one of the following sections.

Download the Source

Create a working directory and download JOP from the GIT server:

```
git clone git://www.soc.tuwien.ac.at/jop.git
```

For a write access clone (for developers) use following URL:

```
git clone ssh://user@www.soc.tuwien.ac.at/home/git/jop.git
```

All sources are downloaded to a directory jop. For the following command change to this directory. Create the needed directories with:

```
make directories
```

Tools

The tools contain Jopa, the microcode assembler, JopSim, a Java based simulation of JOP, and JOPizer, the application builder. The tools are built with following make command:

```
make tools
```

Assemble the Microcode JVM, Compile the Processor

The JVM configured to download the Java application from the serial interface is built with:

```
make jopser
```

This command also invokes Quartus to build the processer. If you want to build it within Quartus follow the following instructions:

1. Start Quartus II and open the project jop.qpf from directory quartus/cycmin in Quartus with *File – Open Project...*.
2. Start the compiler and fitter with *Processing – Start Compilation*.
3. After successful compilation the FPGA is configured with *Tools – Programmer* and *Start*.

Compiling and Downloading the Java Application

A simple *Hello World* application is the default application in the Makefile. It is built and downloaded to JOP with:

```
make japp
```

The "Hello World" message should be printed in the command window.
For a different application change the Makefile targets or override the make variables at the command line. The following example builds and runs some benchmarks on JOP:

```
make japp −e P1=bench P2=jbe P3=DoAll
```

The three variables P1, P2, and P3 are a shortcut to set the directory, the package name, and the main class of the application.

USB based Boards

Several Altera based boards use an FTDI FT2232 USB chip for the FPGA and Java program download. To change the download flow for those boards change the value of the following variable in the Makefile to true:

```
USB=true
```

The Java download channel is mapped to a virtual serial port on the PC. Check the port number in the system properties and set the variable COM_PORT accordingly.

2.1.3 Xilinx Spartan-3 Starter Kit

The Xilinx tool chain is still not well supported by the Makefile or the Ant design flow. Here is a short list on how to build JOP for a Xilinx board:

```
make tools
cd asm
jopser
cd ..
```

Now start the Xilinx IDE wirh the project file jop.npl. It will be converted to a new (binary) jop.ise project. The .npl project file is used as it is simple to edit (ASCII).

- Generate JOP by double clicking 'Generate PROM, ACE, or JTAG File'
- Configure the FPGA according to the board type

The above is a one step build for the processor. The Java application is built and downloaded by:

```
make java_app
make download
```

Now your first Java program runs on JOP/Spartan-3!

2.2 Booting JOP — How Your Application Starts

Basically this is a two step process: (a) configuration of the FPGA and (b) downloading the Java application. There are different possibilities to perform these steps.

2.2.1 FPGA Configuration

FPGAs are usually SRAM based and *lose* their configuration after power down. Therefore the configuration has to be loaded on power up. For development the FPGA can be configured via a download cable (with JTAG commands). This can be done within the IDEs from Altera and Xilinx or with command line tools such as quartus_pgm or jbi32.

For the device to boot automatically, the configuration has to be stored in non volatile memory such as Flash. Serial Flash is directly supported by an FPGA to boot on power up. Another method is to use a standard parallel Flash to store the configuration and additional data (e.g. the Java application). A small PLD reads the configuration data from the Flash and shifts it into the FPGA. This method is used on the Cyclone and ACEX boards.

2.2.2 Java Download

When the FPGA is configured the Java application has to be downloaded into the main memory. This download is performed in microcode as part of the JVM startup sequence. The application is a .jop file generated by JOPizer. At the moment there are three options:

Serial line JOP listens to the serial line and the data is written into the main memory. A simple echo protocol performs the flow control. The baud rate is usually 115 kBaud.

USB Similar to the serial line version, JOP listens to the parallel interface of the FTDI FT2232 USB chip. The FT2232 performs the flow control at the USB level and the echo protocol is omitted.

Flash For stand alone applications the Java program is copied from the Flash (relative Flash address 0, mapped Flash address is 0x80000[2]) to the main memory (usually a 32-bit SRAM).

The mode of downloading is defined in the JVM (jvm.asm). To select a new mode, the JVM has to be assembled and the complete processor has to be rebuilt – a full make run. The generation is performed by the C preprocessor (gcc) on jvm.asm. The serial version is generated by default; the USB or Flash version are generated by defining the preprocessor variables USB or FLASH.

VHDL Simulation To speed up the VHDL simulation in ModelSim there is a forth method where the Java application is loaded by the test bench instead of JOP. This version is generated by defining SIMULATION. The actual Java application is written by jop2dat into a plain text file (mem_main.dat) and read by the simulation test bench into the simulated main memory.

There are four small batch-files in directory asm that perform the JVM generation: jopser, jopusb, jopflash, and jopsim.

2.2.3 Combinations

Theoretically all variants to configure the FPGA can be combined with all variations to download the Java application. However, only two combinations are useful:

1. For VHDL or Java development configure the FPGA via the download cable and download the Java application via the serial line or USB.

2. For a stand-alone application load the configuration and the Java program from the Flash.

[2]All addresses in JOP are counted in 32-bit quantities. However, the Flash is connected only to the lower 8 bits of the data bus. Therefore a store of one word in the main memory needs four loads from the Flash.

2.2.4 Stand Alone Configuration

The Cycore board can be configured to configure the FPGA and load the Java program from Flash at power up. In order to prepare the Cycore board for this configuration the Flash must be programmed. Depending on the I/O capabilities several options are possible:

SLIP With a SLIP connection the Flash can be programmed via TFTP. For this configuration a second serial line is needed.

Ethernet With an Ethernet connection (e.g., the baseio board) TFTP can be used for Flash programming.

Serial Line With a single serial line the utilities util.Mem.java and amd.exe can be used to program the Flash.

The following text describes the Flash programming and PLD reconfiguration for a stand alone configuration. Fist we have to build a JOP version that will load a Java program from the Flash:

```
make jopflash
```

As usual a jop.sof file will be generated. For easier reading of the configuration it will be converted to jop.ttf. This file will be programmed into the Flash starting at address 0x40000. Therefore, we need to save that file and rebuild a JOP version that loads a Java program (the Flash programmer) from the serial line:

```
copy quartus\cycmin\jop.ttf  ttf\cycmin.ttf
make jopser
```

As a next step we will build the Java program that will be programmed into the Flash and save a copy of the .jop file. Hello.java is the embedded version of a *Hello World* program that blinks the WD LED at 1 Hz.

```
make java_app -e P1=test P2=test P3=Hello
copy java\target\dist\bin\Hello.jop  .
```

To program the Flash the programmer tool util.Mem will run on JOP and amd.exe is used at the PC side:

```
make japp -e P1=common P2=util P3=Mem COM_FLAG=
amd Hello.jop COM1
amd ttf\cycmin.ttf  COM1
```

As a last step the PLD will be programmed to enable FPGA configuration form the Flash:

```
make pld_conf
```

The board shall now boot after a power cycle and the LED will blink. To read the output from the serial line the small utility e.exe can be used.

In the case the PLD configuration shall be changed back to JTAG FPGA configuration following make command will reset the PLD:

```
make pld_init
```

Note, that in a stand alone configuration the watchdog (WD) pin has to be toggled every second (e.g., by invoking util.Timer.wd(). When the WD is not toggled the FPGA will be reconfigured after 1.6 seconds.

Due to wrong file permissions the Windows executables amd.exe and USBRunner.exe will not have the execution permission set. Change the setting with the Windows Explorer. The tool amd.exe can also be rebuilt with:

```
make cprog
```

2.3 The Design Flow

This section describes the design flow to build JOP in greater detail.

2.3.1 Tools

There are a few tools necessary to build and download JOP to the FPGA boards. Most of them are written in Java. Only the tools that access the serial line are written in C.[3]

Downloading

These little programs are already compiled and the binaries are checked in into the repository. The sources can be found in directory c_src.

down.exe The workhorse to download Java programs. The mandatory argument is the COM-port. Optional switch -e keeps the program running after the download and echoes the characters from the serial line (System.out in JOP) to stdout. Switch -usb disables the echo protocol to speed up the download over USB.

[3]The Java JDK still comes without the javax.comm package and getting this optional package correctly installed is not that easy.

e.exe Echoes the characters from the serial line to stdout. Parameter is the COM-port.

amd.exe A utility to send data over the serial line to program the on-board Flash. The complementary Java program util.Mem must be running on JOP.

USBRunner.exe Download the FPGA configuration via USB with the FTDI2232C chip (dpsio board).

Generation of Files

These tools are written in Java and are delivered in source form. The source can be found under java/tools/src and the class files are in jop-tools.jar in directory java/tools/dist/lib.

Jopa The JOP assembler. Assembles the microcoded JVM and produces on-chip memory initialization files and VHDL files.

BlockGen converts Altera memory initialization files to VHDL files for a Xilinx FPGA.

JOPizer links a Java application and converts the class information to the format that JOP expects (a .jop file). JOPizer uses the bytecode engineering library[4] (BCEL).

Simulation

JopSim reads a .jop file and executes it in a debug JVM written in Java. Command line option -Dlog="true" prints a log entry for each executed JVM bytecode.

pcsim simulates the BaseIO expansion board for Java debugging on a PC (using the JVM on the PC).

2.3.2 Targets

JOP has been successfully ported to several different FPGAs and boards. The main distribution contains the ports for the FPGAs:

- Altera Cyclone EP1C6 or EP1C12
- Xilinx Spartan-3
- Altera Cyclone-II (Altera DE2 board)

[4] `http://jakarta.apache.org/bcel/`

- Xilinx Virtex-4 (ML40x board)
- Xilinx Spartan-3E (Digilent Nexys 2 board)

For the current list of the supported FPGA boards see the list at the web site.[5] Besides the ports to different FPGAs there are ports to different boards.

Cyclone EP1C6/12

This board is the workhorse for the JOP development and comes in two versions: with an Cyclone EP1C6 or EP1C12. The schematics can be found in Appendix E.1. The board contains:

- Altera Cyclone EP1C6Q240 or EP1C12Q240 FPGA
- 1 MB fast SRAM
- 512 KB Flash (for FPGA configuration and program code)
- 32 MB NAND Flash
- ByteBlasterMV port
- Watchdog with LED
- EPM7064 PLD to configure the FPGA from the Flash (on watchdog reset)
- Voltage regulator (1V5)
- Crystal clock (20 MHz) at the PLL input (up to 640 MHz internal)
- Serial interface (MAX3232)
- 56 general purpose I/O pins

The Cyclone specific files are jopcyc.vhd or jopcyc12 and mem32.vhd. This FPGA board is designed as a module to be integrated with an application specific I/O-board. There exist following I/O-boards:

simpexp A simple bread board with a voltage regulator and a SUBD connector for the serial line

[5] `http://www.jopwiki.com/FPGA_boards`

I/O board	Quartus	I/O top level
simpexp, baseio	cycmin	scio_min.vhd
dspio	usbmin	scio_dspiomin.vhd
baseio	cycbaseio	scio_baseio.vhd
bg263	cybg	scio_bg.vhd
lego	cyclego	scio_lego.vhd
dspio	dspio	scio_dspio.vhd

Table 2.1: Quartus project directories and VHDL files for the different I/O boards

baseio A board with Ethernet connection and EMC protected digital I/O and analog input

bg263 Interface to a GPS receiver, a GPRS modem, keyboard and a display for a railway application

lego Interface to the sensors and motors of the LEGO Mindstorms. This board is a substitute for the LEGO RCX.

dspio Developed at the University of Technology Vienna, Austria for digital signal processing related work. All design files for this board are open-source.

Table 2.1 lists the related VHDL files and Quartus project directories for each I/O board.

Xilinx Spartan-3

The Spartan-3 specific files are jop_xs3.vhd and mem_xs3.vhd for the Xilinx Spartan-3 Starter Kit and jop_trenz.vhd and mem_trenz.vhd for the Trenz Retrocomputing board.

2.4 Eclipse

In folder eclipse there are four Eclipse projects that you can import into your Eclipse workspace. However, do not use *that* directory as your workspace directory. Choose a directory outside of the JOP source tree for the workspace (e.g., your usual Eclipse workspace) and copy the for project folders joptarget, joptools, pc, and pcsim.

All projects use the Eclipse path variable[6] JOP_HOME that has to point to the root directory (.../jop) of the JOP sources. Under *Window – Preferences...* select *General – Workspace*

[6]Eclipse (path) variables are workspace specific.

Project	Content
jop	The target sources
joptools	Tools such as Jopa, JopSim, and JOPizer
pc	Some PC utilities (e.g. Flash programming via UDP/IP)
pcsim	Simulation of the basio hardware on the PC

Table 2.2: Eclipse projects

– *Linked Resources* and create the path variable JOP_HOME with *New...*.

Import the projects with *File – Import..* and *Existing Projects into Workspace*. It is suggested to an Eclipse workspace that is not part of the jop source tree. Select as the root directory (e.g., your Eclipse workspace), select the projects you want to import, select *Copy projects into workspace*, and press *Finish*. Table 2.2 shows all available projects.

Add the libraries from .../jop/java/lib (as external archives) to the build path (right click on the joptools project) of the project joptools.[7]

2.5 Simulation

This section contains the information you need to get a simulation of JOP running. There are two ways to simulate JOP:

- High-level JVM simulation with JopSim
- VHDL simulation (e.g. with ModelSim)

2.5.1 JopSim Simulation

The high level simulation with JopSim is a simple JVM written in Java that can execute the JOP specific application (the .jop file). It is started with:

```
make jsim
```

To output each executing bytecode during the simulation run change in the Makefile the logging parameter to -Dlog="true".

[7]Eclipse can't use path variables for external .jar files.

VHDL file	Function	Initialization file	Generator
sim_jop_types_100.vhd	JOP constant definitions	-	-
sim_rom.vhd	JVM microcode ROM	mem_rom.dat	Jopa
sim_ram.vhd	Stack RAM	mem_ram.dat	Jopa
sim_jbc.vhd	Bytecode memory (cache)	-	-
sim_memory.vhd	Main memory	mem_main.dat	jop2dat
sim_pll.vhd	A dummy entity for the PLL	-	-
sim_uart.vhd	Print characters to stdio	-	-

Table 2.3: Simulation specific VHDL files

2.5.2 VHDL Simulation

This section is about running a VHDL simulation with ModelSim. All simulation files are vendor independent and should run on any versions of ModelSim or a different VHDL simulator. You can simulate JOP even with the free ModelSim XE II Starter Xilinx version, the ModelSim Altera version or the ModelSim Actel version.

To simulate JOP, or any other processor design, in a vendor neutral way, models of the internal memories (block RAM) and the external main memory are necessary. Beside this, only a simple clock driver is necessary. To speed-up the simulation a little bit, a simulation of the UART output, which is used for System.out.print(), is also part of the package.

Table 2.3 lists the simulation files for JOP and the programs that generates the initialization data. The non-generated VHDL files can be found in directory vhdl/simulation. The needed VHDL files and the compile order can be found in sim.bat under modelsim.

The actual version of JOP contains all necessary files to run a simulation with ModelSim. In directory vhdl/simulation you will find:

- A test bench: tb_jop.vhd with a serial receiver to print out the messages from JOP during the simulation
- Simulation versions of all memory components (vendor neutral)
- Simulation of the main memory

Jopa generates various mem_xxx.dat files that are read by the simulation. The JVM that is generated with jopsim.bat assumes that the Java application is preloaded in the main memory. jop2dat generates a memory initialization file from the Java application file (MainClass.jop) that is read by the simulation of the main memory (sim_memory.vhd).

In directory modelsim you will find a small batch file (sim.bat) that compiles JOP and the test bench in the correct order and starts ModelSim. The whole simulation process (including generation of the correct microcode) is started with:

```
make sim
```

After a few seconds you should see the startup message from JOP printed in ModelSim's command window. The simulation can be continued with run -all and after around 6 ms *simulation time* the actual Java main() method is executed. During those 6 ms, which will probably be minutes of simulation, the memory is initialized for the garbage collector.

2.6 Files Types You Might Encounter

As there are various tools involved in the complete build process, you will find files with various extensions. The following list explains the file types you might encounter when changing and building JOP.

The following files are the *source* files:

.vhd VHDL files describe the hardware part and are compiled with either Quartus or Xilinx ISE. Simulation in ModelSim is also based on VHDL files.

.v Verilog HDL. Another hardware description language. Used more in the US.

.java Java — the language that runs native on JOP.

.c There are still some tools written in C.

.asm JOP microcode. The JVM is written in this stack oriented assembler. Files are assembled with Jopa. The result are VHDL files, .mif files, and .dat files for ModelSim.

.bat Usage of these DOS batch files still prohibit running the JOP build under Unix. However, these files get less used as the Makefile progresses.

.xml Project files for Ant. Ant is an attractive substitution to make. Future distributions on JOP will be ant based.

Quartus II and Xilinx ISE need configuration files that describe your project. All files are usually ASCII text files.

.qpf Quartus II Project File. Contains almost no information.

.qsf Quartus II Settings File defines the project. VHDL files that make up your project are listed. Constraints such as pin assignments and timing constraints are set here.

.cdf Chain Description File. This file stores device name, device order, and programming file name information for the programmer.

.tcl Tool Command Language. Can be used in Quartus to automate parts of the design flow (e.g. pin assignment).

.npl Xilinx ISE project. VHDL files that make up your project are listed. The actual version of Xilinx ISE converts this project file to a new format that is not in ASCII anymore.

.ucf Xilinx Foundation User Constraint File. Constraints such as pin assignments and timing constraints are set here.

The Java tools javac and jar produce following file types from the Java sources:

.class A class file contains the bytecodes, a symbol table and other ancillary information and is executed by the JVM.

.jar The Java Archive file format enables you to bundle multiple files into a single archive file. Typically a .jar file contains the class files and auxiliary resources. A .jar file is essentially a zip file that contains an optional META-INF directory.

The following files are generated by the various tools from the source files:

.jop This file makes up the linked Java application that runns on JOP. It is generated by JOPizer and can be either downloaded (serial line or USB) or stored in the Flash (or used by the simulation with JopSim or ModelSim)

.mif Memory Initialization File. Defines the initial content of on-chip block memories for Altera devices.

.dat memory initialization files for the simulation with ModelSim.

.sof SRAM Output File. Configuration file for Altera devices. Used by the Quartus programmer or by quartus_pgm. Can be converted to various (or too many) different format. Some are listed below.

.pof Programmer Object File. Configuration for Altera devices. Used for the Flash loader PLDs.

.jbc JamTM STAPL Byte Code 2.0. Configuration for Altera devices. Input file for jbi32.

.ttf Tabular Text File. Configuration for Altera devices. Used by flash programming utilities (amd and udp.Flash to store the FPGA configuration in the boards Flash.

.rbf Raw Binary File. Configuration for Altera devices. Used by the USB download utility (USBRunner) to configure the dspio board via the USB connection.

.bit Bitstream File. Configuration file for Xilinx devices.

2.7 Information on the Web

Further information on JOP and the build process can be found on the Internet at the following places:

- `http://www.jopdesign.com/` is the main web site for JOP
- `http://www.jopwiki.com/` is a Wiki that can be freely edited by JOP users.
- `http://tech.groups.yahoo.com/group/java-processor/` hosts a mailing list for discussions on Java processors in general and mostly on JOP related topics

2.8 Porting JOP

Porting JOP to a different FPGA platform or board usually consists of adapting pin definitions and selection of the correct memory interface. Memory interfaces for the SimpCon interconnect can be found in directory vhdl/memory.

2.8.1 Test Utilities

To verify that the port of JOP is successful there are some small test programs in asm/src. To run the JVM on JOP the microcode jvm.asm is assembled and will be stored in an on-chip ROM. The Java application will then be loaded by the first microcode instructions in jvm.asm into an external memory. However, to verify that JOP and the serial line are working correctly, it is possible to run small test programs directly in microcode.

One test program (blink.asm) does not need the main memory and is a first test step before testing the possibly changed memory interface. testmon.asm can be used to debug the main memory interface. Both test programs can be built with the make targets jop_blink_test and jop_testmon.

Blinking LED and UART output

The test is built with:

```
make jop_blink_test
```

After download, the watchdog LED should blink and the FPGA will print out 0 and 1 on the serial line. Use a terminal program or the utility e.exe to check the output from the serial line.

Test Monitor

Start a terminal program (e.g. HyperTerm) to communicate with the monitor program and build the test monitor with:

```
make jop_testmon
```

After download the program prints the content of the memory at address 0. The program understands following *commands*:

- A single CR reads the memory at the current addres and prints out the address and memory content
- addr=val; writes *val* into the memory location at address *addr*

One tip: Take care that your terminal program does not send an LF after the CR.

2.9 Extending JOP

JOP is a soft-core processor and customizing it for an application is an interesting opportunity.

2.9.1 Native Methods

The *native* language of JOP is microcode. A native method is implemented in JOP microcode. The interface to this native method is through a *special* bytecode. The mapping between native methods and the special bytecode is performed by JOPizer. When adding a new (*special*) bytecode to JOP, the following files have to be changed:

1. jvm.asm implementation
2. Native.java method signature
3. JopInstr.java mapping of the signature to the name
4. JopSim.java simulation of the bytecode
5. JVM.java (just rename the method name)
6. Startup.java (only when needed in a class initializer)
7. WCETInstruction.java timing information

First implement the native code in JopSim.java for easy debugging. The *real* microcode is added in jvm.asm with a label for the special byctecode. The naming convention is jopsys_name. In Native.java provide a method signature for the native method and enter the mapping between this signature and the name in jvm.asm and in JopInstr.java. Provide the execution time in WCETInstruction.java for WCET analysis.

The native method is accessed by the method provided in Native.java. There is no calling overhead involved in the mechanism. The *native* method gets substituted by JOPizer with a *special* bytecode.

2.9.2 A new Peripheral Device

Creation of a new peripheral devices involves some VHDL coding. However, there are several examples in jop/vhdl/scio available.

All peripheral components in JOP are connected with the SimpCon [127] interface. For a device that implements the Wishbone [94] bus, a SimpCon-Wishbone bridge (sc2wb.vhd) is available (e.g., it is used to connect the AC97 interface in the dspio project).

For an easy start use an existing example and change it to your needs. Take a look into sc_test_slave.vhd. All peripheral components (SimpCon slaves) are connected in one module usually named scio_xxx.vhd. Browse the examples and copy one that best fits your needs.

In this module the address of your peripheral device is defined (e.g. 0x10 for the primary UART). This I/O address is mapped to a negative memory address for JOP. That means 0xffffff80 is added as a base to the I/O address.

By convention this address mapping is defined in com.jopdesign.sys.Const. Here is the UART example:

```
// use negative base address for fast constant load
// with bipush
public static final int IO_BASE = 0xffffff80;
 ...
public static final int IO_STATUS = IO_BASE+0x10;
public static final int IO_UART = IO_BASE+0x10+1;
```

The I/O devices are accessed from Java by *native*[8] functions: Native.rdMem() and Native.wrMem() in pacakge com.jopdesign.sys. Again an example with the UART:

```
// busy wait on free tx buffer
// no wait on an open serial line, just wait
// on the baud rate
while ((Native.rdMem(Const.IO_STATUS)&1)==0) {
    ;
}
Native.wrMem(c, Const.IO_UART);
```

Best practise is to create a new I/O configuration scio_xxx.vhdl and a new Quartus project for this configuration. This avoids the mixup of the changes with a new version of JOP. For the new Quartus project only the three files jop.cdf, jop.qpf, and jop.qsf have to be copied in a new directory under quartus. This new directory is the project name that has to be set in the Makefile:

```
QPROJ=yourproject
```

The new VHDL module and the scio_xxx.vhdl are added in jop.qsf. This file is a plain ASCII file and can be edited with a standard editor or within Quartus.

2.9.3 A Customized Instruction

A customized instruction can be simply added by implementing it in microcode and mapping it to a native function as described before. If you want to include a hardware module

[8]These are not real functions and are substituted by special bytecodes on application building with JOPizer.

that implements this instruction a new microinstruction has to be introduced. Besides mapping this instruction to a native method the instruction has also be added to the microcode assembler Jopa.

2.9.4 Dependencies and Configurations

As JOP and the JVM are a mix of VHDL and Java files, of changes some configurations or changes in central data structures needs an update in several files.

Speed Configuration

By default, JOP is configured for 80 Mhz. To build the 100 MHz configuration, edit quartus/cycmin/jop.qsf and change jop_config_80 to jop_config_100.

Method Cache Configuration

The default configuration (for the Altera Cyclone) is a 4 KB method cache configured with 16 blocks (i.e., a variable block cache). For the Xilinx targets, the cache size is 2KB because Xilinx does not (or did not) support easily configurable block RAMs.

To change from a variable block cache to a dual-block cache, you will need to edit the top-level VHDL. Here is an example from vhdl/top/jopcyc.vhd:

```
entity jop is

generic (
   ram_cnt  : integer := 2;   -- clock cycles for external ram
-- rom_cnt : integer := 3;    -- clock cycles for external rom OK for 20 MHz
   rom_cnt  : integer := 15; -- clock cycles for external rom for 100 MHz
   jpc_width  : integer := 12; -- address bits of java bytecode pc = cache size
   block_bits  : integer := 4   -- 2*block_bits is number of cache blocks
);
```

The power of 2 of the jpc_width is the cache size, and the power of 2 of the block_bits is the number of blocks. To simulate a dual block cache, block_bits has to be set to 1. (To use a single block cache, cache.vhd has to be modified to force a miss at the cache lookup.)

Stack Size

The on-chip stack size can be configured by changing following constants:

- ram_width in jop_config_xx.vhd
- STACK_SIZE in com.jopdesign.sys.Const
- RAM_LEN in com.jopdesign.sys.Jopa

Changing the Class Format

The constants for the offsets of fields in the class format are found in:

- JOPizer: CLS_HEAD, dump()
- GC.java uses CLASS_HEADR
- JMV.java uses CLASS_HEADR + offset (checkcast, instanceof)

2.10 Directory Structure

The top-level directories of the distribution are:

asm Microcode source files. The microcode part of the JVM and test files.

boards Pictures and text for the Eclipse plugin

c_src Some utilities in C (e.g. down.exe and e.exe).

doc LaTeXsources for this handbook and short notes.

eclipse Eclipse project files

ext External VHDL and Verilog sources

java All Java files

- **lib** External .jar files
- **pc** Tools on the PC
- **pcsim** High-level simulation on the PC
- **target** The Java sources for JOP
- **tools** All Java tools

jbc FPGA configuration files for jbi32.exe (generated)

jopc A C version of a JOP JVM simulation – *very* outdated

linux Scripts to start a network and SLIP

modelsim ModelSim simulation

pins Pin definitions for FPGA boards

quartus Quartus project files

rbf FPGA configuration files for USBRunner (generated)

sopc JOP as SoPC component and SRAM components

support Stand-alone Flash programming for the Cycore board

ttf FPGA configuration files for Flash programming (generated)

vhdl The processor sources

- **altera** Altera specific components (PLL, RAM)
- **config** Cycore PLD sources
- **core** The processor core
- **fpu** The floating-point unit
- **memory** Main memory connections via SimpCon
- **scio** I/O components and configurations with SimpCon
- **simpcon** SimpCon bridges and arbiter
- **simulation** Memory and UART for ModelSim simulation
- **start** The VHDL version of *hello world* – a blinking LED
- **testbenches** no real content
- **top** Top-level and configuration (e.g. PLL setting) components
- **vga** A SimpCon VGA controller
- **xilinx** Xilinx specific components (RAM)

xilinx Xilinx project files

2.10.1 The Java Sources for JOP

The most important directory for all Java sources that run on JOP is in java/target.

dist Generated files

- **bin** The linked application (.jop)
- **classes** The class files
- **lib** The application class files in classes.zip – input for JOPizer

src The source

- **app** The applications
- **bench** The embedded benchmark suit
- **common** Utility classes
- **jdk_base** Base classes for the JDK
- **jdk11** JDK around version 1.1
- **jdk14** A test port of JDK 1.4 classes
- **rtapi** Experimental RT API dfinitions
- **test** Various test programs

wcet Output from the WCET analyzer (generated)

2.11 The JOP Hello World Exercise

This exercise gives an introduction into the design flow of JOP. JOP will be built from the sources and a simple *Hello World* program will run on it.

To understand the build process you have to run the build manually. This understanding will help you to find the correct files for changes in JOP and to adjust the Makefile for your needs.

2.11.1 Manual build

Manual build does not mean entering all commands, but calling the correct make target with the required arguments (if any) in the correct order. The idea of this exercise is to obtain knowledge of the directory structure and the dependencies of various design units.

Inspect the Makefile targets and the ones that are called from it before running them.

1. Create your working directory
2. Download the sources from the opencores CVS server
3. Connect the FPGA board to the PC (and the power supply)
4. Perform the build as described in Section 2.1.2.

As a result you should see a message at your command prompt.

2.11.2 Using make

In the root directory (jop) there is a Makefile. Open it with an editor and try to find the corresponding lines of code for the steps you did in the first exercise. Reset the FPGA by cycling the power and run the build with a simple

```
make
```

The whole process should run without errors and the result should be identical to the previous exercise.

2.11.3 Change the Java Program

The whole build process is not necessary when changing the Java application. Once the processor is built, a Java application can be built and downloaded with the following make target:

```
make japp
```

Change HelloWorld.java and run it on JOP. Now change the class name that contains the main() method from HelloWorld to Hello and rerun the Java application build. Now an embedded version of "Hello World" should run on JOP. Besides the usual greeting on the standard output, the LED on the FPGA board should blink at a frequency of 1 Hz. The first periodic task, an essential abstraction for real-time systems, is running on JOP!

2.11.4 Change the Microcode

The JVM is written in microcode and several .vhdl files are generated during assembly. For a test change only the version string[9] in jvm.asm to the actual date and run a full make.

[9]The actual version date will probably be different from the actual sources.

	simpexp	dspio
FPGA	EP1C6	EP1C12
I/O	UART	UART, USB, audio codec, sigma-delta codec
FPGA configuration	ByteBlaster	USBRunner
Java download	serial line	USB

Table 2.4: Differences between the two target boards

```
version = 20090626

version = 20110101
```

The start message should reflect your change. As the microcode was changed a full make run is necessary. The microcode assembly generates VHDL files and the complete processor has to be rebuilt.

2.11.5 Use a Different Target Board

In this exercise, you will alter the Makefile for a different target board. Disconnect the first board and connect the board with an USB port (e.g. the dspio or Lego board).

Table 2.4 lists the differences between the first board (simpexp) and the new one (called dspio). The correct FPGA is already selected in the Quartus project files (jop.qpf). Alter the Makefile to set the variable USB to true. This will change:

1. The Quartus project from cycmin to usbmin
2. The Java download is now over USB instead of the serial line
3. The parameters for the download via down.exe are changed to use the virtual com-port of the USB driver (look into Windows hardware manager to get the correct number) and the switch -usb for the download is added

Now build the whole project with make. Change the Java program and perform only the necessary build step.

2.11.6 Compile a Different Java Application

The class that contains the main method is described by three arguments:

1. The first directory relative to java/target/src (e.g. app or test)
2. The package name (e.g. dsp)
3. The main class (e.g. HalloWorld)

These three values are used by the Makeile and are set in the variables P1, P2, and P3 in the Makefile.

Change the Makefile to compile the embedded Java benchmark jbe.DoAll. The parameters for the Java application can also be given to the make with following command line arguments:

```
make −e P1=bench P2=jbe P3=DoAll
```

The three variables P1, P2, and P3 are a shortcut to set the main class of the application. You can also directly set the variables TARGET_APP_PATH and MAIN_CLASS.

2.11.7 Simulation

This exercise will give you a full view of the possibilities to debug JOP system code or the processor itself. There are two ways to simulate JOP: A simple debugging JVM written in Java (JopSim as part of the tool package) that can execute *jopized* applications and a VHDL level simulation with ModelSim. The make targets are jsim and sim.

2.11.8 WCET Analysis

An important step in real-time system development is the analysis of the WCET of the individual tasks. Compile and run the WCET example Loop.java in package wcet. You can analyze the WCET of the method measure() with following make command:

```
make java_app wcet −e P1=test P2=wcet P3=Loop
```

Change the code in Loop.java to enable measurement of the execution time and compare it with the output of the static analysis. In this simple example the WCET can be measured. However, be aware that most non-trivial code needs static analysis for safe estimates of WCET values.

3 Java and the Java Virtual Machine

Java technology consists of the Java language definition, a definition of the standard library, and the definition of an intermediate instruction set with an accompanying execution environment. This combination helps to make *write once, run anywhere* possible.

The following chapter gives a short overview of the Java programming language. A more detailed description of the Java Virtual Machine (JVM) and the explanation of the JVM instruction set, the so-called bytecodes, follows. The exploration of dynamic instruction counts of typical Java programs can be found in Section **??**.

3.1 Java

Java is a relatively new and popular programming language. The main features that have helped Java achieve success are listed below:

Simple and object oriented: Java is a simple programming language that appears very similar to C. This 'look and feel' of C means that programmers who know C, can switch to Java without difficulty. Java provides a simplified object model with single inheritance[1].

Portability: To accommodate the diversity of operating environments, the Java compiler generates bytecodes – an architecture neutral intermediate format. To guarantee platform independence, Java specifies the sizes of its basic data types and the behavior of its arithmetic operators. A Java interpreter, the Java virtual machine, is available on various platforms to help make 'write once, run anywhere' possible.

Availability: Java is not only available for different operating systems, it is available at no cost. The runtime system and the compiler can be downloaded from Sun's website for Windows, Linux, and Solaris. Sophisticated development environments, such as Netbeans or Eclipse, are available under the GNU Public License.

[1]Java has *single inheritance* of *implementation* – only one class can be extended. However, a class can implement several interfaces, which means that Java has *multiple interface inheritance*.

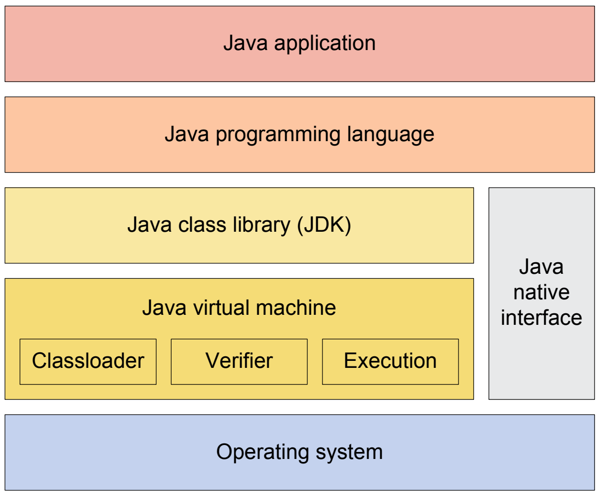


Figure 3.1: Java system overview

Library: The complete Java system includes a rich class library to increase programming productivity. Besides the functionality of a C standard library, it also contains other tools, such as collection classes and a GUI toolkit.

Built-in multithreading: Java supports multithreading at the language level: the library provides the Thread class, the language provides the keyword synchronized for critical sections and the runtime system provides monitor and condition lock primitives. The system libraries have been written to be thread-safe: the functionality provided by the libraries is available without conflicts due to multiple concurrent threads of execution.

Safety: Java provides extensive compile-time checking, followed by a second level of runtime checking. The memory management model is simple – objects are created with the new operator. There are no explicit pointer data types and no pointer arithmetic, but there is automatic garbage collection. This simple memory management model eliminates a large number of the programming errors found in C and C++ programs. A restricted runtime environment, the so-called *sandbox*, is available when executing small Java applications in Web browsers.

As can be seen in Figure 3.1, Java consists of three main components:

1. The Java programming language as defined in [47]

2. The class library, defined as part of the Java specification. All implementations of Java have to contain the library as defined by Sun
3. The Java virtual machine (defined in [82]) that loads, verifies and executes the binary representation (the *class file*) of a Java program

The Java native interface supports functions written in C or C++. This combination is sometimes called *Java technology* to emphasize the fact that Java is more than just another object-oriented language.

However, a number of issues have slowed down the broad acceptance of Java. The original presentation of Java as an Internet language led to the misconception that Java was not a general-purpose programming language. Another obstacle was the first implementation of the JVM as an interpreter. Execution of Java programs was *very* slow compared to compiled C/C++ programs. Although advances in its runtime technology, in particular the just-in-time compiler, have closed the performance gap, it is still a commonly held view that Java is slow.

3.1.1 History

The Java programming language originated as part of the Green project specifically for an embedded device, a handheld wireless PDA. In the early ’90s, Java, which was originally known as Oak [143, 144], was created as the programming tool for this device. The device (known as *7) was a small SPARC-based hardware device with a tiny embedded OS. However, the *7 was never released as a product and Java was officially released in 1995 as the *new* language for the Internet. Over the years, Java technology has become a programming tool for desktop applications, web servers and server applications. These application domains resulted in the split of the Java platform into the Java standard edition (J2SE) and the enterprise edition (J2EE) in 1999. With every new release, the library (defined as part of the language) continued to grow. Java for embedded systems was clearly not an area Sun was interested in pursuing. However, with the arrival of mobile phones, Sun again became interested in this embedded market. Sun defined different subsets of Java, which have now been combined into the Java Micro Edition (J2ME).

In 1999, a document defining the requirements for real-time Java was published by the NIST [88]. Based on these requirements, two groups defined specifications for real-time Java: the Real-Time Core Extension [141] published under the J Consortium and the Real-Time Specification for Java (RTSJ) [25]. A comparison of these two specifications and a comparison with Ada 95’s Real-Time Annex can be found in [30]. The RTSJ was the first Java Specification Request (JSR 1) under the Java Community Process (JCP) and started

Type	Description
boolean	either true or false
char	16-bit Unicode character (unsigned)
byte	8-bit integer (signed)
short	16-bit integer (signed)
int	32-bit integer (signed)
long	64-bit integer (signed)
float	32-bit floating-point (IEEE 754-1985)
double	64-bit floating-point (IEEE 754-1985)

Table 3.1: Java primitive data types

1999. The first release came out 2002 and further enhancement of the RTSJ (to version 1.1) are covered by the JSR 282 (started in 2005). Under JSR 302 (Safety Critical Java Technology) a subset of the RTSJ is currently defined for the safety critical domain (e.g., standard DO-178B/ED-12B [115]). A detailed description of the J2ME and specifications for real-time Java can be found in Chapter 4 of [123].

3.1.2 The Java Programming Language

The Java programming language is a general-purpose object-oriented language. Java is related to C and C++, but with a number of aspects omitted. Java is a strongly typed language, which means that type errors can be detected at compile time. Other errors, such as wrong indices in an array, are checked at runtime. The problematic[2] *pointer* in C and explicit deallocation of memory is completely avoided. The pointer is replaced by a *reference*, i.e., an abstract pointer to an object. Storage for an object is allocated from the heap during creation of the object with new. Memory is freed by automatic storage management, typically using a garbage collector. The garbage collector avoids memory leaks from a missing free() and the safety problems exposed by dangling pointers.

The types in Java are divided into two categories: primitive types and reference types. Table 3.1 lists the available primitive types. Method local variables, class fields and object fields contain either a primitive type value or a reference to an object.

Classes and class instances, the objects, are the fundamental data and code organization

[2]C pointers represent memory addresses as data. Pointer arithmetic and direct access to memory leads to common and hard-to-find program errors.

structures in Java. There are no global variables or functions as there are in C/C++. Each method belongs to a class. This 'everything belongs to a class or an object' combined with the class naming convention, as suggested by Sun, avoids name conflicts in even the largest applications.

New classes can extend exactly one superclass. Classes that do not explicitly extend a superclass become direct subclasses of Object, the root of the whole class tree. This single inheritance model is extended by *interfaces*. Interfaces are abstract classes that only define method signatures and provide no implementation. A concrete class can implement several interfaces. This model provides a simplified form of multiple inheritance.

Java supports multitasking through *threads*. Each thread is a separate flow of control, executing concurrently with all other threads. A thread contains the method stack as thread local data – all objects are shared between threads. Access conflicts to shared data are avoided by the proper use of synchronized methods or code blocks.

Java programs are compiled to a machine-independent bytecode representation as defined in [82]. Although this intermediate representation is defined for Java, other programming languages (e.g., ADA [33]) can also be compiled into Java bytecodes.

3.2 The Java Virtual Machine

The Java virtual machine (JVM) is a definition of an abstract computing machine that executes bytecode programs. The JVM specification [82] defines three elements:

- An instruction set and the meaning of those instructions – the *bytecodes*
- A binary format – the *class file* format. A class file contains the bytecodes, a symbol table and other ancillary information
- An algorithm to *verify* that a class file contains valid programs

In the solution presented in this book, the class files are verified, linked and transformed into an internal representation before being executed on JOP. This transformation is performed with JOPizer and is not executed on JOP. We will therefore omit the description of the class file and the verification process.

The instruction set of the JVM is stack-based. All operations take their arguments from the stack and put the result onto the stack. Values are transferred between the stack and various memory areas. We will discuss these memory areas first, followed by an explanation of the instruction set.

3.2.1 Memory Areas

The JVM contains various runtime data areas. Some of these areas are shared between threads, whereas other data areas exist separately for each thread.

Method area: The method area is shared among all threads. It contains static class information such as field and method data, the code for the methods and the constant pool. The constant pool is a per-class table, containing various kinds of constants such as numeric values or method and field references. The constant pool is similar to a symbol table.

Part of this area, the code for the methods, is very frequently accessed (during instruction fetch) and therefore is a good candidate for caching.

Heap: The heap is the data area where all objects and arrays are allocated. The heap is shared among all threads. A garbage collector reclaims storage for objects.

JVM stack: Each thread has a private stack area that is created at the same time as the thread. The JVM stack is a logical stack that contains following elements:

1. A frame that contains return information for a method
2. A local variable area to hold local values inside a method
3. The operand stack, where all operations are performed

Although it is not strictly necessary to allocate all three elements to the same type of memory we will see in Section 4.4 that the argument-passing mechanism regulates the layout of the JVM stack.

Local variables and the operand stack are accessed as frequently as registers in a standard processor. A Java processor should provide some caching mechanism of this data area.

The memory areas are similar to the various segments in conventional processes (e.g. the method code is analogous to the ‘text’ segment). However, the operand stack replaces the registers in a conventional processor.

3.2.2 JVM Instruction Set

The instruction set of the JVM contains 201 different instructions [82]. This *bytecodes* can be grouped into the following categories:

Load and store: Load instructions push values from the local variables onto the operand stack. Store instructions transfer values from the stack back to local variables. 70 different instructions belong to this category. Short versions (single byte) exist to access the first four local variables. There are unique instructions for each basic type (int, long, float, double and reference). This differentiation is necessary for the bytecode verifier, but is not needed during execution. For example iload, fload and aload all transfer one 32-bit word from a local variable to the operand stack.

Arithmetic: The arithmetic instructions operate on the values found on the stack and push the result back onto the operand stack. There are arithmetic instructions for int, float and double. There is no direct support for byte, short or char types. These values are handled by int operations and have to be converted back before being stored in a local variable or an object field.

Type conversion: The type conversion instructions perform numerical conversions between all Java types: as implicit widening conversions (e.g., int to long, float or double) or explicit (by casting to a type) narrowing conversions.

Object creation and manipulation: Class instances and arrays (that are also objects) are created and manipulated with different instructions. Objects and class fields are accessed with type-less instructions.

Operand stack manipulation: All direct stack manipulation instructions are type-less and operate on 32-bit or 64-bit entities on the stack. Examples of these instructions are dup, to duplicate the top operand stack value, and pop, to remove the top operand stack value.

Control transfer: Conditional and unconditional branches cause the JVM to continue execution with an instruction other than the one immediately following. Branch target addresses are specified relative to the current address with a signed 16-bit offset. The JVM provides a complete set of branch conditions for int values and references. Floating-point values and type long are supported through compare instructions. These compare instructions result in an int value on the operand stack.

Method invocation and return: The different types of methods are supported by four instructions: invoke a class method, invoke an instance method, invoke a method that implements an interface and an invokespecial for an instance method that requires special handling, such as private methods or a superclass method.

A bytecode consists of one instruction byte followed by optional operand bytes. The length of the operand is one or two bytes, with the following exceptions: multianewarray contains 3 operand bytes; invokeinterface contains 4 operand bytes, where one is redundant and one is always zero; lookupswitch and tableswitch (used to implement the Java switch statement) are variable-length instructions; and goto_w and jsr_w are followed by a 4 byte branch offset, but neither is used in practice as other factors limit the method size to 65535 bytes.

3.2.3 Methods

A Java *method* is equivalent to a *function* or *procedure* in other languages. In object oriented terminology this *method* is *invoked* instead of *called*. We will use *method* and *invoke* in the remainder of this text. In Java and the JVM, there are five types of methods:

- Static or class methods
- Virtual methods
- Interface methods
- Class initialization
- Constructor of the parent class (super())

For these five types there are only four different bytecodes:

invokestatic: A class method (declared static) is invoked. As the target does not depend on an object, the method reference can be resolved at load/link time.

invokevirtual: An object reference is resolved and the corresponding method is invoked. The resolution is usually done with a dispatch table per class containing all implemented and inherited methods. With this dispatch table, the resolution can be performed in constant time.

invokeinterface: An interface allows Java to emulate multiple inheritance. A class can implement several interfaces, and different classes (that have no inheritance relation) can implement the same interface. This flexibility results in a more complex resolution process. One method of resolution is a search through the class hierarchy that results in a variable, and possibly lengthy, execution time. A constant time resolution is possible by assigning every interface method a unique number. Each class that implements an interface needs its own table with unique positions for each interface method of the *whole* application.

```
for (;;) {
    instr = bcode[pc++];
    switch ( instr ) {
        ...
        case IADD:
            tos = stack[sp]+stack[sp−1];
            −−sp;
            stack[sp] = tos;
            break;
        ...
    }
}
```

Listing 3.1: A typical JVM interpreter loop

invokespecial: Invokes an instance method with special handling for superclass, private, and instance initialization. This bytecode catches many different cases. This results in expensive checks for common private instance methods.

3.2.4 Implementation of the JVM

There are several different ways to implement a virtual machine. The following list presents these possibilities and analyses how appropriate they are for embedded devices.

Interpreter: The simplest realization of the JVM is a program that interprets the bytecode instructions. The interpreter itself is usually written in C and is therefore easy to port to a new computer system. The interpreter is very compact, making this solution a primary choice for resource-constrained systems. The main disadvantage is the high execution overhead. From a code fragment of the typical interpreter loop, as shown in Listing 3.1, we can examine the overhead: The emulation of the stack in a high-level language results in three memory accesses for a simple iadd bytecode. The instruction is decoded through an indirect jump. Indirect jumps are still a burden for standard branch prediction logic.

Just-In-Time Compilation: Interpreting JVMs can be enhanced with just-in-time (JIT) compilers. A JIT compiler translates Java bytecodes to native instructions during runtime. The time spent on compilation is part of the application execution time. JIT compilers are therefore restricted in their optimization capacity. To reduce the compilation overhead, current JVMs operate in mixed mode: Java methods are executed in interpreter mode and the call frequency is monitored. Often-called methods, the hot spots, are then compiled to native code.

JIT compilation has several disadvantages for embedded systems, notably that a compiler (with the intrinsic memory overhead) is necessary on the target system. Due to compilation during runtime, execution times are hardly predictable.[3]

Batch Compilation: Java can be compiled, in advance, to the native instruction set of the target. Precompiled libraries are linked with the application during runtime. This is quite similar to C/C++ applications with shared libraries. This solution undermines the flexibility of Java: dynamic class loading during runtime. However, this is not a major concern for embedded systems.

Hardware Implementation: A Java processor is the implementation of the JVM in hardware. The JVM bytecode is the native instruction set of such a processor. This solution can result in quite a small processor, as a stack architecture can be implemented very efficiently. A Java processor is memory-efficient as an interpreting JVM, but avoids the execution overhead. The main disadvantage of a Java processor is the lack of capability to execute C/C++ programs. This book describes JOP as an example of a JVM hardware implementation.

3.3 Embedded Java

In embedded systems the architecture of JVMs are more diverse than on desktop or server systems. Figure 3.2 shows variations of Java implementations in embedded systems and an example of the control flow for a web server application. The standard approach of a JVM running on top of an operating system (OS) is shown in sub-figure (a). A network connection bypasses the JVM via native functions and uses the TCP/IP stack implementation and the device drivers of the OS.

A JVM without an OS is shown in sub-figure (b). This solution is often called *running on the bare metal*. The JVM acts as the OS and provides the thread scheduling and the low-level access to the hardware. In that case the network stack can be written entirely in Java. JNode[4] is an approach to implement the OS entirely in Java. This solution becomes popular even in server applications.[5]

[3]Even if the time for the compilation is known, the WCET for a method has to include the compile time! Furthermore, WCET analysis has to know in advance what code will be produced by JIT compilation.

[4]`http://www.jnode.org/`

[5]BEA System offers the JVM LiquidVM that includes basic OS functions and does not need a guest OS.

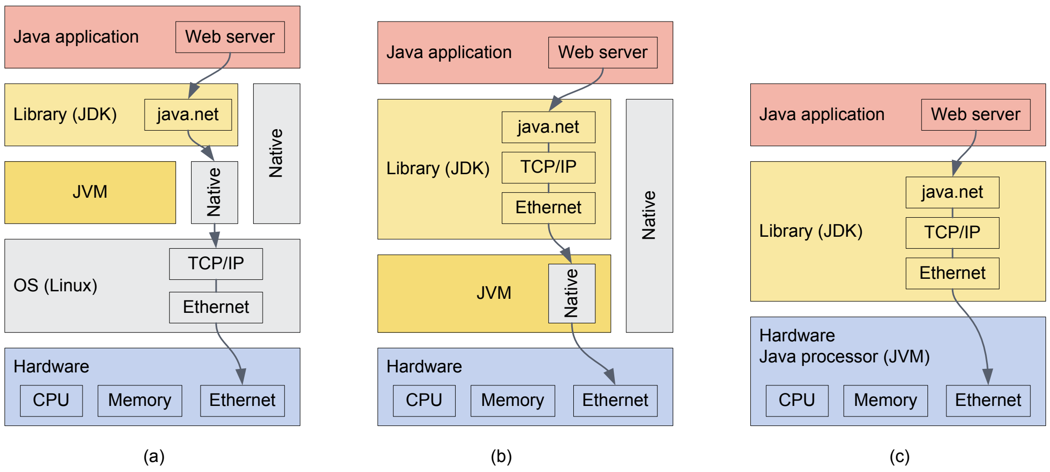


Figure 3.2: Implementation variations for an embedded JVM: (a) standard layers for Java with an operating system – equivalent to desktop configurations, (b) a JVM on the bare metal, and (c) a JVM as a Java processor.

Sub-figure (c) shows an embedded solution where the JVM is part of the hardware layer. That means it is implemented in a Java processor. With this solution the native layer can be completely avoided and all code (application and system code) is written entirely in Java.

Figure 3.2 shows how the flow from the application goes down to the hardware. The example consists of a web server and an Internet connection via Ethernet. In case (a) the application web server talks with java.net in the JDK. The flow goes through a native interface to the TCP/IP implementation and the Ethernet device driver within the OS (usually written in C). The device driver talks with the Ethernet chip. In (b) the OS layer is omitted: the TCP/IP layer and the Ethernet device driver are now part of the Java library. In (c) the JVM is part of the hardware layer and a direct access from the Ethernet driver to the Ethernet hardware is mandatory. Note how part of the network stack moves up from the OS layer to the Java library. Version (c) shows a pure Java implementation of the whole network stack.

3.4 Summary

Java is a unique combination of the language definition, a rich class library and a runtime environment. A Java program is compiled to bytecodes that are executed by a Java virtual machine. Strong typing, runtime checks and avoidance of pointers make Java a *safe* language. The intermediate bytecode representation simplifies porting of Java to different computer systems. An interpreting JVM is easy to implement and needs few system resources. However, the execution speed suffers from interpreting. JVMs with a just-in-time compiler are state-of-the-art for desktop and server systems. These compilers require large amounts of memory and have to be ported for each processor architecture, which means they are not the best choice for embedded systems. A Java processor is the implementation of the JVM as a concrete machine. A Java processor avoids the slow execution model of an interpreting JVM and the memory requirements of a compiler, thus making it an interesting execution system for Java in embedded systems.

4 Hardware Architecture

This chapter presents the architecture of JOP and the motivation behind the various different design decisions we faced. The first sections give an overview of JOP, describe the microcode and the pipeline.

Pipelined instruction processing calls for high memory bandwidth. Caches are needed in order to avoid bottlenecks resulting from the main memory bandwidth. As seen in Chapter 3, there are two memory areas that are frequently accessed by the JVM: the stack and the method area. In this chapter, time-predictable cache solutions for both areas that are implemented in JOP are presented.

4.1 Overview of JOP

This section gives an overview of JOP architecture. Figure 4.1 shows JOP's major function units. A typical configuration of JOP contains the processor core, a memory interface and a number of I/O devices. The module extension provides the link between the processor core, and the memory and I/O modules.

The processor core contains the three microcode pipeline stages *microcode fetch*, *decode* and *execute* and an additional translation stage *bytecode fetch*. The ports to the other modules are the two top elements of the stack (TOS and NOS), input to the top-of-stack (Data), bytecode cache address and data, and a number of control signals. There is no direct connection between the processor core and the external world.

The memory controller implements the simple memory load and store operations and the field and array access bytecodes. It also contains the method cache. The memory interface provides a connection between the main memory and the memory controller. The extension module controls data read and write. The *busy* signal is used by the microcode instruction wait[1] to synchronize the processor core with the memory unit. The core reads bytecode instructions through dedicated buses (BC address and BC data) from the memory

[1]The busy signal can also be used to stall the whole processor pipeline. This was the change made to JOP by Flavius Gruian [48]. However, in this synchronization mode, the concurrency between the memory access module and the main pipeline is lost.

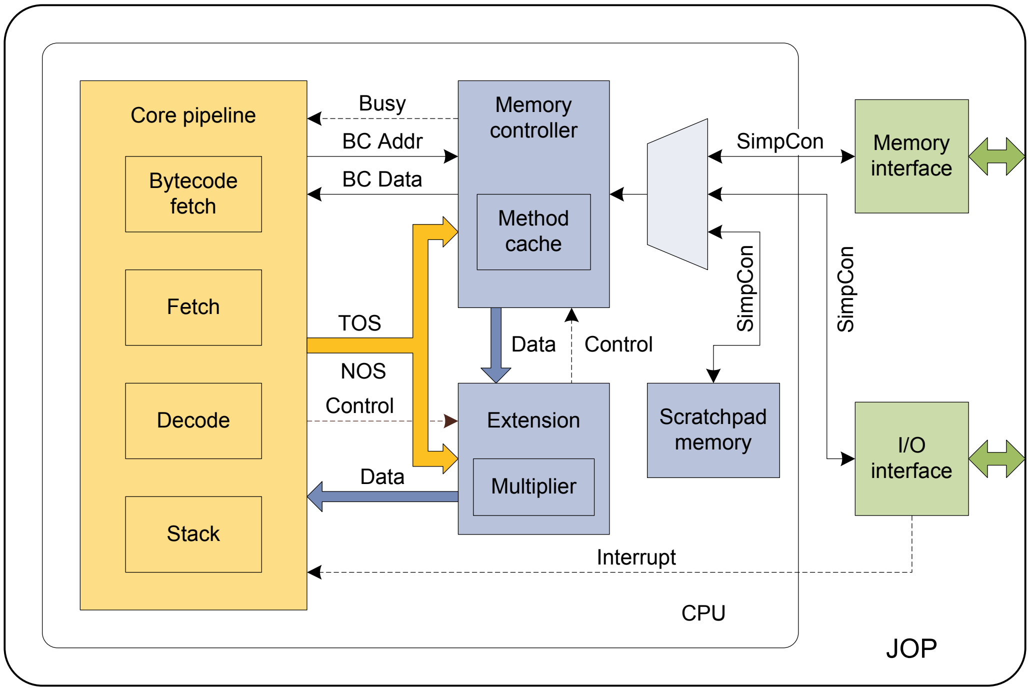


Figure 4.1: Block diagram of JOP

controller. The request for a method to be placed in the cache is performed through the extension module, but the cache hit detection and load is performed by the memory controller independently of the processor core (and therefore concurrently).

The I/O interface contains peripheral devices, such as the system time and timer interrupt for real-time thread scheduling, a serial interface and application-specific devices. Read and write to and from this module are controlled by the memory controller. All external devices[2] are connected to the I/O interface.

The extension module performs two functions: (a) it contains hardware accelerators (such as the multiplier unit in this example) and (b) the multiplexer for the read data that is loaded into the top-of-stack register. The write data from the top-of-stack (TOS) and next-of-stack (NOS) are connected directly to all modules.

The division of the processor into those modules greatly simplifies the adaptation of JOP

[2]The external device can be as simple as a line driver for the serial interface that forms part of the interface module, or a complete bus interface, such as the ISA bus used to connect e.g. an Ethernet chip.

for different application domains or hardware platforms. Porting JOP to a new FPGA board usually results in changes in the memory interface alone. Using the same board for different applications only involves making changes to the I/O interface. JOP has been ported to several different FPGAs and prototyping boards and has been used in different real-world applications (see Chapter 11), but it never proved necessary to change the processor core.

4.2 Microcode

The following discussion concerns two different instruction sets: *bytecode* and *microcode*. Bytecodes are the instructions that make up a compiled Java program. These instructions are executed by a Java virtual machine. The JVM does not assume any particular implementation technology. Microcode is the native instruction set for JOP. Bytecodes are translated, during their execution, into JOP microcode. Both instruction sets are designed for an extended[3] stack machine.

4.2.1 Translation of Bytecodes to Microcode

To date, no hardware implementation of the JVM exists that is capable of executing *all* bytecodes in hardware alone. This is due to the following: some bytecodes, such as new, which creates and initializes a new object, are too complex to implement in hardware. These bytecodes have to be emulated by software.

To build a self contained JVM without an underlying operating system, direct access to the memory and I/O devices is necessary. There are no bytecodes defined for low-level access. These low-level services are usually implemented in *native* functions, which means that another language (C) is native to the processor. However, for a Java processor, bytecode is the *native* language.

One way to solve this problem is to implement simple bytecodes in hardware and to emulate the more complex and *native* functions in software with a different instruction set (sometimes called microcode). However, a processor with two different instruction sets results in a complex design.

Another common solution, used in Sun's picoJava [145], is to execute a subset of the bytecode native and to use a software trap to execute the remainder. This solution entails an overhead (a minimum of 16 cycles in picoJava, see 12.2.1) for the software trap.

[3]An extended stack machine is one in which there are instructions available to access elements deeper down in the stack.

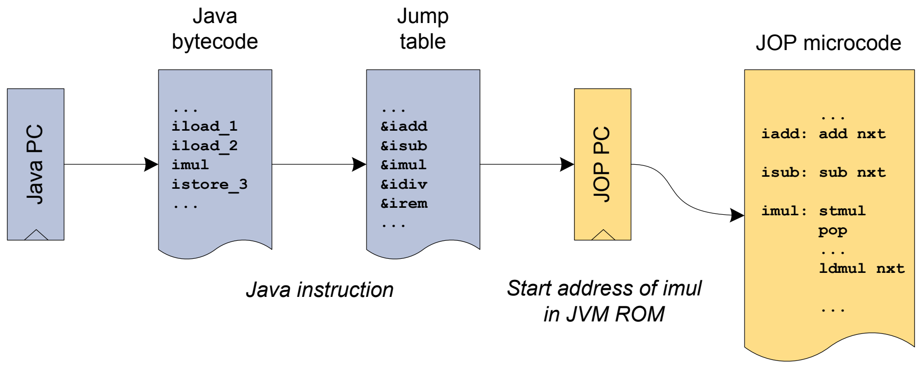


Figure 4.2: Data flow from the Java program counter to JOP microcode

In JOP, this problem is solved in a much simpler way. JOP has a single *native* instruction set, the so-called microcode. During execution, every Java bytecode is translated to either one, or a sequence of microcode instructions. This translation merely adds one pipeline stage to the core processor and results in no execution overheads (except for a bytecode branch that takes 4 instead of 3 cycles to execute). The area overhead of the translation stage is 290 LCs, or about 15% of the LCs of a typical JOP configuration. With this solution, we are free to define the JOP instruction set to map smoothly to the stack architecture of the JVM, and to find an instruction coding that can be implemented with minimal hardware.

Figure 4.2 gives an example of the data flow from the Java program counter to JOP microcode. The figure represents the two pipeline stages bytecode fetch/translate and microcode fetch. The fetched bytecode acts as an index for the jump table. The jump table contains the start addresses for the bytecode implementation in microcode. This address is loaded into the JOP program counter for every bytecode executed. JOP executes the sequence of microcode until the last one. The last one is marked with *nxt* in microcode assembler. This *nxt* bit in the microcode ROM triggers a new translation i.e., a new address is loaded into the JOP program counter. In Figure 4.2 the implementation of bytecode idiv is an example of a longer sequence that ends with microcode instruction ldm c nxt.

The difference to other forms of instruction translation in hardware is that this solution is time predictable. The translation takes one cycle (one pipeline stage) for each bytecode, independent from the execution history. Instruction folding, e.g., implemented in picoJava [90, 145], is also a form of instruction translation in hardware. Folding is used to translate several (stack oriented) bytecode instructions to a RISC type instruction. This translation needs an instruction buffer and the fill level of this instruction buffer depends on the execution history. The length of this history that has to be considered for analysis is not bounded. Therefore this form of instruction translation is not exactly time predictable.

4.2.2 Compact Microcode

For the JVM to be implemented efficiently, the microcode has to *fit* to the Java bytecode. Since the JVM is a stack machine, the microcode is also stack-oriented. However, the JVM is not a pure stack machine. Method parameters and local variables are defined as *locals*. These locals can reside in a stack frame of the method and are accessed with an offset relative to the start of this *locals* area.

Additional local variables (16) are available at the microcode level. These variables serve as scratch variables, like registers in a conventional CPU. Furthermore, the constant pool pointer (cp), the method pointer (mp), and pointers to the method tables of JVM.java and JVMHelp.java are stored in these variables. The 16 variables are located in the on-chip stack memory. However, arithmetic and logic operations are performed on the stack.

Some bytecodes, such as ALU operations and the short form access to *locals*, are directly implemented by an equivalent microcode instruction (with a different encoding). Additional instructions are available to access internal registers, main memory and I/O devices. A relative conditional branch (zero/non zero of TOS) performs control flow decisions at the microcode level. For optimum use of the available memory resources, all instructions are 8 bits long. There are no variable-length instructions and every instruction, with the exception of wait, is executed in a single cycle. To keep the instruction set this dense, the following concept is applied: immediate values and branch offsets are addressed through one indirection. The instruction just contains an index for the constants.

Two types of operands, immediate values and branch distances, normally force an instruction set to be longer than 8 bits. The instruction set is either expanded to 16 or 32 bits, as in typical RISC processors, or allowed to be of variable length at byte boundaries. A first implementation of the JVM with a 16-bit instruction set showed that only a small number of different constants are necessary for immediate values and relative branch distances.

In the current realization of JOP, the different immediate values are collected while the microcode is being assembled and are put into the initialization file for the on-chip memory. These constants are accessed indirectly in the same way as the local variables. They are similar to initialized variables, apart from the fact that there are no operations to change their value during runtime, which would serve no purpose and would waste instruction codes. The microcode local variables, the microcode constants and the stack share the same on-chip memory. Using a single memory block simplifies the multiplexer in the execution stage.

A similar solution is used for branch distances. The assembler generates a VHDL file with a table for all found branch constants. This table is indexed using instruction bits during runtime. These indirections during runtime make it possible to retain an 8-bit instruction set, and provide 16 different immediate values and 32 different branch constants. For a general purpose instruction set, these indirections would impose too many restrictions. As the microcode only implements the JVM, this solution is a viable option.

To simplify the logic for instruction decoding, the instruction coding is carefully chosen. For example, one bit in the instruction specifies whether the instruction will increment or decrement the stack pointer. The offset to access the *locals* is directly encoded in the instruction. This is not the case for the original encoding of the equivalent bytecodes (e.g. *iload_0* is 0x1a and *iload_1* is 0x1b). Whenever a multiplexer depends on an instruction, the selection is directly encoded in the instruction.

4.2.3 Instruction Set

JOP implements 54 different microcode instructions. These instructions are encoded in 8 bits. With the addition of the *nxt* and *opd* bits in every instruction, the effective instruction length is 10 bits.

Bytecode equivalent: These instructions are direct implementations of bytecodes and result in one cycle execution time for the bytecode (except st and ld): pop, and, or, xor, add, sub, st<n>, st, ushr, shl, shr, nop, ld<n>, ld, dup

Local memory access: The first 16 words in the internal stack memory are reserved for internal variables. The next 16 words contain constants. These memory locations are accessed using the following instructions: stm, stmi, ldm, ldmi, ldi

Register manipulation: The stack pointer, the variable pointer and the Java program counter are loaded or stored with: stvp, stjpc, stsp, ldvp, ldjpc, ldsp, star

Bytecode operand: The operand is loaded from the bytecode RAM, converted to a 32-bit word and pushed on the stack with: ld_opd_8s, ld_opd_8u, ld_opd_16s, ld_opd_16u

External memory access: The autonomous memory subsystem and the I/O subsystem are accessed by using the following instructions: stmra, stmwa, stmwd, wait, ldmrd, stbcrd, ldbcstart, stald, stast, stgf, stpf, stcp

Multiplier: The multiplier is accessed with: stmul, ldmul

Microcode branches: Two conditional branches in microcode are available: bz, bnz

```
iadd:   add nxt      // 1 to 1 mapping

//  a and b are scratch variables for the
//  JVM code.
swap:   stm a        // save TOS in variable a
        stm b        // save TOS−1 in variable b
        ldm a        // push a on stack
        ldm b nxt    // push b on stack and fetch next bytecode
```

Listing 4.1: Implementation of iadd and swap

Bytecode branch: All 17 bytecode branch instructions are mapped to one instruction: jbr

A detailed description of the microcode instructions can be found in Appendix C.

4.2.4 Bytecode Example

The example in Figure 4.1 shows the implementation of a single cycle bytecode and an infrequent bytecode as a sequence of JOP instructions. The suffix nxt marks the last instruction of the microcode sequence. In this example, the iadd bytecode is mapped to the equivalent add microcode and executed in a single cycle, whereas swap takes four cycles to execute, and after the last instruction (ldm b nxt), the first instruction for the next bytecode is executed. The scratch variables, as shown in the second example, are stored in the on-chip memory that is shared with the stack cache.

Some bytecodes are followed by operands of between one and three bytes in length (except lookupswitch and tableswitch). Due to pipelining, the first operand byte that follows the bytecode instruction is available when the first microcode instruction enters the execution stage. If this is a one-byte long operand, it is ready to be accessed. The increment of the Java program counter after the read of an operand byte is coded in the JOP instruction (an *opd* bit similar to the *nxt* bit).

In Listing 4.2, the implementation of sipush is shown. The bytecode is followed by a two-byte operand. Since the access to bytecode memory is only one[4] byte per cycle, *opd* and *nxt* are not allowed at the same time. This implies a minimum execution time of $n+1$ cycles for a bytecode with n operand bytes.

[4]The decision is to avoid buffers that would introduce time dependencies over bytecode boundaries.

```
sipush: nop opd          // fetch next byte
        nop opd          // and one more
        ld_opd_16s nxt   // load 16 bit operand
```

Listing 4.2: Bytecode operand load

```
add          // sets the condition for the branch
nop          // one cycle condition delay slot
bz label     // a branch on TOS zero
instr1       // is executed
instr2       // is executed
instr3       // executed on fall through
```

Listing 4.3: Microcode condition delay and branch delay slots

4.2.5 Microcode Branches

At the microcode level two conditional branches that test the TOS are available: bz branch on zero, and bnz branch on not zero. The branches are followed by two delay slots, i.e., the following two instructions are executed independent of the branch condition outcome. Furthermore, the branch condition is also pipelined, i.e., it has to be available one cycle earlier. Listing 4.3 shows the condition delay and the branch delay slots.

4.2.6 Flexible Implementation of Bytecodes

As mentioned above, some Java bytecodes are very complex. One solution already described is to emulate them through a sequence of microcode instructions. However, some of the more complex bytecodes are very seldom used. To further reduce the resource implications for JOP, in this case local memory, bytecodes can even be implemented by *using* Java bytecodes. That means bytecodes (e.g., new or floating point operations) can be implemented in Java. This feature also allows for the easy configuration of resource usage versus performance.

During the assembly of the JVM, all labels that represent an entry point for the bytecode implementation are used to generate the translation table. For all bytecodes for which no such label is found, i.e. there is no implementation in microcode, a *not-implemented* address is generated. The instruction sequence at this address invokes a static method from a system class. This class contains 256 static methods, one for each possible bytecode, ordered by the bytecode value. The bytecode is used as the index in the method table of this system class. A single empty static method consumes three 32-bit words in memory. Therefore, the overhead of this special class is 3 KB, which is 9% of a minimal *hello world* program (34 KB memory footprint).

4.2.7 Summary

In order to handle the great variation in the complexity of Java bytecodes, the bytecodes are translated to a different instruction set, the so-called microcode. This microcode is still an instruction set for a stack machine, but more RISC-like than the CISC-like JVM bytecodes.

At the time of this writing 43 of the 201 different bytecodes are implemented by a single microcode instruction, 92 by a microcode sequence, and 41 bytecodes are implemented in Java. Furthermore, JOP contains additional bytecodes that are used to implement low-level operations, such as direct memory access. Those bytecodes are mapped to native, static methods in com.jopdesign.sys.Native. In the next section we will see how this translation is handled in JOP's pipeline and how it can simplify interrupt handling.

4.3 The Processor Pipeline

JOP is a fully pipelined architecture with single cycle execution of microcode instructions and a novel approach of translation from Java bytecode to these instructions. Figure 4.3 shows the datapath for JOP, representing the pipeline from left to right. Blocks arranged vertically belong to the same pipeline stage.

Three stages form the JOP core pipeline, executing microcode instructions. An additional stage in the front of the core pipeline fetches Java bytecodes – the instructions of the JVM – and translates these bytecodes into addresses in microcode. Bytecode branches are also decoded and executed in this stage. The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches. Besides the usual decode function, the third pipeline stage also generates addresses for the stack RAM (the stack cache). As every stack machine microcode instruction (except nop, wait, and jbr) has either *pop* or *push* characteristics, it is possible to generate fill or spill addresses for the *following* instruction at this stage. The last pipeline stage performs ALU operations, load, store and stack spill or fill. At the execution stage, operations are performed with the two topmost elements of the stack.

The stack architecture allows for a short pipeline, which results in short branch delays. Two branch delay slots are available after a conditional microcode branch. A stack machine

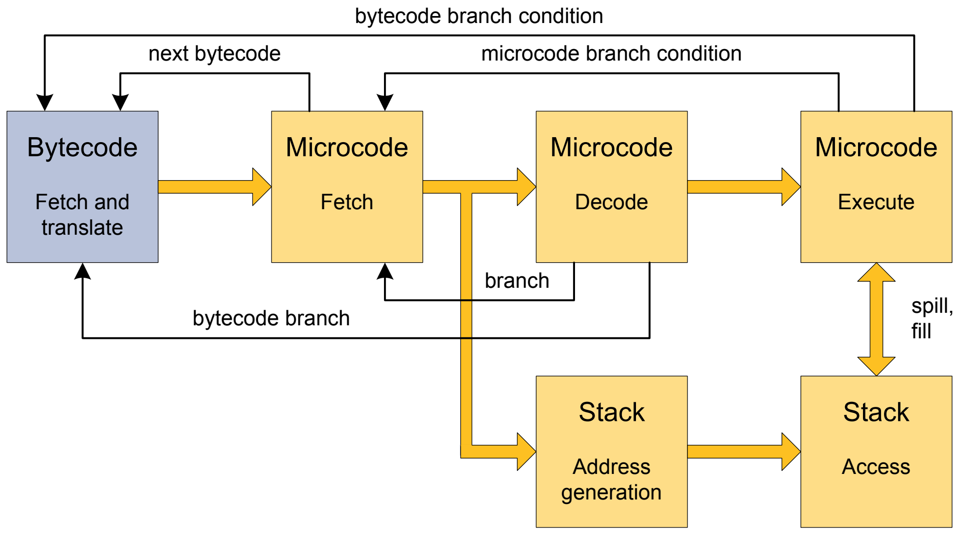


Figure 4.3: Datapath of JOP

with two explicit registers for the two topmost stack elements and automatic fill/spill to the stack cache needs neither an extra write-back stage nor any data forwarding. See Section 4.4 for a detailed description.

The method cache (*Bytecode Cache*), microcode ROM, and stack RAM are implemented with single cycle access in the FPGA's internal memories.

4.3.1 Java Bytecode Fetch

In the first pipeline stage, as shown in Figure 4.4, the Java bytecodes are fetched from the internal memory (*Method cache*). The bytecode is mapped through the translation table into the address (*jpaddr*) for the microcode ROM. Interrupts and exceptions are handled by redirection of the microcode address to the handler code.

The fetched bytecode results in an absolute jump in the microcode (the second stage). If the bytecode is mapped one-to-one with a JOP instruction, the following fetched bytecode again results in a jump in the microcode in the following cycle. If the bytecode is a complex one, JOP continues to execute microcode. At the end of this instruction sequence, the next bytecode, and therefore the new jump address, is requested (signal *nxt*).

The method cache serves as the instruction cache and is filled on method invoke and return. Details about this time-predictable instruction cache can be found in Section 4.5.

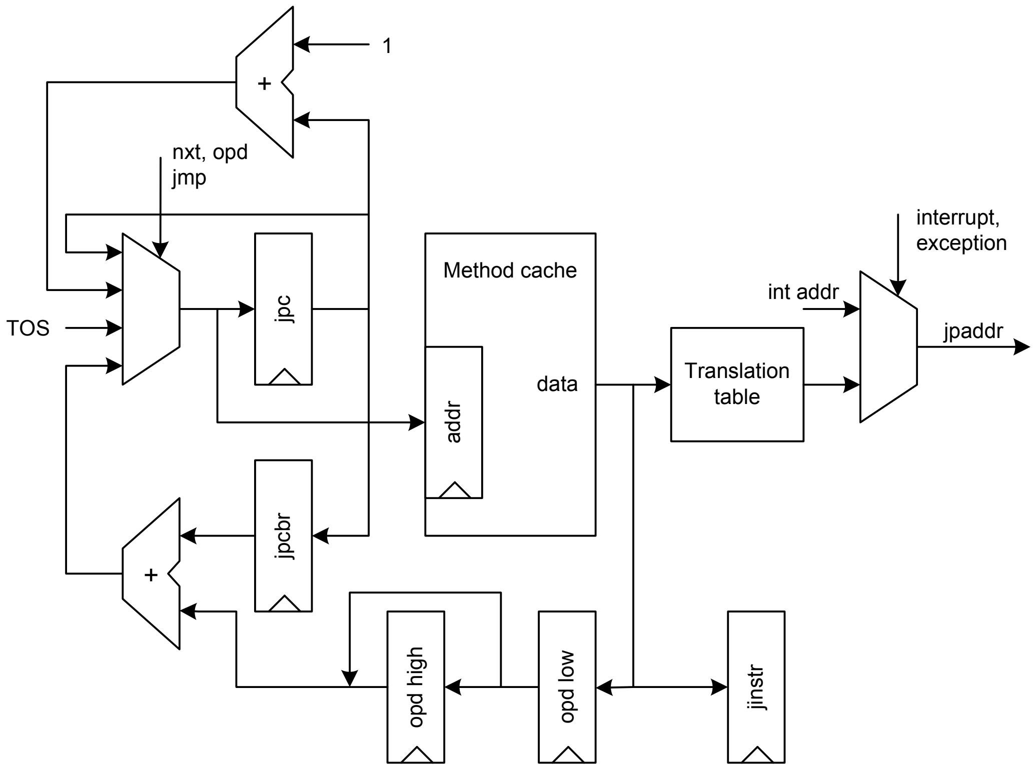


Figure 4.4: Java bytecode fetch and translation

The bytecode is also stored in a register for later use as an operand (requested by signal *opd*). Bytecode branches are also decoded and executed in this stage. Since *jpc* is also used to read the operands, the program counter is saved in *jpcbr* during an instruction fetch. *jinstr* is used to decode the branch type and *jpcbr* to calculate the branch target address.

4.3.2 Microcode Instruction Fetch

The second pipeline stage, as shown in Figure 4.5, fetches microcode instructions from the internal microcode memory and executes microcode branches.

The JOP microcode, which implements the JVM, is stored in the microcode ROM. The program counter *pc* is incremented during normal execution. If the instruction is labeled with *nxt* a new bytecode is requested from the first stage and *pc* is loaded with *jpaddr*.

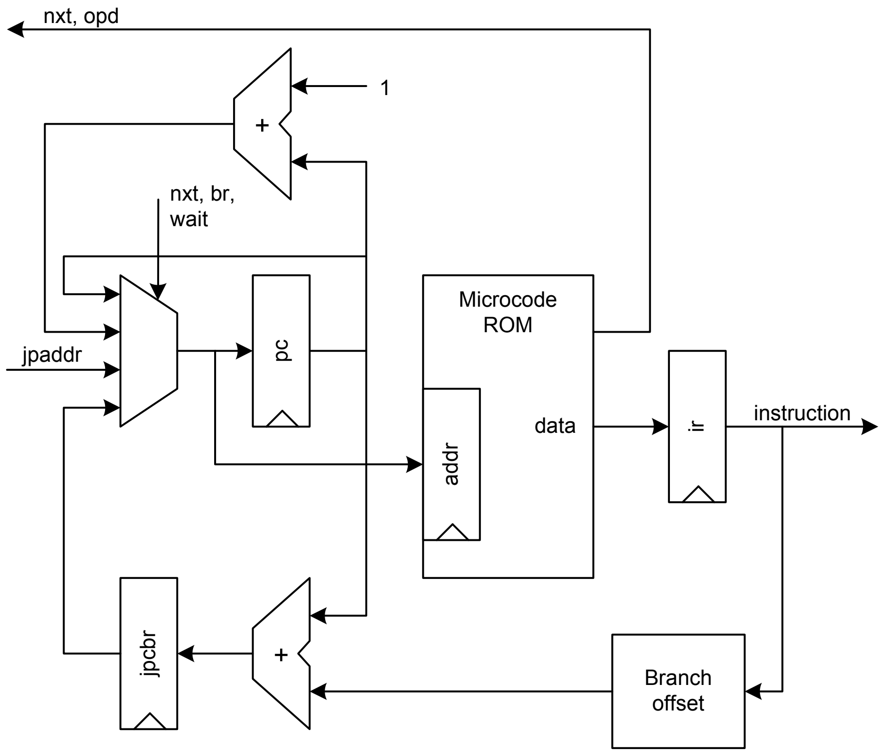


Figure 4.5: Microcode instruction fetch

jpaddr is the starting address for the implementation of that bytecode. The label *nxt* is the flag that marks the end of the microcode instruction stream for one bytecode. Another flag, *opd*, indicates that a bytecode operand needs to be fetched in the first pipeline stage. Both flags are stored in the microcode ROM.

The register *brdly* contains the target address for a conditional branch. The same offset is shared by a number of branch destinations. A table (*branch offset*) is used to store these relative offsets. This indirection means that only 5 bits need to be used in the instruction coding for branch targets and thereby allow greater offsets. The three tables *translation table* (from the bytecode fetch stage), *microcode ROM*, and *branch offset* are generated during the assembly of the JVM code. The outputs are plain VHDL files. For an implementation in an FPGA, recompiling the design after changing the JVM implementation is a straightforward operation. For an ASIC with a loadable JVM, it is necessary to implement

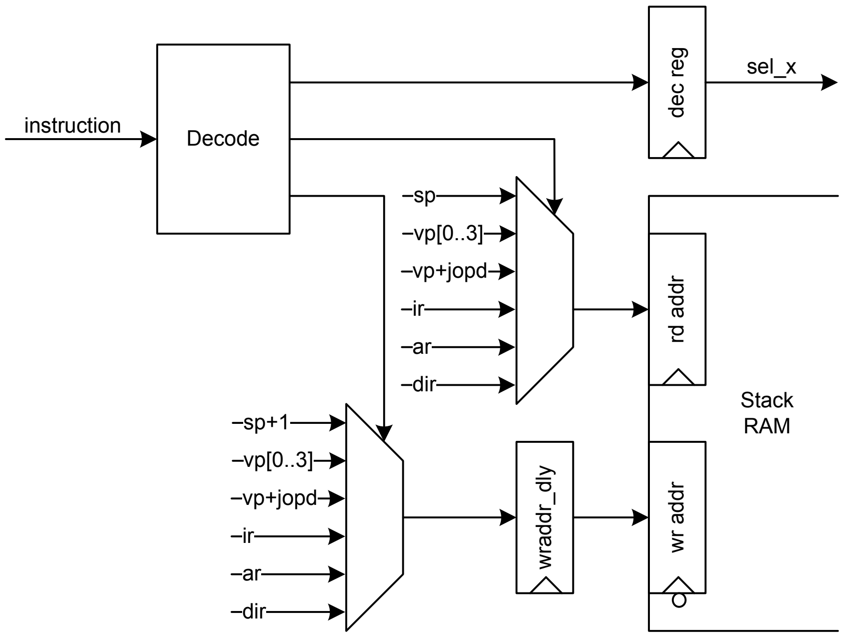


Figure 4.6: Decode and address generation

a different solution.

FPGAs available to date do not allow asynchronous memory access. They therefore force us to use the registers in the memory blocks. However, the output of these registers is not accessible. To avoid having to create an additional pipeline stage just for a register-register move, the read address register of the microcode ROM is fed by the *pc* multiplexer. The memory address register effectively contains the same value as the *pc*.

4.3.3 Decode and Address Generation

Besides the usual decode function, the third pipeline, as shown in Figure 4.6, also generates addresses for the stack RAM.

As we can see in Section 4.4 Table 4.2, read and write addresses are either relative to the stack pointer or to the variable pointer. The selection of the pre-calculated address can be performed in the decode stage. When an address relative to the stack pointer is used (either as read or as write address, never for both) the stack pointer is also decremented or incremented in the decode stage.

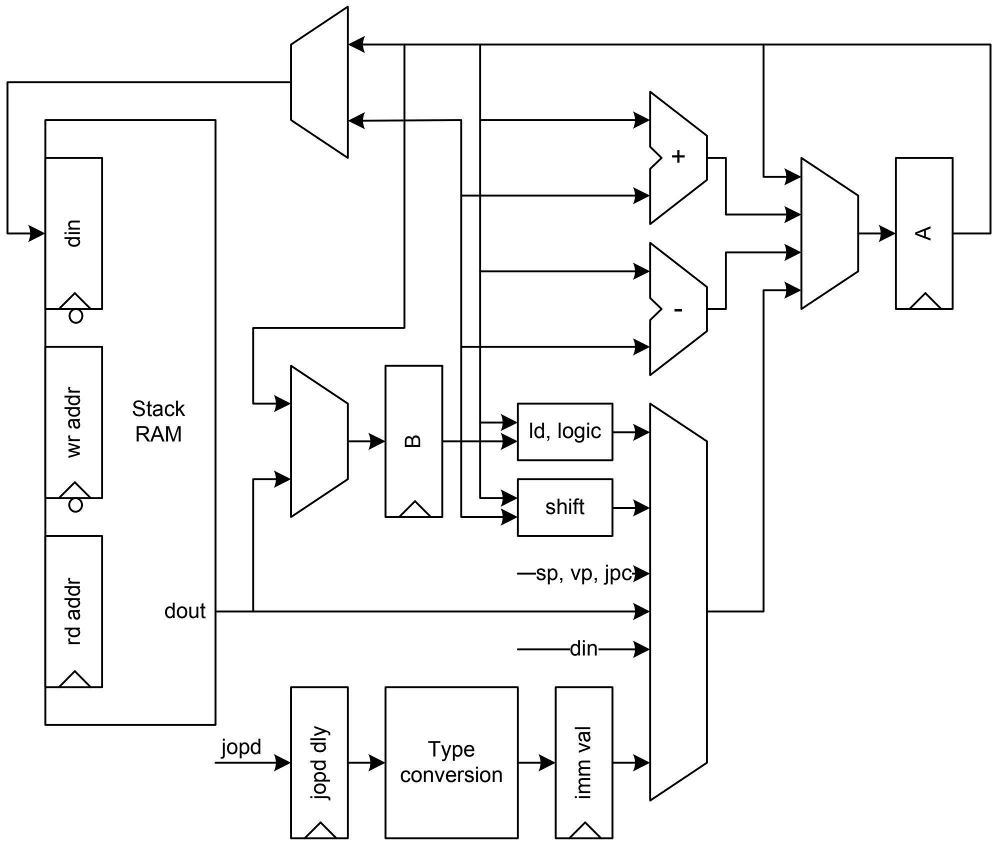


Figure 4.7: Execution stage

Stack machine instructions can be categorized from a stack manipulation perspective as either *pop* or *push*. This allows us to generate fill or spill TOS-1 addresses for the *following* instruction during the decode stage, thereby saving one extra pipeline stage.

4.3.4 Execute

At the execution stage, as shown in Figure 4.7, operations are performed using two discrete registers: TOS and TOS-1, labeled *A* and *B*.

Each arithmetic/logical operation is performed with registers *A* and *B* as the source (top-of-stack and next-of-stack), and register *A* as the destination. All load operations (local variables, internal register, external memory and periphery) result in a value being loaded into register *A*. There is no need for a write-back pipeline stage. Register *A* is also the source for the store operations. Register *B* is never accessed directly. It is read as an implicit operand or for stack spill on push instructions. It is written during the stack spill with the content of the stack RAM or the stack fill with the content of register *A*.

Beside the Java stack, the stack RAM also contains microcode variables and constants. This resource-sharing arrangement not only reduces the number of memory blocks needed for the processor, but also the number of data paths to and from the register *A*.

The inverted clock on data-in and on the write address register of the stack RAM is used to perform the RAM write in the same cycle as the execute operation.

A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill needs neither an extra write-back stage nor any data forwarding. Details of this two-level stack architecture are described in Section 4.4.

4.3.5 Interrupt Logic

Interrupts and (precise) exceptions are considered hard to implement in a pipelined processor [62], meaning implementation tends to be complex (and therefore resource consuming). In JOP, the bytecode-microcode translation is used cleverly to avoid having to handle interrupts and exceptions (e.g., stack overflow) in the core pipeline.

Interrupts are implemented as special bytecodes. These bytecodes are inserted by the hardware in the Java instruction stream. When an interrupt is pending and the next fetched byte from the bytecode cache is an instruction (as indicated by the *nxt* bit in the microcode), the associated special bytecode is used instead of the instruction from the bytecode cache. The result is that interrupts are accepted at bytecode boundaries. The worst-case preemption delay is the execution time of the *slowest* bytecode that is implemented in microcode. Bytecodes that are implemented in Java (see Section 4.2.6) can be interrupted.

The implementation of interrupts at the bytecode-microcode mapping stage keeps interrupts transparent in the core pipeline and avoids complex logic. Interrupt handlers can be implemented in the same way as standard bytecodes are implemented i.e. in microcode or Java.

This special bytecode can result in a call of a JVM internal method in the context of the interrupted thread. This mechanism implicitly stores almost the complete context of the current active thread on the stack. This feature is used to implement the preemptive, fixed priority real-time scheduler in Java [118].

The main source for an interrupt is the μs accurate timer interrupt used by the real-time scheduler. I/O device interrupts can also be connected to the interrupt controller. Hardware generated exceptions, such as stack overflow or array bounds checks, generate a system interrupt. The exception reason can be found in a register.

4.3.6 Summary

In this section, we have analyzed JOP's pipeline. The core of the stack machine constitutes a three-stage pipeline. In the following section, we will see that this organization is an optimal solution for the stack access pattern of the JVM.

An additional pipeline stage in front of this core pipeline stage performs bytecode fetch and the translation to microcode. This organization has zero overheads for more complex bytecodes and results in the short pipeline that is necessary for any processor without branch prediction. This additional translation stage also presents an elegant way of incorporating interrupts virtually *for free*.

4.4 The Stack Cache

The Java programming language defines not only the language but also a binary representation of the program and an abstract machine, the JVM, to execute this binary. The JVM is similar to the Forth abstract machine in that it is also a stack machine. However, the usage of the stack differs from Forth in such a way that a Forth processor is not an ideal hardware platform to execute Java programs.

In this section, the stack usage in the JVM is analyzed. We will see that, besides the access to the top elements of the stack, an additional access path to an arbitrary element of the stack is necessary for an efficient implementation of the JVM.

As the stack is a heavily accessed memory region, the stack – or part of it – has to be placed in the upper level of the memory hierarchy. This part of the stack is referred to as a *stack cache*. We will show that the JVM does not need a full three-port access to the stack. This allows for a simple and elegant design of the stack cache for a Java processor.

Other stack cache implementations use three port memories (e.g., in Komodo [76]) or discrete register files (e.g., picoJava [145] and the aJile JEMCore [55]). The comparison of the JOP stack organization with the other two approaches can be found in [121].

4.4.1 Java Computing Model

The JVM is not a pure stack machine in the sense of, for instance, the stack model in Forth. The JVM operates on a LIFO stack as its *operand stack*. The JVM supplies instructions to load values on the operand stack, and other instructions take their operands from the stack, operate on them and push the result back onto the stack. For example, the iadd instruction pops two values from the stack and pushes the result back onto the stack. These instructions are the stack machine's typical zero-address instructions. The maximum depth of this operand stack is known at compile time. In typical Java programs, the maximum depth is very small. To illustrate the operation notation of the JVM, Table 4.1 shows the evaluation of an expression for a stack machine notation and the JVM bytecodes. Instruction iload_n loads an integer value from a local variable at position *n* and pushes the value on TOS.

The JVM contains another memory area for method local data. This area is known as *local variables*. Primitive type values, such as integer and float, and references to objects are stored in these local variables. Arrays and objects cannot be allocated in a local variable, as in C/C++. They have to be placed on the heap. Different instructions transfer data between the operand stack and the local variables. Access to the first four elements is optimized with dedicated single byte instructions, while up to 256 local variables are accessed with a two-byte instruction and, with the wide modifier, the area can contain up to 65536 values.

$A = B + C * D$	
Stack	JVM
push B	iload_1
push C	iload_2
push D	iload_3
*	imul
+	iadd
pop A	istore_0

Table 4.1: Standard stack notation and the corresponding JVM instructions

These local variables are very similar to registers and it appears that some of these locals can be mapped to the registers of a general purpose CPU or implemented as registers in a Java processor. On method invocation, local variables could be saved in a frame on a stack, different from the operand stack, together with the return address, in much the same way as in C on a typical processor. This would result in the following memory hierarchy:

- On-chip hardware stack for ALU operations
- A small register file for frequently-accessed variables
- A method stack in main memory containing the return address and additional local variables

However, the semantics of method invocation suggest a different model. The arguments of a method are pushed on the operand stack. In the invoked method, these arguments are not on the operand stack but are instead accessed as the first variables in the local variable area. The *real* method local variables are placed at higher indices. Listing 4.4 gives an example of the argument passing mechanism in the JVM. These arguments could be copied to the local variable area of the invoked method. To avoid this memory transfer, the entire variable area (the arguments *and* the variables of the method) is allocated on the operand stack. However, in the invoked method, the arguments are buried deep in the stack.

This asymmetry in the argument handling prohibits passing down parameters through multiple levels of subroutine calls, as in Forth. Therefore, an extra stack for return addresses is of no use for the JVM. This single stack now contains the following items in a frame per method:

```
The Java source:

    int val = foo(1, 2);
    ...
    public int foo(int a, int b) {
        int c = 1;
        return a+b+c;
    }

Compiled bytecode instructions for the JVM:

The invocation sequence:
    aload_0                 // Push the object reference
    iconst_1                // and the parameter onto the
    iconst_2                // operand stack.
    invokevirtual   #2      // Invoke method foo:(II)I.
    istore_1                // Store the result in val.

public int foo(int, int):
    iconst_1                // The constant is stored in a method
    istore_3                // local variable (at position 3).
    iload_1                 // Arguments are accessed as locals
    iload_2                 // and pushed onto the operand stack.
    iadd                    // Operation on the operand stack.
    iload_3                 // Push c onto the operand stack.
    iadd
    ireturn                 // Return value is on top of stack.
```

Listing 4.4: Example of parameter passing and access

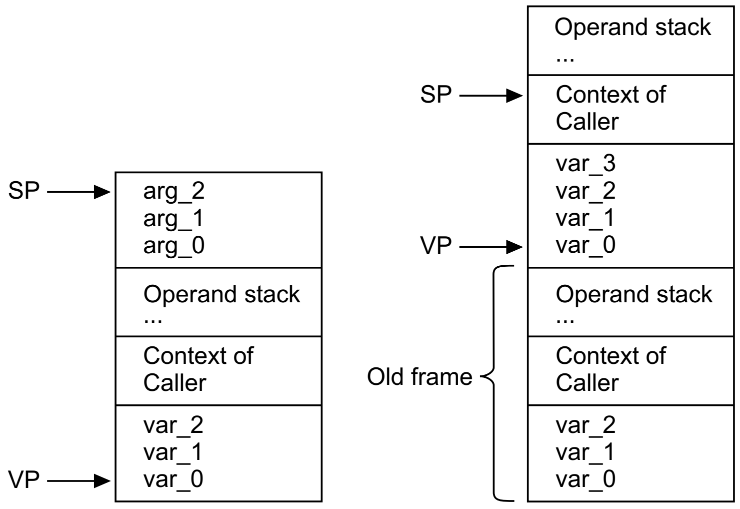


Figure 4.8: Stack change on method invocation

- The local variable area
- Saved context of the caller
- The operand stack

A possible implementation of this layout is shown in Figure 4.8. A method with two arguments, arg_1 and arg_2 (arg_0 is the *this* pointer), is invoked in this example. The invoked method *sees* the arguments as var_1 and var_2. var_3 is the only local variable of the method. SP is a pointer to the top of the stack and VP points to the start of the variable area.

4.4.2 Access Patterns on the Java Stack

The pipelined architecture of a Java processor executes basic instructions in a single cycle. A stack that contains the operand stack *and* the local variables results in the following access patterns:

Stack Operation: Read of the two top elements, operate on them and push back the result on the top of the stack. The pipeline stages for this operation are:

value1 $\leftarrow$ stack[sp], value2 $\leftarrow$ stack[sp-1]

```
result ← value1 op value2, sp ← sp-1
stack[sp] ← result
```

Variable Load: Read a data element deeper down in the stack, relative to a variable base address pointer (VP), and push this data on the top of the stack. This operation needs two pipeline stages:

```
value ← stack[vp+offset], sp ← sp+1
stack[sp] ← value
```

Variable Store: Pop the top element of the stack and write it in the variable relative to the variable base address:

```
value ← stack[sp]
stack[vp+offset] ← value, sp ← sp-1
```

For pipelined execution of these operations, a three-port memory or register file (two read ports and one write port) would be necessary.

In following section, we will discuss access patterns of the JVM and their implication on the functional units of the pipeline. A fast and small architecture for the stack cache of a Java processor is described.

4.4.3 JVM Stack Access Revised

If we analyze the JVM's access patterns to the stack in more detail, we can see that a two-port read is only performed with the two top elements of the stack. All other operations with elements deeper in the stack, local variables load and store, only need one read port. If we only implement the two top elements of the stack in registers, we can use a standard on-chip RAM with one read and one write port.

We will show that all operations can be performed with this configuration. Let A be the top-of-stack, B the element below top-of-stack. The memory that serves as the second level cache is represented by the array sm. Two indices in this array are used: p points to the logical third element of the stack and changes as the stack grows or shrinks, v points to the base of the local variables area in the stack and n is the address offset of a variable. op is a two operand stack operation with a single result (i.e. a typical ALU operation).

Case 1: ALU operation

$A \leftarrow A \; op \; B$
$B \leftarrow sm[p]$
$p \leftarrow p - 1$

The two operands are provided by the two top level registers. A single read access from *sm* is necessary to fill *B* with a new value.

Case 2: Variable load (*Push*)

$sm[p+1] \leftarrow B$
$B \leftarrow A$
$A \leftarrow sm[v+n]$
$p \leftarrow p + 1$

One read access from *sm* is necessary for the variable read. The former TOS value moves down to *B* and the data previously in *B* is written to *sm*.

Case 3: Variable store (*Pop*)

$sm[v+n] \leftarrow A$
$A \leftarrow B$
$B \leftarrow sm[p]$
$p \leftarrow p - 1$

The TOS value is written to *sm*. *A* is filled with *B* and *B* is filled in an identical manner to Case 1, needing a single read access from *sm*.

We can see that all three basic operations can be performed with a stack memory with one read and one write port. Assuming a memory is used that can handle concurrent read and write access, there is no structural access conflict between *A*, *B* and *sm*. That means that all operations can be performed concurrently in a single cycle.

As we can see in Figure 4.8 the operand stack and the local variables area are distinct regions of the stack. A concurrent read from and write to the stack is only performed on a variable load or store. When the read is from the local variables area the write goes to the operand area; a read from the operand area is concurrent with a write to the local variables area. Therefore there is no concurrent read and write to the same location in *sm*. There is no constraint on the read-during-write behavior of the memory (old data, undefined or new data), which simplifies the memory design. In a design where read and write-back are located in different pipeline stages, as in the architectures described above, either the memory must provide the new data on a read-during-write, or external forward logic is necessary.

From the three cases described, we can derive the memory addresses for the read and write port of the memory, as shown in Table 4.2.

Read address	Write address
p	p+1
v+n	v+n

Table 4.2: Stack memory addresses

4.4.4 A Two-Level Stack Cache

As a result of the previous analysis the stack cache of JOP is organized at two levels: first level in two discrete registers for the TOS and NOS; second level as on-chip memory with one read and one write port.

The Datapath

The architecture of the two-level stack cache can be seen in Figure 4.9. Register *A* represents the top-of-stack and register *B* the data below the top-of-stack. ALU operations are performed with these two registers and the result is placed in *A*. During such an ALU operation, *B* is filled with new data from the stack RAM. A new value from the local variable area is loaded directly from the stack RAM into *A*. The data previously in *A* is moved to *B* and the data from *B* is spilled to the stack RAM. *A* is stored in the stack RAM on a store instruction to the local variable. The data from *B* is moved to *A* and *B* is filled with a new value from the stack RAM. With this architecture, the stack machine pipeline can be reduced to three stages:

1. IF – instruction fetch
2. ID – instruction decode
3. EX – execute, load or store

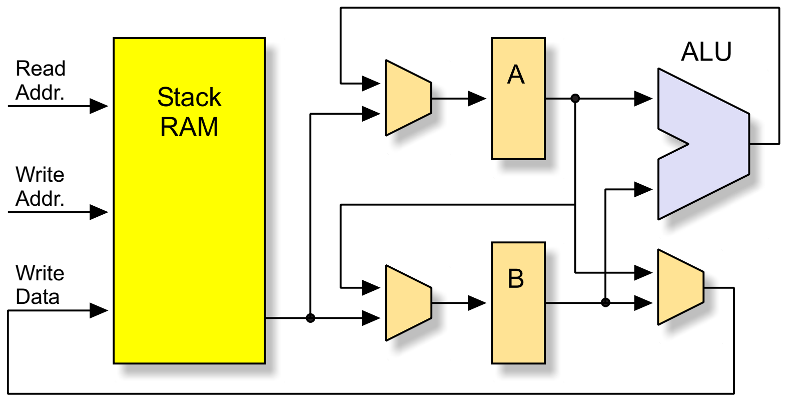


Figure 4.9: The two-level stack cache

Data Forwarding – A Non-Issue

Data dependencies in the instruction stream result in the so-called *data hazards* [63] in the pipeline. Data forwarding is a technique that moves data from a later pipeline stage back to an earlier one to solve this problem. The term *forward* is correct in the temporal domain as data is transferred to an instruction in the future. However, it is misleading in the structural domain as the forward direction is towards the *last* pipeline stage for an instruction.

As the probability of data dependency is very high in a stack-based architecture, one would expect several data forwarding paths to be necessary. However, in the two-level architecture, with its resulting three-stage pipeline, no data hazards will occur and no data forwarding is therefore necessary. This simplifies the decoding stage and reduces the number of multiplexers in the execution path. We will show that none of the three data hazard types [63] is an issue in this architecture. With instructions i and j, where i is issued before j, the data hazard types are:

Read after write: j reads a source before i writes it. This is the most common type of hazard and, in the architectures described above, is solved by using the ALU buffers and the forwarding multiplexer in the ALU datapath. On a stack architecture, write takes three forms:

- Implicit write of TOS during an ALU operation
- Write to the TOS during a load instruction

- Write to an arbitrary entry of the stack with a store instruction

A read also occurs in three different forms:

- Read two top values from the stack for an ALU operation
- Read TOS for a store instruction
- Read an arbitrary entry of the stack with the load instruction

With the two top elements of the stack as discrete registers, these values are read, operated on and written back in the same cycle. No read that depends on TOS or TOS-1 suffers from a data hazard. Read and write access to a local variable is also performed in the same pipeline stage. Thus, the read after write order is not affected. However, there is also an additional hidden read and write: the fill and spill of register B:

B fill: *B* is written during an ALU operation and on a variable store. During an ALU operation, the operands are the values from *A* and the old value from *B*. The new value for *B* is read from the stack memory and does not depend on the new value of *A*. During a variable store operation, *A* is written to the stack memory and does not depend on *B*. The new value for *B* is also read from the stack memory and it is not obvious that this value does not depend on the written value. However, the variable area and the operand stack are distinct areas in the stack (this changes only on method invocation and return), guaranteeing that concurrent read/write access does not produce a data hazard.

B spill: *B* is read on a load operation. The new value of *B* is the old value of *A* and does not therefore depend on the stack memory read. *B* is written to the stack. For the read value from the stack memory that goes to *A*, the argument concerning the distinct stack areas in the case of *B fill* described above still applies.

Write after read: *j* writes a destination before it is read by *i*. This cannot take place as all reads and writes are performed in the same pipeline stage keeping the instruction order.

Write after write: *j* writes an operand before it is written by *i*. This hazard is not present in this architecture as all writes are performed in the same pipeline stage.

4.4.5 Summary

In this section, the stack architecture of the JVM was analyzed. We have seen that the JVM is different from the classical stack architecture. The JVM uses the stack both as an operand stack *and* as the storage place for local variables. Local variables are placed in the stack at a *deeper* position. To load and store these variables, an access path to an arbitrary position in the stack is necessary. As the stack is the most frequently accessed memory area in the JVM, caching of this memory is mandatory for a high-performing Java processor.

A common solution, found in a number of different Java processors, is to implement this stack cache as a standard three-port register file with additional support to address this register file in a stack like manner. The architectures presented above differ in the realization of the register file: as a discrete register or in on-chip memory. Implementing the stack cache as discrete registers is very expensive. A three-port memory is also an expensive option for an ASIC and unusual in an FPGA. It can be emulated by two memories with a single read and write port. However, this solution also doubles the amount of memory.

Detailed analysis of the access patterns to the stack showed that only the two top elements of the stack are accessed in a single cycle. Given this fact, the architecture uses registers to cache only the two top elements of the stack. The next level of the stack cache is provided by a simple on-chip memory. The memory automatically spills and fills the second register. Implementing the two top elements of the stack as fixed registers, instead of elements that are indexed by a stack pointer, also greatly simplifies the overall pipeline.

4.5 The Method Cache

Worst-case execution time (WCET) analysis [109] of real-time programs is essential for any schedulability analysis. To provide a low WCET value, a good processor model is necessary. However, caches for the instructions and data is a classic example of the paradigm *Make the common case fast*, which complicates WCET analysis. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET value. Plenty of effort has gone into research into integrating the instruction cache in the timing analysis of tasks [11, 60, 81] and the influence of the cache on task preemption [78, 31]. The influence of different cache architectures on WCET analysis is described in [61].

We will tackle this problem from the architectural side – an instruction cache organization in which simpler and more accurate WCET analysis is more important than average case performance.

In this section, we will explore the method cache, as it is implemented in JOP, with a novel replacement policy. In Java bytecode only relative branches exist, and a method is therefore only left when a return instruction has been executed.[5] It has been observed that methods are typically short (see [123]) in Java applications. These properties are utilized by a cache architecture that stores complete methods. A complete method is loaded into the cache on both invocation and return. This cache fill strategy lumps all cache misses together and is very simple to analyze.

The method cache was first presented in [120] and is now also used by the Java processor SHAP [102]. Furthermore, the idea has been adapted for a processor that executes compiled C programs [86].

4.5.1 Method Cache Architecture

In this section, we will develop a solution for a time-predictable instruction cache. Typical Java programs consist of short methods. There are no branches out of the method and all branches inside are relative. In the described architecture, the full code of a method is loaded into the cache before execution. The cache is filled on invocations and returns. This means that all cache fills are lumped together with a known execution time. The full loaded method and relative addressing inside a method also result in a simpler cache. Tag memory and address translation are not necessary.

In the method cache several cache blocks (similar to cache lines) are used for a method. The main difference from a conventional cache is that the blocks for a method are all loaded

[5] An uncaught exception also results in a method exit.

```
a() {
    for (;;) {
        b();
        c();
    }
    ...
}
```

Listing 4.5: Code fragment for the replacement example

at once and need to be consecutive. Choosing the block size is now a major design decision. Smaller block sizes allow better memory usage, but the search time for a hit also increases.

With varying numbers of blocks per method, an LRU replacement is impractical. When the method found to be LRU is smaller than the loaded method, this new method invalidates two cached methods.

For the replacement, we will use a pointer *next* that indicates the start of the blocks to be replaced on a cache miss. Two practical replace policies are:

Next block: At the very first beginning, *next* points to the first block. When a method of length l is loaded into the block n, *next* is updated to $(n+l)$ *mod block count*. This replacement policy is effectively FIFO.

Stack oriented: *next* is updated in the same way as before on a method load. It is also updated on a method return – independent of a resulting hit or miss – to point to the first block of the leaving method.

We will show the operation of these different replacement policies in an example with three methods: a(), b() and c() of block sizes 2, 2 and 1. The cache consists of 4 blocks and is therefore too small to hold all the methods during the execution of the code fragment shown in Listing 4.5. Tables 4.3 and 4.4 show the cache content during program execution for both replacement policies. The content of the cache blocks is shown after the execution of the invoke or return instruction. An uppercase letter indicates that this block has been newly loaded. A right arrow depicts the block to be replaced on a cache miss (the *next* pointer). The last row shows the number of blocks that are filled during the execution of the program.

In this example, the stack oriented approach needs fewer fills, as only methods b() and c() are exchanged and method a() stays in the cache. However, if, for example, method b() is the size of one block, all methods can be held in the cache using the the *next block* policy,

	a()	b()	ret	c()	ret	b()	ret	c()	ret	b()	ret
Block 1	A	→a	→a	C	c	B	b	b	→-	B	b
Block 2	A	a	a	→-	A	→a	→a	C	c	B	b
Block 3	→-	B	b	b	A	a	a	→-	A	→a	→a
Block 4	-	B	b	b	→-	B	b	b	A	a	a
Fill	2	4		5	7	9		11	13	15	

Table 4.3: Next block replacement policy

	a()	b()	ret	c()	ret	b()	ret	c()	ret	b()	ret
Block 1	A	→a	a	a	a	→a	a	a	a	→a	a
Block 2	A	a	a	a	a	a	a	a	a	a	a
Block 3	→-	B	→b	C	→c	B	→b	C	→c	B	→b
Block 4	-	B	b	→-	-	B	b	→-	-	B	b
Fill	2	4		5		7		8		10	

Table 4.4: Stack oriented replacement policy

but b() and c() would be still exchanged using the *stack* policy. Therefore, the first approach is used in the JOP's cache.

4.5.2 WCET Analysis

The instruction cache is designed to simplify WCET analysis. Due to the fact that all cache misses are only included in two instructions (*invoke* and *return*), the instruction cache can be ignored on all other instructions. The time needed to load a complete method is calculated using the memory properties (latency and bandwidth) and the length of the method. On an invoke, the length of the invoked method is used, and on a return, the method length of the caller is used to calculate the load time.

With a single method cache this calculation can be further simplified. For every invoke there is a corresponding return. That means that the time needed for the cache load on return can be included in the time for the invoke instruction. This is simpler because both methods, the caller and the callee, are known at the occurrence of the invoke instruction. The information about which method was the caller need not be stored for the return instruction to be analyzed.

With more than one method in the cache, a cache hit detection has to be performed as part of the WCET analysis. If there are only two blocks, this is trivial, as (i) a hit on invoke is only possible if the method is the same as the last invoked (e.g. a single method in a loop) and (ii) a hit on return is only possible when the method is a leaf in the call tree. In the latter case, it is always a hit.

When the cache contains more blocks (i.e. more than two methods can be cached), a part of the call tree has to be taken into account for hit detection. The method cache further complicates the analysis, as the method length also determines the cache content. However, this analysis is still simpler than a cache modeling of a direct-mapped instruction cache, as cache block replacement depends on the call tree instead of instruction addresses.

WCET analysis of cache hits for the method cache is most beneficial for methods invoked in a loop, where the methods are classified as first miss. The basic idea of the method cache analysis is as follows: Within a loop it is statically analyzed if all methods invoked and the invoking method, which contains the loop, fit together in the method cache. If this is the case, all methods will at most miss once. The concrete implementation of the the analysis algorithm is described in [66].

Except for leaf methods, the cache load can be triggered by an invoke instruction of by a return instruction. On an invoke some of the miss penalty is hidden by concurrent microcode execution. Therefore, we have to assume the higher cost of loading methods into the cache on a return instruction for non-leaf methods. Leaf nodes can naturally only miss on an invoke.

In traditional caches, data access and instruction cache fill requests can compete for the main memory bus. For example, a load or store at the end of the processor pipeline competes with an instruction fetch that results in a cache miss. One of the two instructions is stalled for additional cycles by the other instruction. With a data cache, this situation can be even worse. The worst-case scenario for the memory stall time for an instruction fetch or a data load is two miss penalties when both cache reads are a miss. This unpredictable behavior leads to very pessimistic WCET bounds.

A *method cache*, with cache fills only on invoke and return, does not interfere with data access to the main memory. Data in the main memory is accessed with *getfield* and *putfield*, instructions that never overlap with *invoke* and *return*. This property removes another uncertainty found in traditional cache designs.

4.5.3 Caches Compared

In this section, we compare two different cache architectures in a quantitative way. Although our primary concern is predictability, performance remains important. We will therefore first present the results from a conventional direct-mapped instruction cache. These measurements provide a baseline for the evaluation of the described cache architecture.

Cache performance varies with different application domains. As the system is intended for real-time applications, the benchmark for these tests should reflect this fact. However, there are no standard benchmarks available for embedded real-time systems. A real-time application was therefore adapted to create this benchmark. The application is from one node of a distributed motor control system [117] (see also Section 11.4.1). A simulation of the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark for simulating the real-world workload.

The data for all measurements was captured using a simulation of JOP and running the application for 500,000 clock cycles. During this time, the major loop of the application was executed several hundred times, effectively rendering any misses during the initialization code irrelevant to the measurements. As the embedded application is quite small (1366 LOC), small instruction caches have been simulated.

WCET analysis based comparison of the method cache and of standard instruction caches is currently under development. Therefore, we perform only average case measurements for a comparison between a time-predictable cache organization and a standard cache organization. With a simulation of JOP, we measure the cache misses and miss penalties for different configurations of the method cache and a direct-mapped cache. The miss penalty and the resulting effect on the execution time depend on the main memory system. Therefore, we simulate three different memory technologies: static RAM (SRAM), synchronous DRAM (SDRAM), and double data rate (DDR) SDRAM. For the SRAM, a latency of 1 clock cycle and an access time of 2 clock cycles per 32-bit word are assumed. For the SDRAM, a latency of 5 cycles (3 cycles for the row address and 2 cycles for the CAS latency) is assumed. The SDRAM delivers one word (4 bytes) per cycle. The DDR SDRAM has a shorter latency of 4.5 cycles and transfers data on both the rising and falling edge of the clock signal.

The resulting miss cycles are scaled to the bandwidth consumed by the instruction fetch unit. The result is the number of cache fill cycles per fetched instruction byte. In other words: the average main memory access time in cycles per instruction byte. A value of 0.1 means that for every 10 fetched instruction bytes, one clock cycle is spent to fill the cache.

Table 4.5 shows the result for different configurations of a direct-mapped cache. Which

Table 4.5: Direct-mapped cache, average memory access time

Cache size	Block size	SRAM	SDRAM	DDR
1 KB	8 B	**0.18**	0.25	0.19
1 KB	16 B	0.22	**0.22**	0.16
1 KB	32 B	0.31	0.24	**0.15**
2 KB	8 B	**0.11**	0.15	0.12
2 KB	16 B	0.14	**0.14**	**0.10**
2 KB	32 B	0.22	0.17	0.11

Table 4.6: Method cache, average memory access time

Cache size	Block size	SRAM	SDRAM	DDR
1 KB	16 B	0.36	0.21	0.12
1 KB	32 B	0.36	0.21	0.12
1 KB	64 B	0.36	0.22	0.12
1 KB	128 B	0.41	0.24	0.14
2 KB	32 B	0.06	0.04	0.02
2 KB	64 B	0.12	0.08	0.04
2 KB	128 B	0.19	0.11	0.06
2 KB	256 B	0.37	0.22	0.13

configuration performs best depends on the relationship between memory bandwidth and memory latency. The data in bold emphasize the best block size for the different memory technologies. As expected, memories with a higher latency and bandwidth perform better with larger block sizes. For small block sizes, the latency clearly dominates the access time. Although the SRAM has half the bandwidth of the SDRAM and a quarter of the DDR SDRAM, it is faster than the SDRAM memories with a block size of 8 byte. In most cases a block size of 16 bytes is fastest.

Table 4.6 shows the average memory access time per instruction byte for the method cache. Because we load full methods, we have chosen larger block sizes than for a standard cache. All configurations benefit from a memory system with a higher bandwidth. The method cache is less latency sensitive than the direct-mapped instruction cache. For the small 1 KB cache the access time is almost independent of the block size. The capacity misses dominate. From the 2 KB configuration we see that smaller block sizes result in less cache misses. However, smaller block sizes result in more hardware for the hit detection since the method cache is in effect fully associatively. Therefore, we need a balance between the number of blocks and the performance.

The cache conflict is high for the small configuration with 1 KB cache. The direct-mapped cache, backed up with a low-latency main memory, performs better than the method cache. When high-latency memories are used, the method cache performs better than the direct mapped cache. This is expected as the long latency for a transfer is amortized when more data (the whole method) is filled in one request.

A small block size of 32 Bytes is needed in the 2 KB method cache to outperform the direct mapped cache with the low-latency main memory as represented by the SRAM. For higher latency memories (SDRAM and DDR), a method cache with a block size of 128 bytes outperforms the direct mapped instruction cache.

The comparison does not show if the method cache is more easily predictable than other cache solutions. It shows that caching full methods performs similarly to standard caching techniques.

4.5.4 Summary

From the properties of the Java language – usually small methods and relative branches – we derived the novel idea of a *method cache*, i.e. a cache organization in which whole methods are loaded into the cache on method invocation and the return from a method. This cache organization is time-predictable, as all cache misses are lumped together in these two instructions. Using only one block for a single method introduces considerable overheads in comparison with a conventional cache, but is very simple to analyze. We extended this cache to hold more methods, with several smaller blocks per method.

Comparing these organizations quantitatively with a benchmark derived from a real-time application, we have seen that the method cache performs similarly to (and in some configurations even better than) a direct-mapped cache. Only filling the cache on method invocation and return simplifies WCET analysis and removes another source of uncertainty, as there is no competition for the main memory access between instruction cache and data cache.

5 Runtime System

A Java processor alone is not a complete JVM. This chapter describes the definition of a real-time profile for Java and the description of the JVM internal data structures to represent classes and objects.

5.1 A Real-Time Profile for Embedded Java

As standard Java is under-specified for real-time systems and the RTSJ does not fit for small embedded systems a new and simpler real-time profile is defined in this section and implemented on JOP. The guidelines of the specification are:

- High-integrity profile
- Easy syntax
- Easy to implement
- Low runtime overhead
- No syntactic extension of Java
- Minimum change of Java semantics
- Support for time measurement if a WCET analysis tool is not available
- Known overheads (documentation of runtime behavior and memory requirements of every JVM operation and all methods have to be provided)

The real-time profile under discussion is inspired by the restricted versions of the RTSJ described in [111] and [77]. It is intended for high-integrity real-time applications and as a test case to evaluate the architecture of JOP as a Java processor for real-time systems.

The proposed definition is not compatible with the RTSJ. Since the application domain for the RTSJ is different from high-integrity systems, it makes sense for it *not* to be compatible with the RTSJ. Restrictions can be enforced by defining new classes (e.g. setting thread priority in the constructor of a real-time thread alone, enforcing minimum interarrival times for sporadic events).

To verify that this specification is expressive enough for high-integrity real-time applications, Ravenscar-Java (RJ) [77], with the additional necessary RTSJ classes, has been implemented on top of it. However, RJ inherits some of the complexity of the RTSJ. Therefore, the implementation of RJ has a larger memory and runtime overhead than this simple specification.

When the specification for Safety-Critical Java (JSR 302)[1] [64] will be finalized the profile will be adapted for this specification.

5.1.1 Application Structure

The application is divided in two different phases: *initialization* and *mission*. All non time-critical initialization, global object allocations, thread creation and startup are performed in the initialization phase. All classes need to be loaded and initialized in this phase. The mission phase starts after invocation of startMission(). The number of threads is fixed and the assigned priorities remain unchanged. The following restrictions apply to the application:

- Initialization and mission phase
- Fixed number of threads
- Threads are created at initialization phase
- All shared objects are allocated at initialization

5.1.2 Threads

Concurrency is expressed with two types of *schedulable objects*:

Periodic activities are represented by threads that execute in an infinite loop invoking waitForNextPeriod() to get rescheduled in predefined time intervals.

Asynchronous sporadic activities are represented by event handlers. Each event handler is in fact a thread, which is released by an hardware interrupt or a software generated event (invocation of fire()). Minimum interarrival time has to be specified on creation of the event handler.

[1]http://jcp.org/en/jsr/detail?id=302

The classes that implement the *schedulable objects* are:

RtThread represents a periodic task. As usual task work is coded in run(), which gets invoked on missionStart(). A scoped memory object can be attached to an RtThread at creation.

SwEvent represents a software-generated event. It is triggered by fire() and needs to override handle().

Listing 5.1 shows the definition of the basic classes. Listing 5.2 shows the principle coding of a worker thread. An example for creation of two real-time threads and an event handler can be seen in Listing 5.3.

5.1.3 Scheduling

The scheduler is a preemptive, priority-based scheduler with unlimited priority levels and a unique priority value for each schedulable object. No real-time threads or events are scheduled during the initialization phase.

The design decision to use unique priority levels, instead of FIFO within priorities, is based on following facts: Two common ways to assign priorities are rate monotonic and, in a more general form, deadline monotonic assignment. When two tasks are given the same priority, we can choose one of them and assign a higher priority to that task and the task set will still be schedulable. This results in a strictly monotonic priority order and we do not need to deal with FIFO order. This eliminates queues for each priority level and results in a single, priority ordered task list with unlimited priority levels.

Synchronized blocks are executed with priority ceiling emulation protocol. In the current implementation top priority is assumed for all objects. This avoids priority inversions on objects that are not accessible from the application (e.g. objects inside a library).

5.1.4 Memory

The profile does not support a garbage collector.[2] All memory should be allocated at the initialization phase. Without a garbage collector, the heap implicitly becomes immortal memory (as defined by the RTSJ). Scoped memory is currently in a prototyping stage, but will be supported in the future.

[2]This restriction can be relaxed when the real-time GC for JOP is used (see Chapter 7).

```
public class RtThread {

    public RtThread(int priority , int period)
    public RtThread(int priority , int period, int offset )
    public RtThread(int priority , int period, Memory mem)
    public RtThread(int priority , int period, int offset ,
                    Memory mem)

    public void setProcessor(int id )

    public void run()
    public boolean waitForNextPeriod()

    public static void startMission ()

    public static void sleepMs(int millis )
    public static void busyWait(int us)
    public static RtThread currentRtThread()
}

public class SwEvent extends RtThread {

    public SwEvent(int priority , int minTime)
    public SwEvent(int priority , int minTime, Memory mem)

    public final void fire ()
    public void handle()
}
```

Listing 5.1: Schedulable objects

```
public class Worker extends RtThread {

    private SwEvent event;

    public Worker(int p, int t,
                SwEvent ev) {

        super(p, t,
            // create a scoped memory area
            new Memory(10000)
        );
        event = ev;
        init();
    }

    private void init() {
        // all initialzation stuff
        // has to be placed here
    }

    public void run() {

        for (;;) {
            work();          // do some work
            event.fire();    // and fire an event

            // wait for next period
            if (!waitForNextPeriod()) {
                missedDeadline();
            }
        }
        // should never reach this point
    }
}
```

Listing 5.2: A periodic real-time thread

```
// create an Event
Handler h = new Handler(3, 1000);

// create two worker threads with
// priorities according to their periods
FastWorker fw = new FastWorker(2, 2000);
Worker w = new Worker(1, 10000, h);

// change to mission phase for all
// periodic threads and event handler
RtThread.startMission();

// do some non real−time work
// and invoke sleep() or yield ()
for (;;) {
    watchdogBlink();
    RtThread.sleepMs(500);
}
```

Listing 5.3: Start of the application

5.1.5 Restrictions on Java

A list of some of the language features that should be avoided for WCET analyzable real-time threads and bound memory usage:

WCET: Only analyzable language constructs are allowed (see [109]).

Static class initialization: Static class initializer are invoked at JVM boot. No cyclic dependency is allowed and the initialization order is determined at class link time. This violation of the JVM specification is necessary for tight WCET values of bytecodes new, getstatic, and putstatic, which usually trigger class initialization.

Inheritance: Reduce usage of interfaces and overridden methods.

String concatenation: In the immortal memory scope, only string concatenation with string literals is allowed.

Finalization: finalize() has a weak definition in Java. Because real-time systems run *forever*, objects in the heap, which is immortal in this specification, will never be finalized.

Dynamic Class Loading: Due to the implementation and WCET analysis complexity dynamic class loading is avoided.

5.1.6 Interaction of RtThread, the Scheduler, and the JVM

Figure 5.1 shows an interaction example of the scheduler, the application, and the JVM. The interaction diagram shows the message sequences between two application tasks, the scheduler, the JVM and the hardware. The hardware represents interrupt and timer logic. Task 2 is a periodic task with a higher priority than Task 1.

The first event is a timer event to unblock Task 2 for a new period. The generated timer event results in a call of the scheduler. The scheduler performs its scheduling decision and issues a context switch to Task 2. With every context switch the timer is reprogrammed to generate an interrupt at the next time triggered event for a higher priority task. Task 2 performs the periodic work and ceases execution by invocation of waitForNextPeriod(). The scheduler is called and requests an interrupt from the hardware resulting in the same call sequence as with a timer. The software generated interrupt imposes negligible overhead and results in a single entry point for the scheduler. Task 1 is the only ready task in this example and is resumed by the scheduler.

5.1.7 Implementation Results

The initial idea was to implement scheduling and dispatching in microcode. However, many Java bytecodes have a one to one mapping to a microcode instruction, resulting in a single cycle execution. The performance gain of an algorithm coded in microcode is therefore negligible. As a result, almost all of the scheduling is implemented in Java. Only a small part of the dispatcher, a memory copy, is implemented in microcode and exposed with a special bytecode.

Experimental results of basic scheduling benchmarks, such as periodic thread jitter, context switch time for threads and asynchronous events, can be found in [119].

To implement system functions, such as scheduling, in Java, access to JVM and processor internal data structures have to be available. However, Java does not allow memory access or access to hardware devices. In JOP, this access is provided by additional bytecodes. In the Java environment, these bytecodes are represented as static native methods. The compiled invoke instruction for these methods (invokestatic) is replaced by these additional bytecodes in the class file. This solution provides a very efficient way to incorporate low-level functions into a pure Java system. The translation can be performed during class loading to avoid non-standard class files.

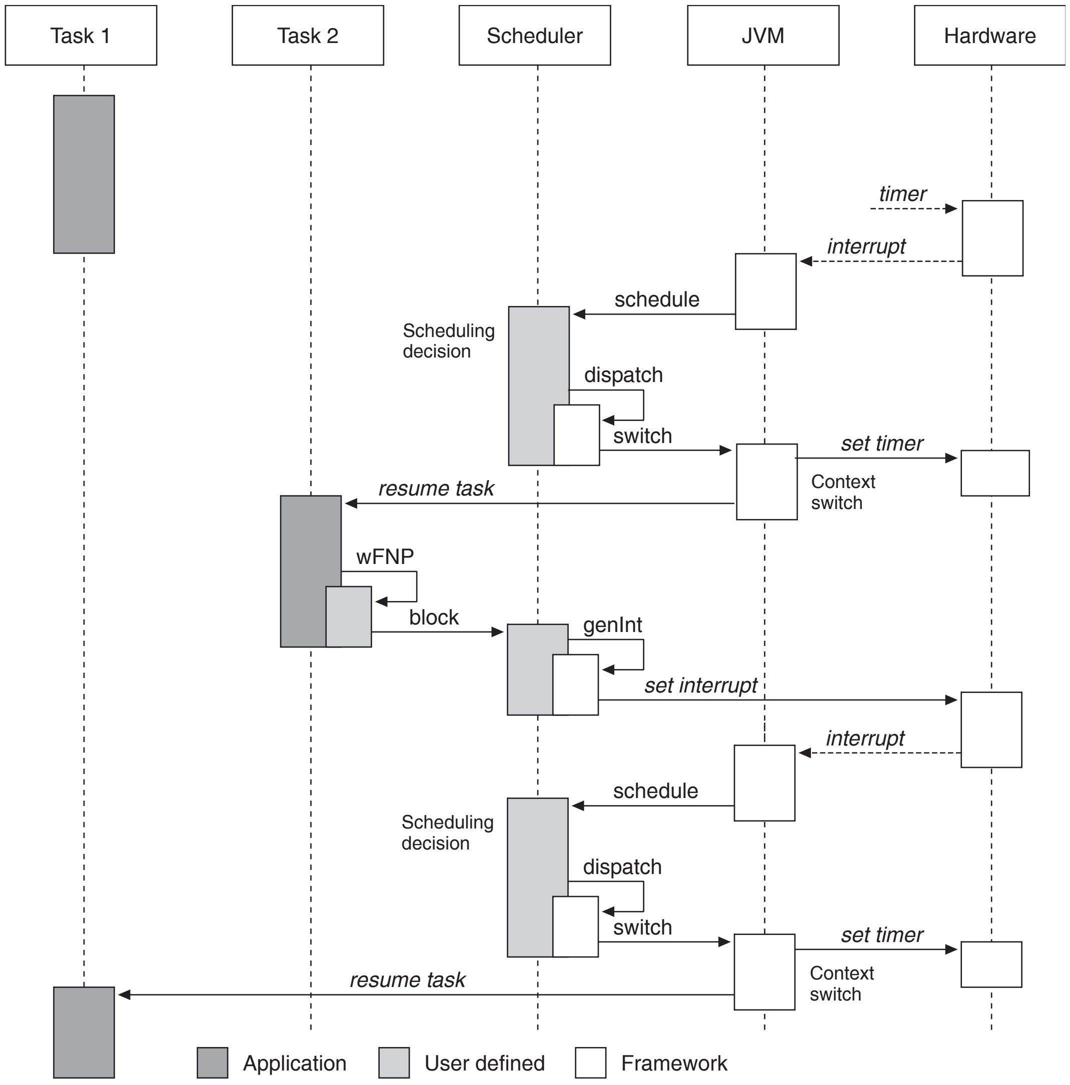


Figure 5.1: Interaction and message exchange between the application, the scheduler, the JVM, and the hardware

A pure Java system, without an underlying RTOS, is an unusual system with some interesting new properties. Java is a safer execution environment than C (e.g. no pointers) and the boundary between *kernel* and *user space* can become quite loose. Scheduling, usually part of the operating system or the JVM, is implemented in Java and executed in the same context as the application.

This property provides an easy path to a framework for user-defined scheduling. In [118] a framework for a user defined scheduler for JOP is presented. The events that result in the scheduling decision are listed. Hooks for these events allow the implementation of a user defined scheduler.

5.1.8 Summary

This section consider a simple profile for real-time Java and the implementation of real-time scheduling on a Java processor. The novelty of the described approach is in implementing functions usually associated with an RTOS in Java. That means that real-time Java is not based on an RTOS, and therefore not restricted to the functionality provided by the RTOS. With JOP, a self-contained real-time system in pure Java becomes possible. The implementation of the specification is successfully used as the basis for a commercial real-time application in the railway industry.

5.2 A Profile for Safety Critical Java

The proposed profile is an refinement of the profile described in Section 5.1. Further development of applications on JOP shall be based on this profile and the current applications (e.g. ÖBB bg and Lift) should be migrated to this profile.

5.2.1 Introduction

Safety-critical Java (SCJ) builds upon a broad research base on using Java (and Ada) for hard real-time systems, sometimes also called high integrity systems. The Ravenscar profile defines a subset of Ada to support development of safety-critical systems [39]. Based on the concepts of Ravenscar Ada a restriction of the RTSJ was first proposed in [111]. These restrictions are similar to SCJ level 1 without the mission concept. The idea was further extended in [77] and named the Ravenscar Java profile (RJ). RJ is an extended subset of RTSJ that removes features considered unsafe for high integrity systems. Another profile for safety-critical systems was proposed within the EU project HIJA [1].

PERC Pico from Aonix [6] defines a Java environment for hard real-time systems. PERC Pico defines its own class hierarchy for real-time Java classes which are based on the RTSJ libraries, but are not a strict subset thereof. PERC Pico introduces stack-allocated scopes, an elaborate annotation system, and an integrated static analysis system to verify scope safety and analyze memory and CPU time resource requirements for hard real-time software components. Some of the annotations used to describe the libraries of the SCJ are derived indirectly from the annotation system used in PERC Pico.

Another definition of a profile for safety-critical Java was published in [137]. In contrast to RJ the authors of that profile argue for new classes instead of reusing RTSJ based classes to avoid inheriting unsafe RTSJ features and to simplify the class hierarchy. A proposal for mission modes within the former profile [126] permits recycling CPU time budgets for different phases of the application. Compared to the mission concept of SCJ that proposal allows periodic threads to vote against the shutdown of a mission. The concept of mission memory is not part of that proposal.

In this chapter we focus on the safety-critical Java (SCJ) specification, a new standard for safety critical applications developed by the JSR 302 expert group [64]. We should note that the JSR 302 has not been finalized, thus we present the version from [126]. When the JSR 302 stabilizes it will be implemented for JOP and will be the future version of the real-time profile.

5.2.2 SCJ Level 1

Level 1 of the SCJ requires that all threads be defined during an initial *initialization* phase. This phase is run only once at virtual machine startup. The second phase, called the *mission* phase, begins only when all threads have been started. This phase runs until virtual machine shutdown. Level 1 supports only two kinds of schedulable objects: periodic threads and sporadic events. The latter can be generated by either hardware or software. This restrictions keeps the schedulability analysis simple. In SCJ priority ceiling emulation is the default monitor control policy. The default ceiling is top priority.

The Java wait and notify primitives are not allowed in SCJ level 0 and 1. This further simplifies analysis. The consequence is that a thread context switch can only occur if a higher priority thread is released or if the current running thread yields (in the case of SCJ by returning from the run() method).

In the RTSJ, periodic tasks are expressed by unbounded loops with, at some point, a call to the waitForNextPeriod() (or wFNP() for short) method of class RealtimeThread. This has the effect of yielding control to the scheduler which will only wake the thread when its next period starts or shortly thereafter. In SCJ, as a simplification, periodic logic is encapsulated

```
package javax. safetycritical ;

public abstract class RealtimeThread {

    protected RealtimeThread(RelativeTime period,
        RelativeTime deadline,
        RelativeTime offset, int memSize)

    protected RealtimeThread(String event,
        RelativeTime minInterval,
        RelativeTime deadline, int memSize)

    abstract protected boolean run();

    protected boolean cleanup() {
        return true;
    }
}

public abstract class PeriodicThread
        extends RealtimeThread {

    public PeriodicThread(RelativeTime period,
        RelativeTime deadline,
        RelativeTime offset, int memSize)

    public PeriodicThread(RelativeTime period)
}
```

Listing 5.4: Periodic thread definition for SCJ

in a run() method which is invoked at the start of every period of a given schedulable object. When the thread returns from run() it is blocked until the next period.

Listing 5.4 shows part of the definition of the SCJ thread classes from [137]. Listing 5.5 shows the code for a periodic thread. This class has only one run() method which performs a periodic computation.

The loop construct with wFNP() is not used. The main intention to avoid the loop construct, with the possibility to split application logic into *mini* phases, is simplification of the

```
new PeriodicThread(
    new RelativeTime(...)) {

        protected boolean run() {
            doPeriodicWork();
            return true;
        }
};
```

Listing 5.5: A periodic application thread in SCJ

WCET analysis. Only a single method has to be analyzed per thread instead of all possible control flow path between wFNP() invocations.

We contrast the SCJ threading with Listing 5.6 where a periodic RTSJ thread is shown. Suspension of the thread to wait for the next period is performed by an explicit invocation of wFNP(). The coding style in this example makes analysis of the code more difficult than necessary. First the initialization logic is mixed with the code of the mission phase, this means that a static analysis may be required to discover the boundary between the startup code and the periodic behavior. The code also performs mode switches with calls to wFNP() embedded in the logic. This makes the worst case analysis more complex as calls to wFNP() may occur anywhere and require deep understanding of feasible control flow paths. Another issue, which does not affect correctness, is the fact that object references can be preserved in local variables across calls to wFNP(). As we will see later this has implications for the GC.

The notation of a run() method that is invoked periodically also simplifies garbage collection (see Chapter 7). When the GC thread is scheduled at a lower priority than the application threads it will never interrupt the run() method. The GC will only execute when all threads have returned from the run() method and the stack is empty. Therefore, stack scanning for roots can be omitted.

```
public void run() {

    State local = new State();
    doSomeInit();
    local.setA();
    waitForNextPeriod();

    for (;;) {
        while (!switchToB()) {
            doModeAwork();
            waitForNextPeriod();
        }
        local.setB();
        while (!switchToA()) {
            doModeBWork();
            waitForNextPeriod();
        }
        local.setA();
    }
}
```

Listing 5.6: Possible logic for a periodic thread in the RTSJ

5.3 JVM Architecture

This section presents the details of the implementation of the JVM on JOP. The representation of objects and the stack frame is chosen to support JOP as a processor for real-time systems. However, since the data structures are realized through microcode they can be easily changed for a system with different needs.

5.3.1 Runtime Data Structures

Memory is addressed as 32-bit data, which means that memory pointers are incremented for every four bytes. No single byte or 16-bit access is necessary. The abstract type reference is a pointer (via a handle indirection) to memory that represents the object or an array. The reference is pushed on the stack before an instruction can operate on it. A null reference is represented by the value 0.

Stack Frame

On invocation of a method, the invoker's context is saved in a newly allocated frame on the stack. It is restored when the method returns. The saved context consists of the following registers:

SP: Immediately before invocation, the stack pointer points to the last argument for the called function. This value is reduced by the argument count (i.e. the arguments are consumed) and saved in the new stack frame.

PC: The pointer to the next bytecode instruction after the invoke instruction.

VP: The pointer to the memory area on the stack that contains the locals.

CP: The pointer to the constant pool of the class from the invoking method.

MP: The pointer to the method structure of the invoking method.

SP, PC and VP are registers in JOP while CP and MP are local variables of the JVM. Figure 5.2 provides an example of the stack before and after invoking a method. In this example, the called method has two arguments and contains two local variables. If the method is a virtual one, the first argument is the reference to the object (the *this*-pointer). The arguments implicitly become locals in the called method and are accessed in the same way as local variables. The start of the stack frame (*Frame* in the figure) needs not to be

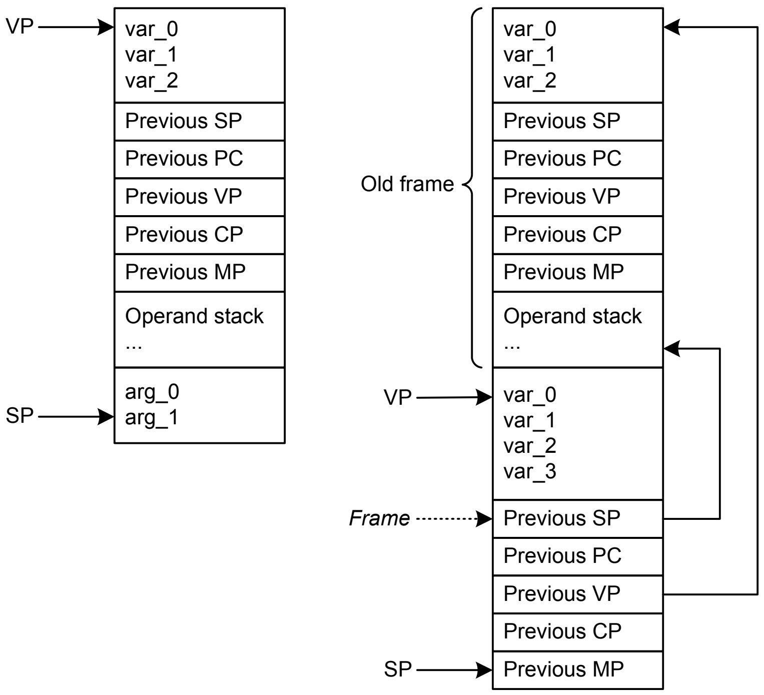


Figure 5.2: Stack change on method invocation

saved. It is not needed during execution of the method or on return. To access the starting address of the frame (e.g. for an exception) it can be calculated with information from the method structure:

$$Frame = VP + arg\_cnt + locals\_cnt$$

Object Layout

Figure 5.3 shows the representation of an object in memory. The object reference points to a handle area and the first element in the handle area points to the first instance variable of the object. At the offset 1 in the handle area, a pointer is located to access class information. To speed-up method invocation, it points directly to the method table of the objects class instead of the beginning of the class data. The handle indirection simplifies the implementation of a compacting GC (see Chapter 7).

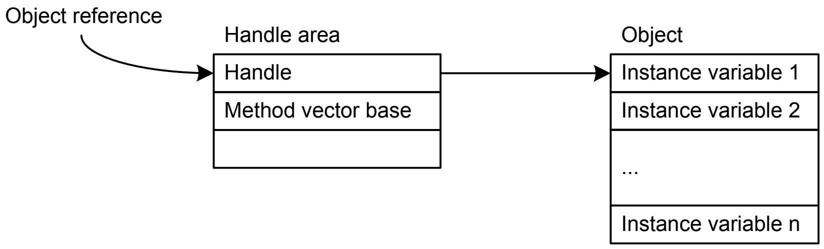


Figure 5.3: Object format

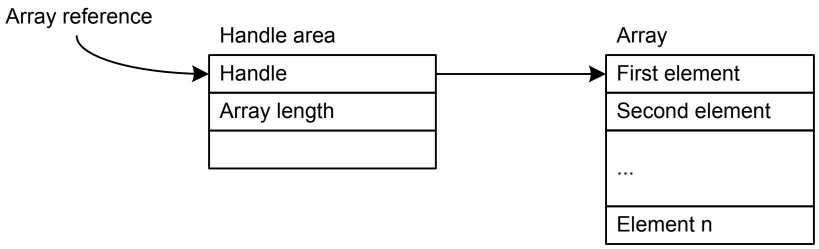


Figure 5.4: Array format

Array Layout

Figure 5.4 shows the representation of an array in memory. The array reference points to a handle area and the first element in the handle area points to the first element of the array. At the offset 1 in the handle area, the length of the array can be found.

Class Structure

Runtime class information, as shown in Figure 5.5, consists of the instance size, GC info, the dispatch table for the methods, a reference to the super class, the constant pool, and an optional interface table. The class variables (static fields) are located at the start of the memory to speedup access to the fields (the constant pool index of getstatic and putstatic is converted at class link time to the address in memory). Furthermore all reference static fields are located in one continuous region for simple GC root scanning. A pointer to the static primitive fields of the class is needed for the implementation of hardware objects.

The class reference is obtained from the constant pool when a new object is created. The method vector base pointer is a reference from an object to its class (see Figure 5.3). It

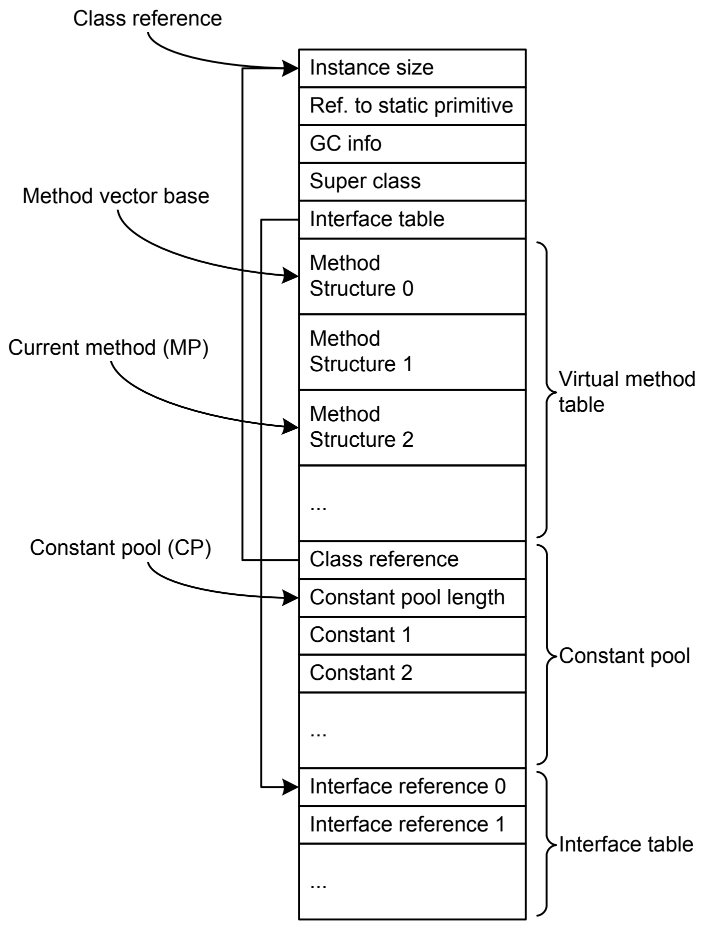


Figure 5.5: Runtime class structure

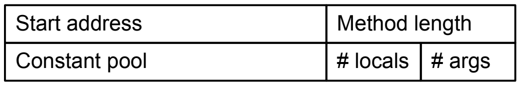


Figure 5.6: Method structure

is used on invokevirtual. The index is the argument of the bytecode to avoid an additional memory access in invoke (the original index into the constant pool is overwritten by the direct index at class link time). A pointer to the method structure of the current method is saved in the JVM variable MP. The method structure, as shown in Figure 5.6, contains the starting address and length of the method (in 32-bit words), argument and local variable count and a pointer to the constant pool of the class. Since the constant pool is an often accessed memory area, a pointer to it is cached in the JVM variable CP.

The interface table contains references to the method structures of the implementation. Only classes that implement an interface contain this table. To avoid searching the class hierarchy on invokeinterface, each interface method is assigned a unique index. This arrangement provides constant execution time, but can lead to large interface tables.

The constant pool contains various constants of a class. The entry at index 0 is the length of the pool. All constants, which are symbolic in the class files, are resolved during class linking. The different constant types and their values after resolving are listed in Table 5.1. The names for the types are the same as in the JVM specification [82].

5.3.2 Class Initialization

According to [82] the static initializers of a class C are executed immediately before one of the following occurs: (i) an instance of C is created; (ii) a static method of C is invoked or (iii) a static field of C is used or assigned. The issue with this definition is that it is not allowed to invoke the static initializers at JVM startup and it is not so obvious when it gets invoked.

It follows that the bytecodes getstatic, putstatic, invokestatic and new can lead to class initialization and the possibility of high WCET values. In the JVM, it is necessary to check every execution of these bytecodes if the class is already initialized. This leads to a loss of performance and is violated in some existing implementations of the JVM. For example, the first version of CACAO [75] invokes the static initializer of a class at compilation time. Listing 5.7 shows an example of this problem.

JOPizer tries to find a correct order of the class initializers and puts this list into the application file. If a circular dependency is detected the application will not be built. The

Constant type	Description
Class	A pointer to a class (class reference)
Fieldref	For static fields: a direct pointer to the field For object fields: the position relative to the object reference
Methodref	For static methods: a direct pointer to the method structure For virtual methods: the offset in the method table (= index*2) and the number of arguments
InterfaceMethodref	A system wide unique index into the interface table
String	A pointer to the string object that represents the string constant
Integer	The constant value
Float	The constant value
Long	This constant value spans two entries in the constant pool
Double	Same as for long constants
NameAndType	Not used
Utf8	Not used

Table 5.1: Constant pool entries

class initializers are invoked at JVM startup.

5.3.3 Synchronization

Synchronization is possible with methods and on code blocks. Each object has a monitor associated with it and there are two different ways to gain and release ownership of a monitor. Bytecodes monitorenter and monitorexit explicitly handle synchronization. In other cases, synchronized methods are marked in the class file with the access flags. This means that all bytecodes for method invocation and return must check this access flag. This results in an unnecessary overhead on methods without synchronization. It would be preferable to encapsulate the bytecode of synchronized methods with bytecodes monitorenter and monitorexit. This solution is used in Suns picoJava-II [146]. The code is manipulated in the class loader. Two different ways of coding synchronization, in the bytecode stream and as access flags, are inconsistent. With JOPizer the same manipulation of the methods is performed to wrap the method code in a synchronized block when the method is defined synchronized.

```
public class Problem {

    private static Abc a;
    public static int cnt; // implicitly set to 0

    static {
        // do some class initializaion
        a = new Abc(); //even this is ok.
    }

    public Problem() {
        ++cnt;
    }
}

// anywhere in some other class, in situation ,
// when no instance of Problem has been created
// the following code can lead to
// the execution of the initializer
int nrOfProblems = Problem.cnt;
```

Listing 5.7: Class initialization can occur very late

5.3.4 Booting the JVM

One interesting issue for an embedded system is how the boot-up is performed. On power-up, the FPGA starts the configuration state machine to read the FPGA configuration data either from a Flash or via a download cable (for development). When the configuration has finished, an internal reset is generated. After that reset, microcode instructions are executed starting from address 0. At this stage, we have not yet loaded any application program (Java bytecode). The first sequence in microcode performs this task. The Java application can be loaded from an external Flash or via a serial line (or an USB port) from a PC. The microcode assembly configured the mode. Consequently, the Java application is loaded into the main memory. To simplify the startup code we perform the rest of the startup in Java itself, even when some parts of the JVM are not yet setup.

In the next step, a minimal stack frame is generated and the special method Startup.boot() is invoked. From now on JOP runs in Java mode. The method boot() performs the following steps:

1. Send a greeting message to *stdout*
2. Detect the size of the main memory
3. Initialize the data structures for the garbage collector
4. Initialize java.lang.System
5. Print out JOP's version number, detected clock speed, and memory size
6. Invoke the static class initializers in a predefined order
7. Invoke the main method of the application class

6 Worst-Case Execution Time

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. WCET estimates can be obtained either by measurement or static analysis. The problem with using measurements is that the execution times of tasks tend to be sensitive to their inputs. As a rule, measurement does not guarantee safe WCET estimates. Instead, static analysis is necessary for hard real-time systems. Static analysis is usually divided into a number of different phases:

Path analysis generates the control flow graph (a directed graph of basic blocks) of the program and annotates (manual or automatic) loops with bounds.

Low-level analysis determines the execution time of basic blocks obtained by the path analysis. A model of the processor and the pipeline provides the execution time for the instruction sequence.

Global low-level analysis determines the influence of hardware features such as caches on program execution time. This analysis can use information from the path analysis to provide less pessimistic values.

WCET Calculation The final WCET is calculated by transforming the control flow graph with the timing information of basic blocks[1] with the program annotations for loops to an integer linear programming (ILP) problem. This problem is solved by an ILP solver. This approach is also called implicit path enumeration (IPET).

For the low-level analysis, a good timing model of the processor is needed. The main problem for the low-level analysis is the execution time dependency of instructions in modern processors that are not designed for real-time systems. JOP is designed to be an easy target for WCET analysis. The WCET of each bytecode can be predicted in terms of the number of cycles it requires. There are no timing dependencies between bytecodes.

WCET analysis has to be done at two levels: at the microcode level and at the bytecode level. The microcode WCET analysis is performed only once for a processor configuration

[1] A basic block is a sequence of instructions without any jumps or jump targets within this sequence.

and described in the next section. The result from this microcode analysis is the timing model of the processor. The timing model is the input for WCET analysis at the bytecode level (i.e. the Java application) as explained in Section 6.2.

The first WCET analysis tool that targets JOP has been developed by Rasmus Pedersen [134]. Benedikt Huber implemented a new version of the analyzer with support of bytecodes implemented in Java and a better method cache approximation has [67]. The new tool also contains a module to extract the low-level bytecode timing from the microcode assembler program (jvm.asm) automatically.

6.1 Microcode WCET Analysis

Each bytecode is implemented by microcode. We can obtain the WCET of a single bytecode by performing WCET analysis at the microcode level. To prove that there are no time dependencies between bytecodes, we have to show that no processor states are *shared* between different bytecodes.

6.1.1 Microcode Path Analysis

To obtain the WCET values for the individual bytecodes we perform the path analysis at the microcode level. First, we have to ensure that a number of restrictions (from [109]) of the code are fulfilled:

- Programs must not contain unbounded recursion. This property is satisfied by the fact that there exists no call instruction in microcode.
- Function pointers and computed gotos complicate the path analysis and should therefore be avoided. Only simple conditional branches are available at the microcode level.
- The upper bound of each loop has to be known. This is the only point that has to be verified by inspection of the microcode.

To detect loops in the microcode we have to find all backward branches (e.g. with a negative branch offset).[2] The branch offsets can be found in a VHDL file (offtbl.vhd) that is generated during microcode assembly. In the current implementation of the JVM there are ten different negative offsets. However, not each offset represents a loop. Most of these branches

[2]The loop branch can be a forward branch. However, the basic blocks of the loop contain at least one backward branch. Therefore we can identify all loops by searching for backward branches only.

are used to share common code. Three branches are found in the initialization code of the JVM. They are not part of a bytecode implementation and can be ignored. The only loop that is found in a regular bytecode is in the implementation of imul to perform a fixed delay. The iteration count for this loop is constant.

A few bytecodes are implemented in Java[3] and can be analyzed in the same way as application code. The bytecodes idiv and irem contain a constant loop. The bytecode lookupswitch[4] performs a linear search through a table of branch offsets. The WCET depends on the table size that can be found as part of the instruction.

As the microcode sequences are very short, the calculation of the control flow graph for each bytecode is done manually.

6.1.2 Microcode Low-level Analysis

To calculate the execution time of basic blocks in the microcode, we need to establish the timing of microcode instructions on JOP. All microcode instructions except wait execute in a single cycle, reducing the low-level analysis to a case of merely counting the instructions.

The wait instruction is used to stall the processor and wait for the memory subsystem to finish a memory transaction. The execution time of the wait instruction depends on the memory system and, if the memory system is predictable, has a known WCET. A main memory consisting of SRAM chips can provide this predictability and this solution is therefore advised. The predictable handling of DMA, which is used for the instruction cache fill, is explained in Section 4.5. The wait instruction is the only way to stall the processor. Hardware events, such as interrupts (see Section 4.3.5), do not stall the processor.

Microcode is stored in on-chip memory with single cycle access. Each microcode instruction is a single word long and there is no need for either caching or prefetching at this stage. We can therefore omit the low-level analysis. No pipeline analysis [42], with its possible unbounded timing effects, is necessary.

6.1.3 Bytecode Independency

We have seen that all microcode instructions except wait take one cycle to execute and are therefore independent of other instructions. This property directly translates to independency of bytecode instructions.

[3]The implementation can be found in the class com.jopdesign.sys.JVM.

[4]lookupswitch is one way of implementing the Java switch statement. The other bytecode, tableswitch, uses an index in the table of branch offsets and has therefore a constant execution time.

The wait microcode instruction provides a convenient way to hide memory access time. A memory read or write can be triggered in microcode and the processor can continue with microcode instructions. When the data from a memory read is needed, the processor explicitly waits, with the wait instruction, until it becomes available.

For a memory store, this wait could be deferred until the memory system is used next (similar to a write buffer). It is possible to initiate the store in a bytecode such as putfield and continue with the execution of the next bytecode, even when the store has not been completed. In this case, we introduce a dependency over bytecode boundaries, as the state of the memory system is *shared*. To avoid these dependencies that are difficult to analyze, each bytecode that accesses memory waits (preferably at the end of the microcode sequence) for the completion of the memory request.

Furthermore, if we would not wait at the end of the store operation we would have to insert an additional wait at the start of every read operation. Since read operations are more frequent than write operations (15% vs. 2.5%, see [123]), the performance gain from the hidden memory store is lost.

6.1.4 WCET of Bytecodes

The control flow of the individual bytecodes together with the basic block length (that directly corresponds with the execution time) and the time for memory access result in the WCET (and BCET) values of the bytecodes. These values can be found in Appendix D.

Basic Bytecodes

Most bytecode instructions that do not access memory have a constant execution time. They are executed by either one microcode instruction or a short sequence of microcode instructions. The execution time in clock cycles equals the number of microinstructions executed. As the stack is on-chip, it can be accessed in a single cycle. We do not need to incorporate the main memory timing into the instruction timing. Most simple bytecodes execute in a single cycle. Table 6.1 shows example instructions, their timing, and their meaning. Access to object, array, and class fields depend on the timing of the main memory.

Object Oriented Bytecodes

Object oriented instructions, array access, and invoke instructions access the main memory. Therefore, we have to model the memory access time. We assume a simple SRAM with a constant access time. Access time that exceeds a single cycle includes additional wait states

Opcode	Instruction	Cycles	Funtion
3	iconst_0	1	load constant 0 on TOS
4	iconst_1	1	load constant 1 on TOS
16	bipush	2	load a byte constant on TOS
17	sipush	3	load a short constant on TOS
21	iload	2	load a local on TOS
26	iload_0	1	load local 0 on TOS
27	iload_1	1	load local 1 on TOS
54	istore	2	store the TOS in a local
59	istore_0	1	store the TOS in local 0
60	istore_1	1	store the TOS in local 1
89	dup	1	duplicate TOS
90	dup_x1	5	complex stack manipulation
96	iadd	1	integer addition
153	ifeq	4	conditional branch

Table 6.1: Execution time of simple bytecodes in cycles

(r_{ws} for a memory read and w_{ws} for a memory write). With a memory with r_{ws} wait states, the execution time for, e.g., getfield is

$$t_{getfield} = 11 + 2r_{ws}$$

The memory subsystem performs reads and writes in parallel to the execution of microcode. Therefore, some access cycles can be hidden. The following example gives the exact execution time of bytecode ldc2_w in clock cycles:

$$t_{ldc2\_w} = 17 + \begin{cases} r_{ws} - 2 & : \quad r_{ws} > 2 \\ 0 & : \quad r_{ws} \leq 2 \end{cases} + \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

Thus, for a memory with two cycles access time ($r_{ws} = 1$), as we use it for a 100 MHz version of JOP with a 15 ns SRAM, the wait state is completely hidden by microcode instructions for this bytecode.

Memory access time also determines the cache load time on a miss. For the current implementation the cache load time is calculated as follows: the wait state c_{ws} for a single word cache load is:

$$c_{ws} = \begin{cases} r_{ws} & : \quad r_{ws} > 1 \\ 1 & : \quad r_{ws} \leq 1 \end{cases}$$

On a method invoke or return, the respective method has to be loaded into the cache on a cache miss. The load time l is:

$$l = \begin{cases} 6+(n+1)(1+c_{ws}) & : \text{cache miss} \\ 4 & : \text{cache hit} \end{cases}$$

where n is the size of the method in number of 32-bit words. For short methods, the load time of the method on a cache miss, or part of it, is hidden by microcode execution. As an example, the exact execution time for the bytecode invokestatic is:

$$t = 74 + r_{ws} + \begin{cases} r_{ws}-3 & : r_{ws} > 3 \\ 0 & : r_{ws} \le 3 \end{cases} + \begin{cases} r_{ws}-2 & : r_{ws} > 2 \\ 4 & : r_{ws} \le 2 \end{cases} + \begin{cases} l-37 & : l > 37 \\ 0 & : l \le 37 \end{cases}$$

For invokestatic a cache load time l of up to 37 cycles is completely hidden. For the example SRAM timing the cache load of methods up to 36 bytes long is hidden. The WCET analysis tool, as described in the next section, knows the length of the invoked method and can therefore calculate the time for the invoke instruction cycle accurate.

Bytecodes in Java

Bytecodes can be implemented in Java on JOP. In this case, a static method from a JVM internal class gets invoked when such a bytecode is executed. For WCET analysis this bytecode is substituted by an invoke instruction to this method. The influence on the cache (the bytecode execution results in a method load) can be analyzed in the same way as for *ordinary* static methods.

Native Methods

Most of the JVM internal functionality in JOP, such as input, output, and thread scheduling, is implemented in Java. However, Java and the JVM do not allow direct access to memory, peripheral devices or processor registers. For this low-level access to system resources, we need *native* methods. For a Java processor, the *native* language is still Java bytecode. We solve this issue by substituting the native method invocation by a special bytecode instruction on class loading. Those special bytecodes are implemented in JOP microcode in the same way as regular bytecodes (see Section 4.2. The execution time of the native methods (or in other words special bytecodes) is given in the same way as the execution time for standard bytecodes.

6.2 WCET Analysis of the Java Application

In hard real-time systems, the estimation of the worst-case execution time (WCET) is essential. WCET analysis is in general an undecidable problem. Program restrictions, as given in [109], make this problem decidable:

1. Programs must not contain any recursion

2. Absence of function pointers

3. The upper bound of each loop has to be known

Recursive algorithms have to be transformed to iterative ones. For our WCET analyzer, the loop bounds are detected by data-flow analysis or have to be annotated in the program source. However, we want to relax the second restriction regarding function pointers. Function pointers are similar to inherited or overridden methods: they are dispatched at runtime. For an object-oriented language this mechanism is fundamental. In contrast to function pointers, e.g., in C, we can statically analyze which methods can be invoked when the whole program is known. Therefore, we allow dynamic dispatching of methods in Java in our analysis.

We replace the function pointer restriction by the following restriction: *Dynamic class loading is not allowed*. As the full application has to be available for WCET analysis, dynamic class loading cannot be supported. For embedded real-time systems this is not a severe restriction.

6.2.1 High-Level WCET Analysis

The high-level WCET analysis, presented in this section, is based on standard technologies [110, 79]. From the class files that make up the application, the relevant information is extracted. The CFG of the basic blocks is extracted from the bytecodes. Annotations for the loop counts are either provided by the data-flow analysis or are extracted from comments in the source. Furthermore, the class hierarchy is examined to find all possible targets for a method invoke and the data-flow analysis tightens the set of possible receivers.

Java bytecode generation has to follow stringent rules [82] in order to pass the class file verification of the JVM. Those restrictions lead to an *analysis friendly* code; e.g. the stack size is known at each instruction. The control flow instructions are well defined. Branches are relative and the destination is within the same method. In the normal program, there is no instruction that loads a branch destination in a local variable or onto the stack.[5] Detection of basic blocks in Java bytecode and construction of the CFG is thus straight forward.

In Java class files there is more information available than in compiled C/C++ executables. All links are symbolic and it is possible to reconstruct the class hierarchy from the class files. Therefore, we can statically determine all possible targets for a virtual method invoke.

WCET analysis at the bytecode level has several advantages when the execution time of the individual bytecodes are known. When compiled with debug information references to the source are stored in the class file. With this information it is possible to extract annotations in the Java source code. We use this feature to annotated loop bounds within comments in the Java source. This form of annotation is simple and less intrusive than using a predefined dummy class as suggested in [23].

6.2.2 WCET Annotations

Simple loop bounds are detected by the data-flow analysis [104]. For cases where the bounds are not detectable, we additionally support annotations in the source. The annotations are written as comments (see Listing 6.1). It would be more convenient to use standard Java annotations introduced with Java 1.5, as they are checked at compile time. However, at the moment Java annotations are not allowed at code block level. A proposal to remove this restriction is currently under review (JSR 308). When compiling Java, the source line information is included in the class file. Therefore, when a loop is detected in the CFG, the relevant source line for the loop header is looked up in the source and the annotation is extracted from the comment.

Annotations given as source comments are simple and less intrusive than using a predefined dummy class [23]. Two variants of the loop bounding annotation are supported: one with an exact bound[6] (=) and one that places an upper bound on the iterations (<=). The extension to more elaborate annotations, as suggested in [109] and [23], can provide even tighter WCET bounds.

[5]The exception are bytecodes jsr and ret that use the stack and a local variable for the return address of a method local subroutine. This construct can be used to implement the finally clauses of the Java programming language. However, this problematic subroutine can be easily inlined [12]. Furthermore, Sun's Java compilers version 1.4.2 and later compile finally blocks without subroutines.

[6]The exact bound has been used to find best-case values for some test settings.

6.2.3 ILP Formulation

The calculation of the WCET is transformed to an ILP problem [110]. In the CFG, each vertex represents a basic block B_i with execution time c_i. With the basic block execution frequency e_i the WCET is:

$$WCET = max \sum_{i=1}^{N} c_i e_i$$

The sum is the objective function for the ILP problem. The maximum value of this expression results in the WCET of the program.

Furthermore, each edge is also assigned an execution frequency f. These execution frequencies represent the control flow through the WCET path. Two primary constraints form the ILP problem: (i) For each vertex, the sum of f_j for the incoming edges has to be equal to the sum of the f_k of the outgoing edges; (ii) The frequency of the edges connecting the loop body with the loop header, is less than or equal to the frequency of the edges entering the loop multiplied by the loop bound.

From the CFG, which represents the program structure, we can extract the flow constraints. With the execution frequency f of the edges and the two sets I_i for the incoming edges to basic block B_i and O_i for the outgoing edges, the execution frequency e_i of B_i is:

$$e_i = \sum_{j \in I_i} f_j = \sum_{k \in O_i} f_k$$

The frequencies f are the integer variables calculated by solving the ILP problem. Furthermore, we add two special vertices to the graph: The start node S and the termination node T. The start node S has only one outgoing edge that points to the first basic block of the program. The execution frequency f_s of this edge is set to 1. The termination node T has only incoming edges with the sum of the frequencies also set to 1; all return statements of a method are connected to the node T. This means that the program is executed once and can only terminate once through T.

Loop bounds are additional constraints for the ILP problem. A special vertex, the loop header, is connected by following edges:

1. Incoming edges that enter the loop with frequency f_h
2. One outgoing edge entering the loop body with frequency f_l
3. Incoming edges that close the loop
4. One loop exit edge

```
public static int loop(boolean b, int val) {

    int i, j;

    for (i=0; i<10; ++i) {          // @WCA loop=10
        if (b) {
            for (j=0; j<3; ++j) {   // @WCA loop=3
                val *= val;
            }
        } else {
            for (j=0; j<4; ++j) {   // @WCA loop=4
                val += val;
            }
        }
    }
    return val;
}
```

Listing 6.1: The example used for WCET analysis

With the maximum loop count (the loop bound) n we formulate the loop constraint as

$$f_l \leq n \sum f_h$$

This explanation is a little bit simplified, as more complex loop conditions have several edges to the loop body. Therefore, the tool considers the set of incoming edges to the loop header that close the loop for the ILP constraint.

Without further global constraints the problem can be solved locally for each method. We start at the leaves of the call tree and calculate the WCET for each method. The WCET value of a method is included in the invoke instruction of the caller method. To incorporate global constraints, such as cache constraints [80], a single CFG is built that contains the whole program by inserting edges from the invoke instruction to the invoked method and back. This is conceptually equivalent to inlining each method.

In Section 6.2.6, we will show how the cache constraints for the method cache can be integrated into the analysis.

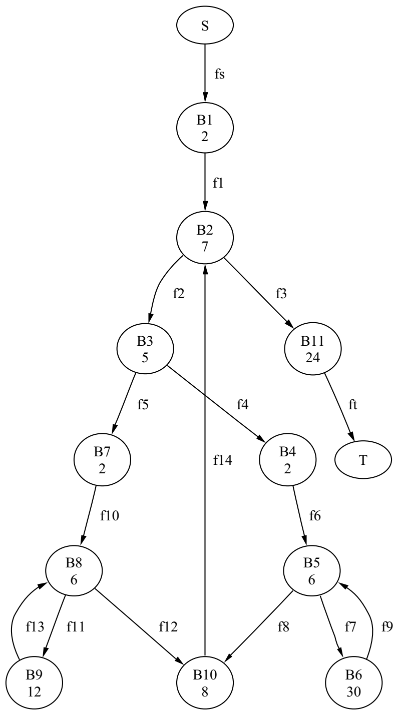


Figure 6.1: CFG of the example

6.2.4 An Example

To illustrate the WCET analysis flow we provide a small example. Listing 6.1 shows the Java source code that contains nested loops with a condition. The simple loop bounds are detected by the data-flow analysis (DFA). We just show in the example how the annotation syntax looks like. In our target hardware, the multiplication takes longer than the addition. Therefore, in this example, it is not obvious which branch will result in the WCET path.

Table 6.2 shows the bytecodes and basic blocks of the example, as generated by our WCET analysis tool. The fourth column gives the execution time of the bytecodes in clock cycles. The fifth column gives the execution time of the basic blocks. These are the values used for the ILP equations.

From the basic blocks, we can construct the CFG as shown in Figure 6.1. The vertices represent the basic blocks and include the execution time in clock cycles. We can identify block B2 as the loop header for the outer loop. B3 is the branch node. B5 and B8 are the

Block	Addr.	Bytecode	Cycles	BB Cycles
B1	0:	iconst_0	1	
	1:	istore_2	1	2
B2	2:	iload_2	1	
	3:	bipush	2	
	5:	if_icmpge 55	4	7
B3	8:	iload_0	1	
	9:	ifeq 30	4	5
B4	12:	iconst_0	1	
	13:	istore_3	1	2
B5	14:	iload_3	1	
	15:	iconst_3	1	
	16:	if_icmpge 48	4	6
B6	19:	iload_1	1	
	20:	iload_1	1	
	21:	imul	19	
	22:	istore_1	1	
	23:	iload_3	1	
	24:	iconst_1	1	
	25:	iadd	1	
	26:	istore_3	1	
	27:	goto 14	4	30
B7	30:	iconst_0	1	
	31:	istore_3	1	2
B8	32:	iload_3	1	
	33:	iconst_4	1	
	34:	if_icmpge 48	4	6
B9	37:	iload_1	1	
	38:	iload_1	1	
	39:	iadd	1	
	40:	istore_1	1	
	41:	iload_3	1	
	42:	iconst_1	1	
	43:	iadd	1	
	44:	istore_3	1	
	45:	goto 32	4	12
B10	48:	iload_2	1	
	49:	iconst_1	1	
	50:	iadd	1	
	51:	istore_2	1	
	52:	goto 2	4	8
B11	55:	iload_1	1	
	56:	ireturn	23	24

Table 6.2: Java bytecode and basic blocks

loop headers for the inner loops.

From the CFG, we can extract the flow constraints by the following fact: The execution frequency of a basic block is equal to the execution frequency of all incoming edges and equal to the execution frequency of all outgoing edges. E.g., for block B2 the execution frequency e_2 is:

$$e_2 = f_1 + f_{14} = f_2 + f_3$$

The loop constraints are formulated by constraining the frequency of the loop's back edges. The loop bounds are automatically transferred from the DFA module or extracted from the source annotations. For the outer loop in the example this is:

$$f_{14} = 10 f_1$$

In Listing 6.2, the resulting equations for the integer linear programming problem, as generated by our tool, are listed. We use the open-source ILP solver lp_solve.[7]

The ILP solver lp_solve gives a result of 1393 cycles. We run this example on the Java processor for verification. As the execution time depends only on a single boolean variable, b (see Listing 6.1), it is trivial to measure the actual WCET. We measure the execution time with a cycle counter for the execution time of the outer loop, i.e., from the start of block B1 until the exit of the loop at block B2. The last block, B11, which contains the return statement, is not part of the measurement. The measured result is 1369 cycles. When we add the 24 cycles for the block B11 to our measured WCET, we get 1393 cycles. This measured result is exactly the same as the analytical result!

6.2.5 Dynamic Method Dispatch

Dynamic runtime-dispatching of inherited or overridden instance methods is a key feature of object-oriented programming. Therefore, we allow dynamic methods as a *controlled* form of function pointers. The full class hierarchy can be extracted from the class files of the application. From the class hierarchy, we can extract all possible receiver methods for an invocation. When all possible receivers are included as alternatives, the resulting WCET bound can be very pessimistic. Therefore, this set can be tightened by a flow-sensitive receiver analysis [104].

We include all possible receivers as alternatives in the ILP constraints. It has to be noted that, without further analysis or annotations, this can lead to pessimistic WCET estimates.

[7] http://lpsolve.sourceforge.net/5.5/

```
/* Objective function */
max: t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11;
/* flow constraints */
S:  fs = 1;
B1: fs = f1;
B2: f14 + f1 = f2 + f3;
B3: f2 = f4 + f5;
B4: f4 = f6;
B5: f9 + f6 = f7 + f8;
B6: f7 = f9;
B7: f5 = f10;
B8: f10 + f13 = f11 + f12;
B9: f11 = f13;
B10: f12 + f8 = f14;
B11: f3 = ft;
T:  ft = 1;
/* loop bounds */
f14 = 10 f1;
f9 = 3 f6;
f13 = 4 f10;
/* execution time (with incoming edges) */
t1 = 2 fs;
t2 = 7 f14 + 7 f1;
t3 = 5 f2;
t4 = 2 f4;
t5 = 6 f9 + 6 f6;
t6 = 30 f7;
t7 = 2 f5;
t8 = 6 f10 + 6 f13;
t9 = 12 f11;
t10 = 8 f12 + 8 f8;
t11 = 24 f3;
```

Listing 6.2: ILP equations of the example

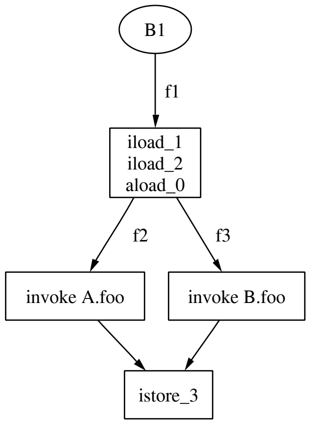


Figure 6.2: Split of the basic block for instance methods

We split the basic block that contains the invoke instruction into three new blocks: The preceding instructions, the invoke instruction, and following instructions. Consider following basic block:

```
iload_1
iload_2
aload_0
invokevirtual foo
istore_3
```

When different versions of foo() are possible receiver methods, we model the invocation of foo() as alternatives in the graph. The example for two classes A and B that are part of the same hierarchy is shown in Figure 6.2. Following the standard rules for the incoming and outgoing edges the resulting ILP constraint for this example is:

$$f_1 = f_2 + f_3$$

6.2.6 Cache Analysis

From the properties of the Java language — usually small methods and relative branches — we derived the novel idea of a *method cache* [120], i.e., an instruction cache organization in which whole methods are loaded into the cache on method invocation and on the return from a method.

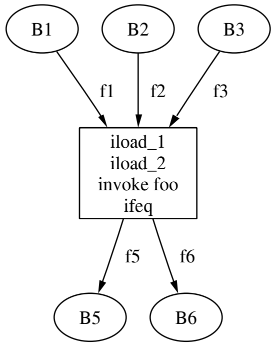


Figure 6.3: Basic block with an invoke instruction

The method cache is designed to simplify WCET analysis. Due to the fact that all cache misses are only included in two instructions (*invoke* and *return*), the instruction cache can be ignored on all other instructions. The time needed to load a complete method is calculated using the memory properties (latency and bandwidth) and the size of the method. On an invoke, the size of the invoked method is used, and on a return, the method size of the caller.

Integration of the method cache into WCET analysis is straight forward. As the cache hits or misses can only happen at method invocation, or return from a method, we can model the miss times as extra vertices in the graph. Figure 6.3 shows an example with 6 connected basic blocks. Basic block B4 is shown as a box and has three incoming edges (f_1,f_2,f_3) and two outgoing edges (f_5,f_6). B4 contains the invocation of method foo(), surrounded by other instructions. The execution frequency e_4 of block B4 in the example is

$$e_4 = f_1 + f_2 + f_3 = f_5 + f_6$$

We split a basic block that contains a method invoke (B4 in our example) into several blocks, so one block contains only the invoke instruction. Misses on invoke and return are modeled as extra blocks with the miss penalty as execution time.

The miss for the return happens during the return instruction. On a miss, the caller method has to be loaded into the cache. Therefore, the miss penalty depends on the caller method size. However, as the return instruction is the last instruction executed in the called method, we can model the return miss time at the caller side after the invoke instruction. This approach simplifies the analysis as both methods, the caller and the called, with their respective sizes, are known at the occurrence of the invoke instruction.

Figure 6.4 shows the resulting graph after the split of block B4 and the inserted vertices for the cache misses. The miss penalty is handled in the same way as execution time of

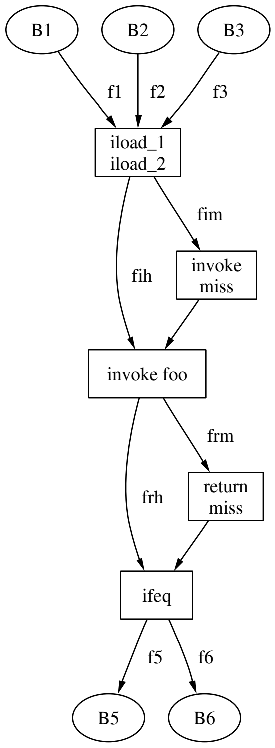


Figure 6.4: Split of the basic block and cache miss blocks

basic blocks for the ILP objective function. The additional constraints for the control flow in our example are

$$e_4 = f_{ih} + f_{im}$$

$$e_4 = f_{rh} + f_{rm}$$

with the invocation hit and miss frequencies f_{ih} and f_{im} and the return hit and miss frequencies f_{rh} and f_{rm}.

It has to be noted that misses are always more expensive than hits. A conservative bound on the hit frequency is a safe approximation when the exact information is missing. As the hit or miss time is contained within a single bytecode execution, there are no issues with timing anomalies [84].

As a next step, we have to formulate the relation between the hit and the miss frequency. In [120], several variants of the method cache are described:

1. A single block that can only cache a single method
2. Several blocks that each can cache a single method
3. A variable block cache where a method can span several blocks

The first two variants are simple to integrate into WCET analysis, but lead to high WCET bounds. In the following, the analysis of the third variant is described, details on analysis of the first two variants can be found in [134].

The *variable block cache* divides the cache in several blocks similar to cache lines in a conventional cache. However, a single method has to be loaded in a continuous region of the cache. The effective replacement policy of the variable block cache is FIFO. FIFO caches need a long access history for standard classification techniques [112]. Due to the FIFO replacement, one can construct examples where a method in a loop can be classified as single miss, but that miss does not need to happen at the first iteration or it can happen on a return from an invoked method.

WCET analysis of cache hits for the method cache is most beneficial for methods invoked in a loop, where the methods are classified as first miss. The basic idea of the method cache analysis is as follows: Within a loop it is statically analyzed if all methods invoked and the invoking method, which contains the loop, fit together in the method cache. If this is the case, all methods miss at most once in the loop. The concrete implementation of the analysis algorithm is a little bit different.

However, the miss of a method that is not a leaf method can happen either at invoke of this method or on the return (from a deeper nested method) to this method. On JOP some of the cache load cycles can be hidden due to the concurrent execution of microcode. At the more complex invoke instruction more cash load time is hidden. Therefore, the miss penalty on a return is higher. All methods from the fitting set that are not leafs need to consider the miss penalty of a return bytecode. Leaf nodes can naturally only miss on an invoke.

With the variable block cache, it could be argued that WCET analysis becomes too complex, but the technique presented above is easy to apply and only needs to perform a simple static analysis. In contrast to a direct-mapped instruction cache, we only need to consider invoke instructions and do not need to know the actual bytecode address of any instruction.

6.2.7 WCET Analyzer

The WCET analyzer (WCA) is an open source Java program first described in [134] and later redesigned [66]. It is based on the standard IPET approach [110], as described in Section 6.2.3. Hereby, WCET is computed by solving a maximum cost circulation problem in a directed graph representing the program's control flow. In contrast to the former description, WCA associates the execution cost to the edges instead of the vertices. For modeling the execution time of JOP those two approaches are equivalent. However, WCA is prepared for other architectures with different execution times of a branch taken or not taken.

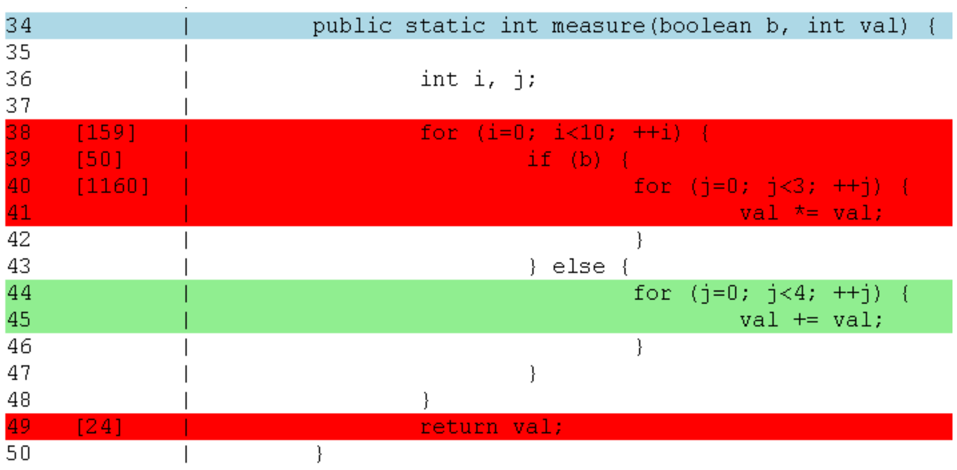
The tool performs the low-level analysis at the bytecode level. The behavior of the method cache is integrated with the static all-fit approximation (see Section 6.2.6). The well known execution times of the different bytecodes (see Section 6.1.4) simplify this part of the WCET analysis, which is usually the most complex one, to a great extent. As there are no pipeline dependencies, the calculation of the execution time for a basic block is merely just adding the individual cycles for each instruction.

To access the class files, we use the Byte Code Engineering Library (BCEL) [34]. BCEL allows inspection and manipulation of class files and the bytecodes of the methods. The WCA extracts the basic blocks from the methods and builds the CFG. Within the CFG, the WCA detects both the loops and the loop head. From the source line attribute of the loop head, the annotation of the loop count is extracted. WCA uses the open-source ILP solver lp_solve. lp_solve is integrated into the WCA by directly invoking it via the Java library binding.

After completing WCET analysis, WCA creates detailed reports to provide feedback to the user and annotate the source code as far as possible. All reports are formatted as HTML pages. Each individual method is listed with basic blocks and execution time of bytecodes, basic blocks, and cache miss times. This output is similar to Table 6.2, but with more detailed information. The WCA also generates a graphical representation of the CFG for each method and for the whole program. Furthermore, the source is annotated with execution time and the WCET path is marked, as shown in Figure 6.5 for the example from Listing 6.1. For this purpose, the solution of the ILP is analyzed, first annotating the nodes of the CFG with execution costs, and then mapping the costs back to the source code.

The Makefile contains the target wcet for the WCET tool. The following example performs WCET analysis of the method measure of the crc example and uses DFA for the loop bound detection. The method that is the root of the application to be analyzed can be set in variable WCET_METHOD.

```
make java_app wcet -e P1=test P2=wcet P3=ShortCrc WCET_DFA=yes
```



```
34         |         public static int measure(boolean b, int val) {
35         |
36         |                 int i, j;
37         |
38  [159]  |                 for (i=0; i<10; ++i) {
39  [50]   |                         if (b) {
40  [1160] |                                 for (j=0; j<3; ++j) {
41         |                                         val *= val;
42         |                                 }
43         |                         } else {
44         |                                 for (j=0; j<4; ++j) {
45         |                                         val += val;
46         |                                 }
47         |                         }
48         |                 }
49  [24]   |                 return val;
50         |         }
```

Figure 6.5: WCET path highlighting in the source code.

6.3 Evaluation

For the evaluation of the WCET tool, we analyze and measure a collection of real-time benchmarks. The measurement gives us confidence that we have no serious bugs in the analysis and an idea of the pessimism of the analyzed WCET. It has to be noted that we actually cannot guarantee to measure the real WCET. If we could measure the WCET, we would not need to perform WCET analysis at all.

Furthermore, WCET analysis gives a safe bound, but this bound may be conservative. Due to the abstractions in the analysis the WCET bound may not be associated with a real feasible execution path. There is no general guarantee that we have knowledge about the worst case path.

6.3.1 Benchmarks

The benchmarks used are shown in Table 6.3, with a short description and the source code size in lines of code (LOC). The benchmarks consists of three groups: small kernel benchmarks, WCET benchmarks provided by the Mälardalen Real-Time Research Center, [8] and three real-world applications.

The benchmark crc calculates a 16-bit CRC for short messages. LineFollower is the code of

[8]Freely available from `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

Benchmark	Program	Description	LOC
Kernel	crc	CRC calculation for short messages	8
	LineFollower	A simple line follower robot	89
	SVM	Embedded machine learning algorithm	115
Mälardalen	BinarySearch	Binary search program	78
	Bubble	Bubblesort program	63
	Crc	Cyclic redundancy check	154
	ExpInt	Exponential integral function	89
	FDCT	Fast discrete cosine transform	222
	Fibonacci	Simple iterative Fibonacci calculation	37
	InsertionSort	Insertion sort program	60
	JanneComplex	Complex nested loops	72
	MatrixCount	Count numbers in a matrix	85
	MatrixMult	Matrix multiplication	104
	NestedSearch	Search in a multi-dimensional array	487
	QuickSort	Quicksort program	166
	Select	Select smallest element from array	136
	SLE	Simultaneous linear equations	128
Applications	Lift	Lift controller	635
	Kfl	*Kippfahrleitung* application	1366
	EjipCmp	Multicore UDP/IP benchmark	1892

Table 6.3: WCET benchmark examples

a simple line-following robot. The SVM benchmark represents an embedded version of the machine learning algorithm called support vector machine (SVM). The SVM is modified minimally to work in the real-time setting [93]. The second group are benchmarks provided by the Mälardalen Real-Time Research Center [85] and ported to Java by Trevor Harmon [58].

The benchmarks Lift and Kfl are real-world examples that are in industrial use. Lift is a simple lift controller, where Kfl is a node in a distributed mast control application. The EjipCmp benchmark is an embedded TCP/IP stack written in Java with an example application consisting of a UDP/IP server and a UDP/IP client, exchanging messages via a loop-back network layer driver. The TCP/IP stack and the example applications are optimized for a chip-multiprocessor system (e.g., with non-blocking communication). In our setup, however, we execute all five tasks serially on a uniprocessor and perform WCET analysis

Program	Measured (cycles)	Estimated (cycles)	Pessimism (ratio)
crc	1449	1513	1.04
LineFollower	2348	2411	1.03
SVM	104880	107009	1.02
BinarySearch	631	636	1.01
Bubble	1262717	1815734	1.44
Crc	191825	383026	2.00
ExpInt	324419	429954	1.33
FDCT	19124	19131	1.00
Fibonacci	1127	1138	1.01
InsertionSort	8927	15410	1.73
JanneComplex	886	6991	7.89
MatrixCount	16420	16423	1.00
MatrixMult	1088497	1088509	1.00
NestedSearch	51777	64687	1.25
QuickSort	9942	115383	11.61
Select	5362	55894	10.42
SLE	20408	50514	2.48
Lift	5446	8466	1.55
Kfl	10411	39555	3.80
EjipCmp	15297	21965	1.44

Table 6.4: Measured and estimated WCET with result in clock cycles

for the aggregation of the five tasks.

The configuration of JOP for the evaluation is with a 4 KB method cache configured for 16 blocks. The memory access times are 2 cycles for a read and 3 cycles for a write.

6.3.2 Analysis and Measurements

Table 6.4 shows the measured execution time and the analyzed WCET. The last column gives an idea of the pessimism of the WCET analysis. It has to be noted (again) that we actually cannot guarantee to measure the real WCET. If we could measure the WCET, we would not need to perform WCET analysis at all. The WCET analysis result of a method is defined as follows: WCA reports the WCET of the executable code within a method including the return statement. The invoke of the to-be-analyzed method is not part of the analysis. With respect to cache analysis the return from the method does not include any miss time – invoke and return miss time are considered as part of the invoke instruction. For the measurement we use a cycle counter and take a time stamp before the first instruction of the method and after the return from the method.

For very simple programs, such as crc, robot, and SVM the pessimism is quite low. The same is true for several of the benchmarks from the Mälardalen benchmark suit. Some are almost cycle accurate. The small overestimation of, e.g., 5 cycles for BinarySearch, results from our measurement methodology and the definition of what the WCET of a single method is. To avoid instrumenting all benchmarks with time stamp code we wrap the benchmarks in a method measure(), vary which benchmark is invoked in measure(), and analyze this method. Therefore, measure() contains an invoke and the control is return to measure(). In the case of small benchmarks the method cache still contains measure() on the return from the benchmark method. However, WCA does not include measure() as a possible method for the static method cache analysis (it is performed on a invoke and no invoke of measure() is seen by WCA). Therefore, the return from the benchmark method is classified as miss.

The pessimism of most of the other benchmarks, including the two applications Lift and EjipCmp, are in an acceptable range below a factor of 2. Three benchmarks, JanneComplex, QuickSort, and Select, stand out with a high pessimism. JanneComplex is an artificial benchmark with a complex loop dependency of an inner loop and the annotation results in a conservative estimation. However, it has to be noted that the benchmarks from Mälardalen are designed to challenge WCET tools. We assume that a safety-critical programmer would not use QuickSort as a first choice for time-predictable code.

The pessimism of Kfl is quite high. The difference between the measurement, and the analysis in the Kfl example, results from the fact that our measurement does not cover the WCET path. We only simulate input values and commands for the mission phase. However, the main loop of Kfl also handles service functions. Those functions are not part of the mission phase, but make up the WCET path. If we omit the non-feasible path to the service routine the WCET drops down to 22875 and the overestimation is 2.2.

The Kfl example show the issue when a real-time application is developed without a WCET analysis tool available. Getting the feedback from the analysis earlier in the design phase can help the programmer to adapt to a WCET aware programming style. In one extreme, this can end up in the single-path programming style [108]. A less radical approach can use some heuristics for a WCET aware programming style. For instance, avoiding the service routines in the application main loop.

The results given in Table 6.4 are comparable to the results presented in [134]. Most benchmarks now take less cycles than in 2006 as the performance of JOP has been advanced. The exceptions are two benchmarks, LineFollower and Kfl, which now have a higher WCET. The line-following robot example was rewritten in a more object-oriented way. Finally, the different number for the Kfl resulted from a bug in the WCA that was corrected after the publication in 2006.

6.4 Discussion

We found that the approach of a time-predictable processor and static WCET analysis is a feasible solution for future safety-critical systems. However, there are some remaining questions that are discussed in the following sections.

6.4.1 On Correctness of WCET Analysis

Safe WCET approximation depends on the correct modeling of the target architecture and the correct implementation of the analysis tools. During the course of this paper we have found a bug in the WCA from 2006 and have also found bugs in the low-level timing description of JOP. However, the open-source approach for the processor and the tools increases the chance that bugs are found. Furthermore, we have compared the results of the new implementation of WCA with two independent implementations of WCET analysis for JOP: The Volta project by Trevor Harmon [57] and the former implementation of WCA [134]. The results differ only slightly due to different versions of JOP that Volta, used and a less accurate cache analysis implemented in the second tool. It should be noted that JOP and the WCA are still considered on-going research prototypes for real-time systems. We would not (yet) suggest to deploy JOP in the next safety-critical application.

6.4.2 Is JOP the Only Target Architecture?

JOP is the primary low-level target for the WCET analysis tool as it is: (a) a simple processor, (b) open-source, and (c) the execution timing is well-documented (see Appendix D). Furthermore, the JOP design is actually the root of a family of Java processors. Flavius Gruian has built a JOP compatible processor, with a different pipeline organization, with Bluespec Verilog [50]. The SHAP Java processor [157], although now with a different pipeline structure and hardware assisted garbage collection, also has its roots in the JOP design. We assume that these processors are also WCET analysis *friendly*. The question remains how easy the analysis can be adapted for other Java processors.

The same analysis is not possible for other Java processors. Either the information on the bytecode execution time is missing[9] or some processor features (e.g., the high variability of the latency for a trap in picoJava) would result in very conservative WCET estimates. Another example that prohibits exact analysis is the mechanism to automatically fill and spill the stack cache in picoJava [145, 146]. The time when the memory (cache) is occupied by this spill/fill action depends on a long instruction history. Also the fill level of the 16-byte-deep prefetch buffer, which is needed for instruction folding, depends on the execution history. All these automatic buffering features have to be modeled quite conservatively. A pragmatic solution is to assume empty buffers at the start of a basic block. As basic blocks are quite short, most of the buffering/prefetching does not help to lower the WCET.

Only for the Cjip processor is the execution time well documented [69]. However, the *measured* execution time of some bytecodes is *higher* than the documented values [123]. Therefore the documentation is not complete to provide a safe processor model of the Cjip for WCET analysis.

The real-time Java processor jamuth [149] is a possible target for the WCA. There is on-going work with Sascha Uhrig to integrate the timing model of jamuth into the WCA. A preliminary version of that model has been tested. The analysis tool need to be changed to model a few pipeline dependencies.

Still, WCET analysis of Java programs on RISC processors with a compiling JVM is a challenge. With a compiling JVM, the WCET friendly bytecodes are further compiled into machine code. Information is lost in the second transformation. This is not the case with a dedicated Java processor such as JOP.

6.4.3 Object-oriented Evaluation Examples

Most of the benchmarks used are written in a rather conservative programming style. The Mälardalen benchmarks are small kernels that do not stress the cache analysis. We would like to use more object-oriented real-time examples for the evaluation. Then, we can evaluate if object-oriented programming style with dynamic method dispatch and rather small methods is a feasible option for future real-time systems.

6.4.4 WCET Analysis for Chip-multiprocessors

Chip-multiprocessors (CMP) are now considered as the future technology for high-performance embedded systems. Christof Pitter has built a CMP version of JOP that includes a time-predictable memory arbiter [97]. During the course of his thesis he has

[9]We tried hard to get this information for the aJile processor.

adapted the low-level timing calculation of the WCET tool to integrate the memory access times with the time-sliced arbiter. Therefore, it is possible to derive WCET values for tasks running on a CMP system with shared main memory. To best of our knowledge, this is the first time-predictable CMP system where WCET analysis is available.

The first estimation of bytecode timings for the CMP system considers only individual bytecodes. However, with variable memory access times due to memory arbitration this estimate is conservative. An extension to basic block analysis and using model checking to analyze possible interactions between the memory arbitration process and the program execution should result in tighter WCET estimates.

The JOP CMP system showed the importance of tighter method cache analysis. Compared to the simple approach of the original WCA tool, the new static cache approximation yielded in 15% tighter WCET estimates for a 8 core CMP system.

6.4.5 Co-Development of Processor Architecture and WCET Analysis

JOP is an example of a time-predictable computer architecture. An extension of this approach to RISC and VLIW based processors is presented in [132]. We consider a co-development of time-predictable computer architecture and WCET analysis for the architectural features as an ideal approach for future real-time systems. A new time-predictable feature (e.g., a data cache organization) can only be evaluated when the analysis is available. The analysis, and what can be analyzed within practical execution time and memory demands, shall guide the computer architect of future time-predictable processors.

6.4.6 Further Paths to Explore

The combination of a bytecode based optimizer (a prototype is available as part of the JOP sources) with WCET analysis can be applied to minimize the WCET path. The optimizer can be guided by the results from the application. This iterative flow can be used to minimize the worst-case path instead of the average-case path in the application.

6.5 Summary

In this chapter, the WCET analysis tool based on the ILP approach has been presented. The architecture of the Java processor greatly simplifies the low-level part of WCET analysis, as it can be performed at the Java bytecode level. An instruction cache, named *method cache*, stores complete methods, and is integrated into the WCET analysis tool.

The WCET analysis tool, with the help of DFA detected loop bounds, provides WCET values for the schedulability analysis. Complex loops still need to be annotated in the source. A receiver type analysis helps to tighten the WCET of object-oriented programs. We have also integrated the method cache into the analysis. The cache can be analyzed at the method level and does not need the full program CFG. Besides the calculation of the WCET, the tool provides user feedback by generating bytecode listings with timing information and a graphical representation of the CFG with timing and frequency information. This representation of the WCET path through the code can guide the developer to write WCET aware real-time code.

The experiments with the WCET analyzer tool have demonstrated that we have achieved our goal: JOP is a simple target for WCET analysis. Most bytecodes have a single execution time (WCET = BCET), and the WCET of a task (the analysis at the bytecode level) depends only on the control flow. No pipeline or data dependencies complicate the low-level part of WCET analysis.

6.6 Further Reading

There are a lot of papers available on WCET analysis and modeling of low-level processor features. This section gives an overview of the related work for our WCET analyzer.

6.6.1 WCET Analysis

Shaw presents in [139] timing schemas to calculate minimum and maximum execution time for common language constructs. The rules allow to collapse the CFG of a program until a final single value represents the WCET. However, with this approach it is not straight forward to incorporate global low-level attributes, such as pipelines or caches. The resulting bounds are not tight enough to be practically useful.

Computing the WCET with an integer linear programming (ILP) solver is proposed in [110] and [79]. The approach is named graph-based and implicit path enumeration respectively. We base our WCET analyzer on the ideas from these two groups.

The WCET is calculated by transforming the calculation to an integer linear programming problem. Each basic block[10] is represented by an edge e_i in the T-graph (timing graph) with the weight of the execution time of the basic block. Vertices v_i in the graph represent the split and join points in the control flow. Furthermore, each edge is also assigned an execution frequency f_i. The constraints resulting from the T-graph and additional

[10]A basic block is a sequence of instructions without any jumps or jump targets within this sequence.

functional constraints (e.g. loop bounds) are solved by an ILP solver. The T-graph is similar to a CFG, where the execution time is modeled in the vertices. The motivation to model the execution time in the edges results from the observation that most basic blocks end with a conditional branch. A conditional branch usually has a different execution time, depending on whether it is taken or not. This difference is represented by two edges with different weights.

In [79] a similar approach with ILP is proposed. However, they use the CFG as the basis to build the ILP problem. The approach is extended to model the instruction cache with cache conflict graphs. The evaluation with an Intel i960 processor shows tight results for small programs [80]. However, the conservative modeling of the register window (overflow/underflow on each function call/return) adds 50 cycles to each call and return. This observation is another argument for a WCET aware processor architecture.

WCET analysis of object-oriented languages is presented by Gustafsson [51]. Gustafsson uses abstract interpretation for automatic flow analysis to find loop bounds and in feasible paths. The work is extended to a modular WCET tool with instruction cache analysis and pipeline analysis [43]. A summary of new complexities to WCET analysis is given in [52].

Cache memories for the instructions and data are classic examples of the paradigm: *Make the common case fast*. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET bound. Plenty of effort has gone into research to integrate the instruction cache into the timing analysis of tasks [11, 60], the cache's influence on task preemption [78, 31], and integration of the cache analysis with the pipeline analysis [59]. Heckmann et. al described the influence of different cache architectures on WCET analysis [61].

An overview of WCET related research is presented in [107] and [152].

6.6.2 WCET Analysis for Java

WCET analysis at the bytecode level became a research topic, at the time Java was considered for future real-time systems. WCET analysis at the bytecode level was first considered in [23]. It is argued that the well formed intermediate representation of a program in Java bytecode, which can also be generated from compilers for other languages (e.g. Ada), is well suited for a portable WCET analysis tool. In that paper, annotations for Java and Ada are presented to guide WCET analysis at bytecode level. The work is extended to address the machine-dependent low-level timing analysis [21]. Worst-case execution frequencies of Java bytecodes are introduced for a machine independent timing information. Pipeline effects (on the target machine) across bytecode boundaries are modeled by a *gain time* for bytecode pairs.

In [106] a portable WCET analysis is proposed. The abstract WCET analysis is performed on the development site and generates abstract WCET information. The concrete WCET analysis is performed on the target machine by replacing abstract values within the WCET formulae by the machine dependent concrete values.

In [22], an extension of [23] and [21], an approach how the portable WCET information can be embedded in the Java class file is given. It is suggested that the final WCET calculation is performed by the target JVM. The calculation is performed for each method and the static call tree is traversed by a recursive call of the WCET analyzer.

6.6.3 WCET Analysis for JOP

A strong indication that JOP is a *WCET friendly* design, are that other real-time analysis projects use JOP as the primary target platform. The first IPET based WCET analysis tool that includes the timing model of JOP is presented in [134]. A simplified version of the method cache, the two block cache, is analyzed for invocations in inner loops. Trevor Harmon developed a tree-based WCET analyzer for interactive back-annotation of WCET estimates into the program source [57, 56]. The tool is extended to integrate JOP's two block method cache [58]. Model checking is used to analyze the timing and scheduling properties of a complete application within a single model [24]. However, even with a simple example, consisting of two periodic and two sporadic tasks, this approach leads to a very long analysis time. In contrast to that approach our opinion is that a combined WCET analysis and schedulability analysis is infeasible for non-trivial applications. Therefore, we stay in the WCET analysis domain, and consider the well established schedulability analysis as an extra step.

Compared to those three tools, which also target JOP, our presented WCET tool is enhanced with: (a) analysis of bytecodes that are implemented in Java; (b) integration of data-flow analysis for loop bounds and receiver methods; (c) a tighter IPET based method cache analysis; and (d) an evaluation of model checking for exact modeling of the method cache.

7 Real-Time Garbage Collection

Automatic memory management or garbage collection greatly simplifies development of large systems. However, garbage collection is usually not used in real-time systems due to the unpredictable temporal behavior of current implementations of a garbage collector. In this chapter we describe a concurrent collector that is scheduled periodically in the same way as ordinary application threads. We provide an upper bound for the collector period so that the application threads will never run out of memory. This chapter is based on following papers: [124, 138, 135].

7.1 Introduction

Garbage Collection (GC) is an essential part of the Java runtime system. GC enables automatic dynamic memory management which is essential to build large applications. Automatic memory management frees the programmer from complex and error prone explicit memory management (malloc and free).

However, garbage collection is considered unsuitable for real-time systems due to the unpredictable blocking times introduced by the GC work. One solution, used in the Real-Time Specification for Java (RTSJ) [25], introduces new thread types with program-managed, scoped memory for dynamic memory requirements. This scoped memory (and static memory called *immortal* memory) is not managed by the GC, and strict assignment rules between different memory areas have to be checked at runtime. This programming model differs largely from standard Java and is difficult to use [87, 101].

We believe that for the acceptance of Java for real-time systems, the restrictions imposed by the RTSJ are too strong. To simplify creation of possible large real-time applications, most of the code should be able to use the GC managed heap. For a collector to be used in real-time systems two points are essential:

- The GC has to be incremental with a short maximum blocking time that has to be known
- The GC has to keep up with the garbage generated by the application threads to avoid out-of-memory stalls

The first point is necessary to limit interference between the GC thread and high-priority threads. It is also essential to minimize the overhead introduced by read- and write-barriers, which are necessary for synchronization between the GC thread and the application threads. The design of a GC within these constraints is the topic of this chapter.

The second issue that has to be considered is scheduling the GC so that the GC collects enough garbage. The memory demands (static and dynamic) by the application threads have to be analyzed. These requirements, together with the properties of the GC, result in scheduling parameters for the GC thread. We will provide a solution to calculate the maximum period of the GC thread that will collect enough memory in each collector cycle so we will never run out of memory. The collector cycle depends on the heap size and the allocation rate of the application threads.

To distinguish between other garbage collectors and a collector for (hard) real-time systems we define a real-time collector as follows:

> A real-time garbage collector provides time predictable automatic memory management for tasks with a bounded memory allocation rate with minimal temporal interference to tasks that use only static memory.

The collector presented in this chapter is based on the work by Steele [142], Dijkstra [38] and Baker [19]. However, the copying collector is changed to perform the copy of an object concurrently by the collector and not as part of the mutator work. Therefore we name it *concurrent-copy* collector.

We will use the terms first introduced by Dijkstra with his *On-the-Fly* concurrent collector [38]. The application is called the *mutator* to reinforce that the application changes (mutates) the object graph while the GC does the collection work. The GC process is simply called *collector*. In the following discussion we will use the color scheme of white, gray, and black objects:

Black indicates that the object and all immediate descendants have been visited by the collector.

Grey objects have been visited, but the descendants may not have been visited by the collector, or the mutator has changed the object.

White objects are unvisited. At the beginning of a GC cycle all objects are white. At the end of the tracing, all white objects are garbage.

At the end of a collection cycle all black objects are live (or floating garbage) and all white objects are garbage.

7.1.1 Incremental Collection

An incremental collector can be realized in two ways: either by doing part of the work on each allocation of a new object or by running the collector as an independent process. For a single-threaded application, the first method is simpler as less synchronization between the application and the collector is necessary. For a multi-threaded environment there is no advantage by interleaving collector work with object allocation. In this case we need synchronization between the collector work done by one thread and the manipulation of the object graph performed by the other mutator thread. Therefore we will consider a concurrent solution where the collector runs in its own thread or processor. It is even possible to realize the collector as dedicated hardware [49].

7.1.2 Conservatism

Incremental collector algorithms are conservative, meaning that objects becoming unreachable during collection are not collected by the collector — they are floating garbage. Many approaches exist to reduce this conservatism in the general case. However, algorithms that completely avoid floating garbage are impractical. For different conservative collectors the worst-case bounds are all the same (i.e., all objects that become unreachable during collection remain floating garbage). Therefore the level of conservatism is not an issue for real-time collectors.

7.1.3 Safety Critical Java

In [137] a profile for safety critical Java (SCJ) is defined. SCJ has two interesting properties that may simplify the implementation of a real-time collector. Firstly, the split between initialization and mission phase, and secondly the simplified threading model (which also mandates that self-blocking operations are illegal in the mission phase). During initialization of the application a SCJ virtual machine does not have to meet any real-time constraints (other than possibly a worst case bound on the entire initialization phase). It is perfectly acceptable to use a non-real-time GC implementation during this phase – even a stop-the-world GC. As the change from initialization to mission phase is explicit, it is clear when the virtual machine must initiate real-time collection and which code runs during the mission phase.

Simplifying the threading model has the following advantage, if the collector thread runs at a lower priority than all other threads in the system, it is the case that when it runs *all* other threads have returned from their calls to `run()`. This is trivially true due to the priority preemptive scheduling discipline.[1] Any thread that has not returned from its run() method will preempt the GC until it returns. This has the side effect that the GC will never see a root in the call stack of another thread. Therefore, the usually atomic operation of scanning call stacks can be omitted in the mission phase. We will elaborate on this property in Section 7.3.

7.2 Scheduling of the Collector Thread

The collector work can be scheduled either *work* based or *time* based. On a work based scheduling, as performed in [140], an incremental part of the collector work is performed at object allocation. This approach sounds quite natural as threads that allocate more objects have to pay for the collector work. Furthermore, no additional collector thread is necessary. The main issue with this approach is to determine how much work has to be done on each allocation – a non trivial question as collection work consists of different phases. A more subtle question is: Why should a high frequency (and high priority) thread increase its WCET by performing collector work that does not have to be done at that period? Leaving the collector work to a thread with a longer period allows higher utilization of the system.

On a time based scheduling of the collector work, the collector runs in its own thread. Scheduling this thread as a *normal* real-time thread is quite natural for a hard real-time system. The question is: which priority to assign to the collector thread? The Metronome collector [18] uses the highest priority for the collector. Robertz and Henriksson [113] and Schoeberl [124] argue for the lowest priority. When building hard real-time systems the answer must take scheduling theory into consideration: the priority is assigned according to the period, either rate monotonic [83] or more general deadline monotonic [13]. Assuming that the period of the collector is the longest in the system and the deadline equals the period the collector gets the lowest priority.

In this section we provide an upper bound for the collector period so that the application threads will never run out of memory. The collector period, besides the WCET of the collector, is the single parameter of the collector that can be incorporated in standard schedulability analysis.

The following symbols are used in this section: heap size for a mark-compact collector (H_{MC}) and for a concurrent-copying collector (H_{CC}) containing both semi-spaces, period of the GC thread (T_{GC}), period of a single mutator thread (T_M), period of mutator thread i (T_i) from a set of threads, and memory amount allocated by a single mutator (a) or by mutator i (a_i) from a set of threads.

[1] If we would allow blocking in the application threads, we would also need to block the GC thread.

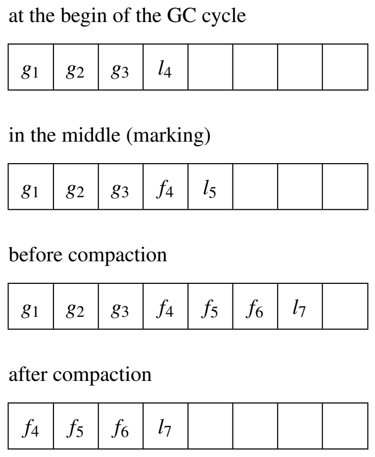


Figure 7.1: Heap usage during a mark-compact collection cycle

We assume that the mutator allocates all memory at the start of the period and the memory becomes garbage at the end. In other words the memory is live for one period. This is the worst-case,[2] but very common as we can see in the following code fragment.

```
for (;;) {
    Node n = new Node();
    work(n);
    waitForNextPeriod();
}
```

The object Node is allocated at the start of the period and n will reference it until the next period when a new Node is created and assigned to n. In this example we assume that no reference to Node is stored (inside work) to an object with a longer lifetime.

7.2.1 An Example

We start our discussion with a simple example[3] where the collector period is 3 times the mutator period ($T_{GC} = 3T_M$) and a heap size of 8 objects ($8a$). We show the heap during one GC cycle for a mark-compact and a concurrent-copy collector. The following letters

[2]See Section 7.2.3 for an example where the worst-case lifetime is two periods.

[3]The relation between the heap size and the mutator/collector proportion is an arbitrary value in this example. We will provide the exact values in the next sections.

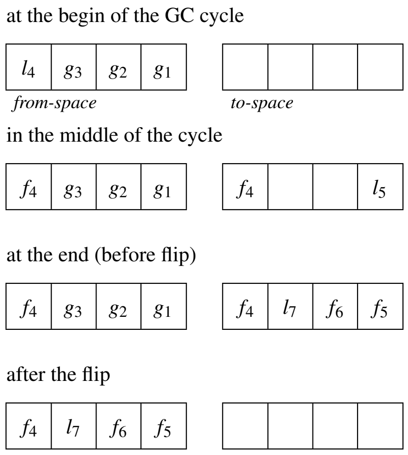


Figure 7.2: Heap usage during a concurrent-copy collection cycle

are used to show the status of a memory cell (that contains one object from the mutator in this example) in the heap: g_i is garbage from mutator cycle i, l is the live memory, and f is floating garbage. We assume that all objects that become unreachable during the collection remain floating garbage.

Figure 7.1 shows the changes in the heap during one collection cycle. At the start there are three objects (g_1, g_2, and g_3) left over from the last cycle (floating garbage) which are collected by the current cycle and one live object l_4. During the collection the live objects become unreachable and are now floating garbage (e.g. f_4 in the second sub-figure). At the end of the cycle, just before compacting, we have three garbage cells (g_1-g_3), three floating garbage cells (f_4-f_6) and one live cell l_7. Compaction moves the floating garbage and the live cell to the start of the heap and we end up with four free cells. The floating garbage will become garbage in the next collection cycle and we start over with the first sub-figure with three garbage cells and one live cell.

Figure 7.2 shows one collection cycle of the concurrent-copy collector. We have two memory spaces: the *from-space* and the *to-space*. Again we start the collection cycle with one live cell and three garbage cells left over from the last cycle. Note that the order of the cells is different from the previous example. New cells are allocated in the to-space from the top of the heap, whereas moved cells are allocated from the bottom of the heap. The second sub-figure shows a snapshot of the heap during the collection: formerly live object l_4 is already floating garbage f_4 and copied into to-space. A new cell l_5 is allocated in the to-space. Before the flip of the two semi-spaces the from-space contains the three garbage cells (g_1-g_3) and the to-space the three floating garbage cells (f_4-f_6) and one live cell l_7. The last sub-figure shows the heap after the flip: The from-space contains the three floating cells which will be garbage cells in the next cycle and the one live cell. The to-space is now empty.

From this example we see that the necessary heap size for a mark-compact collector is similar to the heap size for a copying collector. We also see that the compacting collector has to move more cells (all floating garbage cells and the live cell) than the copying collector (just the one cell that is live at the beginning of the collection).

7.2.2 Minimum Heap Size

In this section we show the memory bounds for a mark-compact collector with a single heap memory and a concurrent-copying collector with the two spaces *from-space* and *to-space*.

Mark-Compact

For the mark-compact collector, the heap H_{MC} can be divided into allocated memory M and free memory F

$$H_{MC} = M + F = G + \overline{G} + L + F \tag{7.1}$$

where G is garbage at the start of the collector cycle that will be reclaimed by the collector. Objects that become unreachable during the collection cycle and will not be reclaimed are floating garbage $\overline{G}$. These objects will be detected in the next collection cycle. We assume the worst case that all objects that die during the collection cycle will not be detected and therefore are floating garbage. L denotes all live,i.e. reachable, objects. F is the remaining free space.

We have to show that we will never run out of memory during a collection cycle ($F \geq 0$). The amount of allocated memory M has to be less than or equal to the heap size H_{MC}

$$H_{MC} \geq M = G + \overline{G} + L \tag{7.2}$$

In the following proof the superscript n denotes the collection cycle. The subscript letters S and E denote the value at the start and the end of the cycle, respectively.

Lemma 1. *For a collection cycle the amount of allocated memory M is bounded by the maximum live data L_{max} at the start of the collection cycle and two times A_{max}, the maximum data allocated by the mutator during the collection cycle.*

$$M \leq L_{max} + 2A_{max} \tag{7.3}$$

Proof. During a collection cycle G remains constant. All live data that becomes unreachable will be floating garbage. Floating garbage $\overline{G}_E$ at the end of cycle n will be detected (as garbage G) in cycle $n+1$.

$$G^{n+1} = \overline{G}_E^n \tag{7.4}$$

The mutator allocates A memory and transforms part of this memory and part of the live data at the start L_S to floating garbage $\overline{G}_E$ at the end of the cycle. L_E is the data that is still reachable at the end of the cycle.

$$L_S + A = L_E + \overline{G}_E \tag{7.5}$$

with $A \leq A_{max}$ and $L_S \leq L_{max}$. A new collection-cycle start immediately follows the end of the former cycle. Therefore the live data remains unchanged.

$$L_S^{n+1} = L_E^n \tag{7.6}$$

We will show that (7.3) is true for cycle 1. At the start of the first cycle we have no garbage ($G = 0$) and no live data ($L_S = 0$). The heap contains only free memory.

$$M_S^1 = 0 \tag{7.7}$$

During the collection cycle the mutator allocates A^1 memory. Part of this memory will be live at the end and the remaining will be floating garbage.

$$A^1 = L_E^1 + \overline{G}_E^1 \tag{7.8}$$

Therefore at the end of the first cycle

$$\begin{aligned} M_E^1 &= L_E^1 + \overline{G}_E^1 \\ M^1 &= A^1 \end{aligned} \tag{7.9}$$

As $A^1 \leq A_{max}$ (7.3) is fulfilled for cycle 1.

Under the assumption that (7.3) is true for cycle n, we have to show that (7.3) holds for cycle $n+1$.

$$M^{n+1} \leq L_{max} + 2A_{max} \tag{7.10}$$

$$
\begin{aligned}
M^n &= G^n + \overline{G}_E^n + L_E^n && && (7.11)\\
M^{n+1} &= G^{n+1} + \overline{G}_E^{n+1} + L_E^{n+1} && && (7.12)\\
&= \overline{G}_E^n + L_S^{n+1} + A^{n+1} && \text{apply (7.4) and (7.5)} && \\
&= \overline{G}_E^n + L_E^n + A^{n+1} && \text{apply (7.6)} && \\
&= L_S^n + A^n + A^{n+1} && \text{apply (7.5)} && (7.13)
\end{aligned}
$$

As $L_S \leq L_{max}$, $A^n \leq A_{max}$ and $A^{n+1} \leq A_{max}$

$$M^{n+1} \leq L_{max} + 2A_{max} \tag{7.14}$$

□

Concurrent-Copy

In the following section we will show the memory bounds for a concurrent-copying collector with the two spaces *from-space* and *to-space*. We will use the same symbols as in Section 7.2.2 and denote the maximum allocated memory in the from-space as M_{From} and the maximum allocated memory in the to-space as M_{To}.

For a copying-collector the heap H_{CC} is divided in two equal sized spaces H_{From} and H_{To}. The amount of allocated memory M in each semi-space has to be less than or equal to $\frac{H_{CC}}{2}$

$$H_{CC} = H_{From} + H_{To} \geq 2M \tag{7.15}$$

Lemma 2. *For a collection cycle, the amount of allocated memory M in each semi-space is bounded by the maximum live data L_{max} at the start of the collection cycle and A_{max}, the maximum data allocated by the mutator during the collection cycle.*

$$M \leq L_{max} + A_{max} \tag{7.16}$$

Proof. Floating garbage at the end of cycle n will be detectable garbage in cycle $n+1$

$$G^{n+1} = \overline{G}_E^n \tag{7.17}$$

Live data at the end of cycle n will be the live data at the start of cycle $n+1$

$$L_S^{n+1} = L_E^n \tag{7.18}$$

The allocated memory M_{From} in the from-space contains garbage G and the live data at the start L_s.

$$M_{From} = G + L_S \tag{7.19}$$

All new objects are allocated in the to-space. Therefore the memory requirement for the from-space does not change during the collection cycle. All garbage G remains in the from-space and the to-space only contains floating garbage and live data.

$$M_{To} = \overline{G} + L \tag{7.20}$$

At the start of the collection cycle, the to-space is completely empty.

$$M_{To\_S} = 0 \tag{7.21}$$

During the collection cycle all live data is copied into the to-space and new objects are allocated in the to-space.

$$M_{To\_E} = L_S + A \tag{7.22}$$

At the end of the collector cycle, the live data from the start L_S and new allocated data A stays either live at the end L_E or becomes floating garbage $\overline{G}_E$.

$$L_S + A = L_E + \overline{G}_E \tag{7.23}$$

For the first collection cycle there is no garbage ($G = 0$) and no live data at the start ($L_S = 0$), i.e. the from-space is empty ($M^1_{From} = 0$). The to-space will only contain all allocated data A^1, with $A^1 \leq A_{max}$, and therefore (7.16) is true for cycle 1.

Under the assumption that (7.16) is true for cycle n, we have to show that (7.16) holds for cycle $n+1$.

$$\begin{aligned} M^{n+1}_{From} &\leq L_{max} + A_{max} \\ M^{n+1}_{To} &\leq L_{max} + A_{max} \end{aligned} \tag{7.24}$$

At the start of a collection cycle, the spaces are flipped and the new to-space is cleared.

$$\begin{aligned} H^{n+1}_{From} &\Leftarrow H^n_{To} \\ H^{n+1}_{To} &\Leftarrow \emptyset \end{aligned} \tag{7.25}$$

The from-space:

$$M^n_{From} = G^n + L^n_S \tag{7.26}$$

$$\begin{aligned} M^{n+1}_{From} &= G^{n+1} + L^{n+1}_S \qquad (7.27) \\ &= \overline{G}^n_E + L^n_E \\ &= L^n_S + A^n \qquad (7.28) \end{aligned}$$

As $L_S \leq L_{max}$ and $A^n \leq A_{max}$

$$M^{n+1}_{From} \leq L_{max} + A_{max} \tag{7.29}$$

The to-space:

$$M^n_{To} = \overline{G}^n_E + L^n_E \tag{7.30}$$

$$\begin{aligned} M^{n+1}_{To} &= \overline{G}^{n+1}_E + L^{n+1}_E \qquad (7.31) \\ &= L^{n+1}_S + A^{n+1} \\ &= L^n_E + A^{n+1} \qquad (7.32) \end{aligned}$$

As $L_E \leq L_{max}$ and $A^{n+1} \leq A_{max}$

$$M^{n+1}_{To} \leq L_{max} + A_{max} \tag{7.33}$$

□

From this result we can see that the dynamic memory consumption for a mark-compact collector is similar to a concurrent-copy collector. This is contrary to the common belief that a copy collector needs the double amount of memory.

We have seen that the double-memory argument against a copying collector does not hold for an incremental real-time collector. We need double the memory of the allocated data during a collection cycle in either case. The advantage of the copying collector over a compacting one is that newly allocated data are placed in the to-space and does not need to be copied. The compacting collector moves all newly created data (that is mostly floating garbage) at the compaction phase.

7.2.3 Garbage Collection Period

GC work is inherently periodic. After finishing one round of collection the GC starts over. The important question is which is the *maximum* period the GC can be run so that the application will never run out of memory. Scheduling the GC at a shorter period does not hurt but decreases utilization.

In the following, we derive the maximum collector period that guarantees that we will not run out of memory. The maximum period T_{GC} of the collector depends on L_{max} and A_{max} for which safe estimates are needed.

We assume that the mutator allocates all memory at the start of the period and the memory becomes garbage at the end. In other words the memory is live for one period. This is the worst case, but very common.

In this section the upper bound of the period for the collector thread is given for n independent mutator threads.

Theorem 1. *For "n" mutator threads with period T_i where each thread allocates a_i memory each period, the maximum collector period T_{GC} that guarantees that we will not run out of memory is*

$$T_{GC} \leq \frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \tag{7.34}$$

$$T_{GC} \leq \frac{H_{CC} - 4\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \tag{7.35}$$

Proof. For n mutator threads with periods T_i and allocations a_i during each period the values for L_{max} and A_{max} are

$$L_{max} = \sum_{i=1}^{n} a_i \tag{7.36}$$

$$A_{max} = \sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \tag{7.37}$$

The ceiling function for A_{max} covers the individual worst cases for the thread schedule and cannot be solved analytically. Therefore we use a conservative estimation A'_{max} instead of A_{max}.

$$A'_{max} = \sum_{i=1}^{n} \left(\frac{T_{GC}}{T_i} + 1 \right) a_i \geq \sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \tag{7.38}$$

From (7.2) and (7.3) we get the minimum heap size for a mark-compact collector

$$\begin{aligned} H_{MC} &\geq L_{max} + 2A_{max} \\ &\geq \sum_{i=1}^{n} a_i + 2\sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \end{aligned} \tag{7.39}$$

For a given heap size H_{MC} we get the conservative upper bound of the maximum collector period T_{GC} [4]

$$2A'_{max} \leq H_{MC} - L_{max}$$

$$2\sum_{i=1}^{n}\left(\frac{T_{GC}}{T_i}+1\right)a_i \leq H_{MC} - L_{max} \tag{7.40}$$

$$T_{GC} \leq \frac{H_{MC} - L_{max} - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{T_i}} \tag{7.41}$$

$$\Rightarrow T_{GC} \leq \frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{Ti}} \tag{7.42}$$

Equations (7.15) and (7.16) result in the minimum heap size H_{CC}, containing both semi-spaces, for the concurrent-copy collector

$$H_{CC} \geq 2L_{max} + 2A_{max}$$

$$\geq 2\sum_{i=1}^{n} a_i + 2\sum_{i=1}^{n}\left\lceil\frac{T_{GC}}{T_i}\right\rceil a_i \tag{7.43}$$

For a given heap size H_{CC} we get the conservative upper bound of the maximum collector period T_{GC}

$$2A'_{max} \leq H_{CC} - 2L_{max}$$

$$2\sum_{i=1}^{n}\left(\frac{T_{GC}}{T_i}+1\right)a_i \leq H_{CC} - 2L_{max} \tag{7.44}$$

$$T_{GC} \leq \frac{H_{CC} - 2L_{max} - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{T_i}} \tag{7.45}$$

$$\Rightarrow T_{GC} \leq \frac{H_{CC} - 4\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{Ti}} \tag{7.46}$$

□

[4] It has to be noted that this is a conservative value for the maximum collector period T_{GC}. The maximum value $T_{GC_{max}}$ that fulfills (7.39) is in the interval

$$\left(\frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{T_i}}, \frac{H_{MC} - \sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{T_i}}\right)$$

and can be found by an iterative search.

Producer/Consumer Threads

So far we have only considered threads that do not share objects for communication. This execution model is even more restrictive than the RTSJ scoped memories that can be shared between threads. In this section we discuss how our GC scheduling can be extended to account for threads that share objects.

Object sharing is usually done by a producer and a consumer thread. I.e., one thread allocates the objects and stores references to those objects in a way that they can be accessed by the other thread. This other thread, the consumer, is in charge to *free* those objects after use.

An example of this sharing is a device driver thread that periodically collects data and puts them into a list for further processing. The consumer thread, with a longer period, takes all available data from the list at the start of the period, processes the data, and removes them from the list. During the data processing, new data can be added by the producer. Note that in this case the list will probably never be completely empty. This typical case cannot be implemented by an RTSJ shared scoped memory. There would be no point in the execution where the shared memory will be empty and can get recycled.

The question now is how much data will be alive in the worst case. We denote T_p as the period of the producer thread τ_p and T_c as the period of the consumer thread τ_c. τ_p allocates a_p memory each period. During one period of the consumer τ_c the producer τ_p allocates

$$\left\lceil \frac{T_c}{T_p} \right\rceil a_p$$

memory. The worst case is that τ_c takes over all objects at the start of the period and frees them at the end. Therefore the maximum amount of live data for this producer/consumer combination is

$$\left\lceil \frac{2T_c}{T_p} \right\rceil a_p$$

To incorporate this extended lifetime of objects we introduce a lifetime factor l_i which is

$$l_i = \begin{cases} 1 & : \text{ for normal threads} \\ \left\lceil \frac{2T_c}{T_i} \right\rceil & : \text{ for producer } \tau_i \text{ and associated consumer } \tau_c \end{cases} \tag{7.47}$$

and extend L_{max} from (7.36) to

$$L_{max} = \sum_{i=1}^{n} a_i l_i \tag{7.48}$$

The maximum amount of memory A_{max} that is allocated during one collection cycle is not changed due to the *freeing* in a different thread and therefore remains unchanged.

The resulting equations for the maximum collector period are

$$T_{GC} \leq \frac{H_{MC} - \sum_{i=1}^{n} a_i l_i - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \tag{7.49}$$

and

$$T_{GC} \leq \frac{H_{CC} - 2\sum_{i=1}^{n} a_i l_i - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \tag{7.50}$$

Static Objects

The discussion about the collector cycle time assumes that all live data is produced by the periodic application threads and the maximum lifetime is one period. However, in the general case we have also live data that is allocated in the initialization phase of the real-time application and stays alive until the application ends. We incorporate this value by including this static live memory L_s in L_{max}

$$L_{max} = L_s + \sum_{i=1}^{n} a_i l_i \tag{7.51}$$

A mark-compact collector moves all static data to the bottom of the heap in the first and second[5] collection cycle after the allocation. It does not have to compact these data during the following collection cycles in the mission phase. The concurrent-copy collector moves these static data in each collection cycle. Furthermore, the memory demand for the concurrent copy is increased by the double amount of the static data (compared to the single amount in the mark-compact collector)[6].

The SCJ application model with an initialization and a mission phase can reduce the amount of live data that needs to be copied (see Section 7.3).

Object Lifetime

Listing 7.1 shows an example of a periodic thread that allocates an object in the main loop and the resulting bytecodes.

[5] A second cycle is necessary as this static data can get intermixed by floating garbage from the first collector cycle.

[6] Or the collector period gets shortened.

```
  public void run() {

      for (;;) {
          Node n = new Node();
          work(n);
          waitForNextPeriod();
      }
  }

public void run();
  Code:
   0:   new #20; //class Node
   3:   dup
   4:   invokespecial   #22; //"<init>":()V
   7:   astore_1
   8:   aload_1
   9:   invokestatic    #26; //work:(Node)V
   12:  aload_0
   13:  invokevirtual   #30; //wFNP:()Z
   16:  pop
   17:  goto    0
```

Listing 7.1: Example periodic thread and the corresponding Java bytecodes

There is a time between allocation of Node and the assignment to n where a reference to the former Node (from the former cycle) and the new Node (on the operand stack) is live. To handle this issue we can either change the values of L_{max} and A_{max} to accommodate this additional object or change the top-level code of the periodic work to explicitly assign a null-pointer to the local variable n as it can be seen in Listing 7.4 from the evaluation section. Programming against the SCJ profile avoids this issues (see Section 7.3).

However, this null pointer assignment is only necessary at the top-level method that invokes waitForNextPeriod and is therefore not as complex as explicit freeing of objects. Objects that are created inside work in our example do not need to be *freed* in this way as the reference to the object gets *lost* on return from the method.

7.3 SCJ Simplifications

The restrictions of the computational model for safety critical Java allow for optimizations of the GC. We can avoid atomic stack scanning for roots and do not have to deal with exact pointer finding. Static objects, which would belong into immortal memory in the RTSJ, can be detected by a special GC cycle at transition to the mission phase. We can treat those objects specially and do not need to collect them during the mission phase. This static memory area is automatically sized.

It has to be noted that our proposal is extending JSR 302. Clearly, adding RTGC to SCJ reduces the importance of scopes and would likely relegate them to the small subset of applications where fast deallocation is crucial. Discussing the interaction between scoped memory and RTGC is beyond the scope of this chapter.

7.3.1 Simple Root Scanning

Thread stack scanning is usually performed atomically. Scanning of the thread stacks with a snapshot-at-beginning write barrier [156] allows optimization of the write barriers to consider only field access (putfield and putstatic) and array access. Reference manipulation in locals and on the operand stack can be ignored for a write barrier. However, this optimization comes at the cost of a possible large blocking time due to the atomicity of stack scanning.

A subtle difference between the RTSJ and the SCJ definition is the possibility to use local variables within run() (see example in Figure 5.6). Although handy for the programmer to preserve state information in locals,[7] GC implementation can greatly benefit from *not* having reference values on the thread stack when the thread suspenses execution.

If the GC thread has the lowest priority and there is no blocking library function that can suspend a real-time thread, then the GC thread will only run when all real-time threads are waiting for their next period – and this waiting is performed after the return from the run() method. In that case the other thread stacks are completely *empty*. We do not need to scan them for roots as the only roots are the references in static (class) variables.

For a real-time GC root scanning has to be exact. With conservative stack scanning, where a primitive value is treated as a pointer, possible large data structures can be kept alive artificially. To implement exact stack scanning we need the information of the stack layout for each possible GC preemption point. For a high-priority GC this point can be at each bytecode (or at each machine instruction for compiling Java). The auxiliary data

[7]Using multiple wFNP() invocations for local mode changes can also come handy. The author has used this fact heavily in the implementation of a modem/PPP protocol stack.

structure to capture the stack layout (and information which machine register will hold a reference for compiled Java) can get quite large [92] or require additional effort to compute.

With a low-priority GC and the RTSJ model of periodic thread coding with wFNP() the number of GC preemption points is decreased dramatically. When the GC runs all threads will be in wFNP(). Only the stack information for those places in the code have to be available. It is also assumed that wFNP() is not invoked very deep in the call hierarchy. Therefore, the stack high will be low and the resulting blocking time short.

As mentioned before, the SCJ style periodic thread model results in an empty stack at GC runtime. As a consequence we do not have to deal with exact stack scanning and need no additional information about the stack layout.

7.3.2 Static Memory

A SCJ copying collector will perform best when all live data is produced by periodic threads and the maximum lifetime of a newly allocated object is one period. However, some data structures allocated in the initialization phase stay alive for the whole application lifetime. In an RTSJ application this data would be allocated in immortal memory. With a real-time GC there is no notion of immortal memory, instead we will use the term *static* memory. Without special treatment, a copying collector will move this data at each GC cycle. Furthermore, the memory demand for the collector increases by the amount of the static data.

As those static objects (mostly) live *forever*, we propose a solution similar to the immortal memory of the RTSJ. All data allocated during the initialization phase (where no application threads are scheduled) is considered potentially static. As part of the transition to the mission phase a *special* collection cycle in a stop-the-world fashion is performed. Objects that are still alive after this cycle are assumed to live forever and make up the *static* memory area. The remaining memory is used for the garbage collected heap.

This static memory will still be scanned by the collector to find references into the heap but it is not collected. The main differences between our static memory and the immortal memory of the RTSJ are:

This static live data will still be scanned by the collector to find references into the heap but it is not collected. The main differences between our immortal memory and the memory areas of the RTSJ are:

- The choice of allocation context is implicit. There is no need to specify where an object must be allocated. We do not have to state explicitly which data belongs to the application life-time data. This information is implicitly gathered by the start-mission transition.

- References from the static memory to the garbage collected heap are allowed contrary to the fact in the RTSJ that references to scoped memories, that have to be used for dynamic memory management without a GC, are not allowed from immortal memory.

The second fact greatly simplifies communication between threads. For a typical producer/consumer configuration the container for the shared data is allocated in immortal memory and the actual data in the garbage collected heap. With this *immortal* memory solution the actual L_{max} only contains allocated memory from the periodic threads.

7.4 Implementation

The collector for JOP is an incremental collector [124, 138] based on the copy collector by Cheney [32] and the incremental version by Baker [19]. To avoid the expensive read barrier in Baker's collector all object copies are performed concurrently by the collector. The collector is concurrent and resembles the collectors presented by Steele [142] and Dijkstra et al. [38]. Therefore we call it the *concurrent-copy* collector.

The collector and the mutator are synchronized by two barriers. A Brooks-style [29] forwarding directs the access to the object either into tospace or fromspace. The forwarding pointer is kept in a separate handle area as proposed in [89]. The separate handle area reduces the space overheads as only one pointer is needed for both object copies. Furthermore, the indirection pointer does not need to be copied. The handle also contains other object related data, such as type information, and the mark list. The objects in the heap only contain the fields and no object header.

The second synchronization barrier is a *snapshot-at-beginning* write-barrier [156]. A snapshot-at-beginning write-barrier synchronizes the mutator with the collector on a reference store into a static field, an object field, or an array.

The whole collector, the `new` operation, and the write barriers are implemented in Java (with the help of native functions for direct memory access). The object copy operation is implemented in hardware and can be interrupted by mutator threads after each word copied [135]. The copy unit redirects the access to the object under copy, depending on the accessed field, either to the original or the new version of the object.

Although we show the implementation on a Java processor, the GC is not JOP specific and can also be implemented on a conventional processor.

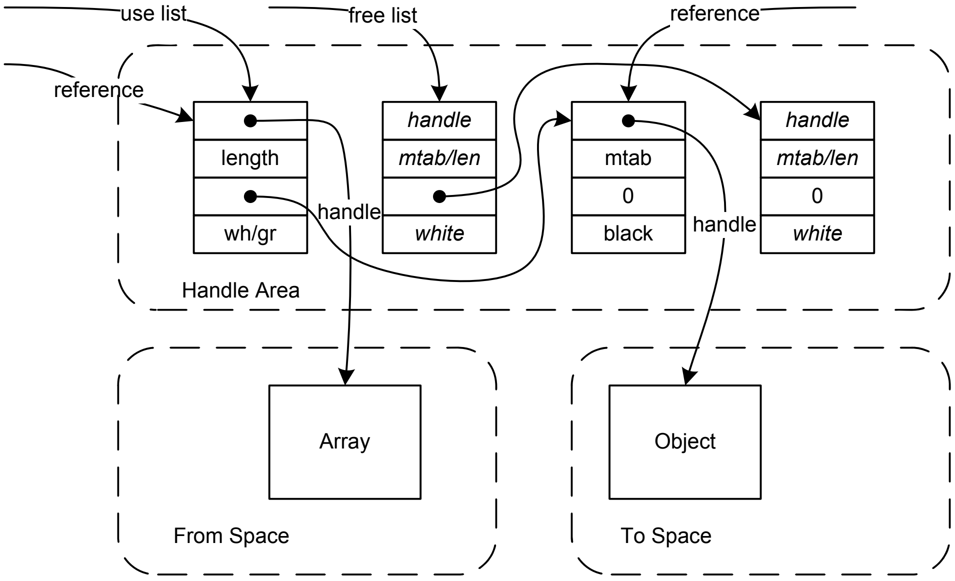


Figure 7.3: Heap layout with the handle area

7.4.1 Heap Layout

Figure 7.3 shows a symbolic representation of the heap layout with the handle area and two semi-spaces, *fromspace* and *tospace*. Not shown in this figure is the memory region for runtime constants, such as class information or string constants. This memory region, although logically part of the heap, is neither scanned, nor copied by the GC. This constant area contains its own handles and all references into this area are ignored by the GC.

To simplify object move by the collector, all objects are accessed with one indirection, called the handle. The handle also contains auxiliary object data structures, such as a pointer to the method table or the array length. Instead of Baker's read barrier we have an additional mark stack which is a threaded list within the handle structure. An additional field (as shown in Figure 7.3) in the handle structure is used for a free list and a use list of handles.

The indirection through a handle, although a very light-weight read barrier, is usually still considered as a high overhead. Metronome [18] uses a forwarding pointer as part of the object and performs forwarding *eagerly*. Once the pointer is forwarded, subsequent uses of the reference can be performed on the direct pointer until a GC preemption point. This optimization is performed by the compiler.

JOP uses a hardware based optimization for this indirection [125]. The indirection is unconditionally performed in the memory access unit. Furthermore, null pointer checks and array bounds checks are done in parallel to this indirection.

There are two additional benefits from an explicit handle area instead of a forwarding pointer: (a) access to the method table or array size needs no indirection, and (b) the forwarding pointer and the auxiliary data structures do not need to be copied by the GC.

The fixed handle area is not subject to fragmentation as all handles have the same size and are recycled at a sweep phase with a simple free list. However, the reserved space has to be sized (or the GC period adapted) for the maximum number of objects that are live or are floating garbage.

7.4.2 The Collector

The collector can operate in two modes: (1) as stop-the-world collector triggered on allocation when the heap is full, or (2) as concurrent real-time collector running in its own thread.

The real-time collector is scheduled periodically at the lowest priority and within each period it performs the following steps:

Flip An atomic flip exchanges the roles of tospace and fromspace.

Mark roots All static references are pushed onto the mark stack. Only a single push operation needs to be atomic. As the thread stacks are empty we do not need an atomic scan of thread stacks.

Mark and copy An object is popped from the mark stack, all referenced objects, which are still white, are pushed on the mark stack, the object is copied to tospace and the handle pointer is updated.

Sweep handles All handles in the use list are checked if they still point into tospace (black objects) or can be added to the handle free list.

Clear fromspace At the end of the collector work the fromspace that contains only white objects is initialized with zero. Objects allocated in that space (after the next flip) are already initialized and allocation can be performed in constant time.

To reduce blocking time, a hardware unit performs copies of objects and arrays in an interruptible fashion, and records the copy position on an interrupt. On an object or array access the hardware knows whether the access should go to the already copied part in the tospace or in the not yet copied part in the fromspace. It has to be noted that splitting larger arrays into smaller chunks, as done in Metronome [18] and in the GC for the JamaicaVM [140], is a software option to reduce the blocking time.

The collector has two modes of operation: one for the initialization phase and one for the mission phase. At the initialization phase it operates in a stop-the-world fashion and gets invoked when a memory request cannot be satisfied. In this mode the collector scans the stack of the single thread conservatively. It has to be noted that each reference points into the handle area and not to an arbitrary position in the heap. This information is considered by the GC to distinguish pointers from primitives. Therefore the chance to keep an object artificially alive is low.

As part of the mission start one stop-the-world cycle is performed to clean up the heap from garbage generated at initialization. From that point on the GC runs in concurrent mode in its own thread and omits scanning of the thread stacks.

Implementation Code Snippets

This sections shows the important code fragments of the implementation. As can be seen, the implementation is quite short.

Flip involves manipulation of a few pointers and changes the meaning of black (toSpace) and white.

```
synchronized (mutex) {
    useA = !useA;
    if (useA) {
        copyPtr = heapStartA;
        fromSpace = heapStartB;
        toSpace = heapStartA;
    } else {
        copyPtr = heapStartB;
        fromSpace = heapStartA;
        toSpace = heapStartB;
    }
    allocPtr = copyPtr+semi_size;
}
```

Root Marking When the GC runs in concurrent mode only the static reference fields form the root set and are scanned. The stop-the-world mode of the GC also scans all stacks from all threads.

```
int addr = Native.rdMem(addrStaticRefs);
int cnt = Native.rdMem(addrStaticRefs+1);
for (i=0; i<cnt; ++i) {
    push(Native.rdMem(addr+i));
}
```

Push All gray objects are pushed on a gray stack. The gray stack is a list threaded within the handle structure.

```
if (Native.rdMem(ref+OFF_GREY)!=0) {
    return;
}
if (Native.rdMem(ref+OFF_GREY)==0) {
    // pointer to former gray list head
    Native.wrMem(grayList, ref+OFF_GREY);
    grayList = ref;
}
```

Mark and Copy The following code snippet shows the central GC loop.

```
for (;;) {

    // pop one object from the gray list
    synchronized (mutex) {
        ref = grayList;
        if (ref==GREY_END) {
            break;
        }
        grayList = Native.rdMem(ref+OFF_GREY);
        // mark as not in list
        Native.wrMem(0, ref+OFF_GREY);
    }

    // push all childs
    // get pointer to object
    int addr = Native.rdMem(ref);
    int flags = Native.rdMem(ref+OFF_TYPE);
    if (flags==IS_REFARR) {
        // is an array of references
        int size = Native.rdMem(ref+OFF_MTAB_ALEN);
        for (i=0; i<size; ++i) {
            push(Native.rdMem(addr+i));
        }
    } else if (flags==IS_OBJ){
        // its a plain object
        // get pointer to method table
        flags = Native.rdMem(ref+OFF_MTAB_ALEN);
        // get real flags
        flags = Native.rdMem(flags+MTAB2GC_INFO);
        for (i=0; flags!=0; ++i) {
            if ((flags&1)!=0) {
                push(Native.rdMem(addr+i));
            }
            flags >>>= 1;
        }
    }

    // now copy it − color it BLACK
    int size = Native.rdMem(ref+OFF_SIZE);
    synchronized (mutex) {
        // update object pointer to the new location
        Native.wrMem(copyPtr, ref+OFF_PTR);
        // set it BLACK
        Native.wrMem(toSpace, ref+OFF_SPACE);
        // copy it
        for (i=0; i<size; ++i) {
            Native.wrMem(Native.rdMem(addr+i), copyPtr+i);
        }
        copyPtr += size;
    }
}
```

Sweep Handles At the end of the mark and copy phase the handle area is swept to find all unused handles (the one that still point into fromSpace) and add them to the free list.

```
    synchronized (mutex) {
        ref = useList;        // get start of the list
        useList = 0;          // new uselist starts empty
    }

    while (ref!=0) {
        int next = Native.rdMem(ref+OFF_NEXT);
        // a BLACK one
        if (Native.rdMem(ref+OFF_SPACE)==toSpace) {
            // add to used list
            synchronized (mutex) {
                Native.wrMem(useList, ref+OFF_NEXT);
                useList = ref;
            }
        // a WHITE one
        } else {
            // add to free list
            synchronized (mutex) {
                Native.wrMem(freeList, ref+OFF_NEXT);
                freeList = ref;
                Native.wrMem(0, ref+OFF_PTR);
            }
        }
        ref = next;
    }
```

Clear Fromspace The last step of the GC clears the fromspace to provide a constant time allocation after the next flip.

```
        for (int i=fromSpace; i<fromSpace+semi_size; ++i) {
            Native.wrMem(0, i);
        }
```

7.4.3 The Mutator

The coordination between the mutator and the collector is performed within the new and newarray bytecodes and within write barriers for JVM bytecodes putfield and putstatic for reference fields, and bytecode aastore. The field access bytecodes are substituted at application link time (run of JOPizer). Only write accesses to reference fields are substituted by special versions of the bytecodes (putfield_ref and putstatic_ref). Therefore, the write barrier code is only executed on reference write access.

Allocation

Objects are allocated black (in tospace). In non real-time collectors it is more common to allocate objects white. It is argued [38] that objects die young and the chances are high

```
synchronized (GC.mutex) {
    // we allocate from the upper part
    allocPtr −= size;
    ref = getHandle(size);
    // mark as object
    Native.wrMem(IS_OBJ, ref+OFF_TYPE);
    // pointer to method table in the handle
    Native.wrMem(cons+CLASS_HEADR, ref+OFF_MTAB_ALEN);
}
```

Listing 7.2: Implementation of bytecode new in JOPs JVM

that the GC never needs to touch them. However, in the worst case no object that is created and becomes garbage during the GC cycle can be reclaimed. Those floating garbage will be reclaimed in the next GC cycle. Therefore, we do not benefit from the white allocation optimization in a real-time GC. Allocating a new object black has the benefit that those objects do not need to be copied. The same argument applies to the chosen write barrier. The code in Listing 7.2 shows the simple implementation of bytecode new:

As the old fromspace is cleared by the GC, the new object is already initialized and new executes in constant time. The methods Native.rdMem() and Native.wrMem() provide direct access to the main memory. Only those two native methods are necessary for an implementation of a GC in pure Java.

Write Barriers

For a concurrent (incremental) GC some coordination between the collector and the mutator are necessary. The usual solution is a write barrier in the mutator to not foil the collector. According to [153] GC concurrent algorithms can be categorized into:

Snapshot-at-beginning Keep the object graph as it was at the the GC start

- Save the to-be-overwritten reference
- More conservative – not an issue for RTs as worst case counts
- Allocate black
- New objects (e.g. new stack frames) do not need a write barrier
- Optimization: with atomic root scan of the thread stacks no write barrier is necessary for locals and the JVM stack

Incremental update *Help* the GC by doing some collection work in the mutator

- Preserve strong tri-color invariant (no pointer from black to white objects)
- On black to white shade the white object (shade the black is unusual)
- Allocate black (in contrast to [38])
- Needs write barriers for locals and manipulation on the stack
- Less conservative than snapshot-at-beginning

The usual choice is snapshot-at-beginning with atomic root scan of all thread stacks to avoid write barriers on locals. Assume the following assignment of a reference:

```
o.r = ref;
```

There are three references involved that can be manipulated:

- The old value of o.r
- The new value ref
- The object o

The three possible write barriers are:

1. Snapshot-at-beginning/weak tri-color invariant:

   ```
    if (white(o.r)) markGrey(o.r);
   o.r = ref;
   ```

2. Incremental/strong tri-color invariant with push forward

   ```
    if (black(o) && white(ref)) markGrey(ref);
   o.r = ref;
   ```

   This barrier can be optimized to only check if ref is white.

3. Incremental/strong tri-color invariant with push back

   ```
    if (black(o) && white(ref)) markGrey(o);
   o.r = ref;
   ```

```
private static void f_putfield_ref (int ref, int value, int index) {

    synchronized (GC.mutex) {

        // snapshot-at-beginning barrier
        int oldVal = Native.getField(ref, index);
        // Is it white?
        if (oldVal != 0
            && Native.rdMem(oldVal+GC.OFF_SPACE) != GC.toSpace) {
            // Mark grey
            GC.push(oldVal)
        }
        Native.putField(ref, index, value);
    }
}
```

Listing 7.3: Snapshot-at-beginning write-barrier in JOPs JVM

We have no stack roots when the collector runs. Therefore we could use the incremental write barrier for object fields only. However, for the worst case all floating garbage will not be found by the GC in the current cycle. Therefore, we use the snapshot-at-begin write barrier in our implementation.

A snapshot-at-beginning write-barrier synchronizes the mutator with the collector on a reference store into a static field, an object field, or an array. The *to be overwritten* field is shaded gray as shown in Listing 7.3. An object is shaded gray by pushing the reference of the object onto the mark stack.[8] Further scanning and copying into tospace – coloring it black – is left to the GC thread. One field in the handle area is used to implement the mark stack as a simple linked list. Listing 7.3 shows the implementation of putfield for reference fields.

Note that field and array access is implemented in hardware on JOP. Only write accesses to reference fields need to be protected by the write-barrier, which is implemented in software. During class linking all write operations to reference fields (putfield and putstatic when accessing reference fields) are replaced by a JVM internal bytecodes (e.g., putfield_ref) to execute the write-barrier code as shown before. The shown code is part of a special class

[8]Although the GC is a copying collector a mark stack is needed to perform the object copy in the GC thread and not by the mutator.

(com.jopdesign.sys.JVM) where Java bytecodes that are not directly implemented by JOP can be implemented in Java [123].

The methods of class Native are JVM internal methods needed to implement part of the JVM in Java. The methods are replaced by regular or JVM internal bytecodes during class linking. Methods getField(ref, index) and putField(ref, value, index) map to the JVM bytecodes getfield and putfield. The method rdMem() is an example of an internal JVM bytecode and performs a memory read. The null pointer check for putfield_ref is implicitly performed by the hardware implementation of getfield that is executed by Native.getField(). The hardware implementation of getfield triggers an exception interrupt when the reference is null. The implementation of the write-barrier shows how a bytecode is substituted by a special version (pufield_ref), but uses in the software implementation the hardware implementation of that bytecode (Naitve.putfield()).

In principle this write-barrier could also be implemented in microcode to avoid the expensive invoke of a Java method. However, the interaction with the GC, which is written in Java, is simplified by the Java implementation. As a future optimization we intend to inline the write-barrier code.

The collector runs in its own thread and the priority is assigned according to the deadline, which equals the period of the GC cycle. As the GC period is usually longer than the mutator task deadlines, the GC runs at the lowest priority. When a high priority task becomes ready, the GC thread will be preempted. Atomic operations of the GC are protected simply by turning the timer interrupt off.[9] Those atomic sections lead to release jitter of the real-time tasks and shall be minimized. It has to be noted that the GC protection with interrupt disabling is not an option for multiprocessor systems.

7.5 Evaluation

To evaluate the proposed real-time GC we execute a simple test application on JOP and measure heap usage and the release time jitter of high priority threads. The test setup consists of JOP implemented in an Altera Cyclone FPGA clocked at 100 MHz. The main memory is a 1 MB SRAM with an access time of two clock cycles. JOP is configured with a 4 KB method cache (a special form of instruction cache) and a 128 entry stack cache. No additional data cache is used.

[9]If interrupt handlers are allowed to change the object graph those interrupts also need to be disabled.

7.5.1 Scheduling Experiments

In this section we test an implementation of the concurrent-copy garbage collector on JOP. The tests are intended to get some confidence that the formulas for the collector periods are correct. Furthermore we visualize the actual heap usage of a running system.

The examples are synthetic benchmarks that emulate worst-case execution time (WCET) by executing a busy loop after allocation of the data. The WCET of the collector was measured to be 10.4 ms when executing it with scheduling disabled during one collection cycle for example 1 and 11.2 ms for example 2. We use 11 ms and 12 ms respectively as the WCET of the collector for the following examples[10].

Listing 7.4 shows our worker thread with the busy loop. The data is allocated at the start of the period and freed after the simulated execution. waitForNextPeriod blocks until the next release time for the periodic thread.

For the busy loop to simulate *real* execution time, and not elapsed time, the constant MIN_US has to be less than the time for two context switches, but larger than the execution time of one iteration of the busy loop. In this case only cycles executed by the busy loop are counted for the execution time and interruption due to a higher priority thread is not part of the execution time measurement.

In our example we use a concurrent-copy collector with a heap size (for both semi-spaces) of 100 KB. At startup the JVM allocates about 3.5 KB data. We incorporate[11] these 3.5 KB as static live data L_s.

Independent Threads

The first example consists of two threads with the properties listed in Table 7.1. T_i is the period, C_i the WCET, and a_i the maximum amount of memory allocated each period. Note that the period for the collector thread is also listed in the table although it is a result of the worker thread properties and the heap size.

With the periods T_i and the memory consumption a_i for the two worker threads we cal-

[10]It has to be noted that measuring execution time is not a safe method to estimate WCET values.

[11]The suggested handling of static data to be moved to *immortal* memory at mission start is not yet implemented.

```
public void run() {

    for (;;) {
        int [] n = new int[cnt];
        // simulate work load
        busy(wcet);
        n = null;
        waitForNextPeriod();
    }
}

final static int MIN_US = 10;

static void busy(int us) {

    int t1, t2, t3;
    int cnt;

    cnt = 0;
    // get the current time in us
    t1 = Native.rd(Const.IO_US_CNT);

    for (;;) {
        t2 = Native.rd(Const.IO_US_CNT);
        t3 = t2-t1;
        t1 = t2;
        if (t3<MIN_US) {
            cnt += t3;
        }
        if (cnt>=us) {
            return;
        }
    }
}
```

Listing 7.4: Example periodic thread with a busy loop

	T_i	C_i	a_i
τ_1	5 ms	1 ms	1 KB
τ_2	10 ms	3 ms	3 KB
τ_{GC}	77 ms	11 ms	

Table 7.1: Thread properties for experiment 1

culate the maximum period T_{GC} for the collector thread τ_{GC} by using Theorem 1

$$\begin{aligned} T_{GC} &\leq \frac{H_{CC} - 2\left(L_s + \sum_{i=1}^{n} a_i\right) - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \\ &\leq \frac{100 - 2(3.5 + 4) - 2 \cdot 4}{2\left(\frac{1}{5} + \frac{3}{10}\right)} \text{ms} \\ &\leq 77\text{ms} \end{aligned}$$

The priorities are assigned rate-monotonic [83] and we perform a quick schedulability check with the periods T_i and the WCETs C_i by calculation of the processor utilization U for all three threads

$$\begin{aligned} U &= \sum_{i=1}^{3} \left(\frac{C_i}{T_i}\right) \\ &= \frac{1}{5} + \frac{3}{10} + \frac{11}{77} \\ &= 0.643 \end{aligned}$$

which is less than the maximum utilization for three tasks

$$\begin{aligned} U_{max} &= m * (2^{\frac{1}{m}} - 1) \\ &= 3 * (2^{\frac{1}{3}} - 1) \\ &\approx 0.78 \end{aligned}$$

In Figure 7.4 the memory trace for this system is shown. The graph shows the free memory in one semi-space (the to-space, which is 50 KB) during the execution of the application. The individual points are recorded with time-stamps at the end of each allocation request.

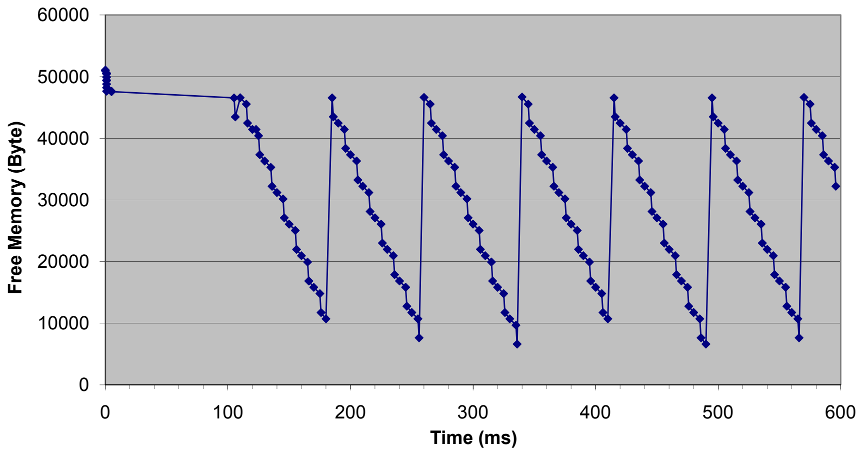


Figure 7.4: Free memory in experiment 1

In the first milliseconds we see allocation requests that are part of the JVM startup (most of it is static data). The change to the mission phase is delayed 100 ms and the first allocation from a periodic thread is at 105 ms. The collector thread also starts at the same time and the first semi-space flip can be seen at 110 ms (after one allocation from each worker thread). We see the 77 ms period of the collector in the jumps in the free memory graph after the flip. The different memory requests of two times 1 KB from thread τ_1 and one time 3 KB from thread τ_2 can be seen every 10 ms.

In this example the heap is used until it is almost full, but the application never runs out of memory and no thread misses a deadline. From the regular allocation pattern we also see that this collector runs concurrently. With a stop-the-world collector we would notice gaps of 10 ms (the measured execution time of the collector) in the graph.

Producer/Consumer Threads

For the second experiment we split our thread τ_1 to a producer thread τ_1 and a consumer thread τ_3 with a period of 30 ms. We assume after the split that the producer's WCET is halved to 500 us. The consumer thread is assumed to be more efficient when working on lager blocks of data than in the former example (C_3=2 ms instead of 6*500 μs). The rest of the setting remains the same (the worker thread τ_2). Table 7.2 shows the thread properties

	T_i	C_i	a_i
τ_1	5 ms	0.5 ms	1 KB
τ_2	10 ms	3 ms	3 KB
τ_3	30 ms	2 ms	
τ_{GC}	55 ms	12 ms	

Table 7.2: Thread properties for experiment 2

for the second experiment.

As explained in Section 7.2.3 we calculate the lifetime factor l_1 for memory allocated by the producer τ_1 with the corresponding consumer τ_3 with period T_3.

$$l_1 = \left\lceil \frac{2T_3}{T_1} \right\rceil = \left\lceil \frac{2 \times 30}{5} \right\rceil = 12$$

The maximum collector period T_{GC} is

$$\begin{aligned} T_{GC} &\leq \frac{H_{CC} - 2\left(L_s + \sum_{i=1}^{n} a_i l_i\right) - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \\ &\leq \frac{100 - 2(3.5 + 1 \cdot 12 + 3 + 0) - 2 \cdot 4}{2\left(\frac{1}{5} + \frac{3}{10} + \frac{0}{30}\right)} \text{ms} \\ &\leq 55\text{ms} \end{aligned}$$

We check the maximum processor utilization:

$$\begin{aligned} U &= \sum_{i=1}^{4} \left(\frac{C_i}{T_i}\right) \\ &= \frac{0.5}{5} + \frac{3}{10} + \frac{2}{30} + \frac{12}{55} \\ &= 0.685 \leq 4 * (2^{\frac{1}{4}} - 1) \approx 0.76 \end{aligned}$$

In Figure 7.5 the memory trace for the system with one producer, one consumer, and one independent thread is shown. Again, we see the 100 ms delayed mission start after the startup and initialization phase, in this example at about 106 ms. Similar to the former example the first collector cycle performs the flip a few milliseconds after the mission start.

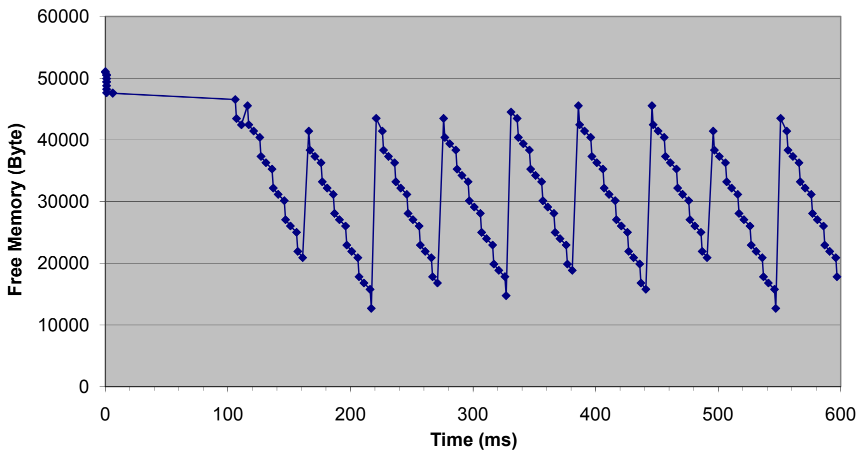


Figure 7.5: Free memory in experiment 2

We see the shorter collection period of 55 ms. The allocation pattern (two times 1 KB and one time 3 KB per 10 ms) is the same as in the former example as the threads that allocate the memory are still the same.

We have also run this experiment for a longer time than shown in Figure 7.5 to see if we find a point in the execution trace where the remaining free memory is less than the value at 217 ms. The pattern repeats and the observed value at 217 ms is the minimum.

7.5.2 Measuring Release Jitter

Our main concern on garbage collection in real-time systems is the blocking time introduced by the GC due to atomic code sections. This blocking time will be seen as release time jitter on the real-time threads. Therefore we want to measure this jitter.

Listing 7.5 shows how we measure the jitter. Method `run()` is the main method of the real-time thread and executed on each periodic release. Within the real-time thread we have no notion about the start time of the thread. As a solution we measure the actual time on the first iteration and use this time as first release time. In each iteration the expected time, stored in the variable `expected`, is incremented by the `period`. In each iteration (except the first one) the actual time is compared with the expected time and the maximum value of the difference is recorded.

```
public boolean run() {

    int t = Native.rdMem(Const.IO_US_CNT);
    if (! notFirst) {
        expected = t+period;
        notFirst = true;
    } else {
        int diff = t−expected;
        if (diff >max) max = diff;
        if (diff <min) min = diff;
        expected += period;
    }
    work();

    return true;
}
```

Listing 7.5: Measuring release time jitter

As noted before, we have no notion about the *correct* release times. We measure only relative to the first release. When the first release is delayed (due to some startup code or interference with a higher priority thread) we have a positive offset in expected. On an exact release in a later iteration the time difference will be negative (in diff). Therefore, we also record the minimum value for the difference between the actual time and the expected time. The maximum measured release jitter is the difference between max and min.

To provide a baseline we measure the release time jitter of a single real-time thread (plus an endless loop in the main method as an idle non-real-time background thread). No GC thread is scheduled. The code is similar to the code in Listing 7.5. A stop condition is inserted that prints out the minimum and maximum time differences measured after 1 million iterations.

Table 7.3 shows the measured jitter for different thread periods. We observed no jitter for periods of 100 μs and longer. At a period of 50 μs the scheduler introduces a considerable amount of jitter. From this measurement we conclude that 100 μs is the practical shortest period we can handle with our system. We will use this period for the high-priority real-time thread in the following measurement with an enabled GC.

Period	Jitter
200 μs	0 μs
100 μs	0 μs
50 μs	17 μs

Table 7.3: Release jitter for a single thread

Thread	Period	Deadline	Priority
τ_{hf}	100 μs	100 μs	5
τ_p	1 ms	1 ms	4
τ_c	10 ms	10 ms	3
τ_{log}	1000 ms	100 ms	2
τ_{gc}	200 ms	200 ms	1

Table 7.4: Thread properties of the test program

7.5.3 Measurements

The test application consisting of three real-time threads (τ_{hf}, τ_p, and τ_c), one logging thread τ_{log}, and the GC thread τ_{gc}. All three real-time threads measure the difference between the expected release time and the actual release time (as shown in Figure 7.5). The minimum and maximum values are recorded and regularly printed to the console by the logging thread τ_{log}. Table 7.4 shows the release parameters for the five threads. Priority is assigned deadline monotonic. Note that the GC thread has a shorter period than the logger thread, but a longer deadline. For our approach to work correctly the GC thread *must* have the lowest priority. Therefore all other threads with a longer period than the GC thread must be assigned a shorter deadline.

Thread τ_{hf} represents a high-frequency thread without dynamic memory allocation. This thread should observe minimal disturbance by the GC thread.

The threads τ_p and τ_c represent a producer/consumer pair that uses dynamically allocated memory for communication. The producer appends the data at a frequency of 1 kHz to a simple list. The consumer thread runs at 100 Hz and processes all currently available data in the list and removes them from the list. The consumer will process between 9 and 11 elements (depending on the execution time of the consumer and the thread phasing).

It has to be noted that this simple and common communication pattern cannot be imple-

Threads	Jitter
τ_{hf}	0 μs
τ_{hf}, τ_{log}	7 μs
τ_{hf}, τ_{log},τ_p,τ_c	14 μs
τ_{hf}, τ_{log},τ_p,τ_c,τ_{gc}	54 μs

Table 7.5: Jitter measured on a 100 MHz processor for the high priority thread in different configurations

mented with the scoped memory model of the RTSJ. First, to use a scope for communication, we have to keep the scope alive with a *wedge* thread [101] when data is added by the producer. We would need to notify this wedge thread by the consumer when all data is consumed. However, there is no single instant available where we can *guarantee* that the list is empty. A possible solution for this problem is described in [101] as *handoff* pattern. The pattern is similar to double buffering, but with an explicit copy of the data. The elegance of a simple list as buffer queue between the producer and the consumer is lost.

Thread τ_{log} is not part of the real-time systems simulated application code. Its purpose is to print the minimum and maximum differences between the measured and expected release times (see former section) of threads τ_{hf} and τ_p to the console periodically.

Thread τ_{gc} is a standard periodic real-time thread executing the GC logic. The GC thread period was chosen quite short in that example. A period in the range of seconds would be enough for the memory allocation by τ_p. However, to stress the interference between the GC thread and the application threads we artificially shortened the period.

As a first experiment we run only τ_{hf} and the logging thread τ_{log} to measure jitter introduced by the scheduler. The maximum jitter observed for τ_{hf} is 7 μs – the blocking time of the scheduler.

In the second experiment we run all threads except the GC thread. For the first 4 seconds we measure a maximum jitter of 14 μs for thread τ_{hf}. After those 4 seconds the heap is full and GC is necessary. In that case the GC behaves in a stop-the-world fashion. When a new object request cannot be fulfilled the GC logic is executed in the context of the allocating thread. As the bytecode new is itself in an atomic region the application is blocked until the GC finishes. Furthermore, the GC performs a conservative scan of all thread stacks. We measure a release delay of 63 ms for all threads due to the blocking during the full collection cycle. From that measurement we can conclude for the sample application and the available main memory: (a) the measured maximum period of the GC thread is in the range of 4 seconds; (b) the estimated execution time for one GC cycle is 63 ms. It has to be noted that measurement is not a substitution for static timing analysis. Providing WCET estimates for a GC cycle is a challenge for future work.

In our final experiment we enabled all threads. The GC is scheduled periodically at 200 ms as the lowest priority thread – the scenario we argue for. The GC logic is set into the concurrent mode on mission start. In this mode the thread stacks are not scanned for roots. Furthermore when an allocation request cannot be fulfilled the application is stopped. This radical stop is intended for testing. In a more tolerant implementation either an out-of-memory exception can be thrown or the requesting thread has to be blocked, its thread stack scanned and released when the GC has finished its cycle.

We ran the experiment for several hours and recorded the maximum release jitter of the real-time threads. For this test we used slightly different periods (prime numbers) to avoid the regular phasing of the threads. The harmonic relation of the original periods can lead to too optimistic measurements. The applications never ran out of memory. The maximum jitter observed for the high priority task τ_{hf} was 54 μs. The maximum jitter for task τ_p was 108 μs. This higher value on τ_p is expected as the execution interferes with the execution of the higher priority task τ_{hf}.

7.5.4 Discussion

With our measurements we have shown that quite short blocking times are achievable. Scheduling introduces a blocking time of about 7–14 μs and the GC adds another 40 μs resulting in a maximum jitter of the highest priority thread of 54 μs. In our first implementation we performed the object copy in pure Java, resulting in blocking times around 200 μs. To speedup the copy we moved this function to microcode. However, the microcoded *memcpy* still needs 18 cycles per 32-bit word copy. Direct support in hardware can lead to a copy time of 4–5 clock cycles per word.

The maximum blocking time of 54 μs on a 100 MHz processor is less than blocking times reported for other solutions.

Blocking time for Metronome (called pause times in the papers) is reported to be 6 ms [17] on a 500 MHz PowerPC at 50% CPU utilization. Those large blocking times are due to the scheduling of the GC at the highest priority with a polling based yield within the GC thread. A fairer comparison is against the *jitter* of the pause time. In [16] the variation of the pause time is given between 500 μs and 2.4 ms on a 1 Ghz machine. It should be noted that Metronome is a GC intended for mixed real-time systems whereas we aim only for hard real-time systems.

Robertz performed a similar measurement as we did for his thesis [114] with a time-triggered GC on a 350 MHz PowerPC. He measured a maximum jitter of 20 μs ($\pm$10 μs) for a high priority task with a period of 500 μs.

It has to be noted that our experiment is a small one and we need more advanced real-time applications for the evaluation of real-time GC. The problem is that it is hard to find even static based real-time application benchmarks (at least applications written for safety critical Java). Running standard benchmarks that measure average case performance (e.g., SPEC jvm98) is not an option to evaluate a real-time collector.

7.6 Analysis

To integrate GC into the WCET and scheduling analysis we need to know the worst-case memory consumption including the maximum lifetime of objects and the WCET of the collector itself.

7.6.1 Worst Case Memory Consumption

Similar to the necessary WCET analysis of the tasks that make up the real-time system, a worst case memory allocation analysis of the tasks is necessary. For objects that are not shared between tasks this analysis can be performed by the same methods known from the WCET analysis. We have to find the worst-case program path that allocates the maximum amount of memory.

The analysis of memory consumption by objects that are shared between tasks for communication is more complex as an inter-task analysis is necessary.

7.6.2 WCET of the Collector

For the schedulability analysis the WCET of the collector has to be known. The collector performs following steps[12]:

1. Traverse the live object graph

2. Copy the live data

3. Initialize the free memory

[12]These steps can be distinct steps as in the mark-compact collector or interleaved as in the concurrent-copy collector.

The execution time of the first step depends on the maximum amount of live data and the number of references in each object. The second step depends on the size of the live objects. The last step depends on the size of the memory that gets freed during the collection cycle. For a concurrent-copy collector this time is constant as a complete semi-space gets initialized to zero. It has to be noted that this initialization could also be done at the allocation of the objects (as the LTMemory from the RTSJ implies). However, initialization in the collector is more efficient and the necessary time is easier to predict.

The maximum allocated memory and the type of the allocated objects determine the control flow (the flow facts) of the collector. Therefore, this information has to be to incorporate into WCET analysis of the collector thread.

7.7 Summary

In this chapter we have presented a real-time garbage collector in order to benefit from a more dynamic programming model for real-time applications. The collector is incremental and scheduled as a normal real-time thread and, according to its deadline, assigned the lowest priority in the system. The restrictions from the SCJ programming model and the low priority result in two advantages: (a) avoidance of stack root scanning and (b) short blocking time of high priority threads. At 100 MHz we measured 40 μs maximum blocking time introduced by the GC thread.

To guarantee that the applications will not run out of memory, the period of the collector thread has to be short enough. We provided the maximum collector periods for a mark-compact collector type and a concurrent-copy collector. We have also shown how a longer lifetime due to object sharing between threads can be incorporated into the collector period analysis.

A critical operation for a concurrent, compacting GC is the atomic copy of large arrays. JOP has been extended by a copy unit that can be interrupted. This unit is integrated with the memory access unit and redirects the access to either fromspace or tospace depending on the array/field index and the value of the copy pointer.

7.8 Further Reading

Garbage collection was first introduced for list processing systems (LISP) in the 1960s. The first collectors were *stop-the-world* collectors that are called when a request for a new element can not be fulfilled. The collector, starting from pointers known as the root set, scans the whole graph of reachable objects and marks these objects live. In a second phase the collector *sweeps* the heap and adds unmarked objects to the free list. On the marked objects, which are live, the mark is reset in preparation for the next cycle.

However, this simple sweep results in a fragmented heap which is an issue for objects of different sizes. An extension, called *mark-compact*, moves the objects to compact the heap instead of the sweep. During this compaction all references to the moved objects are updated to point to the new location.

Both collectors need a stack during the marking phase that can grow in the worst-case up to the number of live objects. Cheney [32] presents an elegant way how this mark stack can be avoided. His GC is called *copying-collector* and divides the heap into two spaces: the *to-space* and the *from-space*. Objects are moved from one space to the other as part of the scan of the object graph.

However, all the described collectors are still stop-the-world collectors. The pause time of up to seconds in large interactive LISP applications triggered the research on incremental collectors that distribute collection work more evenly [142, 38, 19]. These collectors were sometimes called *real-time* although they do not fulfill hard real-time properties that we need today. A good overview of GC techniques can be found in [72] and in the GC survey by Wilson [153].

Baker [19] extends Cheneys [32] copying collector for incremental GC. However, it uses an expensive read barrier that moves the object to the to-space as part of the mutator work. Baker proposes the *Treadmill* [20] to avoid copying. However, this collector works only with objects of equal size and still needs an expensive read barrier.

In [116] a garbage-collected memory module is suggested to provide a real-time collector. A worst-case delay time of 1μs is claimed without giving the processor speed.

Metronome is a collector intended for soft real-time systems [18]. Non real-time applications are used (SPECjvm98) in the experiments. They propose a collector with constant utilization to meet real-time requirements. However, utilization is *not* a real-time measure per se; it should be schedulability or response time instead. In contrast to our proposal the GC thread is scheduled at the highest priority in short periods. To ensure that, despite the high priority of the GC thread, mutator threads will be scheduled, the GC thread runs only for a fraction of time within a time window. This fraction and the size of the time window can be adjusted for different work loads.

Although not mandated, all commercial and academic implementations of the RTSJ [140, 26, 14, 10] and related real-time Java systems [6] also contain a real-time garbage collector.

In [95] two collectors based on [38] and [20] are implemented on a multithreaded microcontroller. Higuera suggests in [65] the use of hardware features from picoJava to speed up RTSJ memory region protection and garbage collection.

The work closest to our scheduling analysis is presented in [113]. The authors provide an upper bound of the GC cycle as[13]

$$T_{GC} \leq \frac{\frac{H-L_{max}}{2} - \sum_{i=1}^{n} a_i}{\sum_{i=1}^{n} \frac{a_i}{Ti}}$$

Although stated that this bound "is thus not dependent of any particular GC algorithm", the result applies only for single heap GC algorithms (e.g. mark-compact) and not for a copying collector. A value for L_{max} is not given in the paper. If we use our value of $L_{max} = \sum_{i=1}^{n} a_i$ the result is

$$T_{GC} \leq \frac{H - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}}$$

This result is the same as in our finding (see Theorem 1) for the mark-compact collector. No analysis is given how objects with longer lifetime and static objects can be incorporated.

[13]We use our symbols in the equation for easier comparison to our finding.

8 Low-level I/O

The following section describes the low-level mechanism for I/O access and interrupts on JOP. As JOP is a Java processor no native functions (usually written in C) are available to access I/O directly or use C written interrupt handler. We need access to those low-level functionality from Java for an embedded system. In the following a hardware abstraction layer (HAL) in Java is proposed, where I/O devices are mapped to Java objects and interrupt handlers can be implemented in Java as Runnable. This section is based on [133] and [74].

8.1 Hardware Objects

Hardware objects map object fields to device registers. Therefore, field access with bytecodes putfield and getfield accesses device registers. With a correct class that represents a device, access to it is safe – it is not possible to read or write to an arbitrary memory address. A memory area (e.g., a video frame buffer) represented by an array is protected by Java's array bounds check. Representing I/O devices as first class objects has following benefits:

Object-oriented: An object representing a hardware device is the most natural integration into an OO language

Safe: The safety of Java is not compromised. We can access only those device registers that are represented by the class definition

Efficient Device register access is performed by single bytecodes getfield and putfield. We avoid expensive native calls.

8.1.1 An Example

Let us consider a simple I/O device, e.g. a parallel input/output (PIO) device – a common device in embedded systems for control applications. The PIO provides an interface between I/O registers and I/O pins. The host captures data on the input pins with a register read and drives data to the output pins with a register write. The example PIO contains two

```
typedef struct {
    int data;
    int control;
} parallel_port;

#define PORT_ADDRESS 0x10000;

int inval, outval;
parallel_port *mypp;
mypp = (parallel_port *) PORT_ADDRESS;
...
inval = mypp->data;
mypp->data = outval;
```

Listing 8.1: Definition and usage of a parallel port in C

registers: the *data register* and the *control register*. Writing to the data register stores the value into a register that drives the output pins. Reading from the data register returns the value that is present at the input pins.

The control register configures the direction for each PIO pin. When bit *n* in the control register is set to 1, pin *n* drives out the value of bit *n* of the data register. A 0 at bit *n* in the control register configures pin *n* as input pin. At reset the port is usually configured as input port – a safe default configuration.[1]

When the I/O address space is memory mapped, such a parallel port is represented in C as a structure and a constant for the address. This definition is part of the board level configuration. Listing 8.1 shows the parallel port example. The parallel port is directly accessed via a pointer in C. For a system with a distinct I/O address space (e.g. x86) access to the device registers is performed via distinct machine instructions. Those instructions are represented by C functions that take the address as argument which is not type-safe.

This simple representation of memory mapped I/O devices in C is both efficient and dangerous. It is efficient as the access via pointers compiles to simple load and store instructions. It is dangerous as wrong pointer manipulation can result in erroneous I/O or memory access. This issue is inherent to C and C programmers are (hopefully) aware of it. A major aspect that makes Java a safer[2] language than C is the avoidance of pointers. A

[1]Direction output can result in a short circuit between the I/O pin and the external device when the logic levels are different.

[2]In this context we consider the safety aspect as safe from programming errors.

```
public final class ParallelPort {
    public volatile int data;
    public volatile int control;
}

int inval, outval;
myport = JVMMagic.getParallelPort();
...
inval = myport.data;
myport.data = outval;
```

Listing 8.2: The parallel port device as a simple Java class

pointer is in effect an address to data manipulated as data – an abstraction that resembles more assembler programming than programming in a high-level language.

On a standard JVM, native functions, written in C or C++, allow the low-level access to devices from Java. This approach is not object-oriented (OO) and incurs a lot of overheads; parameters and return values have to be converted between Java and C. In an OO language the most natural way to represent an I/O device is as an object. Listing 8.2 shows a class definition for our simple parallel port.

The class ParallelPort is equivalent to the structure definition for C in Listing 8.1. Reference myport points to the hardware object. The device register access is similar to the C version. The main difference to the C structure is that the access requires no pointers. To provide this convenient representation of I/O devices as objects we just need some *magic* in the JVM and a mechanism to *create* the device object and receive a reference to it.

8.1.2 Definition

All hardware classes have to extend the abstract class HardwareObject (see Lising 8.3). This empty class serves as type marker. Some implementations use it to distinguish between plain objects and hardware objects for the field access. The package visible only constructor disallows creation of hardware objects by the application code that resides in a different package.

Listing 8.4 shows a class representing a serial port with a status register and a data register. The status register contains flags for receive register full and transmit register empty; the data register is the receive and transmit buffer. Additionally, we define device specific constants (bit masks for the status register) in the class for the serial port. All fields represent

```
public abstract class HardwareObject {
    HardwareObject() {};
}
```

Listing 8.3: The marker class for hardware objects

```
public final class SerialPort extends HardwareObject {

    public static final int MASK_TDRE = 1;
    public static final int MASK_RDRF = 2;

    public volatile int status;
    public volatile int data;

    public void init (int baudRate) {...}
    public boolean rxFull() {...}
    public boolean txEmpty() {...}
}
```

Listing 8.4: A serial port class with device methods

device registers that can change due to activity of the hardware device. Therefore, they must be declared volatile.

In this example we have included some convenience methods to access the hardware object. However, for a clear separation of concerns, the hardware object represents only the device state (the registers). We do not add instance fields to represent additional state, i.e., mixing device registers with heap elements. We cannot implement a complete device driver within a hardware object; instead a complete device driver owns a number of private hardware objects along with data structures for buffering, and it defines interrupt handlers and methods for accessing its state from application processes. For device specific operations, such as initialization of the device, methods in hardware objects are useful.

8.1.3 Access Control

Usually each device is represented by exactly one hardware object (see Section 8.1.7). However, there might be use cases where this restriction should be relaxed. Consider a device

```
public final class SysCounter extends HardwareObject {

    public volatile int counter;
    public volatile int timer;
    public volatile int wd;
}

public final class AppCounter extends HardwareObject {

    public volatile int counter;
    private volatile int timer;
    public volatile int wd;
}
```

Listing 8.5: System and application classes with visibility protection for a single hardware device

where some registers should be accessed by system code only and some other by application code. In JOP there is such a device: a system device that contains a 1 MHz counter, a corresponding timer interrupt, and a watchdog port. The timer interrupt is programmed relative to the counter and used by the real-time scheduler – a JVM internal service. However, access to the counter can be useful for the application code. Access to the watchdog register is required from the application level. The watchdog is used for a sign-of-life from the application. If not triggered every second the complete system is restarted. For this example it is useful to represent one hardware device by two *different* classes – one for system code and one for application code. We can protect system registers by private fields in the hardware object for the application. Listing 8.5 shows the two class definitions that represent the same hardware device for system and application code respectively. Note how we changed the access to the timer interrupt register to private for the application hardware object.

Another option, shown in Listing 8.6, is to declare all fields private for the application object and use setter and getter methods. They add an abstraction on top of hardware objects and use the hardware object to implement their functionality. Thus we still do not need to invoke native functions.

```
public final class AppGetterSetter extends HardwareObject {

    private volatile int counter;
    private volatile int timer;
    private volatile int wd;

    public int getCounter() {
        return counter;
    }

    public void setWd(boolean val) {
        wd = val ? 1 : 0;
    }
}
```

Listing 8.6: System and application classes with setter and getter methods

8.1.4 Using Hardware Objects

Use of hardware objects is straightforward. After obtaining a reference to the object all that has to be done (or can be done) is to read from and write to the object fields. Listing 8.7 shows an example of client code. The example is a *Hello World* program using low-level access to a serial port via a hardware object.

8.1.5 Hardware Arrays

For devices that use DMA (e.g., video frame buffer, disk, and network I/O buffers) we map that memory area to Java arrays. Arrays in Java provide access to raw memory in an elegant way: the access is simple and safe due to the array bounds checking done by the JVM. Hardware arrays can be *used* by the JVM to *implement* higher-level abstractions from the RTSJ such as RawMemory or scoped memory [150].

8.1.6 Garbage Collection

Interaction between the garbage collector (GC) and hardware objects needs to be crafted into the JVM. We do not want to *collect* hardware objects. The hardware object should not

```
import com.jopdesign.io.∗;

public class Example {

    public static void main(String[] args) {

        BaseBoard fact = BaseBoard.getBaseFactory();
        SerialPort sp = fact.getSerialPort ();

        String hello = ”Hello World!”;

        for (int i=0; i<hello.length(); ++i) {
            // busy wait on transmit buffer empty
            while ((sp.status & SerialPort.MASK_TDRE) == 0)
                ;
            // write a character
            sp.data = hello.charAt(i);
        }
    }
}
```

Listing 8.7: A 'Hello World' example with low-level device access via a hardware object

be scanned for references.[3] This is permissible when only primitive types are used in the class definition for hardware objects – the GC scans only reference fields. To avoid collecting hardware objects, we *mark* the object to be skipped by the GC. The type inheritance from HardwareObject can be used as the marker.

For JOP we only define hardware objects with primitive data fields. Therefore, the hardware objects can be ignored by the GC. The GC scans only objects where the handles are in the handle area. When the GC is about to mark and scan an object it first checks if the reference points into the handle area. If not, the reference is skipped by the GC. All handles that lie outside of this area are ignored by the GC. The handles for the hardware objects are allocated in a special memory area (see Section 8.1.9) that is ignored by the GC. The same mechanism is already used by the JVM for some runtime data structures (notable string constants) that reside in their own memory area.

[3]If a hardware coprocessor, represented by a hardware object, itself manipulates an object on the heap and holds the only reference to that object it has to be scanned by the GC.

Handles which are not touched by the GC do not need the additional GC info fields. We need only two fields: (1) the indirection field and (2) the class reference or array length field.

8.1.7 Hardware Object Creation

The idea to represent each device by a single object or array is straightforward, the remaining question is: How are those objects created? An object that represents a device is a typical Singleton [46]. Only a single object should map to one instance of a device. Therefore, hardware objects cannot be instantiated by a simple new: (1) they have to be mapped by some JVM mechanism to the device registers and (2) each device instance is represented by a single object.

Each device object is created by its own factory method. The collection of these methods is the board configuration, which itself is also a Singleton (the application runs on a single board). The Singleton property of the configuration is enforced by a class that contains only static methods. Listing 8.8 shows an example for such a class. The class IOSystem represents a system with three devices: a parallel port, as discussed before to interact with the environment, and two serial ports: one for program download and one which is an interface to a GPS receiver.

This approach is simple, but not very flexible. Consider a vendor who provides boards in slightly different configurations (e.g., with different number of serial ports). With the above approach each board requires a different (or additional) IOSystem class that lists all devices. A more elegant solution is proposed in the next section.

8.1.8 Board Configurations

Replacing the static factory methods by instance methods avoids code duplication; inheritance then gives configurations. With a factory object we represent the common subset of I/O devices by a base class and the variants as subclasses.

However, the factory object itself must still be a Singleton. Therefore the board specific factory object is created at class initialization and is retrieved by a static method. Listing 8.9 shows an example of a base factory and a derived factory. Note how getBaseFactory() is used to get a single instance of the factory. We have applied the idea of a factory two times: the first factory generates an object that represents the board configuration. That object is itself a factory that generates the objects that interface to the hardware device.

The shown example base factory represents the minimum configuration with a single serial port for communication (mapped to System.in and System.out) represented by a Seri-

```
package com.board−vendor.io;

public class IOSystem {

    // some JVM mechanism to create the hardware objects
    private static ParallelPort pp = jvmPPCreate();
    private static SerialPort sp = jvmSPCreate(0);
    private static SerialPort gps = jvmSPCreate(1);

    public static ParallelPort getParallelPort () {
        return pp;
    }

    public static SerialPort getSerialPort () {..}
    public static SerialPort getGpsPort() {..}
}
```

Listing 8.8: A factory with static methods for Singleton hardware objects

alPort. The derived configuration ExtendedBoard (listing 8.10) contains an additional serial port for a GPS receiver and a parallel port for external control.

Furthermore, we show in those examples a different way to incorporate the JVM mechanism in the factory: we define well known constants (the memory addresses of the devices) in the factory and let the native function jvmHWOCreate() return the correct device type.

8.1.9 Implementation

In this subsection the internals of the hardware object creation are described. Just to use hardware objects this section can be skipped. To create new types of hardware objects and the companion factory this section contains the needed details.

In JOP, objects and arrays are referenced through an indirection called *handle*. This indirection is a lightweight read barrier for the compacting real-time GC (see Chapter 7). All handles for objects in the heap are located in a distinct memory region, the handle area. Besides the indirection to the *real* object the handle contains auxiliary data, such as a reference to the class information, the array length, and GC related data. Figure 8.1 shows an example with a small object that contains two fields and an integer array of length 4. The object and the array on the heap just contain the data and no additional hidden fields.

```
public class BaseBoard {

    private final static int SERIAL_ADDRESS = ...;
    private SerialPort serial;
    BaseBoard() {
        serial = (SerialPort) jvmHWOCreate(SERIAL_ADDRESS);
    };
    static BaseBoard single = new BaseBoard();
    public static BaseBoard getBaseFactory() {
        return single;
    }
    public SerialPort getSerialPort() { return serial; }

    // here comes the JVM internal mechanism
    Object jvmHWOCreate(int address) {...}
}
```

Listing 8.9: A base class of a hardware object factory

This object layout greatly simplifies our object to device mapping. We just need a handle where the indirection points to the memory mapped device registers instead of into the heap. This configuration is shown in the upper part of Figure 8.1. Note that we do not need the GC information for the hardware object handles. The factory, which creates the hardware objects, implements this indirection.

As described in Section 8.1.7 we do not allow applications to create hardware objects; the constructor is private (or package visible).[4] Listing 8.11 shows part of the base hardware object factory that creates the hardware object SerialPort and SysDevice. Two static fields (SP_PTR and SP_MTAB) are used to store the handle to the serial port object. The first field is initialized with the base address of the I/O device; the second field contains a pointer to the class information.[5] The address of the static field SP_PTR is returned as the reference to the serial port object.

[4]For creation of hardware objects with new we would need to change the implementation of bytecode new to distinguish between normal heap allocated objects and hardware objects. In the implementation on JOP the hardware object constructor is package visible to allow the factory to create a plane object of that type.

[5]In JOP's JVM the class reference is a pointer to the method table to speed-up the invoke instruction. Therefore, the name is XX_MTAB.

```
public class ExtendedBoard extends BaseBoard {

    private final static int GPS_ADDRESS = ...;
    private final static int PARALLEL_ADDRESS = ...;
    private SerialPort gps;
    private ParallelPort parallel ;
    ExtendedBoard() {
        gps = (SerialPort) jvmHWOCreate(GPS_ADDRESS);
        parallel = ( ParallelPort ) jvmHWOCreate(PARALLEL_ADDRESS);
    };
    static ExtendedBoard single = new ExtendedBoard();
    public static ExtendedBoard getExtendedFactory() {
        return single;
    }
    public SerialPort getGpsPort() { return gps; }
    public ParallelPort getParallelPort () { return parallel ; }
}
```

Listing 8.10: An extended class of a hardware object factory for a board variation

```
public class IOFactory {
    private SerialPort sp;
    private SysDevice sys;

    // Handles should be the first static fields !
    private static int SP_PTR;
    private static int SP_MTAB;
    private static int SYS_PTR;
    private static int SYS_MTAB;

    IOFactory() {
        sp = (SerialPort) makeHWObject(new SerialPort(), Const.IO_UART1_BASE, 0);
        sys = (SysDevice) makeHWObject(new SysDevice(), Const.IO_SYS_DEVICE, 1);
    };
    // that has to be overridden by each sub class to get the correct cp
    private static Object makeHWObject(Object o, int address, int idx) {
        int cp = Native.rdIntMem(Const.RAM_CP);
        return JVMHelp.makeHWObject(o, address, idx, cp);
    }
```

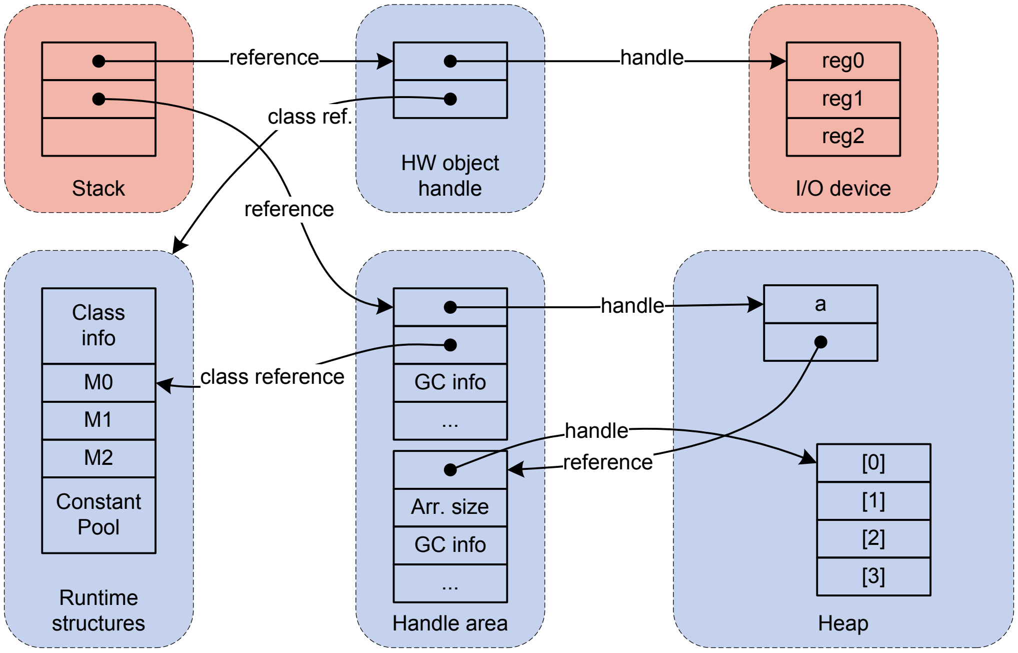


Figure 8.1: Memory layout of the JOP JVM

```
    private static IOFactory single = new IOFactory();
    public static IOFactory getFactory() {
        return single;
    }

    public SerialPort getSerialPort() { return sp; }
    public SysDevice getSysDevice() { return sys; }
}
```

Listing 8.11: Simplified version of the JOP base factory

The class reference for the hardware object is obtained by creating a *normal* instance of SerialPort with new on the heap and copying the pointer to the class information. To avoid using native methods in the factory class we delegate JVM internal work to a helper class in the JVM system package as shown in Listing 8.12. That helper method returns the address of the static field SP_PTR as reference to the hardware object. All methods in class

```
public static Object makeHWObject(Object o, int address, int idx, int
cp) {
    int ref = Native.toInt(o);
    int pcl = Native.rdMem(ref+1);
    int p = Native.rdMem(cp-1);
    p = Native.rdMem(p+1);
    p += idx*2;
    Native.wrMem(address, p);
    Native.wrMem(pcl, p+1);
    return Native.toObject(p);
}
```

Listing 8.12: The helper method in the system class JVMHelp for the hardware object creation

Native, a JOP system class, are *native*[6] methods for low-level functions – the code we want to avoid in application code. Method toInt(Object o) defeats Java's type safety and returns a reference as an int. Method toObject(int addr) is the inverse function to map an address to a Java reference. Low-level memory access methods are used to manipulate the JVM data structures.

To disallow the creation with new in normal application code, the visibility is set to package. The package visibility of the hardware object constructor is a minor issue. To access private static fields of an arbitrary class from the system class we had to change the runtime class information: we added a pointer to the first static primitive field of that class. As addresses of static fields get resolved at class linking, no such reference was needed so far.

8.1.10 Legacy Code

Before the implementation of hardware objects, the access to I/O devices was performed with memory read and write methods. Those methods are Native.rdMem() and Native.wrMem(). Due to historical reasons[7] the same Methods also exist as Native.rd() and Native.wr().

Those native methods are mapped to a system bytecode and perform direct memory access – not a save abstraction at all. However, there exists still some Java code that uses

[6]There are no *real* native functions in JOP – bytecode is the native instruction set. The very few native methods in class Native are replaced by special, unused bytecodes during class linking.

[7]In an older version of JOP I/O and memory had different busses.

those public visible methods. Those methods are depreciated and new device drivers shall use hardware objects.

8.2 Interrupt Handlers

Interrupts are notifications from hardware components to the processor. As a response to the interrupt signal some method needs to be invoked and executed. To allow implementation of first-level interrupt handler (IH) in Java we map interrupt requests to invocations of the run() method of a Runnable. Listing 8.13 shows an example of such an interrupt handler and how it is registered.

```
public class InterruptHandler implements Runnable {

    public static void main(String[] args) {

        // get factory
        InterruptHandler ih = new InterruptHandler();
        fact.registerInterruptHandler(1, ih);

        // enable interrupt 1
        fact.enableInterrupt(1);

        // start normal work
    }

    public void run() {
        // do the first level interrupt handler work
    }

}
```

Listing 8.13: A simple first-level interrupt handler in Java

8.2.1 Synchronization

When an interrupt handler is invoked, it starts with global interrupt disabled. The global interrupt mask is enabled again when the interrupt handler returns. On the uniprocessor version of JOP the monitor is also implemented by simply masking all interrupts. Therefore, those critical sections cannot be interrupted and synchronization between the interrupt handler and a normal thread is fulfilled. For a CMP version of JOP synchronization is performed via a global hardware lock. As the CMP system offers true concurrency, the IH has to protect the access to shared data when the handler and the thread are located on different cores.

A better approach for data sharing between a first-level IH and a device driver task (or second-level IH) is the usage of non-blocking queues. Listing 8.14 shows an example of such a bonded, non-blocking buffer for single reader and single writer communication. The class Buffer is part of the package rtlib.

```
public class Buffer {

    private volatile int [] data;
    private volatile int rdPtr;
    private volatile int wrPtr;

    public Buffer(int size) {
        data = new int[size+1];
        rdPtr = wrPtr = 0;
    }
    public boolean empty() {
        return rdPtr==wrPtr;
    }
    public boolean full () {
        int i = wrPtr+1;
        if (i>=data.length) i=0;
        return i==rdPtr;
    }
    public int read() {
        int i =rdPtr;
        int val = data[i++];
        if (i>=data.length) i=0;
        rdPtr = i;
        return val;
    }
    public void write(int val) {
        int i = wrPtr;
        data[i++] = val;
        if (i>=data.length) i=0;
        wrPtr = i;
    }
    public int cnt() {
        int i = wrPtr−rdPtr;
        if (i<0) i+=data.length;
        return i;
    }
    public int free() {
        return data.length−1−cnt();
    }
    public int checkedRead() {
        if (empty()) return −1;
        return read();
    }
    public boolean checkedWrite(int val) {
        if (full()) return false;
        write(val);
        return true;
    }
}
```

Listing 8.14: A non-blocking integer buffer for a single reader and a single writer. Classical usage is in an interrupt handler.

The buffer data is manipulated by the reader and writer. The size of the buffer is determined by the constructor. The read pointer rdPtr is only manipulated by the reader, the write pointer wrPtr only by the writer. The buffer is empty when rdPtr equals wrPtr and full when wrPtr+1 equals the rdPtr. In method full() two points are notable: (1) the usage of the local variable to get an atomic snapshot of the wrPtr; and (2) the usage of $>=$ instead of $=$ for easier data-flow analysis [104].

The methods read() and write() perform an unchecked read and write from the buffer – the unchecked version is for performance reason. Therefore, before invoking those methods the buffer fill state has to be checked with emtpy() and full(). If read is performed on an empty buffer, the buffer is corrupted – old, alread read data appears as new data. Write to a full buffer drops all former data. The fill stat of the buffer can be queried with cnt() and free(). Method checkedRead() reads one entry from the buffer or returns -1 if the buffer is empty. Method checkedWrite() write one entry into the buffer when not full and returns true if the write operation was successfully.

An object oriented version of this single reader/writer queue is available in class SRSWQueue. Those queues are also handy for communication between threads as blocking can be avoided. The redesigned TCP/IP stack ejip uses those non-blocking queues for communication of network packets between different protocol layers.

8.2.2 Interrupt Handler Registration

Interrupt handlers can be registered for a interrupt number *n*. On the CMP system the interrupt is registered on the core where the method is invoked. Following methods are available in the factory class for interrupt handler registration/deregistration and enable/disable of a specific interrupt.

```
public void registerInterruptHandler(int nr, Runnable logic) { }
public void deregisterInterruptHandler(int nr) { }
public void enableInterrupt(int nr) { }
public void disableInterrupt(int nr) { }
```

Interrupt 0 has a special meaning as it is the reprogrammable timer interrupt for the scheduler. On the transition to the mission phase (startMission()) a scheduler, which is a simple Runable, is registered for each core for the timer interrupt.

8.2.3 Implementation

The original JOP [123, 130] was a very puristic hard real-time processor. There existed only one interrupt – the programmable timer interrupt as time is the primary source for hard real-time events. All I/O requests were handled by periodic threads that poll for pending input data or free output buffers. However, to allow a more flexible programming model additional hardware interrupts are now available.

On a pending interrupt (or exception generated by the hardware) a special bytecode is inserted into the instruction stream (see Section 4.3.5. This approach keeps the interrupt completely transparent to the core pipeline. The special bytecode that is unused by the JVM specification [82] is used to invoke the special method interrupt() from the JVM helper class JVMHelp.

The implemented interrupt controller (IC) is priority based. The number of interrupt sources can be configured. Each interrupt can be triggered in software by a IC register write as well. There is one global interrupt enable and each interrupt line can be enabled or disabled locally. The interrupt is forwarded to the bytecode/microcode translation stage with the interrupt number. When accepted by this stage, the interrupt is acknowledged and the global enable flag cleared. This feature avoids immediate handling of an arriving higher priority interrupt during the first part of the handler. The interrupts have to be enabled again by the handler at a *convenient* time. All interrupts are mapped to the same special bytecode. Therefore, we perform the dispatch of the correct handler in Java. On an interrupt the static method interrupt() from a system internal class gets invoked. The method reads the interrupt number and performs the dispatch to the registered Runnable as illustrated below. Note, how a hardware object of type SysDevice is used to read the interrupt number.

```
static Runnable ih[] = new Runnable[Const.NUM_INTERRUPTS];
static SysDevice sys = IOFactory.getFactory().getSysDevice();

static void interrupt () {
    ih[sys.intNr].run();
}
```

The timer interrupt, used for the real-time scheduler, is located at index 0. The scheduler is just a plain interrupt handler that gets registered at mission start at index 0. At system startup, the table of Runnables is initialized with dummy handlers. The application code provides the handler via a class that implements Runnable and registers that class for an interrupt number.

For interrupts that should be handled by an event handler under the control of the scheduler, the following steps need to be performed on JOP:

1. Create a SwEvent with the correct priority that performs the second level interrupt handler work
2. Create a short first level interrupt handler as Runnable that invokes fire() of the corresponding software event handler
3. Register the first level interrupt handler and start the real-time scheduler

8.2.4 An Example

The system device (sc_sys.vhd) contains input lines for external interrupts (in port io_int). The number of interrupt lines is configurable with num_io_int. Each hardware interrupt can also be triggered by a write into the system device. Listing 8.15 shows registering and using a first level interrupt handler. The interrupt is triggered in software by the main thread.

```
public class InterruptHandler implements Runnable {

    public static void main(String[] args) {

        IOFactory fact = IOFactory.getFactory();
        SysDevice sys = fact.getSysDevice();

        InterruptHandler ih = new InterruptHandler();
        fact . registerInterruptHandler (1, ih );

        // enable software interrupt 1
        fact .enableInterrupt (1);

        for ( int i=0; i<20; ++i) {
            Timer.wd();
            int t = Timer.getTimeoutMs(200);
            while (!Timer.timeout(t)) ;
            // trigger a SW interrupt via the system HW object
            System.out.println(”Trigger”);
            sys.intNr = 1;
            if ( i==10) {
                fact . disableInterrupt (1);
            }
        }
    }

    public void run() {
        System.out.println(” Interrupt fired !”);
    }
}
```

Listing 8.15: Interrupt register/deregister methods in the factory class

8.3 Standard Devices

A minimum version of JOP consists of two standard devices: the system device and a UART device for program download and as a representation of System.ou.

8.3.1 The System Device

The system device contains all the logic for interrupts, CMP interaction, timers, and the watchdog control. The registers definition is shown in Table 8.1.

8.3.2 The UART

The UART contains a contral/status register and a data read/write register is shown in Table 8.2.

Address	Read	Write
0	Clock counter	Interrupt enable
1	Counter in μs	Timer interrupt in μs
2	Interrupt number	SW interrupt
3	—	Watchdog
4	Exception reason	Generate exception
5	Lock request status	Lock request
6	CPU ID	—
7	Polled in JVM startup	Start CMP
8	—	Interrupt mask
9	—	Clear pending interrupts
11	Nr. CPUs	—

Table 8.1: Registers in the system device.

Address	Read	Write
0	status	control
1	receive data	transmit buffer

Table 8.2: Registers in the UART device.

9 The SimpCon Interconnect

SimpCon [127] is the main interconnection interface used for JOP. The I/O modules and the main memory are connected via this standard. In the following chapter an introduction to SimpCon is presented.

The VHDL files in vhdl/scio are SimpCon I/O components (e.g. sc_uart.vhd is a simple UART) and SimpCon I/O configurations. The I/O configurations define the I/O devices and the address mapping for a JOP system. All those configurations start with scio_. The I/O components start with sc_. Configuration scio_min contains the minimal I/O components for a JOP system: the system module sc_sys.vhd and a UART sc_uart.vhd for program download and basic communication (System.in and System.out).

The system module sc_sys.vhd contains the clock counter, the μs counter, timer interrupt, the SW interrupt, exception interrupts, the watchdog port, and the connection to the multiprocessor synchronization unit (cmpsync.vhd).

In directory vhdl/memory the memory controller mem_sc.vhd is a SimpCon master that can be connected to various SRAM memory controllers sc_sram*.vhd. Other memory controller (e.g. the free Altera SDRAM interface) can be connected via SimpCon bridges to Avalon, Wishbone, and AHB slaves (available in vhdl/simpcon).

9.1 Introduction

The intention of the following SoC interconnect standard is to be simple and efficient with respect to implementation resources and transaction latency.

SimpCon is a fully synchronous standard for on-chip interconnections. It is a point-to-point connection between a master and a slave. The master starts either a read or write transaction. Master commands are single cycle to free the master to continue on internal operations during an outstanding transaction. The slave has to register the address when needed for more than one cycle. The slave also registers the data on a read and provides it to the master for more than a single cycle. This property allows the master to delay the actual read if it is busy with internal operations.

The slave signals the end of the transaction through a novel *ready counter* to provide an early notification. This early notification simplifies the integration of peripherals into pipelined masters. Slaves can also provide several levels of pipelining. This feature is announced by two static output ports (one for read and one write pipeline levels).

Off-chip connections (e.g. main memory) are device specific and need a slave to perform the translation. Peripheral interrupts are not covered by this specification.

9.1.1 Features

SimpCon provides following main features:

- Master/slave point-to-point connection
- Synchronous operation
- Read and write transactions
- Early pipeline release for the master
- Pipelined transactions
- Open-source specification
- Low implementation overheads

9.1.2 Basic Read Transaction

Figure 9.1 shows a basic read transaction for a slave with one cycle latency. In the first cycle, the address phase, the rd signals the slave to start the read transaction. The address is registered by the slave. During the following cycle, the read phase,[1] the slave performs the read and registers the data. Due to the register in the slave, the data is available in the third cycle, the result phase. To simplify the master, rd_data stays valid until the next read request response. It is therefore possible for a master to issue a pre-fetch command early. When the pre-fetched data arrives too early it is still valid when the master actually wants to read it. Keeping the read data stable in the slave is *mandatory*.

[1]It has to be noted that the read phase can be longer for devices with a high latency. For simple on-chip I/O devices the read phase can be omitted completely (0 cycles). In that case rdy_cnt will be zero in the cycle following the address phase.

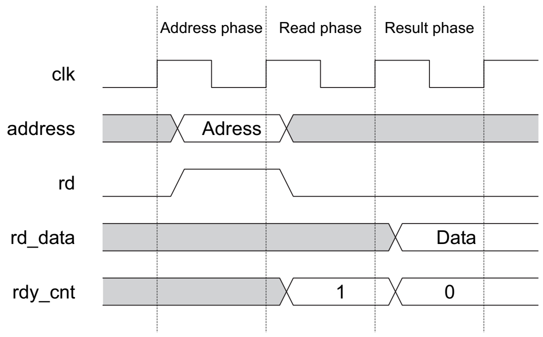


Figure 9.1: Basic read transaction

9.1.3 Basic Write Transaction

A write transaction consists of a single cycle address/command phase started by assertion of wr where the address and the write data are valid. address and wr_data are usually registered by the slave. The end of the write cycle is signalled to the master by the slave with rdy_cnt. See Section 9.3 and an example in Figure 9.3.

9.2 SimpCon Signals

This sections defines the signals used by the SimpCon connection. Some of the signals are optional and may not be present on a peripheral device.

All signals are a single direction point-to-point connection between a master and a slave. The signal details are described by the device that drives the signal. Table 9.1 lists the signals that define the SimpCon interface. The column Direction indicates whether the signal is driven by the master or the slave.

9.2.1 Master Signal Details

This section describes the signals that are driven by the master to initiate a transaction.

address

Master addresses represent word addresses as offsets in the slave's address range. address is valid a single cycle either with rd for a read transaction or with wr and wr_data for a write

Signal	Width	Direction	Required	Description
address	1–32	Master	No	Address lines from the master to the slave port
wr_data	32	Master	No	Data lines from the master to the slave port
rd	1	Master	No	Start of a read transaction
wr	1	Master	No	Start of a write transaction
rd_data	32	Slave	No	Data lines from the slave to the master port
rdy_cnt	2	Slave	Yes	Transaction end signalling
rd_pipeline_level	2	Slave	No	Maximum pipeline level for read transactions
wr_pipeline_level	2	Slave	No	Maximum pipeline level for write transactions
sel_byte	2	Master	No	Reserved for future use
uncached	1	Master	No	Non cached access
cache_flash	1	Master	No	Flush/invalidate a cache

Table 9.1: SimpCon port signals

transaction. The number of bits for address depends on the slave's address range. For a single port slave, address can be omitted.

wr_data

The wr_data signals carry the data for a write transaction. It is valid for a single cycle together with address and wr. The signal is typically 32 bits wide. Slaves can ignore upper bits when the slave port is less than 32 bits.

rd

The rd signal is asserted for a single clock cycle to start a read transaction. address has to be valid in the same cycle.

wr

The wr signal is asserted for a single clock cycle to start a write transaction. address and wr_data have to be valid in the same cycle.

sel_byte

The sel_byte signal is reserved for future versions of the SimpCon specification to add individual byte enables.

uncached

The uncached signal is asserted for a single clock cycle during a read or write transaction to signal that a cache, connected in the SimpCon pipeline, shall not cache the read or write access.

cache_flash

The cache_flash signal is asserted for a single clock cycle invalidates a cache connected in the SimpCon pipeline.

9.2.2 Slave Signal Details

This section describes the signals that are driven by the slave as a response to transactions initiated by the master.

rd_data

The rd_data signals carry the result of a read transaction. The data is valid when rdy_cnt reaches 0 and *stays valid* until a new read result is available. The signal is typically 32 bits wide. Slaves that provide less than 32 bits should pad the upper bits with 0.

rdy_cnt

The rdy_cnt signal provides the number of cycles until the pending transaction will finish. A 0 means that either read data is available or a write transaction has been finished. Values of 1 and 2 mean the transaction will finish in at least 1 or 2 cycles. The maximum value is 3 and means the transaction will finish in 3 or *more* cycles. Note that not all values have to be used in a transaction. Each monotonic sequence of rdy_cnt values is legal.

rd_pipeline_level

The static rd_pipeline_level provides the master with the read pipeline level of the slave. The signal has to be constant to enable the synthesizer to optimize the pipeline level dependent state machine in the master.

wr_pipeline_level

The static wr_pipeline_level provides the master with the write pipeline level of the slave. The signal has to be constant to enable the synthesizer to optimize the pipeline level dependent state machine in the master.

9.3 Slave Acknowledge

Flow control between the slave and the master is usually done by a single signal in the form of *wait* or *acknowledge*. The ack signal, e.g. in the Wishbone specification, is set when the data is available or the write operation has finished. However, for a pipelined master it can be of interest to know it *earlier* when a transaction will finish.

For many slaves, e.g. an SRAM interface with fixed wait states, this information is available inside the slave. In the SimpCon interface, this information is communicated to the master through the two bit ready counter (rdy_cnt). rdy_cnt signals the number of cycles until the read data will be available or the write transaction will be finished. A value of 0 is equivalent to an *ack* signal and 1, 2, and 3 are equivalent to a wait request with the distinction that the master knows how long the wait request will last.

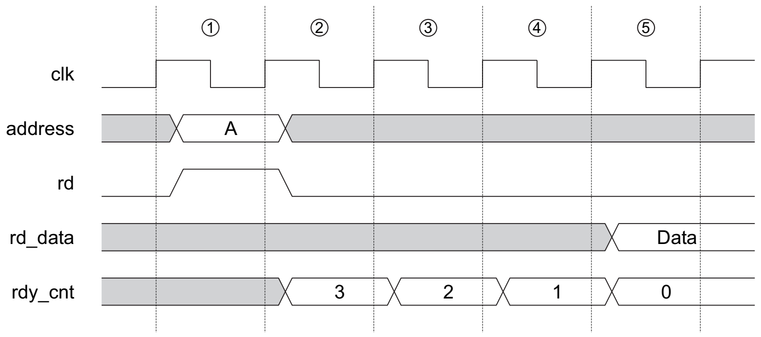


Figure 9.2: Read transaction with wait states

To avoid too many signals at the interconnect, rdy_cnt has a width of two bits. Therefore, the maximum value of 3 has the special meaning that the transaction will finish in 3 or *more* cycles. As a result the master can only use the values 0, 1, and 2 to release actions in its pipeline. If necessary, an extension for a longer pipeline is straightforward with a larger rdy_cnt.[2]

Idle slaves will keep the former value of 0 for rdy_cnt. Slaves that do not know in advance how many wait states are needed for the transaction can produce sequences that omit any of the numbers 3, 2, and 1. A simple slave can hold rdy_cnt on 3 until the data is available and set it then directly to 0. The master has to handle those situations. Practically, this reduces the possibilities of pipelining and therefore the performance of the interconnect. The master will read the data later, which is not an issue as the data stays valid.

Figure 9.2 shows an example of a slave that needs three cycles for the read to be processed. In cycle 1, the read command and the address are set by the master. The slave registers the address and sets rdy_cnt to 3 in cycle 2. The read takes three cycles (2–4) during which rdy_cnt gets decremented. In cycle 4 the data is available inside the slave and gets registered. It is available in cycle 5 for the master and rdy_cnt is finally 0. Both, the rd_data and rdy_cnt will keep their value until a new transaction is requested.

Figure 9.3 shows an example of a slave that needs three cycles for the write to be processed. The address, the data to be written, and the write command are valid during cycle 1. The slave registers the address and the write data during cycle 1 and performs the write operation during cycles 2–4. The rdy_cnt is decremented and a non-pipelined slave can accept a new command after cycle 4.

[2]The maximum value of the ready counter is relevant for the early restart of a waiting master. A longer latency from the slave e.g., for DDR SDRAM, will map to the maximum value of the counter for the first cycles.

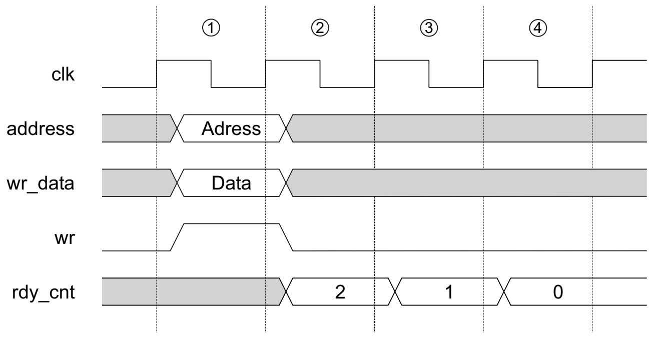


Figure 9.3: Write transaction with wait states

9.4 Pipelining

Figure 9.4 shows a read transaction for a slave with four clock cycles latency. Without any pipelining, the next read transaction will start in cycle 7 after the data from the former read transaction is read by the master. The three bottom lines show when new read transactions (only the rd signal is shown, address lines are omitted from the figure) can be started for different pipeline levels. With pipeline level 1, a new transaction can start in the same cycle when the former read data is available (in this example in cycle 6). At pipeline level 2, a new transaction (either read or write) can start when rdy_cnt is 1, for pipeline level 3 the next transaction can start at a rdy_cnt of 2.

The implementation of level 1 in the slave is trivial (just two more transitions in the state machine). It is recommended to provide at least level 1 for read transactions. Level 2 is a little bit more complex but usually no additional address or data registers are necessary.

To implement level 3 pipelining in the slave, at least one additional address register is needed. However, to use level 3 the master has to issue the request in the same cycle as rdy_cnt goes to 2. That means that this transition is combinatorial. We see in Figure 9.4 that rdy_cnt value of 3 means three or more cycles until the data is available and can therefore not be used to trigger a new transaction. Extension to an even deeper pipeline needs a wider rdy_cnt.

9.5 Interconnect

Although the definition of SimpCon is from a single master/slave point-to-point viewpoint, all variations of multiple slave and multiple master devices are possible.

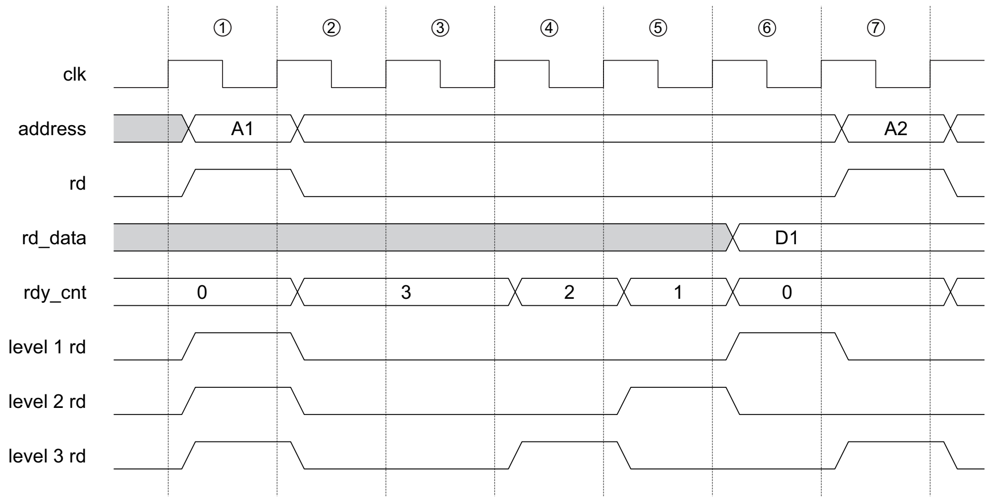


Figure 9.4: Different pipeline levels for a read transaction

Slave Multiplexing

To add several slaves to a single master, rd_data and rdy_cnt have to be multiplexed. Due to the fact that all rd_data signals are already registered by the slaves, a single pipeline stage will be enough for a large multiplexer. The selection of the multiplexer is also known at the transaction start but at least needed one cycle later. Therefore it can be registered to further speed up the multiplexer.

Master Multiplexing

SimpCon defines no signals for the communication between a master and an arbiter. However, it is possible to build a multi-master system with SimpCon. The SimpCon interface can be used as an interconnect between the masters and the arbiter and the arbiter and the slaves. In this case the arbiter acts as a slave for the master and as a master for the peripheral devices. An example of an arbiter for SimpCon, where JOP and a VGA controller are two masters for a shared main memory, can be found in [98]. The same arbiter is also used to build a chip-multiprocessor version of JOP [100].

The missing arbitration protocol in SimpCon results in the need to queue $n-1$ requests in an arbiter for n masters. However, this additional hardware results in a zero cycle bus grant. The master, which gets the bus granted, starts the slave transaction in the same cycle

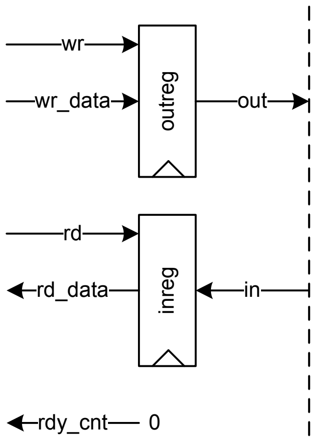


Figure 9.5: A simple input/output port with a SimpCon interface

as the original read/write request.

To add several slaves to a single master the rd_data and rdy_cnt have to be multiplexed. Due to the fact that all rd_data signals are registered by the slaves a single pipeline stage will be enough for a large multiplexer. The selection of the multiplexer is also known at the transaction start but needed at most in the next cycle. Therefore it can be registered to further speed up the multiplexer.

9.6 Examples

This section provides some examples for the application of the SimpCon definition.

9.6.1 I/O Port

Figure 9.5 shows a simple I/O port with a minimal SimpCon interface. As address decoding is omitted for this simple device signal address is not needed. Furthermore we can tie rdy_cnt to 0. We only need the rd or wr signal to enable the port. Listing 9.1 shows the VHDL code for this I/O port.

```
entity sc_test_slave is generic (addr_bits : integer);

port (
    clk         : in  std_logic;
    reset       : in  std_logic;
-- SimpCon interface
    wr_data        : in  std_logic_vector(31 downto 0);
    rd, wr         : in  std_logic;
    rd_data        : out std_logic_vector(31 downto 0);
    rdy_cnt        : out unsigned(1 downto 0);
-- input/output ports
    in_data        : in  std_logic_vector(31 downto 0)
    out_data       : out std_logic_vector(31 downto 0)

); end sc_test_slave;

architecture rtl of sc_test_slave is

begin

    rdy_cnt <= "00";    -- no wait states

process(clk, reset) begin

    if (reset='1') then
        rd_data <= (others => '0');
        out_data <= (others => '0');
    elsif rising_edge(clk) then
        if rd='1' then
            rd_data <= in_data;
        end if;
        if wr='1' then
            out_data <= wr_data;
        end if;
    end if;

end process;

end rtl;
```

Listing 9.1: VHDL source for the simple input/output port

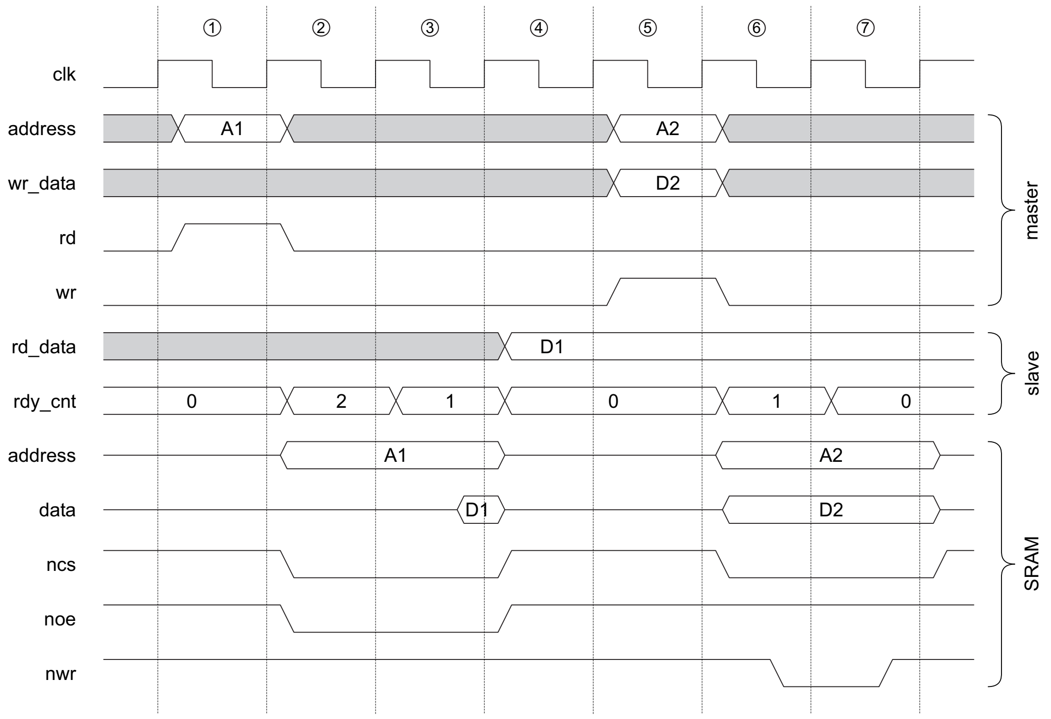


Figure 9.6: Static RAM interface without pipelining

9.6.2 SRAM interface

The following example assumes a master (processor) clocked at 100 MHz and an static RAM (SRAM) with 15 ns access time. Therefore the minimum access time for the SRAM is two cycles. The slack time of 5 ns forces us to use output registers for the SRAM address and write data and input registers for the read data in the I/O cells of the FPGA. These registers fit nicely with the intention of SimpCon to use registers inside the slave.

Figure 9.6 shows the memory interface for a non-pipelined read access followed by a write access. Four signals are driven by the master and two signals by the slave. The lower half of the figure shows the signals at the FPGA pins where the SRAM is connected.

In cycle 1 the read transaction is started by the master and the slave registers the address. The slave also sets the registered control signals ncs and noe during cycle 1. Due to the placement of the registers in the I/O cells, the address and control signals are valid at the FPGA pins very early in cycle 2. At the end of cycle 3 (15 ns after address, ncs and noe

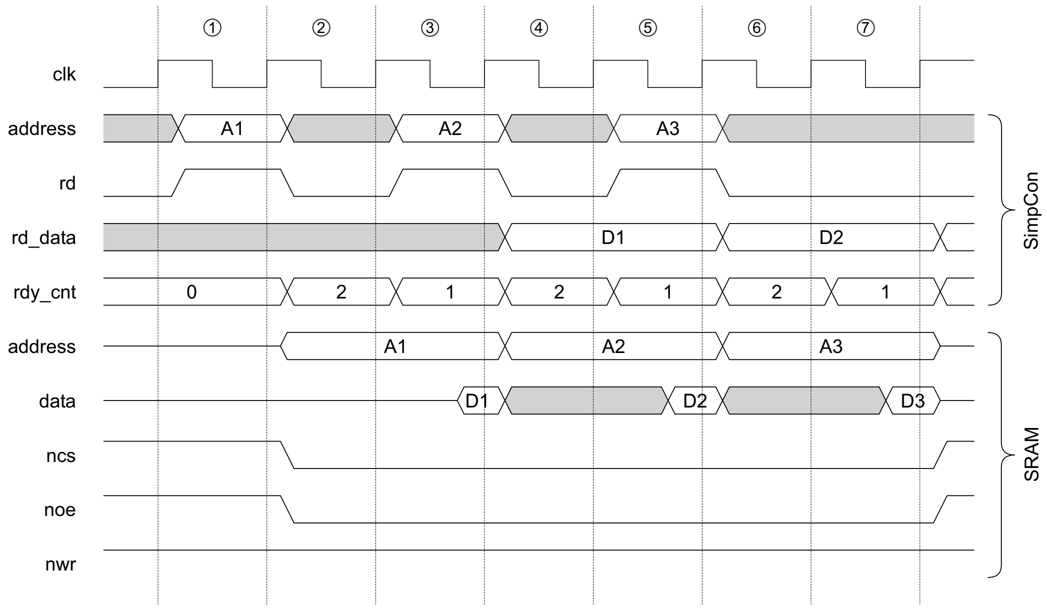


Figure 9.7: Pipelined read from a static RAM

are stable) the data from the SRAM is available and can be sampled with the rising edge for cycle 4. The setup time for the read register is short, as the register can be placed in the I/O cell. The master reads the data in cycle 4 and starts a write transaction in cycle 5. Address and data are again registered by the slave and are available for the SRAM at the beginning of cycle 6. To perform a write in two cycles the nwr signal is registered by a negative triggered flip-flop.

In Figure 9.7 we see a pipelined read from the SRAM with pipeline level 2. With this pipeline level and the two cycles read access time of the SRAM we achieve the maximum possible bandwidth.

We can see the start of the second read transaction in cycle 3 during the read of the first data from the SRAM. The new address is registered in the same cycle and available for the SRAM in the following cycle 4. Although we have a pipeline level of 2 we need no additional address or data register. The read data is available for two cycles (rdy_cnt 2 or 1 for the next read) and the master has the freedom to select one of the two cycles to read the data.

It has to be noted that pipelining with one read per cycle is possible with SimpCon. We just showed a 2 cycle slave in this example. For a SDRAM memory interface the ready counter will stay either at 2 or 1 during the single cycle reads (depending on the slave pipeline level). It will go down to 0 only for the last data word to read.

9.7 Available VHDL Files

Besides the SimpCon documentation, some example VHDL files for slave devices and bridges are available from `http://opencores.org/?do=project&who=simpcon`. All components are also part of the standard JOP distribution.

9.7.1 Components

- sc_pack.vhd defines VHDL records and some constants.
- sc_test_slave.vhd is a very simple SimpCon device. A counter to be read out and a register that can be written and read. There is no connection to the outer world. This example can be used as basis for a new SimpCon device.
- sc_sram16.vhd is a memory controller for 16-bit SRAM.
- sc_sram32.vhd is a memory controller for 32-bit SRAM.
- sc_sram32_flash.vhd is a memory controller for 32-bit SRAM, a NOR Flash, and a NAND Flash as used in the Cycore FPGA board for JOP.
- sc_uart.vhd is a simple UART with configurable baud rate and FIFO width.
- sc_usb.vhd is an interface to the parallel port of the FTDI 2232 USB chip. The register definition is identical to the UART and the USB connection can be used as a drop in replacement for a UART.
- sc_isa.vhd interfaces the old ISA bus. It can be used for the popular CS8900 Ethernet chip.
- sc_sigdel.vhd is a configurable sigma-delta ADC for an FPGA that needs just two external components: a capacitor and a resistor.
- sc_fpu.vhd provides an interface to the 32-bit FPU available at `www.opencores.org`.
- sc_arbiter_*.vhd different zero cycle SimpCon arbiters for CMP systems written by Christof Pitter [99].

9.7.2 Bridges

- sc2wb.vhd is a SimpCon/Wishbone [94] bridge.
- sc2avalon.vhd is a SimpCon/Avalon [4] bridge to integrate a SimpCon based design with Altera's SOPC Builder [5].
- sc2ahbsl.vhd provides an interface to AHB slaves as defined in Gaisler's GRLIB [45]. Many of the available GPL AHB modules from the GRLIB can be used in a SimpCon based design.

9.8 Why a New Interconnection Standard?

There are many interconnection standards available for SoC designs. The natural question is: Why propose a new one? The answer is given in the following section. In summary, the available standards are still in the tradition of backplane busses and do not fit very well for pipelined on-chip interconnections.

9.8.1 Common SoC Interconnections

Several point-to-point and bus standards have been proposed. The following section gives a brief overview of common SoC interconnection standards.

The Advanced Microcontroller Bus Architecture (AMBA) [7] is the interconnection definition from ARM. The specification defines three different busses: Advanced High-performance Bus (AHB), Advanced System Bus (ASB), and Advanced Peripheral Bus (APB). The AHB is used to connect on-chip memory, cache, and external memory to the processor. Peripheral devices are connected to the APB. A bridge connects the AHB to the lower bandwidth APB. An AHB bus transfer can be one cycle at burst operation. With the APB a bus transfer requires two cycles and no burst mode is available. Peripheral bus cycles with wait states are added in the version 3 of the APB specification. ASB is the predecessor of AHB and is not recommended for new designs (ASB uses both clock phases for the bus signals – very uncommon for today's synchronous designs). The AMBA 3 AXI (Advanced eXtensible Interface) [8] is the latest extension to AMBA. AXI introduces out-of-order transaction completion with the help of a 4 bit transaction ID tag. A ready signal acknowledges the transaction start. The master has to hold the transaction information (e.g. address) until the interconnect signals ready. This enhancement ruins the elegant single cycle address phase from the original AHB specification.

Wishbone [94] is a public domain standard used by several open-source IP cores. The Wishbone interface specification is still in the tradition of microcomputer or backplane busses. However, for a SoC interconnect, which is usually point-to-point,[3] this is not the best approach. The master is requested to hold the address and data valid through the whole read or write cycle. This complicates the connection to a master that has the data valid only for one cycle. In this case the address and data have to be registered *before* the Wishbone connect or an expensive (time and resources) multiplexer has to be used. A register results in one additional cycle latency. A better approach would be to register the address and data in the slave. In that case the address decoding in the slave can be performed in the same cycle as the address is registered. A similar issue, with respect to the master, exists for the output data from the slave: As it is only valid for a single cycle, the data has to be registered by the master when the master is not reading it immediately. Therefore, the slave should keep the last valid data at its output even when the Wishbone strobe signal (*wb.stb*) is not assigned anymore. Holding the data in the slave is usually *for free* from the hardware complexity – it is *just* a specification issue. In the Wishbone specification there is no way to perform pipelined read or write. However, for blocked memory transfers (e.g. cache load) this is the usual way to achieve good performance.

The Avalon [4] interface specification is provided by Altera for a system-on-a-programmable-chip (SOPC) interconnection. Avalon defines a great range of interconnection devices ranging from a simple asynchronous interface intended for direct static RAM connection up to sophisticated pipeline transfers with variable latencies. This great flexibility provides an easy path to connect a peripheral device to Avalon. How is this flexibility possible? The *Avalon Switch Fabric* translates between all those different interconnection types. The switch fabric is generated by Altera's SOPC Builder tool. However, it seems that this switch fabric is Altera proprietary, thus tying this specification to Altera FPGAs.

The On-Chip Peripheral Bus (OPB) [68] is an open standard provided by IBM and used by Xilinx. The OPB specifies a bus for multiple masters and slaves. The implementation of the bus is not directly defined in the specification. A distributed ring, a centralized multiplexer, or a centralized AND/OR network are suggested. Xilinx uses the AND/OR approach and all masters and slaves must drive the data busses to zero when inactive.

Sonics Inc. defined the Open Core Protocol (OCP) [91] as an open, freely available standard. The standard is now handled by the OCP International Partnership[4].

[3]Multiplexers are used instead of busses to connect several slaves and masters.
[4]`www.ocpip.org`

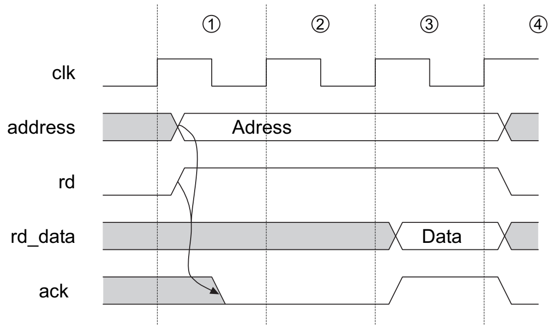


Figure 9.8: Classic basic read transaction

9.8.2 What's Wrong with the Classic Standards?

All SoC interconnection standards, which are widely in use, are still in the tradition of a backplane bus. They force the master to hold the address and control signals until the slave provides the data or acknowledges the write request. However, this is not necessary in a clocked, synchronous system. Why should we force the master to hold the signals? Let the master move on after submitting the request in a single cycle. Forcing the address and control valid for the complete request disables any form of pipelined requests.

Figure 9.8 shows a read transaction with wait states as defined in Wishbone [94], Avalon [4], OPB [68], and OCP [91].[5] The master issues the read request and the address in cycle 1. The slave has to reset the ack in the same cycle. When the slave data is available, the acknowledge signal is set (ack in cycle 3). The master has to read the data and register them within the same clock cycle. The master has to hold the address, write data, and control signal active until the acknowledgement from the slave arrives. For pipelined reads, the ack signal can be split into two signals (available in Avalon and OCP): one to accept the request and a second one to signal the available data.

The master is blind about the status of the outstanding transaction until it is finished. It could be possible that the slave informs the master in how many cycles the result will be available. This information can help in building deeply pipelined masters.

Only the AMBA AHB [7] defines a different protocol. A single cycle address phase followed by a variable length data phase. The slave acknowledgement (HREADY) is only necessary in the data phase avoiding the combinatorial path from address/command to the acknowledgement. Overlapping address and data phase is allowed and recommended for

[5]The signal names are different, but the principle is the same for all mentioned busses.

high performance. Compared to SimpCon, AMBA AHB allows for single stage pipelining, whereas SimpCon makes multi-stage pipelining possible using the ready counter (rdy_cnt). The rdy_cnt signal defines the delay between the address and the data on a read, signalled by the slave. Therefore, the pipeline depth of the bus and the slaves is only limited by the bit width of rdy_cnt.

Another issue with all interconnection standards is the single cycle availability of the read data at the slaves. Why not keep the read data valid as long as there is no new read data available? This feature would allow the master to be more flexible when to read the data. It would allow issuing a read command and then continuing with other instructions – a feature known as data pre-fetching to hide long latencies.

The last argument sounds contradictory to the first argument: provide the transaction data at the master just for a single cycle, but request the slave to hold the data for several cycles. However, it is motivated by the need to free up the master, keep it *moving*, and move the data hold (register) burden into the slave. As data processing bottlenecks are usually found in the master devices, it sounds natural to move as much work as possible to the slave devices to free up the master.

Avalon, Wishbone, and OPB provide a single cycle latency access to slaves due to the possibility of acknowledging a request in the same cycle. However, this feature is a scaling issue for larger systems. There is a combinatorial path from master address/command to address decoding, slave decision on ack, slave ack multiplexing back to the master and the master decision to hold address/command or read the data and continue. Also, the slave output data multiplexer is on a combinatorial path from the master address.

AMBA, AHB, and SimpCon avoid this scaling issue by requesting the acknowledge in the cycle following the command. In SimpCon and AMBA, the select for the read data multiplexer can be registered as the read address is known at least one cycle before the data is available. The later acknowledgement results in a minor drawback on SimpCon and AMBA (nothing is for free): It is not possible to perform a single cycle read or write without pipelining. A single, non pipelined transaction takes two cycles without a wait state. However, a single cycle read transaction is only possible for very simple slaves. Most non-trivial slaves (e.g. memory interfaces) will not allow a single cycle access anyway.

9.8.3 Evaluation

We compare the SimpCon interface with the AMBA and the Avalon interface as two examples of common interconnection standards. As an evaluation example, we interface an external asynchronous SRAM with a tight timing. The system is clocked at 100 MHz and the access time for the SRAM is 15 ns. Therefore, there are 5 ns available for on-chip reg-

Performance	Memory	Interconnect
16,633	32 bit SRAM	SimpCon
14,259	32 bit SRAM	AMBA
14,015	32 bit SRAM	Avalon/PTF
13,920	32 bit SRAM	Avalon/VHDL
15,762	32 bit on-chip	Avalon
14,760	16 bit SRAM	SimpCon
11,322	16 bit SRAM	Avalon
7,288	16 bit SDRAM	Avalon

Table 9.2: JOP performance with different interconnection types

ister to SRAM input and SRAM output to on-chip register delays. As an SoC, we use an actual low-cost FPGA (Cyclone EP1C6 [3] and a Cyclone II).

The master is a Java processor (JOP [123, 130]). The processor is configured with a 4 KB instruction cache and a 512 byte on-chip stack cache. We run a complete application benchmark on the different systems. The embedded benchmark (*Kfl* as described in [122]) is an industrial control application already in production.

Table 9.2 shows the performance numbers of this JOP/SRAM interface on the embedded benchmark. It measures iterations per second and therefore higher numbers are better. One iteration is the execution of the main control loop of the *Kfl* application. For a 32 bit SRAM interface, we compare SimpCon against AMBA and Avalon. SimpCon outperforms AMBA by 17% and Avalon by 19%[6] on a 32 bit SRAM.

The AMBA experiment uses the SRAM controller provided as part of GRLIB [45] by Gaisler Research. We avoided writing our own AMBA slave to verify that the AMBA implementation on JOP is correct. To provide a fair comparison between the single master solutions with SimpCon and Avalon, the AMBA bus was configured without an arbiter. JOP is connected directly to the AMBA memory slave. The difference between the SimpCon and the AMBA performance can be explained by two facts: (1) as with the Avalon interconnect, the master has the information when the slave request is ready at the same cycle when the data is available (compared to the rdy_cnt feature); (2) the SRAM controller is conservative as it asserts HREADY one cycle later than the data is available in the read register (HRDATA). The second issue can be overcome by a better implementation of the SRAM AMBA slave.

[6]The performance is the measurement of the execution time of the whole application, not only the difference between the bus transactions.

The Avalon experiment considers two versions: an SOPC Builder generated interface (PTF) to the memory and a memory interface written in VHDL. The SOPC Builder interface performs slightly better than the VHDL version that generates the Avalon waitrequest signal. It is assumed that the SOPC Builder version uses fixed wait states within the switch fabric.

We also implemented an Avalon interface to a single-cycle on-chip memory. SimpCon is even faster with the 32 bit off-chip SRAM than with the on-chip memory connected via Avalon. Furthermore, we also performed experiments with a 16 bit memory interface to the same SRAM. With this smaller data width the pressure on the interconnection and memory interface is higher. As a result the difference between SimpCon and Avalon gets larger (30%) on the 16 bit SRAM interface. To complete the picture we also measured the performance with an SDRAM memory connected to the Avalon bus. We see that the large latency of an SDRAM is a big performance issue for the Java processor.

9.9 Summary

This chapter describes a simple (with respect to the definition and implementation) and efficient SoC interconnect [127]. The novel signal rdy_cnt allows an early signalling to the master when read data will be valid. This feature allows the master to restart a stalled pipeline earlier to react for arriving data. Furthermore, this feature also enables pipelined bus transactions with a minimal effort on the master and the slave side.

We have compared SimpCon quantitative with AMBA and Avalon, two common interconnection definitions. The application benchmark shows a performance advantage of SimpCon by 17% over AMBA and 19% over Avalon interfaces to an SRAM.

SimpCon is used as the main interconnect for the Java processor JOP in a single master, multiple salves configuration. SimpCon is also used to implement a shared memory chip-multiprocessor version of JOP. Furthermore, in a research project on time-triggered network-on-chip [128] SimpCon is used as the *socket* to this NoC.

The author thanks Kevin Jennings and Tommy Thorn for the interesting discussions about SimpCon, Avalon, and on-chip interconnection in general at the Usenet newsgroup `comp.arch.fpga`.

10 Chip Multiprocessing

In order to generate a small and predictable processor, several advanced and resource-consuming features (such as instruction folding or branch prediction) were omitted from the design. The resulting low resource usage of JOP makes it possible to integrate more than one processor in an FPGA. Since embedded applications are naturally multi-threaded systems, the performance can easily be enhanced using a multi-processor solution.

This chapter describes the configuration of a chip multiprocessor (CMP) version of JOP. The various SimpCon based arbiters have been developed by Christof Pitter and are described in [98, 99, 100]. A multi-processor JVM with shared memory offers research possibilities such as: scheduling of Java threads and synchronization between the processors or WCET analysis for the shared memory access.

The project file to start with is `cyccmp`, a configuration for three processors with a TDMA based arbiter in the Cycore board with the EP1C12.

10.1 Memory Arbitration

A central arbiter regulates the synchronization on memory read and write operations.

10.1.1 Main Memory

Three different arbiter are available for the access policy to the main memory: priority based, fairness based, and a time division multiple access (TDMA) arbiter. The main memory is shared between all cores.

The TDMA based memory arbiter provides a static schedule for the memory access. Therefore, access time to the memory is independent of tasks running on other cores. The worst-case execution time (WCET) of a memory loads or stores can be calculated by considering the worst-case phasing of the memory access pattern relative to the TDMA schedule [96].

In the default configuration each processor cores has an equally sized slot for the memory access. The TDMA schedule can also be optimized for different utilizations of processing cores. The TDMA schedule can be optimized to distribute slack time of tasks to other tasks with a tighter deadline [136].

10.1.2 I/O Devices

Each core contains a set of local I/O devices, needed for the runtime system (e.g., timer interrupt, lock support). The serial interface for program download and a *stdio* device is connected to the first core.

For additional I/O devices two options exist: either they are connected to one core, or shared by all/some cores. The first option is useful when the bandwidth requirement of the I/O device is high. As I/O devices are memory mapped they can be connected to the main memory arbiter in the same way as the memory controller. In that case the I/O devices are shared between the cores and standard synchronization for the access is needed. For high bandwidth demands a dedicated arbiter for I/O devices or even for a single device can be used.

An interrupt line of an I/O device can be connected to a single core or to several cores. As interrupts can be individually disabled in software, a connection of all interrupt lines to all cores provides the most flexible solution.

10.2 Booting a CMP System

One interesting aspect of a CMP system is how the startup or boot-up is performed. On power-up, the FPGA starts the configuration state machine to read the FPGA configuration data either from a Flash memory or via a download cable from the PC during the development process. When the configuration has finished, an internal reset is generated. After this reset, microcode instructions are executed, starting from address 0. At this stage, we have not yet loaded any application program (Java bytecode). The first sequence in microcode performs this task. The Java application can be loaded from an external Flash memory, via a PC serial line, or an USB-port.

In the next step, a minimal stack frame is generated and the special method Startup.boot() is invoked, even though some parts of the JVM are not yet setup. From now on JOP runs in Java mode. The method boot() performs the following steps:

- Send a greeting message to *stdout*
- Detect the size of the main memory
- Initialize the data structures for the garbage collector
- Initialize java.lang.System
- Print out JOP's version number, detected clock speed, and memory size
- Invoke the static class initializers in a predefined order
- Invoke the main method of the application class

The boot-up process is the same for all processors until the generation of the internal reset and the execution of the first microcode instruction. From that point on, we have to take care that *only one* processor performs the initialization steps.

All processors in the CMP are functionally identical. Only one processor is designated to boot-up and initialize the whole system. Therefore, it is necessary to distinguish between the different CPUs. We assign a unique CPU identity number (CPU ID) to each processor. Only processor CPU0 is designated to do all the boot-up and initialization work. The other CPUs have to wait until CPU0 completes the boot-up and initialization sequence. At the beginning of the booting sequence, CPU0 loads the Java application. Meanwhile, all other processors are waiting for an *initialization finished* signal of CPU0. This busy wait is performed in microcode. When the other CPUs are enabled, they will run the same sequence as CPU0. Therefore, the initialization steps are guarded by a condition on the CPU ID.

10.3 CMP Scheduling

There are two possibilities to run multiple threads on the CMP system:

1. A single thread per processor
2. Several threads on each processor

For the configuration of one thread per processor the scheduler does not need to be started. Running several threads on each core is managed via the JOP real-time threads RtThread.

The scheduler on each core is a preemptive, priority based real-time scheduler. As each thread gets a unique priority, no FIFO queues within priorities are needed. The best analyzable real-time CMP scheduler does not allow threads to migrate between cores. Each thread is pinned to a single core at creation. Therefore, standard scheduling analysis can be performed on a per core base. Threads cannot migrate from one core to another one.

Similar to the uniprocessor version of JOP, the application is divided into an initialization phase and a mission phase. During the initialization phase, a predetermined core executes only one thread that has to create all data structures and the threads for the mission phase. During transition to the mission phase all created threads are started.

The uniprocessor real-time scheduler for JOP has been enhanced to facilitate the scheduling of threads in the CMP configuration. Each core executes its own instance of the scheduler. The scheduler is implemented as Runnable, which is registered as an interrupt handler for the core local timer interrupt. The scheduling is not tick-based. Instead, the timer interrupt is reprogrammed after each scheduling decision. During the mission start, the other cores and timer interrupts are enabled.

Another interesting option to use a CMP system is to execute exactly one thread per core. In this configuration scheduling overheads can be avoided and each core can reach an utilization of 100% without missing a deadline. To explore the CMP system without a scheduler, a mechanism is provided to register objects, which implement the Runnable interface, for each core. When the other cores are enabled, they execute the run method of the Runnable as their *main* method.

10.3.1 One Thread per Core

The first processor executes, as usual, main(). To execute code on the other cores a Runnable has to be registered for each core. After registering those Runnables the other cores need to be started. The code in Listing 10.1 shows an example that can be found in test/cmp/HelloCMP.java.

10.3.2 Scheduling on the CMP System

Running several threads on each core is possible with RtThread and setting the core for each thread with RtThread.setProcessor(nr). The example in Listing 10.2 (test/cmp/RtHelloCMP.java) shows registering of 50 threads on all available cores. On missionStart() the threads are distributed to the cores, a scheduler for each core registered as timer interrupt handler, and the other cores started.

```
public class HelloCMP implements Runnable {

    int id;
    static Vector msg;

    public HelloCMP(int i) {
        id = i;
    }

    public static void main(String[] args) {

        msg = new Vector();
        System.out.println("Hello World from CPU 0");

        SysDevice sys = IOFactory.getFactory().getSysDevice();
        for (int i=0; i<sys.nrCpu-1; ++i) {
            Runnable r = new HelloCMP(i+1);
            Startup.setRunnable(r, i);
        }

        // start the other CPUs
        sys.signal = 1;
        // print their messages
        for (;;) {
            int size = msg.size();
            if (size!=0) {
                StringBuffer sb = (StringBuffer) msg.remove(0);
                System.out.println(sb);
            }
        }
    }

    public void run() {
        StringBuffer sb = new StringBuffer();
        sb.append("Hello World from CPU ");
        sb.append(id);
        msg.addElement(sb);
    }
}
```

Listing 10.1: A CMP version of Hello World

```
public class RtHelloCMP extends RtThread {

    public RtHelloCMP(int prio, int us) { super(prio, us); }

    int id;
    public static Vector msg;
    final static int NR_THREADS = 50;

    public static void main(String[] args) {
        msg = new Vector();
        System.out.println("Hello World from CPU 0");
        SysDevice sys = IOFactory.getFactory().getSysDevice();
        for (int i=0; i<NR_THREADS; ++i) {
            RtHelloCMP th = new RtHelloCMP(1, 1000*1000);
            th.id = i;
            th.setProcessor(i%sys.nrCpu);
        }
        RtThread.startMission();        // start mission and other CPUs
        for (;;) {                      // print their messages
            RtThread.sleepMs(5);
            int size = msg.size();
            if (size!=0) {
                StringBuffer sb = (StringBuffer) msg.remove(0);
                for (int i=0; i<sb.length(); ++i) {
                    System.out.print(sb.charAt(i));
    }}}}

    public void run() {
        StringBuffer sb = new StringBuffer();
        StringBuffer ping = new StringBuffer();
        sb.append("Thread "); sb.append((char) ('A'+id)); sb.append(" start on CPU ");
        sb.append(IOFactory.getFactory().getSysDevice().cpuId); sb.append("\r\n");
        msg.addElement(sb);
        waitForNextPeriod();
        for (;;) {
            ping.setLength(0);
            ping.append((char) ('A'+id));
            msg.addElement(ping);
            waitForNextPeriod();
    }}
}
```

Listing 10.2: A CMP version of Hello World with the scheduler

11 Evaluation

In this chapter, we present the evaluation results for JOP. In the following section, the hardware platform that is used for benchmarking is described. This is followed by a comparison of JOP's resource usage with other soft-core processors. In Section 11.3 the performance of a number of different solutions for embedded Java is compared with embedded application benchmarks. Comparison at bytecode level can be found in [122]. This chapter concludes with a description of real-world applications based on JOP.

11.1 Hardware Platforms

During the development of JOP and its predecessors, several different FPGA boards were developed. The first experiments involved using Altera FPGAs EPF8282, EPF8452, EPF10K10 and ACEX 1K30 on boards that were connected to the printer port of a PC for configuration, download and communication. The next step was the development of a stand-alone board with FLASH memory and static RAM. This board was developed in two variants, one with an ACEX 1K50 and the other with a Cyclone EP1C6 or EP1C12. Both boards are pin-compatible and are used in commercial applications of JOP. The Cyclone board is the hardware that is used for the following evaluations.

This board is an ideal development platform for JOP. Static RAM and FLASH are connected via independent buses to the FPGA. All unused FPGA pins and the serial line are available via four connectors. The FLASH can be used to store configuration data for the FPGA and application program/data. The FPGA can be configured with a ByteBlasterMV download cable or loaded from the FLASH (with a small CPLD on board). As the FLASH is also connected to the FPGA, it can be programmed from the FPGA. This allows for upgrades of the Java program and even the processor core itself in the field. The board is slightly different from other FPGA prototyping boards, in that its connectors are on the bottom side. Therefore, it can be used as a module (60 mm x 48 mm), i.e. as part of a larger board that contains the periphery. The Cyclone board contains:

- Altera Cyclone EP1C6Q240 or EP1C12Q240
- Step Down voltage regulator (1V5)

- Crystal clock (20 MHz) at the PLL input (up to 640 MHz internal)
- 512 KB FLASH (for FPGA configuration and program code)
- 1 MB fast asynchronous RAM (15 ns)
- Up to 128 MB NAND FLASH
- ByteBlasterMV port
- Watchdog with a LED
- EPM7064 PLD to configure the FPGA from the FLASH on watchdog reset
- Serial interface driver (MAX3232)
- 56 general-purpose I/O pins

The RAM consists of two independent 16-bit banks (with their own address and control lines). Both RAM chips are on the bottom side of the PCB, directly under the FPGA pins. As the traces are very short (under 10 mm), it is possible to use the RAMs at full speed without reflection problems. The two banks can be combined to form 32-bit RAM or support two independent CPU cores. Pictures and the schematic of the board can be found in Appendix E.1.

The expansion board Baseio hosts the CPU module and provides a complete Java processor system with Internet connection. A step down switching regulator with a large AC/DC input range supplies the core board. All input and output pins are EMC/ESD-protected and routed to large connectors (5.08 mm Phoenix). Analog comparators can be used to build sigma-delta ADCs. For FPGA projects with a network connection, a CS8900 Ethernet controller with an RJ45 connector is included on the expansion board. Pictures and the schematic of the board can be found in Appendix E.2.

11.2 Chip Area and Clock Frequency

Cost is an important issue for embedded systems. The cost of a chip is directly related to the die size (the cost per die is roughly proportional to the square of the die area [62]). Processors for embedded systems are therefore optimized for minimum chip size. In this section, we will compare JOP with different processors in terms of size. One major design objective in the development of JOP was to create a small system that can be implemented in a low-cost FPGA.

Soft-core	Logic Cells	Memory	Frequency
JOP	3,300	7.6 KB	100 MHz
YARI	6,668	18.9 KB	75 MHz
LEON3	7,978	10.9 KB	35 MHz
picoJava	27,560	47.6 KB	40 MHz

Table 11.1: Resource consumption and maximum operating frequency of JOP, YARI, LEON3, and picoJava.

Table 11.1 compares the resource consumption and maximum clock frequency of a time-predictable processor (JOP), a standard MIPS architecture (YARI), the LEON SPARC processor, and a complex Java processor (picoJava), when implemented in the same FPGA (Altera EP1C6/12 FPGA [3]). For the resource comparison we compare the consumption of the two basic structures of an FPGA; Logic cells (LC) and embedded memory blocks. The maximum frequency for all soft-core processors is in the same technology.

JOP is configured with a 1 KB stack cache, 2 KB microcode ROM, and 4 KB method cache with 16 blocks. YARI is a MIPS compatible soft-core [27], optimized for FPGA technology. YARI is configured with a 4-way set-associative instruction cache and a 4-way set-associative write-through data cache. Both caches are 8 KB. LEON3 [44], the open-source implementation of the SPARC V8 architecture, has been ported to the exact same hardware that was used for the JOP numbers. LEON3 is representative for a RISC processor that is used in embedded real-time systems (e.g., by ESA for space missions). The size a frequency numbers of picoJava-II [90] are taken from an implementation in a Altera Cyclone-II FPGA [103].

The streamlined architecture of JOP results in a small design: JOP is half the size of the MIPS core YARI or the SPARC core LEON. Compared with picoJava, JOP consumes about 12% of the resources. JOP's size allows implementing a CMP version of JOP even in a low-cost FPGA. The simple pipeline of JOP achieves the highest clock frequency of the three designs. From the frequency comparison we can estimate that the maximum clock frequency of JOP in an ASIC will also be higher than a standard RISC pipeline in an ASIC.

To prove that the VHDL code for JOP is as portable as possible, JOP was also implemented in a Xilinx Spartan-3 FPGA [154]. Only the instantiation and initialization code for the on-chip memories is vendor-specific, whilst the rest of the VHDL code can be shared for the different targets. JOP consumes about the same LC count in the Spartan device, but has a slower clock frequency (83 MHz).

Processor	Core (gate)	Memory (gate)	Sum. (gate)
JOP	20K	93K	113K
picoJava	128K	314K	442K
aJile	25K	912K	937K
Pentium MMX			1125K

Table 11.2: Gate count estimates for various processors

Table 11.2 provides gate count estimates for JOP, picoJava, the aJile processor, and, as a reference, an old Intel Pentium MMX processor. Equivalent gate count for an LC[1] varies between 5.5 and 7.4 – we chose a factor of 6 gates per LC and 1.5 gates per memory bit for the estimated gate count for JOP in the table. JOP is listed in the typical configuration that consumes 3300 LCs. The Pentium MMX contains 4.5M transistors [40] that are equivalent to 1125K gates.

We can see from the table that the on-chip memory dominates the overall gate count of JOP, and to an even greater extent, of the aJile processor. The aJile processor is roughly the same size as the Pentium MMX, and both are about 10 times larger than JOP.

11.3 Performance

One important question remains: is a time-predictable processor slow? We evaluate the average case performance of JOP by comparing it with other embedded Java systems: Java processors from industry and academia and two just-in-time (JIT) compiler based systems. For the comparison we use JavaBenchEmbedded,[2] a set of open-source Java benchmarks for embedded systems. Kfl and Lift are two real-world applications, described in Section 11.4, adapted with a simulation of the environment to run as stand-alone benchmarks. UdpIp is a simple client/server test program that uses a TCP/IP stack written in Java.

Table 11.3 shows the raw data of the performance measurements of different embedded Java systems for the three benchmarks. The numbers are iterations per second whereby a higher value represents better performance. Figure 11.1 shows the results scaled to the performance of JOP.

[1]The factors are derived from the data provided for various processors in Chapter 12 and from the resource estimates in [121].

[2]Available at `http://www.jopwiki.com/JavaBenchEmbedded`.

	Kfl	UdpIp	Lift
Cjip	176	91	
jamuth	3400	1500	
EJC	9893	2882	
SHAP	11570	5764	12226
aJ100	14148	6415	
JOP	19907	8837	18930
picoJava	23813	11950	25444
CACAO/YARI	39742	17702	38437

Table 11.3: Application benchmark performance on different Java systems. The table shows the benchmark results in iterations per second – a higher value means higher performance.

The numbers for JOP are taken from an implementation in the Altera Cyclone FPGA [3], running at 100 MHz. JOP is configured with a 4 KB method cache and a 1 KB stack cache.

Cjip [70] and aJ100 [2] are commercial Java processors, which are implemented in an ASIC and clocked at 80 and 100 Mhz, respectively. Both cores do not cache instructions. The aj100 contains a 32 KB on-chip stack memory. jamuth [149] and SHAP [157] are Java processors that are implemented in an FPGA. jamuth is the commercial version of the Java processor Komodo [76], a research project for real-time chip multithreading. jamuth is configured with a 4 KB direct-mapped instruction cache for the measurements. The architecture of SHAP is based on JOP and enhanced with a hardware object manager. SHAP also implements the method cache [102]. The benchmark results for SHAP are taken from the SHAP website.[3] SHAP is configured with a 2 KB method cache and 2 KB stack cache.

picoJava [90] is a Java processor developed by Sun. picoJava is no longer produced and the second version (picoJava-II) was available as open-source Verilog code. Puffitsch implemented picoJava-II in an FPGA (Altera Cyclone-II) and the performance numbers are obtained from that implementation [103]. picoJava is configured with a direct-mapped instruction cache and a 2-way set-associative data cache. Both caches are 16 KB.

EJC [41] is an example of a JIT system on a RISC processor (32-bit ARM720T at 74 MHz). The ARM720T contains an 8 KB unified cache. To compare JOP with a JIT based system in exactly the same hardware we use the research JVM CACAO [75] on top of the MIPS compatible soft-core YARI [28]. YARI is configured with a 4-way set-

[3]`http://shap.inf.tu-dresden.de/`, accessed December, 2008

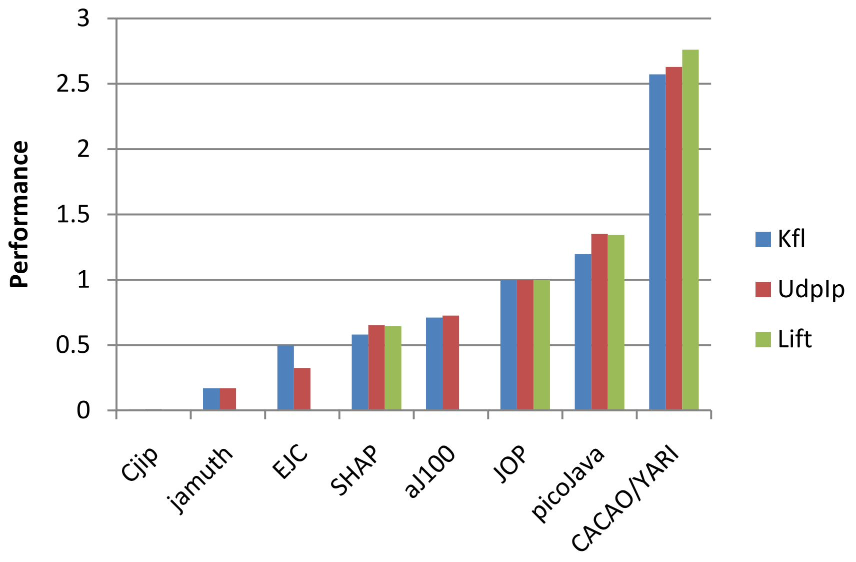


Figure 11.1: Performance comparison of different Java systems with embedded application benchmarks. The results are scaled to the performance of JOP

associative instruction cache and a 4-way set-associative write-through data cache. Both caches are 8 KB.

The measurements do not provide a clear answer to the question of whether a time-predictable architecture is slow. JOP is about 40% faster than the commercial Java processor aJ100, but picoJava is 30% faster than JOP and the JIT/RISC combination (CACAO/YARI) is about 2.7 times faster than JOP. We conclude that a time-predictable solution will never be as fast in the average case as a solution optimized for the average case.

11.4 Applications

Since the start of the development of JOP in late 2000 it has been successfully deployed in several embedded control and automation systems. The following section highlights three different industrial real-time applications that are based on JOP. This section is based on [129]; the first application is also described in [117].

Implementation of a processor in an FPGA is a little bit more expensive than using an

ASIC processor. However, additional application logic, such as a communication controller or an AD converter, can also be integrated into the FPGA. Integration of the processor and the surrounding logic in the same reprogrammable chip is a flexible solution: one can even produce the PCB before all logic components are developed as the interconnection is programmed on-chip and not routed on the PCB. For low-volume projects, as those presented in this section, this flexibility reduces development cost and therefore outweighs the cost of the FPGA device. It has to be noted that low-cost FPGAs, that are big enough for JOP, are available at $11 for a single unit.

Furthermore, most embedded systems are implemented as distributed systems and even very small and memory constraint devices need to communicate. In control applications this communication has to be performed under real-time constraints. We show in this section different communication systems that are all based on simple communication patterns.

11.4.1 The Kippfahrleitung

The first commercial project where JOP had to prove that a Java processor is a valuable option for embedded real-time systems was a distributed motor control system.

In rail cargo, a large amount of time is spent on loading and unloading of goods wagons. The contact wire above the wagons is the main obstacle. Balfour Beatty Austria developed and patented a technical solution, the so-called *Kippfahrleitung*, to tilt up the contact wire. Figure 11.2 shows the construction of the mechanical tilt system driven by an asynchronous motor (just below the black tube). The little box mounted on the mast contains the control system. The black cable is the network interconnection of all control systems. In Figure 11.3 the same mast is shown with the contact wire tilted up.

The contact wire is tilted up on a distance of up to one kilometer. For a maximum distance of 1 km the whole system consists of 12 masts. Each mast is tilted by an asynchronous motor. However, the individual motors have to be synchronized so the tilt is performed in a smooth way. The maximum difference of the position of the contact wire is 10 cm. Therefore, a control algorithm has to slow down the faster motors.

Hardware

Each motor is controlled by its own embedded system (as seen in Figure 11.2) by silicon switches. The system measures the position of the arm with two end sensors and a revolving sensor. It also supervises the supply voltage and the amount of current through the motor. Those values are transmitted to the base station.

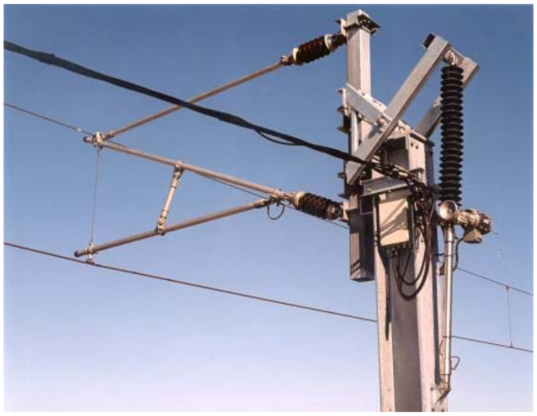


Figure 11.2: A *Kippfahrleitung* mast in down position

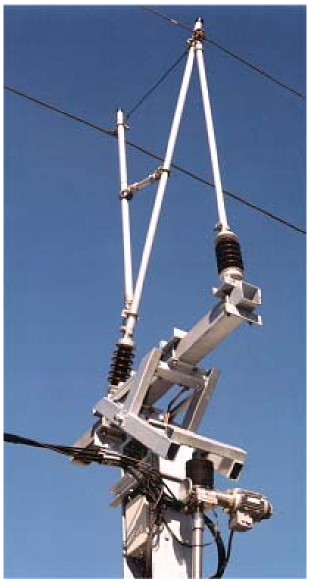


Figure 11.3: The mast in the up position with the tilted contact wire

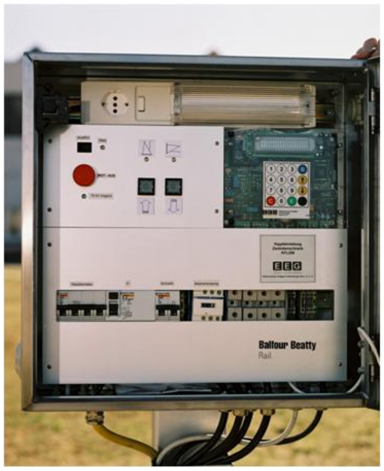


Figure 11.4: The base station with the operator interface

The base station, shown in Figure 11.4, provides the user interface for the operator via a simple display and a keyboard. It is usually located at one end of the line. The base station acts as master and controls the deviation of individual positions during the tilt. In technical terms, this is a distributed, embedded real-time control system, communicating over a shared network. The communication bus (up to one kilometer) is attached via an isolated RS485 data interface.

Although this system is not a mass product, there are nevertheless cost constraints. Even a small FPGA is more expensive than a general purpose CPU. To compensate for this, additional chips for the memory and the FPGA configuration were optimized for cost. One standard 128 KB Flash is used to hold FPGA configuration data, the Java program and a logbook. External main memory is reduced to 128 KB with an 8-bit data bus. Furthermore, all peripheral components, such as two UARTS, four sigma delta ADCs, and I/O ports are integrated in the FPGA.

Five silicon switches in the power line are controlled by the application program. A wrong setting of the switches due to a software error could result in a short circuit. Simple logic in the FPGA (coded in VHDL) can enforce the proper conditions for the switches. The sigma-delta ADCs are used to measure the temperature of the silicon switches and the current through the motor.

```
private static void forever() {

    for (;;) {
        Msg.loop();
        Triac.loop();
        if (Msg.available) {
            handleMsg();
        } else {
            chkMsgTimeout();
        }
        handleWatchDog();
        Timer.waitForNextInterval();
    }
}
```

Listing 11.1: The cyclic executive (simplified version)

Software Architecture

The main task of the program is to measure the position using the revolving sensor and to communicate with the base station under real-time constraints. The conservative style of a cyclic executive was chosen for the application. At application start all data structures are allocated and initialized. In the mission phase no allocation takes place and the cyclic executive loop is entered and never exited. The simple infinite loop, unblocked at constant time intervals, is shown in Listing 11.1. At the time the application was developed no static WCET analysis tool for Java was available. The actual execution time was measured and the maximum values have been recorded regularly. The loop and communication periods have been chosen to leave slack fur unexpected execution time variations. However, the application code and the Java processor are fully WCET analyzable, as shown later [134]. The application is used in Chapter 6 as a test case for the WCET analysis tool.

No interrupts or direct memory access (DMA) devices that can influence the execution time are used in the simple system. All sensors and the communication port are polled in the cyclic executive.

Communication

Communication is based on a master/slave model. Only the base station (the master) is allowed to send a request to a single mast station. This station is then required to reply within bounded time. The master handles timeout and retry. If an irrecoverable error occurs, the base station switches off the power for all mast stations, including the power supplies to the motors. This is the safe state of the whole system.

In a master/slave protocol no media access protocol is needed. In the case of a failure in the slave that delays a message collision can occur. The collision is detected by a violation of the message CRC. Spurious collisions are tolerated due to the retry of the base station. If the RS485 link is broken and only a subset of the mast stations reply the base station, the base station switches of the power supply for the whole system.

On the other hand the mast stations supervise the base station. The base station is required to send the requests on a regular basis. If this requirement is violated, each mast station switches off its motor. The local clocks are not synchronized. The mast stations measure the time elapsed since the last request from the base station and locally switch off based on a timeout.

The maximum distance of 1 km determines the maximum baud rate of the RS485 communication network. The resulting 12 masts on such a long line determine the number of packets that have to be sent in one master/slave round. Therefore, the pressure is high on the packet length. The data is exchanged in small packets of four bytes, including a one-byte CRC. To simplify the development, commands to reprogram the Flash in the mast stations and to force a reset are included. Therefore, it is possible to update the program, or even change the FPGA configuration, over the network.

11.4.2 The SCADA Device TeleAlarm

TeleAlarm (TAL) is a typical remote terminal unit of a supervisory control and data acquisition (SCADA) system. It is used by the Lower Austria's energy provider EVN (electricity, gas, and heating) to monitor the distribution plant. TeleAlarm also includes output ports for remote control of gas valves.

Hardware

The TAL device consists of a CPU FPGA module and an I/O board. The FPGA module contains an Altera Cyclone device, 1 MB static memory, 512 KB Flash, and 32 MB NAND Flash. The I/O board contains several EMC protected digital input and output ports, two 20 mA input ports, Ethernet connection, and a serial interface. Furthermore, the device

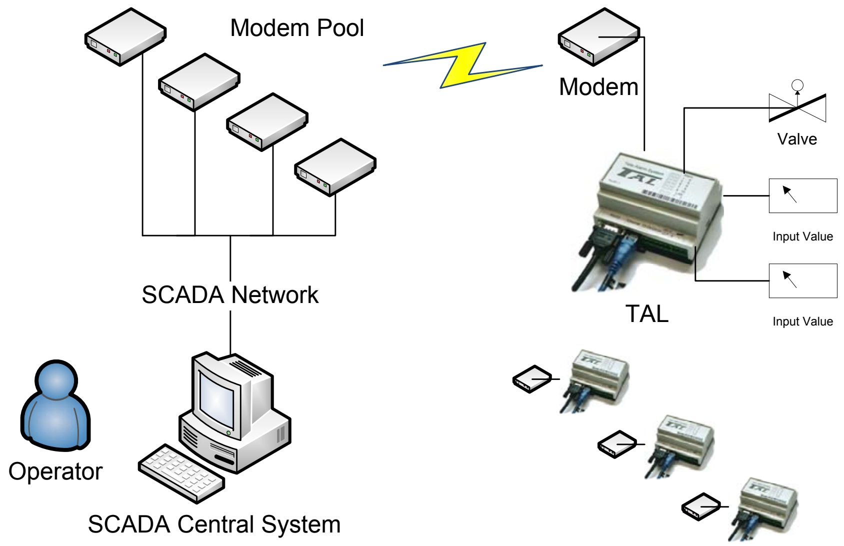


Figure 11.5: EVN SCADA system with the modem pool and TALs as remote terminal units

performs loading of a rechargeable battery to survive power down failures. On power down, an important event for a energy provider, an alarm is sent. The rechargeable battery is also monitored and the device switches itself off when the minimal voltage threshold is reached. This event is sent to the SCADA system before the power is switched off.

The same hardware is also used for a different project: a lift control in an automation factory in Turkey. The simple lift control software is now used as a test case for WCET tool development (see Chapter 6).

Communication

The communication between the TAL and the main supervisory control system is performed with a proprietary protocol. On a value change, the TAL sends the new data to the central system. Furthermore, the remote units are polled by the central system at a regular base. The TAL itself also sends the actual state regularly. TAL can communicate via Internet/Ethernet, a modem, and via SMS to a mobile phone.

EVN uses a mixture of dial-up network and leased lines for the plant communication. The dial-up modems are hosted by EVN itself. For safety and security reason there is no connection between the control network and the office network or the Internet.

Figure 11.5 shows the SCADA system setup at EVN. Several TALs are connected via modems to the central modem pool. The modem pool itself is connected to the central server. It has to be noted that there are many more TALs in the field than modems in the pool. The communication is usually very short (several seconds) and performed on demand and on a long regular interval. Not shown in the figure are additional SCADA stations and other remote terminal units from other manufacturers.

11.4.3 Support for Single Track Railway Control

Another application of JOP is in a communication device with soft real-time properties – Austrian Railways' (ÖBB) new support system for single-track lines. The system helps the superintendent at the railway station to keep track of all trains on the track. He can submit commands to the engine drivers of the individual trains. Furthermore, the device checks the current position of the train and generates an alarm when the train enters a track segment without a clearance.

At the central station all track segments are administered and controlled. When a train enters a non-allowed segment all trains nearby are warned automatically. This warning generates an alarm at the locomotive and the engine driver has to perform an emergency stop.

Figure 11.6 gives an overview of the system. The display and command terminal at the railway station is connected to the Intranet of the railway company. On the right side of the figure a picture of the terminal that is connected to the Internet via GPRS and to a GPS receiver is shown. Each locomotive that enters the track is equipped with either one or two of those terminals.

It has to be noted that this system is not a safety-critical system. The communication over a public mobile phone network is not reliable and the system is not certified for safety. The intension is just to *support* the superintendent and the engine drivers.

Hardware

Each locomotive is equipped with a GPS receiver, a GPRS modem, and the communication device (terminal). The terminal is a custom made device. The FPGA module is the same as in TAL, only the I/O board is adapted for this application. The I/O board contains several serial line interfaces for the GPS receiver, the GPRS modem, debug and download, and display connection. Auxiliary I/O ports connected to relays are reserved for future use. A possible extension is to stop the train automatically.

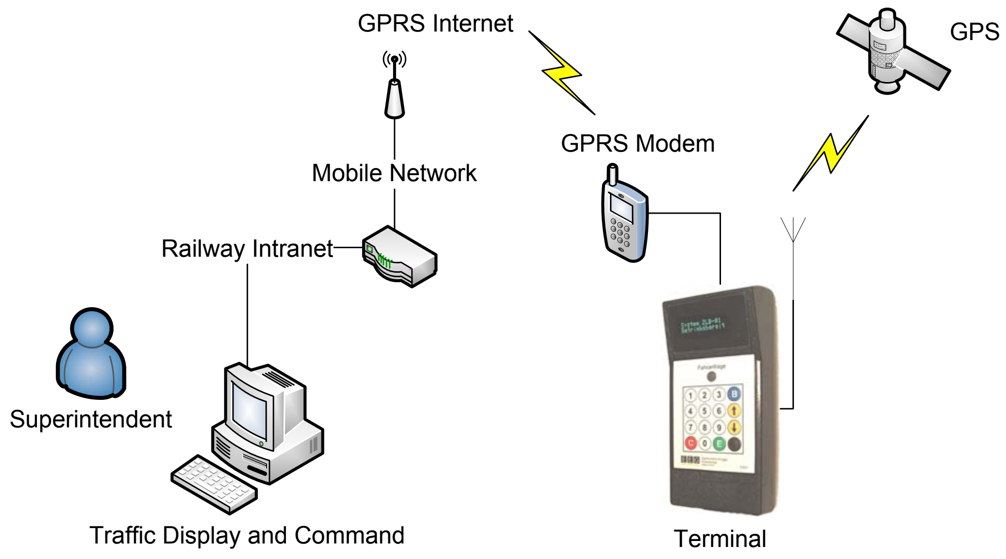


Figure 11.6: Support system for single track railway control for the Austrian railway company

Communication

The current position of the train is measured with GPS and the current track segment is calculated. The number of this segment is regularly sent to the central station. To increase the accuracy of the position, differential GPS correction data is transmitted to the terminal. The differential GPS data is generated by a ground base reference located at the central station.

The exchange of positions, commands, and alarm messages is performed via a public mobile phone network (via GPRS). The connection is secured via a virtual private network that is routed by the mobile network provider to the railway company's Intranet. The application protocol is command/response and uses UDP/IP as transport layer. Both systems (the central server and the terminal) can initiate a command. The system that sends the command is responsible for retries when no response arrives. The deadline for the communication of important messages is in the range of several seconds. After several non-successful retries the operator is informed about the communication error. He is than in charge to perform the necessary actions.

Besides the application specific protocol a TFTP server is implemented in the terminal.

It is used to update the track data for the position detection and to upload a new version of the software. The flexibility of the FPGA and an Internet connection to the embedded system allows to upgrade the software and even the processor in the field.

11.4.4 Communication and Common Design Patterns

Although we described embedded systems from quite different application domains we have been facing similar challenges. All systems are distributed systems and therefore need to communicate. Furthermore, they are real-time systems (at least with soft deadlines) and need to trust the communication and perform regular checks. The issues in the design of embedded real-time systems are quite similar in the three described projects. We found that several design patterns are used over and over and describe three of them in this section.

Master/Slave Designs

Developing safe embedded systems is an exercise in reducing complexity. One paradigm to simplify embedded software development is the master/slave pattern. Usually a single master is responsible to initiate commands to the slaves. The single master is also responsible to handle reliable communication. The master/slave pattern also fits very well with the command/response pattern for the communication.

Dealing with Communication Errors

Communication is not per se reliable. The RS485 link at the Kippfahrleitung operates in a rough environment and electromagnetic influences can lead to packet loss. The TAL system can suffer from broken phone lines. The single track control system operates on a public mobile phone network – a network without any guarantees for the GPRS data traffic. Therefore, we have to find solutions to operate in a safe and controlled manner the distributed system despite the chance of communication errors and failures.

Reliable communication is usually provided by the transport layer, TCP/IP in the case of the Internet. However, the timeouts in TCP/IP are way longer than the communication deadlines within control systems. The approach in all three presented projects is to use a datagram oriented protocol and perform the timeout and retransmission at the application level. To simplify the timeout handling a simple command and response pattern is used. One partner sends a command and expects the response within a specific time bound. The command initiator is responsible for retransmission after the timeout. The response partner just needs to reply to the command and does not need to remember the state of the communication. After several timeouts the communication error is handled by an upper layer.

Either the operator is informed (in the SCADA and the railway control system) or the whole system is brought into a safe state (in the motor control project).

Communication errors are either transient or longer lasting. Transient communication errors are lost packets due to network overload or external electromagnetic influences. In a command/response system the lost packets (either the command or the response) is detected by a timeout on the response. A simple retransmission of the command can correct those transient errors.

A longer network failure, e.g. caused by a wire break, can be detected by too many transmission retries. In such a case the system has to enter some form of safe state. Either the power is switched off or a human operator has to be informed. The individual timeout values and the number of retries depend, similar to thread periods, on the controlled environment. In the Kippfahrleitung the maximum timeout is in the millisecond range, whereas in the SCADA system the timeout is several minutes.

Software Update

Correction of implementation bugs during development can be very costly when physical access to the embedded system is necessary for a software update. Furthermore, a system is usually never really finished. When the system is in use the customer often finds new ways to enhance the system or requests additional features.

Therefore, an important feature of a networked embedded system is a software and parameter update in the field. In the first project the software update is performed via a home-made protocol. The other projects use the Internet protocol to some extent and therefore TFTP is a natural choice. TFTP is a very simple protocol that can be implemented within about 100 lines of code. It is applicable even in very small and resource constraint embedded devices.

11.4.5 Discussion

Writing embedded control software in Java is still not very common due to the lack of small and efficient implementations of the JVM. Our Java processor JOP is a solution for some embedded systems.

Using Java as the implementation language was a pleasure during programming and debugging. We did not waste many hours to hunt for pointer related bugs. The stricter (compared to C) type system of Java also catches many more programming errors at compile time. However, when using Java in a small embedded system one should not expect that a full blown Java library is available. Almost all of the code had to be written without library support. Embedded C programmers are aware of that fact, but Java programmers are new in the embedded domain and have to learn the difference between a PC and a 1 MB memory embedded system.

Up to date FPGAs in embedded control systems are only used for auxiliary functions or to implement high-performance DPS algorithm directly in hardware. Using the FPGA as the main processor is still not very common. However, combining the main processor with some peripheral devices in the same chip can simplify the PCB layout and also reduce the production cost. Furthermore, a field-reprogrammable hardware device offers a great deal of flexibility: When some part of the software becomes the bottleneck, an implementation of that function in hardware can be a solution. Leaving some headroom in the logic resources can extend the lifetime of the product.

For a prototype, JOP has been attached to a time-triggered network-on-chip [128]. It would be an interesting exercise to implement a JOP based node in a time-triggered distributed system as proposed by [73]. The combination of a real-time Java processor and a real-time network can ensure real-time characteristics for the whole system.

11.5 Summary

In this chapter, we presented an evaluation of JOP. We have seen that JOP is the smallest hardware realization of the JVM available to date. Due to the efficient implementation of the stack architecture, JOP is also smaller than a *comparable* RISC processor in an FPGA. Implemented in an FPGA, JOP has the highest clock frequency of all known Java processors.

We compared JOP against several embedded Java systems. JOP is about 40% faster than the commercial Java processor aJ100, but picoJava and a JIT/RISC combination are faster than JOP. These results show that a time-predictable architecture does not need to be slow, but will never be as fast as an architecture optimized for average case performance.

Furthermore, we have presented three industrial applications implemented in Java on an embedded, real-time Java processor. All projects included custom designed hardware (digital functions) and the central computation unit implemented in a single FPGA. The applications are written in pure Java without the need for native methods in C. Java proved to be a productive implementation language for embedded systems. Usage of JOP in four real-world applications showed that the processor is mature enough to be used in commercial projects.

12 Related Work

Several projects provide solutions to speedup execution of Java programs in embedded systems. Two different approaches can be found to improve Java bytecode execution by hardware. The first type operates as a Java coprocessor in conjunction with a general-purpose microprocessor. This coprocessor is placed in the instruction fetch path of the main processor and translates Java bytecodes to sequences of instructions for the host CPU or directly executes basic Java bytecodes. The complex instructions are emulated by the main processor. Java chips in the second category replace the general-purpose CPU. All applications therefore have to be written in Java. While the first type enables systems with mixed code capabilities, the additional component significantly raises costs. This chapter gives an overview of the most important Java processors and coprocessors from academia and industry.

12.1 Java Coprocessors

The simplest enhancement for Java is a translation unit, which substitutes the switch statement of an interpreter JVM (bytecode decoding) through hardware and/or translates simple bytecodes to a sequence of RISC instructions on the fly.

A standard JVM interpreter contains a loop with a large switch statement that decodes the bytecode (see Listing 3.1). This switch statement is compiled to an indirect branch. The destinations of these indirect branches change frequently and do not benefit from branch-prediction logic. This is the main overhead for simple bytecodes on modern processors. The following approaches enhance the execution of Java programs on a standard processor through the substitution of the memory read and switch statement with bytecode fetch and decode through hardware.

12.1.1 Jazelle

Jazelle [9] is an extension of the ARM 32-bit RISC processor, similar to the Thumb state (a 16-bit mode for reduced memory consumption). The Jazelle coprocessor is integrated into the same chip as the ARM processor. The hardware bytecode decoder logic is implemented in less than 12K gates. It accelerates, according to ARM, some 95% of the executed bytecodes. 140 bytecodes are executed directly in hardware, while the remaining 94 are emulated by sequences of ARM instructions. This solution also uses code modification with *quick* instructions to substitute certain object-related instructions after link resolution. All Java bytecodes, including the emulated sequences, are re-startable to enable a fast interrupt response time.

A new ARM instruction puts the processor into the Java state. Bytecodes are fetched and decoded in two stages, compared to a single stage in ARM state. Four registers of the ARM core are used to cache the top stack elements. Stack spill and fill is handled automatically by the hardware. Additional registers are reused for the Java stack pointer, the variable pointer, the constant pool pointer and locale variable 0 (the *this* pointer in methods). Keeping the complete state of the Java mode in ARM registers simplifies its integration into existing operating systems.

12.2 Java Processors

Java Processors are primarily used in an embedded system. In such a system, Java is the native programming language and all operating system related code, such as device drivers, are implemented in Java. Java processors are simple or extended stack architectures with an instruction set that resembles more or less the bytecodes from the JVM.

12.2.1 picoJava

Sun's picoJava is the Java processor used as a reference for new Java processors and as the basis for research into improving various aspects of a Java processor. Ironically, this processor was never released as a product by Sun. After Sun decided to not produce picoJava in silicon, Sun licensed picoJava to Fujitsu, IBM, LG Semicon and NEC. However, these companies also did not produce a chip and Sun finally provided the full Verilog code under an open-source license.

Sun introduced the first version of picoJava [90] in 1997. The processor was targeted at the embedded systems market as a pure Java processor with restricted support of C. picoJava-I contains four pipeline stages. A redesign followed in 1999, known as picoJava-II. This is the version described below. picoJava-II was freely available with a rich set of documentation [145, 146]. The probably first implementation of picoJava-II has been done by Wolfgang Puffitsch [103, 105]. This implementation enabled the comparison of JOP with picoJava-II in a similar FPGA (see Chapter 11)

The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions and is the most complex Java processor available. The processor can be implemented in about 440K gates [37]. Simple Java bytecodes are directly implemented in hardware, most of them execute in one to three cycles. Other performance critical instructions, for instance invoking a method, are implemented in microcode. picoJava traps on the remaining complex instructions, such as creation of an object, and emulates this instruction. A trap is rather expensive and has a minimum overhead of 16 clock cycles. This minimum value can only be achieved if the trap table entry is in the data cache and the first instruction of the trap routine is in the instruction cache. The worst-case trap latency is 926 clock cycles [146]. This great variation in execution times for a trap hampers tight WCET estimates.

picoJava provides a 64-entry stack cache as a register file. The core manages this register file as a circular buffer, with a pointer to the top of stack. The stack management unit automatically performs spill to and fill from the data cache to avoid overflow and underflow of the stack buffer. To provide this functionality the register file contains five memory ports. Computation needs two read ports and one write port, the concurrent spill and fill operations the two additional read and write ports. The processor core consists of following six pipeline stages:

Fetch: Fetch 8 bytes from the instruction cache or 4 bytes from the bus interface to the 16-byte-deep prefetch buffer.

Decode: Group and precode instructions (up to 7 bytes) from the prefetch buffer. Instruction folding is performed on up to four bytecodes.

Register: Read up to two operands from the register file (stack cache).

Execute: Execute simple instructions in one cycle or microcode for multi-cycle instructions.

Cache: Access the data cache.

Writeback: Write the result back into the register file.

The integer unit together with the stack unit provides a mechanism, called instruction folding, to speed up common code patterns found in stack architectures. When all entries are contained in the stack cache, the picoJava core can fold these four instructions into one RISC-style single cycle operation.

12.2.2 aJile JEMCore

aJile's JEMCore is a direct-execution Java processor that is available as both an IP core and a stand alone processor [2, 55]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. JEM2 is an enhanced version of JEM1, created in 1997 by the Rockwell-Collins Advanced Architecture Microprocessor group. Rockwell-Collins originally developed JEM for avionics applications by adapting an existing design for a stack-based embedded processor. Rockwell-Collins decided not to sell the chip on the open market. Instead, it licensed the design exclusively to aJile Systems Inc., which was founded in 1999 by engineers from Rockwell-Collins, Centaur Technologies, Sun Microsystems, and IDT.

The core contains 24 32-bit wide registers. Six of them are used to cache the top elements of the stack. The datapath consists of a 32-bit ALU, a 32-bit barrel shifter and the support for floating point operations (disassembly/assembly, overflow and NaN detection). The control store is a 4K by 56 ROM to hold the microcode that implements the Java bytecode. An additional RAM control store can be used for custom instructions. This feature is used to implement the basic synchronization and thread scheduling routines in microcode. It results in low execution overhead with a thread-to-thread yield in less than one μs (at 100 MHz). An optional Multiple JVM Manager (MJM) supports two independent, memory protected JVMs. The two JVMs execute time-sliced on the processor. According to aJile, the processor can be implemented in 25K gates (without the microcode ROM). The MJM needs additional 10K gates.

Two silicon versions of JEM exist today: the aJ-80 and the aJ-100. Both versions comprise a JEM2 core, the MJM, 48 KB zero wait state RAM and peripheral components, such as timer and UART. 16 KB of the RAM is used for the writable control store. The remaining 32 KB is used for storage of the processor stack. The aJ-100 provides a generic 8-bit, 16-bit or 32-bit external bus interface, while the aJ-80 only provides an 8-bit interface. The aJ-100 can be clocked up to 100 MHz and the aJ-80 up to 66 MHz. The power consumption is about 1mW per MHz.

aJile was a member of the initial Real-Time for Java Expert Group. However, up to now, no implementation of the RTSJ on top of the aJile processor emerged. One nice feature of this processor is its availability. Low-level access to devices via the RTSJ RawMemoryAccess objects has been shown on the aJile processor [54]. A relatively cheap development system, the JStamp [147], was used to compare this processor with JOP.

The aJile processor is intended as a solution for real-time systems. However, no information is available about bytecode execution times. As this processor is a commercial product and has been on the market for some time, it is expected that its JVM implementation confirms to Java standards, as defined by Sun.

12.2.3 Cjip

The Cjip processor [53, 70] supports multiple instruction sets, allowing Java, C, C++, and assembler to coexist. Internally, the Cjip uses 72 bit wide microcode instructions, to support the different instruction sets. At its core, Cjip is a 16-bit CISC architecture with on-chip 36 KB ROM and 18 KB RAM for fixed and loadable microcode. Another 1 KB RAM is used for eight independent register banks, string buffer and two stack caches. Cjip is implemented in 0.35-micron technology and can be clocked up to 66 MHz. The logic core consumes about 20% of the 1.4-million-transistor chip. The Cjip has 40 program controlled I/O pins, a high-speed 8 bit I/O bus with hardware DMA and an 8/16 bit DRAM interface.

The JVM is implemented largely in microcode (about 88% of the Java bytecodes). Java thread scheduling and garbage collection are implemented as processes in microcode. Microcode is also used to implement virtual peripherals such as watchdog timers, display and keyboard interfaces, sound generators, and multimedia codecs.

Microcode instructions execute in two or three cycles. A JVM bytecode requires several microcode instructions. The Cjip Java instruction set and the extensions are described in detail in [69]. For example: a bytecode nop executes in 6 cycles while an iadd takes 12 cycles. Conditional bytecode branches are executed in 33 to 36 cycles. Object oriented instructions, such getfield, putfield, or invokevirtual are not part of the instruction set.

12.2.4 Lightfoot

The Lightfoot 32-bit core [35] is a hybrid 8/32-bit processor based on the Harvard architecture. Program memory is 8 bits wide and data memory is 32 bits wide. The core contains a 3-stage pipeline with an integer ALU, a barrel shifter, and a 2-bit multiply step unit. There are two different stacks with the top elements implemented as registers and memory extension. The data stack is used to hold temporary data – it is not used to implement the JVM stack frame. As the name implies, the return stack holds return addresses for subroutines and it can be used as an auxiliary stack. The processor architecture specifies three different instruction formats: soft bytecodes, non-returnable instructions, and single-byte instructions that can be folded with a return instruction. The core is available in VHDL and can be implemented in less than 30K gates. Lightfood is now part of the VS2000 Typhoon Family Microcontroller.[1]

[1] http://www.velocitysemi.com/processors.htm

12.2.5 LavaCORE

LavaCORE [36] is another Java processor targeted at Xilinx FPGA architectures.[2] It implements a set of instructions in hardware and firmware. Floating-point operations are not implemented. A 32x32-bit dual-ported RAM implements a register-file. For specialized embedded applications, a tool is provided to analyze which subset of the JVM instructions is used. The unused instructions can be omitted from the design. The core can be implemented in 1926 CLBs (= 3800 LCs) in a Virtex-II (2V1000-5) and runs at 20 MHz.

12.2.6 Komodo, jamuth

Komodo [158] is a multithreaded Java processor with a four-stage pipeline. It is intended as a basis for research on real-time scheduling on a multithreaded microcontroller [76]. Simple bytecodes are directly implemented, while more complex bytecodes, such as iaload, are implemented as a microcode sequence. The unique feature of Komodo is the instruction fetch unit with four independent program counters and status flags for four threads. A priority manager is responsible for hardware real-time scheduling and can select a new thread after each bytecode instruction. The follow-up project, jamuth [149], is a commercial version of Komodo.

Komodo's multithreading is similar to hyper-threading in modern processors that are trying to hide latencies in instruction fetching. However, this feature leads to very pessimistic WCET values (in effect rendering the performance gain useless). The fact that the pipeline clock is only a quarter of the system clock also wastes a considerable amount of potential performance.

12.2.7 FemtoJava

FemtoJava [71] is a research project to build an application specific Java processor. The bytecode usage of the embedded application is analyzed and a customized version of FemtoJava is generated. FemtoJava implements up to 69 bytecode instructions for an 8 or 16 bit datapath. These instructions take 3, 4, 7 or 14 cycles to execute. Analysis of small applications (50 to 280 byte code) showed that between 22 and 69 distinct bytecodes are used. The resulting resource usage of the FPGA varies between 1000 and 2000 LCs. With the reduction of the datapath to 16 bits the processor is not Java conformant.

[2] `http://www.lavacore.com/`

12.2.8 jHISC

The jHISC project [148] proposes a high-level instruction set architecture for Java. This project is closely related to picoJava. The processor consumes 15500 LCs in an FPGA and the maximum frequency in a Xilinx Virtex FPGA is 30 MHz. According to [148] the prototype can only run simple programs and the performance is estimated with a simulation. In [155] the clocks per instruction (CPI) values for jHISC are compared against picoJava and JOP. However, it is not explained with which application the CPI values are collected. We assume that the CPI values for picoJava and JOP are derived from the manual and do not include any effects of pipeline stalls or cache misses.

12.2.9 SHAP

The SHAP Java processor [157], although now with a different pipeline structure and hardware assisted garbage collection, has its roots in the JOP design. SHAP is enhanced with a hardware object manager. That unit redirects field and array access during a copy operation of the GC unit. SHAP also implements the method cache [102].

12.2.10 Azul

Azul Systems provides an impressive multiprocessor system for transactions oriented server workloads [15]. A single Vega chip contains 54 64-bit RISC cores, optimized for the execution of Java programs. Up to 16 Vega processors can be combined to a cache coherent multiprocessor system with 864 processors cores, supporting up to 768 GB of shared memory.

13 Summary

In this chapter we will undertake a short review of the project and summarize the contributions. Java for real-time systems is a new and active research area. This chapter offers suggestions for future research, based on the described Java processor.

The research contributions made by this work are related to two areas: real-time Java and resource-constrained embedded systems.

13.1 A Real-Time Java Processor

The goal of time-predictable execution of Java programs was a first-class guiding principle throughout the development of JOP:

- The execution time for Java bytecodes can be exactly predicted in terms of the number of clock cycles. JOP is therefore a straightforward target for low-level WCET analysis. There is no mutual dependency between consecutive bytecodes that could result in unbounded timing effects.

- In order to provide time-predictable execution of Java bytecodes, the processor pipeline is designed without any prefetching or queuing. This fact avoids hard-to-analyze and possibly unbounded pipeline dependencies. There are no pipeline stalls, caused by interrupts or the memory subsystem, to complicate the WCET analysis.

- A pipelined processor architecture calls for higher memory bandwidth. A standard technique to avoid processing bottlenecks due to the higher memory bandwidth is caching. However, standard cache organizations improve the average execution time but are difficult to predict for WCET analysis. Two time-predictable caches are implemented in JOP: a *stack cache* as a substitution for the data cache and a *method cache* to cache the instructions.

  As the stack is a heavily accessed memory region, the stack – or part of it – is placed in local memory. This part of the stack is referred to as the *stack cache* and described in Section 4.4. Fill and spill of the stack cache is subjected to microcode control and therefore time-predictable.

In Section 4.5, a novel way to organize an instruction cache, as *method cache*, is given. The cache stores complete methods, and cache misses only occur on method invocation and return. Cache block replacement depends on the call tree, instead of instruction addresses. This *method cache* is easy to analyze with respect to worst-case behavior and still provides substantial performance gain when compared to a solution without an instruction cache.

- The time-predictable processor described above provides the basis for real-time Java. To enable real-time Java to operate on resource-constrained devices, a simple real-time profile was defined in Section 5.1 and implemented in Java on JOP. The beauty of this approach is in implementing functions usually associated with an RTOS in Java. This means that real-time Java is not based on an RTOS, and therefore not restricted to the functionality provided by the RTOS. With JOP, a self-contained real-time system in pure Java becomes possible.

  The tight integration of the scheduler and the hardware that generates schedule events results in low latency and low jitter of the task dispatch.

- The defined real-time profile suggests a new way to handle hardware interrupts to avoid interference between blocking device drivers and application tasks. Hardware interrupts other than the timer interrupt are represented as asynchronous events with an associated thread. These events are *normal* schedulable objects and subject to the control of the scheduler. With a minimum interarrival time, these events, and the associated device drivers, can be incorporated into the priority assignment and schedulability analysis in the same way as normal application tasks.

The contributions described above result in a time-predictable execution environment for real-time applications written in Java, without the resource implications and unpredictability of a JIT-compiler. The described processor architecture is a straightforward target for low-level WCET analysis.

Implementing a real-time scheduler in Java opens up new possibilities. The scheduler is extended to provide a framework for user-defined scheduling in Java. In Section **??**, we analyzed which events are exposed to the scheduler and which functions from the JVM need to be available in the user space. A simple-to-use framework to evaluate new scheduling concepts is given.

13.2 A Resource-Constrained Processor

Embedded systems are usually very resource-constrained. Using a low-cost FPGA as the main target technology forced the design to be small. The following architectural features address this issue:

- The architecture of JOP is best described as:

  > The JVM is a CISC stack architecture, whereas JOP is a RISC stack architecture.

  JOP contains its own instruction set, called microcode in this handbook, with a novel way of mapping bytecodes to microcode addresses. This mapping has zero overheads as described in Section 4.2. Basic bytecode instructions have a one-to-one mapping to microcode instructions and therefore execute in a single cycle. The stack architecture allows compact encoding of microinstructions in 8 bits to save internal memory.

  This approach allows flexible implementation of Java bytecodes in hardware, as a microcode sequence, or even in Java itself.

- The analysis of the JVM stack usage pattern in Section 4.4 led to the design of a resource-efficient two-level stack cache. This two-level stack cache fits to the embedded memory technologies of current FPGAs and ASICs and ensures fast execution of basic instructions.

  Part of the stack cache, which is implemented in an on-chip memory, is also used for microcode variables and constants. This resource sharing not only reduces the number of memory blocks needed for the processor, but also the number of data paths to and from the execution unit.

- Interrupts are considered hard to handle in a pipelined processor, resulting in a complex (and therefore resource consuming) implementation. In JOP, the above mentioned bytecode-microcode mapping is used in a clever way to avoid interrupt handling in the core pipeline. Interrupts generate special bytecodes that are inserted in a transparent way in the bytecode stream. Interrupt handlers can be implemented in the same way as bytecodes are implemented: in microcode or in Java.

The above design decisions where chosen to keep the size of the processor small without sacrificing performance. JOP is the smallest Java processor available to date that provides the basis for an implementation of the CLDC specification (see Section **??**). JOP is a fast execution environment for Java, without the resource implications and unpredictability of a JIT-compiler. The average performance of JOP is similar to that of mainstream, non real-time Java systems.

JOP is a flexible architecture that allows different configurations for different application domains. Therefore, size can be traded against performance. As an example, resource intensive instructions, such as floating point operations, can be implemented in Java. The flexibility of an FPGA implementation also allows adding application-specific hardware accelerators to JOP.

The small size of the processor allows the use of low-cost FPGAs in embedded systems that can compete against standard microcontroller. JOP has been implemented in several different FPGA families and is used in different real-world applications.

Programs for embedded and real-time systems are usually multi-threaded and a small design provides a path to a multi-processor system in a mid-sized FPGA or in an ASIC.

A tiny architecture also opens new application fields when implemented in an ASIC. Smart sensors and actuators, for example, are very sensitive to cost, which is proportional to the die area.

13.3 Future Work

JOP provides a basis for various directions for future research. Some suggestions are given below:

Real-time garbage collector: In Section 7, a real-time garbage collector was presented. Hardware support of a real-time GC would be an interesting topic for further research.

Another question that remains with a real-time GC is the analysis of the worst-case memory consumptions of tasks (similar to the WCET values), and scheduling the GC so that it can keep up with the allocation rate.

Hardware accelerator: The flexibility of an FPGA implementation of a processor opens up new possibilities for hardware accelerators. A further step would be to generate an application specific-system in which part of the application code is moved to hardware. Ideally, the hardware description should be extracted automatically from the Java source. Preliminary work in this area, using JOP as its basis, can be found in [48, 151].

Hardware scheduler: In JOP, scheduling and dispatch is done in Java (with some microcode support). For tasks with very short periods, the scheduling overheads can prove to be too high. A scheduler implemented in hardware can shorten this time, due to the parallel nature of the algorithm.

Instruction cache: The cache solution, described in Section 4.5, provides predictable instruction cache behavior while, in the average case, still performing in a similar way to a direct-mapped cache. However, an analysis tool for the worst-case behavior is still needed. With this tool, and a more complex analysis tool for traditional instruction caches, we also need to verify that the worst-case miss penalty is lower than with a traditional instruction cache.

A second interesting aspect of the method cache is the fact that the replacement decision on a cache miss only occurs on method invoke and return. The infrequency of this decision means that more time is available for more advanced replacement algorithms.

Real-time Java: Although there is already a definition for real-time Java, i.e. the RTSJ [25], this definition is not necessarily adequate. There is ongoing research on how memory should be managed for real-time Java applications: scoped memory, as suggested by the RTSJ, usage of a real-time GC, or application managed memory through memory pools. However, almost no research has been done into how the Java library, which is major part of Java's success, can be used in real-time systems or how it can be adapted to do so. The question of what the best memory management is for the Java standard library remains unanswered.

Java computer: How would a processor architecture and operating system architecture look in a 'Java only' system? Here, we need to rethink our approach to processes, protection, kernel- and user-space, and virtual memory. The standard approach of using memory protection between different processes is necessary for applications that are programmed in languages that use memory addresses as data, i.e. pointer usage and pointer manipulation. In Java, no memory addresses are visible and pointer manipulation is not possible. This very important feature of Java makes it a *safe* language. Therefore, an error-free JVM means we do not need memory protection between processes and we do not need to make a distinction between kernel and user space (with all the overhead) in a Java system. Another reason for using virtual addresses is link addresses. However, in Java this issue does not exist, as all classes are linked dynamically and the code itself (i.e. the bytecodes) only uses relative addressing.

Another issue here is the paging mechanism in a virtual memory system, which has to be redesigned for a Java computer. For this, we need to merge the virtual memory management with the GC. It does not make sense to have a virtual memory manager that works with plain (e.g. 4 KB) memory pages without knowledge about object lifetime. We therefore need to incorporate the virtual memory paging with a generational GC. The GC knows which objects have not been accessed for a long time and can be swapped out to the hard disk. Handling paging as part of the GC process also avoids page fault exceptions and thereby simplifies the processor architecture.

Another question is whether we can substitute the process notation with threads, or whether we need several JVMs on a Java only system. It depends. If we can live with the concept of shared static class members, we can substitute heavyweight processes with lightweight threads. It is also possible that we would have to define some further thread local data structures in the system.

It is the opinion of the author that Java is a promising language for future real-time systems. However, a number of issues remain to be solved. JOP, with its time-predictable execution of Java bytecodes, is an important part of a real-time Java system.

A Publications

2003

- Martin Schoeberl. Using a Java Optimized Processor in a Real World Application. In *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, pages 165–176, Austria, Vienna, June 2003.

- Martin Schoeberl. Design Decisions for a Java Processor. In *Tagungsband Austrochip 2003*, pages 115–118, Linz, Austria, October 2003.

- Martin Schoeberl. JOP: A Java Optimized Processor. In R. Meersman, Z. Tari, and D. Schmidt, editors, *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *Lecture Notes in Computer Science*, pages 346–359, Catania, Italy, November 2003. Springer.

2004

- Martin Schoeberl. Restrictions of Java for Embedded Real-Time Systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004.

- Martin Schoeberl. Design Rationale of a Processor Architecture for Predictable Real-Time Execution of Java Programs. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

- Martin Schoeberl. Real-Time Scheduling on a Java Processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

- Martin Schoeberl. Java Technology in an FPGA. In *Proceedings of the International Conference on Field-Programmable Logic and its applications (FPL 2004)*, Antwerp, Belgium, August 2004.
- Martin Schoeberl. A Time Predictable Instruction Cache for a Java Processor. In Robert Meersman, Zahir Tari, and Angelo Corsario, editors, *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *Lecture Notes in Computer Science*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

2005

- Flavius Gruian, Per Andersson, Krzysztof Kuchcinski, and Martin Schoeberl. Automatic generation of application-specific systems based on a micro-programmed java core. In *Proceedings of the 20th ACM Symposium on Applied Computing, Embedded Systems track*, Santa Fee, New Mexico, March 2005.
- Martin Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.
- Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.
- Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

2006

- Martin Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, Munich, Germany, March 2006.
- Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.
- Martin Schoeberl. Instruction Cache für Echtzeitsysteme, April 2006. Austrian patent AT 500.858.

- Rasmus Pedersen and Martin Schoeberl. An embedded support vector machine. In *Proceedings of the Fourth Workshop on Intelligent Solutions in Embedded Systems (WISES 2006)*, pages 79–89, Jun. 2006.
- Rasmus Pedersen and Martin Schoeberl. Exact roots for a real-time garbage collector. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*, Paris, France, October 2006.
- Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*, Paris, France, October 2006.

2007

- Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P. Ravn. A profile for safety critical java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 94–101, Santorini Island, Greece, May 2007. IEEE Computer Society.
- Martin Schoeberl. Mission modes for safety critical java. In *5th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems*, May 2007.
- Raimund Kirner and Martin Schoeberl. Modeling the function cache for worst-case execution time analysis. In *Proceedings of the 44rd Design Automation Conference, DAC 2007*, San Diego, CA, USA, June 2007. ACM.
- Martin Schoeberl. A time-triggered network-on-chip. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.
- Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.
- Wolfgang Puffitsch and Martin Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

- Martin Schoeberl. Architecture for object oriented programming languages. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.
- Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.
- Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.
- Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

2008

- Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.
- Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, St. Louis, MO, United States, April 2008.
- Martin Schoeberl, Stephan Korsholm, Christian Thalinger, and Anders P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.
- Stephan Korsholm, Martin Schoeberl, and Anders P. Ravn. Interrupt Handlers in Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.
- Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond Klefstad. Toward libraries for real-time Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

- Christof Pitter and Martin Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, Jun. 2008.
- Martin Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, July 2008.
- Peter Puschner and Martin Schoeberl. On composable system timing, task timing, and WCET analysis. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Prague, Czech Republic, July 2008.
- Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Number ISBN 978-3-8364-8086-4. VDM Verlag Dr. Müller, July 2008.
- Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, September 2008.
- Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, September 2008.
- Walter Binder, Martin Schoeberl, Philippe Moret, and Alex Villazon. Cross-profiling for embedded Java processors. In *Proceedings of the 5th International Conference on the Quantitative Evaluation of SysTems (QEST 2008)*, St Malo, France, September 2008.
- Walter Binder, Alex Villazon, Martin Schoeberl, and Philippe Moret. Cache-aware cross-profiling for Java processors. In *Proceedings of the 2008 international conference on Compilers, architecture, and synthesis forembedded systems (CASES 2008)*, Atlanta, Georgia, October 2008. ACM.

2009

- Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.
- Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

- Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.
- Florian Brandner, Tommy Thorn, and Martin Schoeberl. Embedded JIT compilation with CACAO on YARI. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.
- Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.
- Martin Schoeberl and Peter Puschner. Is chip-multiprocessing the end of real-time scheduling? In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, July 2009. OCG.
- Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based WCET analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, July 2009. OCG.
- Philippe Moret, Walter Binder, Martin Schoeberl, Alex Villazon, and Danilo Ansaloni. Analyzing performance and dynamic behavior of embedded Java software with calling-context cross-profiling. In *Proceedings of the 7th International Conference on the Principles and Practice of Programming in Java (PPPJ 2009)*, Calgary, Alberta, Canada, August 2009. ACM.
- Martin Schoeberl, Walter Binder, Philippe Moret, and Alex Villazon. Design space exploration for Java processors with cross-profiling. In *Proceedings of the 6th International Conference on the Quantitative Evaluation of SysTems (QEST 2009)*, Budapest, Hungary, September 2009. IEEE Computer Society.
- Philippe Moret, Walter Binder, Alex Villazon, Danilo Ansaloni, and Martin Schoeberl. Locating performance bottlenecks in embedded Java software with calling-context cross-profiling. In *Proceedings of the 6th International Conference on the Quantitative Evaluation of SysTems (QEST 2009)*, Budapest, Hungary, September 2009. IEEE Computer Society.

- Jack Whitham, Neil Audsley, and Martin Schoeberl. Using hardware methods to improve time-predictable performance in real-time java systems. In *Proceedings of the 7th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2009)*, Madrid, Spain, September 2009. ACM Press.

2010

- Martin Schoeberl and Wolfgang Puffitsch. Non-blocking real-time garbage collection. *Trans. on Embedded Computing Sys.*, accepted, 2010.
- Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *Trans. on Embedded Computing Sys.*, accepted, 2010.
- Walter Binder, Martin Schoeberl, Philippe Moret, and Alex Villazon. Cross-profiling for Java processors. *Software: Practice and Experience*, accepted, 2010.

B Acronyms

ADC	Analog to Digital Converter
ALU	Arithmetic and Logic Unit
ASIC	Application-Specific Integrated Circuit
BCET	Best Case Execution Time
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CLDC	Connected Limited Device Configuration
CPI	average Clock cycles Per Instruction
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
EDF	Earliest Deadline First
EMC	Electromagnetic Compatibility
ESD	Electrostatic Discharge
FIFO	Fist In, First Out
FPGA	Field Programmable Gate Array
GC	Garbage Collect(ion/or)
IC	Instruction Count
ILP	Instruction Level Parallelism
JOP	Java Optimized Processor
J2ME	Java2 Micro Edition
J2SE	Java2 Standard Edition
JDK	Java Development Kit
JIT	Just-In-Time
JVM	Java Virtual Machine
LC	Logic Cell
LRU	Least-Recently Used
MBIB	Memory Bytes read per Instruction Byte
MCIB	Memory Cycles per Instruction Byte

MP	Miss Penalty
MTIB	Memory Transactions per Instruction Byte
MUX	Multiplexer
OO	Object Oriented
OS	Operating System
RISC	Reduced Instruction Set Computer
RT	Real-Time
RTOS	Real-Time Operating System
RTSJ	Real-Time Specification for Java
SCADA	Supervisory Control And Data Acquisition
SDRAM	Synchronous DRAM
SRAM	Static Random Access Memory
TOS	Top Of Stack
UART	Universal Asynchronous Receiver/Transmitter
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
WCET	Worst-Case Execution Time

C JOP Instruction Set

The instruction set of JOP, the so-called microcode, is described in this appendix. Each instruction consists of a single instruction word (8 bits) without extra operands and executes in a single cycle[1]. Table C.1 lists the registers and internal memory areas that are used in the dataflow description.

Name	Description
A	Top of the stack
B	The element one below the top of stack
stack[]	The stack buffer for the rest of the stack
sp	The stack pointer for the stack buffer
vp	The variable pointer. Points to the first local in the stack buffer
ar	Address register for indirect stack access
pc	Microcode program counter
offtbl	Table for branch offsets
jpc	Program counter for the Java bytecode
opd	8 bit operand from the bytecode fetch unit
opd_{16}	16 bit operand from the bytecode fetch unit
memrda	Read address register of the memory subsystem
memwra	Write address register of the memory subsystem
memrdd	Read data register of the memory subsystem
memwrd	Write data register of the memory subsystem
mula, mulb	Operands of the hardware multiplier
mulr	Result register of the hardware multiplier
membcr	Bytecode address and length register of the memory subsystem
bcstart	Method start address register in the method cache

Table C.1: JOP hardware registers and memory areas

[1]The only multicycle instruction is wait and depends on the access time of the external memory

pop

Operation	Pop the top operand stack value
Opcode	`00000000`
Dataflow	$B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	pop
Description	Pop the top value from the operand stack.

and

Operation	Boolean AND int
Opcode	`00000001`
Dataflow	$A \wedge B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	iand
Description	Build the bitwise AND (conjunction) of the two top elements of the stack and push back the result onto the operand stack.

or

Operation	Boolean OR int
Opcode	`00000010`
Dataflow	$A \vee B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	ior
Description	Build the bitwise inclusive OR (disjunction) of the two top elements of the stack and push back the result onto the operand stack.

xor

Operation	Boolean XOR int
Opcode	`00000011`
Dataflow	$A \not\equiv B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	ixor
Description	Build the bitwise exclusive OR (negation of equivalence) of the two top elements of the stack and push back the result onto the operand stack.

add

Operation	Add int
Opcode	`00000100`
Dataflow	$A + B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	iadd
Description	Add the two top elements from the stack and push back the result onto the operand stack.

sub

Operation	Subtract int
Opcode	`00000101`
Dataflow	$A - B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	isub
Description	Subtract the two top elements from the stack and push back the result onto the operand stack.

stmul

Operation	Multiply int
Opcode	`00000110`
Dataflow	$A \rightarrow mula$ $B \rightarrow mulb$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	The top value from the stack is stored as first operand for the multiplier. The value one below the top of stack is stored as second operand for the multiplier. This operation starts the multiplier. The result is read with the ldmul instruction.

stmwa

Operation	Store memory write address
Opcode	`00000111`
Dataflow	$A \rightarrow memwra$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	The top value from the stack is stored as write address in the memory subsystem for a following stmwd.

stmra

Operation	Store memory read address
Opcode	`00001000`
Dataflow	$A \rightarrow memrda$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	The top value from the stack is stored as read address in the memory subsystem. This operation starts the concurrent memory read. The processor can continue with other operations. When the datum is needed a wait instruction stalls the processor till the read access is finished. The value is read with ldmrd.

stmwd

Operation Store memory write data

Opcode `00001001`

Dataflow $A \rightarrow memwrd$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The top value from the stack is stored as write data in the memory subsystem. This operation starts the concurrent memory write The processor can continue with other operations. The wait instruction stalls the processor till the write access is finished.

stald

Operation	Start array load
Opcode	`00001010`
Dataflow	$A \rightarrow memidx$ $B \rightarrow A$ $B \rightarrow memptr$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	xaload
Description	The top value from the stack is stored as array index, the next as reference in the memory subsystem. This operation starts the concurrent array load. The processor can continue with other operations. The wait instruction stalls the processor till the read access is finished. A null pointer or out of bounds exception is generated by the memory subsystem and thrown at the next bytecode fetch.

stast

Operation Start array store

Opcode `00001011`

Dataflow
$A \rightarrow memval$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$
$nextcycle$
$A \rightarrow memidx$
$B \rightarrow A$
$B \rightarrow memptr$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent xastore

Description In the first cycle the top value from the stack is stored as value into the memory subsystem. A microcode pop has to follow. In the second cycle the top value from the stack is stored as array index, the next as reference in the memory subsystem. This operation starts the concurrent array store. The processor can continue with other operations. The wait instruction stalls the processor till the write access is finished. A null pointer or out of bounds exception is generated by the memory subsystem and thrown at the next bytecode fetch.

stgf

Operation Start getfield

Opcode `00001100`

Dataflow $A \rightarrow memidx$
$B \rightarrow A$
$B \rightarrow memptr$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent getfield

Description The top value from the stack is stored as field index, the next as reference in the memory subsystem. This operation starts the concurrent getfield. The processor can continue with other operations. The wait instruction stalls the processor till the read access is finished. A null pointer exception is generated by the memory subsystem and thrown at the next bytecode fetch.

stpf

Operation Start putfield

Opcode `00001101`

Dataflow
$A \rightarrow memval$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$
$nextcycle$
$A \rightarrow memidx$
$B \rightarrow A$
$B \rightarrow memptr$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent putfield

Description In the first cycle the top value from the stack is stored as value into the memory subsystem. A microcode pop has to follow. In the second cycle the top value from the stack is stored as field index, the next as reference in the memory subsystem. This operation starts the concurrent putfield. The processor can continue with other operations. The wait instruction stalls the processor till the write access is finished. A null pointer exception is generated by the memory subsystem and thrown at the next bytecode fetch.

stcp

Operation Start copy step

Opcode `00001110`

Dataflow
$A \rightarrow memidx$
$B \rightarrow memsrc$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$
$nextcycle$
$B \rightarrow memdest$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description In the first cycle the top value from the stack is stored as index, the next as reference to which the index must be added for the source position. A microcode pop has to follow. In the second cycle the reference for the destination is stored in the memory subsystem. This operation starts the concurrent copy step. The processor can continue with other operations. The wait instruction stalls the processor till the write access is finished. The memory subsystem translates addresses such that copying of memory areas may be interrupted without affecting the consistency of data. Copying has to be done with increasing indices for correct operation. A negative index stops the address translation.

stbcrd

Operation	Start bytecode read
Opcode	`00001111`
Dataflow	$A \rightarrow membcr$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	The top value from the stack is stored as address and length of a method in the memory subsystem. This operation starts the memory transfer from the main memory to the bytecode cache (DMA). The processor can continue with other operations. The wait instruction stalls the processor till the transfer has finished. No other memory accesses are allowed during the bytecode read.

st<n>

Operation	Store 32-bit word into local variable
Opcode	`000100nn`
Dataflow	$A \rightarrow stack[vp+n]$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp-1 \rightarrow sp$
JVM equivalent	astore_<n>, istore_<n>, fstore_<n>
Description	The value on the top of the operand stack is popped and stored in the local variable at position *n*.

st

Operation	Store 32-bit word into local variable
Opcode	`00010100`
Dataflow	$A \rightarrow stack[vp+opd]$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp-1 \rightarrow sp$
JVM equivalent	astore, istore, fstore
Description	The value on the top of the operand stack is popped and stored in the local variable at position *opd*. *opd* is taken from the bytecode instruction stream.

stmi

Operation Store in local memory indirect

Opcode `00010101`

Dataflow
$A \rightarrow stack[ar]$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The top value from the operand stack is stored in the local memory (stack) at position ar.

stvp

Operation Store variable pointer

Opcode `00011000`

Dataflow
$A \rightarrow vp$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The value on the top of the operand stack is popped and stored in the variable pointer (`vp`).

stjpc

Operation	Store Java program counter
Opcode	`00011001`
Dataflow	$A \rightarrow jpc$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	The value on the top of the operand stack is popped and stored in the Java program counter (jpc).

star

Operation	Store adress register
Opcode	`00011010`
Dataflow	$A \rightarrow ar$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	The value on the top of the operand stack is popped and stored in the address register (ar). Due to a pipeline delay the register is valid on cycle later for usage by ldmi and stmi.

stsp

Operation Store stack pointer

Opcode `00011011`

Dataflow $A \rightarrow sp$
$B \rightarrow A$
$stack[sp] \rightarrow B$

JVM equivalent –

Description The value on the top of the operand stack is popped and stored in the stack pointer (sp).

ushr

Operation Logical shift rigth int

Opcode `00011100`

Dataflow $B >>> A \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent iushr

Description The values are popped from the operand stack. An int result is calculated by shifting the TOS-1 value rigth by s position, with zero extension, where s is the value of the low 5 bits of the TOS. The result is pushed onto the operand stack.

shl

Operation Shift left int

Opcode `00011101`

Dataflow
$B << A \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent ishl

Description The values are popped from the operand stack. An int result is calculated by shifting the TOS-1 value left by s position, where s is the value of the low 5 bits of the TOS. The result is pushed onto the operand stack.

shr

Operation Arithmetic shift rigth int

Opcode `00011110`

Dataflow
$B >> A \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent ishr

Description The values are popped from the operand stack. An int result is calculated by shifting the TOS-1 value rigth by s position, with sign extension, where s is the value of the low 5 bits of the TOS. The result is pushed onto the operand stack.

stm

Operation	Store in local memory
Opcode	`001nnnnn`
Dataflow	$A \rightarrow stack[n]$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	The top value from the operand stack is stored in the local memory (stack) at position n. These 32 memory destinations represent microcode local variables.

bz

Operation	Branch if value is zero
Opcode	`010nnnnn`
Dataflow	if $A = 0$ then $pc + offtbl[n] + 2 \rightarrow pc$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	If the top value from the operand stack is zero a microcode branch is taken. The value is popped from the operand stack. Due to a pipeline delay, the zero flag is delayed one cycle, i.e. the value from the last but one instruction is taken. The branch is followed by two branch delay slots. The branch offset is taken from the table $offtbl$ indexed by n.

bnz

Operation	Branch if value is not zero
Opcode	`011nnnnn`
Dataflow	if $A \neq 0$ then $pc + offtbl[n] + 2 \rightarrow pc$ $B \rightarrow A$ $stack[sp] \rightarrow B$ $sp - 1 \rightarrow sp$
JVM equivalent	–
Description	If the top value from the operand stack is not zero a microcode branch is taken. The value is popped from the operand stack. Due to a pipeline delay, the zero flag is delayed one cycle, i.e. the value from the last but one instruction is taken. The branch is followed by two branch delay slots. The branch offset is taken from the table $offtbl$ indexed by n.

nop

Operation	Do nothing
Opcode	`10000000`
Dataflow	$-$
JVM equivalent	nop
Description	The famous no operation instruction.

wait

Operation Wait for memory completion

Opcode `10000001`

Dataflow –

JVM equivalent –

Description This instruction stalls the processor until a pending memory instruction (stmra, stmwd or stbcrd) has completed. Two consecutive wait instructions are necessary for a correct stall of the decode and execute stage.

jbr

Operation Conditional bytecode branch and goto

Opcode `10000010`

Dataflow –

JVM equivalent ifnull, ifnonnull, ifeq, ifne, iflt, ifge, ifgt, ifle, if_acmpeq, if_acmpne, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, goto

Description Execute a bytecode branch or goto. The branch condition and offset are calculated in the bytecode fetch unit. Arguments must be removed with pop instructions in the following microcode instructions.

ldm

Operation Load from local memory

Opcode `101nnnnn`

Dataflow
$stack[n] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The value from the local memory (stack) at position n is pushed onto the operand stack. These 32 memory destinations represent microcode local variables.

ldi

Operation Load from local memory

Opcode `110nnnnn`

Dataflow
$stack[n+32] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The value from the local memory (stack) at position $n+32$ is pushed onto the operand stack. These 32 memory destinations represent microcode constants.

ldmrd

Operation Load memory read data

Opcode `11100010`

Dataflow
$memrdd \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The value from the memory system after a memory read is pushed onto the operand stack. This operation is usually preceded by two wait instructions.

ldmul

Operation Load multiplier result

Opcode `11100101`

Dataflow
$mulr \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent (imul)

Description The result of the multiplier is pushed onto the operand stack.

ldbcstart

Operation	Load method start
Opcode	`11100111`
Dataflow	$bcstart \rightarrow A$ $A \rightarrow B$ $B \rightarrow stack[sp+1]$ $sp+1 \rightarrow sp$
JVM equivalent	–
Description	The method start address in the method cache is pushed onto the operand stack.

ld<n>

Operation	Load 32-bit word from local variable
Opcode	`111010nn`
Dataflow	$stack[vp+n] \rightarrow A$ $A \rightarrow B$ $B \rightarrow stack[sp+1]$ $sp+1 \rightarrow sp$
JVM equivalent	aload_<n>, iload_<n>, fload_<n>
Description	The local variable at position *n* is pushed onto the operand stack.

ld

Operation Load 32-bit word from local variable

Opcode `11101100`

Dataflow
$stack[vp+opd] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent aload, iload, fload

Description The local variable at position *opd* is pushed onto the operand stack. *opd* is taken from the bytecode instruction stream.

ldmi

Operation Load from local memory indirect

Opcode `11101101`

Dataflow
$stack[ar] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The value from the local memory (stack) at position ar is pushed onto the operand stack.

ldsp

Operation	Load stack pointer
Opcode	`11110000`
Dataflow	$sp \rightarrow A$ $A \rightarrow B$ $B \rightarrow stack[sp+1]$ $sp+1 \rightarrow sp$
JVM equivalent	–
Description	The stack pointer is pushed onto the operand stack.

ldvp

Operation	Load variable pointer
Opcode	`11110001`
Dataflow	$vp \rightarrow A$ $A \rightarrow B$ $B \rightarrow stack[sp+1]$ $sp+1 \rightarrow sp$
JVM equivalent	–
Description	The variable pointer is pushed onto the operand stack.

ldjpc

Operation	Load Java program counter
Opcode	`11110010`
Dataflow	$jpc \rightarrow A$ $A \rightarrow B$ $B \rightarrow stack[sp+1]$ $sp+1 \rightarrow sp$
JVM equivalent	–
Description	The Java program counter is pushed onto the operand stack.

ld_opd_8u

Operation	Load 8-bit bytecode operand unsigned
Opcode	`11110100`
Dataflow	$opd \rightarrow A$ $A \rightarrow B$ $B \rightarrow stack[sp+1]$ $sp+1 \rightarrow sp$
JVM equivalent	–
Description	A single byte from the bytecode stream is pushed as `int` onto the operand stack.

ld_opd_8s

Operation Load 8-bit bytecode operand signed

Opcode `11110101`

Dataflow $opd \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent (bipush)

Description A single byte from the bytecode stream is sign-extended to an int and pushed onto the operand stack.

ld_opd_16u

Operation Load 16-bit bytecode operand unsigned

Opcode `11110110`

Dataflow $opd\_16 \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description A 16-bit word from the bytecode stream is pushed as int onto the operand stack.

ld_opd_16s

Operation Load 16-bit bytecode operand signed

Opcode `11110111`

Dataflow
$opd\_16 \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent (sipush)

Description A 16-bit word from the bytecode stream is sign-extended to an int and pushed onto the operand stack.

dup

Operation Duplicate the top operand stack value

Opcode `11111000`

Dataflow
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent dup

Description Duplicate the top value on the operand stack and push it onto the operand stack.

D Bytecode Execution Time

Table D.1 lists the bytecodes of the JVM with their opcode, mnemonics, the implementation type and the execution time on JOP. In the implementation column *hw* means that this bytecode has a microcode equivalent, *mc* means that a microcode sequence implements the bytecode, *Java* means the bytecode is implemented in Java, and a '-' indicates that this bytecode is not yet implemented. For bytecodes with a variable execution time the minimum and maximum values are given.

Opcode	Instruction	Implementation	Cycles
0	nop	hw	1
1	aconst_null	hw	1
2	iconst_m1	hw	1
3	iconst_0	hw	1
4	iconst_1	hw	1
5	iconst_2	hw	1
6	iconst_3	hw	1
7	iconst_4	hw	1
8	iconst_5	hw	1
9	lconst_0	mc	2
10	lconst_1	mc	2
11	fconst_0	Java	
12	fconst_1	Java	
13	fconst_2	Java	
14	dconst_0	-	
15	dconst_1	-	
16	bipush	mc	2
17	sipush	mc	3
18	ldc	mc	7+r
19	ldc_w	mc	8+r

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
20	ldc2_w[20]	mc	17+2*r
21	iload	mc	2
22	lload	mc	11
23	fload	mc	2
24	dload	mc	11
25	aload	mc	2
26	iload_0	hw	1
27	iload_1	hw	1
28	iload_2	hw	1
29	iload_3	hw	1
30	lload_0	mc	2
31	lload_1	mc	2
32	lload_2	mc	2
33	lload_3	mc	11
34	fload_0	hw	1
35	fload_1	hw	1
36	fload_2	hw	1
37	fload_3	hw	1
38	dload_0	mc	2
39	dload_1	mc	2
40	dload_2	mc	2
41	dload_3	mc	11
42	aload_0	hw	1
43	aload_1	hw	1
44	aload_2	hw	1
45	aload_3	hw	1
46	iaload[46]	hw	7+3*r
47	laload	mc	43+4*r
48	faload[46]	hw	7+3*r
49	daload	-	
50	aaload[46]	hw	7+3*r
51	baload[46]	hw	7+3*r
52	caload[46]	hw	7+3*r

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
53	saload[46]	hw	7+3*r
54	istore	mc	2
55	lstore	mc	11
56	fstore	mc	2
57	dstore	mc	11
58	astore	mc	2
59	istore_0	hw	1
60	istore_1	hw	1
61	istore_2	hw	1
62	istore_3	hw	1
63	lstore_0	mc	2
64	lstore_1	mc	2
65	lstore_2	mc	2
66	lstore_3	mc	11
67	fstore_0	hw	1
68	fstore_1	hw	1
69	fstore_2	hw	1
70	fstore_3	hw	1
71	dstore_0	mc	2
72	dstore_1	mc	2
73	dstore_2	mc	2
74	dstore_3	mc	11
75	astore_0	hw	1
76	astore_1	hw	1
77	astore_2	hw	1
78	astore_3	hw	1
79	iastore[79]	hw	10+2*r+w
80	lastore[1]	mc	48+2*r+2*w
81	fastore[79]	hw	10+2*r+w
82	dastore	-	
83	aastore	Java	
84	bastore[79]	hw	10+2*r+w
85	castore[79]	hw	10+2*r+w

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
86	sastore[79]	hw	10+2*r+w
87	pop	hw	1
88	pop2	mc	2
89	dup	hw	1
90	dup_x1	mc	5
91	dup_x2	mc	7
92	dup2	mc	6
93	dup2_x1	mc	8
94	dup2_x2	mc	10
95	swap[2]	mc	4
96	iadd	hw	1
97	ladd	mc	26
98	fadd	Java	
99	dadd	-	
100	isub	hw	1
101	lsub	mc	38
102	fsub	Java	
103	dsub	-	
104	imul	mc	35
105	lmul	Java	
106	fmul	Java	
107	dmul	-	
108	idiv	Java	
109	ldiv	Java	
110	fdiv	Java	
111	ddiv	-	
112	irem	Java	
113	lrem	Java	
114	frem	Java	
115	drem	-	
116	ineg	mc	4
117	lneg	mc	34
118	fneg	Java	

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
119	dneg	-	
120	ishl	hw	1
121	lshl	mc	28
122	ishr	hw	1
123	lshr	mc	28
124	iushr	hw	1
125	lushr	mc	28
126	iand	hw	1
127	land	mc	8
128	ior	hw	1
129	lor	mc	8
130	ixor	hw	1
131	lxor	mc	8
132	iinc	mc	8
133	i2l	mc	5
134	i2f	Java	
135	i2d	-	
136	l2i	mc	3
137	l2f	-	
138	l2d	-	
139	f2i	Java	
140	f2l	-	
141	f2d	-	
142	d2i	-	
143	d2l	-	
144	d2f	-	
145	i2b	Java	
146	i2c	mc	2
147	i2s	Java	
148	lcmp	Java	
149	fcmpl	Java	
150	fcmpg	Java	
151	dcmpl	-	

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
152	dcmpg	-	
153	ifeq	mc	4
154	ifne	mc	4
155	iflt	mc	4
156	ifge	mc	4
157	ifgt	mc	4
158	ifle	mc	4
159	if_icmpeq	mc	4
160	if_icmpne	mc	4
161	if_icmplt	mc	4
162	if_icmpge	mc	4
163	if_icmpgt	mc	4
164	if_icmple	mc	4
165	if_acmpeq	mc	4
166	if_acmpne	mc	4
167	goto	mc	4
168	jsr	*not used*	
169	ret	*not used*	
170	tableswitch[170]	Java	
171	lookupswitch[171]	Java	
172	ireturn[172]	mc	23+r+l
173	lreturn[173]	mc	25+r+l
174	freturn[172]	mc	23+r+l
175	dreturn[173]	mc	25+r+l
176	areturn[172]	mc	23+r+l
177	return[177]	mc	21+r+l
178	getstatic	mc	7+r
179	putstatic	mc	8+w
180	getfield	hw	11+2*r
181	putfield	hw	13+r+w
182	invokevirtual[182]	mc	98+4r+l
183	invokespecial[183]	mc	74+3*r+l
184	invokestatic[183]	mc	74+3*r+l

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
185	invokeinterface[185]	mc	112+6r+l
186	unused_ba	-	
187	new[187]	Java	
188	newarray[188]	Java	
189	anewarray	Java	
190	arraylength	mc	6+r
191	athrow[3]	Java	
192	checkcast	Java	
193	instanceof	Java	
194	monitorenter	mc	19
195	monitorexit	mc	22
196	wide	*not used*	
197	multianewarray[4]	Java	
198	ifnull	mc	4
199	ifnonnull	mc	4
200	goto_w	*not used*	
201	jsr_w	*not used*	
202	breakpoint	-	
203	reserved	-	
204	reserved	-	
205	reserved	-	
206	reserved	-	
207	reserved	-	
208	reserved	-	
209	jopsys_rd[209]	mc	4+r
210	jopsys_wr	mc	5+w
211	jopsys_rdmem	mc	4+r
212	jopsys_wrmem	mc	5+w
213	jopsys_rdint	mc	3
214	jopsys_wrint	mc	3
215	jopsys_getsp	mc	3
216	jopsys_setsp	mc	4
217	jopsys_getvp	hw	1

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
218	jopsys_setvp	mc	2
219	jopsys_int2ext[219]	mc	14+r+n*(23+w)
220	jopsys_ext2int[220]	mc	14+r+n*(23+r)
221	jopsys_nop	mc	1
222	jopsys_invoke	mc	
223	jopsys_cond_move	mc	5
224	getstatic_ref	mc	12+2*r
225	putstatic_ref	Java	
226	getfield_ref	mc	11+2*r
227	putfield_ref	Java	
228	getstatic_long	mc	
229	putstatic_long	mc	
230	getfield_long	mc	
231	putfield_long	mc	
232	jopsys_memcpy	mc	
233	reserved	-	
234	reserved	-	
235	reserved	-	
236	invokesuper	mc	-
237	reserved	-	
238	reserved	-	
239	reserved	-	
240	sys_int[240]	Java	
241	sys_exc[240]	Java	
242	reserved	-	
243	reserved	-	
244	reserved	-	
245	reserved	-	
246	reserved	-	
247	reserved	-	
248	reserved	-	
249	reserved	-	
250	reserved	-	

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
251	reserved	-	
252	reserved	-	
253	reserved	-	
254	sys_noimp	Java	
255	sys_init	*not used*	

Table D.1: Implemented bytecodes and execution time in cycles

Memory Timing

The external memory timing is defined in the top level VHDL file (e.g. jopcyc.vhd) with ram_cnt for the number of cycles for a read and write access. At the moment there is no difference for a read and write access. For the 100 MHz JOP with 15 ns SRAMs this access

[1]The exact value is given below.

[2]Not tested as javac does not emit the swap bytecode.

[3]A simple version that stops the JVM. No catch support.

[4]Only dimension 2 supported.

[20]The exact value is $17+\begin{cases} r-2 & : r>2 \\ 0 & : r\leq 2 \end{cases} + \begin{cases} r-1 & : r>1 \\ 0 & : r\leq 1 \end{cases}$

[46]The exact value is *no hidden wait states at the moment.*

[79]The exact value is *no hidden wait states at the moment.*

[170]tableswitch execution time depends to a great extent on the caching of the corresponding Java method or the memory transfer time for the method.

[171]lookupswitch execution time depends to a great extent on the caching of the corresponding Java method or the memory transfer time for the method. lookupswitch also depends on the argument as it performs a linear search in the jump table.

[172]The exact value is: $23+\begin{cases} r-3 & : r>3 \\ 0 & : r\leq 3 \end{cases} + \begin{cases} l-10 & : l>10 \\ 0 & : l\leq 10 \end{cases}$

[173]The exact value is: $25+\begin{cases} r-3 & : r>3 \\ 0 & : r\leq 3 \end{cases} + \begin{cases} l-11 & : l>11 \\ 0 & : l\leq 11 \end{cases}$

[177]The exact value is: $21+\begin{cases} r-3 & : r>3 \\ 0 & : r\leq 3 \end{cases} + \begin{cases} l-9 & : l>9 \\ 0 & : l\leq 9 \end{cases}$

[182]The exact value is: $100+2r+\begin{cases} r-3 & : r>3 \\ 0 & : r\leq 3 \end{cases} + \begin{cases} r-2 & : r>2 \\ 0 & : r\leq 2 \end{cases} + \begin{cases} l-37 & : l>37 \\ 0 & : l\leq 37 \end{cases}$

[183]The exact value is: $74+r+\begin{cases} r-3 & : r>3 \\ 0 & : r\leq 3 \end{cases} + \begin{cases} r-2 & : r>2 \\ 0 & : r\leq 2 \end{cases} + \begin{cases} l-37 & : l>37 \\ 0 & : l\leq 37 \end{cases}$

[185]The exact value is: $114+4r+\begin{cases} r-3 & : r>3 \\ 0 & : r\leq 3 \end{cases} + \begin{cases} r-2 & : r>2 \\ 0 & : r\leq 2 \end{cases} + \begin{cases} l-37 & : l>37 \\ 0 & : l\leq 37 \end{cases}$

[187]new execution time depends to a great extent on the caching of the corresponding Java method or the memory transfer time for the method. new also depends on the size of the created object as the memory for the object is filled with zeros – This will change with the GC

[188]newarray execution time depends to a great extent on the caching of the corresponding Java method or the memory transfer time for the method. newarray also depends on the size of the array as the memory for the object is filled with zeros – This will change with the GC

[209]The native instructions jopsys_rd and jopsys_wr are alias to the jopsys_rdmem and jopsys_wrmem instructions just for compatibility to existing Java code. I/O devices are now memory mapped. In the case for simple I/O devices there are no wait states and the exact values are 4 and 5 cycles respective.

[219]The exact value is $14+r+n(23+\begin{cases} w-8 & : w>8 \\ 0 & : w\leq 8 \end{cases})$. n is the number of words transferred.

[220]The exact value is $14+r+n(23+\begin{cases} r-10 & : r>10 \\ 0 & : r\leq 10 \end{cases})$. n is the number of words transferred.

time is two cycles (ram_cnt=2, 20 ns). Therefore the wait state n_{ws} is 1 (ram_cnt-1). A basic memory read in microcode is as follows:

```
stmra   // start read with address store
wait    //  fill  the pipeline with two
wait    // wait instructions
ldmrd   // push read result on TOS
```

In this sequence the *last* wait executes for $1+n_{ws}$ cycles. Therefore the whole read sequence takes $4+n_{ws}$ cycles. For the example with ram_cnt=2 this basic memory read takes 5 cycles.

A memory write in microcode is as follows:

```
stmwa   // store address
stmwd   // store data and start the write
wait    //  fill  the pipeline with wait
wait    // wait for the memory ready
```

The last wait again executes for $1+n_{ws}$ cycles and the basic write takes $4+n_{ws}$ cycles. For the native bytecode jopsys_wrmem an additional nop instruction for the nxt flag is necessary.

The read and write wait states r_{ws} and w_{ws} are:

$$r_{ws} = w_{ws} = \begin{cases} ram\_cnt - 1 & : \quad ram\_cnt > 1 \\ 0 & : \quad ram\_cnt \leq 1 \end{cases}$$

In the instruction timing we use r and w instead of r_{ws} and w_{ws}. The wait states can be hidden by other microcode instructions between stmra/wait and stmwd/wait. The exact value is given in the footnote.

Instruction Timing

The bytecodes that access memory are indicated by an r for a memory read and an w for a memory write at the cycles column (r and w are the additional wait states). The wait cycles for the memory access have to be added to the execution time. These two values are implementation dependent (clock frequency versus RAM access time, data bus width); for the Cyclone EP1C6 board with 15 ns SRAMs and 100 MHz clock frequency these values are both 1 cycle (ram_cnt-1).

For some bytecodes, part of the memory latency can be hidden by executing microcode during the memory access. However, these cycles can only be subtracted when the wait states (r or w) are larger then 0 cycles. The exact execution time with the subtraction of the saved cycles is given in the footnote.

Cache Load

Memory access time also determines the cache load time on a miss. For the current implementation the cache load time is calculated as follows: the wait state c_{ws} for a single word cache load is:

$$c_{ws} = \begin{cases} r_{ws} & : \quad r_{ws} > 1 \\ 1 & : \quad r_{ws} \le 1 \end{cases}$$

On a method invoke or return, the respective method has to be loaded into the cache on a cache miss. The load time l is:

$$l = \begin{cases} 6 + (n+1)(1 + c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cache hit} \end{cases}$$

where n is the size of the method in number of 32-bit words. For short methods, the load time of the method on a cache miss, or part of it, is hidden by microcode execution. The exact value is given in the footnote.

lastore

$$t_{lastore} = 48 + 2r_{ws} + w_{ws} + \begin{cases} w_{ws} - 3 & : \quad w_{ws} > 3 \\ 0 & : \quad w_{ws} \le 3 \end{cases}$$

E Printed Circuit Boards

This chapter provides the schematics of the Cycore FPGA boards and several I/O extension boards.

E.1 Cyclone FPGA Board

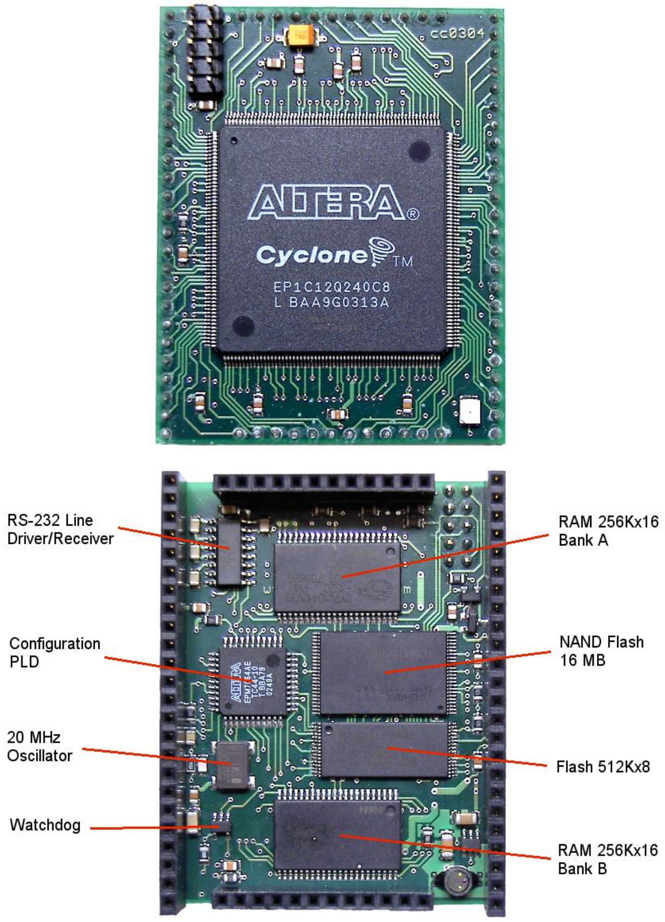


Figure E.1: Top and bottom side of the Cyclone FPGA board

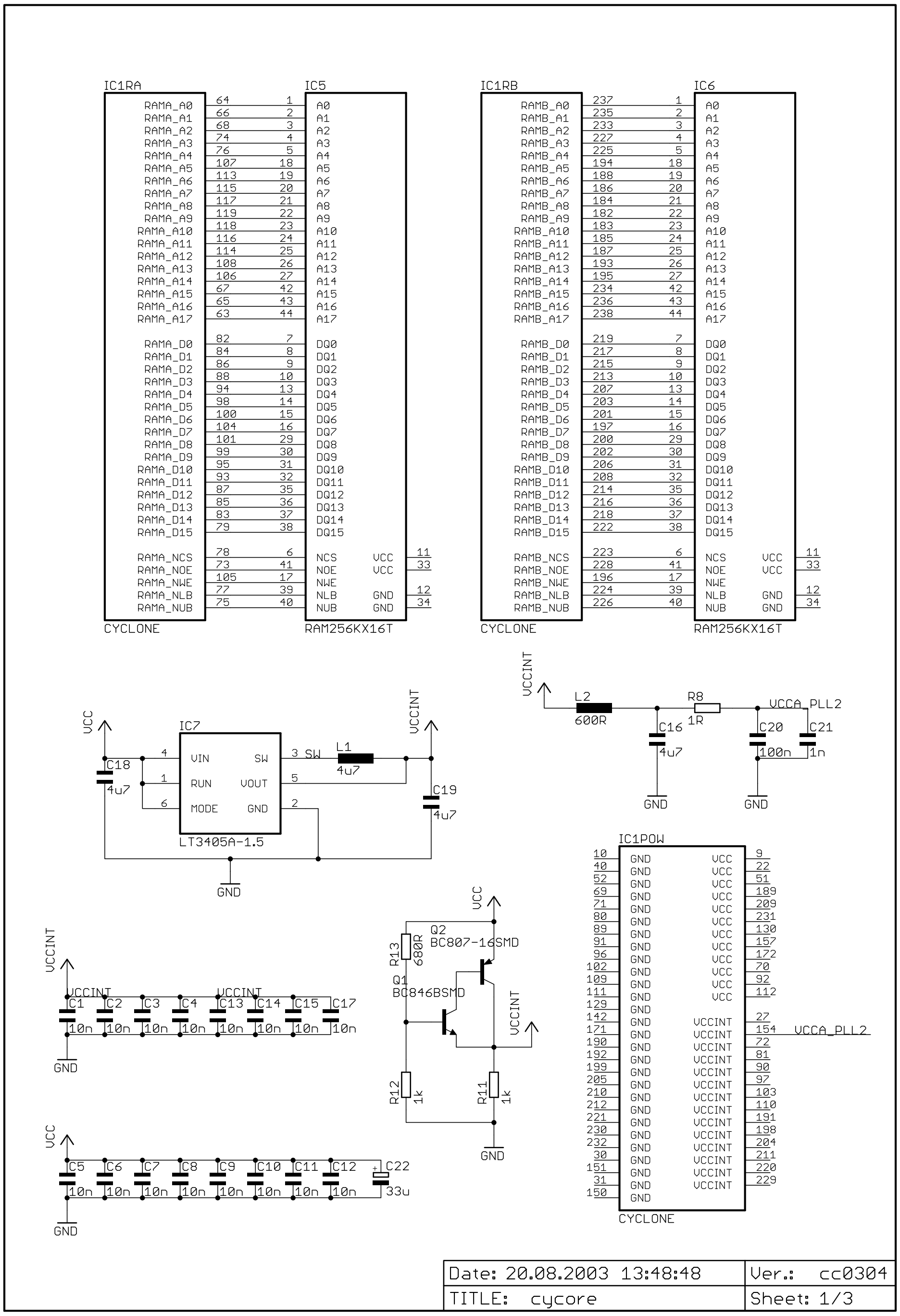


Figure E.2: Schematic of the Cyclone FPGA board, page 1

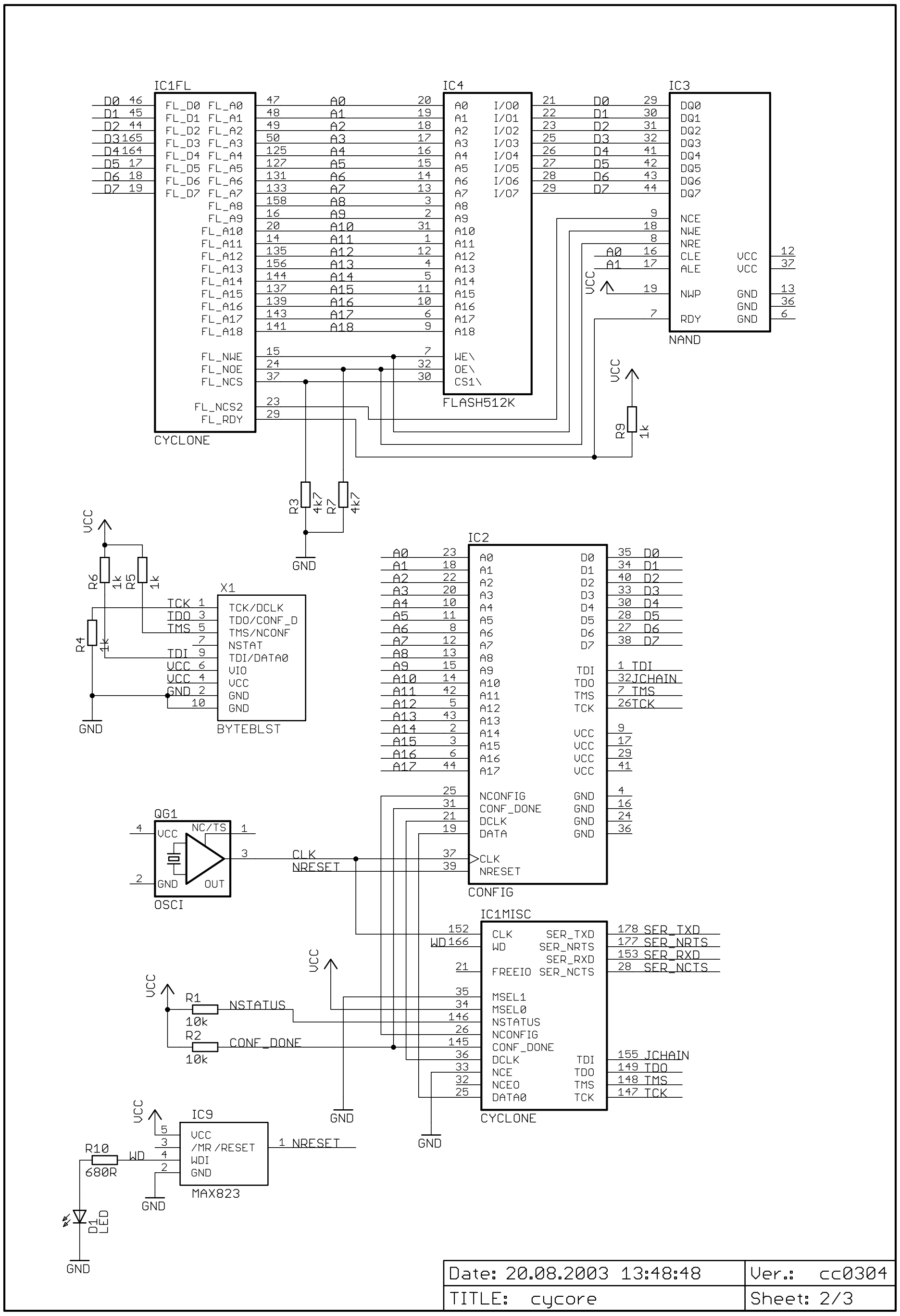


Figure E.3: Schematic of the Cyclone FPGA board, page 2

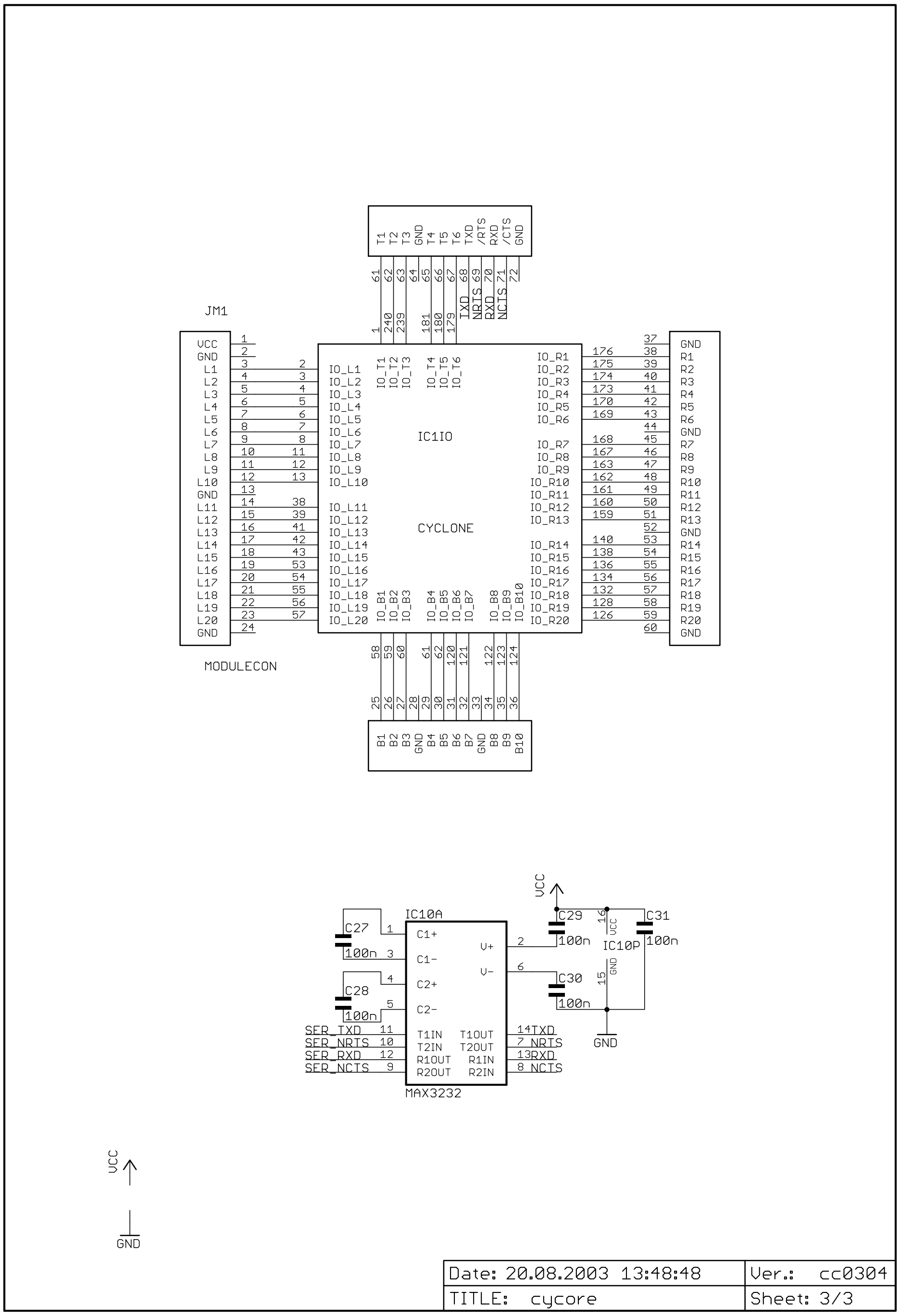


Figure E.4: Schematic of the Cyclone FPGA board, page 3

E.2 Baseio Board

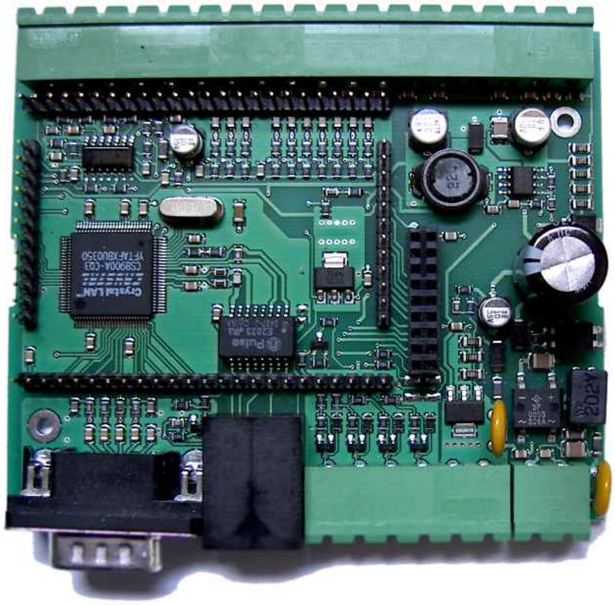


Figure E.5: The Baseio extension board

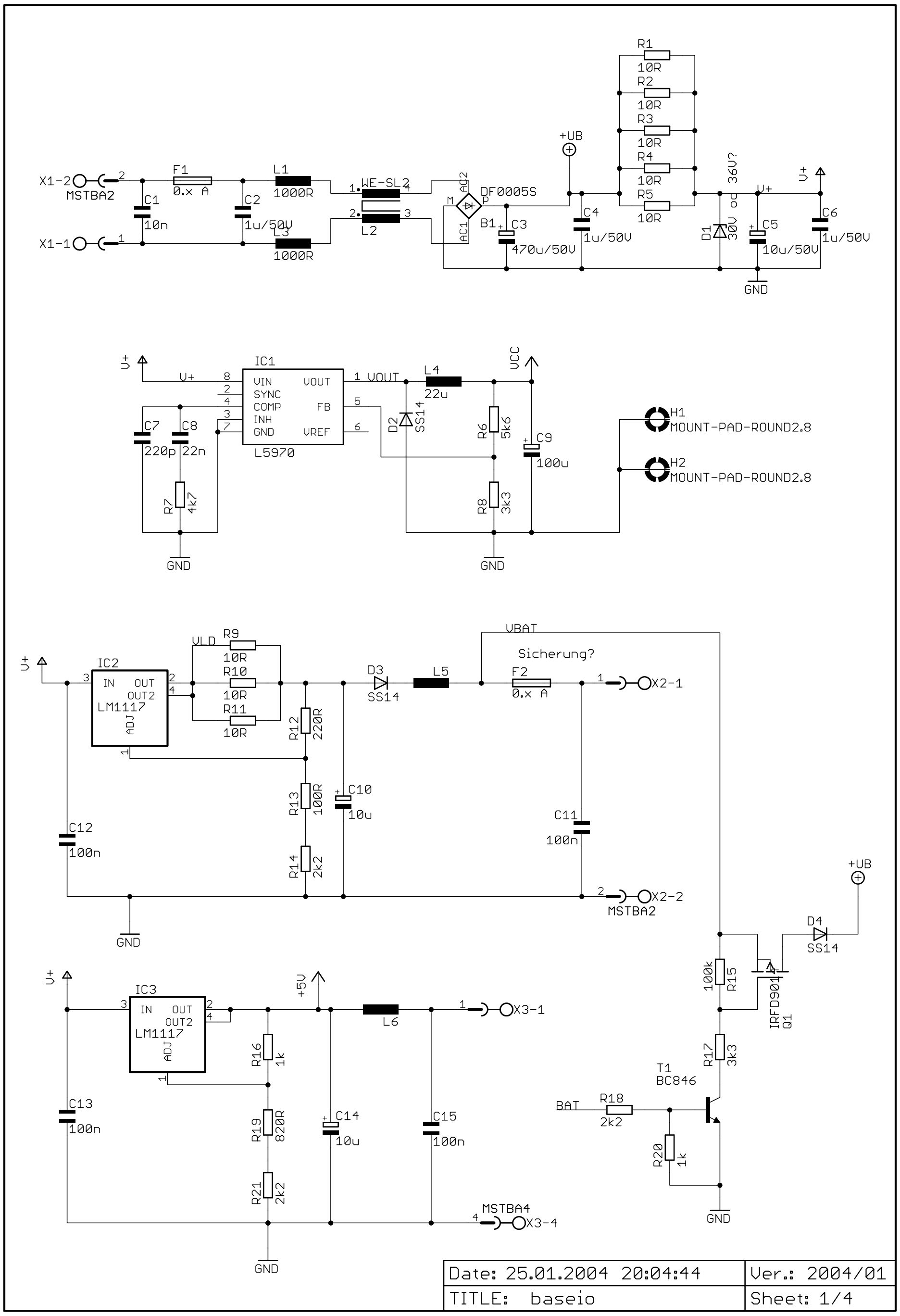


Figure E.6: Schematic of the Baseio extension board, page 1

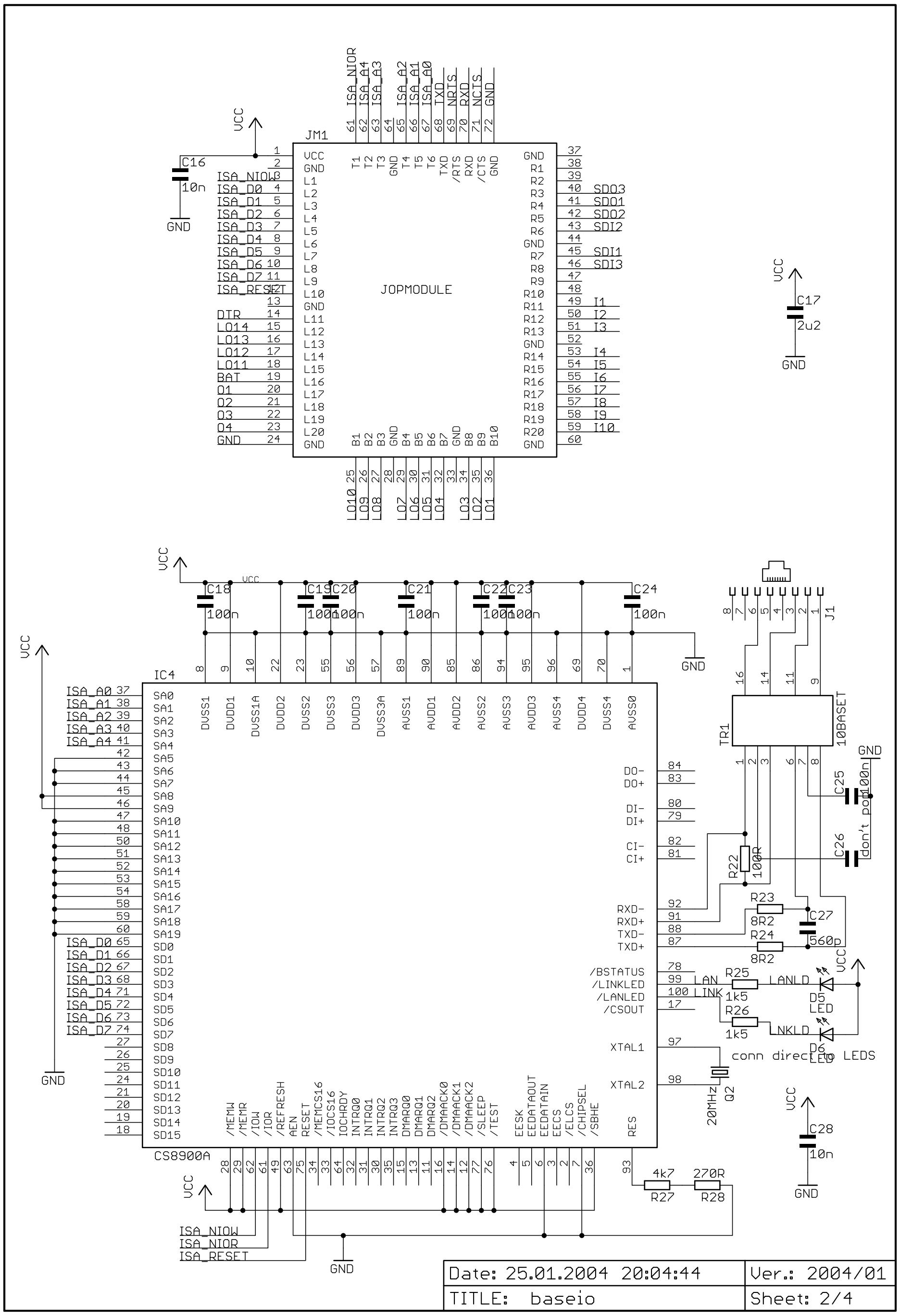


Figure E.7: Schematic of the Baseio extension board, page 2

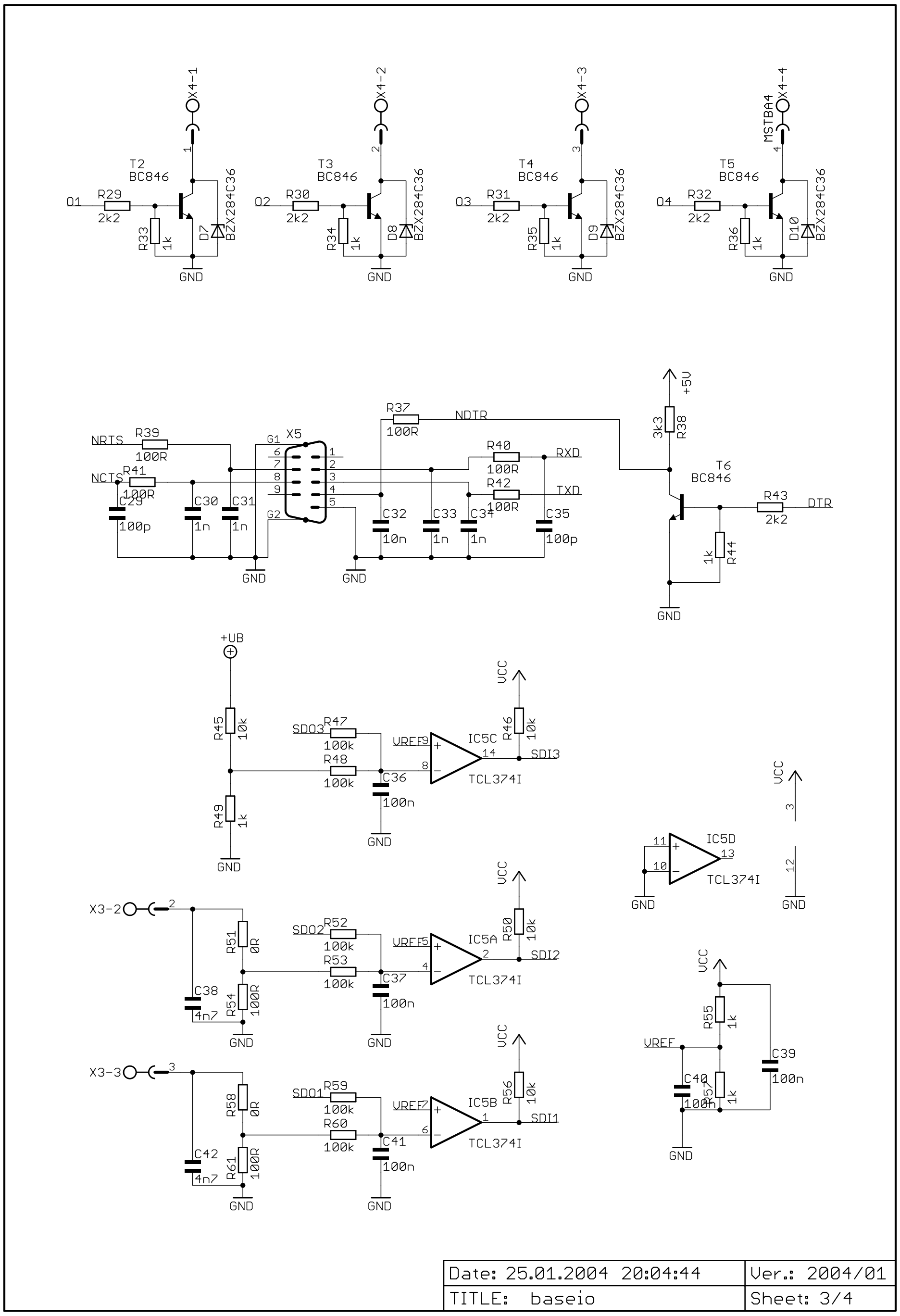


Figure E.8: Schematic of the Baseio extension board, page 3

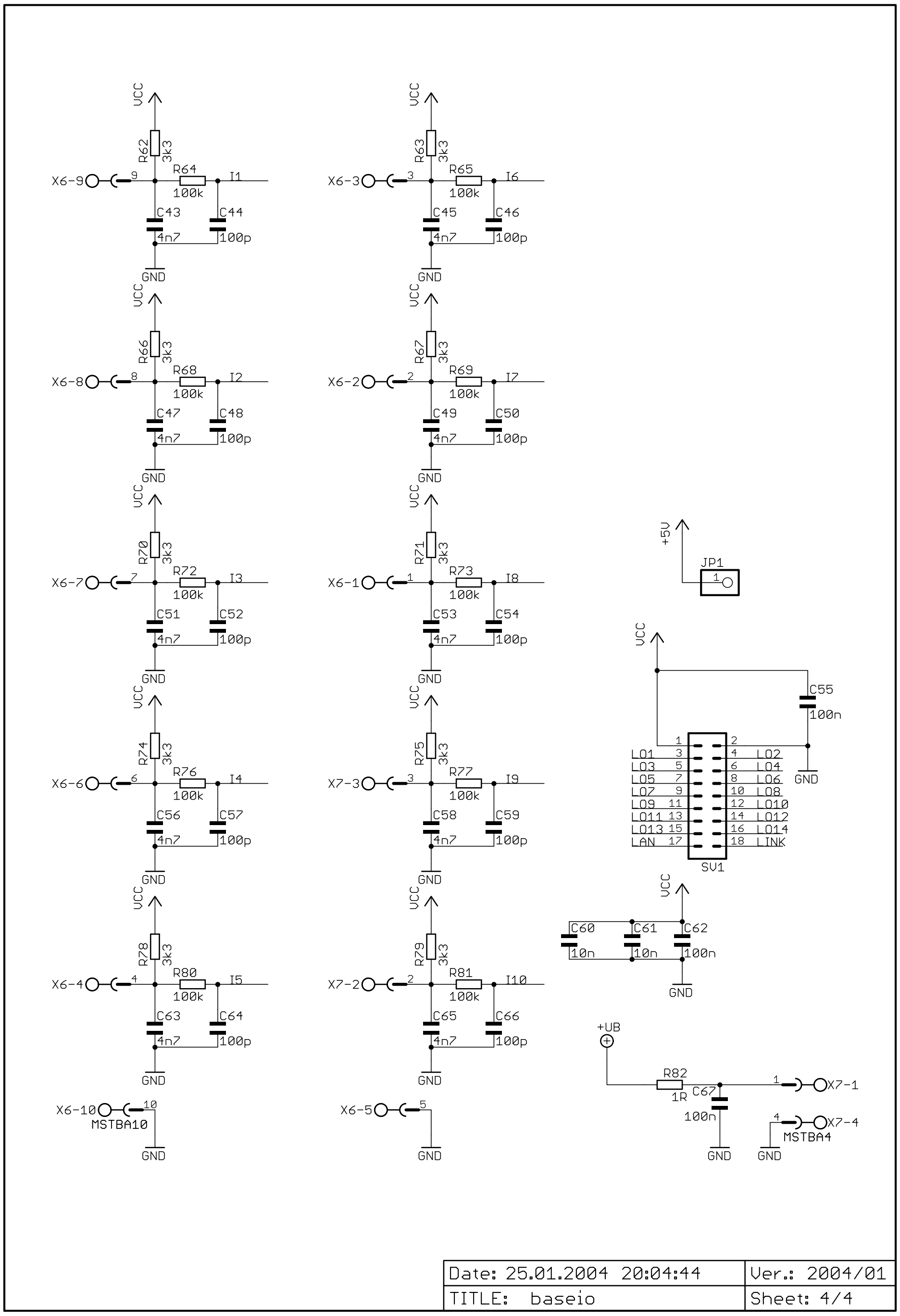


Figure E.9: Schematic of the Baseio extension board, page 4

E.3 Dspio Board

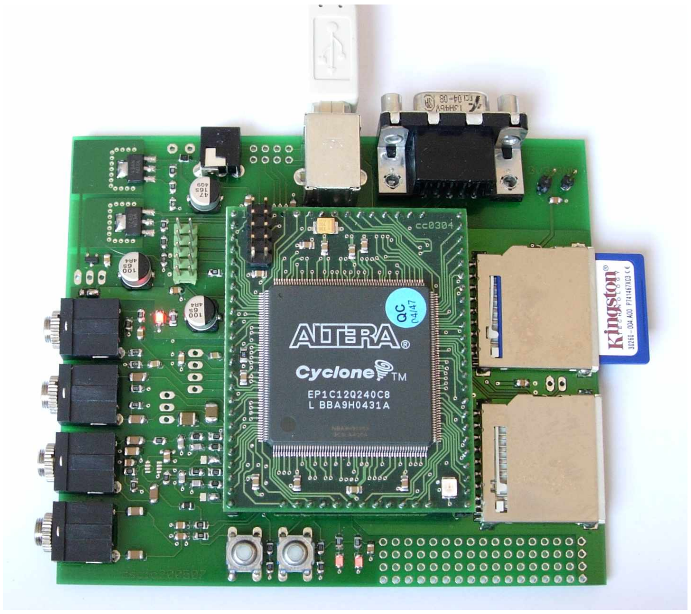


Figure E.10: The dspio extension board

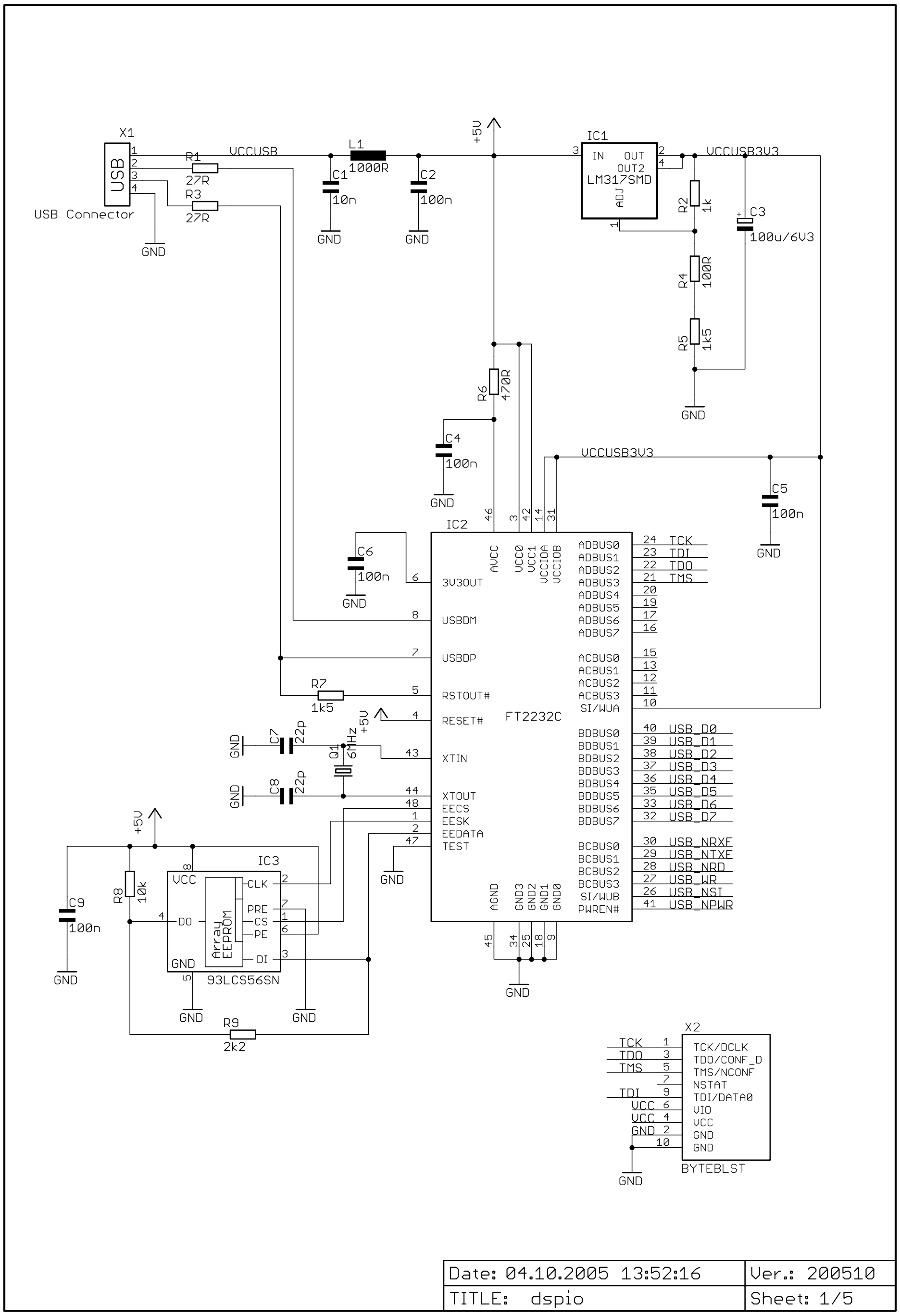

X1
USB
USB Connector
R1 27R
R3 27R
VCCUSB
L1 1000R
C1 10n
C2 100n
+5V
IC1
IN OUT OUT2 ADJ
LM317SMD
VCCUSB3V3
R2 1k
C3 100u/6V3
R4 100R
R5 1k5
R6 470R
C4 100n
VCCUSB3V3
C5 100n
IC2
C6 100n
AVCC VCC0 VCC1 VCCIOA VCCIOB
3V3OUT
USBDM
USBDP
RSTOUT#
RESET#
XTIN
XTOUT
EECS
EESK
EEDATA
TEST
ADBUS0 ADBUS1 ADBUS2 ADBUS3 ADBUS4 ADBUS5 ADBUS6 ADBUS7
ACBUS0 ACBUS1 ACBUS2 ACBUS3 SI/WUA
BDBUS0 BDBUS1 BDBUS2 BDBUS3 BDBUS4 BDBUS5 BDBUS6 BDBUS7
BCBUS0 BCBUS1 BCBUS2 BCBUS3 SI/WUB PWREN#
AGND GND3 GND2 GND1 GND0
FT2232C
TCK TDI TDO TMS
USB_D0 USB_D1 USB_D2 USB_D3 USB_D4 USB_D5 USB_D6 USB_D7
USB_NRXF USB_NTXE USB_NRD USB_WR USB_NSI USB_NPWR
R7 1k5
C7 22p
C8 22p
Q1 6MHz
IC3
VCC DO GND CLK PRE CS PE DI
Array EEPROM
93LCS56SN
R8 10k
C9 100n
R9 2k2
GND
X2
TCK/DCLK TDO/CONF_D TMS/NCONF NSTAT TDI/DATA0 VIO VCC GND GND
TCK TDO TMS TDI VCC VCC GND
BYTEBLST
Date: 04.10.2005 13:52:16
Ver.: 200510
TITLE: dspio
Sheet: 1/5

Figure E.11: Schematic of the dspio extension board, page 1

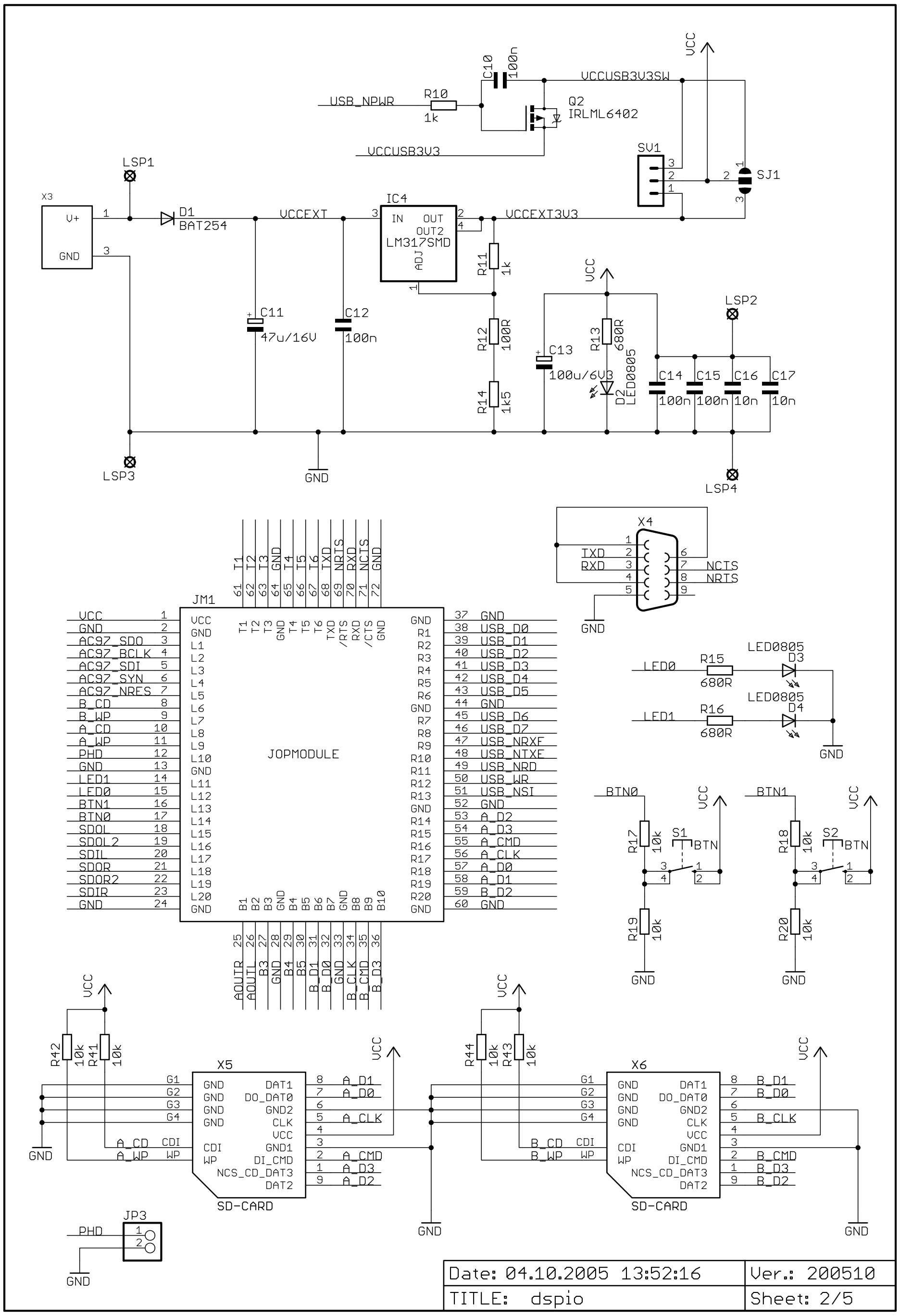


Figure E.12: Schematic of the dspio extension board, page 2

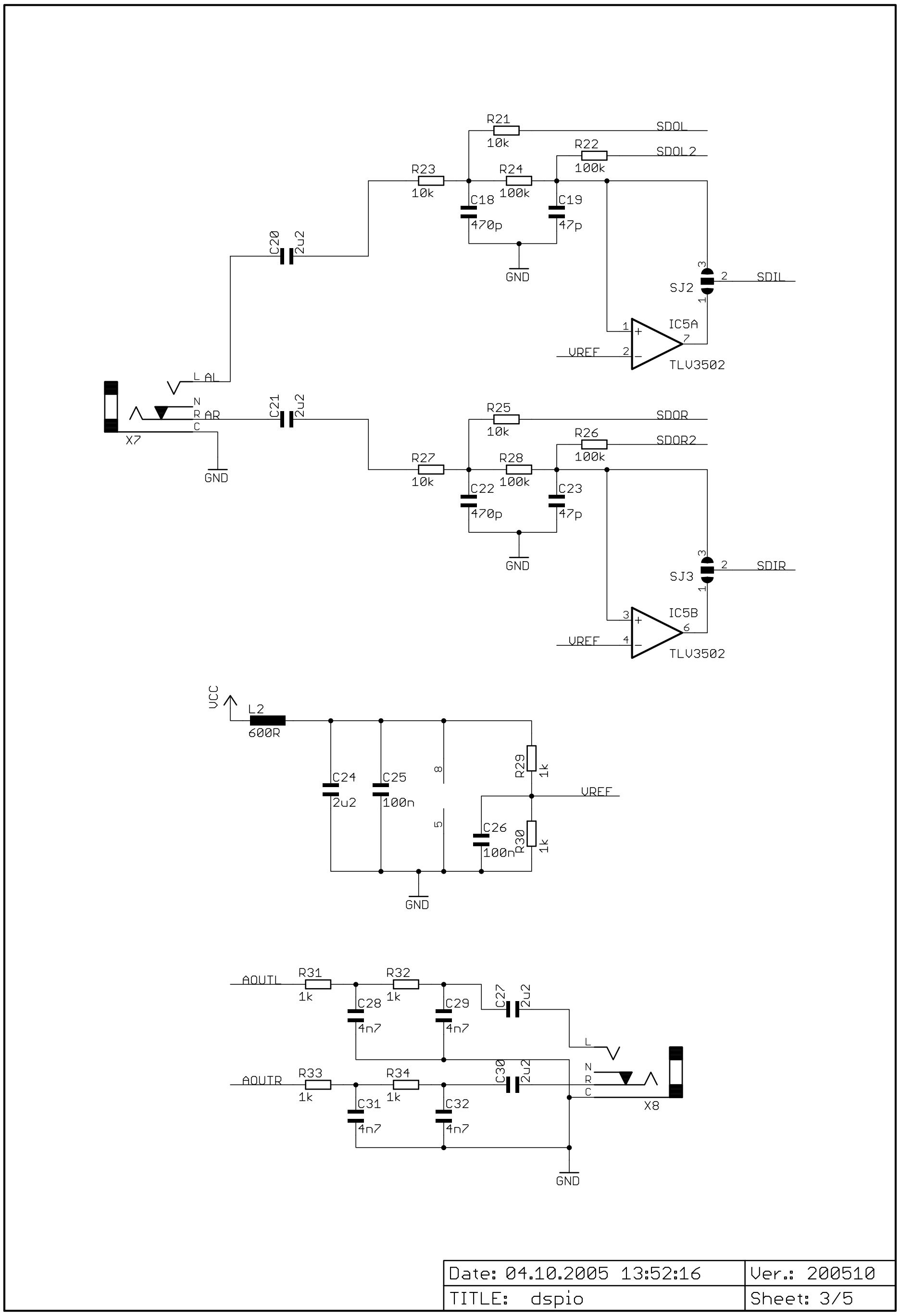


Figure E.13: Schematic of the dspio extension board, page 3

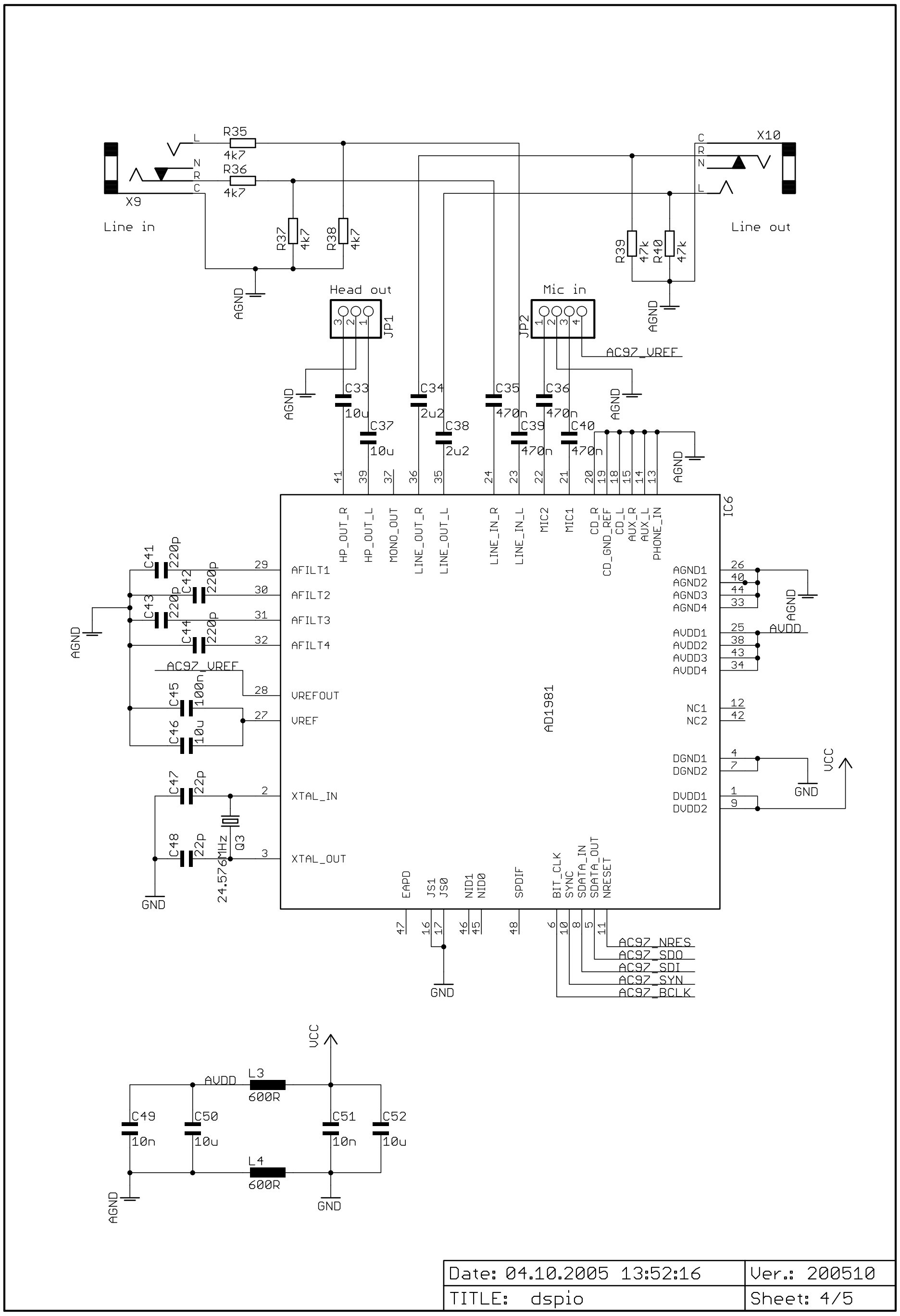


Figure E.14: Schematic of the dspio extension board, page 4

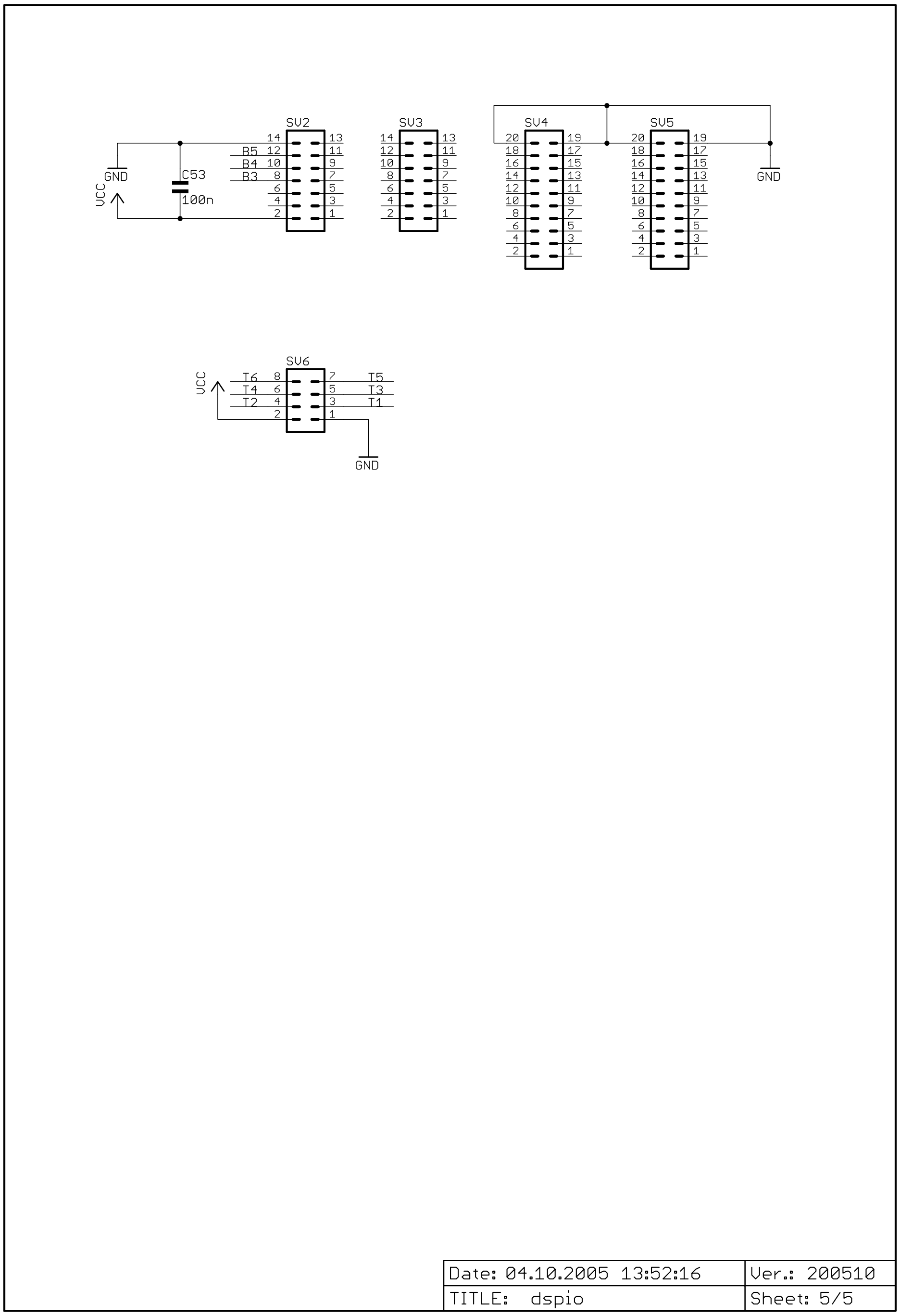


Figure E.15: Schematic of the dspio extension board, page 5

E.4 Simpexp Board

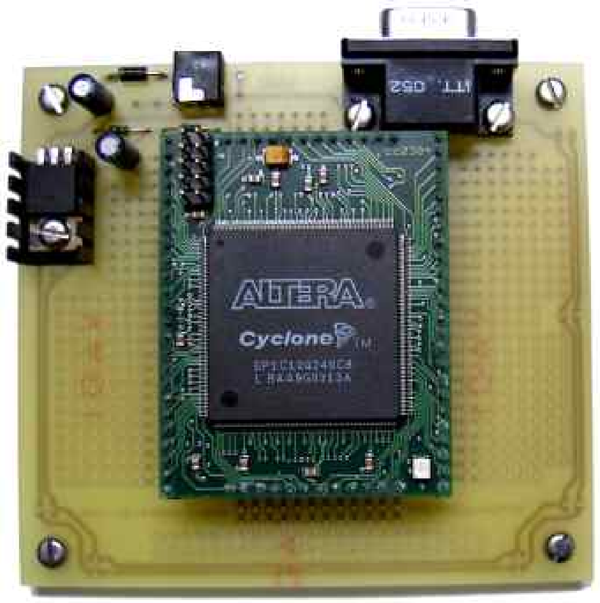


Figure E.16: The simpexp extension board

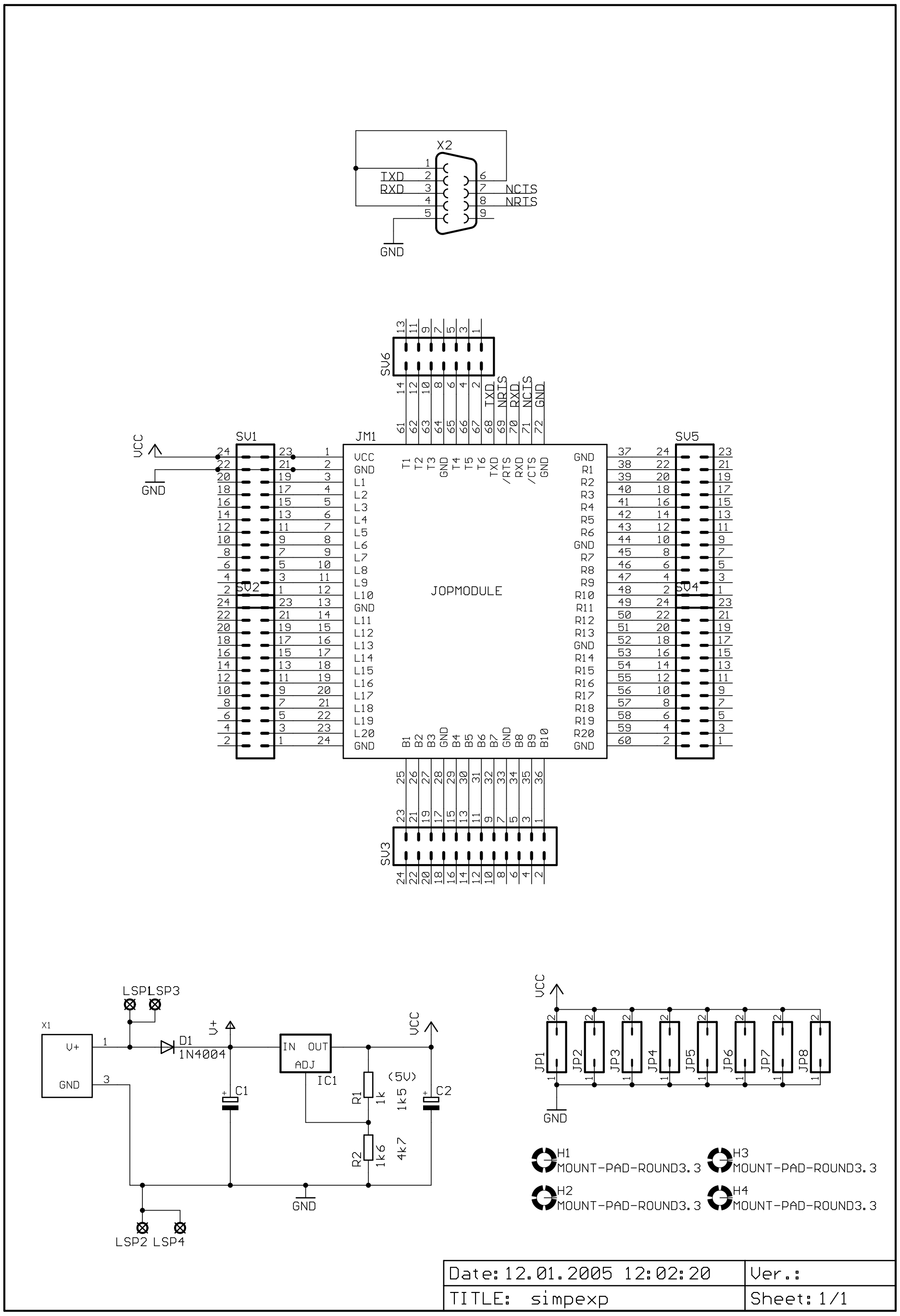


Figure E.17: Schematic of the simpexp extension board

Bibliography

[1] Hija safety critical java proposal. available at http://www.aicas.com/papers/scj.pdf, May 2006.

[2] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[3] Altera. Cyclone FPGA Family Data Sheet, ver. 1.2, April 2003.

[4] Altera. Avalon interface specification, April 2005.

[5] Altera. Quartus ii version 7.1 handbook, May 2007.

[6] Aonix. Perc pico 1.1 user manual. http://research.aonix.com/jsc/pico-manual.4-19-08.pdf, April 2008.

[7] ARM. AMBA specification (rev 2.0), May 1999.

[8] ARM. AMBA AXI protocol v1.0 specification, March 2004.

[9] ARM. Jazelle technology: ARM acceleration technology for the Java platform. white paper, 2004.

[10] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[11] Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, 1994.

[12] Cyrille Artho and Armin Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. *Electronic Notes in Theoretical Computer Science*, 141(1):109–128, December 2005.

[13] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[14] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time java virtual machine. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 249–258, New York, NY, USA, 2007. ACM.

[15] Azul. Azul compute appliances. Whitepaper, 2009.

[16] David F. Bacon, Perry Cheng, David Grove, Michael Hind, V. T. Rajan, Eran Yahav, Matthias Hauswirth, Christoph M. Kirsch, Daniel Spoonhower, and Martin T. Vechev. High-level real-time programming in java. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 68–78, New York, NY, USA, 2005. ACM Press.

[17] David F. Bacon, Perry Cheng, and V. T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In Robert Meersman and Zahir Tari, editors, *OTM Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 466–478. Springer, 2003.

[18] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[19] Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[20] Henry G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70, 1992.

[21] Iain Bate, Guillem Bernat, Greg Murphy, and Peter Puschner. Low-level analysis of a portable Java byte code WCET analysis framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[22] Iain Bate, Guillem Bernat, and Peter Puschner. Java virtual machine support for portable worst-case execution time analysis. In *ISORC. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, USA, Jan 2002.

[23] Guillem Bernat, Alan Burns, and Andy Wellings. Portable worst-case execution time analysis using Java byte code. In *Proc. 12th EUROMICRO Conference on Real-time Systems*, Jun 2000.

[24] Thomas Bogholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 106–114, New York, NY, USA, 2008. ACM.

[25] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[26] Gregory Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, and Frédéric Parain. Mackinac: Making HotSpot^{TM} real-time. In *ISORC*, pages 45–54. IEEE Computer Society, 2005.

[27] Florian Brandner, Tommy Thorn, and Martin Schoeberl. Embedded JIT compilation with CACAO on YARI. Technical Report RR 35/2008, Institute of Computer Engineering, Vienna University of Technology, Austria, June 2008.

[28] Florian Brandner, Tommy Thorn, and Martin Schoeberl. Embedded JIT compilation with CACAO on YARI. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[29] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Prgrm. Chrm. G. L. Steele, Jr., editor, *LISP and Functional Programming. Conference Record of the 1984 ACM Symposium, Austin, Texas, August 6-8, 1984*, number ISBN 0-89791-142-3, New York, 1984. ACM.

[30] Ben Brosgol and Brian Dobbing. Real-time convergence of Ada and Java. In *Proceedings of the 2001 annual ACM SIGAda international conference on Ada*, pages 11–26. ACM Press, 2001.

[31] José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro J. Gil, and Andy J. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 204–213, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.

[32] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[33] Cyrille Comar, Gary Dismukes, and Franco Gasperoni. Targeting gnat to the java virtual machine. In *Proceedings of the conference on TRI-Ada '97*, pages 149–161. ACM Press, 1997.

[34] Markus Dahm. Byte code engineering with the BCEL API. Technical report, Freie Universitat Berlin, April 2001.

[35] DCT. Lightfoot 32-bit Java processor core. data sheet, September 2001.

[36] Derivation. Lavacore configurable Java processor core. data sheet, April 2001.

[37] S. Dey, P. Sanchez, D. Panigrahi, L. Chen, C. Taylor, and K. Sekar. Using a soft core in a SOC design: Experiences with picoJava. *IEEE Design and Test of Computers*, 17(3):60–71, July 2000.

[38] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[39] Brian Dobbing and Alan Burns. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, pages 1–6. ACM Press, 1998.

[40] M. Eden and M. Kagan. The Pentium processor with MMX technology. In *Proceedings of Compcon '97*, pages 260–262. IEEE Computer Society, 1997.

[41] EJC. The ejc (embedded java controller) platform. Available at http://www.embedded-web.com/index.html.

[42] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[43] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):437–455, August 2003.

[44] Jiri Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[45] Jiri Gaisler, Edvin Catovic, Marko Isomäki, Kristoffer Carlsson, and Sandi Habinc. GRLIB IP core user's manual, version 1.0.14. Available at http://www.gaisler.com/, February 2007.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[47] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[48] Flavius Gruian, Per Andersson, Krzysztof Kuchcinski, and Martin Schoeberl. Automatic generation of application-specific systems based on a micro-programmed java core. In *Proceedings of the 20th ACM Symposium on Applied Computing, Embedded Systems track*, Santa Fee, New Mexico, March 2005.

[49] Flavius Gruian and Zoran Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005*, pages 281–294. Springer-Verlag GmbH, October 2005.

[50] Flavius Gruian and Mark Westmijze. Bluejep: a flexible and high-performance java embedded processor. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 222–229, New York, NY, USA, 2007. ACM.

[51] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Uppsala University, 2000.

[52] Jan Gustafsson. Worst case execution time analysis of object-oriented programs. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 71–76, 2002.

[53] Tom R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.

[54] David Hardin, Mike Frerking, Philip Wiley, and Gregory Bollella. Getting down and dirty: Device-level programming using the real-time specification for Java. In *Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, pages 457–464, 2002.

[55] David S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.

[56] Trevor Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.

[57] Trevor Harmon and Raymond Klefstad. Interactive back-annotation of worst-case execution time analysis for Java microprocessors. In *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, August 2007.

[58] Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, St. Louis, MO, United States, April 2008.

[59] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999.

[60] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, 1995.

[61] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[62] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 2002.

[63] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[64] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[65] Teresa Higuera, Valerie Issarny, Michel Banatre, Gilbert Cabillic, Jean-Philippe Lesot, and Frederic Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[66] Benedikt Huber. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.

[67] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based WCET analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, July 2009. OCG.

[68] IBM. On-chip peripheral bus architecture specifications v2.1, April 2001.

[69] Imsys. ISAJ reference 2.0, January 2001.

[70] Imsys. Im1101c (the Cjip) technical reference manual / v0.25, 2004.

[71] S.A. Ito, L. Carro, and R.P. Jacobi. Making Java work for microcontroller applications. *IEEE Design & Test of Computers*, 18(5):100–110, 2001.

[72] Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[73] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[74] Stephan Korsholm, Martin Schoeberl, and Anders P. Ravn. Interrupt handlers in Java. In *Proceedings of the 11th IEEE International Symposium on*

Object/component/service-oriented Real-time distributed Computing (ISORC 2008), Orlando, Florida, USA, May 2008. IEEE Computer Society.

[75] Andreas Krall and Reinhard Grafl. CACAO – A 64 bit JavaVM just-in-time compiler. In Geoffrey C. Fox and Wei Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997. ACM.

[76] Jochen Kreuzinger, Uwe Brinkschulte, Matthias Pfeffer, Sascha Uhrig, and Theo Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[77] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140. ACM Press, 2002.

[78] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[79] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[80] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 298, Washington, DC, USA, 1995. IEEE Computer Society.

[81] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 380–387. IEEE Computer Society, 1995.

[82] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[83] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[84] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.

[85] Mälardalen Real-Time Research Center. WCET benchmarks. Available at `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, accessed 2009.

[86] Stefan Metzlaff, Sascha Uhrig, Jörg Mische, and Theo Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proceedings of the 9th workshop on Memory performance (MEDEA 2008)*, pages 38–45, New York, NY, USA, 2008. ACM.

[87] Albert F. Niessner and Edward G. Benowitz. RTSJ memory areas and their affects on the performance of a flight-like attitude control system. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), LNCS*, 2003.

[88] K. Nilsen, L. Carnahan, and M. Ruark. Requirements for real-time extensions for the Java platform. Available at http://www.nist.gov/rt-java/, September 1999.

[89] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 113–133. Springer-Verlag, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[90] J. Michael O'Connor and Marc Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[91] OCP-IP Association. Open core protocol specification 2.1. http://www.ocpip.org/, 2005.

[92] Rasmus Pedersen and Martin Schoeberl. Exact roots for a real-time garbage collector. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 77–84, New York, NY, USA, 2006. ACM Press.

[93] Rasmus Ulslev Pedersen. Hard real-time analysis of two java-based kernels for stream mining. In *Proceedings of the 1st Workshop on Knowledge Discovery from Data Streams (IWKDDS, ECML PKDD 2006)*, Berlin, Germany, September 2006.

[94] Wade D. Peterson. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores, revision: B.3. Available at http://www.opencores.org, September 2002.

[95] Matthias Pfeffer, Theo Ungerer, Stephan Fuhrmann, Jochen Kreuzinger, and Uwe Brinkschulte. Real-time garbage collection for a multithreaded java microcontroller. *Real-Time Systems*, 26(1):89–106, 2004.

[96] Christof Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, 2008.

[97] Christof Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[98] Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, pages 317 – 322, Amsterdam, Netherlands, August 2007.

[99] Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 144–151, Vienna, Austria, September 2007. ACM Press.

[100] Christof Pitter and Martin Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, Jun. 2008.

[101] Filip Pizlo, J. M. Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on, Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 101–110, 2004.

[102] Thomas B. Preusser, Martin Zabel, and Rainer G. Spallek. Bump-pointer method caching for embedded java processors. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, pages 206–210, New York, NY, USA, 2007. ACM.

[103] Wolfgang Puffitsch. picoJava-II in an FPGA. Master's thesis, Vienna University of Technology, 2007.

[104] Wolfgang Puffitsch. Supporting WCET analysis with data-flow analysis of Java bytecode. Research Report 16/2009, Institute of Computer Engineering, Vienna University of Technology, Austria, February 2009.

[105] Wolfgang Puffitsch and Martin Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 213–221, Vienna, Austria, September 2007. ACM Press.

[106] Peter Puschner and Guillem Bernat. Wcet analysis of reusable portable code. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[107] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.

[108] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.

[109] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[110] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

[111] Peter Puschner and Andy Wellings. A profile for high integrity real-time Java programs. In *4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

[112] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.

[113] Sven Gestegard Robertz and Roger Henriksson. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM Press.

[114] Sven Gestegøard Robertz. *Automatic memory management for flexible real-time systems*. PhD thesis, Department of Computer Science Lund University, 2006.

[115] RTCA/DO-178B. Software considerations in airborne systems and equipment certification. December 1992.

[116] William J. Schmidt and Kelvin D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, NY, USA, 1994. ACM Press.

[117] Martin Schoeberl. Using a Java optimized processor in a real world application. In *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, pages 165–176, Austria, Vienna, June 2003.

[118] Martin Schoeberl. Real-time scheduling on a Java processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[119] Martin Schoeberl. Restrictions of Java for embedded real-time systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004. IEEE.

[120] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[121] Martin Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[122] Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[123] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[124] Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006. IEEE.

[125] Martin Schoeberl. Architecture for object oriented programming languages. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 57–62, Vienna, Austria, September 2007. ACM Press.

[126] Martin Schoeberl. Mission modes for safety critical Java. In *Software Technologies for Embedded and Ubiquitous Systems, 5th IFIP WG 10.2 International Workshop (SEUS 2007)*, volume 4761 of *Lecture Notes in Computer Science*, pages 105–113. Springer, May 2007.

[127] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[128] Martin Schoeberl. A time-triggered network-on-chip. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, pages 377 – 382, Amsterdam, Netherlands, August 2007.

[129] Martin Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, July 2008.

[130] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[131] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Number ISBN 978-3-8364-8086-4. VDM Verlag Dr. Müller, July 2008.

[132] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[133] Martin Schoeberl, Stephan Korsholm, Christian Thalinger, and Anders P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[134] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[135] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*. ACM Press, September 2008.

[136] Martin Schoeberl and Peter Puschner. Is chip-multiprocessing the end of real-time scheduling? In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, July 2009. OCG.

[137] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P. Ravn. A profile for safety critical Java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 94–101, Santorini Island, Greece, May 2007. IEEE Computer Society.

[138] Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[139] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, 1989.

[140] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. Number ISBN: 3-8311-3893-1. aicas Books, 2002.

[141] International J Consortium Specification. Real-time core extensions, draft 1.0.14. Available at http://www.j-consortium.org/, September 2000.

[142] Guy L. Steele. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[143] Sun. A brief history of the green project. Available at: http://today.java.net/jag/old/green/.

[144] Sun. Java technology: The early years. Available at: http://java.sun.com/features/1998/05/birthday.html.

[145] Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.

[146] Sun. *picoJava-II Programmer's Reference Manual*. Sun Microsystems, March 1999.

[147] Systronix. Jstamp real-time native Java module. data sheet.

[148] Y.Y. Tan, C.H. Yau, K.M. Lo, W.S. Yu, P.L. Mok, and A.S. Fong. Design and implementation of a java processor. *Computers and Digital Techniques, IEE Proceedings-*, 153:20–30, 2006.

[149] Sascha Uhrig and Jörg Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.

[150] Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[151] Jack Whitham, Neil Audsley, and Martin Schoeberl. Using hardware methods to improve time-predictable performance in real-time java systems. In *Proceedings of the 7th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2009)*, Madrid, Spain, September 2009. ACM Press.

[152] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[153] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[154] Xilinx. Spartan-3 FPGA family: Complete data sheet, ver. 1.2, January 2005.

[155] Tan Yiyu, Richard Li Lo Wan Yiu, Yau Chi Hang, and Anthony S. Fong. A java processor with hardware-support object-oriented instructions. *Microprocessors and Microsystems*, 30(8):469–479, 2006.

[156] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

[157] Martin Zabel, Thomas B. Preusser, Peter Reichel, and Rainer G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Aug. 2007.

[158] R. Zulauf. Entwurf eines Java-Mikrocontrollers und prototypische Implementierung auf einem FPGA. Master's thesis, University of Karlsruhe, 2000.

Index