

An Introduction into the Design Flow for JOP

Martin Schoeberl
martin@jopdesign.com

February 23, 2009

This section 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.

1 Introduction

JOP [2], 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 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 Ant¹ based build process is also available.

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

¹<http://ant.apache.org/>

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:

```
javac
gcc
make
cvs
quartus_map
```

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

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.

1.2.1 Download the Source

Create a working directory and download JOP from the www.opencores.org CVS server:

```
cvs -d :pserver:anonymous@cvs.opencores.org:/cvsroot/anonymous \
-z9 co -P jop
```

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

```
make directories
```

1.2.2 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
```

1.2.3 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 processor. If you want to build it within Quartus follow the following instructions:

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

1.2.4 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.

1.2.5 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.

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 with the project file `jop.npl`. It will be converted to a new (binary) `jop.isc` 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 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.

2.1 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 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`.

²All 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.

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`.

2.3 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.

2.4 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. First 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
```

3 The Design Flow

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

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

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

3.1.1 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.

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).

3.1.2 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 library⁴ (BCEL).

3.1.3 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).

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

⁴<http://jakarta.apache.org/bcel/>

- Xilinx Spartan-3
- Altera Cyclone-II (Altera DE2 board)
- 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.

3.2.1 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 ???. 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

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

⁵http://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 1: Quartus project directories and VHDL files for the different I/O boards

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

3.2.2 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.

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.

All projects use the Eclipse path variable⁶ `JOP_HOME` that has to point to the root directory (`.../jop`) of the JOP sources. Under **Window – Preferences...** select **General – Workspace – 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 root directory `.../jop/eclipse`, select the projects you want to import, select **Copy projects into workspace**, and press **Finish**. Table 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

⁶Eclipse (path) variables are workspace specific.

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

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: Eclipse projects

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)

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"`.

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 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

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 3: Simulation specific VHDL files

- 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.

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.
- . **xm1** 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.
- . **tc1** 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 runs 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.

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

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`.

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`.

8.1.1 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.

8.1.2 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 address 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.

9 Extending JOP

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

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 bytecode. 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 the 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.

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` [3] interface. For a device that implements the `Wishbone` [1] 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 `0xfffff80` 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 = 0xfffff80;
...
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 package `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) {
```

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

```
        i
    }
    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.

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 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`.

9.4 Dependencies and Configurations

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

9.4.1 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`

9.4.2 Changing the Class Format

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

References

- [1] 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.
- [2] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.
- [3] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workshop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.