1 Bytecode Execution Time

Table 1.1 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
20	ldc2_w[20]	mc	17+2*r
21	iload	mc	2
22	lload	mc	11
23	fload	mc	2
24	dload	mc	11

Table 1.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
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]	mc	7+3*r
47	laload	mc	43+4*r
48	faload[46]	mc	7+3*r
49	daload	-	
50	aaload[46]	mc	7+3*r
51	baload[46]	mc	7+3*r
52	caload[46]	mc	7+3*r
53	saload[46]	mc	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

Table 1.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
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]	mc	10+2*r+w
80	lastore[1]	mc	48+2*r+2*w
81	fastore[79]	mc	10+2*r+w
82	dastore	-	
83	aastore	Java	
84	bastore[79]	mc	10+2*r+w
85	castore[79]	mc	10+2*r+w
86	sastore[79]	mc	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	Java	
98	fadd	Java	
99	dadd	-	
100	isub	hw	1
101	lsub	Java	
102	fsub	Java	

Table 1.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
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	Java	
118	fneg	Java	
119	dneg	-	
120	ishl	hw	1
121	lshl	Java	
122	ishr	hw	1
123	lshr	Java	
124	iushr	hw	1
125	lushr	Java	
126	iand	hw	1
127	land	Java	
128	ior	hw	1
129	lor	Java	
130	ixor	hw	1
131	lxor	Java	
132	iinc	mc	8
133	i2l	Java	
134	i2f	Java	
135	i2d	-	
136	l2i	mc	3
137	l2f	-	
138	l2d	-	
139	f2i	Java	
140	f2l	-	
141	f2d	-	

Table 1.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
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	-	
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	12+2*r
179	putstatic	mc	13+r+w
180	getfield	mc	17+2*r

Table 1.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
181	putfield	mc	20+r+w
182	invokevirtual[182]	mc	100+4r+l
183	invokespecial[183]	mc	74+3*r+l
184	invokestatic[183]	mc	74+3*r+l
185	invokeinterface[185]	mc	114+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	11
195	monitorexit	mc	10/14
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
218	jopsys_setvp	mc	2
219	jopsys_int2ext[219]	mc	14+r+n*(23+w)

Table 1.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
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	
225	putstatic_ref	mc	
226	getfield_ref	mc	
227	putfield_ref	mc	
228	getstatic_long	mc	
229	putstatic_long	mc	
230	getfield_long	mc	
231	putfield_long	mc	
232	reserved	-	
233	reserved	-	
234	reserved	-	
235	reserved	-	
236	reserved	-	
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	-	
251	reserved	-	
252	reserved	-	
253	reserved	-	
254	sys_noimp	Java	
255	sys_init	*not used*	

Table 1.1: 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 100MHz JOP with 15ns SRAMs this access time is two cycles (ram_cnt=2, 20ns). 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
```

[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. IO devices are now memory mapped. In the case for simple IO 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.

[240]*Is the interrupt and the exception still a bytecode or is it now inserted just as microcode address?*

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 15ns SRAMs and 100MHz 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

For the method cache load the cache wait state c_{ws} is:

$$c_{ws} = \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

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

$$l = \begin{cases} 6 + (n+1)(2 + c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cach hit} \end{cases}$$

with n as the length 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

$$t_{lastore} = 48 + 2r_{ws} + w_{ws} + \begin{cases} w_{ws} - 3 & : \quad w_{ws} > 3 \\ 0 & : \quad w_{ws} \leq 3 \end{cases}$$

get/putfield/ref/long

TODO: add different values for 32-bit, 64-bit and reference type.

2 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 2.1 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
ioar	Address register of the IO subsystem
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 2.1: 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.

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.

stmwa

Operation Store memory write address

Opcode `00001001`

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.

stmwd

Operation Store memory write data

Opcode `00001010`

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 `00001011`

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 `00001100`

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 hast 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.

stmul

Operation Multiply int

Opcode 00001101

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.

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.