CS201 W'00 Lecture Notes
Lecture 7

Java-based Systems

Real Things:

- **Java**, the language.

– A programming language more or less like many others.

- **Java Virtual Machine (JVM)**.

– Stack-based virtual machine based on **byte code** instructions.

– A standard target for Java compiler front-ends, and for sending programs over networks.

– Typically interpreted by a software interpreter.

– Can be further compiled into native code for particular architecture.

– Can be used as ISA for hardware processor.

- **picoJava II**

– A machine architecture for direct execution of JVM programs in hardware.

– Licensed to various chip-makers; actual chips scheduled soon for embedded systems.

Java-based Systems

Tanenbaum's Idealized Things:

- **IJVM** = Integer JVM

– Subset of JVM instructions, treated as a hardware ISA

- **Mic-1,2,3,4**

– Microarchitectures (processor implementations) for IVJM.

– These microarchitectures have (**microcode**) instruction sets and programs of their own

– Microprograms can be specified using **mal**, a microcode assembly language.

IJVM

- **Stack-based** machine (in two senses)
- Instructions are **byte-codes**: one byte opcode followed by zero or more bytes (or 2-byte pairs) of parameters.
- Instructions are grouped into **methods** (i.e., procedures).

Memory model:

Each method can access four areas:

- **Constant Pool**

– read-only area containing constant integers and pointers to method code

– accessed by 16-bit unsigned offset (in 32-bit words) from start

- **Local Variable Frame**

– one per per method invocation

– arguments 1,2,... are accessed as local variables 1,2,...

– numbers of arguments and local variables are fixed parameters of method

– accessed by 8-bit or 16-bit unsigned **variable number** (counts 32-bit words) from start

- **Operand Stack**

– used to support stack-based arithmetic

Example: $2 + (3 - 5)$ becomes: `BIPUSH 2; BIPUSH 3; BIPUSH 5; ISUB; IADD`

– also used for arguments and return value for method calls

Example: `a = f(1,2,3)` becomes

`BIPUSH 1; BIPUSH 2; BIPUSH 3; INVOKEVIRTUAL f-offset; ISTORE a-varnum`

– accessed only implicitly via pushes and pops (of 32-bit words)

– maximum size is fixed parameter of method (Tanenbaum doesn't use this fact, though)

- **Method Area**

– read-only memory containing method byte-code sequences

– byte-aligned; jumps and branches use byte offsets

– first 2 bytes of method contain argument count; second 2 bytes contain local variable count; code begins at 5th byte

Instruction Set

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Figure 4-11. The IJVM instruction set. The operands *byte*, *const*, and *varnum* are 1 byte. The operands *disp*, *index*, and *offset* are 2 bytes.

byte and **const** are signed byte literals

offset is signed 2-byte offset within current method

varnum is normally unsigned byte offset into local variables, but after `WIDE` it is unsigned 2-byte offset

disp is unsigned 2-byte offset into constant pool; constant points to method in method area.

index is unsigned 2-byte offset into constant pool; constant is 32-bit literal.

Memory Model Implementation

Implementation maintains implicit pointers:

- CPP = base of constant pool
- PC = current instruction in method area
- LV = base of local variables
- SP = top of current operand stack

Since each method activation needs its own local variable area and operand stack, it makes sense to store (both) these areas on a **stack**.

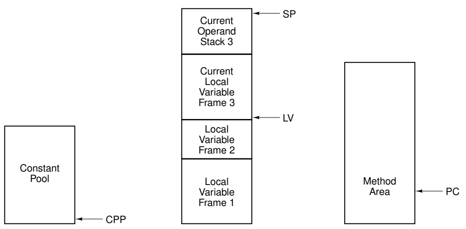

Figure 4-10. The various parts of the IJVM memory.

Note that this memory layout is **not** visible in IJVM ISA itself.

Method Call and Return

Why a single stack? Easier to manage, and get small efficiency gain because we don't have to copy arguments from caller's stack to callee's local variable area.

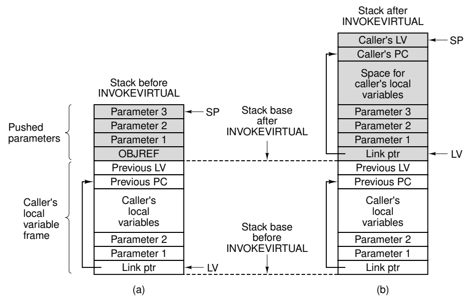

Figure 4-12. (a) Memory before executing INVOKEVIRTUAL. (b) After executing it.

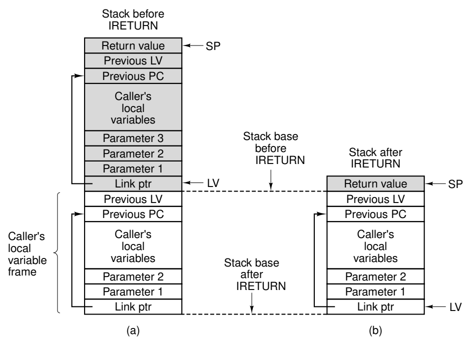

Figure 4-13. (a) Memory before executing IRETURN. (b) After executing it.

Mic-1 Data Path

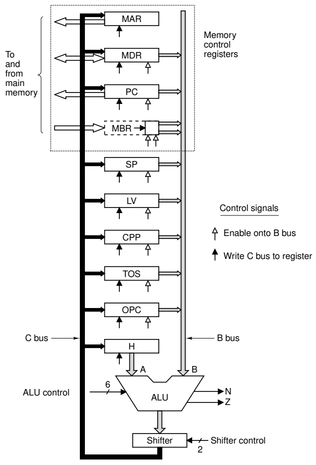

Figure 4-1. The data path of the example microarchitecture used in this chapter.

MBR = Byte data fetched from method area
MBRU = Sign extended version of MBR
TOS = Register copy of top of stack pointed to by SP
OPC = Temporary register
H = Holding register for "A" operand

ALU Operations

Inputs are H register (via A bus) and B bus.

Controlled by 6 control lines:

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B − A
1	1	0	1	1	0	B − 1
1	1	1	0	1	1	−A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	−1

Figure 4-2. Useful combinations of ALU signals and the function performed.

Also, shifter is controlled by two lines:

SLL8: shift left logical (0 fill) by 8 bits
SRA1: shift right arithmetic (sign extend) by 1 bit

Outputs are C bus and N,Z signals.

Memory Operations

Access to Constant Pool and Stack memory is via `MAR` and `MDR`.

To read: set address in `MAR` and assert `READ`; data will appear in `MDR` at **end** of **next** cycle.

To write: set address in `MAR` and data in `MDR` and assert `WRITE`; data will be written immediately.

Access to Method area is via `PC` and `MBR`.

To read: set address in `PC` and assert `FETCH`; data will appear in `MBR` at **end** of **next** cycle.

Timing

Clock cycle must be long enough to drive signals from instruction decoding and input registers all the way to output registers.

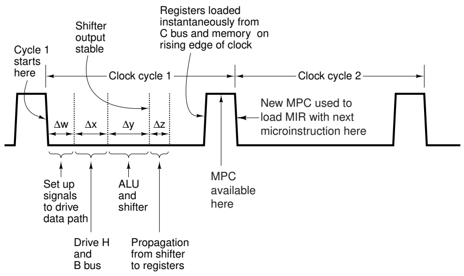

Figure 4-3. Timing diagram of one data path cycle.

Recall that memory reads initiated in cycle n produce a result in MDR or MBR at the **end** of cycle $n+1$.

Note that the contents of `MBR` **are** available to help choose the next state or micro-operation for cycle $n+2$.

(Similarly, the `N` and `Z` signals generated by an operation in cycle n are available to help chose the next state at cycle $n+1$.)