CSE/EE 322: Computer Architecture II
Spring 2009

The JAM machine Series

Peter M. Kogge
The Original JAM-8

- Roots in JVM: Java Virtual Machine
  - Target of most Java compilers, & designed for portability
  - Usually executed by SPIM-like JVM interpreter program – but not always

- “Stack-based” ISA
  - Radically different from the MIPS you studied
    - No data register files, just four “pointer” registers
    - Variable length instructions
  - Forces you to really understand “multi-cycle” instruction execution
    - AND memory-intensive computing
  - Will help your understanding of the ubiquitous Java interpreter

- The JAM-8: used since 2000 in Comp Arch II as design target
  - Selected subset of JVM instructions (multiples of 8 bits)
  - Some instructions modified for ease of design
  - All data reduced to 8 bits, with 8 bit addresses (256 byte memory)
  - Rich suite of potential speedup techniques

- Reasons for Change to JAM3D
  - Run programs that consume more than 256 bytes of instructions and data
  - Simplify the design a bit
The JAM3D

- Same subset of instructions but re-encoded in 12 bit words
  - I did add a few extra for interpreter only (multiply & divide) that will make for more interesting benchmarks and performance analysis

- All data and memory addresses = 12 bits (4K words of memory)
  - Allows your interpreter to run “looong” programs to gather “real” data

- Only 2 different instruction lengths instead of three

- Enough “space” to allow discussion of ISA extensions
  - Multi-threading
  - Graphics (12-bit word contains three 4-bit R, G, B intensities)
  - …
Why Not a JAM-“16” Instead

- **Plus**
  - Would have given bigger memory capacity
  - More closely tracked real JVM
  - Matches real memory widths

- **Minus**
  - Data and instructions now different basic units
  - Requires more complex memory interface
    - Alignment issues
    - Little vs Big Endian considerations
  - And complexity adds little to your ability to design
A Conceptual High Level Picture of JAM3D

Datapath

Programmer-visible Registers

- Vars: pointer to local vars
- Frame: State for current method
- Optop: pointer to top of stack
- PC: Program counter

ALU

12-bit address

Programmer-invisible Registers

- MAR: Memory Address register
- MDR: Memory Data Register
- TOS: Top of Stack register
- IR: Instruction Register

12-bit data

Memory


Possible implementation: iadd: MDR = Mem[optop];
TOS = TOS + MDR;
optop = optop + 2;
PC = PC + 1; IR = Mem[PC];
### A Subset of the JVM from P&H 2.14 (on CD)

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
<th>Java bytecode</th>
<th>Size (bits)</th>
<th>MIPS Instr.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>ladd</td>
<td>8</td>
<td>add</td>
<td>NOS=TOS+NOS: pop</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>lsub</td>
<td>8</td>
<td>sub</td>
<td>NOS=TOS+NOS: pop</td>
</tr>
<tr>
<td></td>
<td>inc</td>
<td>liinc</td>
<td>8</td>
<td>addl</td>
<td>Frame[18] = Frame[18] + 18</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load local int/address</td>
<td>iload I8/aload I8</td>
<td>16</td>
<td>1w</td>
<td>TOS=Frame[18]</td>
</tr>
<tr>
<td></td>
<td>load local int/address</td>
<td>iload I8/aload_(0,1,2,3)</td>
<td>8</td>
<td>1w</td>
<td>TOS=Frame[0,1,2,3]</td>
</tr>
<tr>
<td></td>
<td>store local int/address</td>
<td>istore I8/astore I8</td>
<td>16</td>
<td>sw</td>
<td>Frame[18]:TOS: pop</td>
</tr>
<tr>
<td></td>
<td>load int/address from array</td>
<td>iload/aload</td>
<td>8</td>
<td>1w</td>
<td>NOS= *NOS[TOS]: pop</td>
</tr>
<tr>
<td></td>
<td>store int/address into array</td>
<td>iastore</td>
<td>8</td>
<td>sw</td>
<td>*NOS[NOS][TOS]: pop2</td>
</tr>
<tr>
<td></td>
<td>load half from array</td>
<td>saload</td>
<td>8</td>
<td>1h</td>
<td>NOS= *NOS[TOS]: pop</td>
</tr>
<tr>
<td></td>
<td>store half into array</td>
<td>sastore</td>
<td>8</td>
<td>sh</td>
<td>*NOS[NOS][TOS]: pop2</td>
</tr>
<tr>
<td></td>
<td>load byte from array</td>
<td>baload</td>
<td>8</td>
<td>1b</td>
<td>NOS= *NOS[TOS]: pop</td>
</tr>
<tr>
<td></td>
<td>store byte into array</td>
<td>bastore</td>
<td>8</td>
<td>sb</td>
<td>*NOS[NOS][TOS]: pop2</td>
</tr>
<tr>
<td></td>
<td>load immediate</td>
<td>bipush I8, sipush I16</td>
<td>16, 24</td>
<td>addl</td>
<td>jumpl; TOS=I8 or I16</td>
</tr>
<tr>
<td></td>
<td>load immediate</td>
<td>lconst(-1,0,1,2,3,4,5)</td>
<td>8</td>
<td>addl</td>
<td>jumpl; TOS=-4,0,1,2,3,4,5</td>
</tr>
<tr>
<td>Logical</td>
<td>and</td>
<td>land</td>
<td>8</td>
<td>and</td>
<td>NOS=TOS&amp;&amp;NOS: pop</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>lor</td>
<td>8</td>
<td>or</td>
<td>NOS=TOS</td>
</tr>
<tr>
<td></td>
<td>shift left</td>
<td>ishl</td>
<td>8</td>
<td>sll</td>
<td>NOS&lt;&lt;TOS: pop</td>
</tr>
<tr>
<td></td>
<td>shift right</td>
<td>lshr</td>
<td>8</td>
<td>srl</td>
<td>NOS&gt;&gt;TOS: pop</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>if_icmpeq I16</td>
<td>24</td>
<td>beq</td>
<td>If TOS==NOS, go to I16; pop2</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>if_icmpne I16</td>
<td>24</td>
<td>bne</td>
<td>If TOS!=NOS, go to I16; pop2</td>
</tr>
<tr>
<td></td>
<td>compare</td>
<td>if_icmp{l,1,2,3,4,5}</td>
<td>24</td>
<td>slt</td>
<td>If TOS&lt;l,&lt;=,&gt;,&gt;=NOS, go to I16; pop2</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>goto I16</td>
<td>24</td>
<td>j</td>
<td>go to I16</td>
</tr>
<tr>
<td></td>
<td>return</td>
<td>ret, return</td>
<td>8</td>
<td>jr</td>
<td>TOS=PC+3</td>
</tr>
<tr>
<td></td>
<td>jump to subroutine</td>
<td>jsr I16</td>
<td>24</td>
<td>jsr</td>
<td>go to I16; push; TOS=PC+3</td>
</tr>
<tr>
<td>Stack management</td>
<td>remove from stack</td>
<td>pop, pop2</td>
<td>8</td>
<td>pop</td>
<td>pop, pop2</td>
</tr>
<tr>
<td></td>
<td>duplicate on stack</td>
<td>dup</td>
<td>8</td>
<td>push</td>
<td>TOS=NOS</td>
</tr>
<tr>
<td></td>
<td>swap top 2 positions on stack</td>
<td>swap</td>
<td>8</td>
<td>T=NOS; NOS=TOS; TOS=T</td>
<td></td>
</tr>
<tr>
<td>Safety check</td>
<td>check for null reference</td>
<td>ifnull I16, ifnonnull I16</td>
<td>24</td>
<td>if</td>
<td>If TOS==1, null, go to I16</td>
</tr>
<tr>
<td></td>
<td>get length of array</td>
<td>arraylength</td>
<td>8</td>
<td>push</td>
<td>TOS = length of array</td>
</tr>
<tr>
<td></td>
<td>check if object a type</td>
<td>instanceof I16</td>
<td>24</td>
<td>if</td>
<td>If TOS matches type of Const[16]; TOS = 0 otherwise</td>
</tr>
<tr>
<td>Invocation</td>
<td>invoke method</td>
<td>invokevirtual I16</td>
<td>24</td>
<td>invoke</td>
<td>Method in Const[16], dispatching on type</td>
</tr>
<tr>
<td>Allocation</td>
<td>create new class instance</td>
<td>new I16</td>
<td>24</td>
<td>Allocate object type Const[16] on heap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>create new array</td>
<td>newarray I16</td>
<td>24</td>
<td>Allocate array type Const[16] on heap</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.14.1: Java bytecode architecture versus MIPS.** Although many bytecodes are simple, those in the last half-dozen rows above are complex and specific to Java. Bytcodes are 1 to 5 bytes in length, hence their name. The Java asembler uses the prefix 1 for 32-bit integer, 8 for reference/address, 5 for 16-bit integers (short), and 8 for 8-bit bytes. We use I8 for an 8-bit constant and I16 for a 16-bit constant. MIPS uses registers for operands, but the JVM uses a stack. The compiler knows the maximum size of the operand stack for each method and simply allocates space for it in the current frame. Here is the notation in the meaning column: TOS Top Of Stack, NOS next position below TOS, NOS: next position below NOS; pop: remove TOS; pop2: remove TOS and NOS; and push add a position to the stack. NOS and *NOS mean access the memory location pointed to by the address in the stack at those positions. Const[] refers to the run time constant pool of a class created by the JVM, and Frame[] refers to the variables of the local method frame. The only missing MIPS instructions from Figure 2.27 are for and, or, slt, and srl. The missing bytecodes are a few arithmetic and logical operators, some tricky stack management, compares to 0 and branch, support for branch tables, type conversions, more variations of the complex, Java-specific instructions plus operations on floating-point data, 64-bit integers (long), and 16-bit characters.
Metrics for JAM3D Designs

- **Performance**
  - Individual CPI: cycles per each instruction – esp. more “challenging” ones
  - Total cycle count for test program(s)
  - Total instructions executed for test program(s)
  - Average CPI for test programs
  - Perhaps: achievable clock frequency when synthesized for XiLinx

- **Cost**
  - Number of transistors
  - Computed from spreadsheet of basic logic blocks

- **Energy & Power (new)**
  - Energy per instruction EPI: cycles * “activity”
  - Energy per program EPP: sum of EPI
  - Power per program: EPP/(cycles*clock)

- **Cost-Performance** = #Transistors * average CPI

- **Power-Performance** = power per program * time per program
JAM-8 Designs: Prior Years Results

Cost vs CPI

Cost vs MaxFinder Cycles

Cost-Performance vs CPI

Legend:
- Logic Alone 2003
- Memory+Logic 2003
- Logic Alone 2004
- Memory+Logic 2004
- Logic Alone 2005
- Memory + Logic 2005
Change to Labs

JAM-8 Activities
1. JAM-8 interpreter
2. Cache using Shade
3. Pipelining using Shade: GONE
4. Branch Prediction using Shade
5. JAM-8 Behavioral interpreter
6. JAM-8 Structural interpreter

JAM-3D Activities
1. JAM3D interpreter
2. JAM3D Benchmark Development
3. JAM3D Mix Analysis
4. JAM3D Cache Simulation
5. JAM3D Behavioral implementation
6. JAM3D Branch Prediction
7. JAM3D Structural implementation
8. JAM3D on XiLinx: NEW
   1. Includes lab on memory synthesis

Blue Labs build on your JAM3D interpreter
Gold Labs: build towards Xilinx implementation
Lab Component Relationships

ISA Understanding

1. Benchmark Development
2. Interpreter
3. Mix Analysis
4. Cache Simulation
5. Structural Implementation
6. Branch Prediction
7. Behavioral Implementation
8. Synthesized Implementation

Microarchitectural Techniques

Increase your Analysis Skills
Develop your Design Skills

Implementation

Increase your Analysis Skills
Develop your Design Skills
Lab Organization

- Meet in Lab every 2 weeks

- Traditional:
  - Go over next lab component
  - Meet as group
  - Get help from TA
  - Use lab facilities (esp. at end)

- New:
  - Collate and discuss results from last component - in group
  - Discuss options for next component – in group

- Deliverables
  - Comparative data due to TA by Tuesday noon of lab week
  - Lab reports due in class on Thursday of lab week
Non-Design Labs: Benchmark Development

- One or more common benchmarks will be made available

- Each group will select a different short “benchmark”
  - With two data sets: a short “debug” and a long “performance”
  - Convert to JAM3D code
  - Extract statistics from their interpreter
  - Post results to class web site in advance of lab discussion

- Work done in 2 halves
  - 1\textsuperscript{st} Half:
    - benchmark selection and coding (in favorite prog. lang.)
    - Translation to JAM3D assembly, with gathering of static statistics
  - 2\textsuperscript{nd} Half: get running on interpreter (and gather dynamic statistics)

- In Lab discussion
  - Statistics will be combined and compared
  - “Meaning” in terms of impacts on performance will be discussed
Non-Design Labs: Cache interpreter

- Each group will add a simple “cache interpreter” to their ISA interpreter
  - Goal is to measure hit rate as a function of cache parameters
- Common benchmark & individual group program will be run on interpreter with range of cache parameters
  - Again results will be posted to web site
- In Lab Discussion:
  - Again statistics will be correlated and combined
  - Most appropriate cache parameters will be discussed
Non-Design Labs: Branch Predictor

- Each group will add a simple “branch predictor” to their ISA interpreter
  - Goal is to measure prediction rate as a function of design parameters

- Common benchmark & individual group program will be run on interpreter with range of parameters
  - Again results will be posted to web site

- In Lab Discussion:
  - Again statistics will be correlated and combined
  - Most appropriate predictor parameters will be discussed