Chapter 5 - Instruction Set Architecture

- Memory models
- Registers
- Instructions
- ISA’s
  - Pentium
  - UltraSparc
  - JVM
- Homework: Chapter 5 # 2, 5, 6, 23, 28
  - Due 5/17

Interface between software and hardware. This is NOT the OS or the “assembly language” level! Some machines implement part of the “architecture” in software!
Main Memory

• Alignment

![Alignment Diagram]

• Timing
  – Store … Load?

Words, bytes, bits.
  1) memory/bus access unit
  2) instruction operand unit

These may not be the same! Early microprocessors had a memory access size of a byte. So, words (e.g., 32 bit integers) could start on any byte, didn’t matter.

Current processors have 32 or 64 bit memory and bus.
Easiest if words are aligned on word boundaries
But, backward compatibility…sigh.

Second issue: when does access actually happen? Will the data from the store actually be in main memory by the time the load is executed? Maybe, maybe not. Sometimes compilers have to worry about this. Life is getting very complex for compiler writers. We won’t worry about this.
Three classes of registers:
1. Invisible to ISA level (all reg in MIC-1, pretty much)
2. Visible, dedicated registers (PC in MIC-1, sort of)
3. Visible, general purpose.
   1. Right shows basic registers visible at ISA level in Pentium

Let’s look at Pentium registers in detail:
EAX-EDX are general 32 bit registers. There are instructions that specifically reference the lower 16 or lower 8 bits of each of these, in which case AX, AH, AL, etc.

But not completely general. For example, there is an ISA instruction MUL SRC.
This multiplies EAX by the contents of register SRC and puts the result in EAX(low order) and EDX(high order). SRC an be EBX or ECX or?

ESI/EDI gp, also used as string source/destination pointers
EBP points to base of current stack frame (e.g., like IJVM LV)
ESP is like IJVM SP

CS-GS are “segment” pointers - ignore.

EIP is PC, Eflags is PSw

Current also have 8 x87 FPU registers, 8 MMX registers(64 bit), 8 SSE registers (128 bit)
Here are all the registers in the PIV arch. At the ISA level
Pentium Instruction Set

- Two operand format

<table>
<thead>
<tr>
<th>Mover</th>
<th>Transfer of control</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV DEST,SRC</td>
<td>Move SRC to DEST</td>
</tr>
<tr>
<td>PUSH SRC</td>
<td>Push SRC onto the stack</td>
</tr>
<tr>
<td>POP DEST</td>
<td>Pop from the stack to DEST</td>
</tr>
<tr>
<td>XCHG DEST,DI</td>
<td>Exchange DEST and DI</td>
</tr>
<tr>
<td>LOD DSTM</td>
<td>Load effective address of SRC into DEST</td>
</tr>
<tr>
<td>CDQ DSTM</td>
<td>Conditional move</td>
</tr>
</tbody>
</table>

Additive

| ADD DEST,SRC | Add SRC to DEST          |
| SUB DEST,SRC | Subtract SRC from DEST   |
| AND SRC     | AND SRC with SRC         |
| OR SRC      | OR SRC with SRC          |
| XOR SRC     | XOR SRC with SRC         |
| ADX SRC     | ADX SRC with SRC         |
| ADD DSTM,SRC | Add SRC to DSTM, then add carry bit |
| SUB DSTM,SRC | Subtract from DSTM, then subtract carry bit |
| INC DSTM    | Inc DSTM                  |
| DEC DSTM    | Dec DSTM                  |
| MUL DSTM    | MUL DSTM                  |

Binary coded decimal

| ADD DEST, SRC | Add DEST and SRC          |
| SUB DEST, SRC | Subtract SRC from DEST    |
| AND SRC      | AND SRC with SRC          |
| OR SRC       | OR SRC with SRC           |
| XOR SRC      | XOR SRC with SRC          |
| ADX SRC      | ADX SRC with SRC          |
| ADD SRC      | ADD SRC with DERiven      |
| ADD SRC      | ADD SRC with DERiven      |

Boolean

| AND DEST, SRC | AND DEST and SRC          |
| OR DEST, SRC  | OR DEST and SRC           |
| XOR DEST, SRC | XOR DEST and SRC          |
| BSR DST, SRC  | BSR DST with DERiven      |
| BSF DST, SRC  | BSF DST with DERiven      |
| PUSHA DST     | Push high word of DST onto stack |
| PUSHA DST     | Push high word of DST onto stack |
| PUSHA DST     | Push high word of DST onto stack |
| PUSHA DST     | Push high word of DST onto stack |

Tensor objects

| TEST SRC, SRC | Test SRC and SRC           |
| CMP SRC, SRC  | CMP SRC and SRC            |

Bitwise operations, set flags

<table>
<thead>
<tr>
<th>SRC DSTM</th>
<th>MOV DEST, SRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC DSTM</td>
<td>MOV DEST, SRC</td>
</tr>
<tr>
<td>SRC DSTM</td>
<td>MOV DEST, SRC</td>
</tr>
<tr>
<td>SRC DSTM</td>
<td>MOV DEST, SRC</td>
</tr>
</tbody>
</table>

But not completely general. For example, there is an ISA instruction MUL SRC.

This multiplies EAX by the contents of register SRC and puts the result in EAX(low order) and EDX (high order). SRC an be EBX or ECX or?
Data Types

<table>
<thead>
<tr>
<th>Value</th>
<th>8 bit Int</th>
<th>16 bit Int</th>
<th>32 bit int</th>
<th>BCD</th>
<th>32 bit float</th>
<th>ASCII</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0x03</td>
<td>0x0003</td>
<td>0x00000003</td>
<td>0x03</td>
<td>0x40400000</td>
<td>0x33</td>
<td>0x00000003</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td>0x0020</td>
<td>0x00000020</td>
<td>0x03</td>
<td>0x40380000</td>
<td>0x3332</td>
<td>0x00000020</td>
</tr>
</tbody>
</table>

These, except perhaps Floating point, should be straightforward. Perhaps address is a bit surprising. What is the difference between addr and 32 bit int? Nothing, really, if addr is 32 bit...

What about Unicode? Unicode is a 16 bit code, remember? The high order is 0X00 for latin-1, so the character 3 is 0X0033 in UNICODE.
Floating Point

3.0 = 0100 0000 0100 0000 0000 0000 0000 0000 ..

Bits 1 8 23

Sign Exponent

Fraction

Only a finite set of numbers can be represented.
Why do we care? Ever heard of Chaos theory?
IEEE
1 bit sign (0 = +, 1 = -)
8 (11) bit exponent - excess 127 or excess 1023
23 (52) bit fraction - add “1.” in front.

SO 0x40400000 =
positive
80 is exponent = 128 - 127 = 1
fraction = 1.1 but note this is binary!
Who doesn’t understand binary fractions?

1.101 = 1 + 1/2+0/4+1/8 = 1.675
JUST LIKE 4.506 = 4+5/10+0/100+6/1000 in decimal!

so + 1.1*2^1 = 11 (binary) = 3(decimal) (because “1.1” is binary!)

What about 0 in floating point? Just all zeros (except for sign bit, which is arbitrary).
IEEE Floating Point

- 1.0 - 0x3F800000
- 0.5 - 0x3F000000
- 2.25 - 0x40900000

My error - high order bit of significand does NOT have to be 1.

DeNormalized numbers have an exponent of ZERO.

In 32 bit float, exponent is excess 127, but 0 and 255 are NOT used as exponents, so range is +/- 1 126
In the microprogram for Mic-2, if _icmpeq6 goes to T when Z is set ot 1. However, code at T is same as goto1? Would it have been possible to go to goto1 directly?

NO!!!! Why not? How does a branch work in Mic? What does that microinstruction look like anyway?
B = H (remember, this is MIC-2
Mem = 0
C = 0
ALU = B-A
J = JAMZ
Addr = xx

If H == OPC, then Z = 1.
So? WHAT is addr sent to MPC if Z == 1?
1xx!
But where is GOTO1 - 0XA7! Why? Because it is the first microinstruction of an IJVM GOTO, which has opcode 0xA7!
JVM Instruction Set

- Most instr one byte
  - ADD
  - POP

- One byte arg
  - ILOAD IND8
  - BIPUSH CON8

- Two byte arg
  - SIPUSH CON16
  - IF_ICMPEQ OFFSET16

We will be examining three instruction sets: JVM, UltraSparc, and finally, Pentium.

JVM is, like IJVM, a stack-oriented instruction set.

Only Load/Store reference the stack
Push references the constant pool
Various branches reference code pool
Except for Array, which we won’t discuss, these are the only ones that need anything other than the opcode to define. So, all other instructions are just one byte long.

JVM supports 8, 16, 32 and 64 bit integers, 32 bit and 64 bit float
It also supports 16 bit (UNICODE) chars.

Many of the instructions above are actually four different instructions! Where it says “type LOAD IND8”, for example, there are actually four instructions: ILOAD IND8, LLOAD IND8, FLOAD IND*, and DLOAD IND*

I = Integer (32 bit)
L = LONG (64 bit)
F = Float (32 bit)
D = Double (64 bit float)
JVM Instruction formats and Addressing Modes

- **OPCODE/STACK**
  - ADD

- **OPCODE+ADDR/INDEXED**
  - ILOAD IND8

- **OPCODE+OP1/IMMEDIATE**
  - BIPUSH CON8

JVM arith is stack-based - no addressing needed there.

There are three areas of memory to be addressed:

The stack (for local vars), the constant pool, and the code area. Each is addressed as an offset from an implicit register, one dedicated to each area. This use of an offset is called INDEXED addressing.

Since there is only one for each area, again these don’t have to be named in instructions, so only one operand: the offset

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Pentium II</th>
<th>UltraSPARC II</th>
<th>JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Direct</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Register indirect</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Based-indexed</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Infix, Prefix, Reverse Polish

<table>
<thead>
<tr>
<th>Infix</th>
<th>Reverse Polish notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B × C</td>
<td>A B C × +</td>
</tr>
<tr>
<td>A × B + C</td>
<td>A B × C +</td>
</tr>
<tr>
<td>A × B + C × D</td>
<td>A B × C D × +</td>
</tr>
<tr>
<td>(A + B) / (C - D)</td>
<td>A B + C D - /</td>
</tr>
<tr>
<td>A × B / C</td>
<td>A B × C /</td>
</tr>
<tr>
<td>((A + B) × C + D) / (E + F + G)</td>
<td>A B + C D + E F + G + /</td>
</tr>
</tbody>
</table>

- (8+2*5)/(1+3*2-4)

Reverse polish actually.

What is prefix? *+ABC…

Could we BIPUSH 8 AFTER IMUL? (yes, but ONLY because op is + - wouldn’t work for -!

Everyone should know this - who doesn’t feel comfortable with these conversions? Expect a mid 2 question on this…
Arrays and Indexed Addressing

public void plusOne (int [] x; int I) {
    j = x[i];
x[I] = j+1;
}

• ILOAD 1
• ILOAD 2
• IALOAD
• ISTORE 3
• ILOAD 1
• ILOAD 2
• ILOAD 3
• BIPUSH 1
• IADD
• IASTORE
• RETURN

JVM has dedicated instructions for handling arrays.

Public void plusOne (int [] x; int I) {
    j = x[i];
x[I] = j+1;
}

The ADDRESS of the array, rather than its VALUE, will be pushed on the stack! (call by REFERENCE!)

The VALUE of I will be pushed on the stack.

IALOAD pops the top two values on the stack, adds them, and pushes the value AT THAT ADDR onto the stack!

This is indexed addressing, except that the operand values are on the stack instead of in the instruction. We’ll see another form of this when we look at the Ultrasparc.

So let’s step through the call to plusOne at the right of the screen

Note the cpu must know how long an array entry is! So, a variety of ALOAD and ASTORE instructions, one for each value length
Instruction formats galore

- ILOAD 1  •  0x1501  •  2/4
- BIPUSH 0 •  0x0100  •  2/0
- SIPUSH 0 •  0x110000  •  3/0
- ILOAD_1 •  0x1B  •  1/4
- WIDE ILOAD 1 •  0xC4150001  •  (1+3)/4
- ICONST_0 •  0x03  •  1/0
- LDC 3 •  0x1203  •  2/4
- LDC_W 257 •  0x130101  •  3/4

We’ve seen at least two ways to set a register to zero:

ILOAD 1 (assuming LV 1 contains 0)
BIPUSH 0

Both take two bytes for instruction, and ILOAD takes an additional 4 bytes for data. So, one might expect ILOAD to take longer.

This must be very common. And, we haven’t used up all 256 available opcodes yet, so why not dedicate one just to this?

ILOAD_1 (0x1A) loads local var 0 on the stack! Now we can do it in just 1 byte! (Issue isn’t memory use, although that matters, but really how many memory accesses we need to perform to accomplish the instruction).

WIDE ILOAD 1: WIDE is a PREFIX that says the ILOAD index is 16 bits, not 8 bits. Why?
Need to access lv beyond 255! (Who would write code that way? Never mind…)

Finally, ICONST_0 is the simplest of all: one byte that says push a zero (int) on the stack.

Yet another way to do this is with LDC: to load from the constant area. Note different way to use 16 bit index. Why? Because 16 bit index is MUCH more common for constant area..

So, how would we ACTUALLY do this?
If we really want to set to the constant 0, ICONST_0.
If we really want to set to some less common constant < 255, BIPUSH n
If we really want to set to some even less common constant < 65…. SIPUSH n
If we really want to set to some even less common constant LDC < 8 bit offset of n>
Flow of Control

- IADD (1)
- LCMP (1)
- FCMPL (1)
- GOTO OFFSET16 (3)
- IF_ICMPrel OFFSET16 (3)
- Ifrel OFFSET16 (3)
- IFNULL OFFSET16 (3)

Instructions that “do something” set a condition code (IADD, etc)
Also explicit compare instr. For non-integer data types
Finally, Conditional and unconditional branch test condition code and set PC

LCMP Takes two two-word long integers off the stack and compares them. If the two integers are the same, the int 0 is pushed onto the stack. If value2 is greater than value1, the int 1 is pushed onto the stack. If value1 is greater than value2, the int -1 is pushed onto the stack.

; This is like the Java expression:
;   (x > 0)
; where x is a long.

lload_1    ; push the long int in local variable 1
lconst_0   ; push the long integer 0 onto the stack
lcmp       ; compare the two long integers

; The integer result on the stack is:
;   0 if local variable 1 equals 0
;  -1 if local variable 1 is less than 0
;   1 if local variable 1 is greater than 0

lcmp opcode = 0x94 (148)

Fcmlp: If the two numbers are the same, the integer 0 is pushed onto the stack. If value2 is greater than value1, the integer 1 is pushed onto the stack. If value1 is greater than value2, the
Goto considered Harmful

- Procedures
  - Simple
  - Recursive

Up to 1970 or so everyone used goto at the higher level language level, just like in assembler. However, we now believe avoiding goto leads to much more readable code.

Avoiding Goto means we need other flow control:
Blocks (braces), loops (for, while, loop), procedures, co-routines.
You should know about most of these. We’ll review recursive once more, ignore co-routines.
Towers of Hanoi

public void towers(int n, int i, int j) {
    int k;
    if (n == 1)
        System.out.println("Move a disk from "+ i + " to "+ j);
    else {
        k = 6 - i - j;
        towers(n - 1, i, k);
        towers(1, i, j);
        towers(n - 1, k, j);
    }
}

Claim - the above code can solve an arbitrary Towers of Hanoi problem. (assuming we start with a valid stack on one peg, others empty of anything smaller.)

Let’s step through and see what happens.

Call sequence from top level will be:

Towers (3, 1, 3)
Towers (2, 1, 2)
Towers (1, 1, 3)
Towers (2, 2, 3)

But it doesn’t end there! These do further calls.
Towers of Hanoi

```java
towers(int n, int i, int j) {
    if (n == 1)
        System.out.println("Move a disk from " + i + " to " + j);
    else {
        k = 6 - i - j;
        towers(n - 1, i, k);
        towers(1, i, j);
        towers(n - 1, k, j);
    }
}
```

So what does a procedure call have to do?

Quite a lot, same for a return. - set up arguments, save FP and PC, allocate locals, …

This is called a procedure epilog, and the faster it is, the better.

JVM has INVOKEVIRTUAL, of which we saw a simple version when we studied JVM. This did almost everything EXCEPT put arguments in place. Still, it takes 22 mic-1 instructions! A BIPUSH only takes 3!, and ILOAD 5, an ISTORE 6, and IADD3. So 4-7 times as long as a basic operation! (and RETURN takes another 8)!
So what does Towers look like in JVM?

Ignoring the print statement, 27 inst, 36 bytes of code (by my count), all constants are immediate.

Anything else to notice?
UltraSparc is a very regular, RISC-style instruction set. No legacy that needs support.

Only load/store/branch refer to memory. This is it. Small instruction set.

This is a register-oriented machine, with support for byte, 16-bit, 32-bit, and 64 bit ops.

32 general purpose registers, although by convention some are dedicated by software for specific uses. Well, actually 31 registers and R0 - the constant 0.

Unlike IJVM, UltraSparc doesn’t use operand stack for procedure calls - uses registers for that - note allocation of R16-23 for local vars, R8-13 for parameters, R31 for return addr.

So what might i = j+k look like?

Skip stuff about register windows for this class.

Ok, let’s look at some specific instructions
UltraSparc
Instruction Set

Format 1 (op = 1): CALL

Format 2 (op = 0): SETHI & Branches (Bicc, FBicc, CBicc, SETHI)

Format 3 (op = 2 or 3): Remaining instructions

<table>
<thead>
<tr>
<th>Register</th>
<th>Alt. name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>G0</td>
<td>Hardwired to 0. Stores into it are just ignored.</td>
</tr>
<tr>
<td>R1 – R7</td>
<td>G1 – G7</td>
<td>Holds global variables</td>
</tr>
<tr>
<td>R8 – R13</td>
<td>O0 – O5</td>
<td>Holds parameters to the procedure being called</td>
</tr>
<tr>
<td>R14</td>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>R15</td>
<td>ST</td>
<td>Scratch register</td>
</tr>
<tr>
<td>R16 – R23</td>
<td>L0 – L7</td>
<td>Holds local variables for the current procedure</td>
</tr>
<tr>
<td>R24 – R29</td>
<td>I0 – I5</td>
<td>Holds incoming parameters</td>
</tr>
<tr>
<td>R30</td>
<td>FP</td>
<td>Pointer to the base of the current stack frame</td>
</tr>
<tr>
<td>R31</td>
<td>FT</td>
<td>Holds return address for the current procedure</td>
</tr>
</tbody>
</table>

Loads and Stores are simple, an opcode, a register, and a memory addr.

Twice as many loads - need to specify how to extend sign if loaded is less than 64-bit.

Upper right shows instruction formats.
First two bits just define instruction format!

Top format is for a procedure call only. Load/Store use Format 3 op3 (bottom).
Rd is destination register (5 bits = 32 registers)
Op3 specifies “actual” opcode - load/store/etc

Rs1 and rs2 point to two registers that are added together to get addr of location to load/store
(or, format 3 second variety - a register push an immediate, sign-extended, offset)

The second load format is just what we did in IJVM with ILOAD 5.
In UltraSparc, that might be LDSW (R30,5),R15 (R30 is base of stack frame, R15 is scratch reg).
## UltraSparc Instruction Set

### Format 1 (op = 1): CALL

<table>
<thead>
<tr>
<th>op</th>
<th>disp30</th>
</tr>
</thead>
</table>

### Format 2 (op = 0): SETHI & Branches (Bicc, FBicc, CBecc)

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op2</th>
<th>imm22</th>
</tr>
</thead>
</table>

### Format 3 (op = 2 or 3): Remaining instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op2</th>
<th>rs1</th>
<th>i=0</th>
<th>asi</th>
<th>rs2</th>
</tr>
</thead>
</table>

### Arithmetic

<table>
<thead>
<tr>
<th>Register</th>
<th>Alt. name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>G0</td>
<td>Hardwired to 0. Stores into it are just ignored.</td>
</tr>
<tr>
<td>R1 – R7</td>
<td>G1 – G7</td>
<td>Holds global variables</td>
</tr>
<tr>
<td>R8 – R13</td>
<td>O0 – O5</td>
<td>Holds parameters to the procedure being called</td>
</tr>
<tr>
<td>R14</td>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>R15</td>
<td>G7</td>
<td>Scratch register</td>
</tr>
<tr>
<td>R16 – R23</td>
<td>L0 – L7</td>
<td>Holds local variables for the current procedure</td>
</tr>
<tr>
<td>R24 – R29</td>
<td>I0 – I5</td>
<td>Holds incoming parameters</td>
</tr>
<tr>
<td>R30</td>
<td>FP</td>
<td>Pointer to the base of the current stack frame</td>
</tr>
<tr>
<td>R31</td>
<td>I7</td>
<td>Holds return address for the current procedure</td>
</tr>
</tbody>
</table>

Arithmetic is TRIADIC: op1 +/-/* op2 -> dst

Again, Format 3 type 2 or 3

Rd is dst

Rs1 is source 1

Either rs2 or simm13 (immediate) is source2
Towers of Hanoi - UltraSparc

```java
public void towersn(int n, int i, int k) {
    if (n == 1) {
        System.out.println("Move a disk from "+i+" to "+k);
        return;
    } else {
        int j = 6 - i - k;
        towers(n - 1, k, j);
        towers(n - 1, i, j);
        towers(n - 1, k, i);
    }
}
```

So what does Towers look like in UltraSparc?

Locals are in registers, as are parameters, so if statement is fast.

Let’s skip the print statement Next interesting thing to notice is the nop at the end - why?
Because SPARC always executes the next instruction after a branch! (It’s the pipeline!)

At the else, again, I, J, K are locals, so they are in registers. Just one machine instruction to do each.
(AND, each of these instructions can be executed in ONE microcycle!)

Next weirdness: let’s look at the setup for one of the tower recursive calls.
Three parameters to pass, but wait, the third parameter is set AFTER the call! Again, one instruction after a branch is already in the pipeline.

Same thing after the return. (but I’m not going to explain register windows…)

20 instructions (vs 27 in IJVM), 80 bytes (vs 37 in IJVM)
Like the UltraSparc, register-oriented rather than stack oriented.

But, unlike UltraSparc, DIADIC instead of TRIADIC. (DST = SRC1 for things like ADD).

So ADD, DST, SRC means add contents of src to dest and put result back in dst.

Source OR dest can be a memory ref, but not both (in general).

Also, Arith/Logical can reference memory directly. This makes instructions MUCH more complex to decode, lots of formats because of legacy back to 8088.
**Pentium ISA**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 ib</td>
<td>ADC AL,imm8</td>
<td>Add with carry imm8 to AL</td>
</tr>
<tr>
<td>15 iw</td>
<td>ADC AX,imm16</td>
<td>Add with carry imm16 to AX</td>
</tr>
<tr>
<td>15 id</td>
<td>ADC EAX,imm32</td>
<td>Add with carry imm32 to EAX</td>
</tr>
<tr>
<td>89/2 ib</td>
<td>ADC r/m8,imm8</td>
<td>Add with carry imm8 to r/m8</td>
</tr>
<tr>
<td>81/2 lw</td>
<td>ADC r/m16,imm16</td>
<td>Add with carry imm16 to r/m16</td>
</tr>
<tr>
<td>81/2 ld</td>
<td>ADC r/m32,imm32</td>
<td>Add with CF imm32 to r/m32</td>
</tr>
<tr>
<td>83/2 ib</td>
<td>ADC r/m16,imm8</td>
<td>Add with CF sign-extended imm8</td>
</tr>
<tr>
<td>83/2 ib</td>
<td>ADC r/m32,imm8</td>
<td>Add with CF sign-extended imm8</td>
</tr>
<tr>
<td>10 lr</td>
<td>ADC r/m8,r8</td>
<td>Add with carry byte register to r/m</td>
</tr>
<tr>
<td>11 lr</td>
<td>ADC r/m16,r16</td>
<td>Add with carry r16 to r/m16</td>
</tr>
<tr>
<td>11 lr</td>
<td>ADC r/m32,r32</td>
<td>Add with CF r32 to r/m32</td>
</tr>
<tr>
<td>12 lr</td>
<td>ADC r8,r/m8</td>
<td>Add with carry r/m8 to byte register</td>
</tr>
<tr>
<td>13 lr</td>
<td>ADC r16,r/m16</td>
<td>Add with carry r/m16 to r16</td>
</tr>
<tr>
<td>13 lr</td>
<td>ADC r32,r/m32</td>
<td>Add with CF r32 to r32</td>
</tr>
</tbody>
</table>

Another difference from ultrasparc/JVM is that arithmetic/logical instruction can reference memory directly. So don’t need load/store.

Notice how many “Add” instructions there are!
And, instructions can be quite long (up to 16 bytes!)

imm8 - immediate 8 bit constant in instruction
imm16 - immediate 16 bit
imm32 - immediate 32 bit
r/m8 - either one of the 8bit registers (AL, AH, ...) OR a pointer to an 8bit value in memory.
r/m16, r/m32 - as you might guess (16 bit regs: AX, BX, CX, DX, SP, BP, SI, DI)
Like UltraSparc, Pentium takes fewer instructions than IJVM to compute K. But notice Kisin memory rather than in a register as was in UltraSparc. And, instructions are longer. So hard to compare.

But we can say this will take four memory references.

If in a loop, no big deal, (because data will be in cache). But first time we will pay a hit.

Also, my best guess is that this code is about 1.5X the size of the sparc code.

Overall, 33 instructions, longest yet, even though it has highly complex instruction set. Complex instructions should mean shorter code, no?

If Sparc can do same thing in 20 inst it takes 33 to do on Pentium, how do you compare “MIPS”? (You don’t try, you run benchmarks).

What else to see here?

Well, how about procedure call: Pretty much the same, push K on stack, then load I, push it on stack, then load n, subtract 1, and push it on stack. Finally call towers recursively. Now we see where some of the extra instructions came from…
Traps -
No such memory address
Arithmetic overflow (NAN)
Stack overflow (memory allocated to stack exhausted).

For 2, we could write ISA level code to check after every arithmetic operation, but SLOOOOW.

Instead, steal an idea from the MIC-1 conditional branch architecture:
Force PC to new value based on conditions during instruction execution.

Interrupts -

Same idea as trap, but for external events.
For example, suppose we are waiting for a disk to come around to the sector we want to read. What should CPU do?
Could be in a loop checking if disk is there yet. But: could take 10 ms: that’s 100,000 inst on a 1GIPS machine! What a waste!

Again, another input line: Force PC to new value based on external (typically I/O) events.