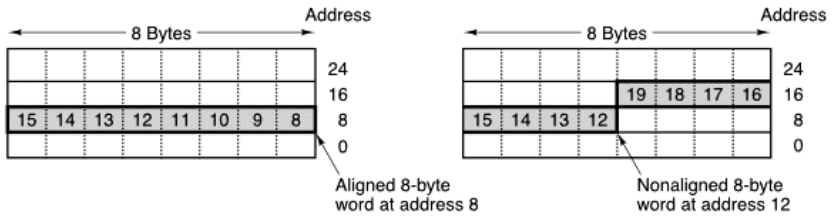
Chapter 5 - Instruction Set Architecture

- 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



- 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.

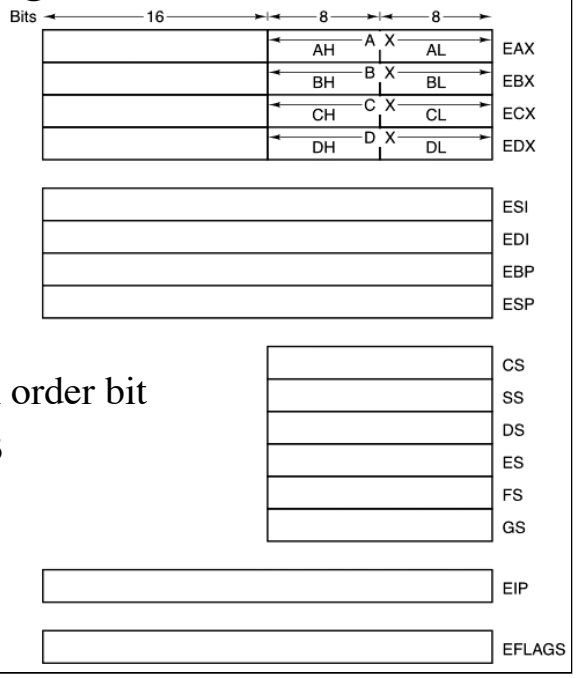
Easiest if words are aligned on word boundaries

But, backward compatiblity…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.

Registers

- PSW (Eflags)
  - N - result negative
  - Z - result Zero
  - V - overflow
  - C - carry out of high order bit
  - A - carry out of bit 3
  - P - even parity



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)

Pentium IV

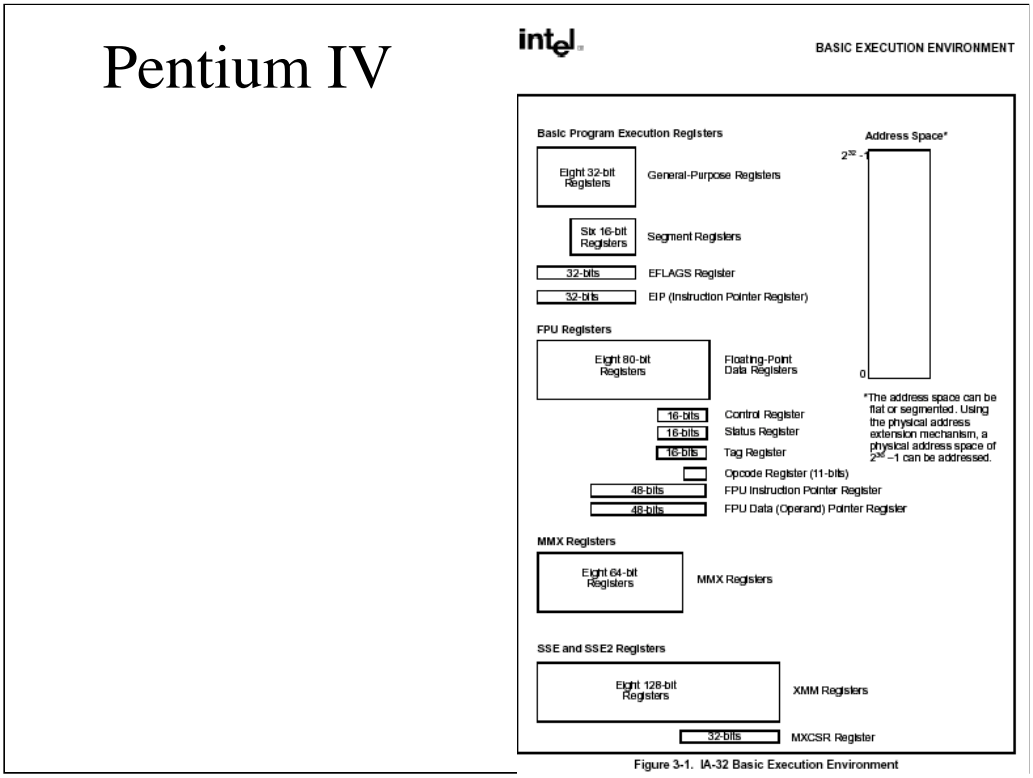


Figure 3-1. IA-32 Basic Execution Environment

Here are all the registers in the PIV arch. At the ISA level

Pentium Instruction Set

- Two operand format

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOV DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination

# = shift/rotate count
LV = # locals

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

Value	8 bit Int	16 bit Int	32 bit int	BCD	32 bit float	ASCII	Address
3	0x03	0x0003	0x00000003	0x03	0x40400000	0x33	0x00000003
32	0x20	0x0020	0x00000020	0x0302	0x04038000	0x3332	0x00000020

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×		
Unsigned integer	×	×	×		
Binary coded decimal integer	×				
Floating point			×	×	

Figure 5-6. The Pentium II numeric data types. Supported

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer	×	×	×	×	
Binary coded decimal integer					
Floating point			×	×	×

Figure 5-7. The UltraSPARC II numeric data types.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer					
Binary coded decimal integer					
Floating point			×	×	

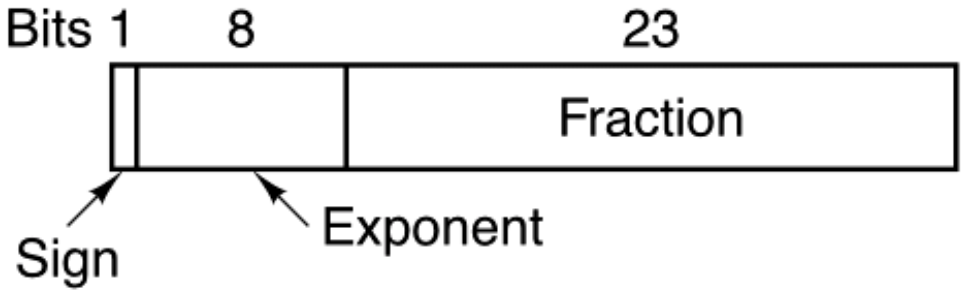
Figure 5-8. The JVM numeric data types.

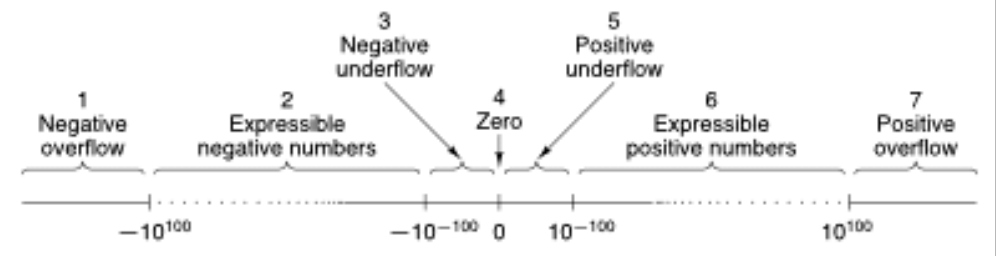
These, except perhaps Floating point, should be straightforward. Perhaps address is a bit surprising. What is the difference between addr and 32 bit int? Nothing, really, if addr is 32 bit…

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 ..





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

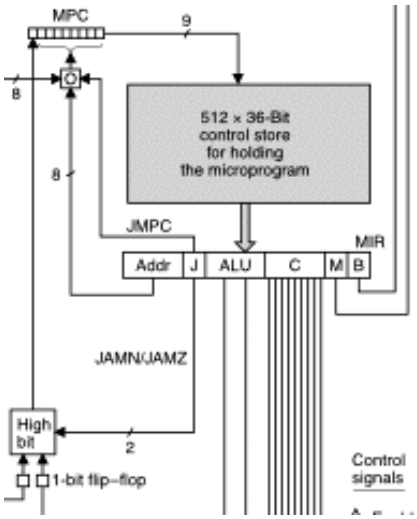
- 1.0 - 0x3F800000
- 0.5 - 0x3F000000
- 2.25 - 0x40900000

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

Chapter 4 # 24



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

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer					
Binary coded decimal integer					
Floating point			×	×	

Figure 5-8. The JVM numeric data types.

Loads

typeLOAD IND8	Push local variable onto stack
typeALOAD	Push array element on stack
BALOAD	Push byte from an array on stack
SALOAD	Push short from an array on stack
CALOAD	Push char from an array on stack
AALOAD	Push pointer from an array on "

Stores

typeSTORE IND8	Pop value and store in local var
typeASTORE	Pop value and store in array
BASTORE	Pop byte and store in array
SASTORE	Pop short and store in array
CASTORE	Pop char and store in array
AASTORE	Pop pointer and store in array

Pushes

BIPUSH CON8	Push a small constant on stack
SIPUSH CON16	Push 16-bit constant on stack
LDC IND8	Push constant from const pool
typeCONST_#	Push immediate constant
ACONST_NULL	Push a null pointer on stack

Arithmetic

typeADD	Add
typeSUB	Subtract
typeMUL	Multiple
typeDIV	Divide
typeREM	Remainder
typeNEG	Negate

Boolean/shift

iiAND	Boolean AND
iiOR	Boolean OR
iiXOR	Boolean EXCLUSIVE OR
iiSHL	Shift left
iiSHR	Shift right
iiUSHR	Unsigned shift right

Conversion

x2y	Convert x to y
i2c	Convert integer to char
i2b	Convert integer to byte

Stack management

DUPxx	Six instructions for duping
POP	Pop an int from stk and discard
POP2	Pop two ints from stk and discard
SWAP	Swap top two ints on stack

Comparison

IF_ICMPrel OFFSET16	Conditional branch
IF_ACMPEQ OFFSET16	Branch if two ptrs equal
IF_ACMPNE OFFSET16	Branch if ptrs unequal
IFrel OFFSET16	Test 1 value and branch
IFNULL OFFSET16	Branch if ptr is null
IFNONNULL OFFSET16	Branch if ptr is nonnull
LCMP	Compare two longs
FCMPL	Compare 2 floats for <
FCMPG	Compare 2 floats for >
DCMPL	Compare doubles for <
DCMPG	Compare doubles for >

Transfer of control

INVOKEVIRTUAL IND16	Method invocation
INVOKESTATIC IND16	Method invocation
INVOKEINTERFACE ...	Method invocation
INVOKESPECIAL IND16	Method invocation
JSR OFFSET16	Invoke finally clause
typeRETURN	Return value
ARETURN	Return pointer
RETURN	Return void
RET IND8	Return from finally
GOTO OFFSET16	Unconditional branch

Arrays

ANEWARRAY IND16	Create array of ptrs
NEWARRAY ATYPE	Create array of atype
MULTINEWARRAY IN16,D	Create multidim array
ARRAYLENGTH	Get array length

Miscellaneous

IINC IND8,CON8	Increment local variable
WIDE	Wide prefix
NOP	No operation
GETFIELD IND16	Read field from object
PUTFIELD IND16	Write field to object
GETSTATIC IND16	Get static field from class
NEW IND16	Create a new object
INSTANCEOF OFFSET16	Determine type of obj
CHECKCAST IND16	Check object type
ATHROW	Throw exception
LOOKUPSWITCH ...	Sparse multiway branch
TABLESWITCH ...	Dense multiway branch
MONITORENTER	Enter a monitor
MONITOREXIT	Leave a monitor

IND8/16 = index of local variable type, x, y = I, L, F, D
CON8/16, D, ATYPE = constant OFFSET16 for branch

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 dicuss, 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 "typeLOAD IND8", for example, there are actually four instructions: ILOAD IND8, LLOAD IND8, FLOAD IND*, and DLOAD IND*

I = Integer (32 bit)

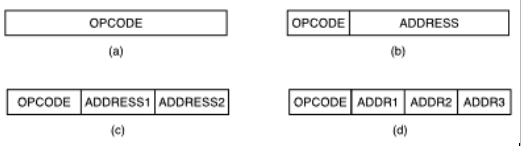
L = LONG (64 bit)

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**



Addressing mode	Pentium II	UltraSPARC II	JVM
Immediate	×	×	×
Direct	×		
Register	×	×	
Register indirect	×		
Indexed	×	×	×
Based-indexed		×	
Stack			×

ILOAD	IND8	SIPUSH	CON	16	ADD

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

Infix, Prefix, Reverse Polish

Infix	Reverse Polish notation
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C – D)	A B + C D – /
A × B / C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

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

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

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

Loads

typeLOAD IND8	Push local variable onto stack
typeALOAD	Push array element on stack
BALOAD	Push byte from an array on stack
SALOAD	Push short from an array on stack
CALOAD	Push char from an array on stack
AALOAD	Push pointer from an array on "

public void plusOne (int [] x; int I) {

j = x[i];

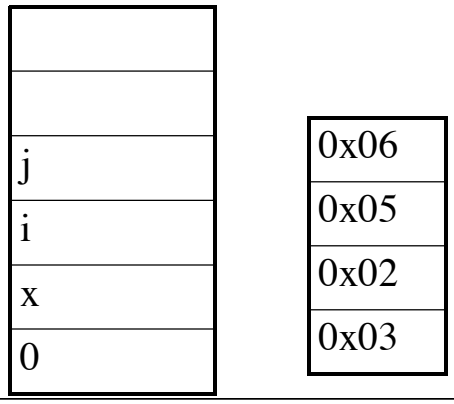
x[I] = j+1;

}

- **ILOAD 1**
- **ILOAD 2**
- **IALOAD**
- **ISTORE 3**
- **ILOAD 1**
- **ILOAD 2**
- **ILOAD 3**
- **BIPUSH 1**
- **IADD**
- **IASTORE**
- **RETURN**

int z = {3, 2, 5, 6};

plusOne(z, 2);



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)

If (x > y) {…

FLOAD x
FLOAD y
FCMPL
IFle L2

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

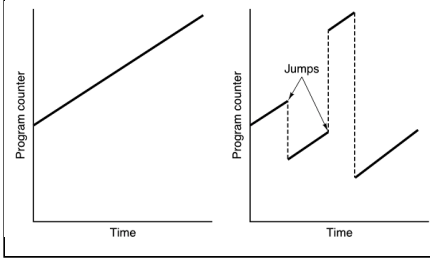
; The integer result on the stack is:
; 0 if local variable 1 equals 0
; -1 if local variable 1 is less than 0
; 1 if local variable 1 is greater than 0

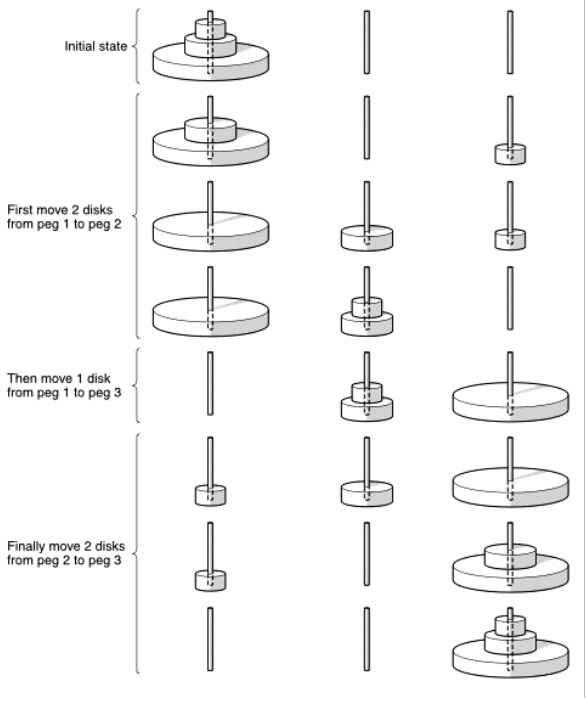
lcmp opcode = 0x94 (148)

Fcmpl: 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:

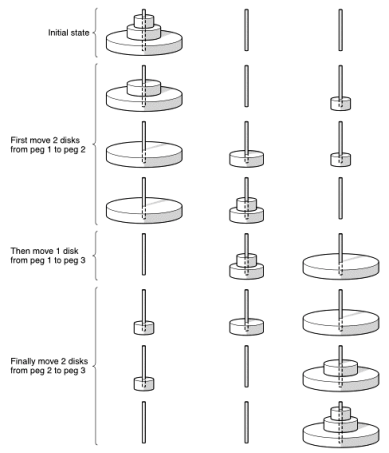
Blocks (braces), loops (for, while, loop), procedures, co-routines.

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);
  }
}
```



Tower called with n=3, i=1, j=3
Tower called with n=2, i=1, j=2
Tower called with n=1, i=1, j=3
Move a disk from 1 to 3
Tower called with n=1, i=1, j=2
Move a disk from 1 to 2
Tower called with n=1, i=3, j=2
Move a disk from 3 to 2
Tower called with n=1, i=1, j=3
Move a disk from 1 to 3
Tower called with n=2, i=2, j=3
Tower called with n=1, i=2, j=1
Move a disk from 2 to 1
Tower called with n=1, i=2, j=3
Move a disk from 2 to 3
Tower called with n=1, i=1, j=3
Move a disk from 1 to 3

Claim - the above code can solve an aribitrary 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)

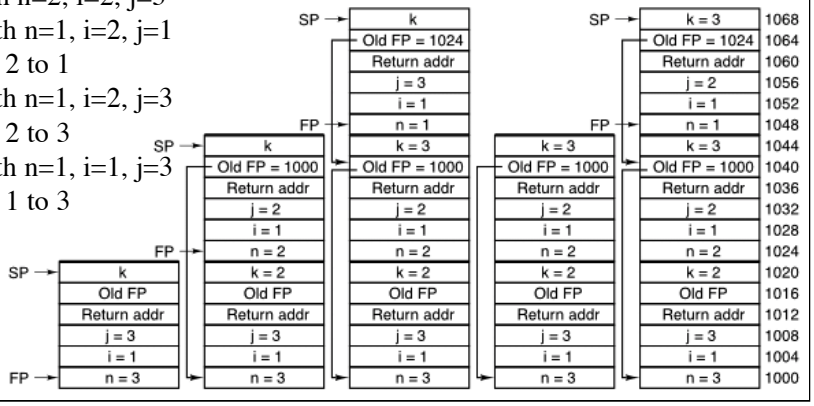
Butit doesn't end there! These do further calls.

Towers of Hanoi

```
Tower called with n=3, i=1, j=3
 Tower called with n=2, i=1, j=2
  Tower called with n=1, i=1, j=3
Move a disk from 1 to 3
  Tower called with n=1, i=1, j=2
Move a disk from 1 to 2
  Tower called with n=1, i=3, j=2
Move a disk from 3 to 2
 Tower called with n=1, i=1, j=3
Move a disk from 1 to 3
 Tower called with n=2, i=2, j=3
  Tower called with n=1, i=2, j=1
Move a disk from 2 to 1
  Tower called with n=1, i=2, j=3
Move a disk from 2 to 3
  Tower called with n=1, i=1, j=3
Move a disk from 1 to 3
```

```
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);
  }
}
```



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 IJVM. 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)!

Towers of Hanoi

```
        ILOAD_0             // local 0 = n; push n
        ICONST_1            // push 1
        IF_ICMPNE L1        // if (n != 1) goto L1

        GETSTATIC #13       // n == 1; this code handles the println statement
        NEW #7              // allocate buffer for the string to be built
        DUP                 // duplicate the pointer to the buffer
        LDC #2              // push pointer to string "move a disk from "
        INVOKESPECIAL #10   // copy the string to the buffer
        ILOAD_1             // push i
        INVOKEVIRTUAL #11   // convert i to string and append to the new buffer
        LDC #1              // push pointer to string " to "
        INVOKEVIRTUAL #12   // append this string to the buffer
        ILOAD_2             // push j
        INVOKEVIRTUAL #11   // convert j to string and append to buffer
        INVOKEVIRTUAL #15   // string conversion
        INVOKEVIRTUAL #14   // call println
        RETURN              // return from towers
L1:     BIPUSH 6            // Else part: compute k = 6 – i – j
        ILOAD_1             // local 1 = i; push i
        ISUB                // top-of-stack = 6 – i
        ILOAD_2             // local 2 = j; push j
        ISUB                // top-of-stack = 6 – i – j
        ISTORE_3            // local 3 = k = 6 – i – j; stack is now empty

        ILOAD_0             // start working on towers(n – 1, i, k); push n
        ICONST_1            // push 1
        ISUB                // top-of-stack = n – 1
        ILOAD_1             // push i
        ILOAD_3             // push k
        INVOKESTATIC #16    // call towers(n – 1, 1, k)

        ICONST_1            // start working on towers(1, i, j); push 1
        ILOAD_1             // push i
        ILOAD_2             // push j
        INVOKESTATIC #16    // call towers(1, i, j)

        ILOAD_0             // start working on towers(n – 1, k, j); push n
        ICONST_1            // push 1
        ISUB                // top-of-stack = n – 1
        ILOAD_3             // push k
        ILOAD_2             // push j
        INVOKESTATIC #16    // call towers(n – 1, k, j)
        RETURN              // return from towers
```

```
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);
   }
}
```

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 Instruction Set

- I = j+k;
  - ?

Loads

LDSB ADDR,DST	Load signed byte (8 bits)
LDUB ADDR,DST	Load unsigned byte (8 bits)
LDSH ADDR,DST	Load signed halfword (16 bits)
LDUH ADDR,DST	Load unsigned halfword (16)
LDSW ADDR,DST	Load signed word (32 bits)
LDUW ADDR,DST	Load unsigned word (32 bits)
LDX ADDR,DST	Load extended (64-bits)

Stores

STB SRC,ADDR	Store byte (8 bits)
STH SRC,ADDR	Store halfword (16 bits)
STW SRC,ADDR	Store word (32 bits)
STX SRC,ADDR	Store extended (64 btis)

Arithmetic

ADD R1,S2,DST	Add
ADDCC "	Add and set icc
ADDC "	Add with carry
ADDCCC "	Add with carry and set icc
SUB R1,S2,DST	Subtract
SUBCC "	Subtract and set icc
SUBC "	Subtract with carry
SUBCCC "	Subtract with carry and set icc
MULX R1,S2,DST	Multiply
SDIVX R1,S2,DST	Signed divide
UDIVX R1,S2,DST	Unsigned divide
TADCC R1,S2,DST	Tagged add

Shifts/rotates

SLL R1,S2,DST	Shift left logical (64 bits)
SLLX R1,S2,DST	Shift left logical extended (64)
SRL R1,S2,DST	Shift right logical (32 bits)
SRLX R1,S2,DST	Shift right logical extended (64)
SRA R1,S2,DST	Shift right arithmetic (32 bits)
SRAX R1,S2,DST	Shift right arithmetic ext. (64)

Boolean

AND R1,S2,DST	Boolean AND
ANDCC "	Boolean AND and set icc
ANDN "	Boolean NAND
ANDNCC "	Boolean NAND and set icc
OR R1,S2,DST	Boolean OR
ORCC "	Boolean OR and set icc
ORN "	Boolean NOR
ORNCC "	Boolean NOR and set icc
XOR R1,S2,DST	Boolean XOR
XORCC "	Boolean XOR and set icc
XNOR "	Boolean EXCLUSIVE NOR
XNORCC "	Boolean EXCL. NOR and set icc

Transfer of control

BPcc ADDR	Branch with prediction
BPr SRC,ADDR	Branch on register
CALL ADDR	Call procedure
RETURN ADDR	Return from procedure
JMPL ADDR,DST	Jump and Link
SAVE R1,S2,DST	Advance register windows
RESTORE "	Restore register windows
Tcc CC,TRAP#	Trap on condition
PREFETCH FCN	Prefetch data from memory
LDSTUB ADDR,R	Atomic load/store
MEMBAR MASK	Memory barrier

Miscellaneous

SETHI CON,DST	Set bits 10 to 31
MOVcc CC,S2,DST	Move on condition
MOVr R1,S2,DST	Move on register
NOP	No operation
POPC S1,DST	Population count
RDCCR V,DST	Read condition code register
WRCCR R1,S2,V	Write condition code register
RDPC V,DST	Read program counter

SRC = source register
DST = destination register
R1 = source register
S2 = source: register or immediate
ADDR = memory address

TRAP# = trap number
FCN = function code
MASK = operation type
CON = constant
V = register designator

CC = condition code set
R =destination register

cc = condition
r = LZ,LEZ,Z,NZ,GZ,GEZ

Register	Alt. name	Function
R0	G0	Hardwired to 0. Stores into it are just ignored.
R1 – R7	G1 – G7	Holds global variables
R8 – R13	O0 – O5	Holds parameters to the procedure being called
R14	SP	Stack pointer
R15	O7	Scratch register
R16 – R23	L0 – L7	Holds local variables for the current procedure
R24 – R29	I0 – I5	Holds incoming parameters
R30	FP	Pointer to the base of the current stack frame
R31	I7	Holds return address for the current procedure

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?

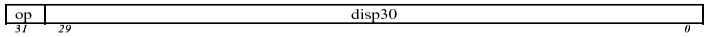
Skip stuff about register windows for this class.

Ok, let's look at some specific instructions

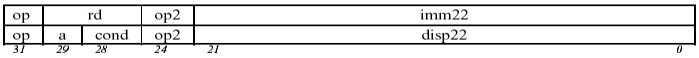
UltraSparc Instruction Set

Format	*op*	Instructions
1	1	CALL
2	0	Bicc, FBfcc, CBccc, SETHI
3	3	memory instructions
3	2	arithmetic, logical, shift, and remaining

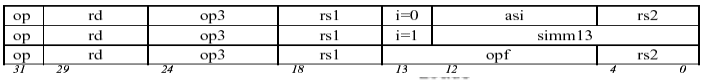
Format 1 (*op* = 1): CALL



Format 2 (*op* = 0): SETHI & Branches (Bicc, FBfcc, CBccc)



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



Register	Alt. name	Function
R0	G0	Hardwired to 0. Stores into it are just ignored.
R1 – R7	G1 – G7	Holds global variables
R8 – R13	O0 – O5	Holds parameters to the procedure being called
R14	SP	Stack pointer
R15	O7	Scratch register
R16 – R23	L0 – L7	Holds local variables for the current procedure
R24 – R29	I0 – I5	Holds incoming parameters
R30	FP	Pointer to the base of the current stack frame
R31	I7	Holds return address for the current procedure

LDSB ADDR,DST	Load signed byte (8 bits)
LDUB ADDR,DST	Load unsigned byte (8 bits)
LDSH ADDR,DST	Load signed halfword (16 bits)
LDUH ADDR,DST	Load unsigned halfword (16)
LDSW ADDR,DST	Load signed word (32 bits)
LDUW ADDR,DST	Load unsigned word (32 bits)
LDX ADDR,DST	Load extended (64-bits)

Stores

STB SRC,ADDR	Store byte (8 bits)
STH SRC,ADDR	Store halfword (16 bits)
STW SRC,ADDR	Store word (32 bits)
STX SRC,ADDR	Store extended (64 btis)

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)

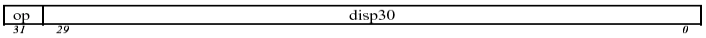
The second load format is just what we did in IJVM with ILOAD 5.

In UltraSparc, that might be LDSW (R30,5),R15 (R30 is base of stack frame, R15 is scratch reg).

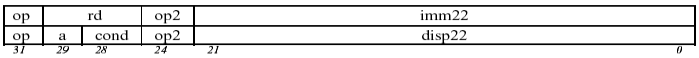
UltraSparc Instruction Set

Format	*op*	Instructions
1	1	CALL
2	0	Bicc, FBfcc, CBccc, SETHI
3	3	memory instructions
3	2	arithmetic, logical, shift, and remaining

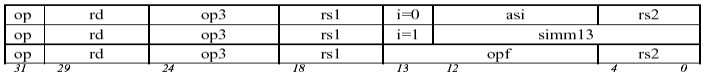
Format 1 (*op* = 1): CALL



Format 2 (*op* = 0): SETHI & Branches (Bicc, FBfcc, CBccc)



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



Register	Alt. name	Function
R0	G0	Hardwired to 0. Stores into it are just ignored.
R1 – R7	G1 – G7	Holds global variables
R8 – R13	O0 – O5	Holds parameters to the procedure being called
R14	SP	Stack pointer
R15	O7	Scratch register
R16 – R23	L0 – L7	Holds local variables for the current procedure
R24 – R29	I0 – I5	Holds incoming parameters
R30	FP	Pointer to the base of the current stack frame
R31	I7	Holds return address for the current procedure

Arithmetic

ADD R1,S2,DST	Add
ADDCC "	Add and set icc
ADDC "	Add with carry
ADDCCC "	Add with carry and set icc
SUB R1,S2,DST	Subtract
SUBCC "	Subtract and set icc
SUBC "	Subtract with carry
SUBCCC "	Subtract with carry and set icc
MULX R1,S2,DST	Multiply
SDIVX R1,S2,DST	Signed divide
UDIVX R1,S2,DST	Unsigned divide
TADCC R1,S2,DST	Tagged add

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

```
towers:                 save %sp, –112, %sp
        cmp N, 1        ! if (n == 1)
        bne Else        ! if (n != 1) goto Else

        sethi %hi(format), Param0 ! printf("Move a disk from %d to %d\n", i, j)
        or Param0, %lo(format), Param0! Param0 = address of format string
        mov I, Param1   ! Param1 = i
        call printf     ! call printf BEFORE parameter 2 (j) is set up
        mov J, Param2   ! use the delay slot after call to set up parameter 2
        b Done          ! we are done now
        nop             ! fill delay slot

Else:   mov 6, K        ! start k = 6 –i – j
        sub K, J, K     ! k = 6 – j
        sub K, I, K     ! k = 6 – i – j

        add N, –1, Scratch  ! start towers(n – 1, i, k)
        mov Scratch, Param0 ! Scratch = N – 1
        mov I, Param1   ! parameter 1 = i
        call towers     ! call towers BEFORE parameter 2 (k) is set up
        mov K, Param2   ! use the delay slot after call to set up parameter 2

        mov 1, Param0   ! start towers(1, i, j)
        mov I, Param1   ! parameter 1 = i
        call towers     ! call towers BEFORE parameter 2 (j) is set up
        mov J, Param2   ! parameter 2 = j

        mov Scratch, Param0 ! start towers(n – 1, k, j)
        mov K, Param1   ! parameter 1 = k
        call towers     ! call towers BEFORE parameter 2 (j) is set up
        mov J, Param2   ! parameter 2 = j

Done:   ret             ! return
        restore         ! use the delay slot after ret to restore windows
```

```
BIPUSH 6        // Else part: compute k = 6 – i – j
ILOAD_1         // local 1 = i; push i
ISUB            // top-of-stack = 6 – i
ILOAD_2         // local 2 = j; push j
ISUB            // top-of-stack = 6 – i – j
ISTORE_3        // local 3 = k = 6 – i – j; stack is n
```

```
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);
   }
}
```

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!)

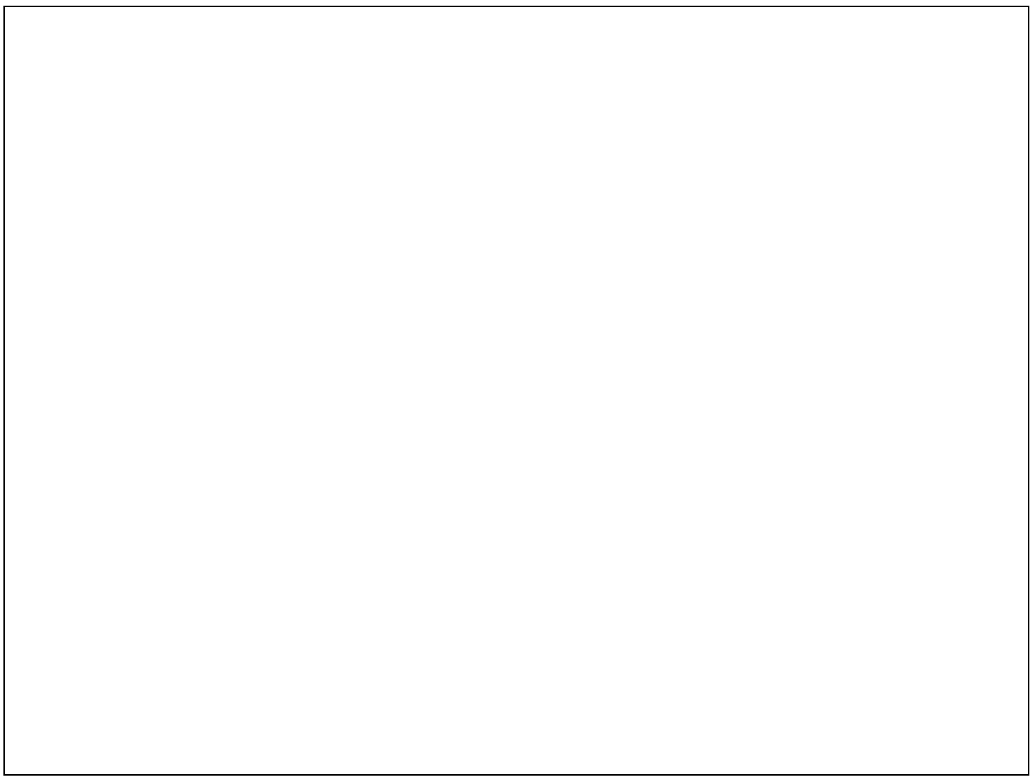
At the else, again, I, J, K are locals, so they are in registers. Just one machine instruction to do each.

(AND, each of these instructions can be executed in ONE microcycle!)

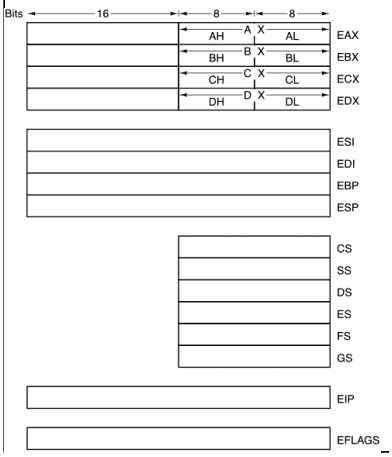
Next weirdness: let's look at the setup for one of the tower recursive calls. Three parameters to pass, but wait, the third parameter is set AFTER the call! Again, one instruction after a branch is already in the pipeline.

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)



Pentium Instruction Set



Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOV DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination

# = shift/rotate count
LV = # locals

Like the UltraSparc, register-oriented rather than stack oriented.

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

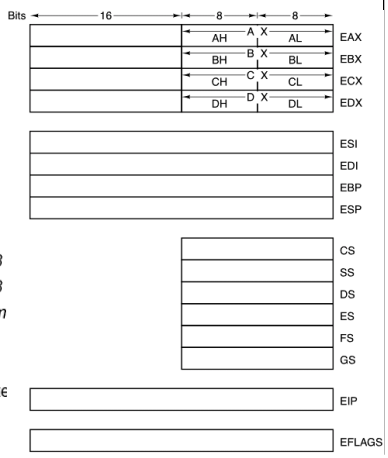
So ADD, DST, SRC means add contents of src to dest and put result back in dst.

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

Opcode	Instruction	Description
14 *ib*	ADC AL,*imm8*	Add with carry *imm8* to AL
15 *iw*	ADC AX,*imm16*	Add with carry *imm16* to AX
15 *id*	ADC EAX,*imm32*	Add with carry *imm32* to EAX
80 /2 *ib*	ADC *r/m8*,*imm8*	Add with carry *imm8* to *r/m8*
81 /2 *iw*	ADC *r/m16*,*imm16*	Add with carry *imm16* to *r/m16*
81 /2 *id*	ADC *r/m32*,*imm32*	Add with CF *imm32* to *r/m32*
83 /2 *ib*	ADC *r/m16*,*imm8*	Add with CF sign-extended *imm8*
83 /2 *ib*	ADC *r/m32*,*imm8*	Add with CF sign-extended *imm8*
10 /r	ADC *r/m8*,*r8*	Add with carry byte register to *r/m*
11 /r	ADC *r/m16*,*r16*	Add with carry *r16* to *r/m16*
11 /r	ADC *r/m32*,*r32*	Add with CF *r32* to *r/m32*
12 /r	ADC *r8*,*r/m8*	Add with carry *r/m8* to byte registe
13 /r	ADC *r16*,*r/m16*	Add with carry *r/m16* to *r16*
13 /r	ADC *r32*,*r/m32*	Add with CF *r/m32* to *r32*



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)

Towers ala Pentium

```
_towers:                PUSH EBP; save EBP (frame pointer)
        MOV EBP, ESP    ; set new frame pointer above ESP
        CMP [EBP+8], 1  ; if (n == 1)
        JNE L1          ; branch if n is not 1
        MOV EAX, [EBP+16]   ; printf(" ...", i, j);
        PUSH EAX        ; note that parameters i, j and the format
        MOV EAX, [EBP+12]   ; string are pushed onto the stack
        PUSH EAX        ; in reverse order. This is the C calling convention
        PUSH OFFSET FLAT:format; offset flat means the address of format
        CALL _printf    ; call printf
        ADD ESP, 12     ; remove params from the stack
        JMP Done        ; we are finished
L1:     MOV EAX, 6      ; start k = 6 – i – j
        SUB EAX, [EBP+12]   ; EAX = 6 – i
        SUB EAX, [EBP+16]   ; EAX = 6 – i – j
        MOV [EBP+20], EAX   ; k = EAX
        PUSH EAX        ; start towers(n – 1, i, k)
        MOV EAX, [EBP+12]   ; EAX = i
        PUSH EAX        ; push i
        MOV EAX, [EBP+8]    ; EAX = n
        DEC EAX         ; EAX = n – 1
        PUSH EAX        ; push n – 1
        CALL _towers    ; call towers(n – 1, i, 6 – i – j)
        ADD ESP, 12     ; remove params from the stack
        MOV EAX, [EBP+16]   ; start towers(1, i, j)
        PUSH EAX        ; push j
        MOV EAX, [EBP+12]   ; EAX = i
        PUSH EAX        ; push i
        PUSH 1          ; push 1
        CALL _towers    ; call towers(1, i, j)
        ADD ESP, 12     ; remove params from the stack
        MOV EAX, [EBP+12]   ; start towers(n – 1, 6 – i – j, i)
        PUSH EAX        ; push i
        MOV EAX, [EBP+20]   ; push 20
        PUSH EAX        ; push k
        MOV EAX, [EBP+8]    ; EAX = n
        DEC EAX         ; EAX = n–1
        PUSH EAX        ; push n – 1
        CALL _towers    ; call towers(n – 1, 6 – i – j, i)
        ADD ESP, 12     ; adjust stack pointer
Done:   LEAVE           ; prepare to exit
        RET 0           ; return to the caller
```

```
BIPUSH 6            // Else part: compute k = 6 – i – j
ILOAD_1             // local 1 = i; push i
ISUB                // top-of-stack = 6 – i
ILOAD_2             // local 2 = j; push j
ISUB                // top-of-stack = 6 – i – j
ISTORE_3            // local 3 = k = 6 – i – j; stack is nc
```

```
add N, –1, Scratch      ! start towers(n – 1, i, k)
mov Scratch, Param0     ! Scratch = N – 1
mov I, Param1           ! parameter 1 = i
call towers             ! call towers BEFORE parameter 2 (k) is set up
mov K, Param2           ! use the delay slot after call to set up parameter 2
```

Like UltraSparc, Pentium takes fewer instructions than IJVM to compute K. But notice Kisin memory rather than in a register as it 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 wil 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?

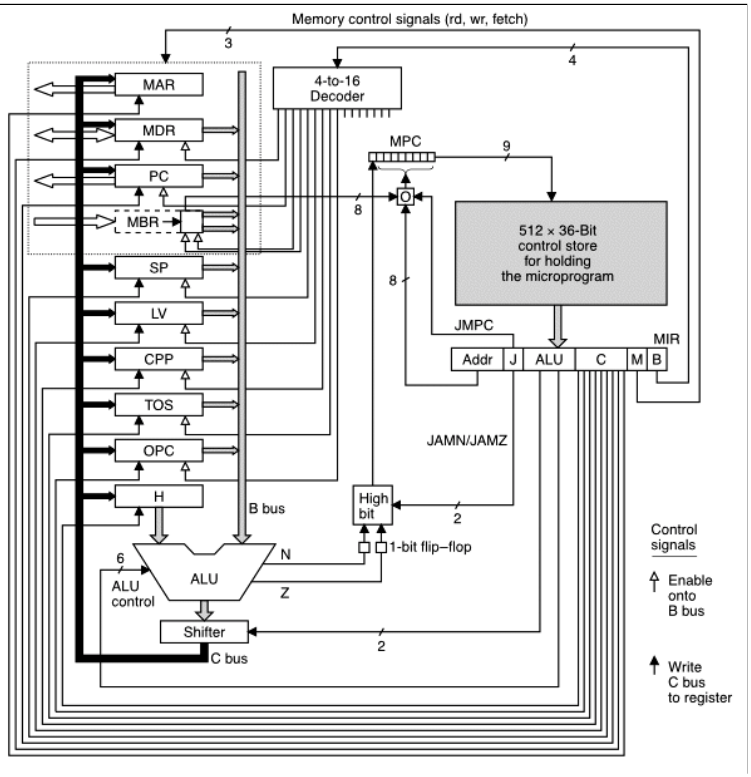
If Sparc can do same thing in 20 inst it takes 33 to do on Pentium, how do you compare “MIPS”?

(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 and Interrupts



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 SLOOOW.

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 shold 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.