CS352H: Computer Systems Architecture

Lecture 3: Instruction Set Architectures II

September 3, 2009

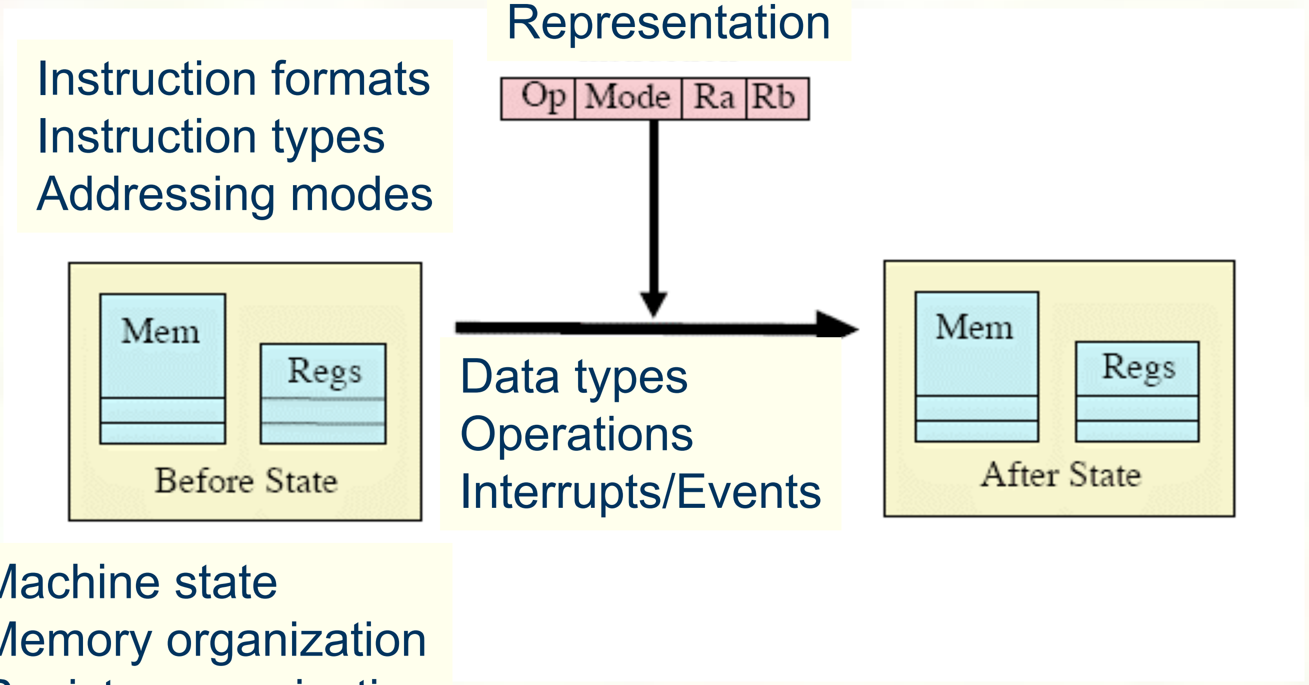
ISA is a Contract

- Between the programmer and the hardware:
  - Defines visible state of the system
  - Defines how the state changes in response to instructions
- Programmer obtains a model of how programs will execute
- Hardware designer obtains a formal definition of the correct way to execute instructions
- ISA Specification:
  - Instruction set
  - How instructions modify the state of the machine
  - Binary representation
- Today:
  - ISA principles
  - ISA evolution

ISA Specification

Instruction
Representation

Instruction formats
Instruction types
Addressing modes



Data types
Operations
Interrupts/Events

Machine state
Memory organization
Register organization

Architecture vs. Implementation

- Architecture defines what a computer system does in response to an instruction and data
- Architectural components are visible to the programmer

- Implementation defines how a computer system does it
  - Sequence of steps
  - Time (cycles)
  - Bookkeeping functions

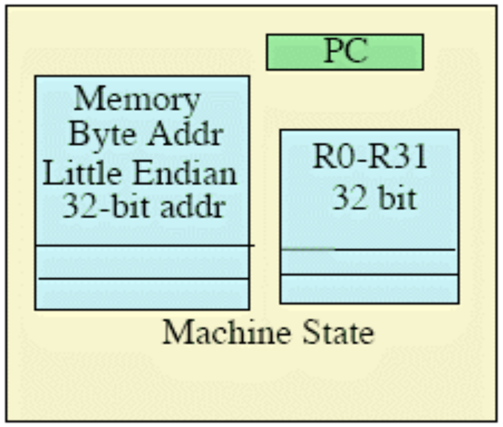
Architecture or Implementation?

- Number of GP registers
- Width of memory word
- Width of memory bus
- Binary representation of:

  add r3, r3, r9
- # of cycles to execute a FP instruction
- Size of the instruction cache
- How condition codes are set on an ALU overflow

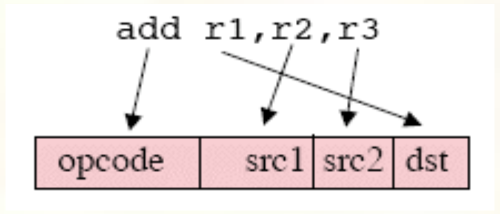
Machine State

- Registers (size & type)
  - PC
  - Accumulators
  - Index
  - General purpose
  - Control
- Memory
  - Visible hierarchy (if any)
  - Addressability
    - Bit, byte, word
    - Endian-ness
    - Maximum size
  - Protection



Components of Instructions

- Operations (opcodes)
- Number of operands
- Operand specifiers (names)
  - Can be implicit



- Instruction classes
  - ALU
  - Branch
  - Memory
  - …
- Instruction encodings

Number of Operands

None	halt nop	
One	not R4	R4 ← ~R4
Two	add R1, R2	R1 ← R1+R2
Three	add R1, R2, R3	R1 ← R2+R3
> three	madd R4,R1,R2,R3	R4 ← R1+(R2*R3)

Effect of Number of Operands

- Given

  E = (C + D) * (C – D)

- And C, D and E in R1, R2 and R3 (resp.)

3 operand machine

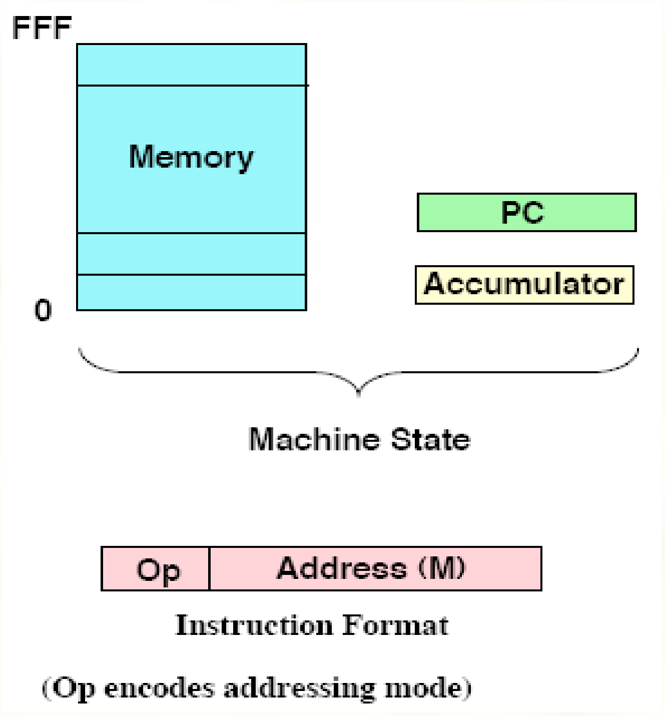
```
add     R3, R1, R2
sub     R4, R1, R2
mult    R3, R4, R3
```

2 operand machine

```
mov     R3, R1
add     R3, R2
sub     R2, R1
mult    R3, R2
```

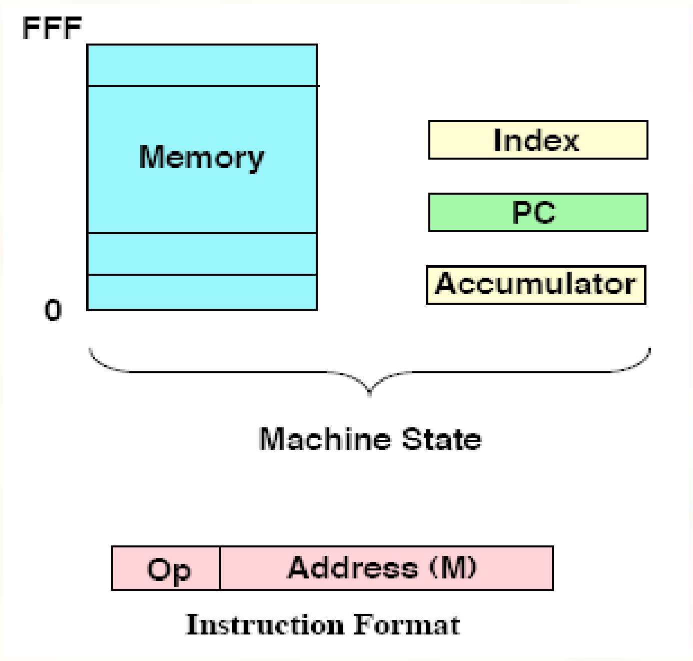
Evolution of Register Organization

- In the beginning…
  - The accumulator
- Two instruction types: op & store
  - A ← A op M
  - A ← A op *M
  - *M ← A
- One address architecture
  - One memory address per instruction
- Two addressing modes:
  - Immediate: M
  - Direct: *M
- Inspired by "tabulating" machines



The Index Register

- Add indexed addressing mode
  - A ← A op (M+I)
  - A ← A op *(M+I)
  - *(M+I) ← A
- Useful for array processing
  - Addr. of X[0] in instruction
  - Index value in index register
- One register per function:
  - PC: instructions
  - I: data addresses
  - A: data values
- Need new instruction to use I
  - inc I
  - cmp I



Example of Effect of Index Register

```
Sum = 0;
for (i=0; i<n; i++) sum = sum + y[i];
```

Without Index Register

```
Start:    CLR     i
          CLR     sum
Loop:     LOAD    IX
          AND     #MASK
          OR      i
          STORE   IX
          LOAD    sum
IX:       ADD     y
          STORE   sum
          LOAD    i
          ADD     #1
          STORE   i
          CMP     n
          BNE     Loop
```

With Index Register

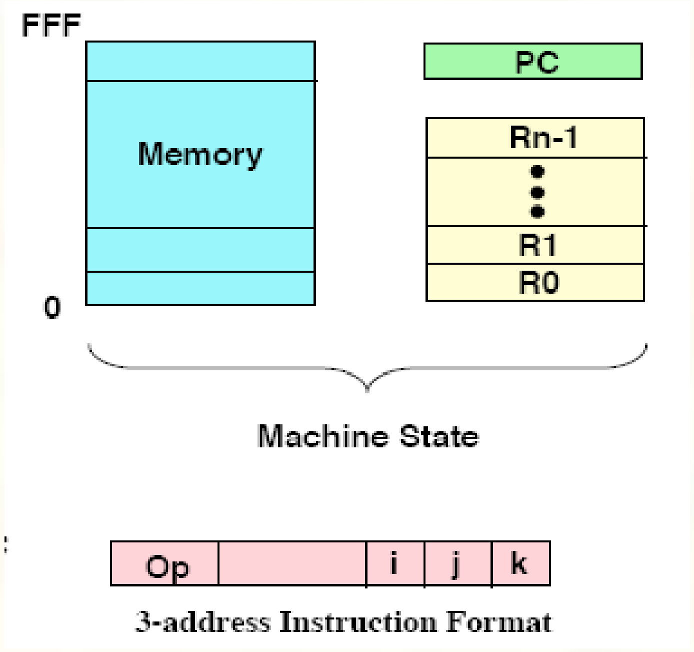
```
Start:    CLRA
          CLRX
Loop:     ADDA    y(X)
          INCX
          CMPX    n
          BNE     Loop
```

<u>But what about…</u>

```
Sum = 0;
for (i=0; i<n; i++)
        for (j=0; j<n; j++)
                sum = sum + x[j] * y[i];
```

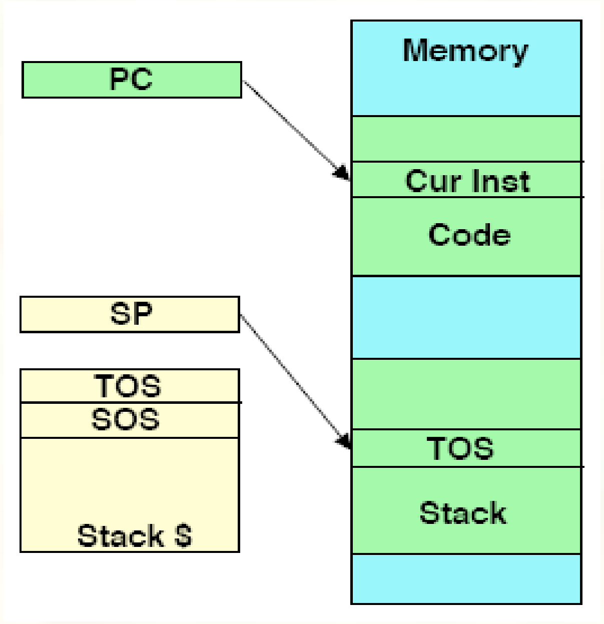
1964: General-Purpose Registers

- Merge accumulators (data) & index registers (addresses)
  - Simpler
  - More orthogonal (opcode independent of register)
  - More fast local storage
  - But addresses and data must be the same size
- How many registers?
  - More: fewer loads
  - But more instruction bits
- IBM 360



Stack Machines

- Register state: PC & SP
- All instructions performed on TOS & SOS
- Implied stack Push & Pop
  - TOS ← TOS op SOS
  - TOS ← TOS op M
  - TOS ← TOS op *M
- Many instructions are zero address!
- Stack cache for performance
  - Like a register file
  - Managed by hardware
- Pioneered by Burroughs in early 60’s
- Renaissance due to JVM



Register-Based ISAs

- Why do register-based architectures dominate the market?
  - Registers are faster than memory
  - Can "cache" variables
    - Reduces memory traffic
    - Improves code density
  - More efficient use by compiler than other internal storage (stack)
    - (A*B) – (B*C) – (A*D)
- What happened to Register-Memory architectures?
  - More difficult for compiler
  - Register-Register architectures more amenable to fast implementation
- General- versus special-purpose registers?
  - Special-purpose examples in MIPS: PC, Hi, Lo
  - Compiler wants an egalitarian register society

Stack Code Examples

A = B + C * D;
E = A + F[J] + C;

```
push    D
push    C
mul
push    B
add
push    J
pushx   F
push    C
add
add
pop     E
```

Pure stack
(zero addresses)
11 instr, 7 addr

```
push    D
mul     C
add     B
push    J
pushx   F
add     C
add
pop     E
```

One address stack

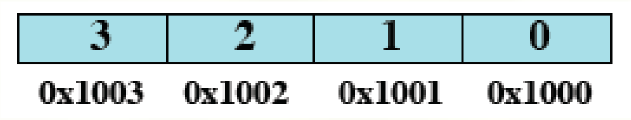
8 instr, 7 addr

```
load    R1, D
load    R2, C
mul     R3, R2, R1
load    R4, B
add     R5, R4, R3
load    R6, J
load    R7, F(R6)
add     R8, R7, R2
add     R9, R5, R8
store   R9, E
```

Load/Store Arch
(Several GP registers)
10 instr, 6 addr

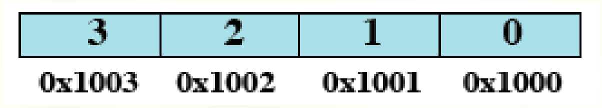
Memory Organization

- ISA specifies five aspects of memory:
  - Smallest addressable unit
  - Maximum addressable units of memory
  - Alignment
  - Endian-ness
  - Address modes

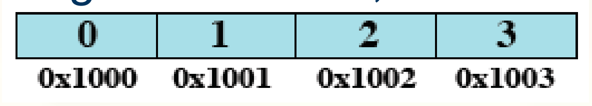


Bytes: any address
Half words: even addresses
Words: Multiples of 4

Little Endian: Intel, DEC



Big Endian: IBM, Motorola



Today: Configurable

Addressing Modes are Driven by Program Usage

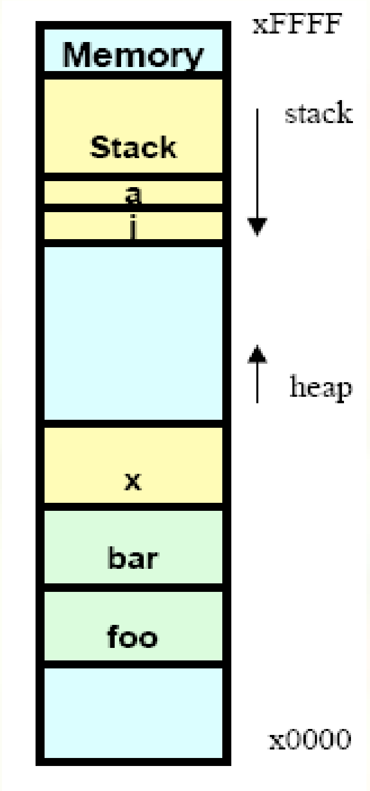
argument

procedure

```
double x[100];                  // global
void foo(int a) {               // argument
  int j;                        // local
  for (j=0; j<10; j++)
    x[j] = 3 + a*x[j-1];
  bar(a);
}
```

array reference

constant



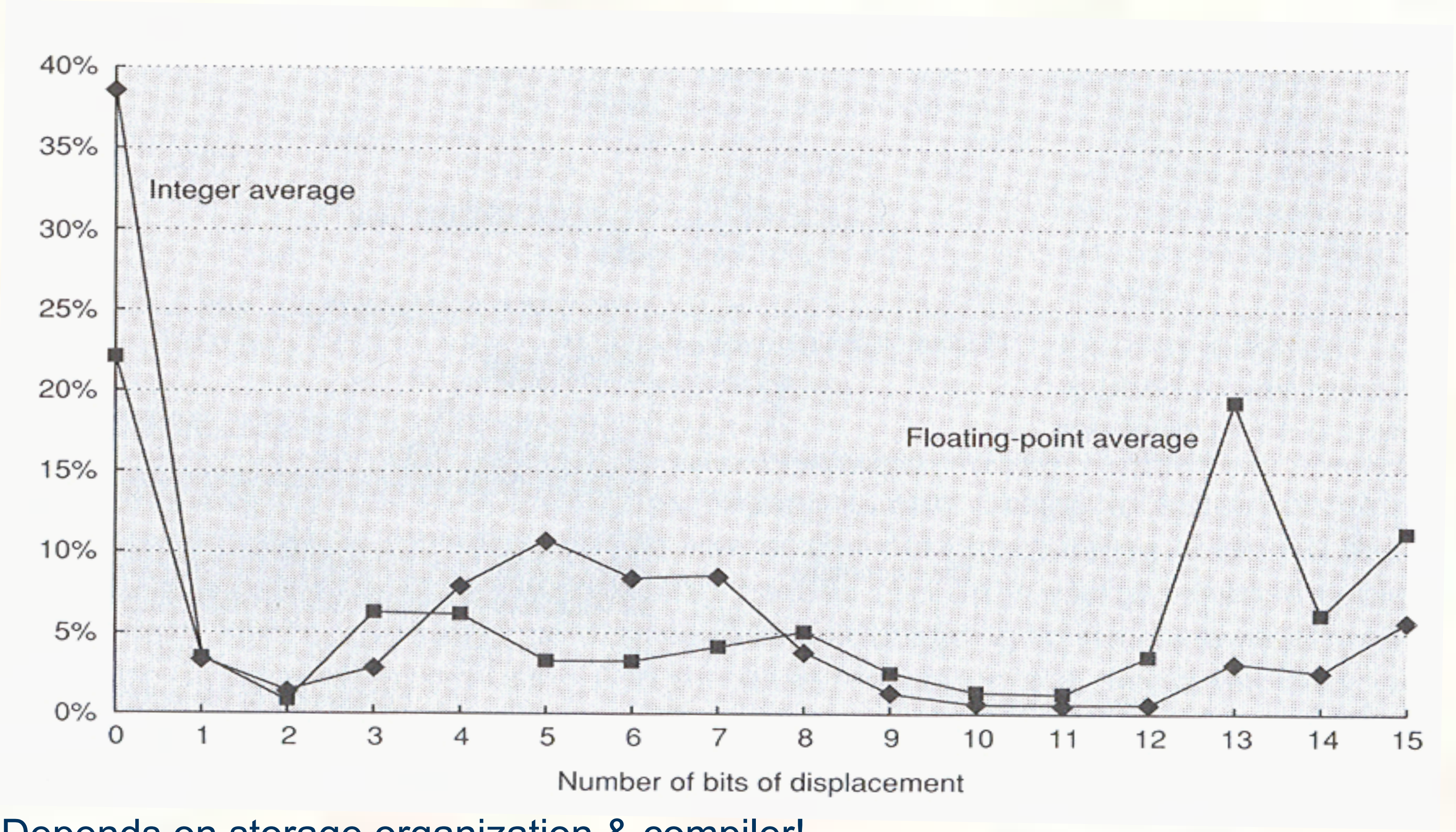
Addressing Mode Types

#n	immediate
(0x1000)	absolute (aka direct)
Rn	register
(Rn)	register indirect
-(Rn)	predecrement
(Rn)+	postincrement
*(Rn)	memory indirect
*(Rn)+	postincrement indirect
d(Rn)	displacement
d(Rn)[Rx]	scaled

Why Only three Addressing Modes in MIPS?

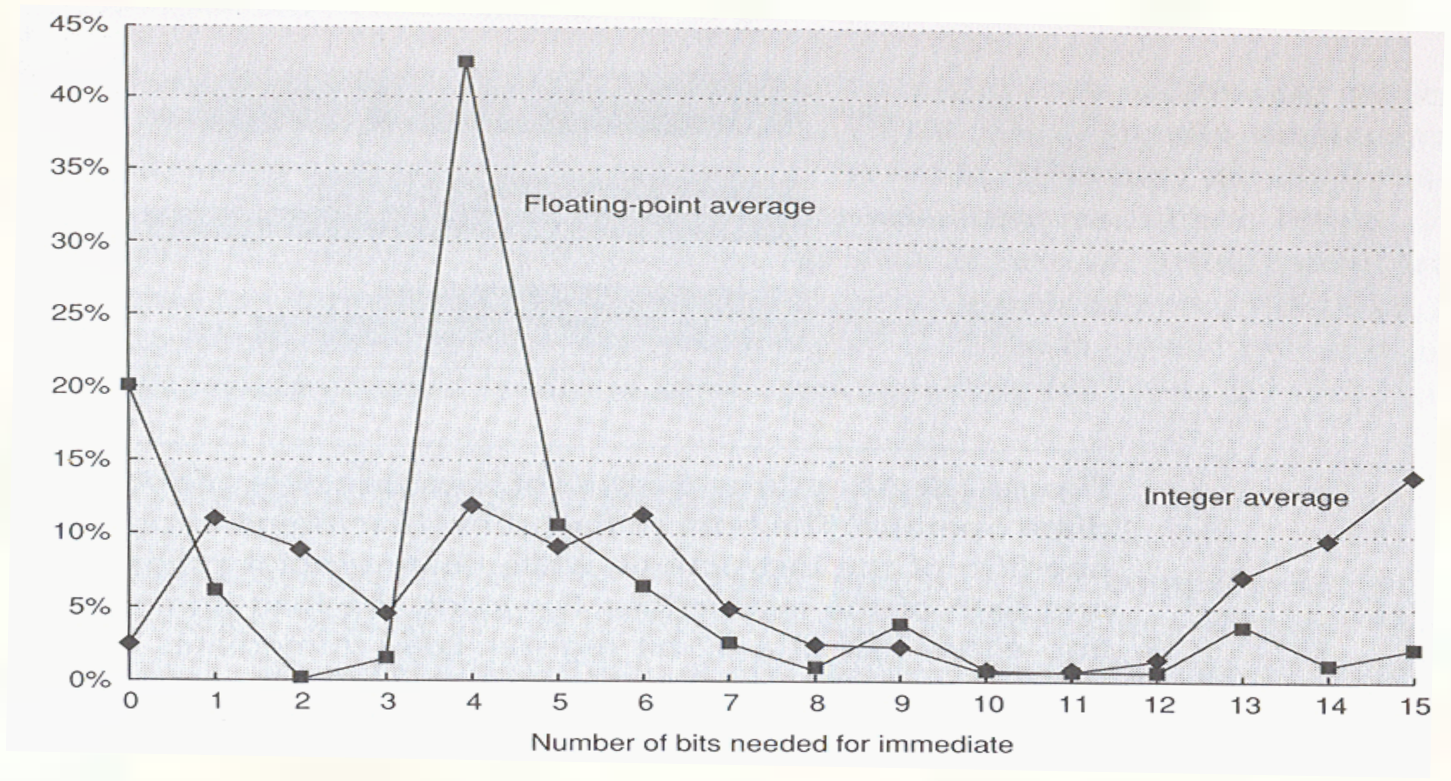
- Studies of code generated for GP computers:
  - Register mode: ~50%
  - Immediate + Displacement: 35% - 40%
  - The Vax had 27 addressing modes!
- But special-purpose ISAs make more extensive use of other modes
  - Auto-increment in DSP processing

How Many Bits for Displacement?



Depends on storage organization & compiler!
DEC Alpha data

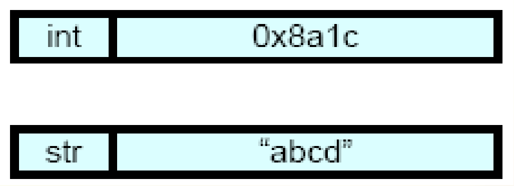
How Many Bits for Immediates?



- Same DEC Alpha study as displacement data
- A study of the Vax (with support for 32-bit immediates) showed that 20% to 25% of immediate values required more than 16 bits

Data Types

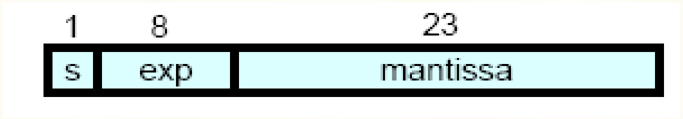
- How the contents of memory & registers are interpreted
- Can be identified by
  - Tag
  - Use
- Driven by application:
  - Signal processing: 16-bit fixed point (fractions)
  - Text processing: 8-bit characters
  - Scientific processing: 64-bit floating point
- GP computers:
  - 8, 16, 32, 64-bit
  - Signed & unsigned
  - Fixed & floating



Symbolics tags

Example: 32-bit Floating Point

- Specifies mapping from bits to real numbers
- Format
  - Sign bit (S)
  - 8-bit exponent (E)
  - 23- bit mantissa (M)



- Interpretation
  - Value = $(-1)^S * 2^{(E-127)} * 1.M$
- Operations:
  - Add, sub, mult, div, sqrt

- "Integer" operations can also have fractions
  - Assume the binary point is just to the right of the leftmost bit
  - 0100 1000 0000 1000 = $2^{-1} + 2^{-4} + 2^{-12} = 0.56274$

Instruction Types

- ALU
  - Arithmetic `(add, sub, mult, …)`
  - Logical `(and, or, srl, …)`
  - Data type conversions `(cvtf2i, …)`
  - Fused memory/arithmetic
- Data movement
  - Memory reference `(lw, sb, …)`
  - Register to register `(movi2fp, …)`
- Control
  - Test/compare `(slt, …)`
  - Branch, jump `(beq, j, jr, …)`
  - Procedure call `(jal, …)`
  - OS entry `(trap)`
- Complex
  - String compare, procedure call (with save/restore), …

Control Instructions

- Implicit
  - PC ← PC + 4
- Unconditional jumps
  - PC ← X (direct)
  - PC ← PC + X (PC relative)
    - X can be a constant or a register
- Conditional jumps (branches): > 75% of control instr.
  - PC ← PC + ((cond) ? X : 4)
- Procedure call/return
- Predicated instructions
- Conditions
  - Flags
  - In a register
  - Fused compare and branch

Methods for Conditional Jumps

- Condition codes
  - Tests special bits set by ALU
    - Sometimes this is done for free
    - CC is extra state constraining instruction order
  - X86, ARM, PowerPC
- Condition register
  - Tests arbitrary register for result of comparison
    - Simple
    - But uses up a register
  - Alpha, MIPS
- Fused compare and branch
  - Comparison is part of the branch
    - Single instruction
    - Complicates pipelining
  - PA-RISC, VAX, MIPS

Long Branches

- beq $7, $8, Label
- What if Label is "far away"?
  - PC-relative address cannot be encoded in 16 bits

- Transform to:

```
                    bne $7, $8, NearbyLabel
                    j FarAwayLabel
NearbyLabel:
```

Predication

- Branches introduce discontinuities
- If (condition) then

  ```
      this
  else
      that
  ```

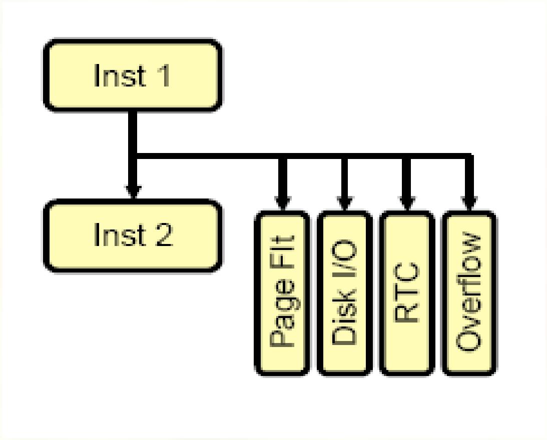
- Might translate into

  ```
          R11 ← (condition)
          beq       R11, R0, L1
          this
          j         L2
  L1:     that
  L2:
  ```

- Forced to wait for "beq"
- With predication both *this* and *that* are evaluated but only the results of the "correct" path are kept
- (condition) *this*
  (not condition) *that*
- Need
  - Predicated instructions
  - Predicate registers
  - Compiler
- IA-64
  - 64 1-bit predicate registers
  - Instructions include extra bits for predicates

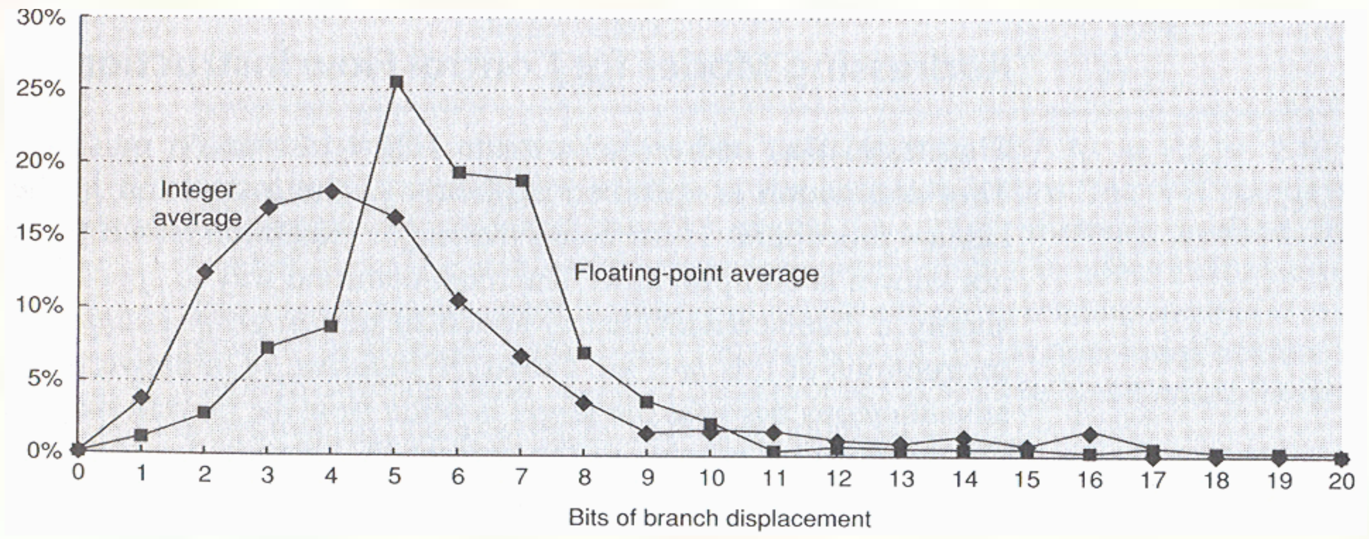
Exceptions/Events

- Implied multi-way branch after every instruction
  - External events (interrupts)
    - I/O completion
  - Internal events
    - Arithmetic overflow
    - Page fault
- What happens?
  - EPC ← PC of instruction causing fault
  - PC ← HW table lookup (based on fault)
  - Return to EPC + 4 (sort of)
- What about complex "lengthy" instructions?



Control Instructions: Miscellaneous

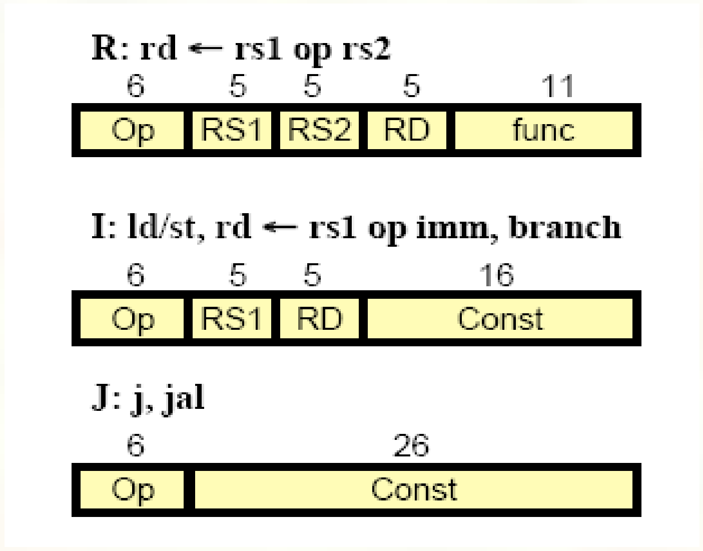
- How many bits for the branch displacement?



- Procedure call/return
  - Should saving and restoring of registers be done automatically?
  - Vax callp instruction

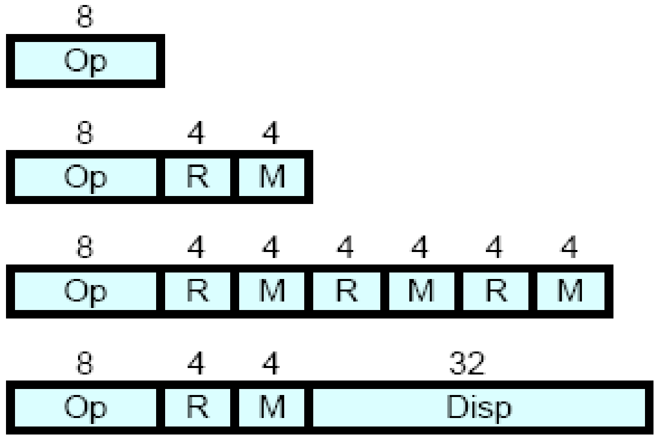
Instruction Formats

- Need to specify all kinds of information
  - R3 $\leftarrow$ R1 + R2
  - Jump to address
  - Return from call
- Frequency varies
  - Instructions
  - Operand types
- Possible encodings:
  - Fixed length
  - Few lengths
  - Byte/bit variable



Variable-Length Instructions

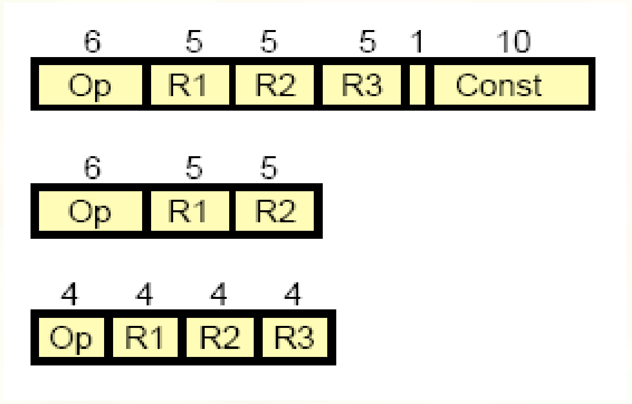
- More efficient encodings
  - No unused fields/operands
  - Can use frequencies when determining opcode, operand & address mode encodings
- Examples
  - VAX
  - Intel x86 (byte variable)
  - Intel 432 (bit variable)
- At a cost of complicating fast implementation
  - Where is the next instruction?
  - Sequential operand location determination



VAX instrs: 1-53 bytes!

Compromise: A Couple of Lengths

- Better code density than fixed length
  - An issue for embedded processors
- Simpler to decode than variable-length
- Examples:
  - ARM Thumb
  - MIPS 16
- Another approach
  - On-the-fly instruction decompression (IBM CodePack)



Next Lecture

- Finish ISA Principles
- A brief look at the IA-32 ISA
- RISC vs. CISC
- The MIPS ALU
- Hwk #1 due