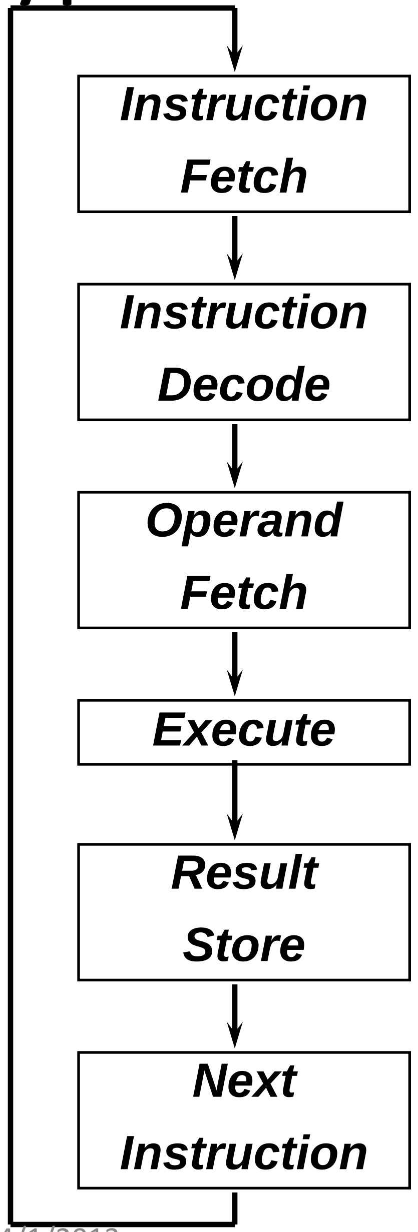
Typical Processor Execution Cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in register or storage for later use

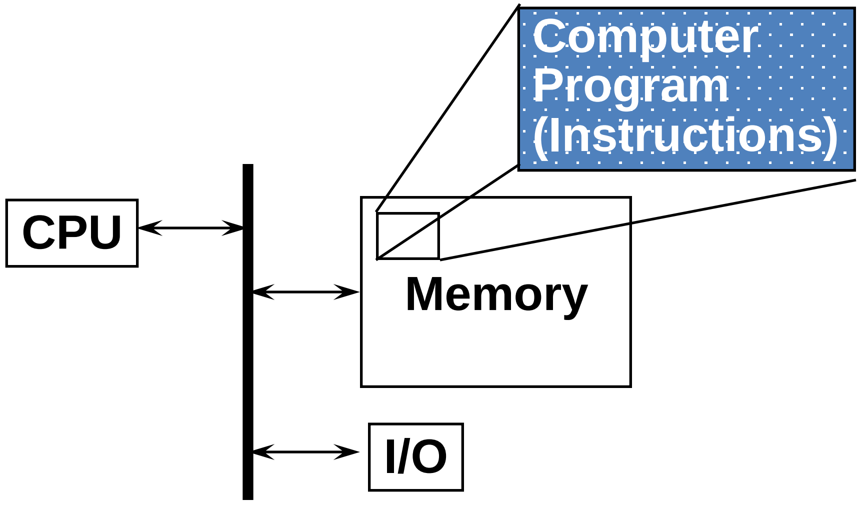
Determine successor instruction

Instruction and Data Memory

Programmer's View

ADD	**01010**
SUBTRACT	**01110**
AND	**10011**
OR	**10001**
COMPARE	**11010**
.	**.**
.	**.**
.	**.**

Computer's View



Princeton (Von Neumann) Architecture

--- Data and Instructions mixed in same unified memory

--- Program as data

--- Storage utilization

--- Single memory interface

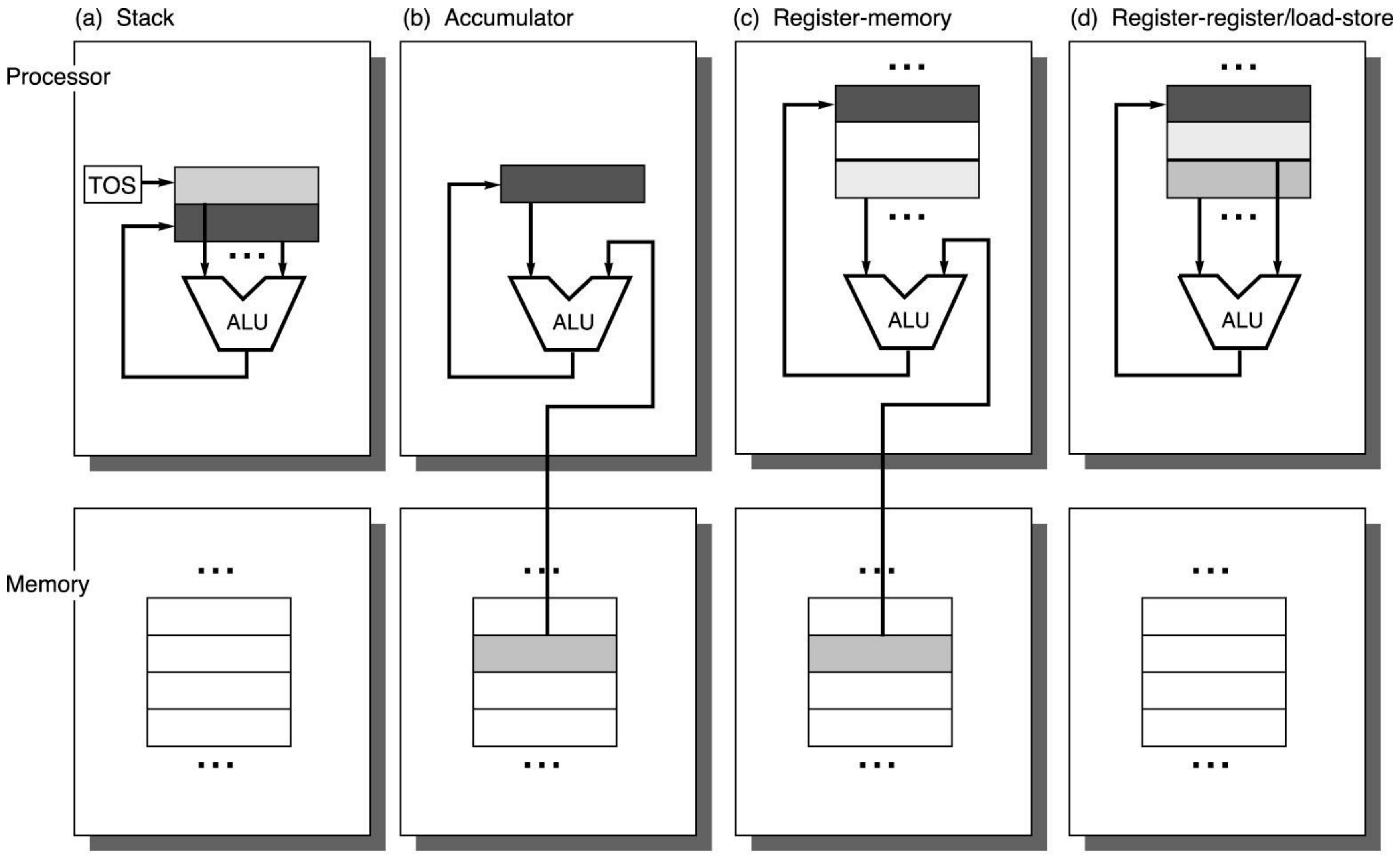
Harvard Architecture

--- Data & Instructions in separate memories

--- Has advantages in certain high performance implementations

--- Can optimize each memory

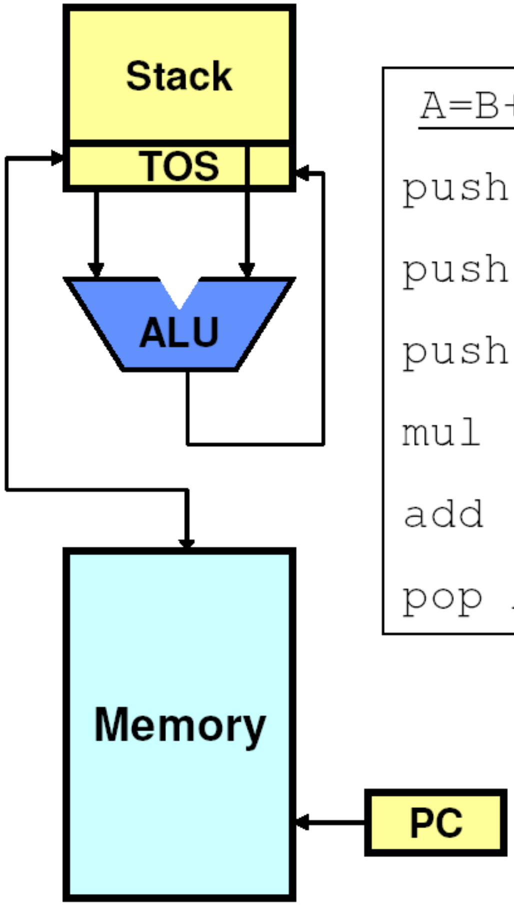
Basic Addressing Classes



© 2003 Elsevier Science (USA). All rights reserved.

Stack Architectures

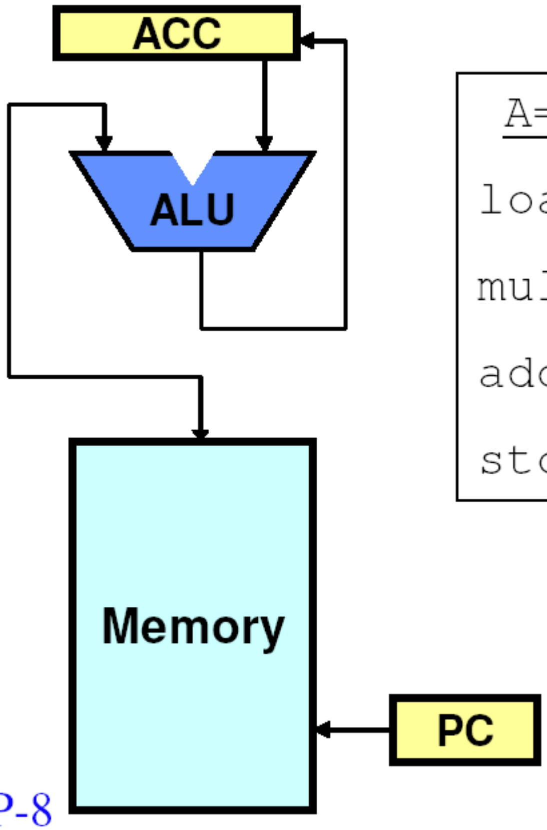
- Stack: First-In Last-Out data structure (FILO)
- Instruction operands
  - None for ALU operations
  - One for push/pop
- Advantages:
  - Short instructions
  - Compiler is easy to write
- Disadvantages
  - Code is inefficient
    - Fix: random access to stacked values
  - Stack size & access latency
    - Fix : register file or cache for top entries
- Examples
  - 60s: Burroughs B5500/6500, HP 3000/70
  - Today: Java VM



```
A=B+(C*D)
push B
push C
push D
mul
add
pop A
```

Accumulator Architectures

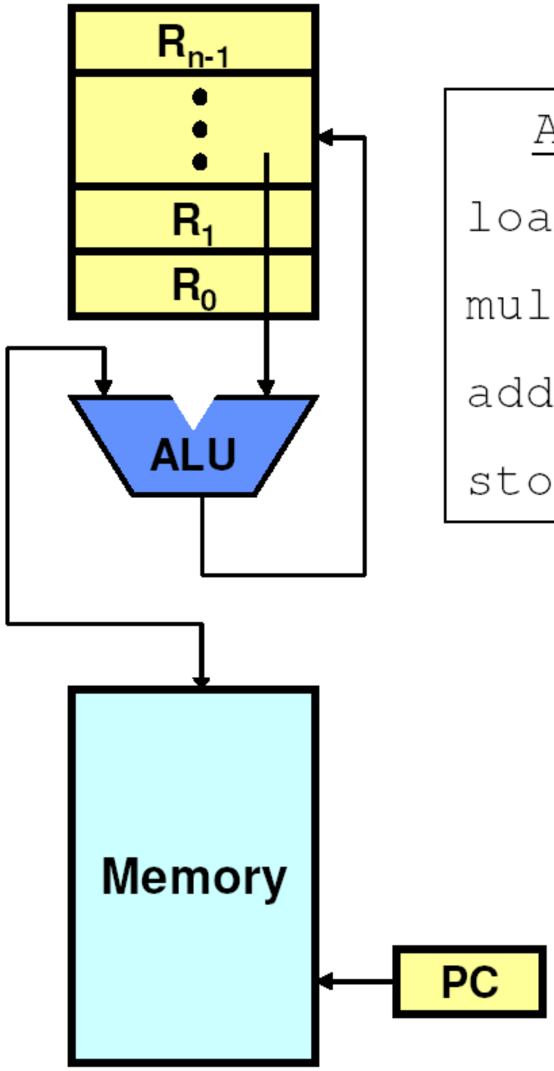
- Single register (accumulator)
- Instructions
  - ALU (Acc ← Acc + *M)
  - Load to accumulator (Acc ← *M)
  - Store from accumulator (*M ← Acc)
- Instruction operands
  - One explicit (memory address)
  - One implicit (accumulator)
- Attributes:
  - Short instructions
  - Minimal internal state; simple design
  - Many loads and stores
- Examples:
  - Early machines: IBM 7090, DEC PDP-8
  - Today: DSP architectures



```
A=B+(C*D)
load C
mul D
add B
store A
```

Register-to-Memory Architectures

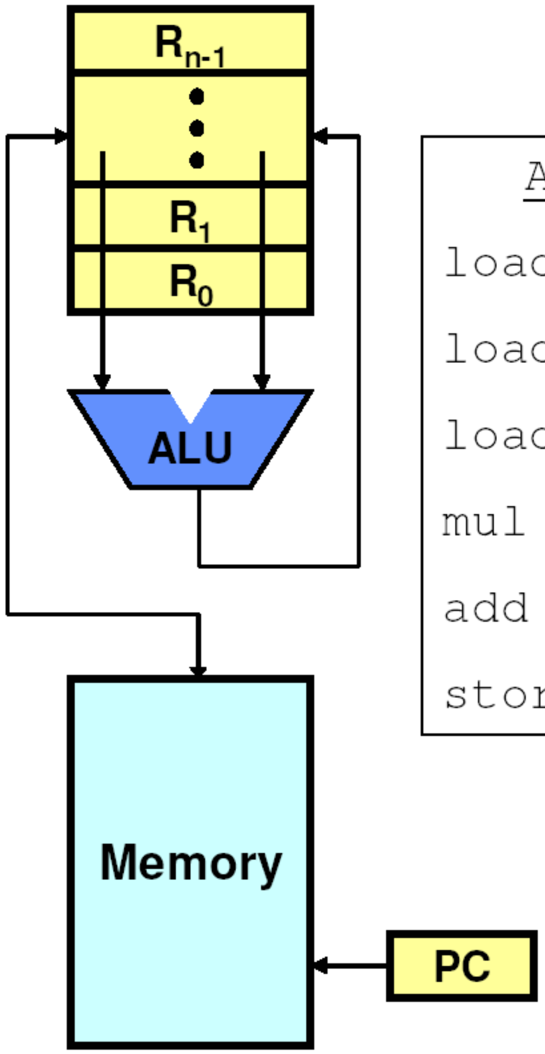
- One memory address in ALU ops
- Typically 2-operand ALU ops
- Advantages
  - Small instruction count
  - Dense encoding
- Disadvantages
  - Result destroys an operand
  - Instruction length varies
  - Clocks per instruction varies
  - Harder to pipeline
- Examples
  - IBM 360/370, VAX



```
A=B+(C*D)

load R1<-C

mul  R1<-R1*D

add  R1<-R1+B

store A<-R1
```

Register-to-Register: Load-Store Architectures

- No memory addresses in ALU ops
- Typically 3-operand ALU ops
  - Bigger encoding, but simplifies register allocation
- Advantages
  - Simple fixed-length instructions
  - Easily pipelined
- Disadvantages
  - Higher instruction count
- Examples
  - CDC6600, CRAY-1, most RISCs



```
A=B+(C*D)
load R1<-C
load R2<-D
load R3<-B
mul  R4<-R1*R2
add  R5<-R4+R3
store A<-R5
```

Memory-to-Memory Architectures

- All ALU operands from memory addresses
- Advantages
  - No register wastage
  - Lowest instruction count
- Disadvantages
  - Large variation in instruction length
  - Large variation in clocks per instructions
  - Huge memory traffic
- Examples
  - VAX

```
   D=B+(C*D)
mul D <- C*D
add D <- D+B
```

Comparing Number of Instructions

Code sequence for (C = A + B) for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

$$ExecutionTime = \frac{1}{Performance} = Instructions \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

General Purpose Registers Dominate

- Advantages of registers
  - Registers are faster than memory
  - Registers compiler technology has evolved to efficiently generate code for register files
    - E.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Registers can hold variables
    - Memory traffic is reduced, so program is sped up (since registers are faster than memory)
  - Code density improves (since register named with fewer bits than memory location)
  - Registers imply operand locality

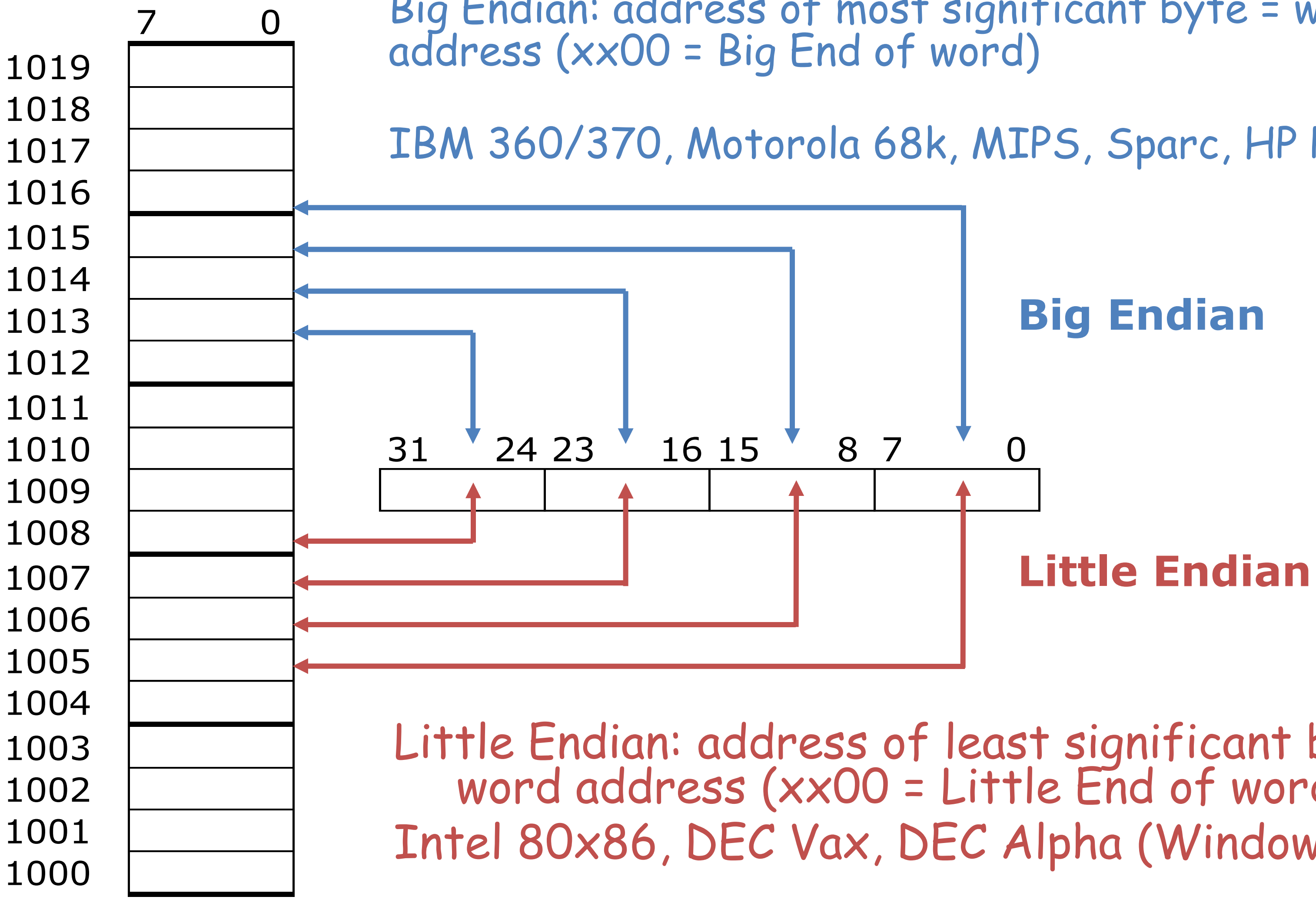
Typical Operations (since 1960)

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate
Graphics (MMX)	parallel subword ops (4 16bit add)

Memory Addressing: Endianess

Big Endian: address of most significant byte = word address (xx00 = Big End of word)

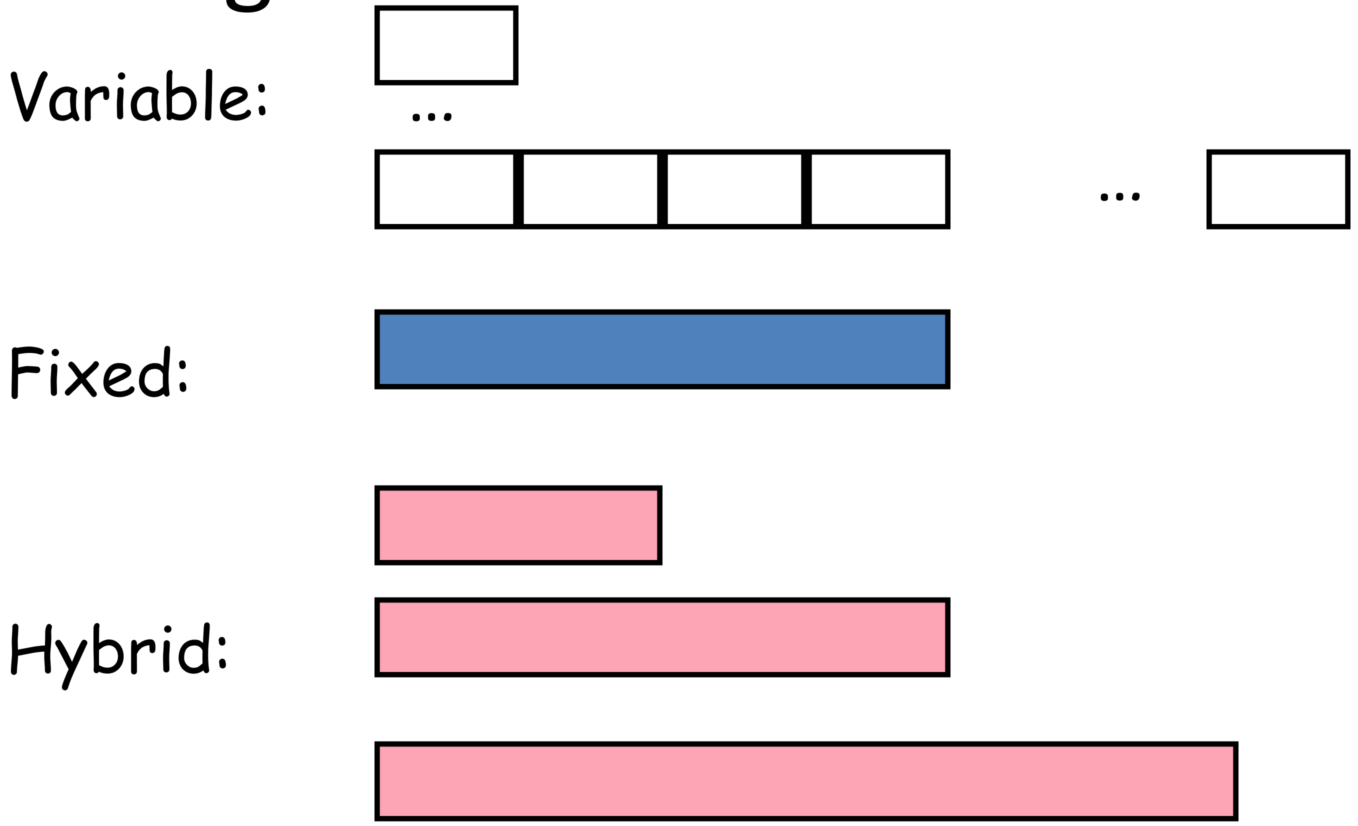
IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA



Little Endian: address of least significant byte = word address (xx00 = Little End of word)

Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

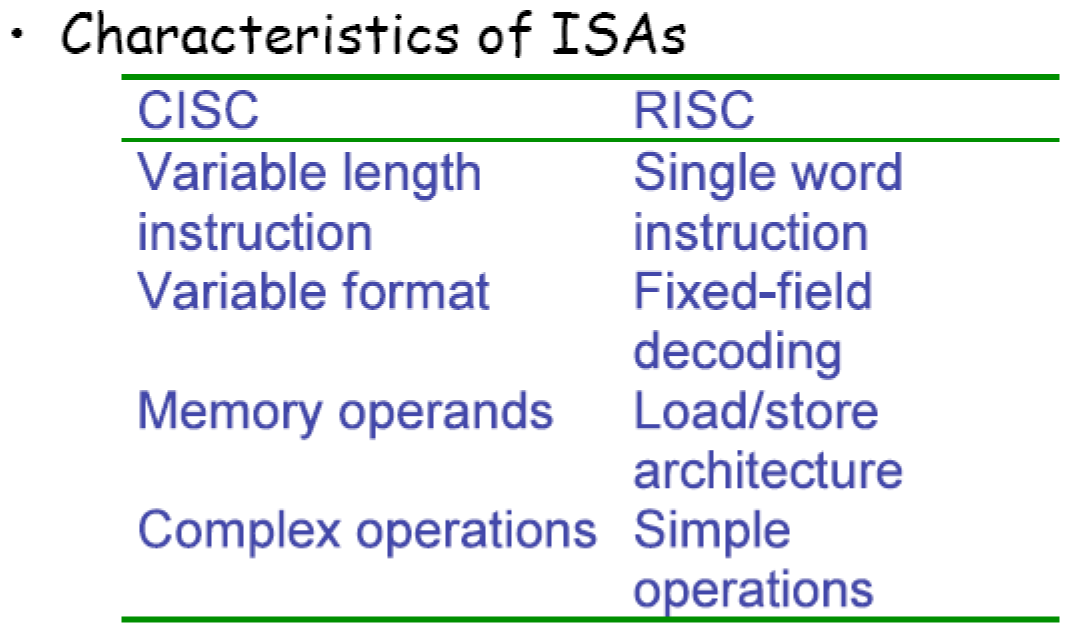
Encoding



- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions
- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density
- Some architectures actually exploring on-the-fly decompression for more density.

RISC vs. CISC

- CISC (complex instruction set computer)
  - VAX, Intel X86, IBM 360/370, etc.
- RISC (reduced instruction set computer)
  - MIPS, DEC Alpha, SUN Sparc, IBM 801



- Characteristics of ISAs

CISC	RISC
Variable length instruction	Single word instruction
Variable format	Fixed-field decoding
Memory operands	Load/store architecture
Complex operations	Simple operations

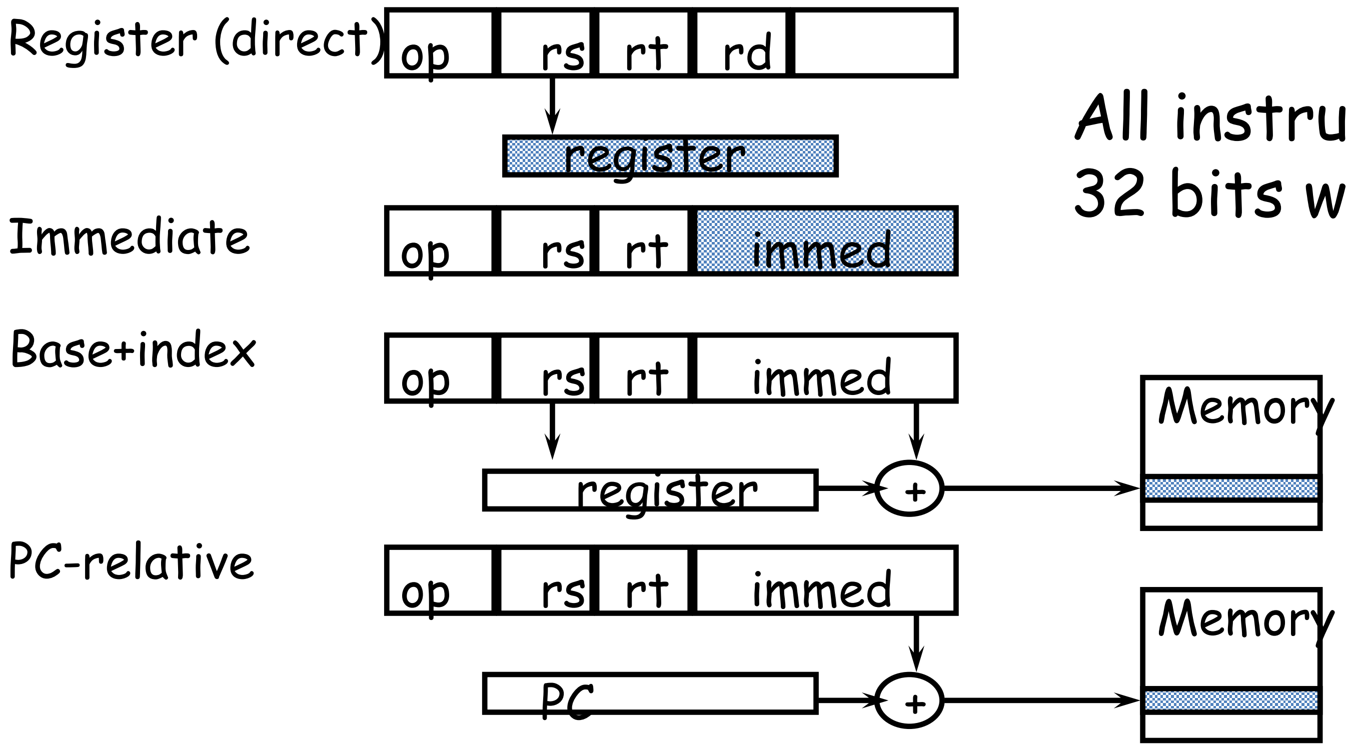
RISC – CISC Instruction Set Design

- The historical background:
  - In first 25 years (1945-70) performance came from both technology and design.
  - Design considerations:
    - small and slow memories: compact programs are fast.
    - small no. of registers: memory operands.
    - attempts to bridge the semantic gap: model high level language features in instructions.
    - no need for portability: same vendor application, OS and hardware.
    - backward compatibility: every new ISA must carry the good and bad of all past ones.

Result: powerful and complex instructions that are rarely used.

MIPS Instruction Formats

MIPS (originally an acronym for **Microprocessor without Interlocked Pipeline Stages**) is a reduced instruction set computer (RISC) instruction set architecture(ISA) developed by MIPS Computer Systems (now MIPS Technologies).



All instructions 32 bits wide

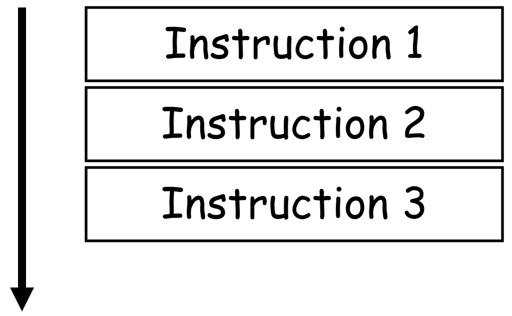
Instruction Set Design Metrics

- Static Metrics
  - How many bytes does the program occupy in memory?
- Dynamic Metrics
  - How many instructions are executed?
  - How many bytes does the processor fetch to execute the program?
  - How many clocks are required per instruction?

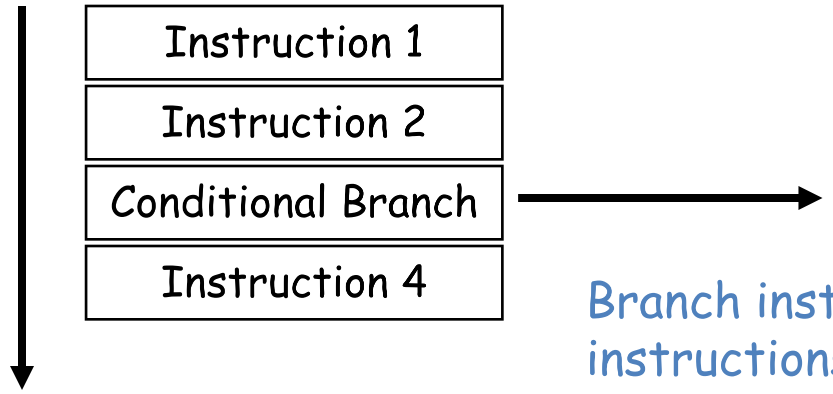
$$ExecutionTime = \frac{1}{Performance} = Instructions \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

Instruction Sequencing

- The next instruction to be executed is typically implied
  - Instructions execute sequentially
  - Instruction sequencing increments a Program Counter



- Sequencing flow is disrupted conditionally and unconditionally
  - The ability of computers to test results and conditionally instructions is one of the reasons computers have become so useful



Branch instructions are ~20% of all instructions executed