System I

Instruction Set Architecture

Haifeng Liu

Zhejiang University

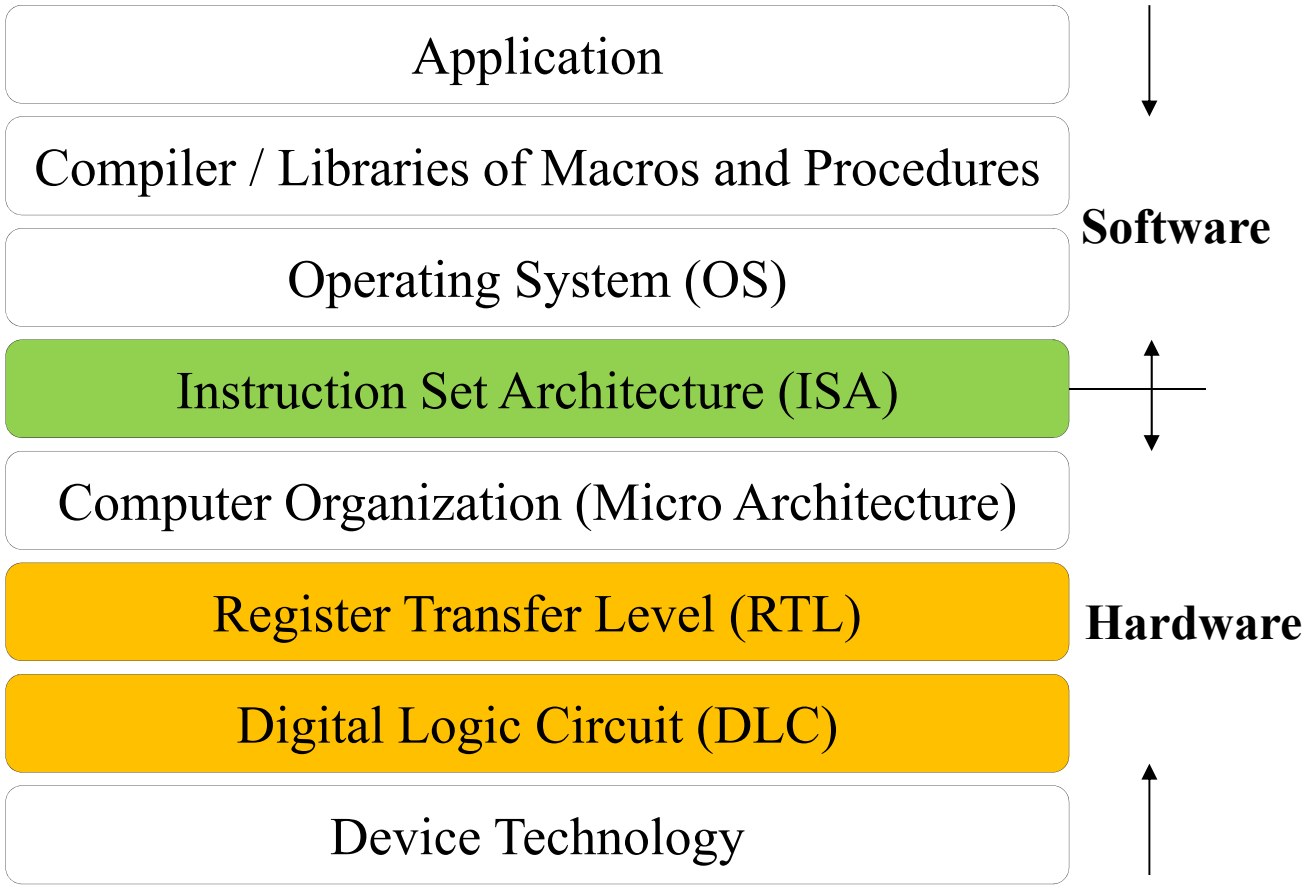
Overview

- Introduction to ISA
- About ISA design

Overview

- Introduction to ISA
- About ISA design

Instruction Set Architecture (ISA)



What is An Instruction?

- Instruction = opcode + operands
- Elements of an instruction
  - Operation code (Opcode)
    - Do this
  - Source Operand reference(s)
    - To this
  - Result Operand reference(s)
    - Put the answer here
  - Next Instruction Reference
    - When you are done, do this instruction next

Where Are the Operands?

- Main memory
- CPU register
- I/O device
- In instruction itself

- To specify which register, which memory location, or which I/O device, we'll need some addressing scheme for each

Instruction Formats

- Instruction length
  - Whether short, long, or variable
- Number of operands
  - 3/2/1/0
- Number of addressable registers
- Memory organization
  - Whether byte- or word addressable
- Addressing modes
  - Choose any or all: direct, indirect or indexed

Instruction Set

- The complete collection of instructions that are understood by a CPU

- The instruction set is ultimately represented in binary **machine code** also referred to as **object code**
  - Usually represented by assembly code to human programmer

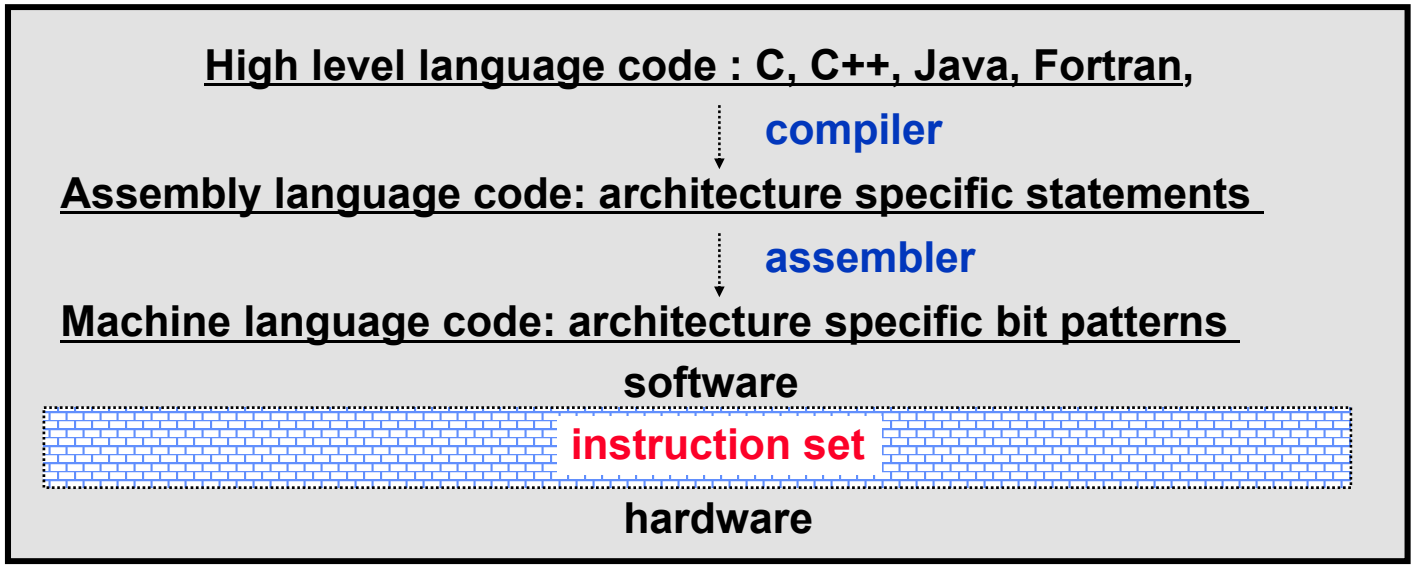
Instruction Set Architecture (ISA)

- The physical hardware that is controlled by the instructions is referred to as the *Instruction Set Architecture* (ISA)
  - One can think of an ISA as a hardware functionality of a given computer.
  - ISA refers to the actual programmer visible machine interface such as instruction set, registers, memory organization and exception (i.e., interrupt) handling.

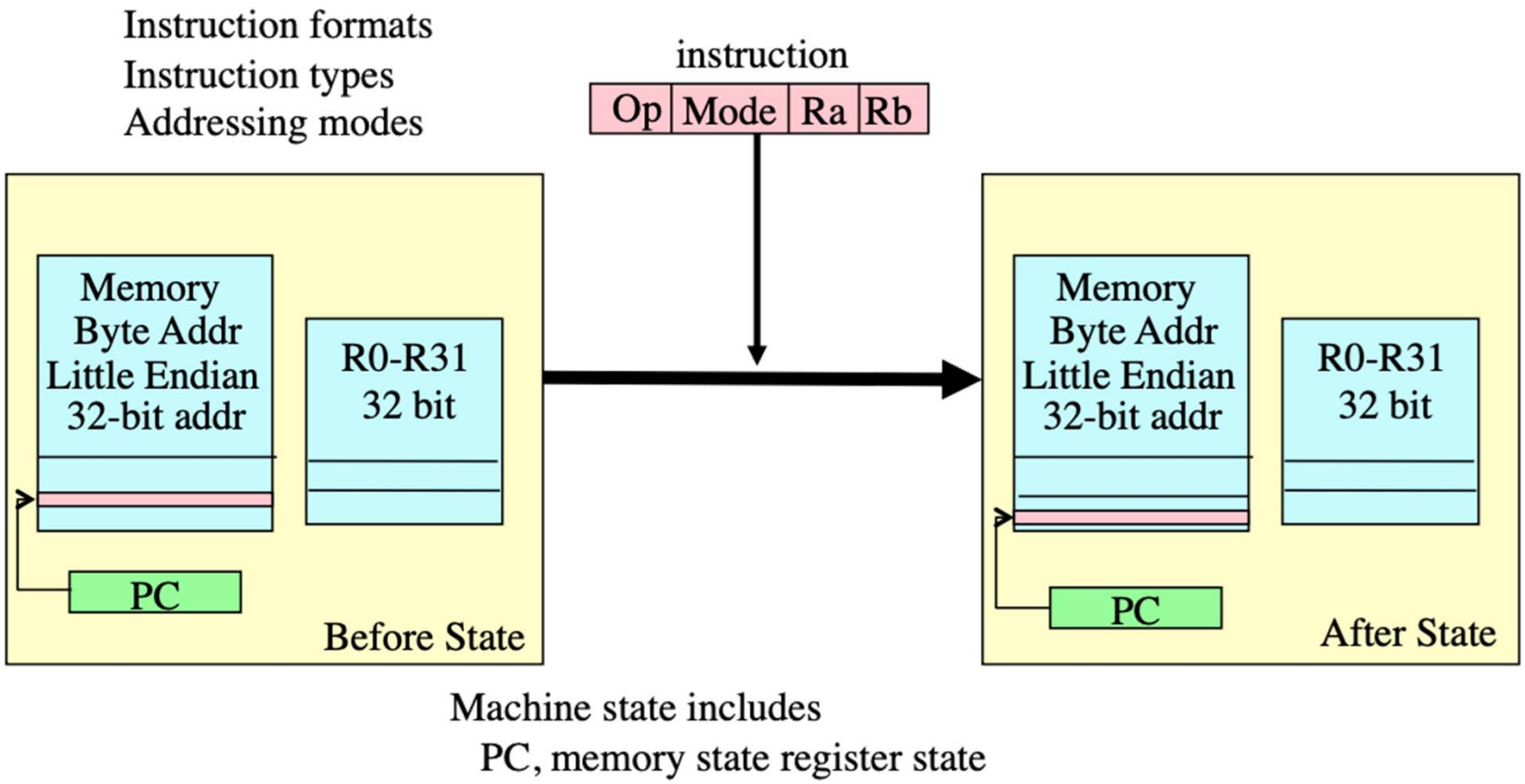
- Also called (computer) architecture
  - Implementation --> actual realization of ISA
  - ISA can have multiple implementations
  - ISA allows software to direct hardware
  - ISA defines machine language

Instruction Set Architecture (ISA)

- Specification of a microprocessor design
  - The ISA defines the CPU, or a CPU family (e.g., x86)
    - Not only a collection of instructions,
    - Includes the CPU view of memory, registers number and roles, etc.
  - ISA ≠ CPU architecture (μ-architecture)
    - E.g., x86: Xeon ≠ Celeron, same ISA
- Interface between user and machine's functionality
  - The ISA is the contract between s/w and h/w



ISA Specifies How the Computer Changes State



Different ISAs: More Like Regional Dialects

It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations…. The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.

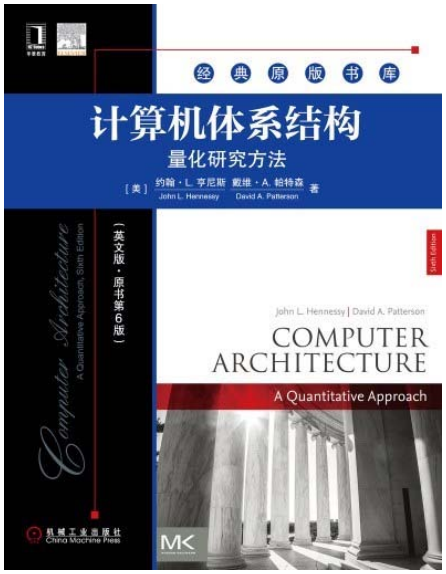
Burks, Goldstine, and von Neumann, 1947

Some Thoughts of Instruction Formats

- Short instruction length
- Enough bits reserved for new opcodes
- Distinct encoding and the illustration
- Number of operands
- Instruction alignment
  - Byte-/word- level alignment
- Always keep regularity
  - Fixed instruction length
  - Fixed bits of opcodes
  - Fixed placement of operands
  - …

Design Principles

- Underlying design principles, as articulated by Hennessy and Patterson:
  - Simplicity favors regularity
  - Make the common case fast
  - Smaller is faster
  - Good design demands good compromises



Design Principles

- Simplicity favors regularity
  - E.g., instruction size, instruction formats, data formats
  - Eases implementation by simplifying hardware
- Make the common case fast
  - Fewer bits to read, move, & write
  - Use/reuse the register file instead of memory
- Smaller is faster
  - E.g., small constants are common, thus immediate fields can be small
- Good design demands good compromises
  - Special formats for important exceptions
  - E.g., a jump far away (beyond a small constant)

Overview

- Introduction to ISA
- About ISA design
  - Issues in the design
  - Operands and addressing modes
  - Types of operations and encodings
  - Evolution and classification

About ISA Design

- Issues in the design
- Operands and addressing modes
- Types of operations and encodings
- Evolution and classification

Basic Design Issues

- What data types are supported and what size
- What operations (and how many) should be provided
  - LD/ST/INC/BRN sufficient to encode any computation, or just Sub and Branch!
  - But not useful because programs too long!
- How (and how many) operands are specified
  - Most operations are dyadic (e.g., A <- B + C)
  - Some are monadic (e.g., A <- ~B)
- Location of operands and result
  - Where other than memory?
  - How many explicit operands?
  - How are memory operands located?
  - Which can or cannot be in memory?
  - How are they addressed
- How to encode these into consistent instruction formats
  - Instructions should be multiples of basic data/address widths
  - Encoding

Typical instruction set:

• 32-bit word
• Basic operand addresses are 32 bits long
• Basic operands, like integers, are 32 bits long
• In general case, instruction could reference 3 operands (A := B + C)

Typical challenge:

• Encode operations in a small number of bits

About ISA Design

- Issues in the design
- Operands and addressing modes
- Types of operations and encodings
- Evolution and classification

Operands

- One important design factor is the number of operands contained in each instruction
  - Has a significant impact on the word size and complexity of the CPU
  - E.g., lots of operands generally implies longer word size needed for an instruction
- Consider how many operands we need for an ADD instruction
  - If we want to add the contents of two memory locations together, then we need to be able to handle at least two memory addresses
  - Where does the result of the add go? We need a third operand to specify the destination
  - What instruction should be executed next?
    - Usually the next instruction, but sometimes we might want to jump or branch somewhere else
    - Do we need a fourth operand to specify the next instruction to execute?
- If all of these operands are memory addresses, we need a really long instruction!

Number of Operands

- In practice, we won't really see a four-address instruction.
  - Too much additional complexity in the CPU
  - Long instruction word
  - All operands won't be used very frequently

- Most instructions have one, two, or three operand addresses
  - The next instruction is obtained by incrementing the program counter, with the exception of branch instructions

- Let's describe a hypothetical set of instructions to carry out the computation for:

$$\mathbf{Y = (A-B) / (C + (D * E))}$$

Three-Operand Instructions

- If we had a three-operand instruction, we could specify two source operands and a destination to store the result.
- Here is a possible sequence of instructions for our equation:

Y = (A-B) / (C + (D * E))

- SUB R1, A, B ; Register R1 ← A - B
- MUL R2, D, E ; Register R2 ← D * E
- ADD R2, R2, C ; Register R2 ← R2 + C
- DIV R1, R1, R2 ; Register R1 ← R1 / R2

- The three-address format is fairly convenient because we have the flexibility to dictate where the result of computations should go. *Note that after this calculation is done, we haven't changed the contents of any of our original locations A, B, C, D, or E.*

Two-Operand Instructions

- How can we cut down the number of operands?
  - Might want to make the instruction shorter
- Typical method is to assign a default destination operand to hold the results of the computation
  - Result always goes into this operand
  - Overwrites and old data in that location
- Let's say that the default destination is the first operand in the instruction
  - First operand might be a register, memory, etc.

Two-Operand Instructions

- Here is a possible sequence of instructions for our equation (say the operands are registers):

Y = (A-B) / (C + (D * E))

```
SUB  A, B        ; Register A ← A - B
MUL D, E         ; Register D ← D * E
ADD D, C         ; Register D ← D + C
DIV  A, D        ; Register A ← A / D
```

- Get same end result as before, but we changed the contents of registers A and D
- If we had some later processing that wanted to use the original contents of those registers, we must make a copy of them before performing the computation

```
MOV  R1, A       ; Copy A to register R1
MOV  R2, D       ; Copy D to register R2
SUB  R1, B       ; Register R1 ← R1 - B
MUL R2, E        ; Register R2 ← R2 * E
ADD R2, C        ; Register R2 ← R2 + C
DIV  R1, R2      ; Register R1 ← R1 / R2
```

- Now the original registers for A-E remain the same as before, but at the cost of some extra instructions to save the results.

One-Operand Instructions

- Can use the same idea to get rid of the second operand, leaving only one operand
- The second operand is left implicit; e.g., could assume that the second operand will always be in a register such as the Accumulator:

Y = (A-B) / (C + (D * E))

```
LDA   D          ; Load ACC with D
MUL   E          ; Acc ← Acc * E
ADD   C          ; Acc ← Acc + C
STO   R1         ;  Store Acc to R1
LDA   A          ; Acc ← A
SUB   B          ; Acc ← A-B
DIV   R1         ; Acc ← Acc / R1
```

- Many early computers relied heavily on one-address based instructions, as it makes the CPU much simpler to design. As you can see, it does become somewhat more unwieldy to program.

Zero-Operand Instructions

- In some cases, we can have zero-operand instructions
- Uses the **Stack**
  - Section of memory where we can add and remove items in LIFO order
    - Last In, First Out
  - Envision a stack of trays in a cafeteria; the last tray placed on the stack is the first one someone takes out
  - The stack in the computer behaves the same way, but with data values
    - PUSH A ; Places A on top of stack
    - POP A ; Removes value on top of stack and puts result in A
    - ADD ; Pops top two values off stack, pushes result back on

Stack-Based Instructions

Y = (A-B) / (C + (D * E))

Instruction	**Stack Contents (stack increase ->)**
PUSH B	; B
PUSH A	; B, A
SUB	; (A-B)
PUSH E	; (A-B), E
PUSH D	; (A-B), E, D
MUL	; (A-B), (E*D)
PUSH C	; (A-B), (E*D), C
ADD	; (A-B), (E*D+C)
DIV	; (A-B) / (E*D+C)

How many operands are best?

- More operands
  - More complex (powerful?) instructions
  - Fewer instructions per program
- More registers
  - Inter-register operations are quicker
- Fewer operands
  - Less complex (powerful?) instructions
  - More instructions per program
  - Faster fetch/execution of instructions

Design Tradeoff Decisions

- Operation repertoire
  - How many ops?
  - What can they do?
  - How complex are they?
- Data types
  - What types of data should ops perform on?
- Registers
  - Number of registers, what ops on what registers?
- Addressing
  - Mode by which an address is specified (more on this later)

Addressing

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- The actual location of an operand is its ***effective address***.
- Certain addressing modes allow us to determine the address of an operand dynamically.

Addressing Modes

- Addressing refers to how an operand refers to the data we are interested in for a particular instruction
- In the Fetch part of the instruction cycle, there are several common ways to handle addressing in the instruction
  - Immediate
  - Direct
  - Indirect
  - Register Direct/Indirect
  - Relative, Indexed and Based

Immediate Addressing

- The operand directly contains the value we are interested in working with
  - E.g., ADD #5
    - Means add the number 5 to something
  - This uses immediate addressing for the value 5
  - The two's complement representation for the number 5 is directly stored in the ADD instruction
  - Must know value at assembly time

Direct Addressing

- The operand contains an address with the data
  - E.g., ADD 100
    - Means to add (Contents of Memory Location 100) to something
  - Downside: Need to fit entire address in the instruction, may limit address space
    - E.g., 32-bit word size and 32-bit addresses. Do we have a problem here?
    - Some solutions: specify offset only, use implied segment
  - Must know address at assembly time

Indirect Addressing

- The operand contains an address, and that address contains the address of the data
  - E.g., Add [100]
    - Means "The data at memory location 100 is an address. Go to the address stored there and get that data and add it to the Accumulator"
  - Downside: Requires additional memory access
  - Upside: Can store a full address at memory location 100
    - First address must be fixed at assembly time, but second address can change during runtime! This is very useful for dynamically accessing different addresses in memory (e.g., traversing an array)
- Indirect Addressing can be thought of as additional instruction subcycle

Register Direct/Indirect Addressing

- Can do direct addressing with registers
  - E.g., ADD R5
    - Means to add (Contents of Register 5) to something
  - Upside: Not that many registers, don't have previous problem of the direct addressing

- Can also do indirect addressing with registers
  - E.g., Add [R3]
    - Means "The data in register 3 is an address. Go to that address in memory, get the data, and add it to the Accumulator"

Other Addressing Modes

- ***Relative addressing*** uses the PC register as an offset, which is added to the address in the operand to determine the effective address of the data.
- ***Indexed addressing*** uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- ***Based addressing*** is similar except that a base register is used instead of an index register.
- The difference between Indexed addressing and Based addressing is that an index register holds an offset relative to the address given in the instruction, a base register holds a base address where the address field represents a displacement from this base.

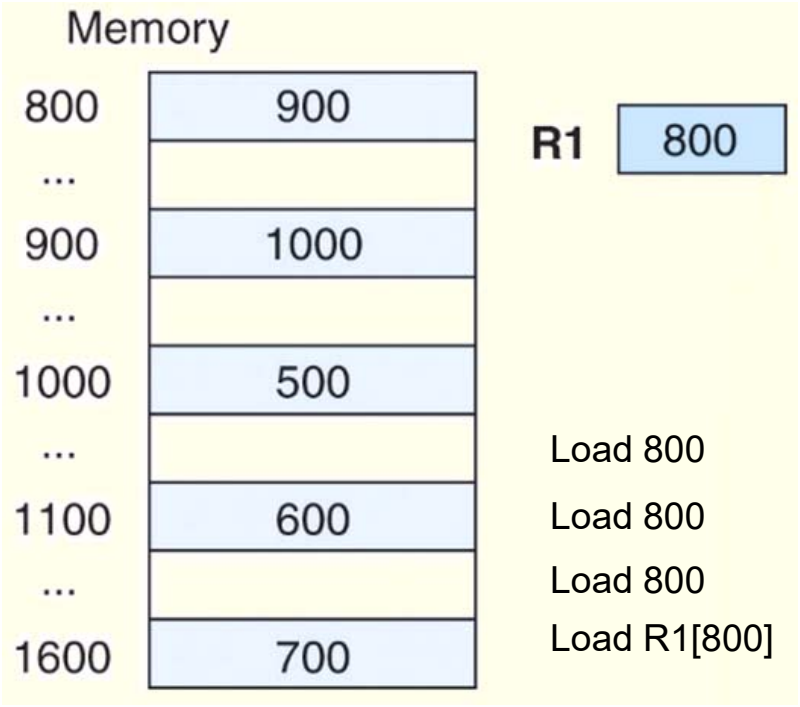
Summary - Addressing Modes

Addressing Mode	Syntax	Meaning
Immediate	#K	K
Direct	K	M[K]
Indirect	(K)	M[M[K]]
Register	(Rn)	M[Rn]
Register Indexed	(Rm + Rn)	M[Rm + Rn]
Register Based	(Rm + X)	M[Rm + X]
Register Based Indexed	(Rm + Rn + X)	M[Rm + Rn + X]

Four ways of computing the address of a value in memory: (1) a constant value known at assembly time, (2) the contents of a register, (3) the sum of two registers, (4) the sum of a register and a constant. The table gives names to these and other addressing modes.

Addressing Example

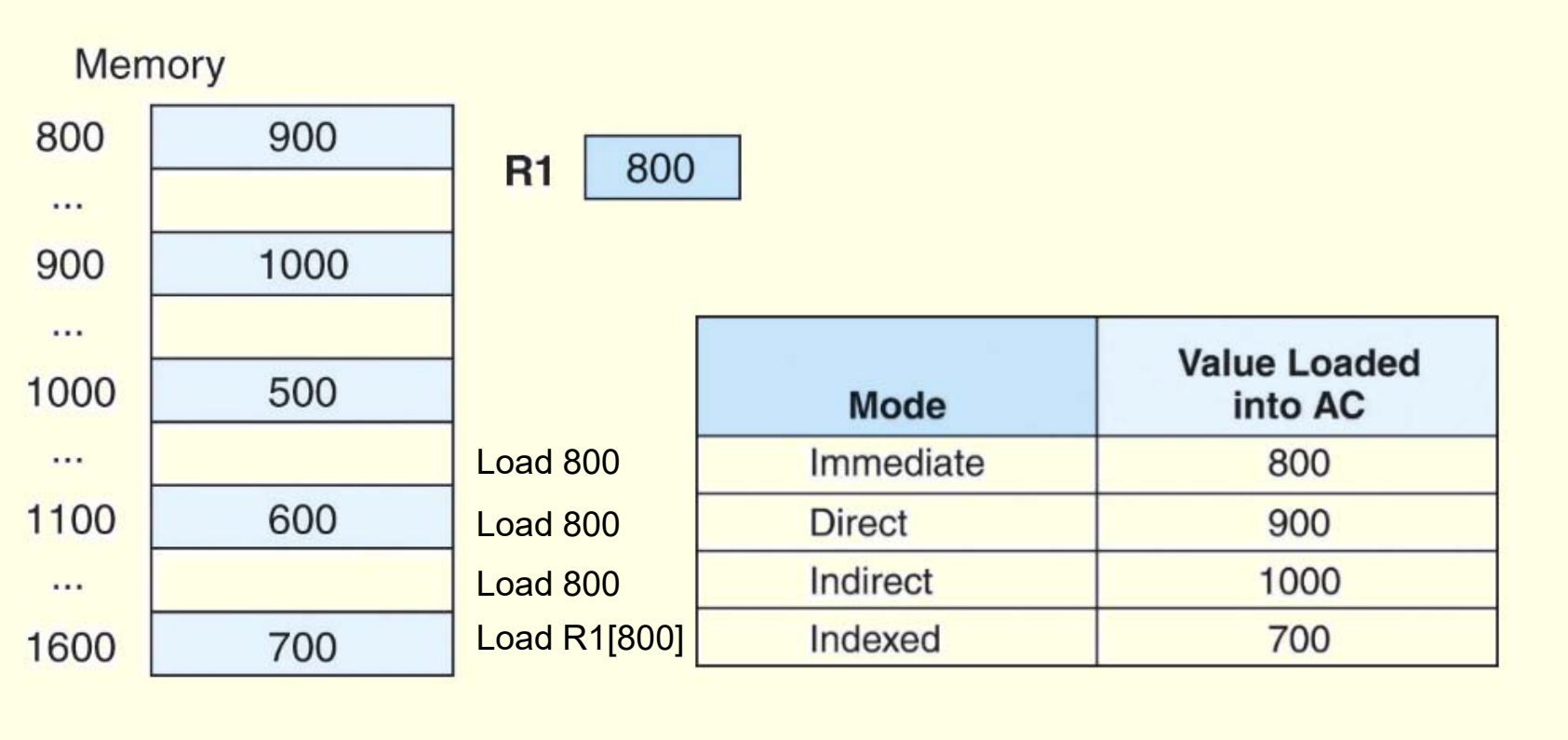
- What value is loaded into the accumulator for each addressing mode?



Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

Addressing Example

- These are the values loaded into the accumulator for each addressing mode.



Mode	Value Loaded into AC
Immediate	800
Direct	900
Indirect	1000
Indexed	700

Load R1 Using Indirect Addressing?

About ISA Design

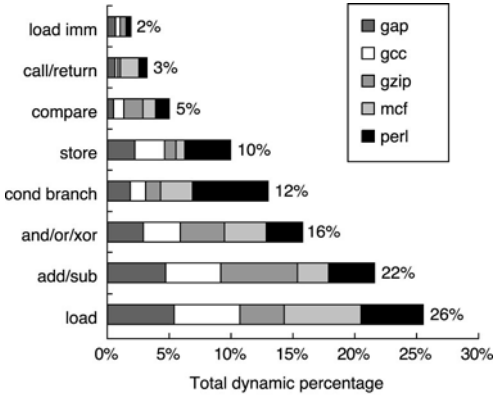
- Issues in the design
- Operands and addressing modes
- Types of operations and encodings
- Evolution and classification

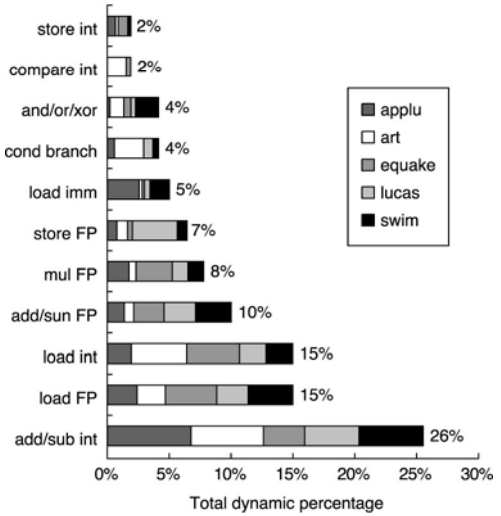
Types of Operations

- Arithmetic and Logic
- Shift
- Data Transfer
  - E.g., MOV/LOAD/STORE
- String
- Control
  - BRANCH/JMP/CALL/RET/…
- System
  - HALT/INTERRUPT ON/INTERRUPT OFF/SWITCH…
- Input/Output
- …

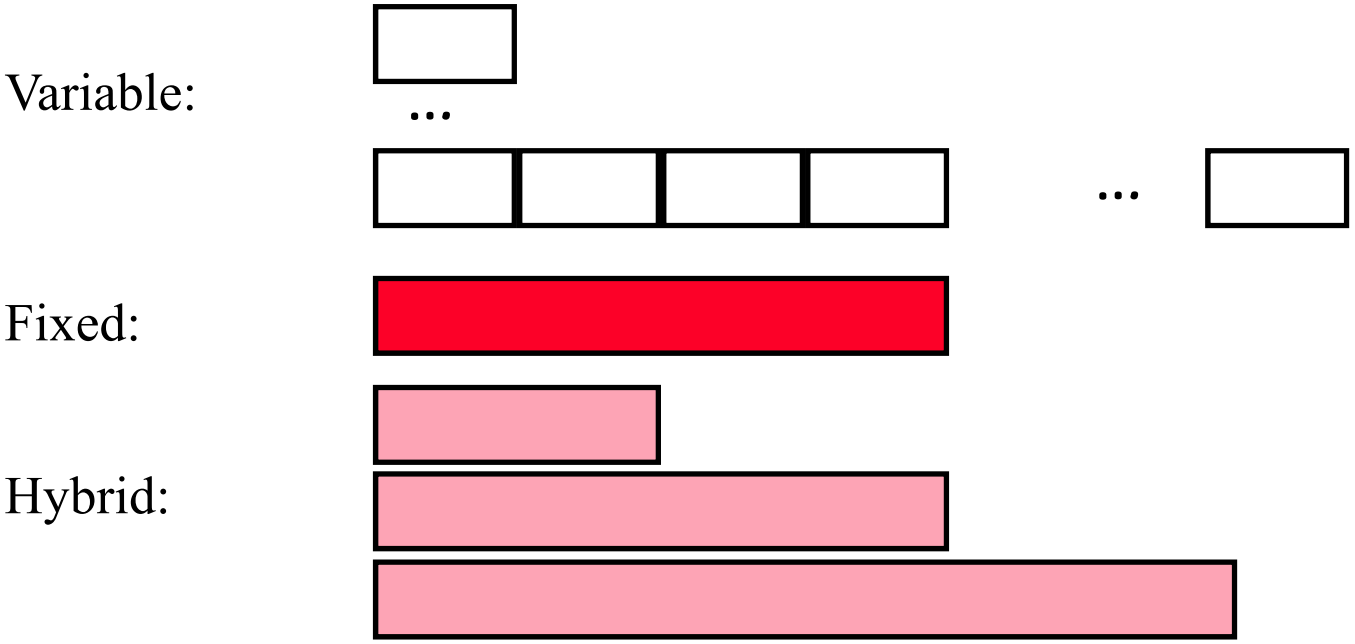
Typical Operations (little change since 1960)

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)





Types of Encodings

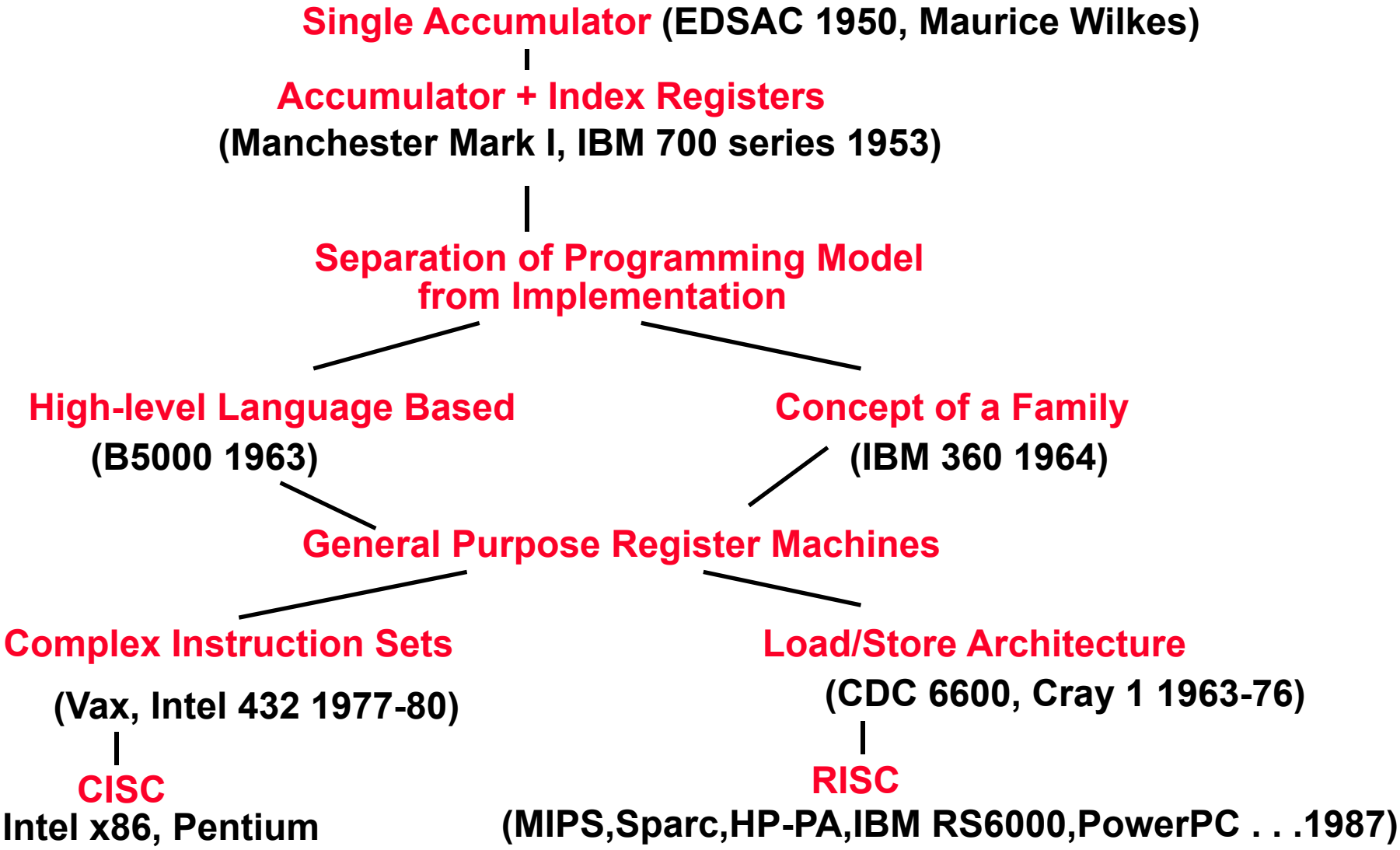


- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions
- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density
-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

About ISA Design

- Issues in the design
- Operands and addressing modes
- Types of operations and encodings
- Evolution and classification

Evolution of Instruction Sets



CISC

- Complex Instruction Set Computers
- Close “semantic gap” between programming and execution
  - Smaller code size (memory was expensive!)
  - Simplify compilation
- Another state machine (controlled by microcode) inside the machine
- Example: x86, Intel 432, IBM 360, DEC VAX

CISC Example: x86

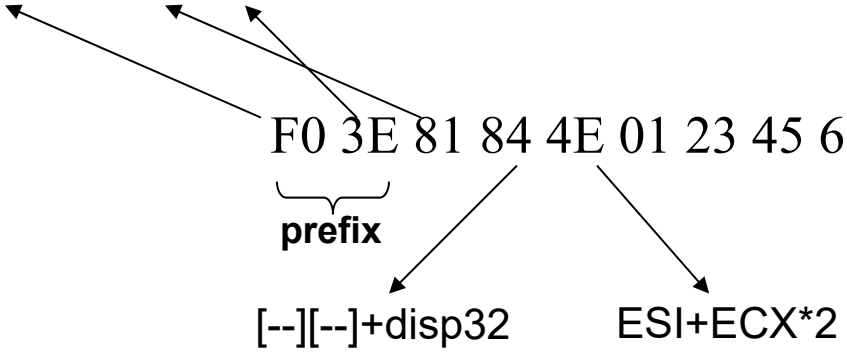
- MOVSD ;; move a double word, 1-byte instruction

MOVSD // m32[DS:EDI] = m32[DS:ESI]

- REP;; 1-byte prefix to repeat string operations

REP MOVSD // count set up in ECX

LOCK ADD ds:[esi+ecx*2+0x67452301], 0xEFCDAB89 // 13-byte



RISC

- Observation made by IBM (John Cocke, Eckert-Mauchly Award'85, Turing Award'87,…)
  - Few of the available instructions are used

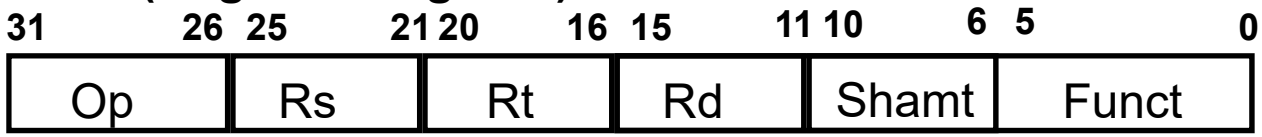
- CISC : "n+1" phenomenon
  - Adding an instruction requiring an extra level of decoding logic can slow down the entire ISA
- Reduced Instruction Set Computer
  - Originated at IBM in 1975, a telephone project
  - To achieve 12 MIPS (300 calls per sec, 20k inst per call)
  - Simple instructions
  - IBM 801 in 1978
  - More compiler effort to gain performance

A Typical RISC

- Smaller number of instructions
- Fixed format instruction (e.g., 32 bits)
- 3-address, reg-to-reg arithmetic instructions
- Single cycle operation for execution
- Load-store architecture
- Simple address modes
  - Base + displacement
- No indirection
- Simple branch conditions
- Hardwired control (No microcode)
- More compiler effort
- Examples:
  - RISC I and RISC II at Berkeley
  - MIPS (Microprocessors without Interlocked Pipe Stage) at Stanford
  - IBM RISC Technology, Sun Sparc, HP PA-RISC, ARM

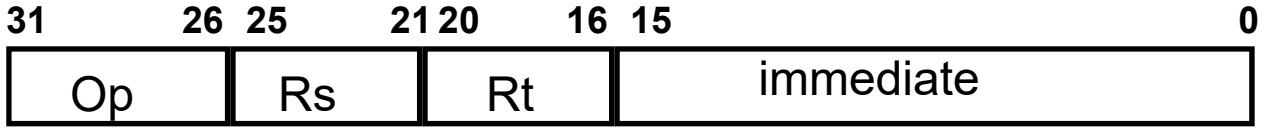
RISC Example: MIPS

R-format (Register-Register)



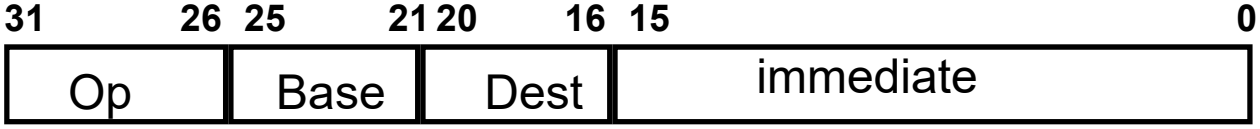
add $1, $2, $3

I-format (Register-Immediate)



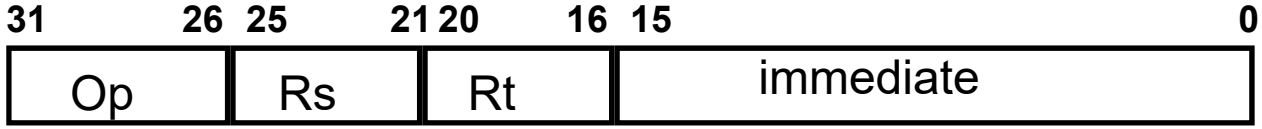
addi $1, $2, -5

I-format (Load/Store)



lw $1, 24($9)

I-format (Branch)



beq L1, $4, $0

J-format (Jump / Call)

j L2

CISC vs. RISC

CISC	RISC
Variable length instructions	Fixed-length instructions, single-cycle operation
Abundant instructions and addressing modes	Fewer instructions and addressing modes
Long, complex decoding	Simple decoding
Contain mem-to-mem operations	Load/store architecture
Use microcode	No microinstructions, directly decoded and executed by HW logic
Closer semantic gap (shift complexity to microcode)	Needs smart compilers, or intelligent hardware to reorder instructions
IBM 360, DEC VAX, x86, Moto 68030	IBM 801, MIPS, RISC I, IBM POWER, Sun Sparc

- Some definitions were from the paper by Colwell et al. in 1985

CISC vs. RISC (Reality)

	CISC			RISC		
	IBM 370/168	VAX 11/780	Xerox Dorado	IBM 801	Berkeley RISC1	Stanford MIPS
Year introduced	1973	1978	1978	1980	1981	1983
# instructions	208	303	270	120	39	55
Microcode	54KB	61KB	17KB	0	0	0
Instruction size	2 to 6 B	2 to 57 B	1 to 3 B	4B	4B	4B
Execution model	Reg-reg Reg-mem Mem-mem	Reg-reg Reg-mem Mem-mem	Stack	Reg-reg	Reg-reg	Reg-reg

Observation and Controversy

- "Instruction Set and Beyond: Computers, Complexity and Controversy" by Bob Colwell (Eckert-Mauchly Award, 2005) and gang from CMU, also see response from RISC camp: Patterson (Eckert-Mauchly Award, 2008) and Hennessy (Eckert-Mauchly Award, 2001)

- **CISC/RISC classification should *NOT* be a dichotomy!**

- Case in point: MicroVAX-32 by DEC, a single chip implementation
  - Subsetting VAX instructions (but still, 175 instructions!)
  - Emulate complex instructions
  - A RISC or a CISC? (Well, it has variable length instructions, not a ld/st machine, with a microcode control, have all VAX addressing mode)

Observation and Controversy

- Effective processor design = CISC experiences + RISC tenets
- RISC features are not incompatible or mutually exclusive
  - Large register file (w/ register windows)
- RISC/CISC issues are best considered in light of their **function-to-implementation level** assignment

Modern X86 Machine Design

- **CISC outfit + RISC inside**
- E.g., Intel P6/Netburst/Core, AMD Athlon/Phenom/Opteron
- Each x86 instruction is decoded into “micro-op” (μ op) or “RISC-op” on-the-fly
- Internal microarchitecture resembles RISC design philosophy
- Processor dynamically schedules “μ ops”
- Compiler’s scheduling is still beneficial

Recent ISA Design Trend

- Look at this instruction in MIPS (CISC or RISC?)

  CABS.LE.PS $fcc0, $f8, $f10 ;; |y|≤|w| , |x| ≤ |w|?

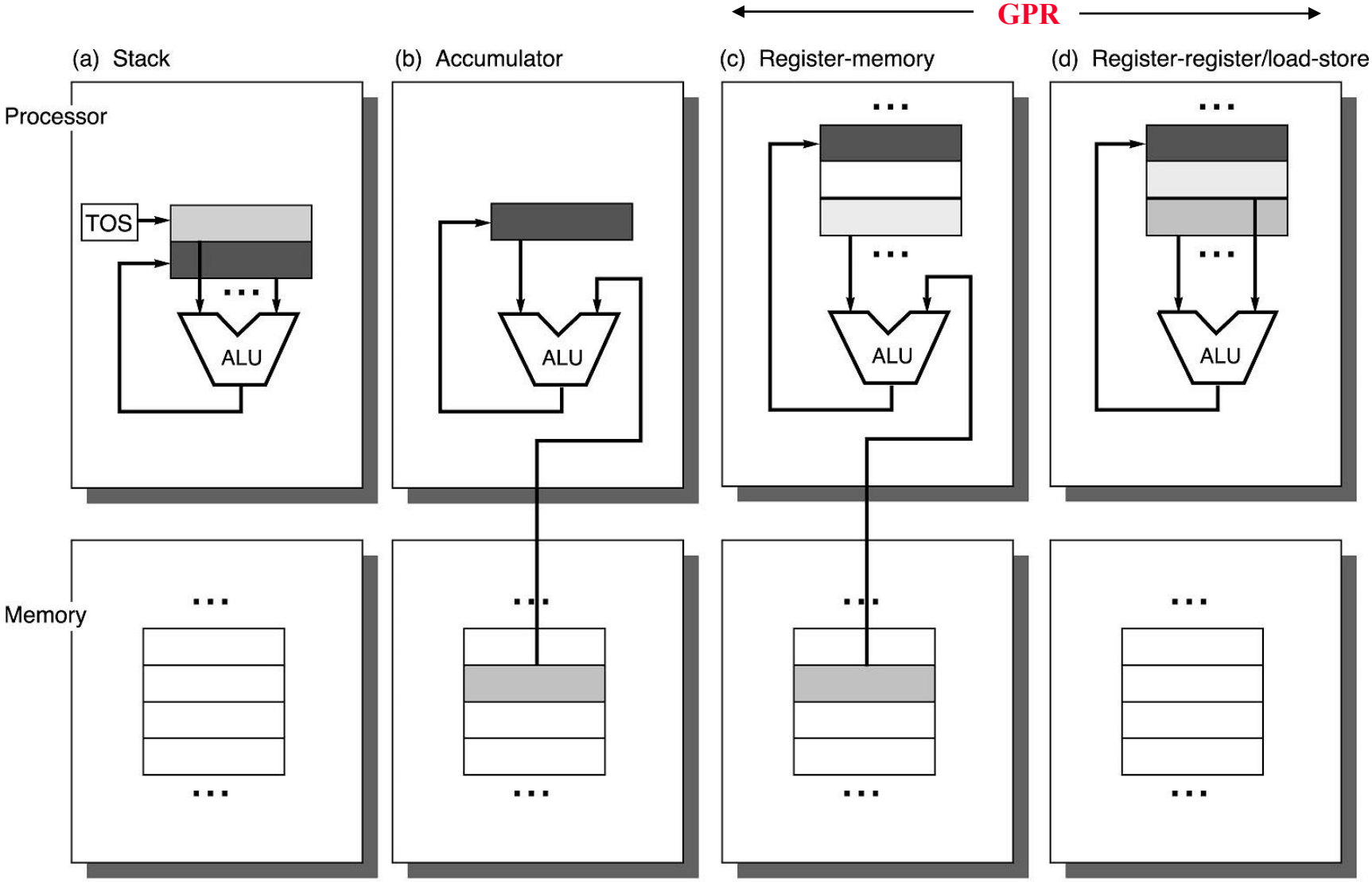
- Many complex instructions emerged for new apps
  - Viterbi instruction for wireless communication/DSP
  - Sum of absolute differences in SSE (PSAD) or other DSP: C = ∑|A-B| for MPEG (motion estimation)

- In embedded domain, code size is critical
- Reducing programming efforts
- Optimizing performance via
  - Specialized hardware (accelerator-based)
  - Co-processor (controlled by main processor)
  - ISA plug-in (flexible)

Classification of ISAs

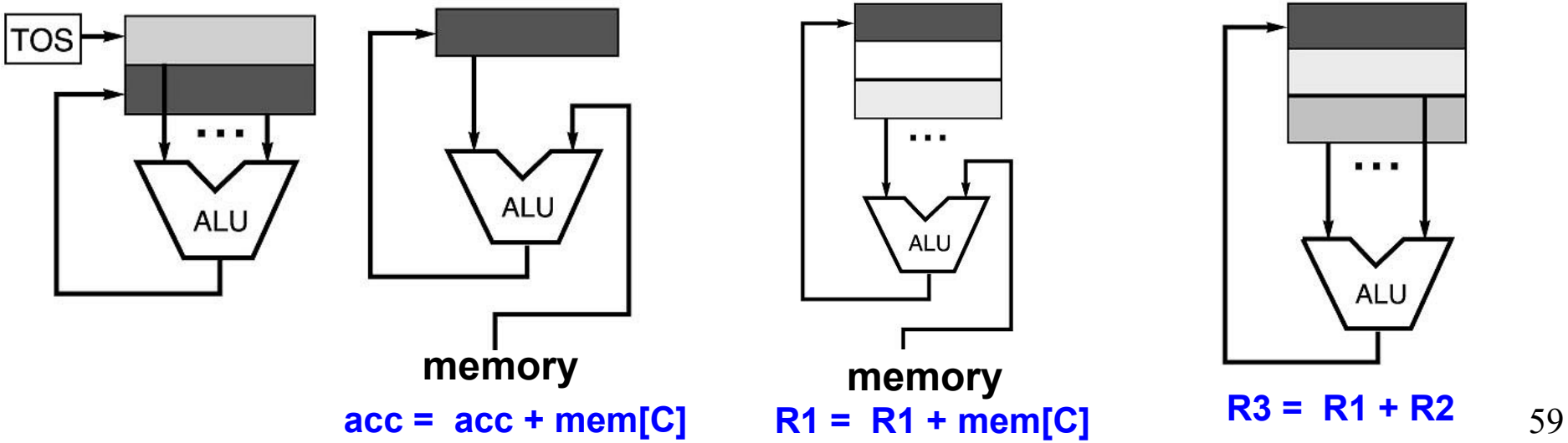
- Accumulator (before 1960, e.g., 68HC11):
  - 1-address add A acc ← acc + mem[A]
- Stack (1960s to 1970s):
  - 0-address add tos ← tos + next
- Memory-Memory (1970s to 1980s):
  - 2-address add A, B mem[A] ← mem[A] + mem[B]
  - 3-address add A, B, C mem[A] ← mem[B] + mem[C]
- Register-Memory (1970s to present, e.g., 80x86):
  - 2-address add R1, A R1 ← R1 + mem[A]
    load R1, A R1 ← mem[A]
- Register-Register (Load/Store) (1960s to present, e.g., MIPS):
  - 3-address add R1, R2, R3 R1 ← R2 + R3
    load R1, R2 R1 ← mem[R2]
    store R1, R2 mem[R1] ← R2

Operand Locations in Four ISA Classes



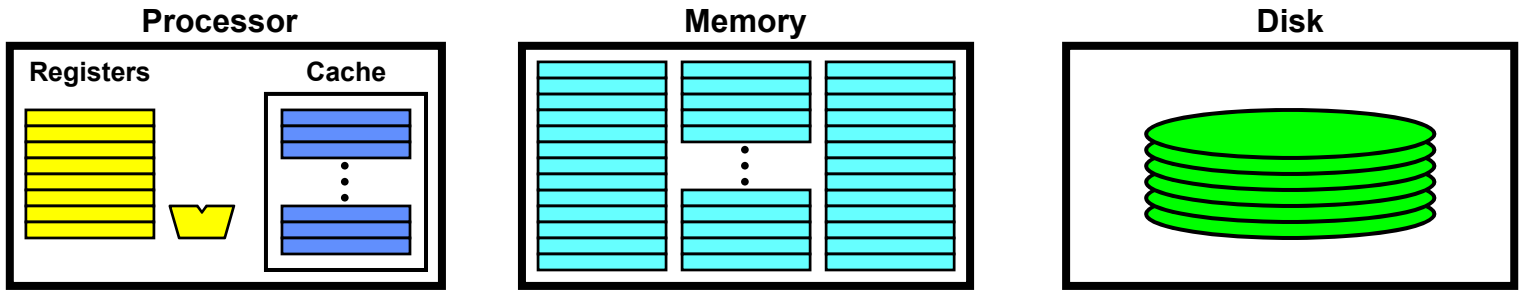
Code Sequence C = A + B for Four Instruction Sets

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



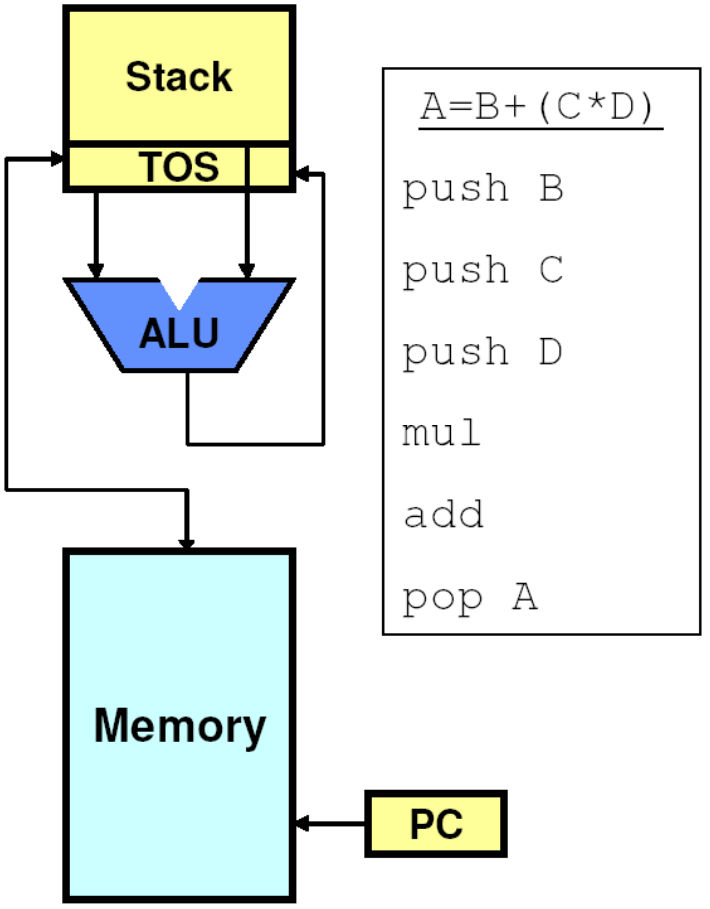
More About General Purpose Registers

- Why do almost all new architectures use GPRs?
  - Registers are much faster than memory (even cache)
    - Register values are available immediately
    - When memory isn't ready, processor must wait ("stall")
- Registers are convenient for variable storage
    - Compiler assigns some variables just to registers
    - More compact code since small fields specify registers (compared to memory addresses)



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



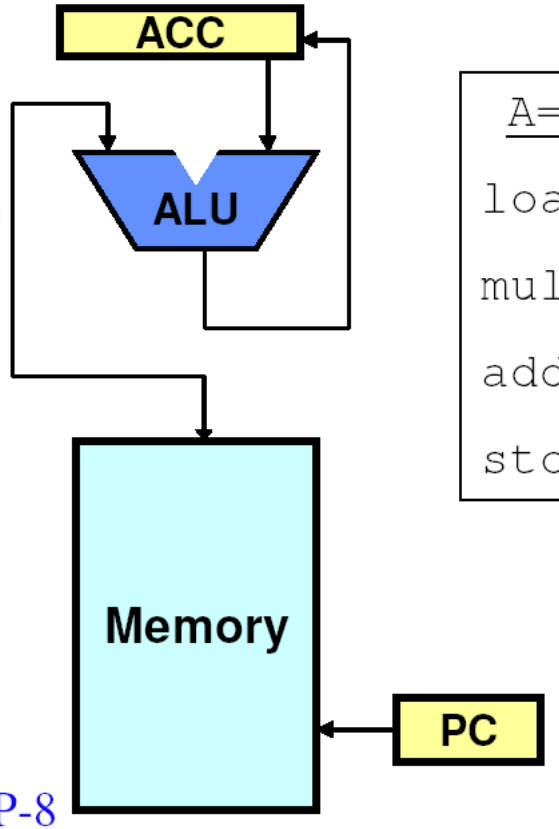
Stacks: Pros and Cons

- Pros
  - Good code density (implicit top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures

- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when need, so additional instructions like TOP and SWAP are needed
  - Difficult to write an optimizing compiler for stack architectures

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
```

Accumulators: Pros and Cons

- Pros
  - Very low hardware requirements
  - Easy to design and understand

- Cons
  - Accumulator becomes the bottleneck
  - Little ability for parallelism or pipelining
  - High memory traffic

Memory-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
```

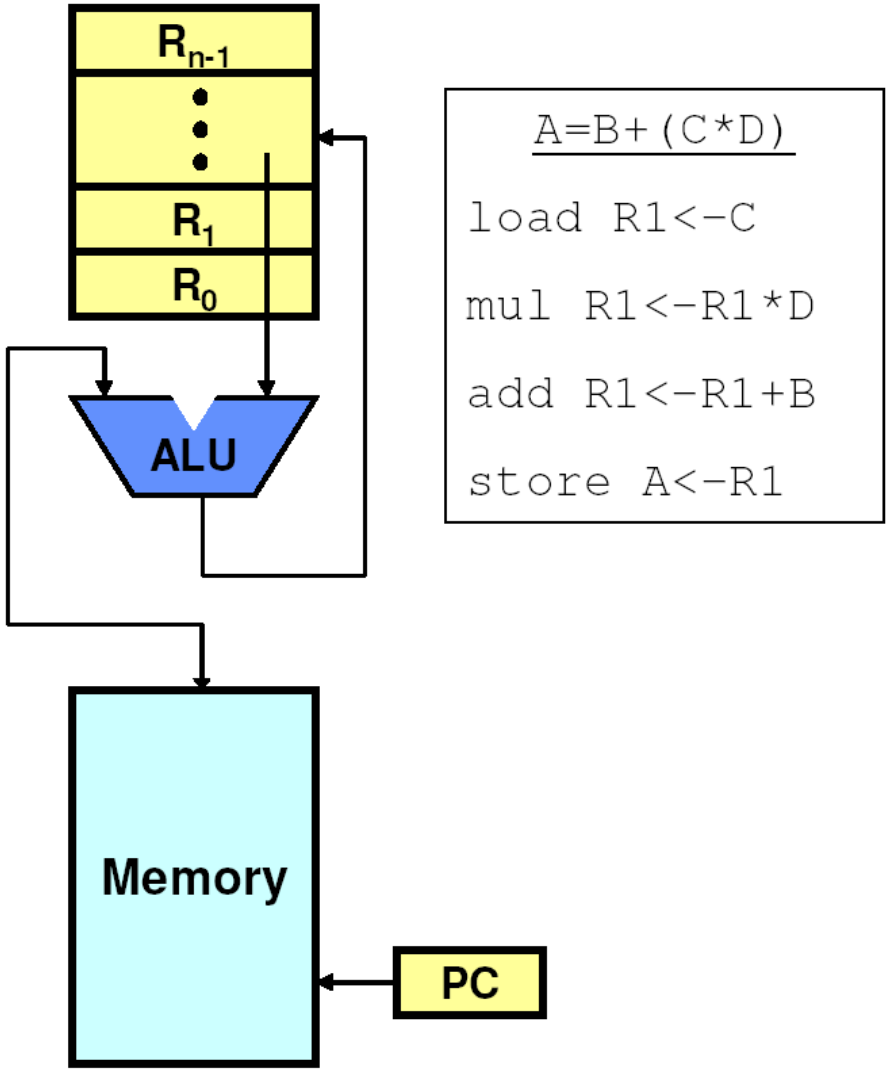
Memory-Memory: Pros and Cons

- Pros
  - Requires fewer instructions (especially if 3 operands)
  - Easy to write compilers for (especially if 3 operands)

- Cons
  - Very high memory traffic (especially if 3 operands)
  - Variable number of clocks per instruction
  - With two operands, more data movements are required

Register-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



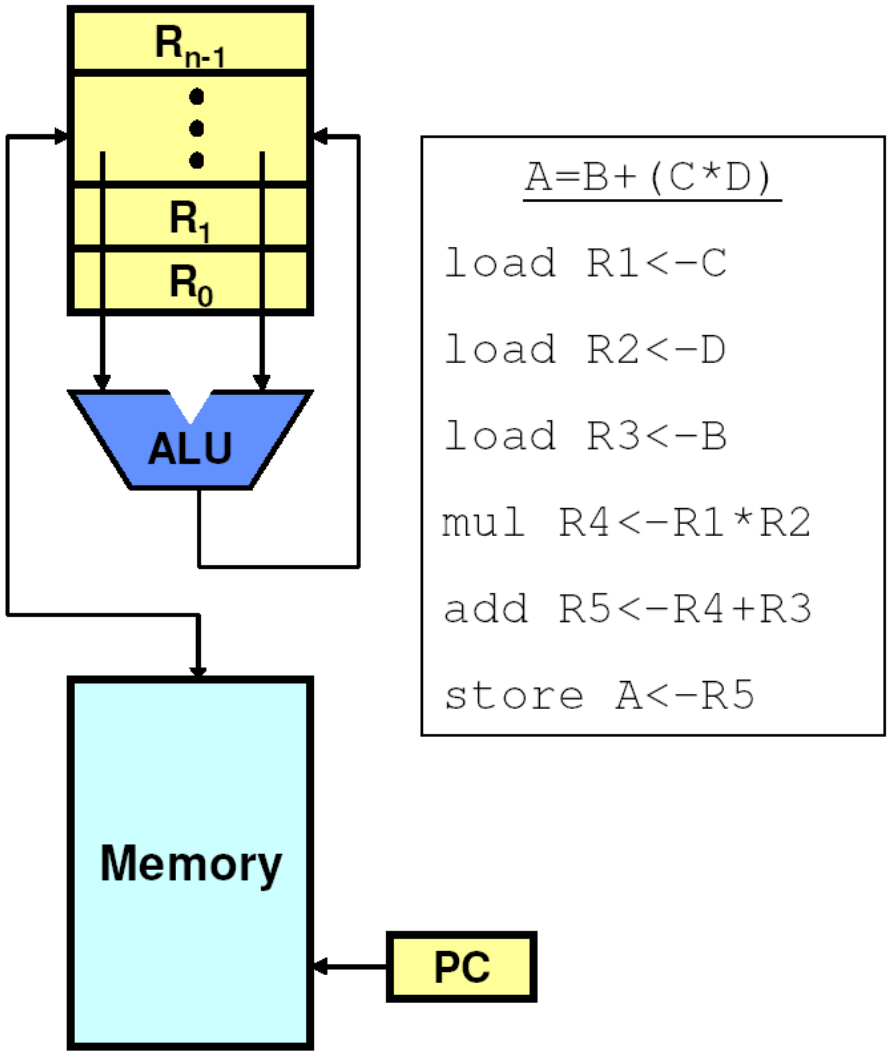
Register-Memory: Pros and Cons

- Pros
  - Some data can be accessed without loading first
  - Instruction format easy to encode
  - Good code density

- Cons
  - Operands are not equivalent (poor orthogonal)
  - Variable number of clocks per instruction
  - May limit number of registers

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
```

Load-Store: Pros and Cons

- Pros
  - Simple, fixed length instruction encodings
  - Instructions take similar number of cycles
  - Relatively easy to pipeline and make superscalar

- Cons
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler

Registers: Advantages and Disadvantages

- Advantages
  - Faster than cache or main memory (no addressing mode or tags)
  - Deterministic (no misses)
  - Can replicate (multiple read ports)
  - Short identifier (typically 3 to 8 bits)
  - Reduce memory traffic
- Disadvantages
  - Need to save and restore on procedure calls and context switch
  - Can't take the address of a register (for pointers)
  - Fixed size (can't store strings or structures efficiently)
  - Compiler must manage
  - Limited number

Every ISA designed after 1980 uses a load-store ISA (i.e., RISC, to simplify CPU design).