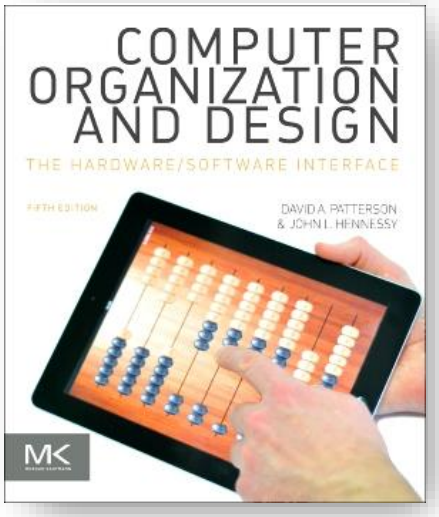
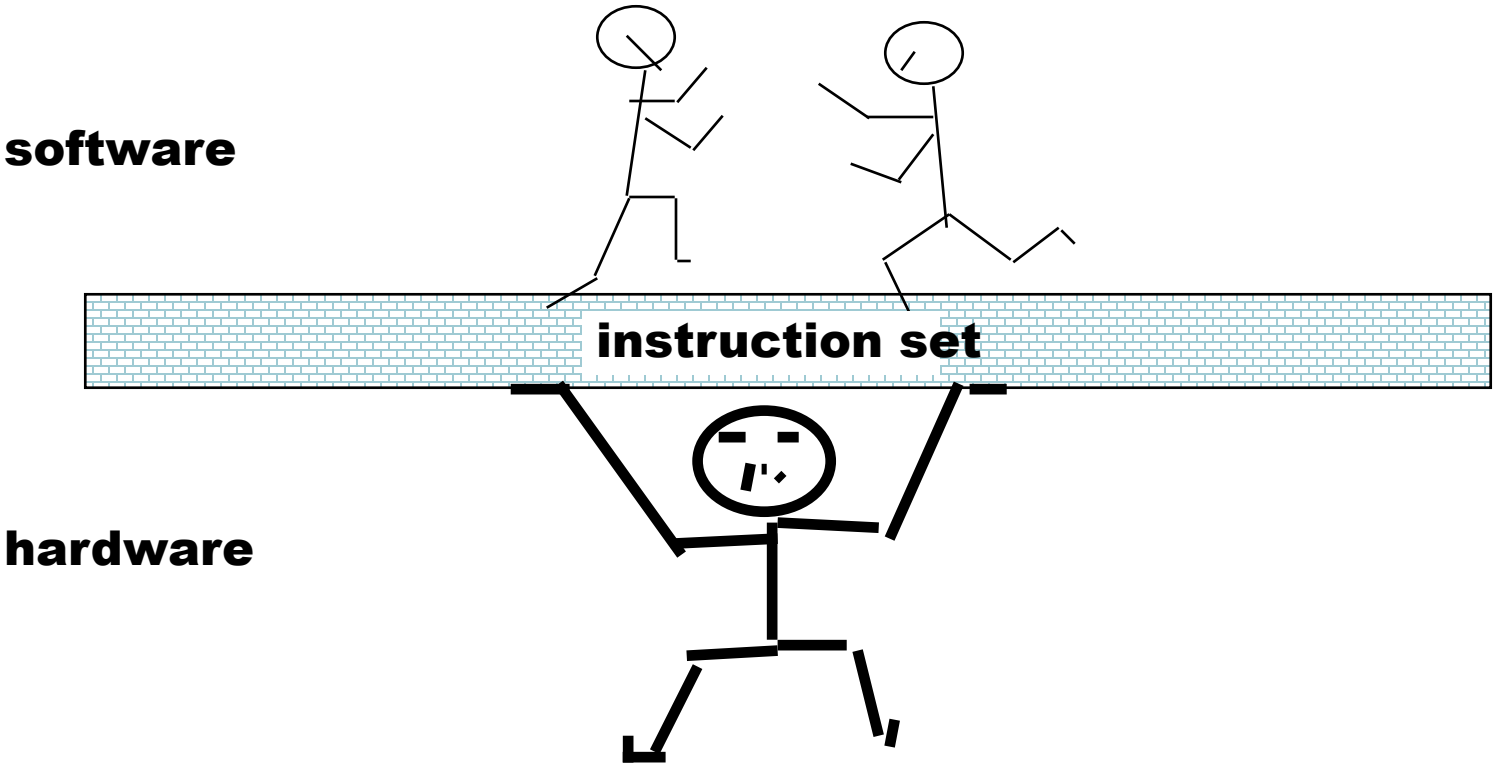


ESE 345 Computer Architecture

Instruction Set Design

The Instruction Set: a Critical Interface



Instructions:

- Language of the Machine
- Vocabulary of all “words” a computer understands (a conceptual view of a computer visible to programmers): *instruction set architecture (ISA)*
- Why might you want the same ISA? Why might you want different ISAs?
  - e.g. iPhone and iPad use the same ISA
  - e.g. iPhone and Macbook use different ISAs
  -

Machine Language

- Single ISA
  - Leverage common compilers, operating systems, etc.
  - BUT fairly easy to retarget these for different ISAs (e.g. Linux, gcc)
- Multiple ISAs
  - Specialized instructions for specialized applications
  - Different tradeoffs in resources used (e.g. functionality, memory demands, complexity, power consumption, etc.)
  - Competition and innovation is good, especially in emerging environments (e.g. mobile devices)

Stored Program Concept

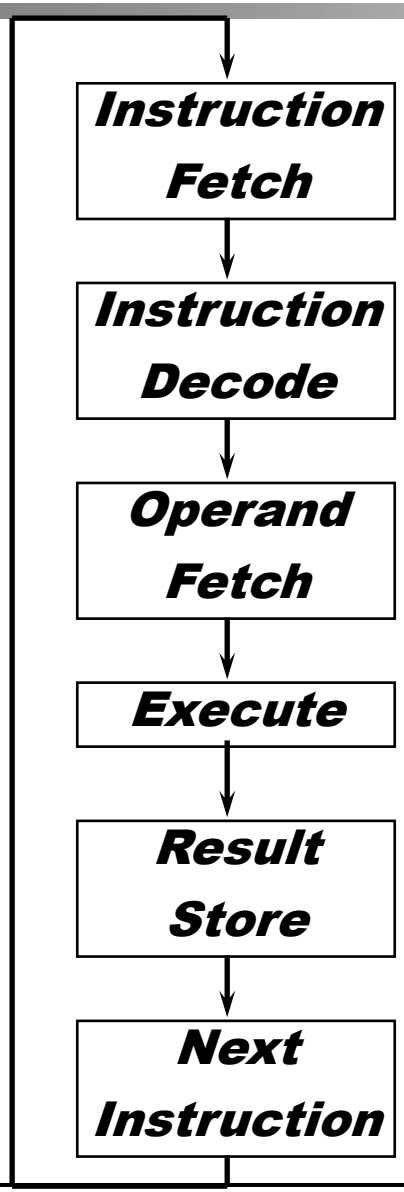
- Instructions are bits
- Programs (sequences of instructions) are stored in memory
  — to be read or written just like data

Control instruction sequencing in early computers

manual control	calculators	
automatic control		
external (paper tape)	Harvard Mark I , 1944; Zuse's Z1, WW2	
internal		
plug board	ENIAC	1946
read-only memory	ENIAC	1948
read-write memory	EDVAC	1947 *(concept)*
The same storage can be used to store program & data	***EDSAC***	***1950 Maurice Wilkes***

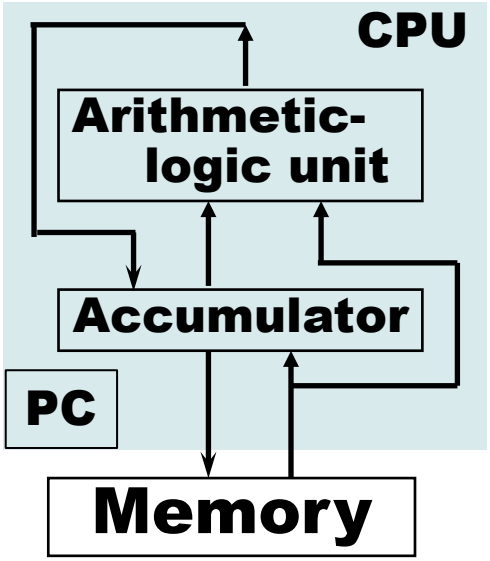
- Fetch & Execute Cycle

Instruction Set Architecture: What Must be Specified?



- Instruction Format or Encoding
  - how is it decoded?
- 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?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
  - *fetch-decode-execute is implicit!*

Single Accumulator Computer (early 50's)



- A single data register called accumulator (AC)
- ALU with one ALU operand is from the accumulator, another from memory M
- The ALU result is written to AC
- Load and store instructions to move data between AC and memory M using absolute addresses
- Special registers: Program Counter (PC), a quotient register
- Circuits to implement control flow change operations (conditional and unconditional)

The Earliest Instruction Sets

Single Accumulator - A carry-over from the calculators.

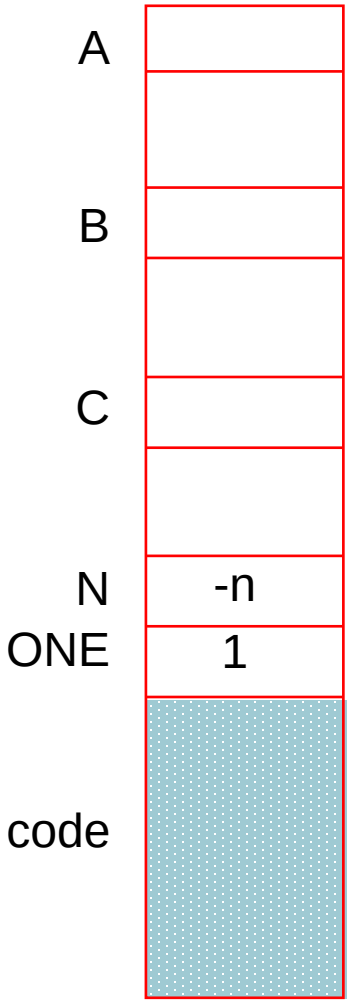
LOAD	x	AC ← M[x]
STORE	x	M[x] ← (AC)
ADD	x	AC ← (AC) + M[x]
SUB	x	
MUL	x	Involved a quotient register
DIV	x	
SHIFT LEFT		AC ← 2 × (AC)
SHIFT RIGHT		
JUMP	x	PC ← x
JGE	x	if (AC) ≥ 0 then PC ← x
LOAD ADR	x	AC ← Extract address field(M[x])
STORE ADR	x	

Typically less than 2 dozen instructions!

Programming: Single Accumulator Machine

$C_i \leftarrow A_i + B_i, \quad 1 \le i \le n$

```
LOOP    LOAD     N
        JGE      DONE
        ADD      ONE
        STORE    N
F1      LOAD     A
F2      ADD      B
F3      STORE    C
        JUMP     LOOP
DONE    HLT
```



How to modify the addresses A, B and C ?

Self-Modifying Code

$$C_i \leftarrow A_i + B_i, \quad 1 \le i \le n$$

```
LOOP    LOAD        N
        JGE         DONE
        ADD         ONE
        STORE       N
F1      LOAD        A
F2      ADD         B
F3      STORE       C
        LOAD ADR    F1
        ADD         ONE
        STORE ADR   F1
        LOAD ADR    F2
        ADD         ONE
        STORE ADR   F2
        LOAD ADR    F3
        ADD         ONE
        STORE ADR   F3
        JUMP        LOOP
DONE    HLT
```

modify the program for the next iteration

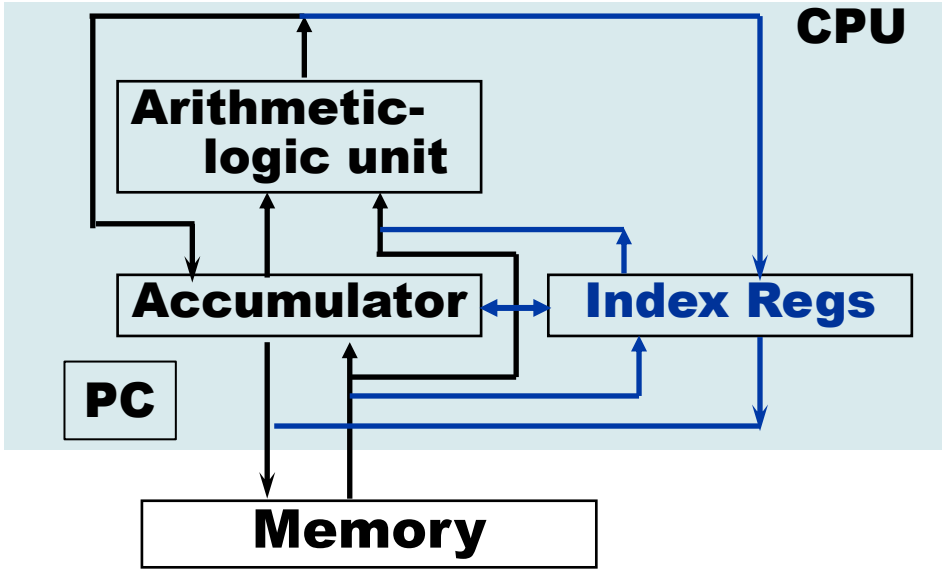
Each iteration involves

	total	*book-keeping*
instruction fetches	*17*	*14*
operand fetches	*10*	*8*
stores	*5*	*4*

Index Registers

Tom Kilburn, Manchester University, mid 50's

One or more specialized registers to simplify address calculation



Modify existing instructions

LOAD	x, IX	AC ← M[x + (IX)]
ADD	x, IX	AC ← (AC) + M[x + (IX)]
...		

Add new instructions to manipulate *index registers*

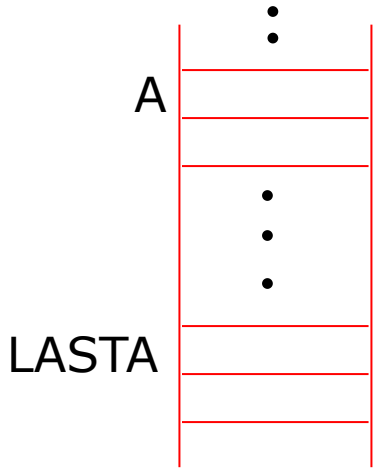
JZi	x, IX	if (IX)=0 *then* PC ← x
		else IX ← (IX) + 1
LOADi	x, IX	IX ← M[x] (truncated to fit IX)
...		

Index registers have accumulator-like characteristics

Using Index Registers

$$C_i \leftarrow A_i + B_i, \quad 1 \le i \le n$$

```
        LOADi   -n, IX
LOOP    JZi     DONE, IX
        LOAD    LASTA, IX
        ADD     LASTB, IX
        STORE   LASTC, IX
        JUMP    LOOP
DONE    HALT
```



- *Program does not modify itself*
- *Efficiency has improved dramatically (ops / iter)*

	with index regs	without index regs
instruction fetch	5(2)	17 (14)
operand fetch	2	10 (8)
store	1	5 (4)

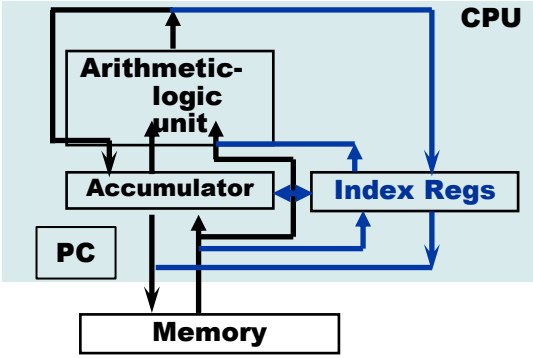
- *Costs:* Instructions are 1 to 2 bits longer
  Index registers with ALU-like circuitry
  Complex control

Operations on Index Registers

To increment index register by k

AC ← (IX) *new instruction*
AC ← (AC) + k
IX ← (AC) *new instruction*

also the AC must be saved and restored.



It may be better to increment IX directly

INCi k, IX IX ← (IX) + k

More instructions to manipulate index register

STOREi x, IX M[x] ← (IX) (extended to fit a word)

…

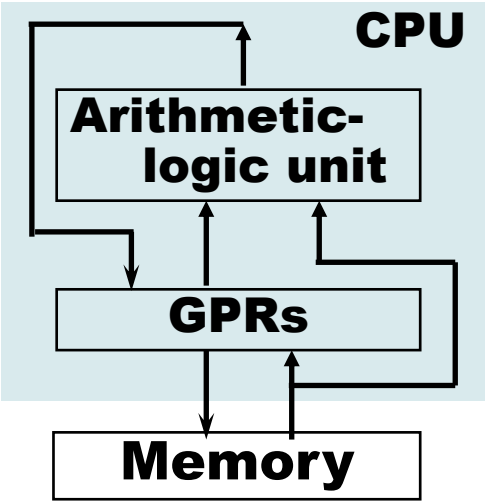
IX begins to look like an accumulator

⇒ several index registers

several accumulators

⇒***General Purpose Registers***

**when technology progress made it practical in the 60's*



Evolution of Addressing Modes

1. Single accumulator, absolute address

   LOAD x

2. Single accumulator, index registers

   LOAD x, IX

3. Indirection

   LOAD (x)

4. Multiple accumulators, index registers, indirection

   LOAD R, IX, x

   or LOAD R, IX, (x) the meaning?

   R <- M[M[x] + (IX)]

   or R <- M[M[x + (IX)]]

5. Indirect through registers

   LOAD R_I, (R_J)

6. With index and base addresses in general-purpose registers

   LOAD R_I, R_J, (R_K) R_J = index, R_K = base addr

Variety of Instruction Formats

- ***One address formats****:* Accumulator machines
  - Accumulator is always other source and destination operand

- ***Two address formats****:* the destination is same as one of the operand sources (e.g., CISC Intel architecture)

(Reg + Reg) to Reg	$R_I \leftarrow (R_I) + (R_J)$
(Reg + Mem) to Reg	$R_I \leftarrow (R_I) + M[\boldsymbol{x}]$

  - ***x*** can be specified directly or via a register
  - effective address calculation for ***x*** could include indexing, indirection, ...

- ***Three address formats****:* One destination and up to two operand sources per instruction (e.g., RISC MIPS architecture)

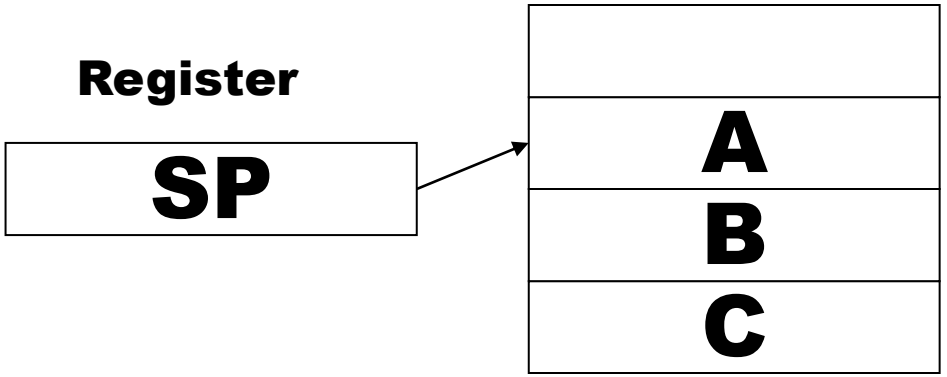
(Reg + Reg) to Reg	$R_I \leftarrow (R_J) + (R_K)$
(Reg + Mem) to Reg	$R_I \leftarrow (R_J) + M[x]$

Zero Address Formats

- Operands on a stack

  add M[sp-1] ← M[sp] + M[sp-1]
  load M[sp] ← M[M[sp]]

  - Stack can be in registers or in memory (usually top of stack cached in registers)



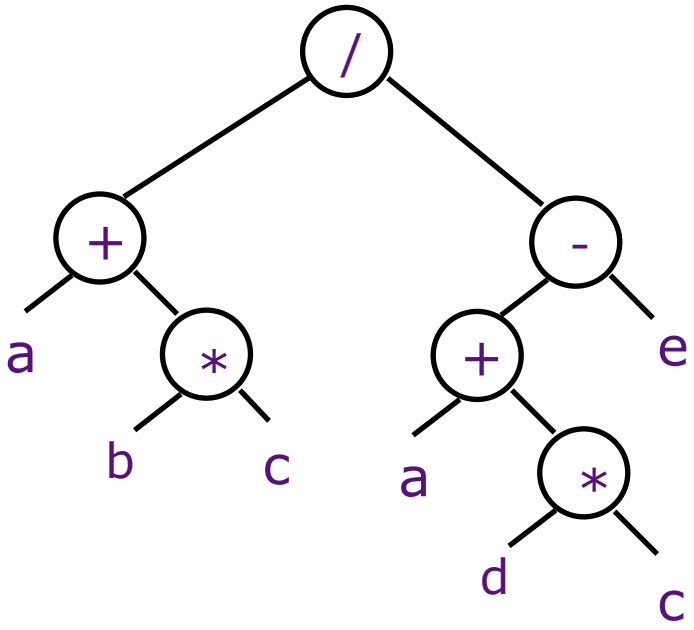
Burrough's B5000 Stack Architecture:

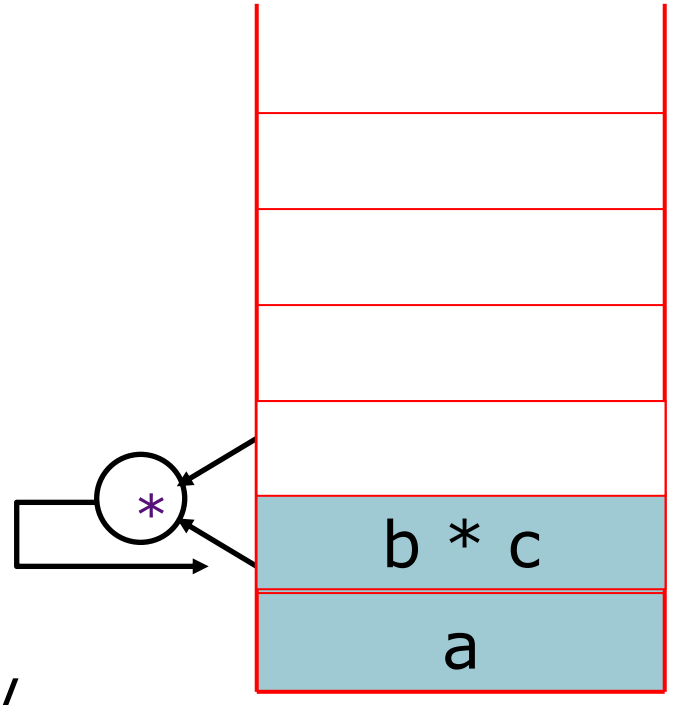
An ALGOL Machine, Robert Barton, 1960

- Machine implementation can be completely hidden if the programmer is provided only a high-level language interface.
- ***Stack machine* organization** because stacks are convenient for:
  1. expression evaluation;
  2. subroutine calls, recursion, nested interrupts;
  3. accessing variables in block-structured languages.
- B6700, a later model, had many more innovative features
  - tagged data
  - virtual memory
  - multiple processors and memories

Evaluation of Expressions

(a + b * c) / (a + d * c - e)





Evaluation Stack

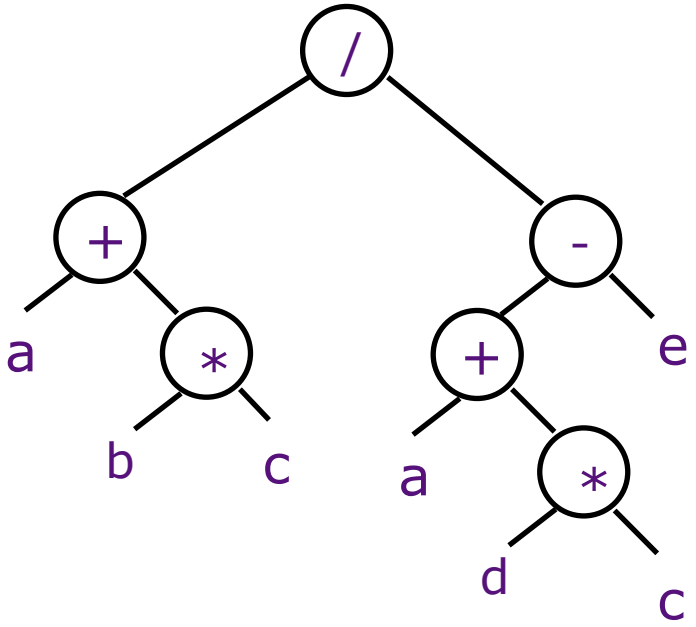
Reverse Polish

a b c * + a d c * + e - /

push a push b push c multiply

Evaluation of Expressions

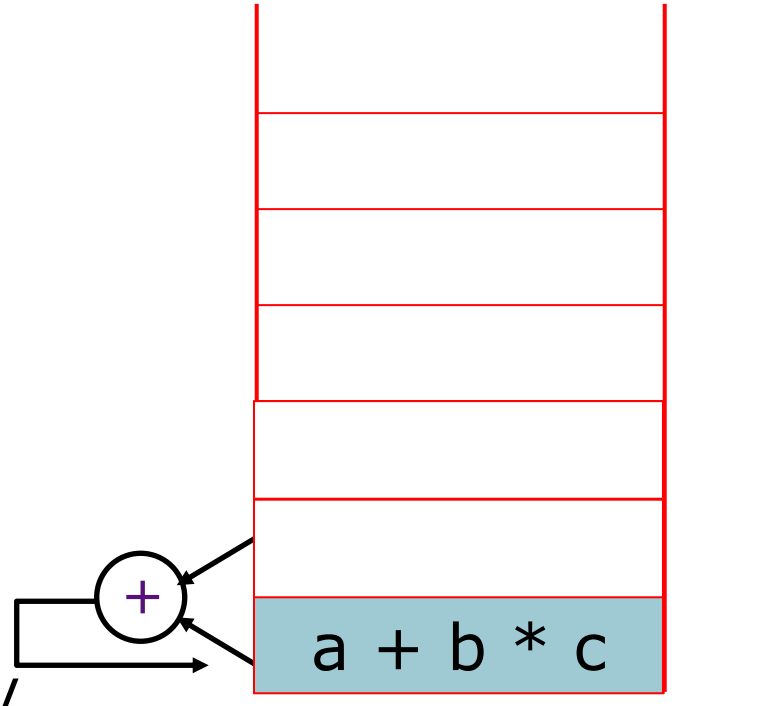
(a + b * c) / (a + d * c - e)



Reverse Polish

a b c * + a d c * + e - /

add



Evaluation Stack

Hardware Organization of the Stack

- Stack is part of the processor state
  - ⇒ *stack must be bounded and small*
    number of Registers,
    not the size of main memory

- Conceptually stack is unbounded
  - ⇒ *a part of the stack is included in the processor state; the rest is kept in the main memory*

Stack Operations and Implicit Memory References

- Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

Each *push* operation⇒ 1 memory reference
pop operation ⇒ 1 memory reference

No Good!

- Better performance by keeping the top N elements in registers, and memory references are made only when register stack overflows or underflows.

Stack Size and Expression Evaluation

a b c * + a d c * + e - /

a and c are "loaded" twice

⇒ *not the best use of registers!*

program	*stack (size = 4)*
push a	R0
push b	R0 R1
push c	R0 R1 R2
*	R0 R1
+	R0
push a	R0 R1
push d	R0 R1 R2
push c	R0 R1 R2 R3
*	R0 R1 R2
+	R0 R1
push e	R0 R1 R2
-	R0 R1
/	R0

Register Usage in a GPR Machine

(a + b * c) / (a + d * c - e)

	Load	R0	a
	Load	R1	c
	Load	R2	b
Reuse R2	Mul	R2	R1
	Add	R2	R0
	Load	R3	d
Reuse R3	Mul	R3	R1
	Add	R3	R0
	Load	R0	e
Reuse R0	Sub	R3	R0
	Div	R2	R3

More control over register usage since registers can be named explicitly

Load	Ri m
Load	Ri (Rj)
Load	Ri (Rj) (Rk)

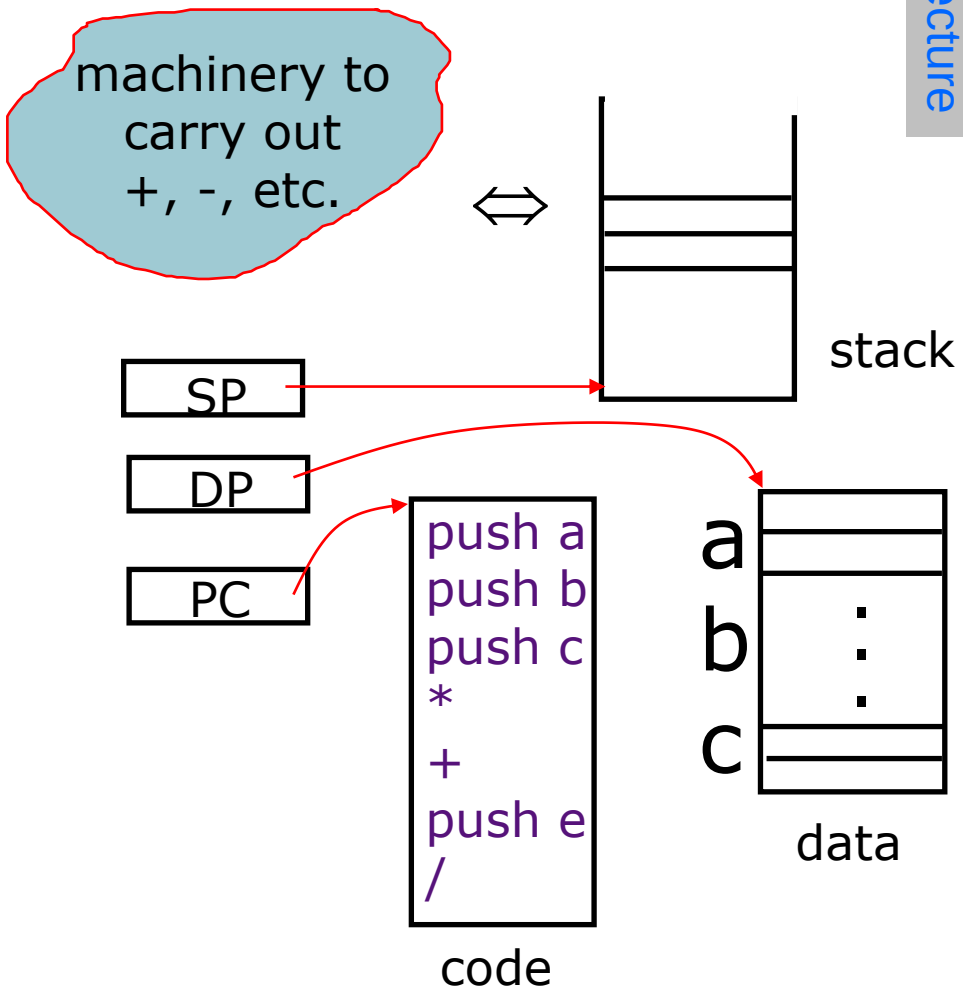
$\Rightarrow$

- *eliminates unnecessary Loads and Stores*
- *fewer Registers*

but instructions may be longer!

Stack Machines: Essential features

- In addition to push, pop, + etc., the instruction set must provide the capability to
  - *refer to any element in the data area*
  - *jump to any instruction in the code area*
  - *move any element in the stack frame to the top*



Stack Machines (Mostly) Died by 1980

1. Stack programs are not smaller if short (Register) addresses are permitted.

2. Modern compilers can manage fast register space better than the stack discipline.

Early language-directed architectures often did not take into account the role of compilers!

B5000, B6700, HP 3000, ICL 2900, Symbolics 3600

Stacks post-1980

- Inmos Transputers (1985-2000)
  - Designed to support many parallel processes in Occam language
  - Fixed-height stack design simplified implementation
  - Stack trashed on context swap (fast context switches)
  - Inmos T800 was world's fastest microprocessor in late 80's
- Forth machines
  - Direct support for Forth execution in small embedded real-time environments
  - Several manufacturers (Rockwell, Patriot Scientific)
- Java Virtual Machine
  - Designed for software emulation, not direct hardware execution
  - Sun PicoJava implementation + others
- Intel x87 floating-point unit
  - Severely broken stack model for FP arithmetic
  - Deprecated in Pentium-4, replaced with SSE2 FP registers

Compatibility Problem at IBM

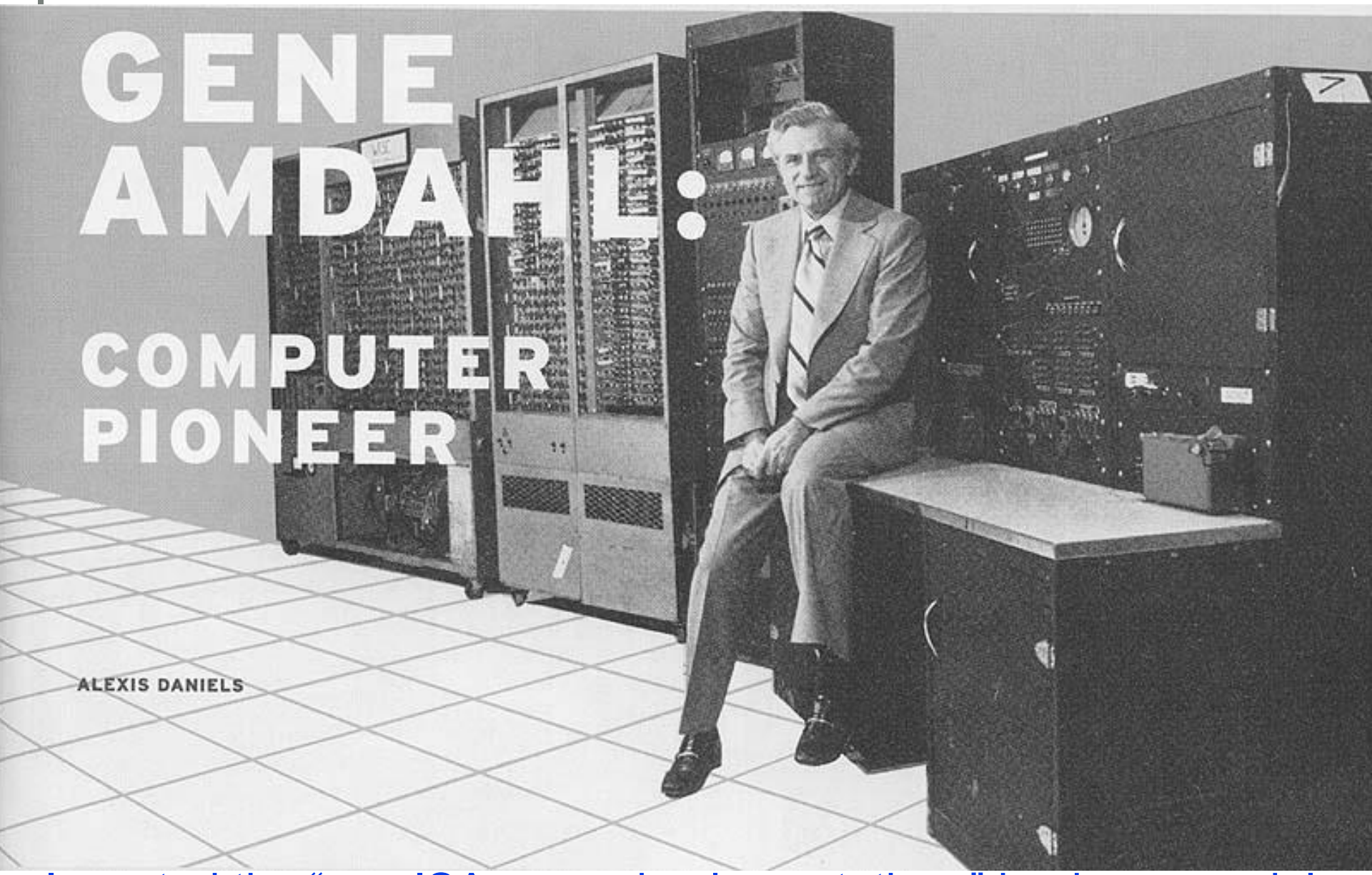
By early 60's, *IBM had 4 incompatible lines of computers!*

701	→	7094
650	→	7074
702	→	7080
1401	→	7010

Each system had its own

- Instruction set
- I/O system and Secondary Storage:
  magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche
  business, scientific, real time, ...

⇒ *IBM 360*



Invented the “one ISA, many implementations” business model.

IBM 360 : Design Premises
Amdahl, Blaauw and Brooks, 1964

- The design must lend itself to growth and successor machines
- General method for connecting I/O devices
- Total performance - answers per month rather than bits per microsecond
- Machine must be capable of supervising itself without manual intervention
- Built-in hardware fault checking and locating aids to reduce down time
- Simple to assemble systems with redundant I/O devices, memories etc. for fault tolerance
- Some problems required floating-point larger than 36 bits

IBM 360: *A General-Purpose Register (GPR) Machine*

- Processor State
  - 16 General-Purpose 32-bit Registers
    - *may be used as index and base register*
    - *Register 0 has some special properties*
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - *PC, Condition codes, Control flags*
- A 32-bit machine with 24-bit addresses
  - But no instruction contains a 24-bit address!
- Data Formats
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words

The IBM 360 is why bytes are 8-bits long today!

IBM 360: Initial Implementations

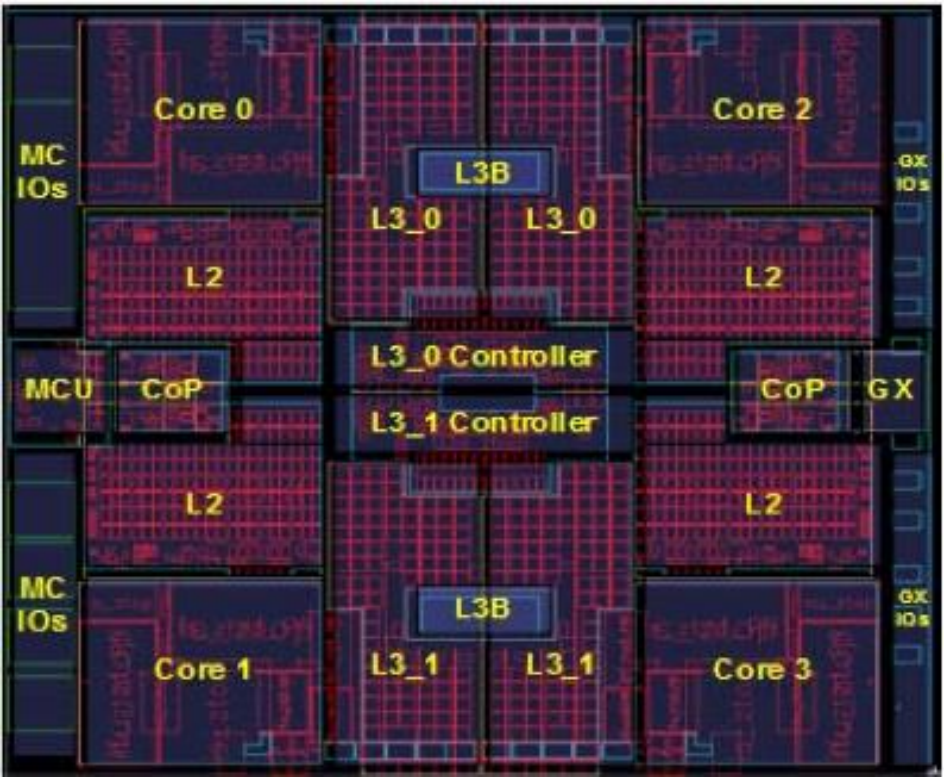
	Model 30	. . .	*Model 70*
Storage	8K - 64 KB		256K - 512 KB
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Local Store	Main Store		Transistor Registers
Control Store	Read only 1μsec		Conventional circuits

IBM 360 instruction set architecture (ISA) completely hid the underlying technological differences between various models.

***Milestone**: The first true ISA designed as portable hardware-software interface!*

With minor modifications it still survives today!

IBM 360: 47 years later…
The zSeries z11 Microprocessor

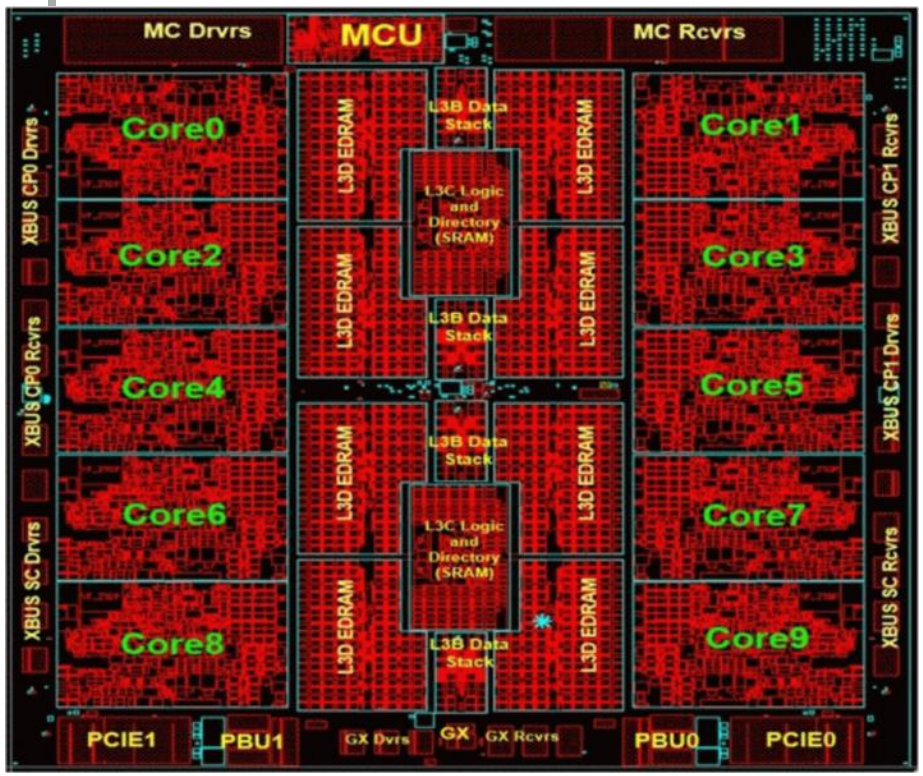


[IBM, HotChips, 2010]

- 5.2 GHz in IBM 45nm PD-SOI CMOS technology
- 1.4 billion transistors in 512 mm^2
- 64-bit virtual addressing
  - original S/360 was 24-bit, and S/370 was 31-bit extension
- Quad-core design
- Three-issue out-of-order superscalar pipeline
- Out-of-order memory accesses
- Redundant datapaths
  - every instruction performed in two parallel datapaths and results compared
- 64KB L1 I-cache, 128KB L1 D-cache on-chip
- 1.5MB private L2 unified cache per core, on-chip
- On-Chip 24MB eDRAM L3 cache
- Scales to 96-core multiprocessor with 768MB of shared L4 eDRAM

IBM 360: z14 in 2017

The zSeries z14 (PU) Microprocessor



[Launched on July 17, 2017]

- 5.2 GHz in IBM 14nm FinFet-SOI CMOS technology
- 7.1 billion transistors in 696 mm^2
- 64-bit virtual addressing
  - original S/360 was 24-bit, and S/370 was 31-bit extension
- 10-core design
- Three-issue out-of-order superscalar pipeline
- Out-of-order memory accesses
- 128KB L1 I-cache, 128KB L1 D-cache on-chip
- 4MB private L2 unified cache per core, on-chip
- On-Chip 128MB eDRAM L3 cache
- Scales to 170-core multiprocessing with 672MB of shared L4 eDRAM
- Central Processor Assist for Cryptographic Function (CPACF)
  - Dedicated co-processor for each core
  - 4x Advanced Encryption Standard (AES) speedup
  - Support for True Random Number Generator
  - New support for SHA-3 standard

Data Types

Bit: 0, 1

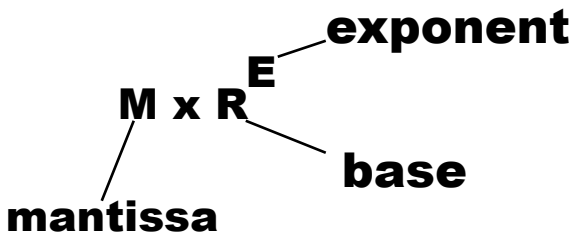
Bit String: sequence of bits of a particular length
- 4 bits is a nibble
- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word
- 128 bits is a quad-word

Character:
- ASCII 7 bit code
- UNICODE 16 bit code

Decimal:
- digits 0-9 encoded as 0000b thru 1001b
- two decimal digits packed per 8 bit byte

Integers:
- 2's Complement

Floating Point:
- Single Precision
- Double Precision
- Extended Precision



How many +/- numbers?
Where is decimal point?
How are +/- exponents represented?

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

2s-Complement Signed Negation

- Ones' complement (invert bits) and add 1
  - Ones' complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Top 10 80x86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

- Simple instructions dominate instruction frequency

Operation Summary

Support these simple instructions, since they will dominate the number of instructions executed:

load,
store,
add,
subtract,
move register-register,
and,
shift,
compare equal, compare not equal,
branch,
jump,
call,
return;

Methods of Testing Condition

- Condition Codes

  Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

  ex: add r1, r2, r3

  bz label

- Condition Register

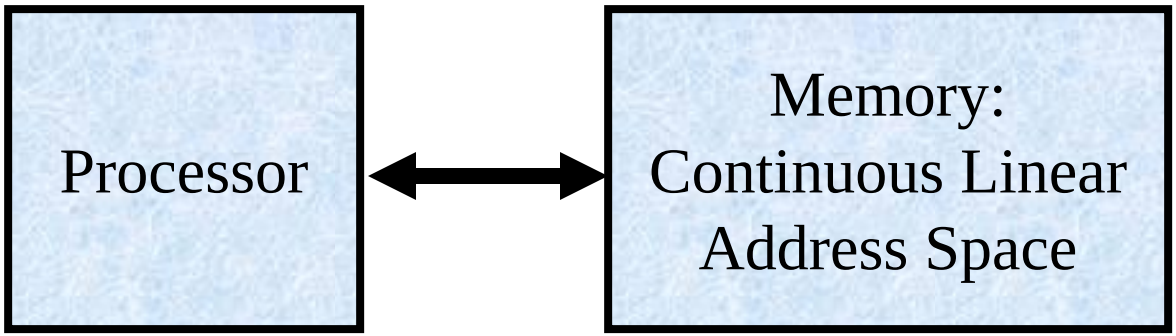
  Ex: cmp r1, r2, r3

  bgt r1, label

- Compare and Branch

  Ex: bgt r1, r2, label

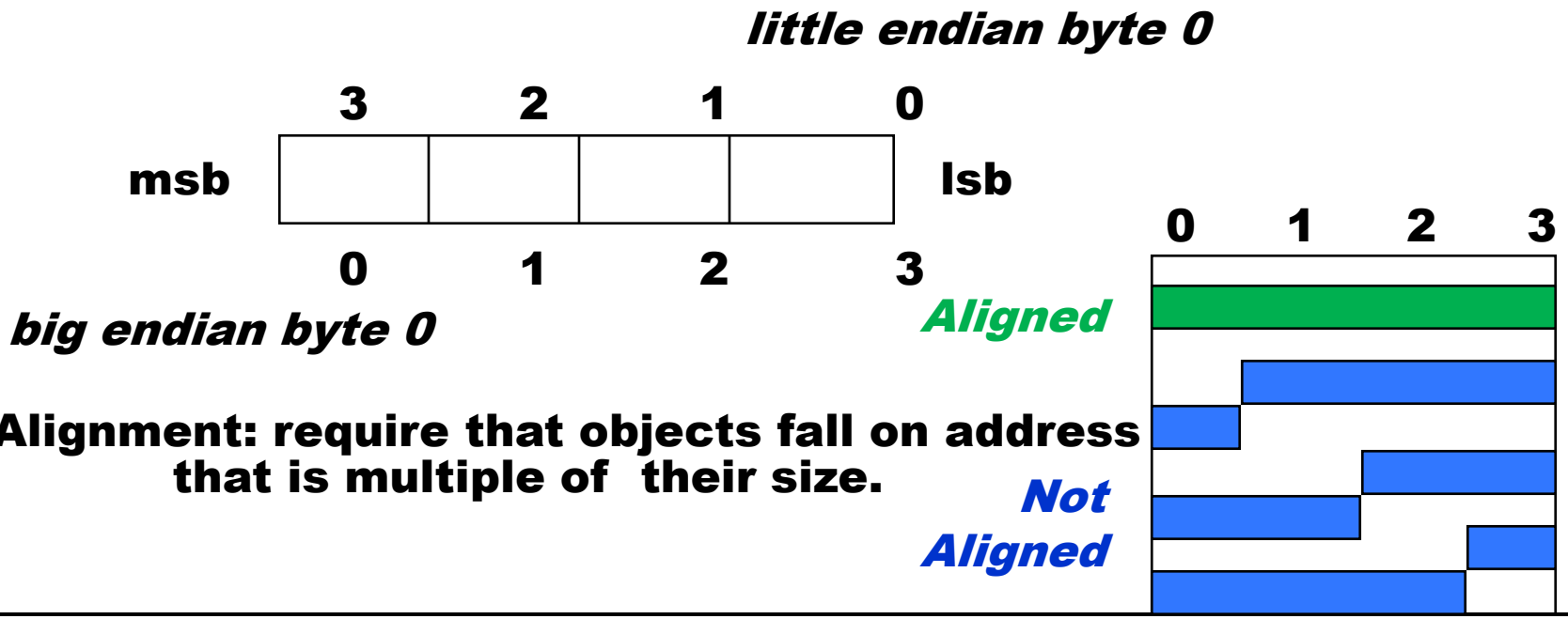
Memory Addressing



- Since 1980 almost every machine uses addresses to level of 8-bits (byte)
- 2 questions for design of ISA:
  - How do byte addresses map onto words?
  - Can a word be placed on any byte boundary?

Addressing Objects: Endianess and Alignment

- **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)



Alignment: require that objects fall on address that is multiple of their size.

Addressing Modes

Addressing mode	Example	Meaning
Register	Add R4,R3	R4← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Post-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Pre-decrement	Add R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

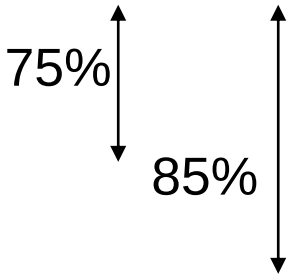
Why Post-increment/Pre-decrement? Scaled?

Addressing Modes

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3` w += i	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3` w += 3	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)` w += a[100 + i]	`Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)` w += a[i]	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)` w += a[i + j]	`Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)` w += a[1001]	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)` w += a[*p]	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+` a[i++]	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*.
Autodecrement	`Add R1, –(R2)` a[i--]	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]` w += a[100 + i + d*j]	`Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Addressing Mode Usage?

3 programs measured on machine with all address modes (VAX)



--- **Displacement:** 42% avg, 32% to 55%

--- **Immediate:** 33% avg, 17% to 43%

--- **Register deferred (indirect):** 13% avg, 3% to 24%

--- **Scaled:** 7% avg, 0% to 16%

--- **Memory indirect**: 3% avg, 1% to 6%

--- **Misc:** 2% avg, 0% to 3%

75% displacement & immediate
85% displacement, immediate & register indirect

Reduced Instruction Set Computing

- The early trend was to add more and more instructions to do elaborate operations – this became known as *Complex Instruction Set Computing* (CISC)
- Opposite philosophy later began to dominate: *Reduced Instruction Set Computing* (RISC)
  - Simpler (and smaller) instruction set makes it easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

Common RISC Simplifications

- **Fixed instruction length:**
  Simplifies fetching instructions from memory
- **Simplified addressing modes:**
  Simplifies fetching operands from memory
- **Few and simple instructions in the instruction set:**
  Simplifies instruction execution
- **Minimize memory access instructions (load/store):**
  Simplifies necessary hardware for memory access
- **Let compiler do heavy lifting:**
  Breaks complex statements into multiple assembly instructions

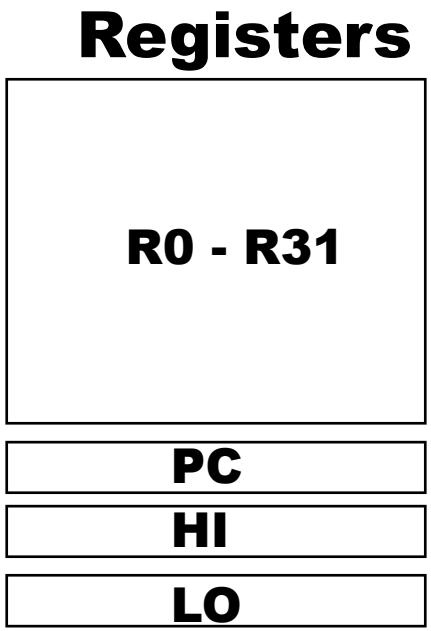
A "Typical" RISC ISA

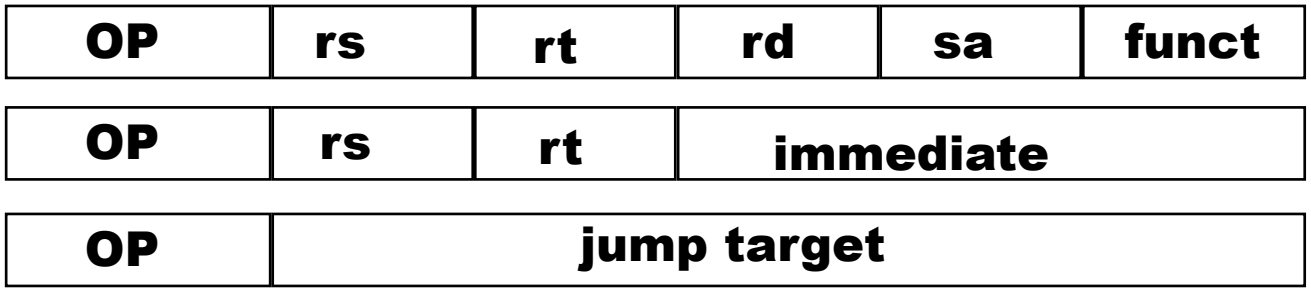
- 32-bit fixed format instruction (3 formats)
- 32 32- or 64-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

MIPS R3000 Instruction Set Architecture

- Register Set
  - 32 general 32-bit registers
  - Register zero ($R0) always zero
  - Hi/Lo for multiplication/division
- Instruction Categories
  - Load/Store
  - Computational
    - Integer/Floating point
  - Jump and Branch
  - Memory Management
  - Special
- 3 Instruction Formats: all 32 bits wide



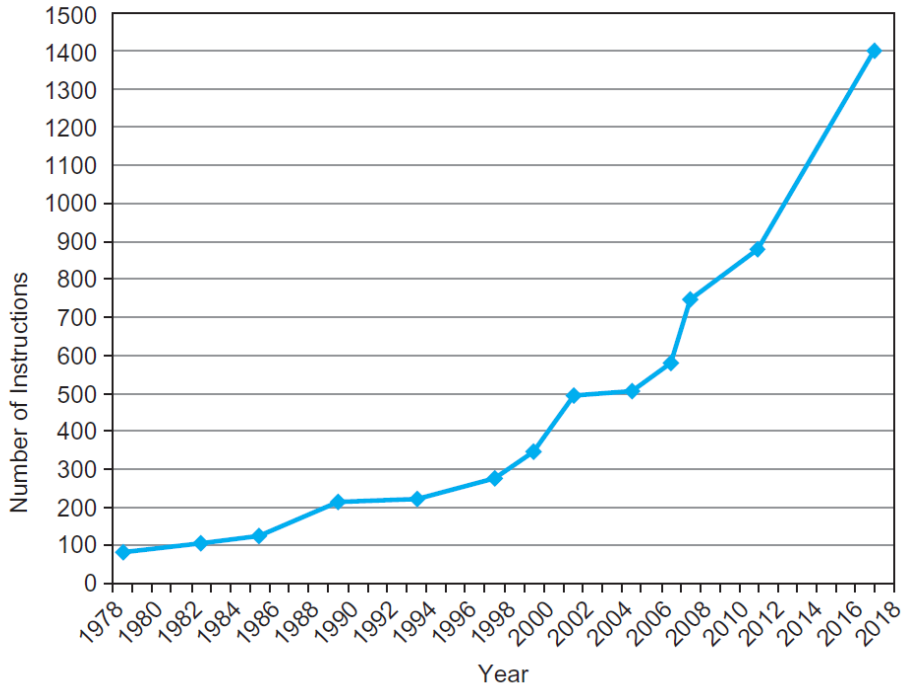


Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do result in more instructions



x86 instruction set

Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Mikhail Dorojevets (SBU)