Computer Architecture I

CS-GY-6133

Topic: Introduction and Instruction Set Architectures (ISA)

Instructor: Siddharth Garg

Logistics

- Lectures
  - Fridays, 6PM-8:30PM, Pfizer Auditorium, Dibner
  - Quizzes/exercises to keep you involved (bring your laptops!)
  - 15 minute break to keep you refreshed
  - **Tweet your questions @csgy6133 during class**
- Office hours
  - Fridays 1PM-3PM
  - 2 Metrotech, 10.076 *or* **Skype ID siddharth.j.garg**
- TAs
  - Team of 6 TAs. Names and office hours will be announced soon.

Grading (HWs and Labs)

- 3 Homeworks (20% of your final grade)
  - First HW to be released 9/16, due 9/30
  - Groups of two
- 3 programming labs (20% of your final grade)
  - First lab. to be released 9/30; due in 2 weeks
  - Groups of two (same as HW groups)
- Important: submit your partner's name before next class on NYUClasses (under "Assignments")

Grading (Midterm and Final)

- Midterm scheduled for Nov. 11th during class hours (25% of total grade)
  - Location will be announced later
- Final exam scheduled for Dec. 16th during class hours (35% of total grade)
  - Note: officially, final exam is scheduled for Dec. 23rd but kills your (and my) Christmas break
  - Location will be announced later

Plagiarism

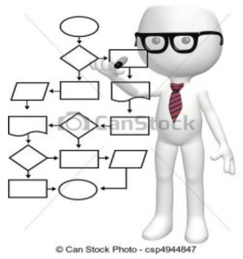
- Don’t do it
- Don’t do it
- Don’t do it
- Don’t do it
- Don’t do it
- Don’t do it
- Don’t do it
- Don’t do it

- Why not?
  - You won’t learn much
  - You’ll likely get caught
- What happens if I get caught?
  - Lose 50% of your total course grade on first strike
  - Second strike and you fail the class

Introduction

Algorithm

Step-by-step set of operations to solve a certain problem



Problem-oriented Programming Language

Examples: C/C++, Java, Python etc.

Compiler

Assembly Language

Step-by-step instructions that *your computer can understand*!

Instruction Set Archite SA)

Examples:

Many differ ys of executing the ame ISA

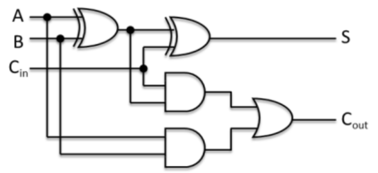
xamples: ....?

Computer Architecture

Micro-architecture

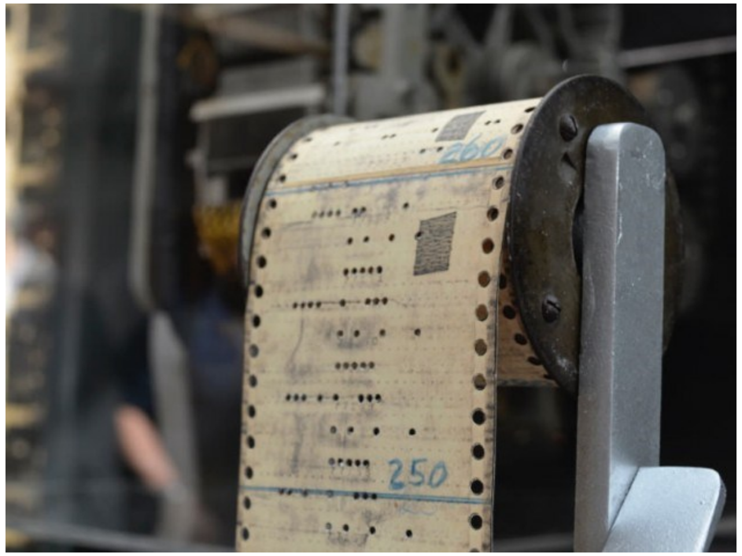
Hardware-level description of how processor executes ISA

Digital Logic/Hardware



Boolean logic gates, flip-flops/registers

Historical Perspective: Harvard Mark I



- 60 "constant registers" (23 decimals + sign)
  - Manually programmed constants
- 72 "storage counters" (23 decimals + sign)
  - To hold intermediate results of computation
- Instruction Format
  - 24 decimals
  - Fed with punch card

OUT (operand)	IN (operand, result stored here)	MISC. (which operation is performed)
Storage Counter 1 (Decimal Code: 1)	Storage Counter 2 (Decimal Code: 2)	SUB (Decimal Code: 32)

$$SC2 \leftarrow SC2 - SC1$$

No branch instructions!

- "Harvard Architecture" - separate memory for instructions and data

EDVAC: Electronic Discrete Variable Automatic Computer

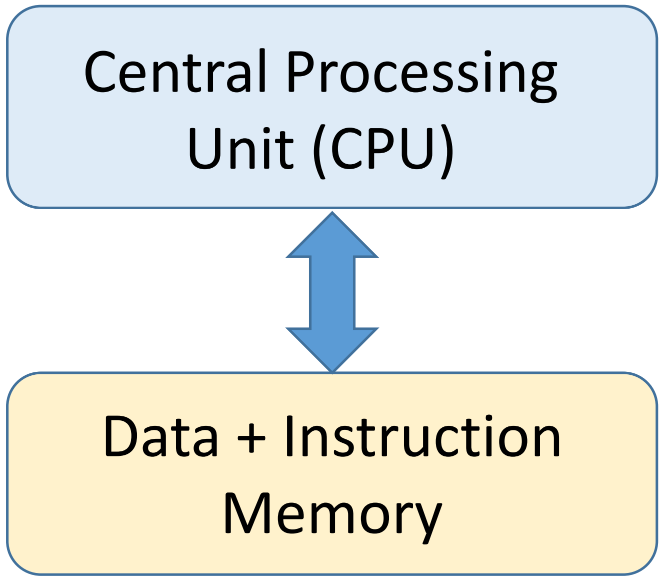
- Internal magnetic "tape" based memory consisting of 1024 44-bit words
  - Digital (0,1) instead of decimal (0,1,2,..9) representation
  - Each word could represent either data or an instruction
  - Up to 16 different instructions, including branch instructions

Op Code (4 bits)	OP 1 (10 bits)	OP 2 (10 bits)	OP 3 (10 bits)	Next Instruction
Which operation (add, sub, mult, branch…)	Address of 1st operand	Address of 2nd operands	Address where result stored	Address from where next instruction fetched

- Von Neumann Architecture: Unified memory for data and instructions

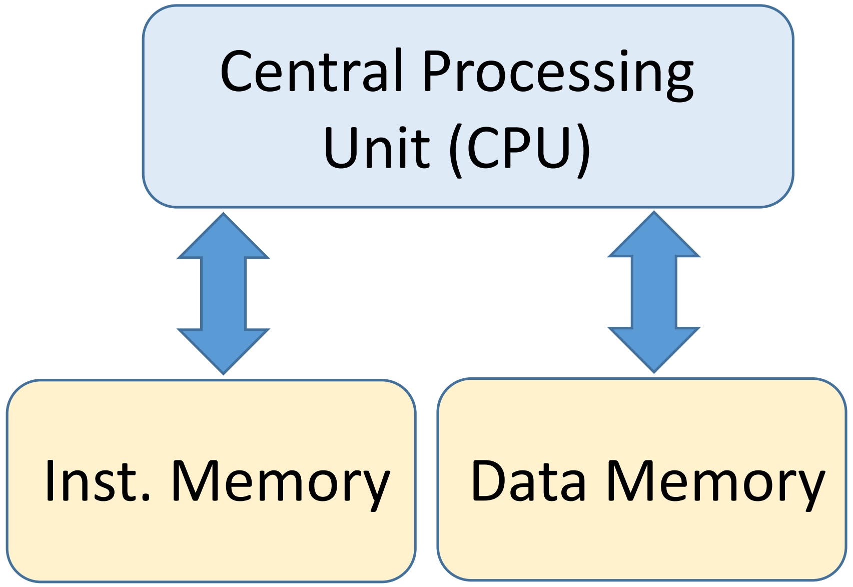
Harvard Vs. Von Neumann Architecture

Von Neumann



- Memory must be a RAM (random access memory) with read/write capability

Harvard



- Data memory must be a RAM (random access memory) with read/write capability
- Instruction memory can be a ROM (read only memory)

Harvard Vs. Von Neumann: Pros/Cons

- Harvard Architecture
  - Con: dedicated memory for code results in inefficient use of memory capacity
  - Pro: Code memory can be read only (faster, smaller, cheaper)
  - Pro: code is protected

- Von Neumann Architecture
  - Pro: flexible use of available memory capacity
  - Con: Memory needs to support read/write
  - Con: code can be maliciously re-written

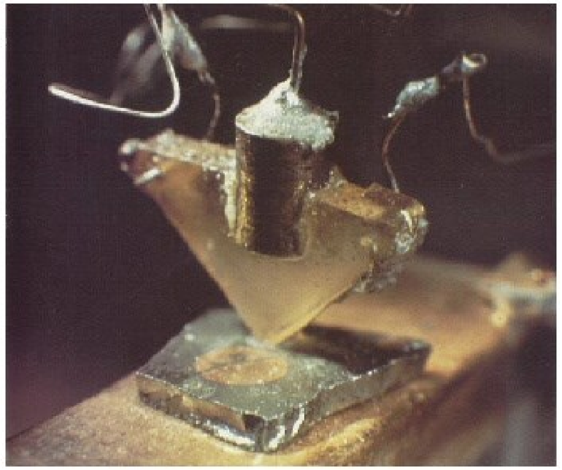
"Lecture 1: In-class Problem 1" on newclasses.nyu.edu under "Assignments"

Invention of the Transistor

- Vacuum tubes ruled in first half of 20th century Large, expensive, power-hungry, unreliable

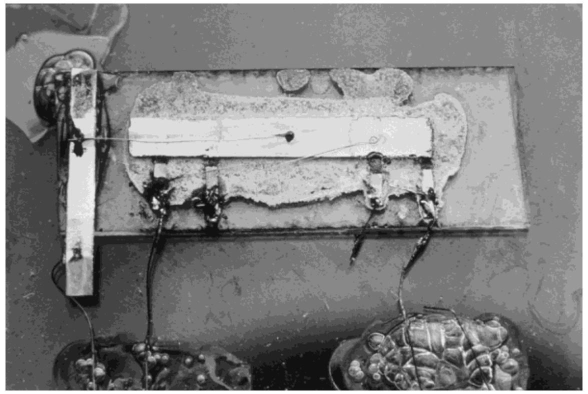
ENIAC had 18000 vacuum tubes and consumed 150 kW of power

- 1947: first point contact transistor
  - John Bardeen and Walter Brattain at Bell Labs



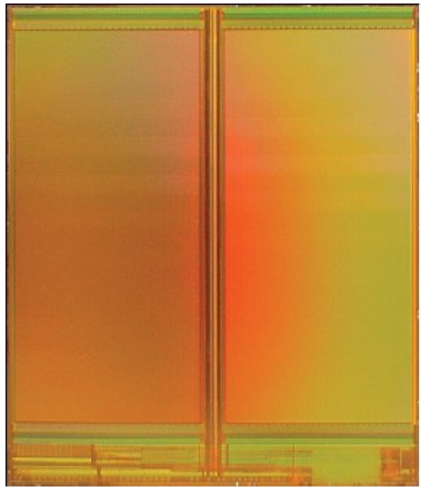
From Transistors to Integrated Circuits

- 1958: First integrated circuit
  - Flip-flop using two transistors
  - Built by Jack Kilby at Texas Instruments



Courtesy Texas Instruments

- 2010
  - Intel Core i7 μprocessor
    - 2.3 billion transistors
  - 64 Gb Flash memory
    - > 16 billion transistors

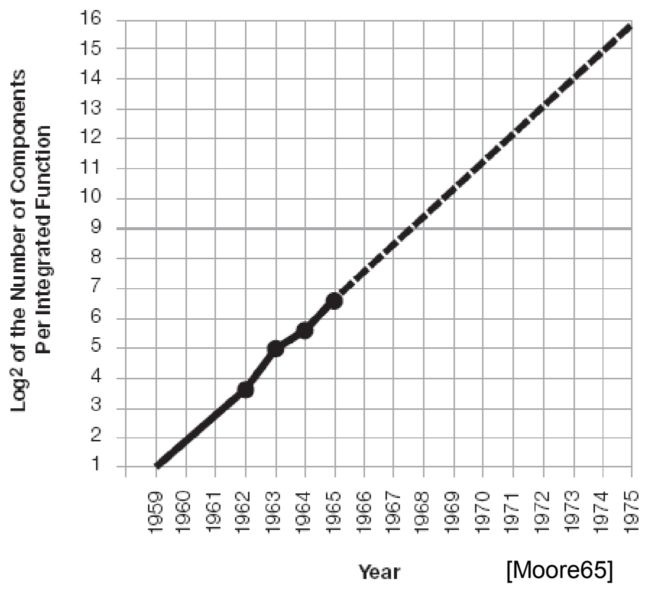


[Trinh09]

© 2009 IEEE

Moore’s Law: Then

- 1965: Gordon Moore plotted transistor on each chip
  - Fit straight line on semilog scale
  - Transistor counts have doubled every 26 months



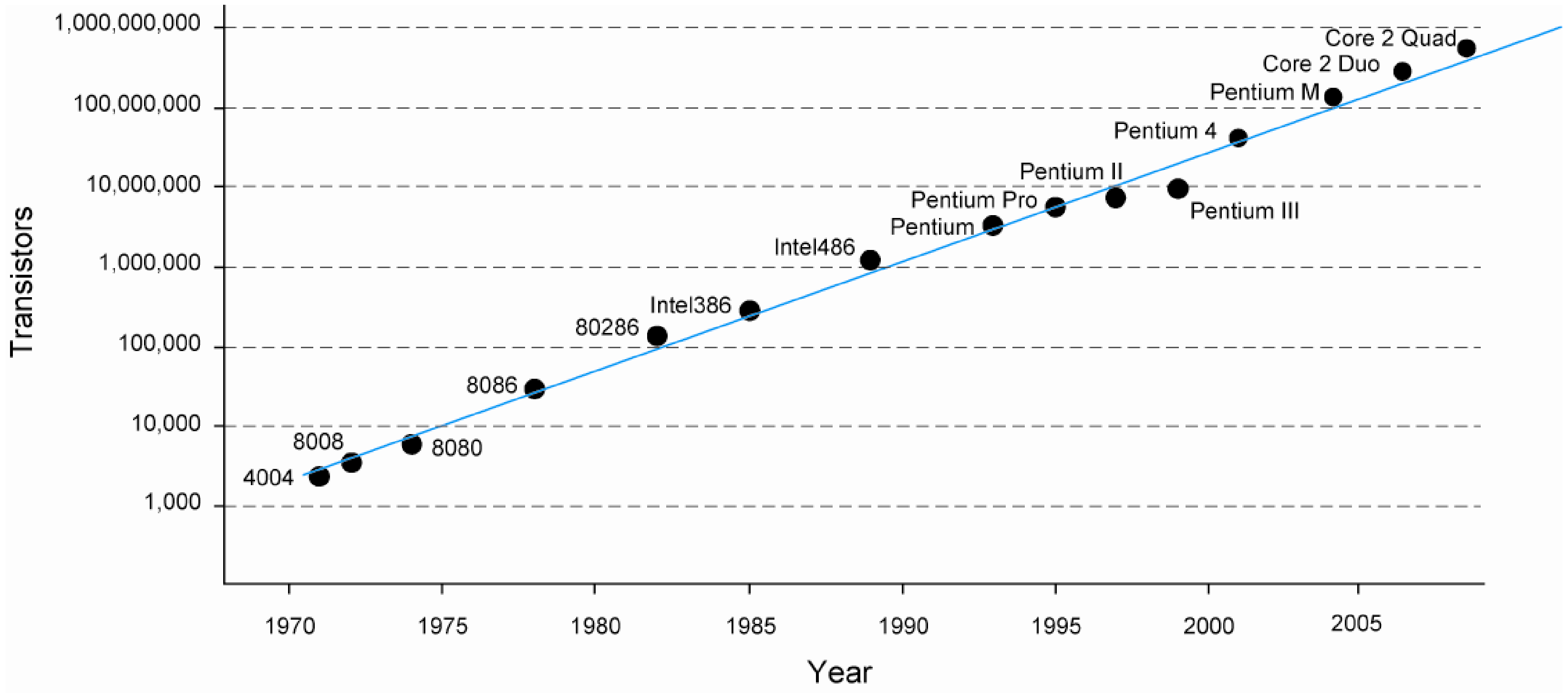
[Moore65]

Electronics Magazine

<u>Integration Levels</u>

SSI:	10 gates
MSI:	1000 gates
LSI:	10,000 gates
VLSI:	> 10k gates

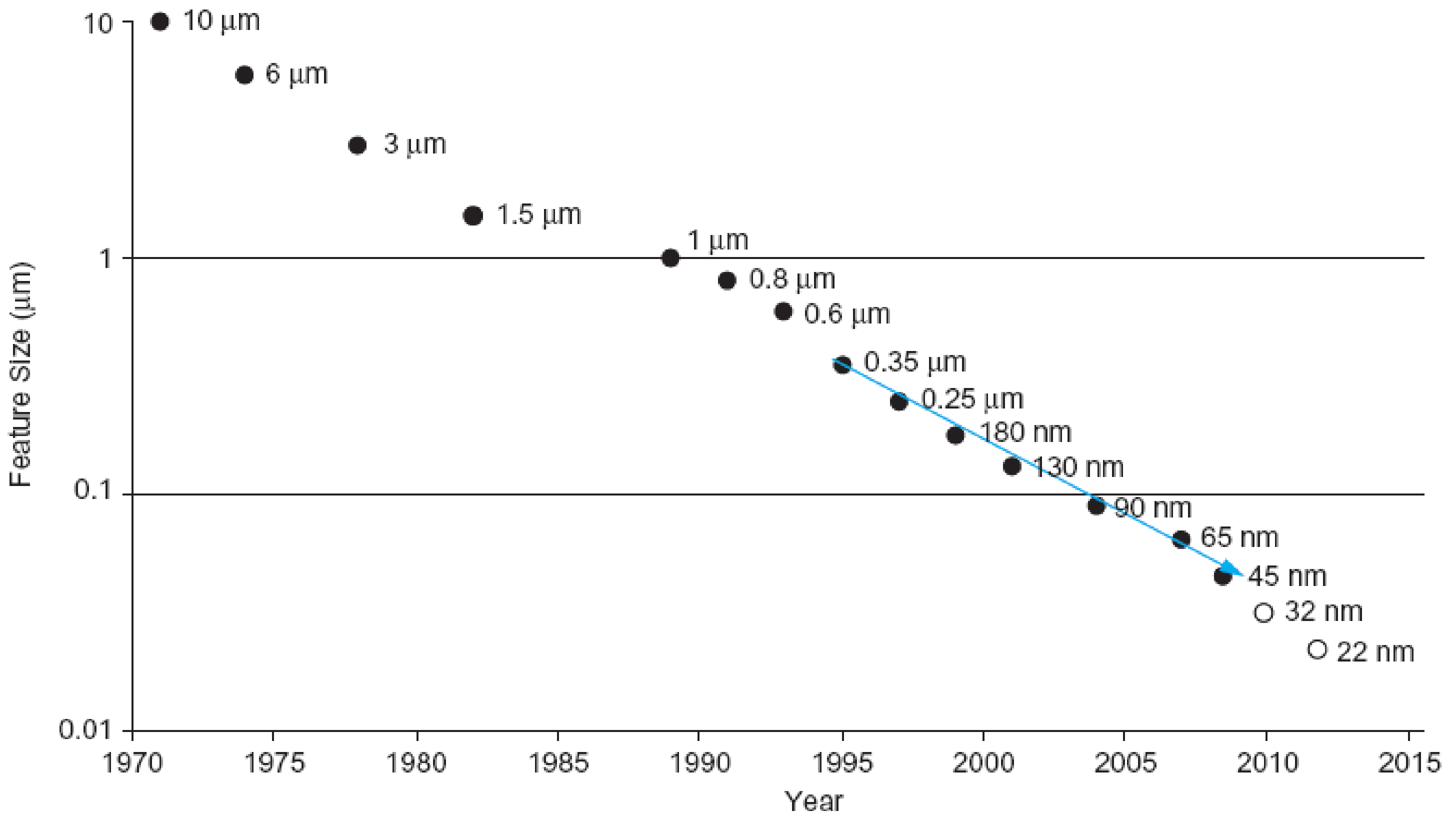
And Now…



Transistors
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
Year
1970
1975
1980
1985
1990
1995
2000
2005
4004
8008
8080
8086
80286
Intel386
Intel486
Pentium
Pentium Pro
Pentium II
Pentium III
Pentium 4
Pentium M
Core 2 Duo
Core 2 Quad

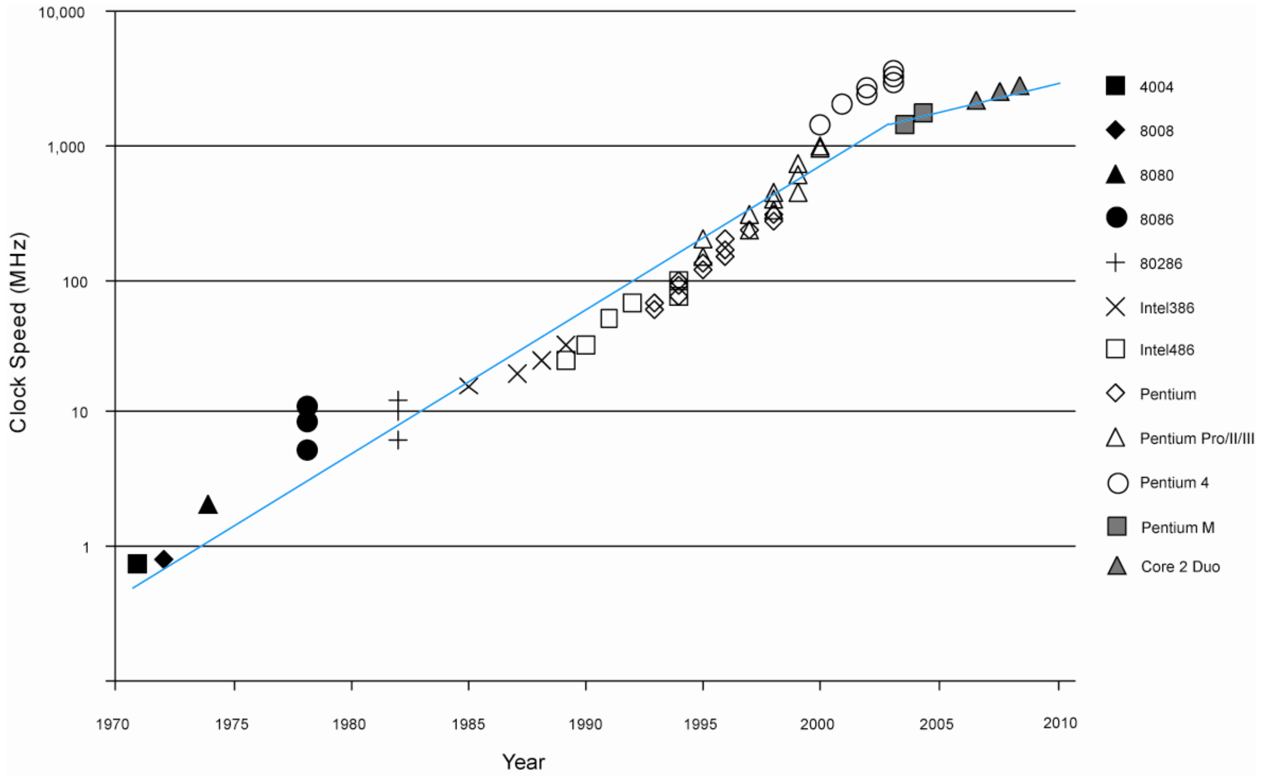
Feature Size

- Minimum feature size shrinking 30% every 2-3 years

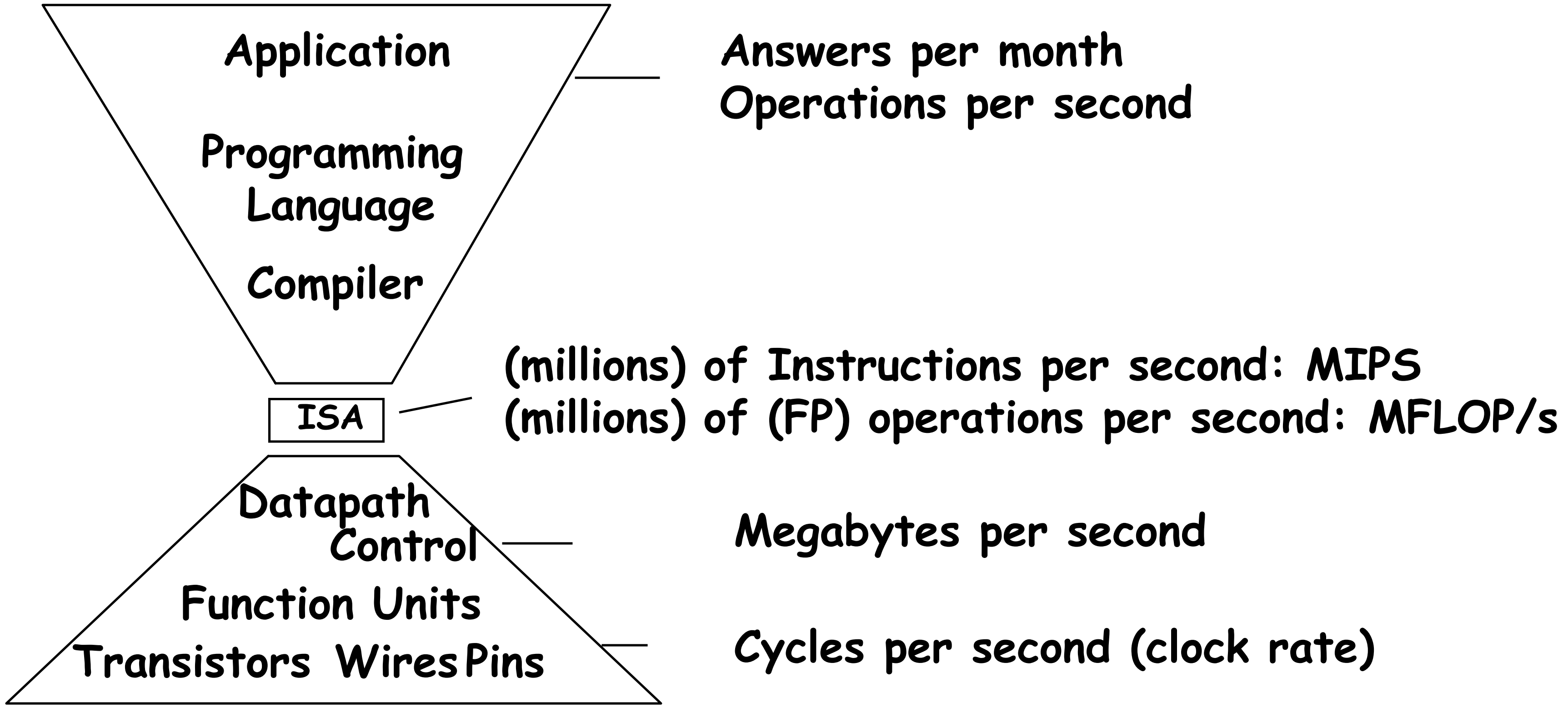


Corollaries

- Many other factors grow exponentially
  - Ex: clock frequency and therefore processor performance

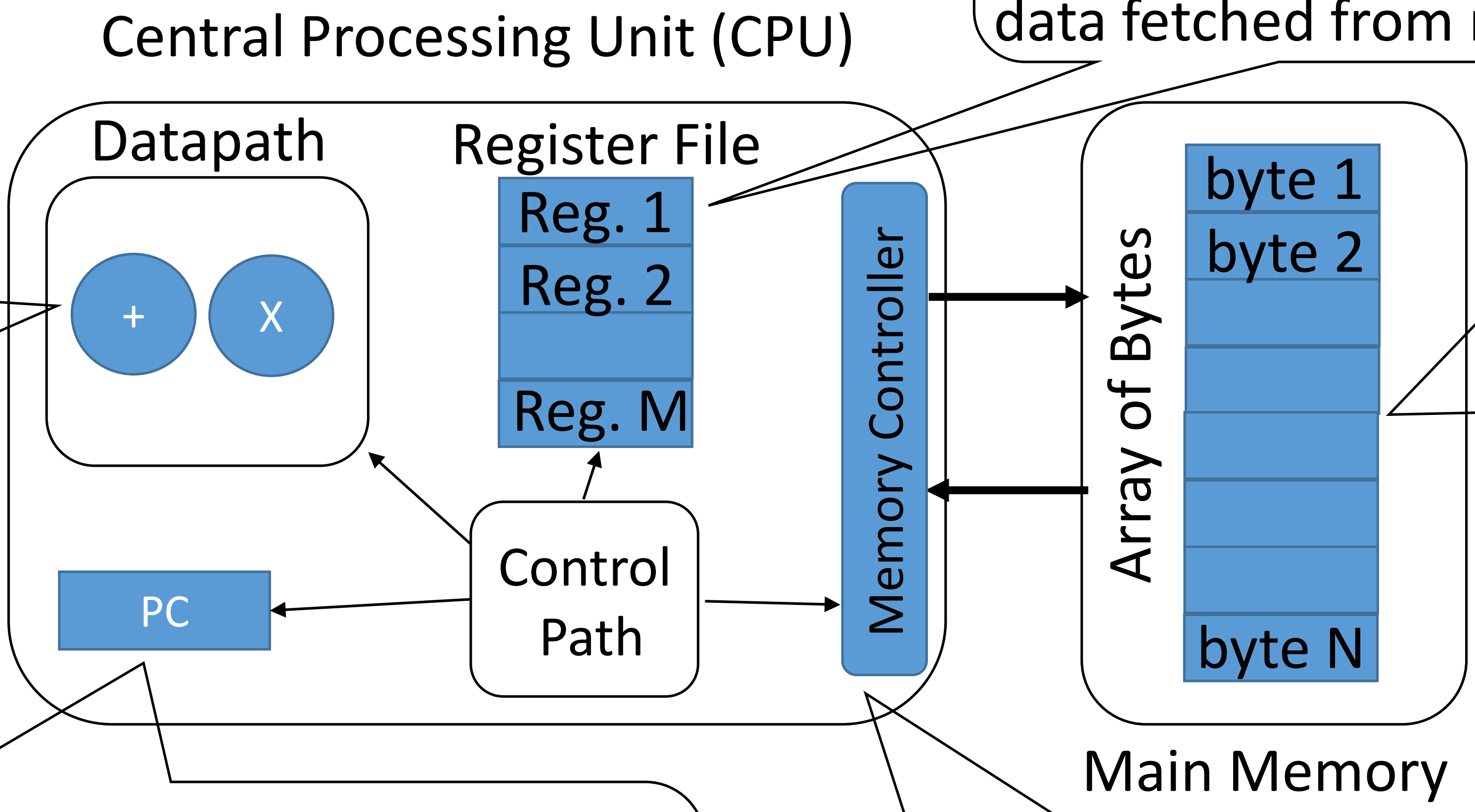


Metrics of Performance



Modern Computers

Fast local access to data. Based on idea of temporal locality: frequent reuse of data fetched from memory



Adders, multipliers etc.

- Unified: stores both instructions and data
- Random access: CPU can read/write any byte of data

Instructions are in processed sequentially (one after the other). Address of current instruction stored in Program counter (PC)

Handles read/write requests from CPU to main memory

Role of the ISA

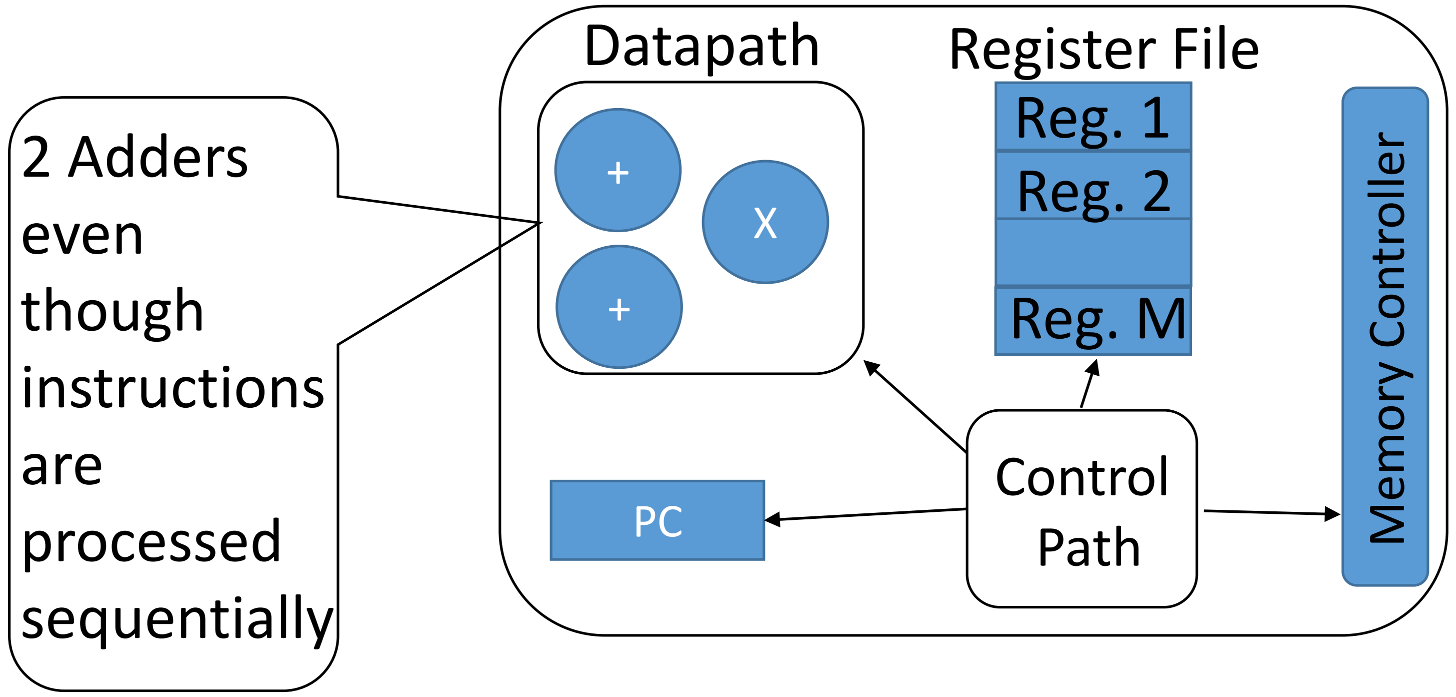
- Instruction Set Architecture (ISA) serves as a contract between programmer and hardware
  - Functional description of storage locations and the operations it can perform

- Storage locations:
  - Registers: typically a small number of registers. Ex: MIPS ISA defines 32 32-bit registers.
  - Memory: much larger, but off-chip and higher access cost. MIPS ISA has a 32-bit memory address, so in theory 2^{32} Bytes = 4GB of data.

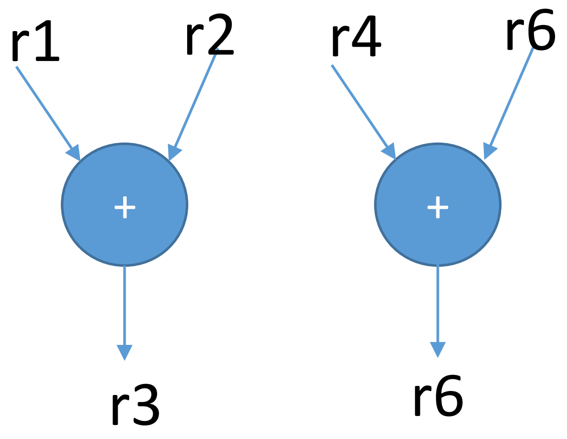
- Operations:
  - Integer Add, Integer Multiply, Branch, FP Add, FP Multiply, FP Divide

Role of the Micro-architecture

- The micro-architecture is responsible for faithfully executing programs written using the ISA
  - Same ISA can have multiple different micro-architectural implementations with different performance and power consumption

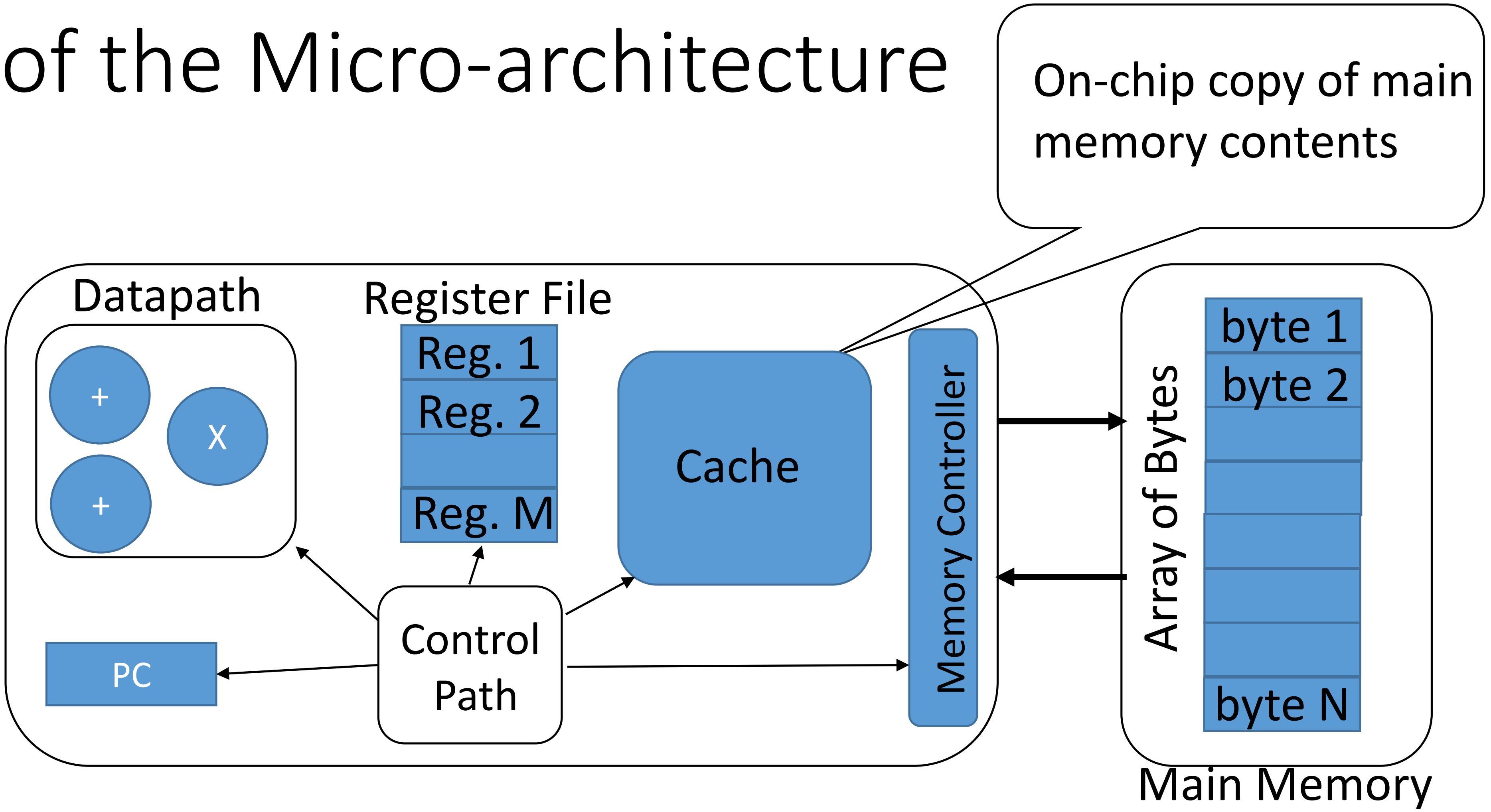


add r3, r1, r2
add r6, r5, r4



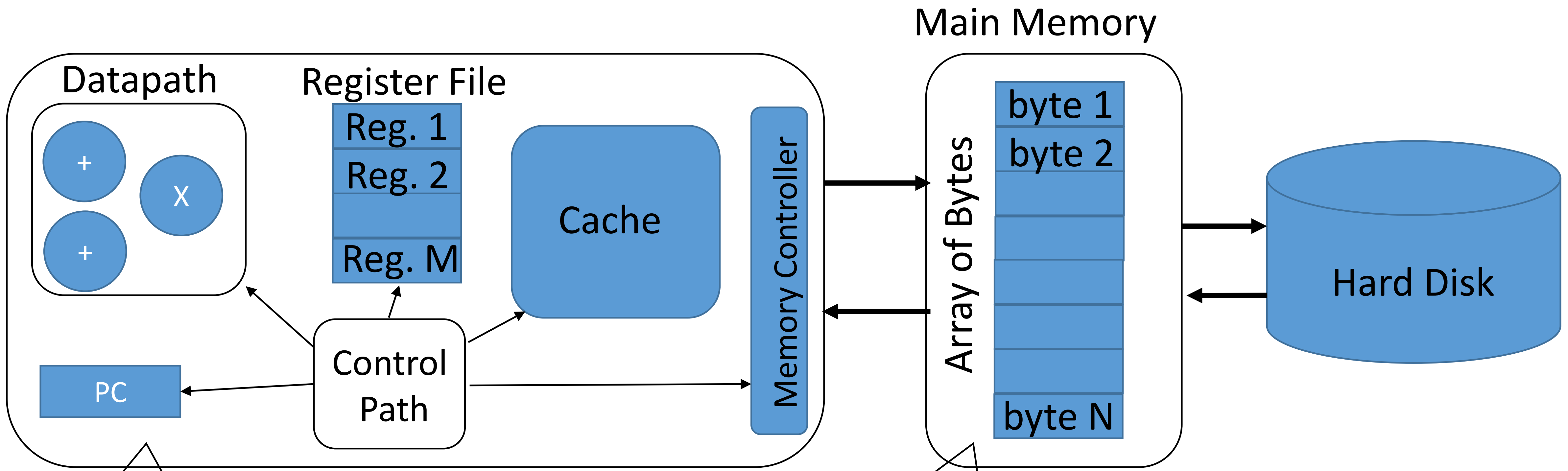
More area/power but higher performance!
(Identical functionality)

Role of the Micro-architecture



- Cache is managed entirely by hardware
- Programmer and oblivious to its existence

Role of the Micro-architecture



32-bit PC implies 4GB of addressable memory

Actual main memory capacity can be <4 GB

- Hardware/OS managed virtual memory provides the appearance of a 4GB contiguous address space

ISAs are Abundant

CISC ISAs

intel

x86
x86-64
IA64

Laptop
Desktop

AMD

x86
x86-64 (AMD64)

Laptop
Desktop

digital
COMPAQ

VAX

Servers

IBM

IBM360

Servers

RISC ISAs

ARM

T32 (Thumb)
A32 (ARM)
A64 (ARM-64)

Mobile

MIPS TECHNOLOGIES

MIPS32
MIPS64

Embedded

Sun microsystems

SPARC

Servers

digital
COMPAQ

Alpha

IBM
MOTOROLA

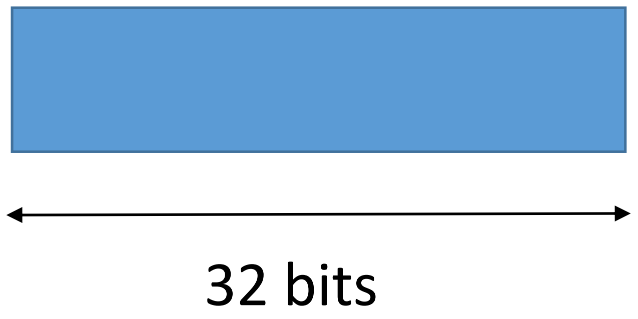
PowerPC

Laptop
Desktop

ISA Design: Instruction Length

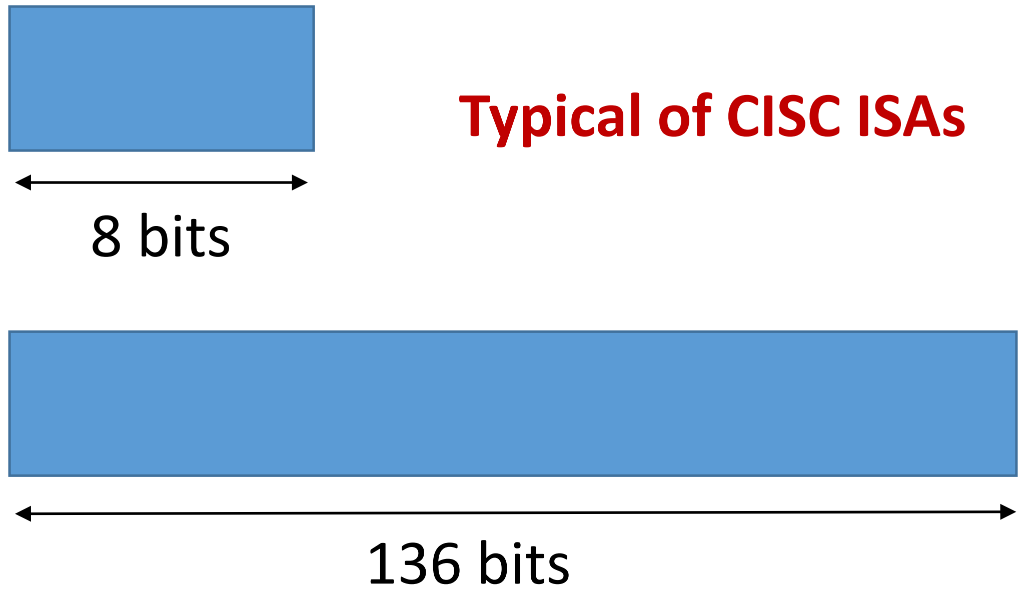
- An instruction is a basic unit of an ISA

- Fixed versus variable length instructions
  - **Fixed**: Each instruction in the MIPS ISA is 32 bits (4 Bytes) long
  - **Variable**: x86 instructions vary from 1 to 17 Bytes long



Typical of RISC ISAs

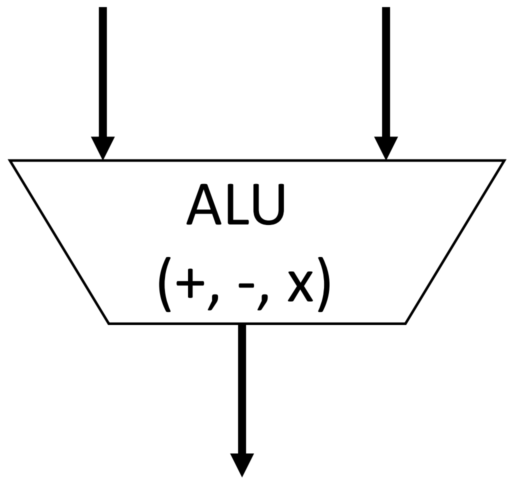
Trade-offs?



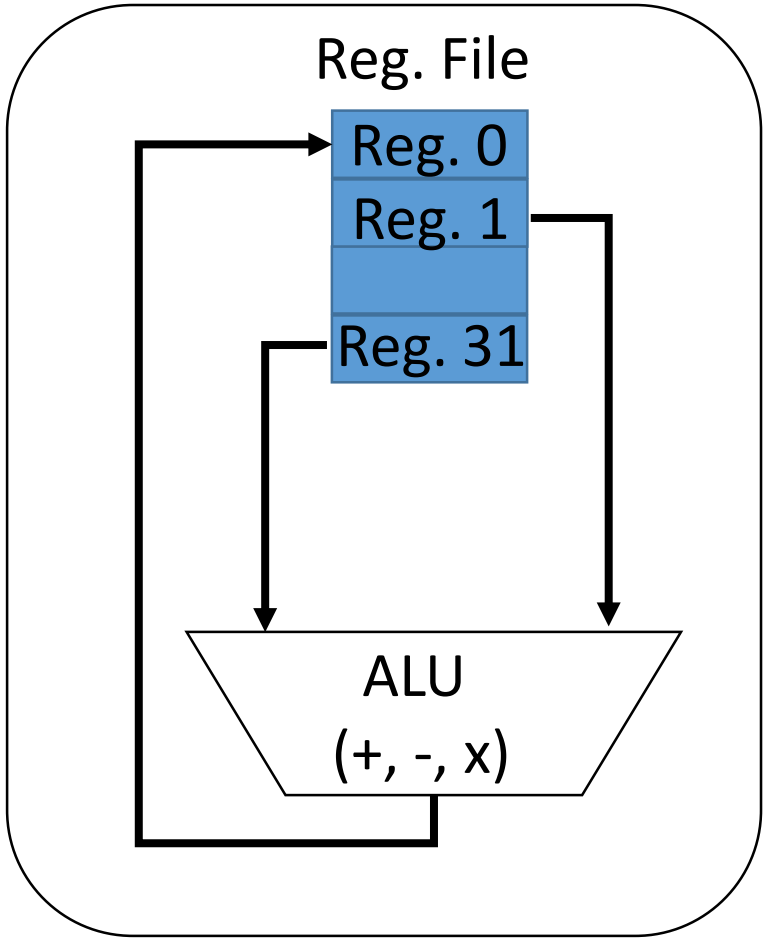
Typical of CISC ISAs

ISA Design: Operand Addresses

Where do the operands come from?



Where is the result written?



add rs, rt, rd
(\$rd ←\$rs + \$rt)

Value of register rt

sub rs, rt, rd
(\$rd ←\$rs - \$rt)

Instruction specifies 2 source and 1 destination register

"3-Address" ISA (E.g., MIPS ISA)

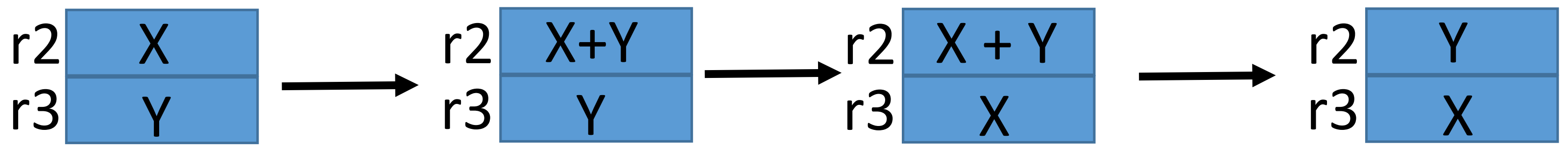
Example: Swap Using Add/Sub

```
X = X + Y;
Y = X – Y; //X +Y – Y = X
X = X - Y ; //X +Y – X = Y
```

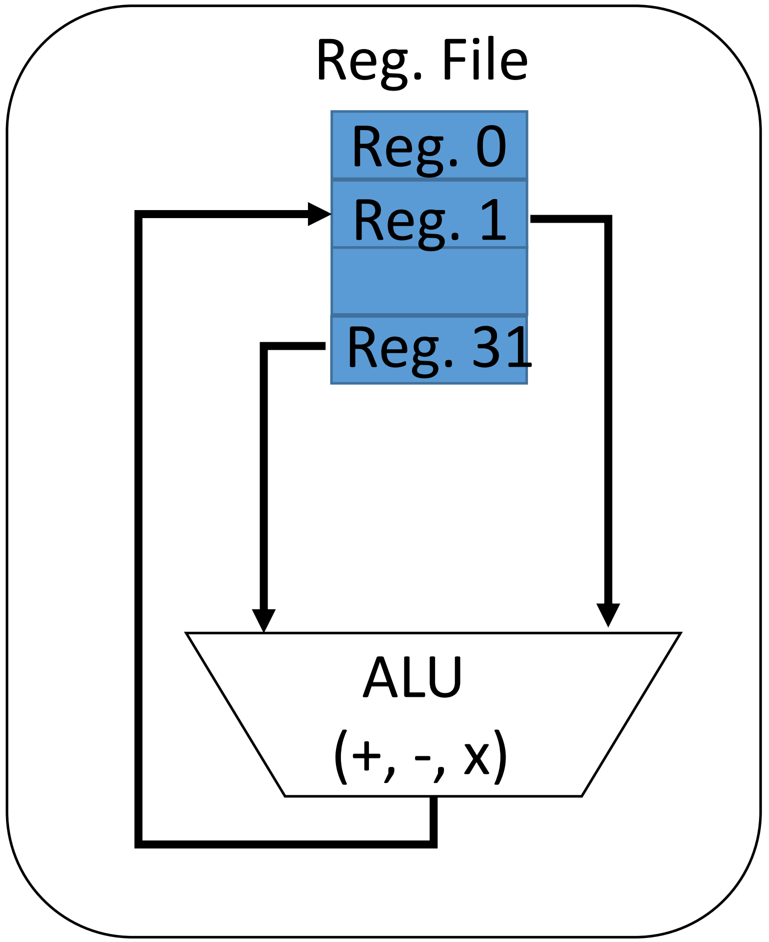
There are simpler ways to implement a swap in the MIPS ISA!

- Swap the contents of Reg. 2 (r2) and Reg. 3 (r3) using only *add* and *sub* operations

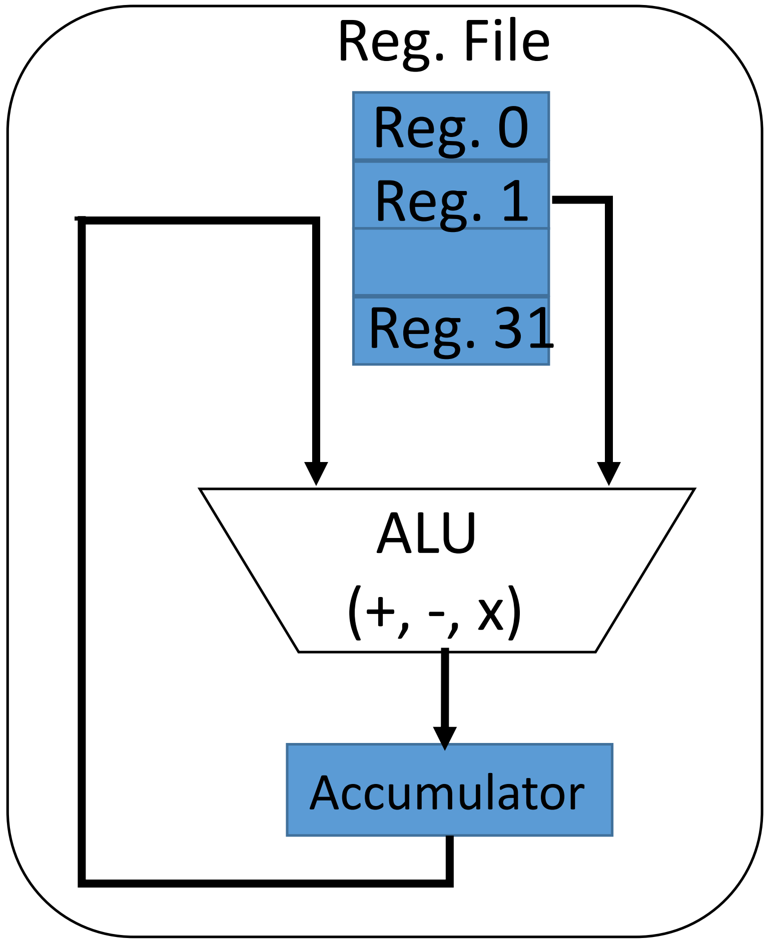
```
add r2, r3, r2
sub r2, r3, r3
sub r2, r3, r2
```



1- and 2-Address ISAs



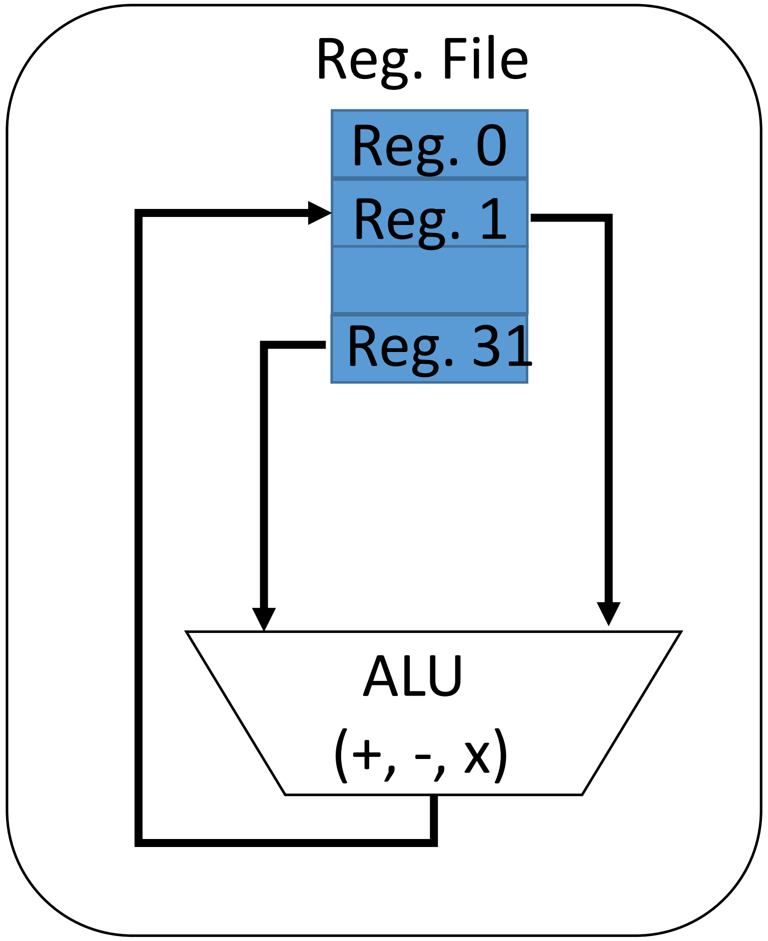
Write result back to a source register
“2-Address” ISA
(Have we seen an example today?)



Special purpose “Accumulator” serves as an operand and as the destination
“1-Address” ISA
(Also called an “Accumulator Architecture”)

Swap Routine for 2-Address ISA

```
X = X + Y;
Y = X – Y; //X +Y – Y = X
X = X - Y ; //X +Y – X = Y
```



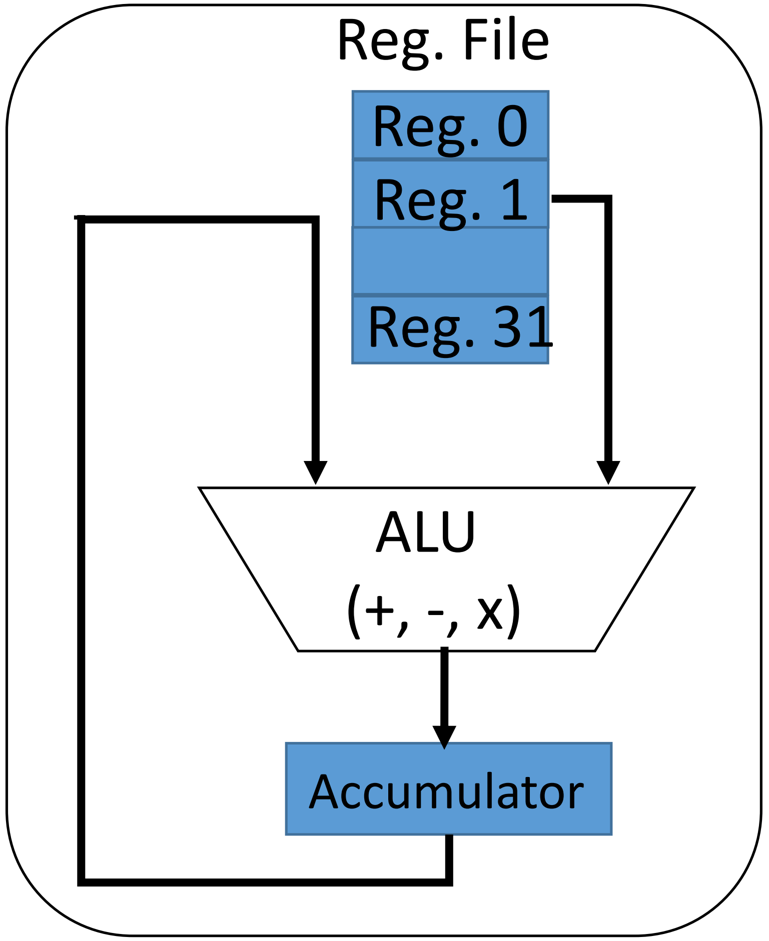
add rs, rd
($rd ←$rd + $rs)

sub rs, rd
($rd ←$rd - $rs)

X is stored in r2
Y is stored in r3
Assume r4 can be used as temp

```
add r3, r2  //r2 = X+Y
sub r4, r4  //r4=0
add r2, r4 //r4=X+Y
sub r3, r4  //r4=X
sub r3, r3  //r3=0
add r4, r3 //r3 = X
sub r3, r2 //r2 = Y
```

Swap Routine for 1-Address ISA



add rs
($acc ←$acc + $rs)

sub rs
($acc ←$acc - $rs)

mvto rs
($rs ←$acc)

mvfrm rs
($acc ←$rs)

X = X + Y;
Y = X – Y; //X +Y – Y = X
X = X - Y ; //X +Y – X = Y

X is stored in r2
Y is stored in r3
Assume r4 can be used as temp

```
mvfrm r2  //acc=X
add r3  //acc=X+Y
mvto r2 //r2=X+Y
sub r3  //acc = X
mvto r3 //r3 = X
mvfrm r2 //acc=X+Y
sub r3 //acc = Y
mvto r2  //r2=Y
```

"Lecture 1: In-class Problem 2" on newclasses.nyu.edu under "Assignments"

So Which is "Better"?

- Metrics?
  - Code size (Bytes)
  - Design complexity

```
add r2, r3, r2
sub r2, r3, r3
sub r2, r3, r2
```

Assume 5-bit opcode for all 3 ISAs

How many bits requires to address 32 registers?

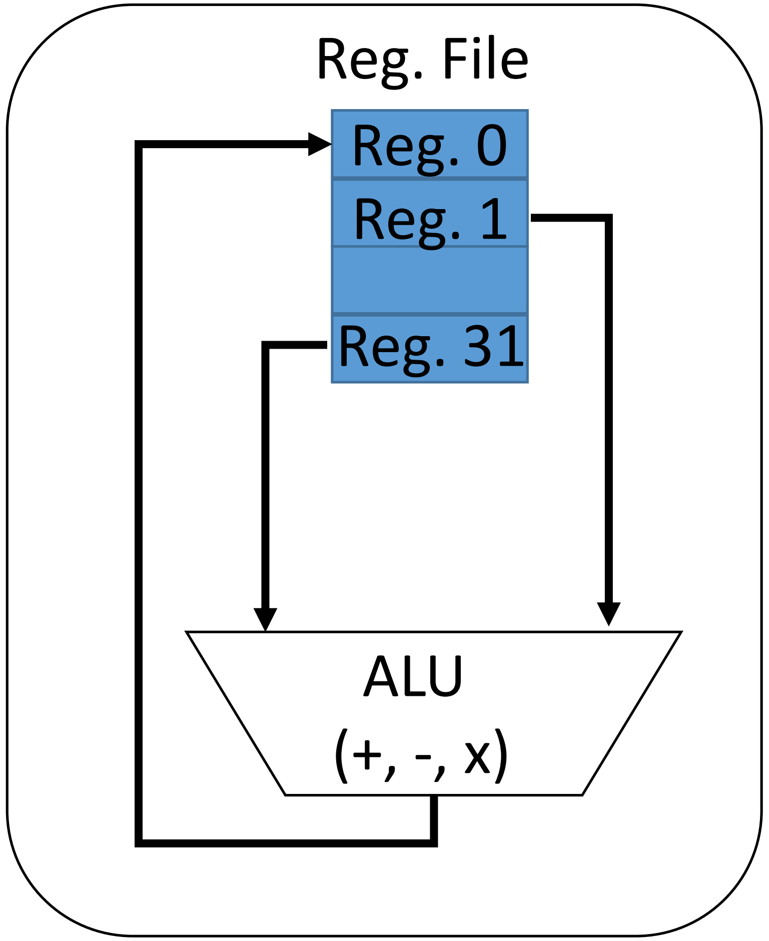
```
add r3, r2
sub r4, r4
add r4, r3
sub r3, r4
sub r3, r4
add r3, r2
 sub r2, r3
```

```
mvfrm r2
add r3
mvto r2
sub r3
mvto r3
mvfrm r2
sub r3
mvto r2
```

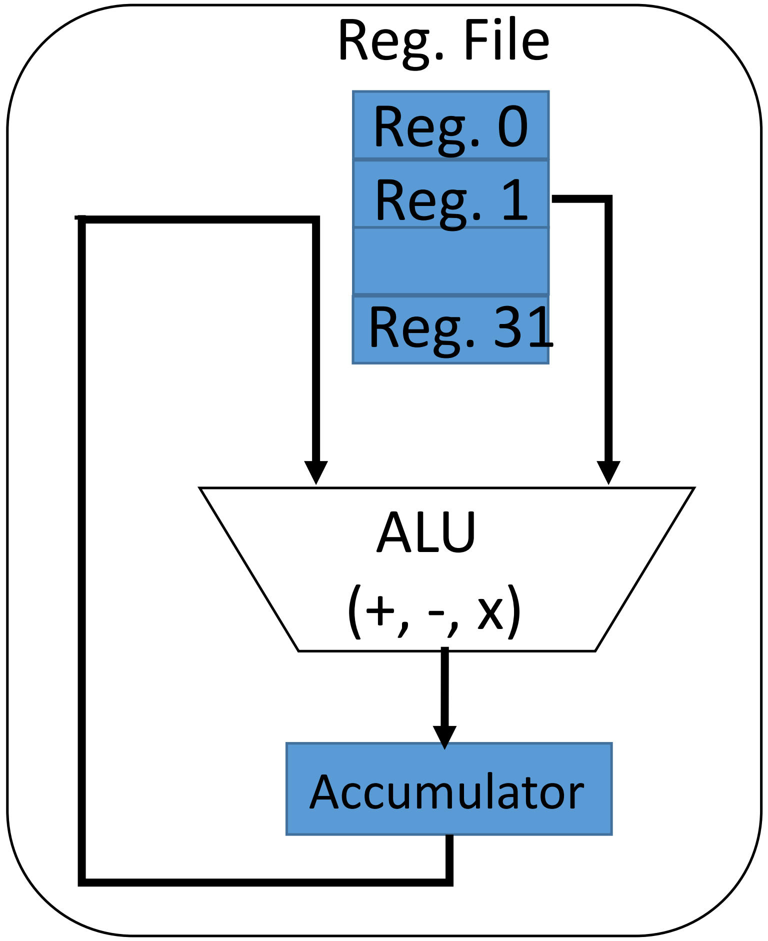
- Code size = Num Insts x Bits/Inst
  - 3-Address ISA
  - 2-Address ISA
  - 1-Address ISA

So Which is "Better"?

- Metrics?
  - Code size (Bytes)
  - Design complexity

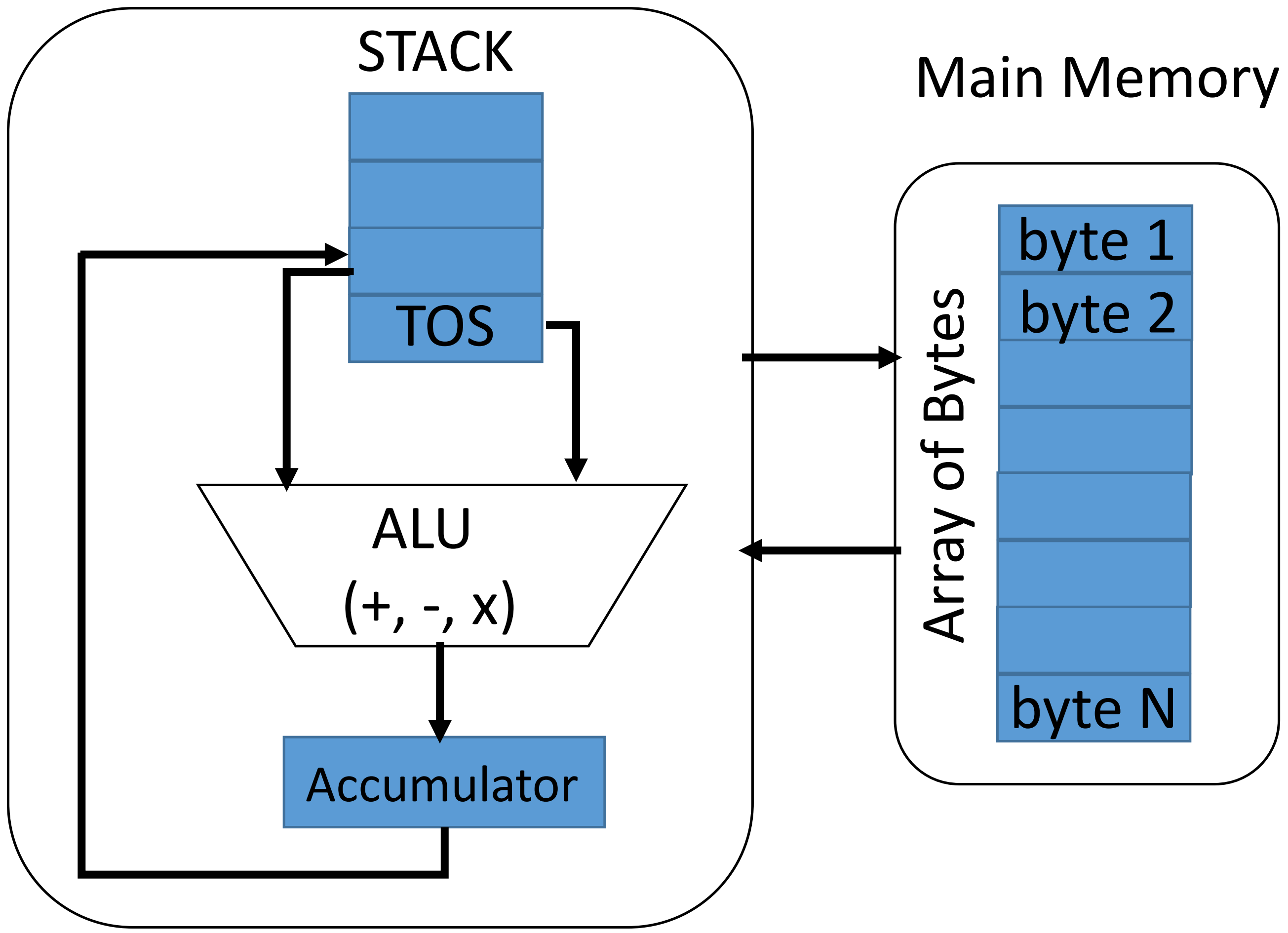


2 Read ports
1 Write port



1 Read ports
+ 1 R/W accum

0-Address ISA (Stack Machines)

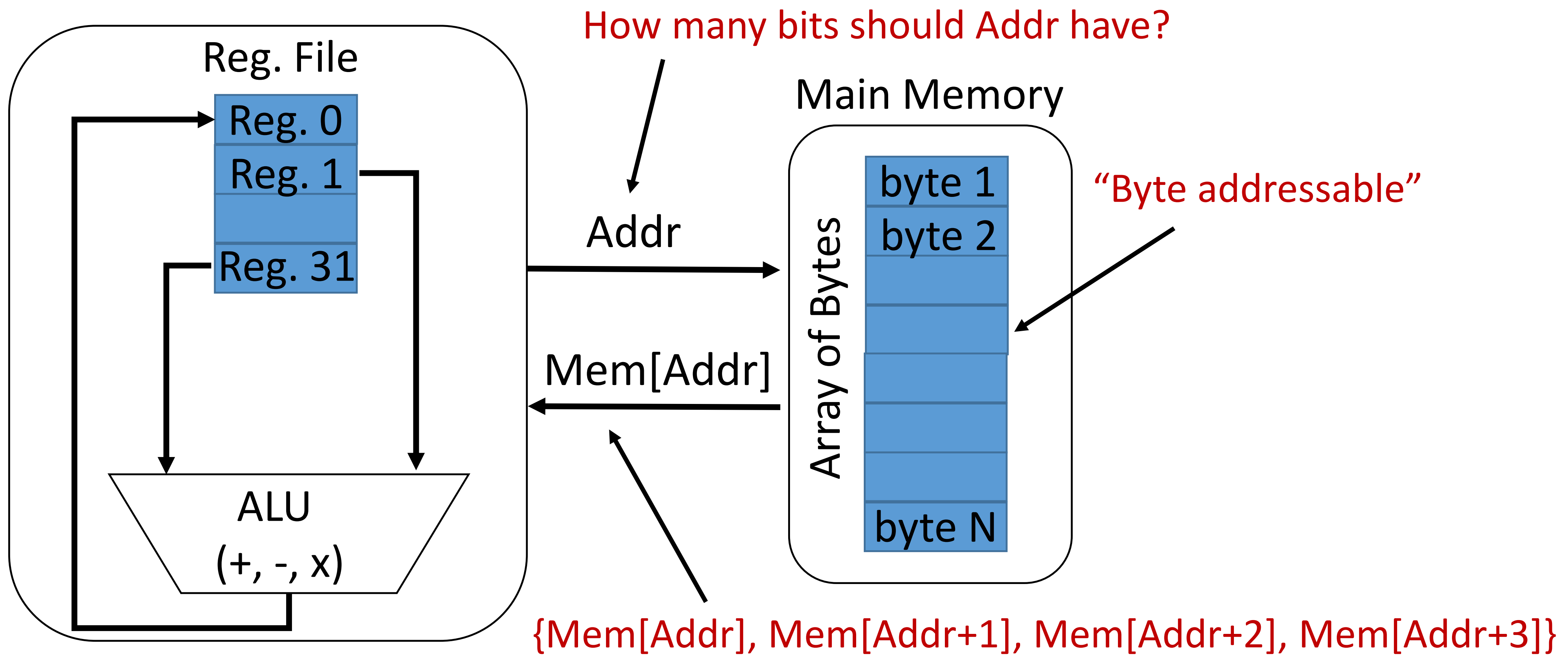


pop M
Mem[M] ← $TOS
TOS ← TOS-1

push M
TOS ← TOS+1
$TOS ← Mem[M]

add
$TOS-1 ←$TOS + $(TOS-1)
TOS ←TOS-1

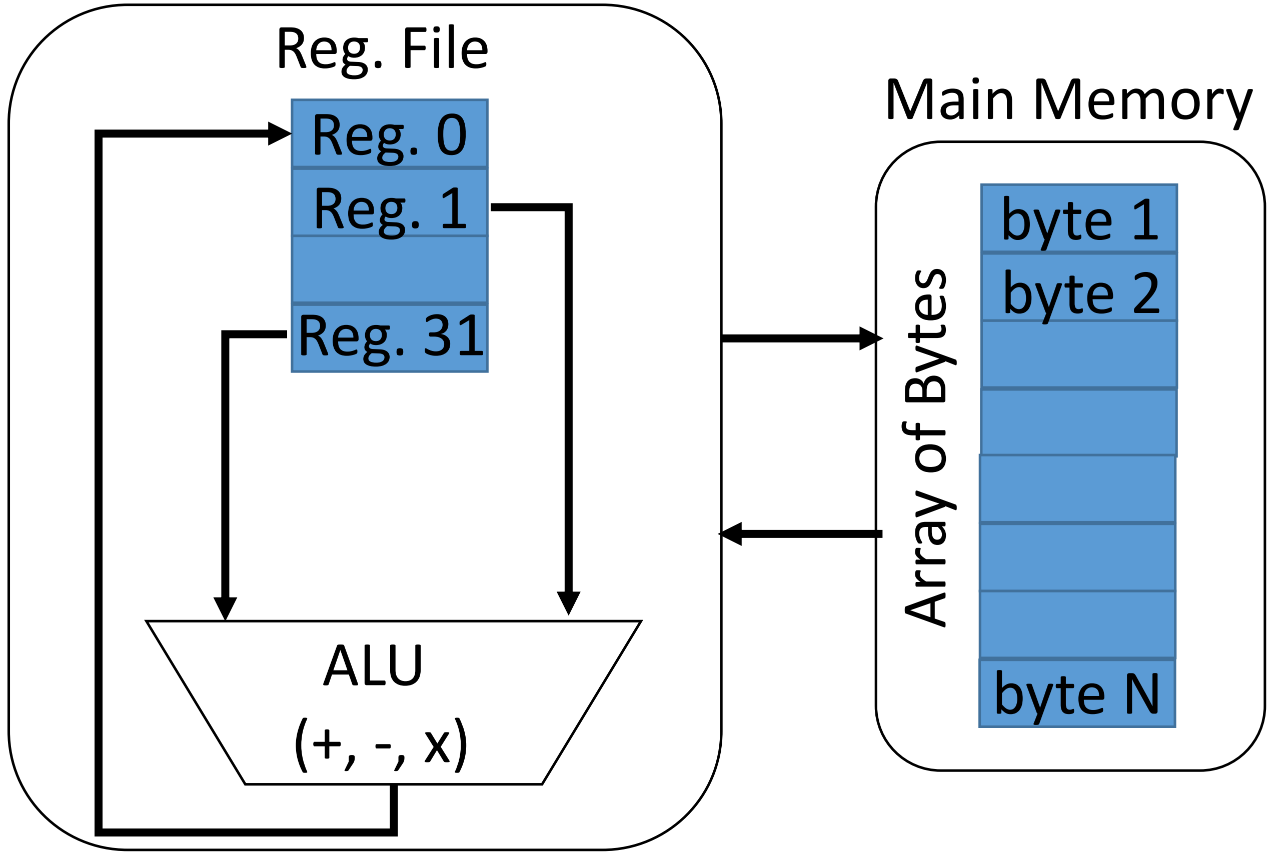
How is Memory Accessed



- Can also be bit-addressable, 32-bit addressable, 64-bit addressable, etc.

Load-Store ISA

- Instructions like *add*, *sub* etc. operate *only* on registers (examples: MIPS, ARM, 3/2/1/0-Address ISAs we have seen so far)
  - Separate Load/Store instructions to fetch data from memory into registers and to write data back from registers to memory



add rs, rt, rd
(\$rd ←\$rs + \$rt)

load rt, M
(\$rt ←Mem[M])

Load data at memory location M into register rt

store rt, M
(Mem[M] ←\$rt)

Store data in register rt to memory location M

Memory-Memory ISA

- Allow instructions like add/sub to operate *directly* on data in memory (Examples include: x86, VAX etc.)

Memory address

add Ms, Mt, Md

(Mem[Md] ← Mem[Ms] + Mem[Mt])

- Most generally, each of the two operands and the result can be read-from/written-to either memory or register file
  - Pros?
  - Cons?

Addressing Modes

- How to specify where to read from/write to

Absolute

load rt, M
(\$rt ←Mem[M])

Directly provide memory address

Register Indirect

load rs, rt
(\$rt ←Mem[\$rs])

Memory address from register rs

Displaced/Based

load rs, rt, offset
(\$rt ←Mem[offset+\$rs])

Add an offset to address stored in register rs

Indexed

load rs, rt, rd
(\$rd ←Mem[\$rs + \$rt])

Base memory address and offset from registers

Memory Indirect

load rt, rs
(\$rt ←Mem[Mem[\$rs]]

Use value at Mem[\$rs] as address

Auto-increment

load rt, rs
(\$rt ←Mem[\$rs]
\$rs ←\$rs + 1)

Same as Register indirect + value in rs is automatically incremented

- Orthogonal ISA: each op supports every addressing mode (ex: VAX)

What Makes a Good ISA

- Ease of hardware implementations
  - Decoder for fixed vs. variable instruction lengths
  - Simple vs. complex addressing modes

- Ease of software implementation
  - Can the programmer/compiler use the ISA easily?
  - How many assembly instructions to represent a single line of code in a high-level language? “semantic gap”

- Backwards compatibility
  - ISA will have to be supported into the future; new instructions can be added but existing instructions cannot be removed

CISC Vs. RISC

- CISC: “Complex” Instruction Set Computer
- RISC: “Reduced” Instruction Set Computer
  - Each CISC instruction = multiple RISC instructions

- CISC vs. RISC debate dominated the 80s/early 90s but is largely resolved now
  - CISC ISAs like x86 are “broken down” internally into simpler RISC-like instructions (microcode)
  - x86 is “externally CISC but internally RISC”
  - Other popular ISAs like ARM and MIPS are RISC

ISA Comparison

	VAX	x86	MIPS
ISA Type	CISC	CISC (but internally RISC)	RISC
Instruction Length	Variable	Variable	Fixed
Addressing Modes	3-Address; memory-memory; orthogonal addressing modes	2-Address; register-memory	3-Address; load/store ISA