# ECE411: Computer Organization and Design

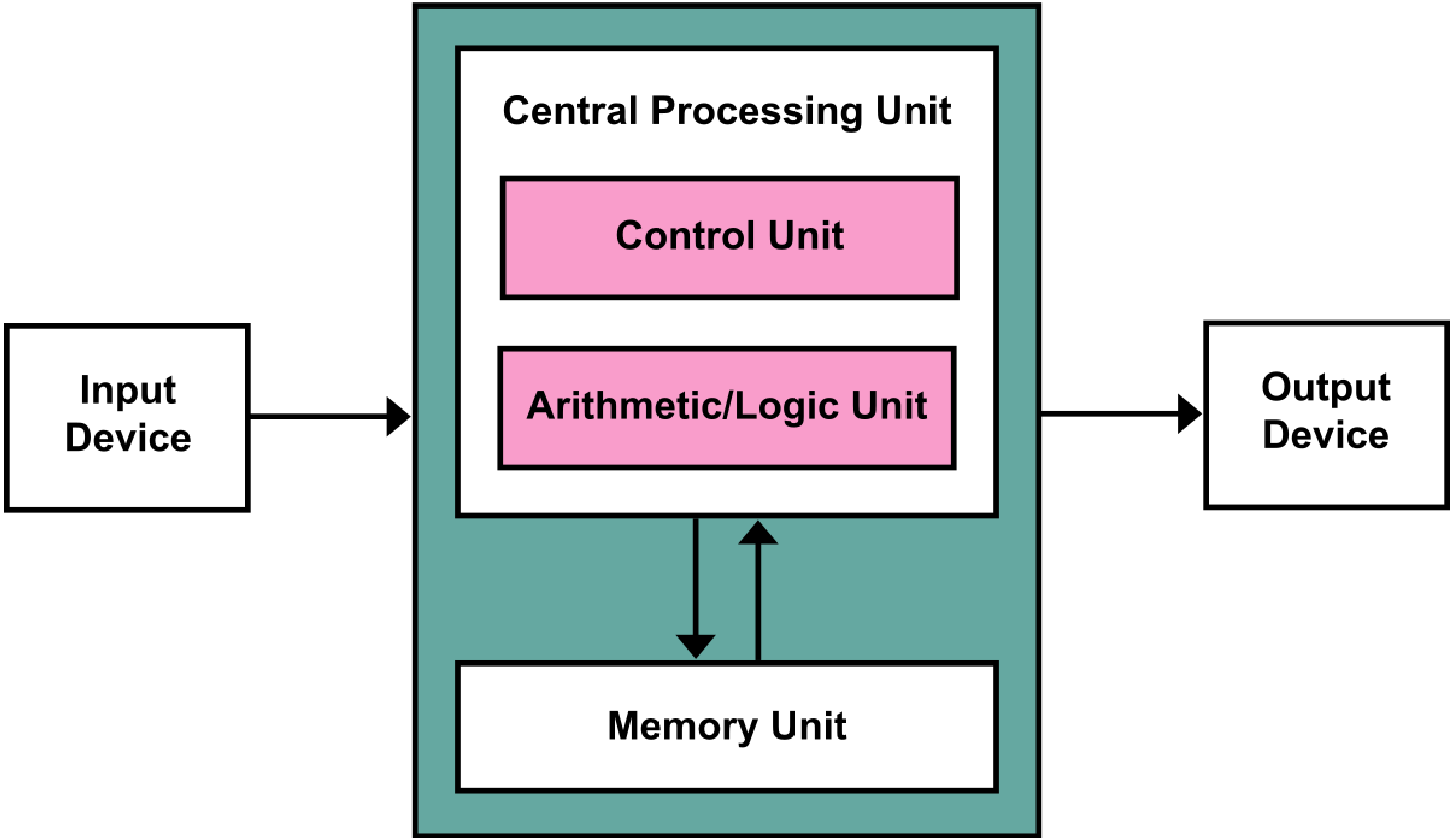
## Lecture 2: Instruction Set Architecture

Rakesh Kumar

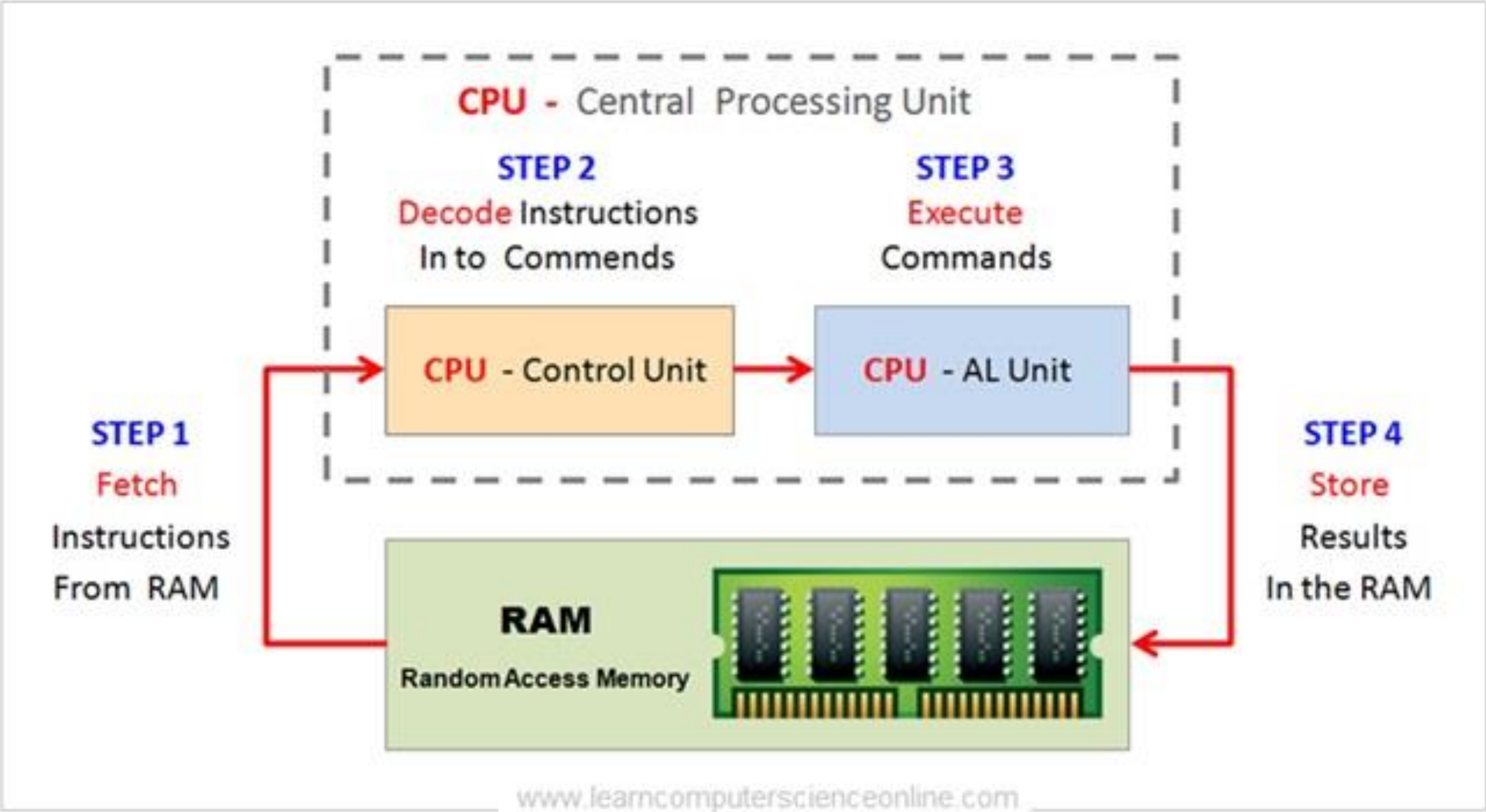
ILLINOIS

# Von-Neumann
# Computer Architecture

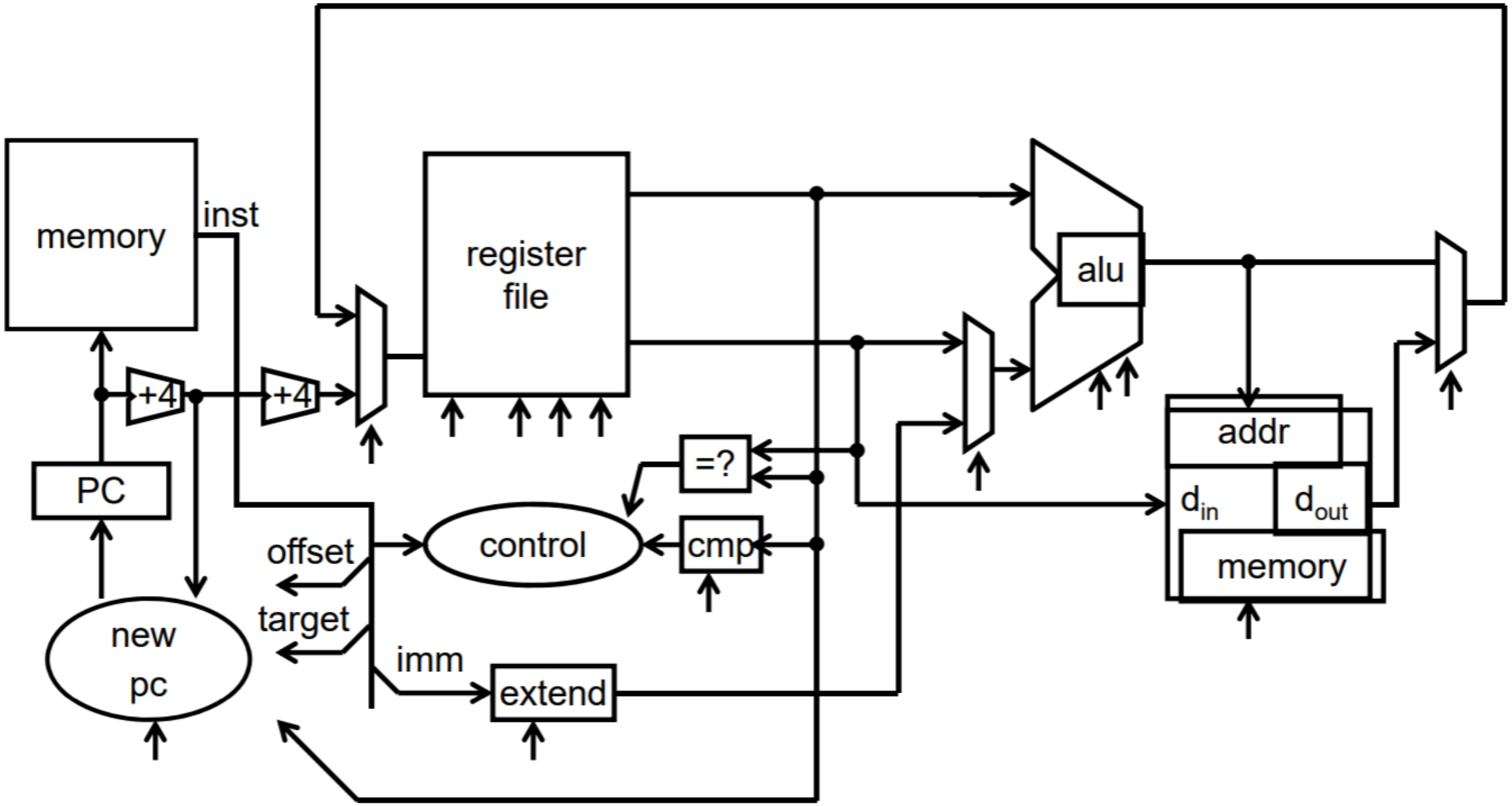
# Von-Neumann Computer Organization



# Instruction Execution Cycles



# A Simple Processor



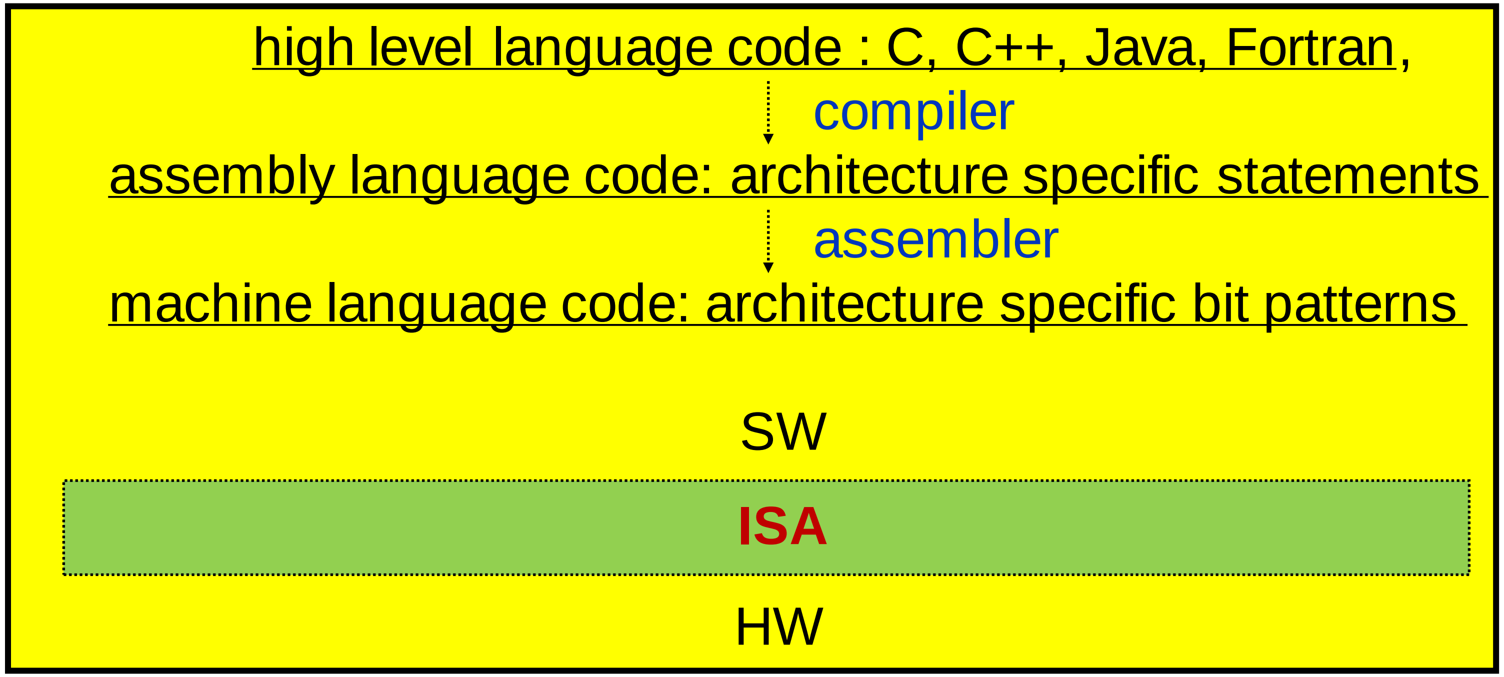
# A Simple Processor (Detailed Explanation)

- executes instructions, which are represented as bit patterns with specific fields interpreted differently
- contains a register file, which holds data that will be referenced by instructions
- has a program counter that holds the address of the instruction being executed
- can access memory
  - use address to select the location we want to access
  - random-access: time to access a piece of data is independent of which address the data is stored in
- I/O
  - keyboard, mouse, display, network
  - long-term data storage: hard disk, CD-ROM, SSD, etc.

# Instruction Set Architecture (ISA)

## Q: What is the function of ISA?

- serves as an interface between software and hardware
- provides a mechanism by which the software tells the hardware what should be done



high level language code : C, C++, Java, Fortran,

compiler

assembly language code: architecture specific statements

assembler

machine language code: architecture specific bit patterns

SW

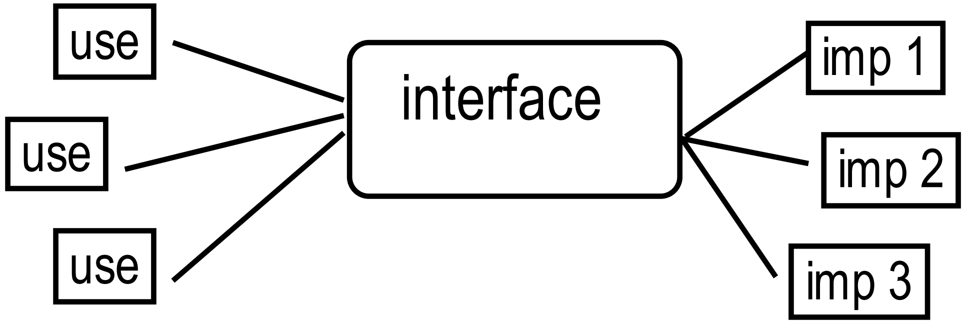
**ISA**

HW

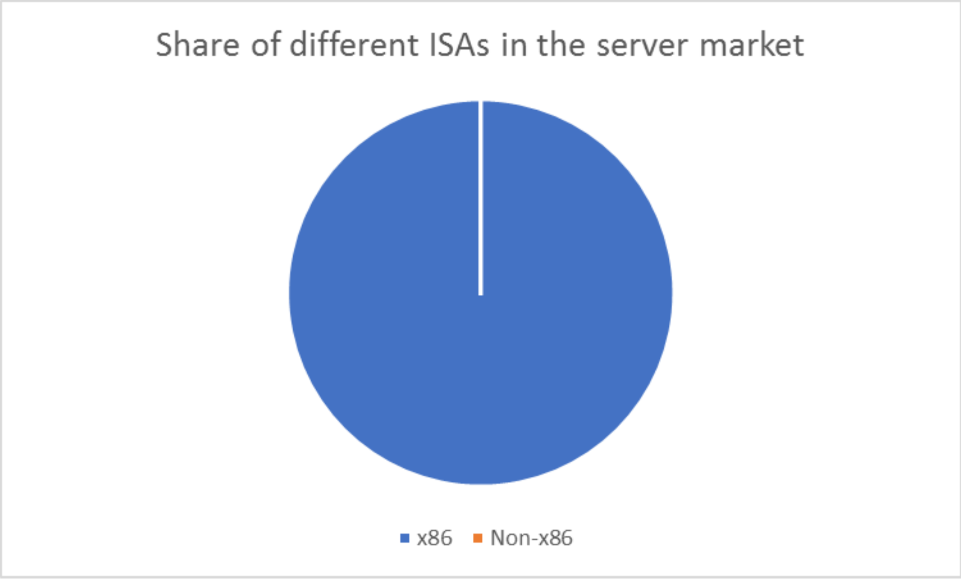
# Interface Design

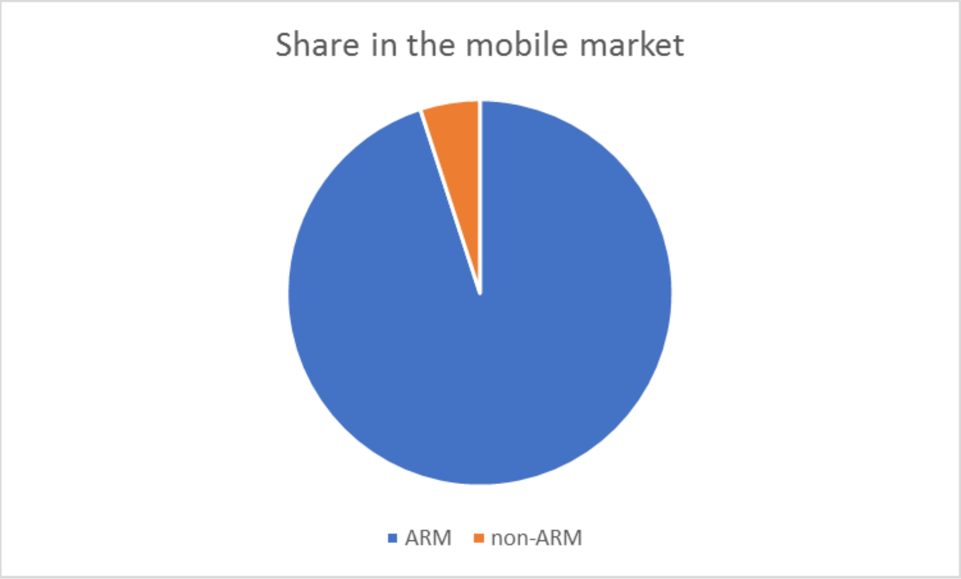
## Q: What does interface mean in computer systems/architecture design?

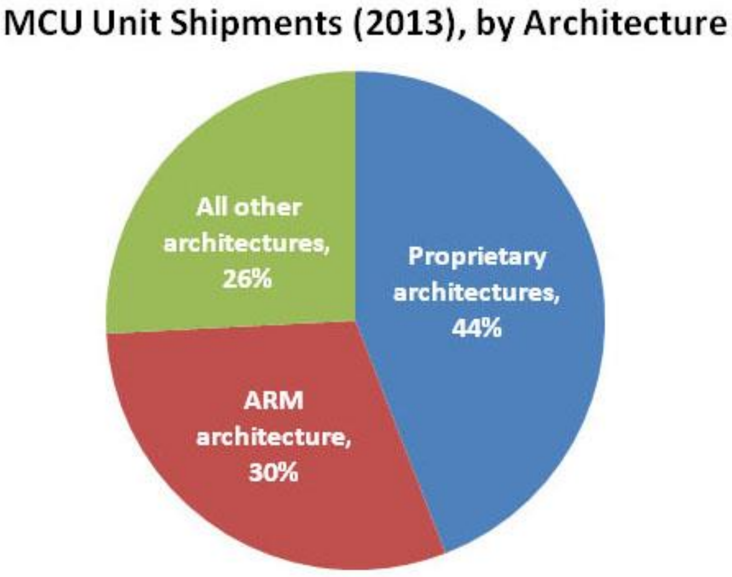
- a good interface:
  - lasts through many implementations (portability, compatibility)
  - is used in different ways (generality)
  - provides convenient functionality to higher levels
  - permits an efficient implementation at lower levels

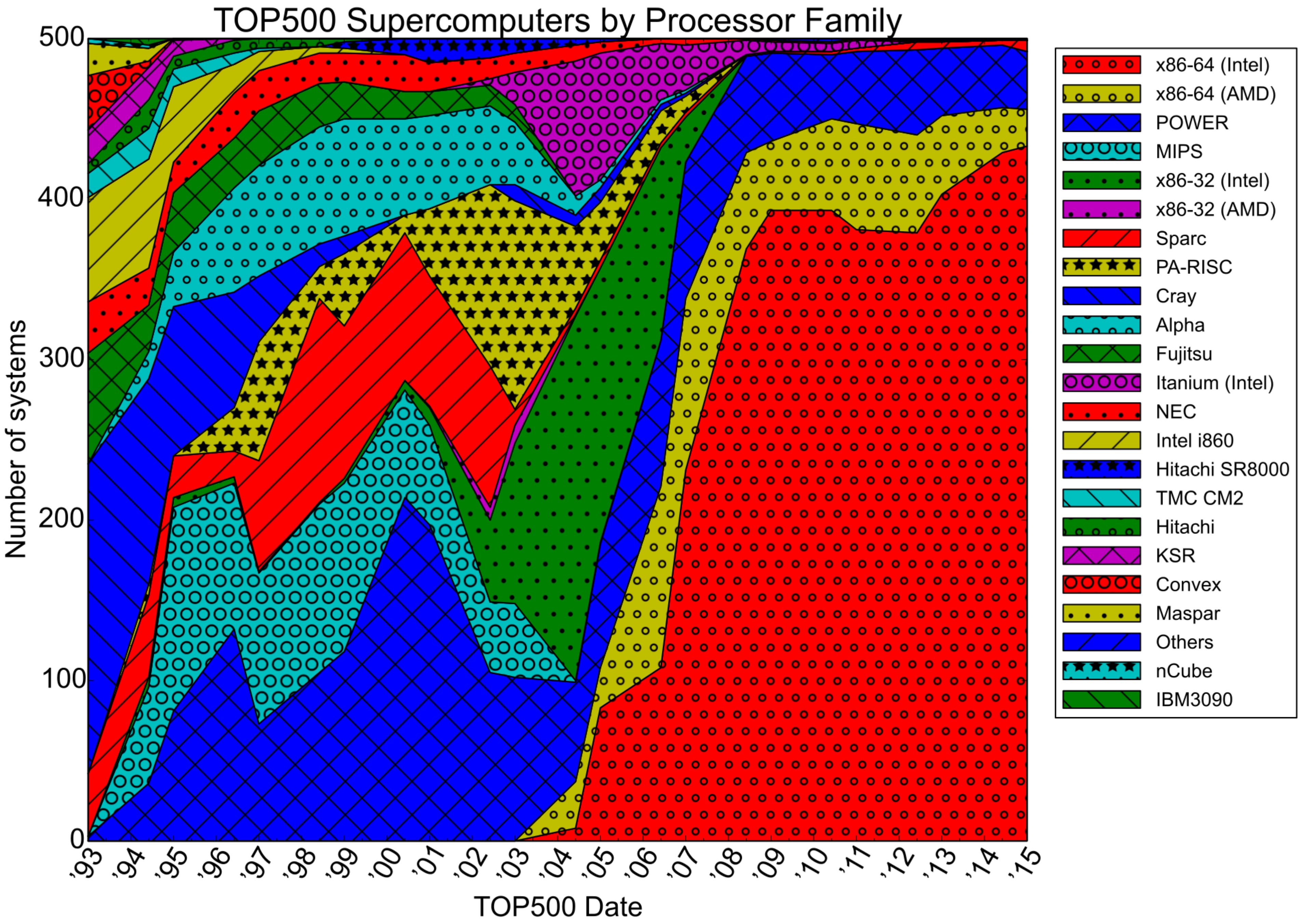


# Today’s ISAs


Share of different ISAs in the server market


Share in the mobile market

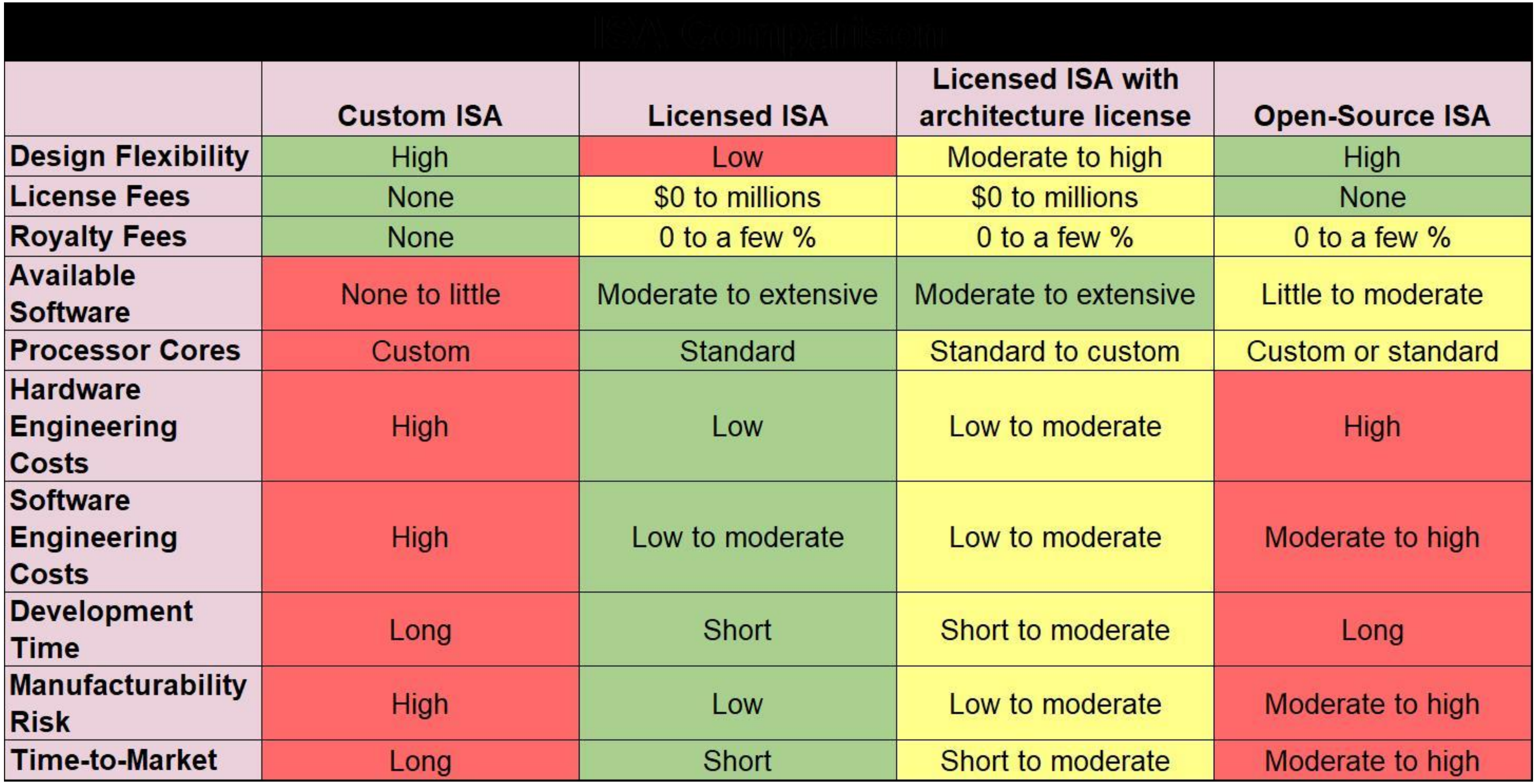

MCU Unit Shipments (2013), by Architecture


TOP500 Supercomputers by Processor Family
Number of systems
0
100
200
300
400
500
TOP500 Date
'93
'94
'95
'96
'97
'98
'99
'00
'01
'02
'03
'04
'05
'06
'07
'08
'09
'10
'11
'12
'13
'14
'15
x86-64 (Intel)
x86-64 (AMD)
POWER
MIPS
x86-32 (Intel)
x86-32 (AMD)
Sparc
PA-RISC
Cray
Alpha
Fujitsu
Itanium (Intel)
NEC
Intel i860
Hitachi SR8000
TMC CM2
Hitachi
KSR
Convex
Maspar
Others
nCube
IBM3090

# Case Study: ARM in server space

- 1. There’s no desktop development environment (where apps can be developed/debugged before deploying on the cloud).
  2. Very few players, so no reason to develop a software ecosystem.
  3. Very public failures, discourages new efforts.
- 4. Poor performance of existing ARM servers
- 5. Geopolitics may discourage ARM further
  6. Resurgence of AMD means that need for ARM is weaker.

# Case Study: RISC-V



ISA Comparison

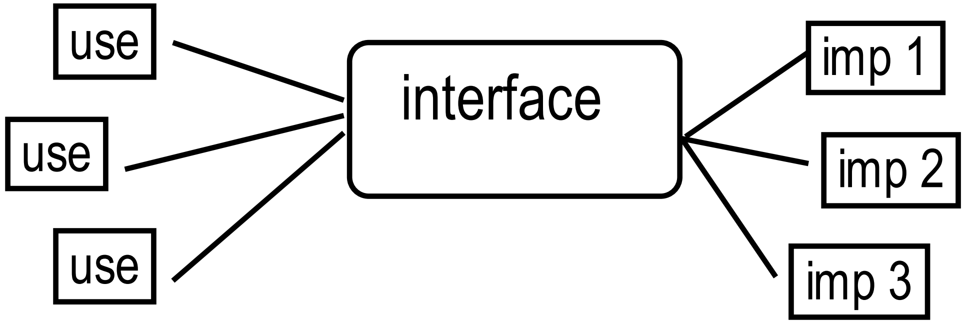
| | Custom ISA | Licensed ISA | Licensed ISA with architecture license | Open-Source ISA |
|---|---|---|---|---|
| Design Flexibility | High | Low | Moderate to high | High |
| License Fees | None | $0 to millions | $0 to millions | None |
| Royalty Fees | None | 0 to a few % | 0 to a few % | 0 to a few % |
| Available Software | None to little | Moderate to extensive | Moderate to extensive | Little to moderate |
| Processor Cores | Custom | Standard | Standard to custom | Custom or standard |
| Hardware Engineering Costs | High | Low | Low to moderate | High |
| Software Engineering Costs | High | Low to moderate | Low to moderate | Moderate to high |
| Development Time | Long | Short | Short to moderate | Long |
| Manufacturability Risk | High | Low | Low to moderate | Moderate to high |
| Time-to-Market | Long | Short | Short to moderate | Moderate to high |

https://www.forbes.com/sites/tiriasresearch/2018/04/05/what-you-need-to-know-about-processor-architectures/#72fec5344f57

# Interface Design

## Q: What does interface mean in computer systems/architecture design?

- a good interface:
  - lasts through many implementations (portability, compatibility)
  - is used in different ways (generality)
  - provides convenient functionality to higher levels
  - permits an efficient implementation at lower levels



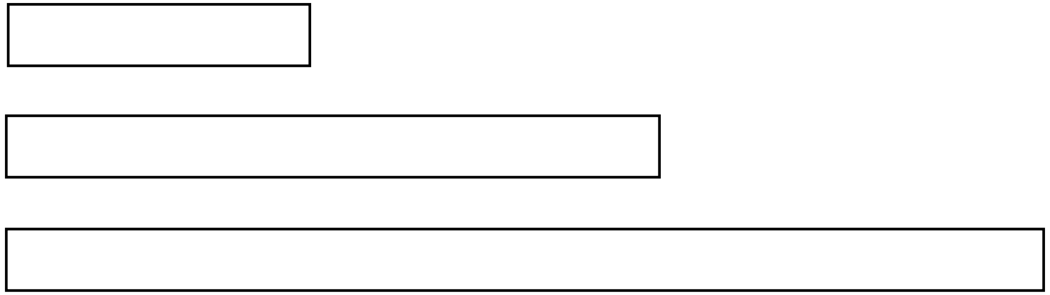
# Instruction Set Design Issues

Q: As a hardware architect, what you should consider in ISA design?

- instruction set design issues include:
  - where are operands stored?
    - registers, memory, stack, accumulator
  - how many explicit operands are there?
    - 0, 1, 2, or 3
  - how is the operand location specified?
    - register, immediate, indirect, . . .
  - what type & size of operands are supported?
    - byte, int, float, double, string, vector. . .
  - what operations are supported?
    - add, sub, mul, move, compare . . .

# Instruction Length

**variable:**



x86 – Instructions vary from 1 to 17 Bytes long

**fixed:**

MIPS, PowerPC, and most other RISC’s:
all instruction are 4 Bytes long

# Instruction Length

- variable-length instructions (x86):
  - require multi-step, complex fetch and decode (-)
  - allow smaller binary programs that require less disk storage, less DRAM at runtime, less memory, bandwidth and better cache efficiency (+)

- fixed-length instructions (RISC's)
  - allow easy fetch and decode (+)
  - simplify pipelining and parallelism (+)
  - result in larger binary programs that require more disk storage, more DRAM at runtime, more memory bandwidth and lower cache efficiency (-)

# Case Study: ARM

How can we overcome the limitation of fixed-length instructions?

- ARM (Advanced RISC Machine)
  - started with fixed, 32-bit instruction length
  - added thumb instructions
    - a subset of the 32-bit instructions
    - all encoded in 16 bits
    - all translated into equivalent 32-bit instructions within the processor pipeline at runtime
    - can access only 8 general purpose registers
  - motivated by many resource constrained embedded applications that require less disk storage, less DRAM at runtime, less memory bandwidth and better cache efficiency

Q: How about RISC-V?

# How many registers?

- most computers have a small set of registers
  - on-chip memory to hold values that will be used soon
  - a typical instruction use 2 or 3 register values
- advantages of a small number of registers:
  - it requires fewer instruction bits to specify which one
  - less hardware
  - faster access (shorter wires, fewer gates)
  - faster context switch (when all registers need saving)
- advantages of a larger number:
  - fewer loads and stores needed
  - easier to express several operations in parallel

# Where do operands reside?

When the ALU needs them?

- stack machine:
  - push loads memory into 1st register ("top of stack"), moves other regs down
  - pop does the reverse
  - add combines contents of first two registers, moves rest up
- accumulator machine:
  - only 1 register (called the "accumulator")
  - instruction includes store and ACC← ACC + MEM
- register-memory machine :
  - arithmetic instructions can use data in registers and/or memory
- load-store machine (aka register-register machine):
  - arithmetic instructions can only use data in registers

# Classifying 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] |

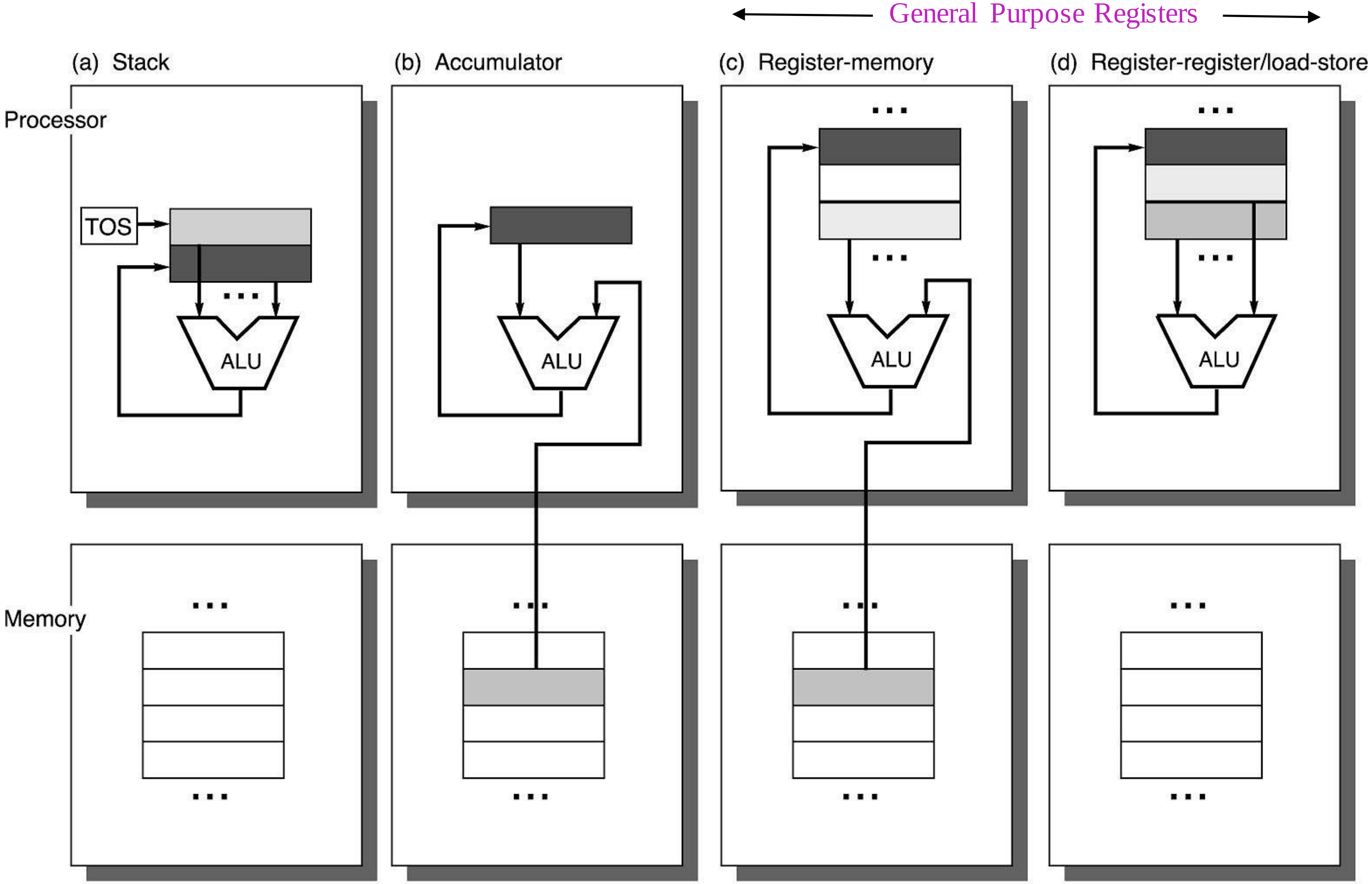
**register-memory (1970s to present, e.g., 80x86):**

| | | |
|---|---|---|
| 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 |

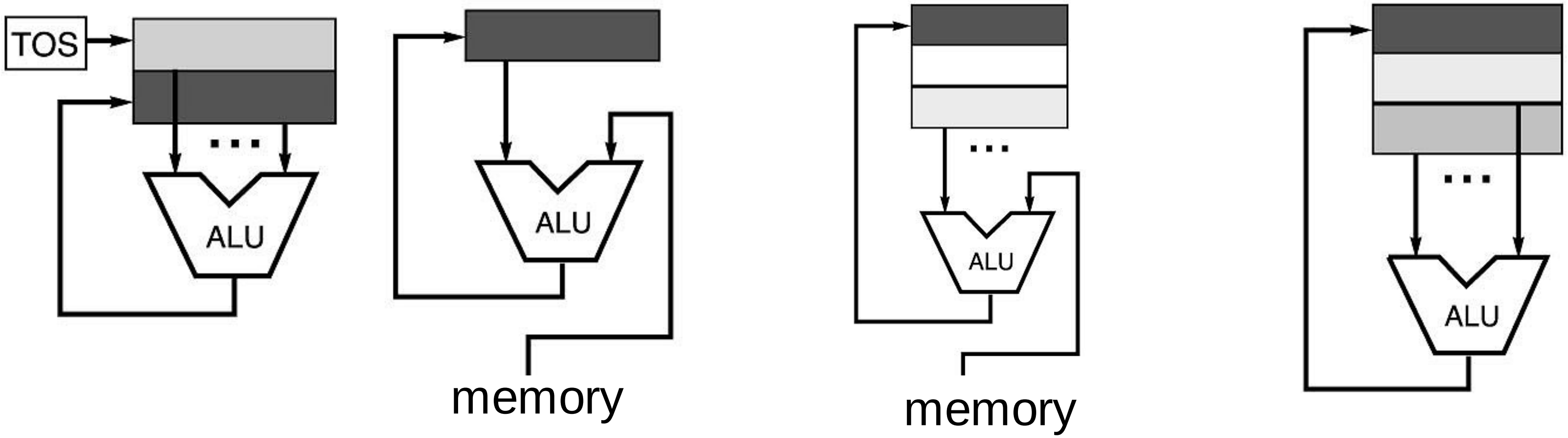
# Operand Locations in Four ISA Classes



# Four Instruction Sets

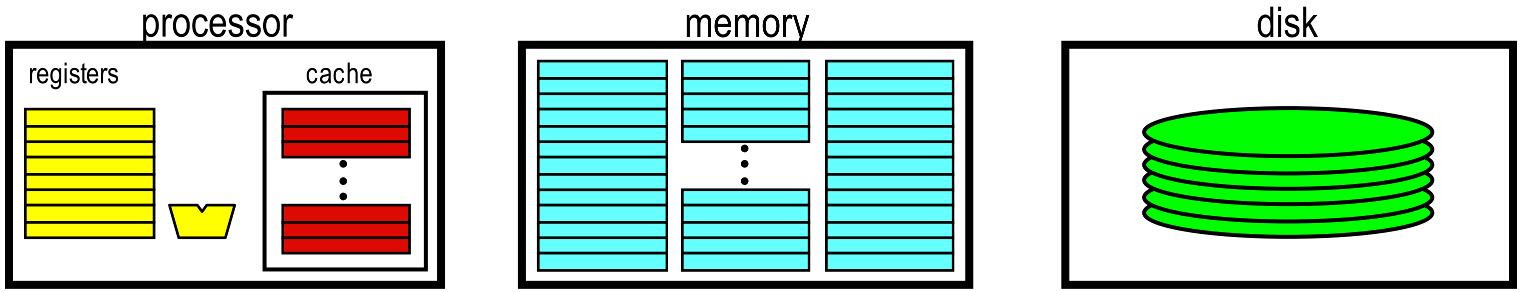
- code sequence  C = A + B

| stack | accumulator | Register (register-memory) | register (load-store) |
|---|---|---|---|
| push A<br>push B<br>add<br>pop C | load A<br>add B<br>store C | load R1, A<br>add R1, B<br>store C, R1 | load R1,A<br>load R2, B<br>add R3, R1, R2<br>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)

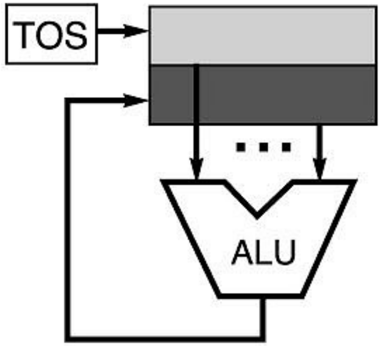


# Latency Numbers Every Programmer Should Know

| | | | | |
|---|---|---|---|---|
| Registers | 0.5 – 2 ns | | | |
| L1 cache reference | 0.5 ns | | | |
| Branch mispredict | 5 ns | | | |
| L2 cache reference | 7 ns | | | 14x L1 cache |
| Mutex lock/unlock | 25 ns | | | |
| Main memory reference | 100 ns | | | 20x L2 cache, 200x L1 cache |
| Compress 1K bytes with Zippy | 3,000 ns | 3 us | | |
| Send 1K bytes over 1 Gbps network | 10,000 ns | 10 us | | |
| Read 4K randomly from SSD* | 150,000 ns | 150 us | | ~1GB/sec SSD |
| Read 1 MB sequentially from memory | 250,000 ns | 250 us | | |
| Round trip within same datacenter | 500,000 ns | 500 us | | |
| Read 1 MB sequentially from SSD* | 1,000,000 ns | 1,000 us | 1 ms | ~1GB/sec SSD, 4X memory |
| Disk seek | 10,000,000 ns | 10,000 us | 10 ms | 20x datacenter roundtrip |
| Read 1 MB sequentially from disk | 20,000,000 ns | 20,000 us | 20 ms | 80x memory, 20X SSD |
| Send packet CA->Netherlands->CA | 150,000,000 ns | 150,000 us | 150 ms | |

*Credits: measured by Google (numbers vary on different machines)*

# Stack Architecture



- instruction set:
  - add, sub, mult, div, . . .
  - push A, pop A

- example: A*B - (A+C*B)
  - push A
  - push B
  - mul
  - push A
  - push C
  - push B
  - mul
  - add
  - sub

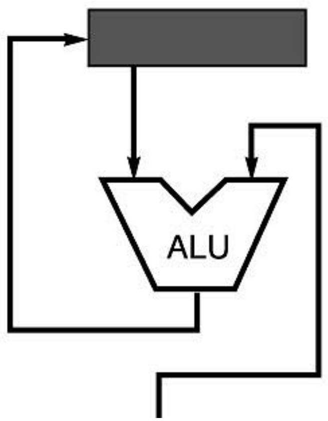
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | B | A*B | A | C | B | B*C | A+B*C | result |
| | A | | A*B | A | C | A | A*B | |
| | | | | A*B | A | A*B | | |
| | | | | | A*B | | | |

# 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 swap are needed
  - difficult to write an optimizing compiler for stack architectures

# Accumulator Architectures



acc = acc +,-,*,/ mem[A]

- instruction set:
  - add A, sub A, mult A, div A, . . .
  - load A, store A

- example: A*B - (A+C*B)
  - load B
  - mul C
  - add A
  - store D
  - load A
  - mul B
  - sub D

B B*C A+B*C A+B*C A A*B result

# 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 Architecture

- instruction set:

|  |  |  |  |
|---|---|---|---|
| (3 operands) | add A, B, C | sub A, B, C | mul A, B, C |
| (2 operands) | add A, B | sub A, B | mul A, B |

- example: A*B - (A+C*B)

```
  (3 operands)                 (2 operands)
    mul D, A, B                mov D, A
    mul E, C, B                mul D, B
    add E, A, E                mov E, C
    sub E, D, E                mul E, B
                               add E, A
                               sub E, D
```

# 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

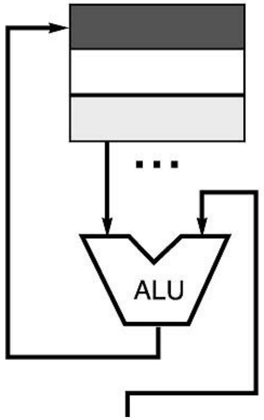
# Register-Memory Architecture

- instruction set:

```
add R1,  A  sub R1, A          mul R1, B
load R1, A  store R1, A
```

- example:  A*B - (A+C*B)

```
load R1, A
mul R1, B          /*       A*B              */
store R1, D
load R2, C
mul R2, B          /*       C*B              */
add R2, A          /*       A + CB           */
sub R2, D          /*       AB - (A + C*B)   */
```



R1 =  R1 +,-,*,/ mem[B]

# Register-Memory: Pros and Cons

- pros
  - some data can be accessed w/o loading first
  - instruction format easy to encode
  - good code density

- cons
  - operands are not equivalent (execution time is determined by the slowest part)

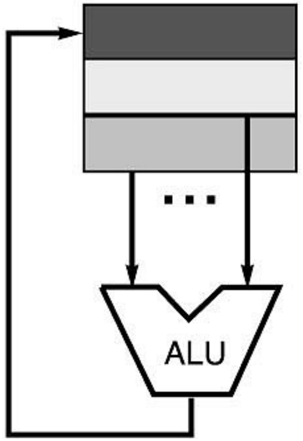
# Load-Store (Register-Register) Architecture

- instruction set:

```
add R1,  R2, R3      sub R1, R2, R3  mul R1, R2, R3
load R1, A           store R1, A     move R1, R2
```

- example:  A*B - (A+C*B)

```
load R1, A
load R2, B
load R3, C
mul R7, R3, R2     /*     C*B             ∗/
add R8, R7, R1     /*     A + C*B         ∗/
mul R9, R1, R2     /*     A∗B             ∗/
sub R10, R9, R8    /*     A∗B - (A+C*B)   ∗/
```



R3 =  R1 +,-,*,/ R2

# Register-Register: 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

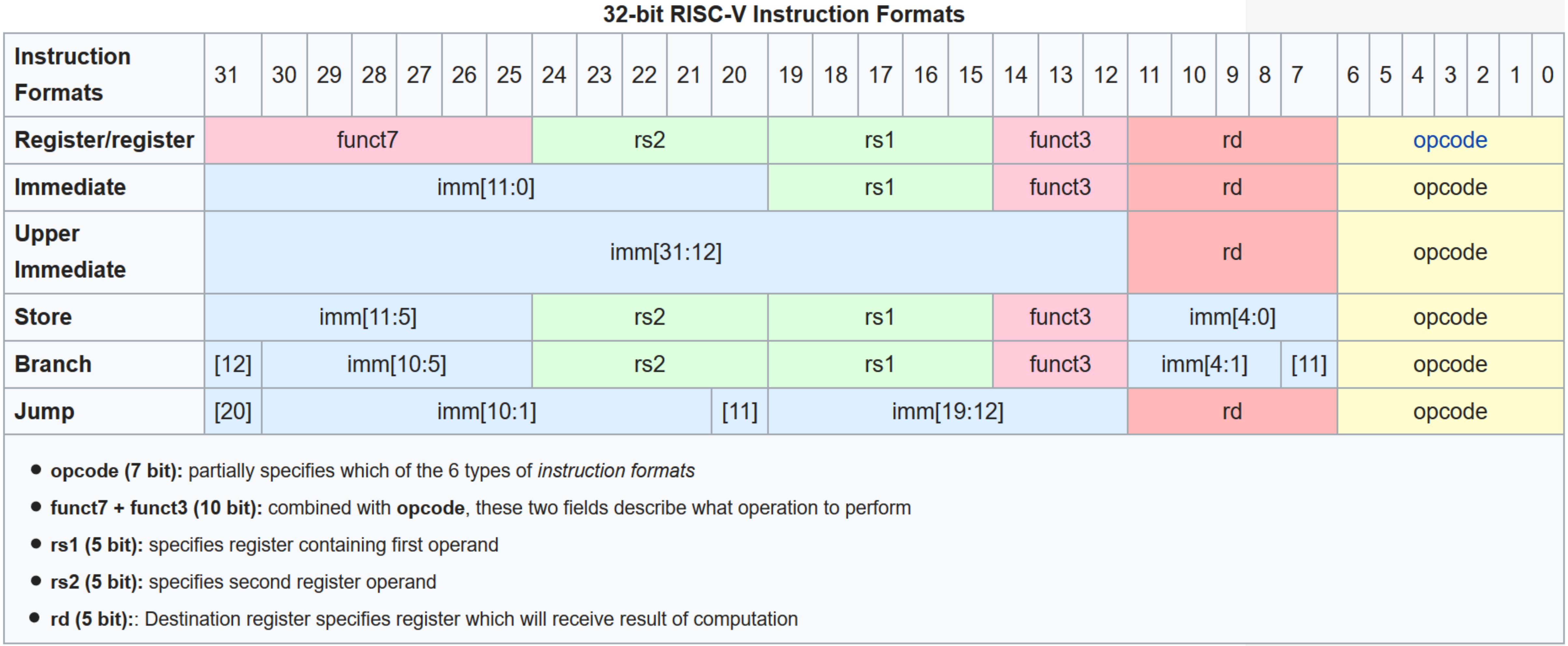
# Registers: Advantages and Disadvantages (Review)

- 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 (e.g., RISC, to simplify CPU design).

# Instruction Format



32-bit RISC-V Instruction Formats

| Instruction Formats | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Register/register** | funct7 | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | |
| **Immediate** | imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | |
| **Upper Immediate** | imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | opcode | | | | | | |
| **Store** | imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:0] | | | | | opcode | | | | | | |
| **Branch** | [12] | imm[10:5] | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1] | | | | [11] | opcode | | | | | | |
| **Jump** | [20] | imm[10:1] | | | | | | | | | | [11] | imm[19:12] | | | | | | | | rd | | | | | opcode | | | | | | |

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit)::** Destination register specifies register which will receive result of computation

RISC-V ISA manual: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

# Types of Operations

| | |
|---|---|
| ● arithmetic and logic: | AND, ADD |
| ● data transfer: | MOVE, LOAD, STORE |
| ● control | BRANCH, JUMP, CALL |
| ● system | OS CALL, VM |
| ● floating point | ADDF, MULF, DIVF |
| ● decimal | ADDD, CONVERT |
| ● string | MOVE, COMPARE |
| ● graphics | (DE)COMPRESS |

# Conditional branch

- how do you specify the destination of a branch/jump?
  - theoretically, the destination is a full address
    - 16 bits for LC3b
    - 32 bits for MIPS, RISC-V
- studies show that almost all conditional branches go short distances from the current program counter (loops, if-then-else)
  - we can specify a relative address in much fewer bits than an absolute address
  - e.g., beq $1, $2, 100=>if ($1 == $2) PC = PC + 100 * 4

# Jumps

- need to be able to jump to an absolute address sometimes
  - jump -- j 10000  => PC = 10000
- need to be able to do procedure calls and returns
  - jump and link--jal 100000 => $31 = PC + 4; PC = 10000
    - used for procedure calls
  - jump register -- jr $31 => PC = $31
    - used for returns, but can be useful for lots of other things

| OP | target |
|---|---|

# ISA Tradeoffs (An Example of MIPS)

| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|---|
| r format | OP | rs | rt | rd | sa | funct |
| i format | OP | rs | rt | immediate | | |
| j format | OP | target | | | | |

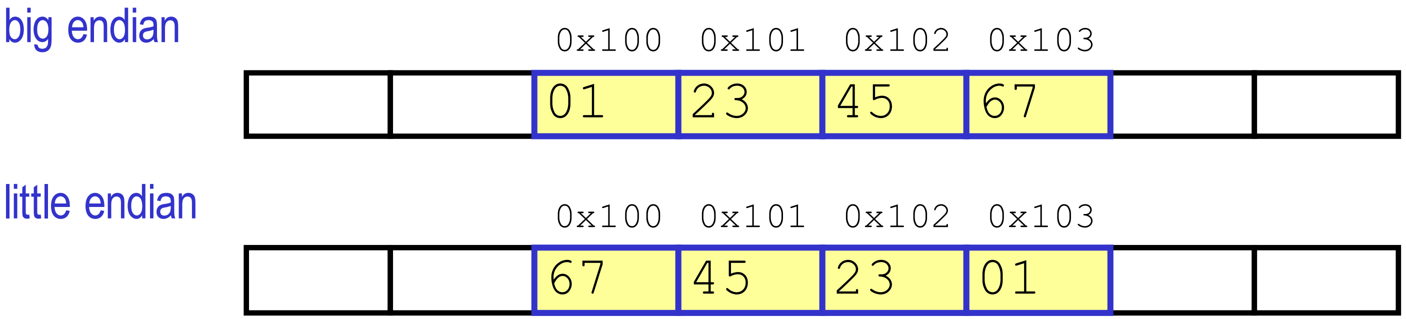
- what if?
  - 64 registers
  - 20-bit immediate
  - 4 operand instruction (e.g., Y = AX + B)

# Byte Ordering

- how should bytes within multi-byte word be ordered in memory?
- conventions
  - Sun’s, Mac’s are “big endian” machines
    - least significant byte has highest address
  - Alphas, PC’s are “little endian” machines
    - least significant byte has lowest address

# Byte Ordering Example

- big endian
  - least significant byte has highest address
- little endian
  - least significant byte has lowest address
- example
  - variable x has 4-byte representation 0x01234567
  - address given by &x is 0x100



big endian

|  |  | 0x100 | 0x101 | 0x102 | 0x103 |  |  |
|---|---|---|---|---|---|---|---|
|  |  | 01 | 23 | 45 | 67 |  |  |

little endian

|  |  | 0x100 | 0x101 | 0x102 | 0x103 |  |  |
|---|---|---|---|---|---|---|---|
|  |  | 67 | 45 | 23 | 01 |  |  |

# Reading Byte-Reversed Listings

- Disassembly
  - text representation of binary machine code
  - generated by program that reads the machine code
- Example fragment

```
address     instruction code          assembly rendition
8048365:    5b                        pop    %ebx
8048366:    81 c3 ab 12 00 00         add    $0x12ab,%ebx
804836c:    83 bb 28 00 00 00 00      cmpl   $0x0,0x28(%ebx)
```

- Deciphering numbers
  - value: `0x12ab`
  - pad to 4 bytes: `0x000012ab`
  - split into bytes: `00 00 12 ab`
  - reverse: `ab 12 00 00`

# Takeaways

Function of ISA

ISA Design Issues

Classical ISAs: Their Pros and Cons

RISC-V Instruction Format

Practice RISC-V in your MPs

Byte ordering: big-endian vs. little-endian

# Announcement

- Next lecture
  - performance, energy, and power metric: [HP2] Chapter 1.5, 1.8
- MP assignment
  - MP1 deadline: 9/1/2022