The Hacker Within: Computer Architecture

March 18, 2015

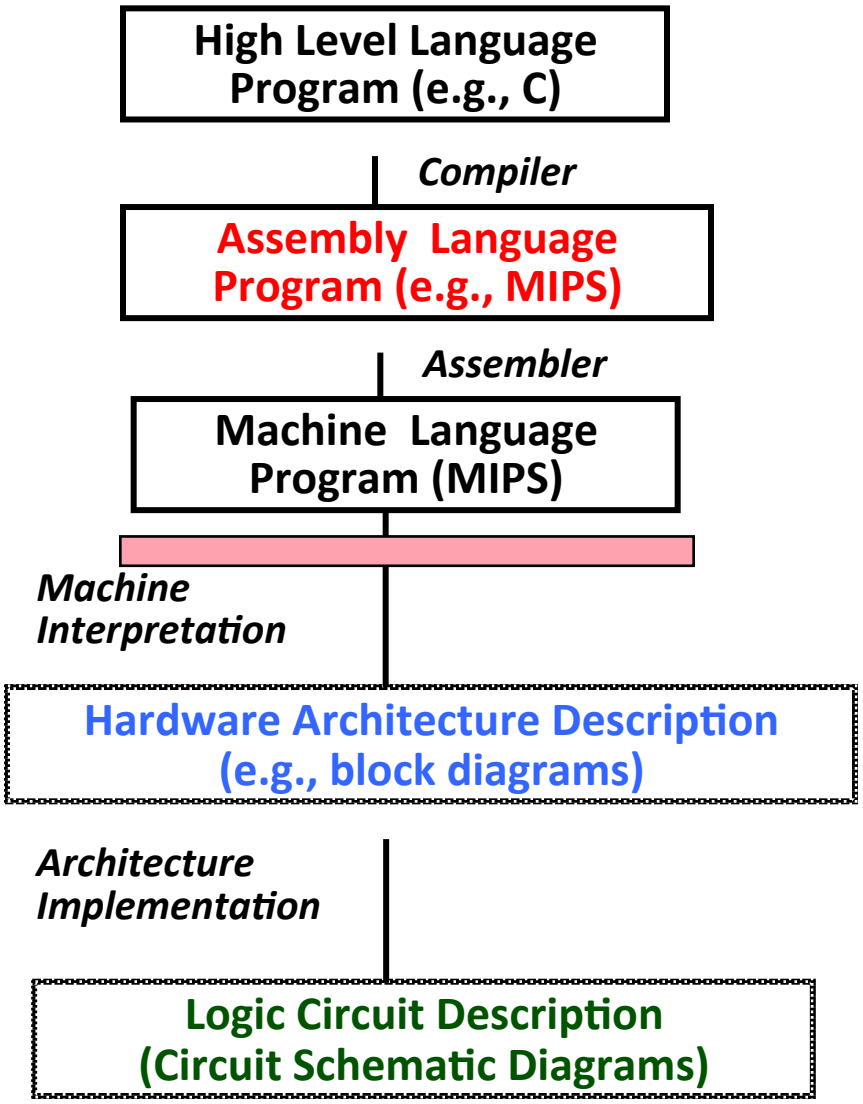
Images and Slides from Dan Garcia & Miki Lustig's CS61C

Presenter: Alex Chong

6 Great Ideas in Computer Architecture

1. Abstraction (Layers of Representation/Interpretation)
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

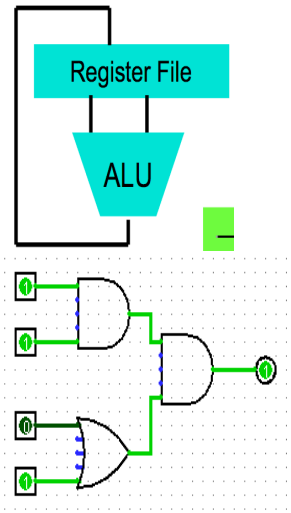


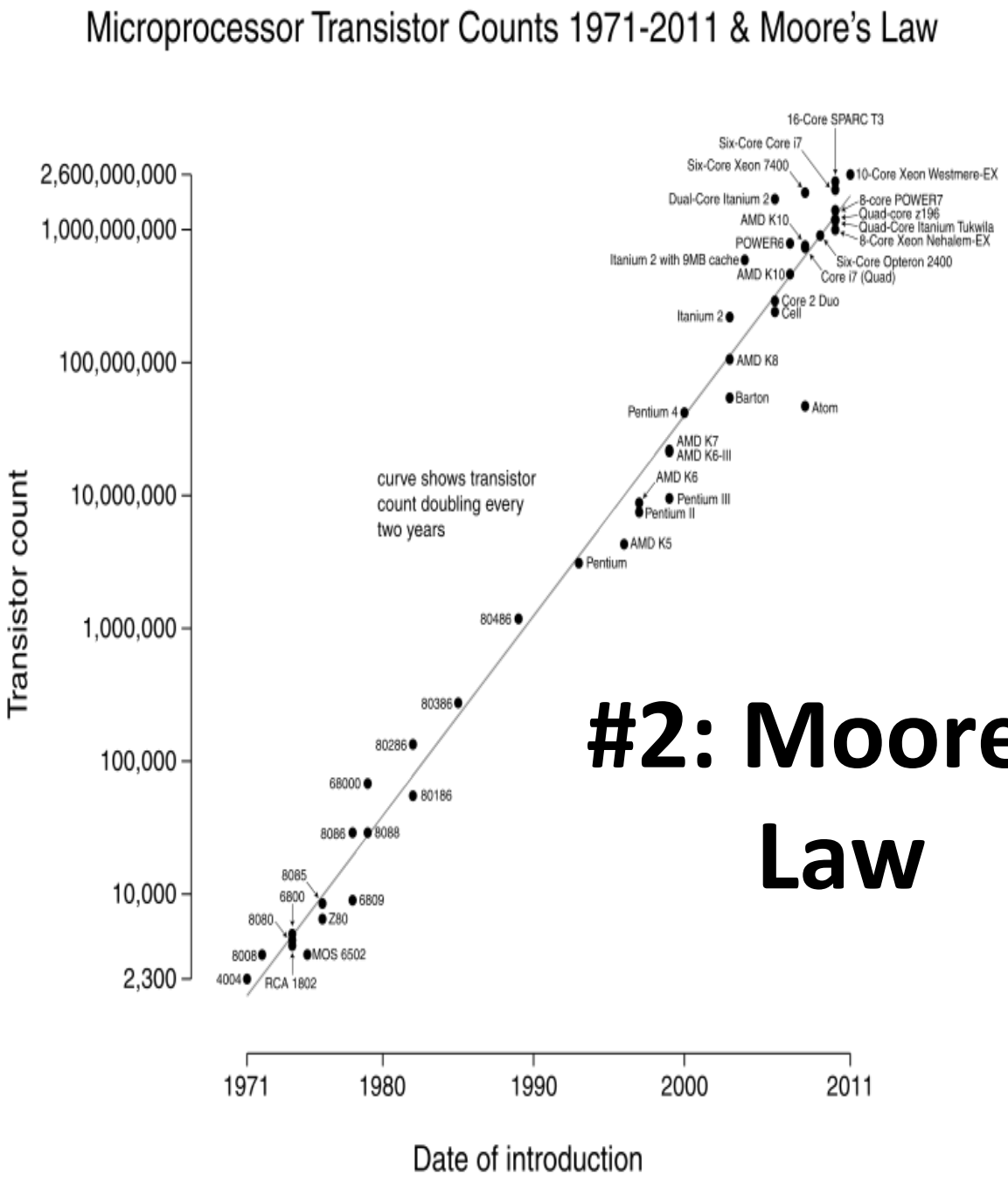

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

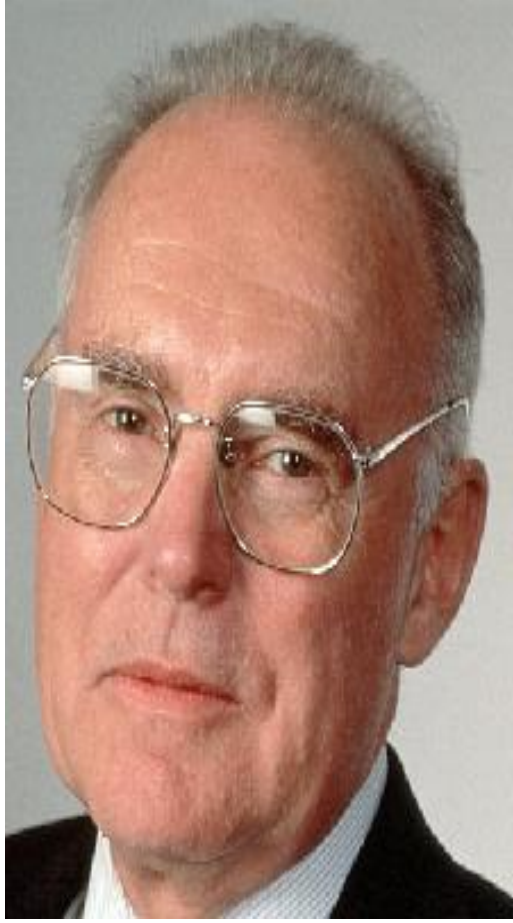
Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





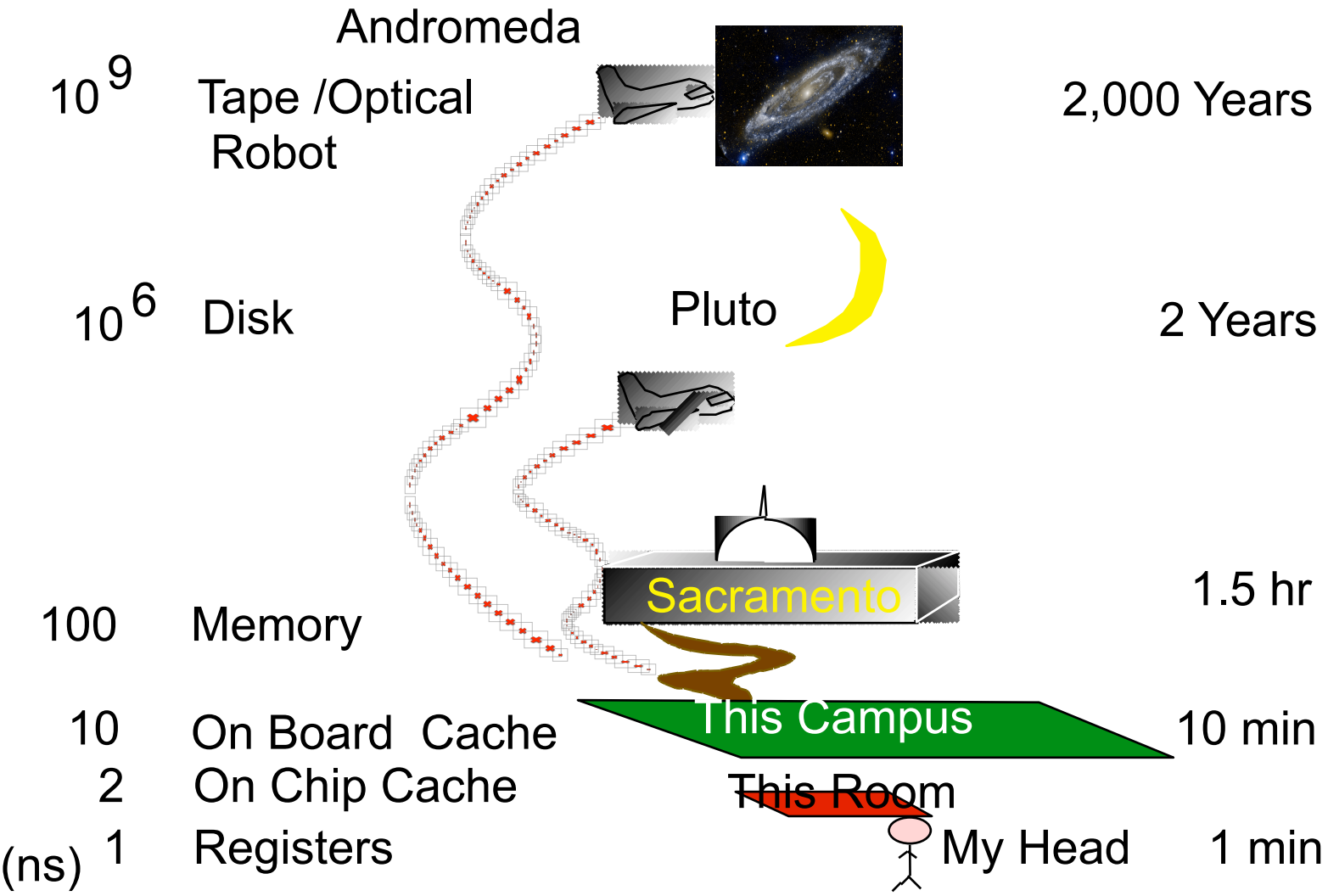
#2: Moore's Law

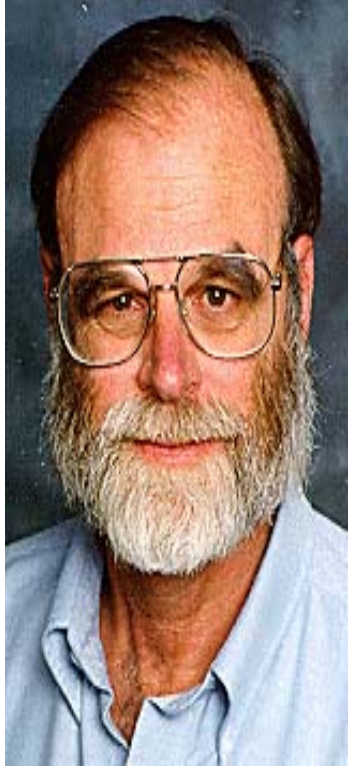


Gordon Moore
Intel Cofounder
B.S. Cal 1950!

Predicts:
2X Transistors / chip every 2 years

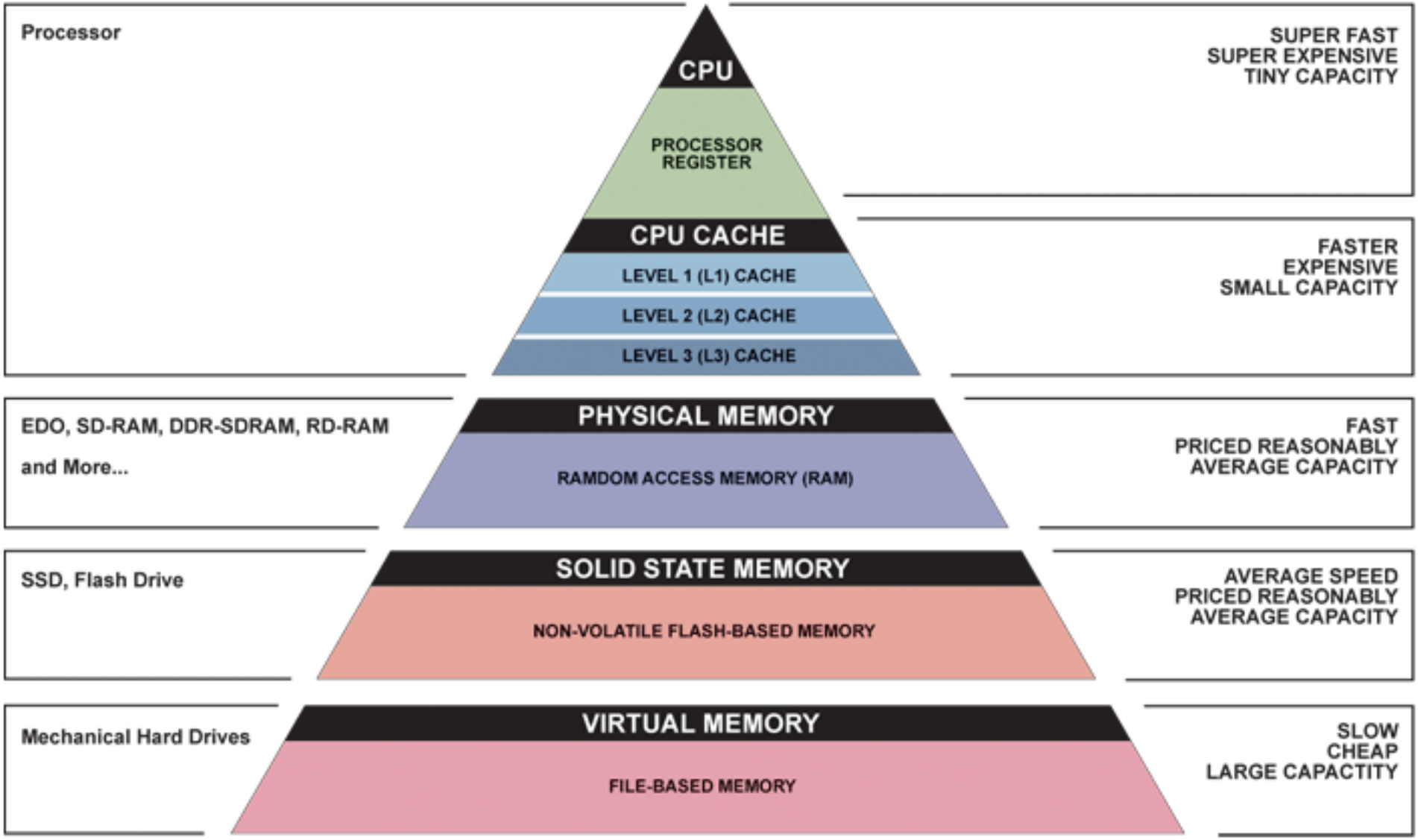
Jim Gray's Storage Latency Analogy: How Far Away is the Data?



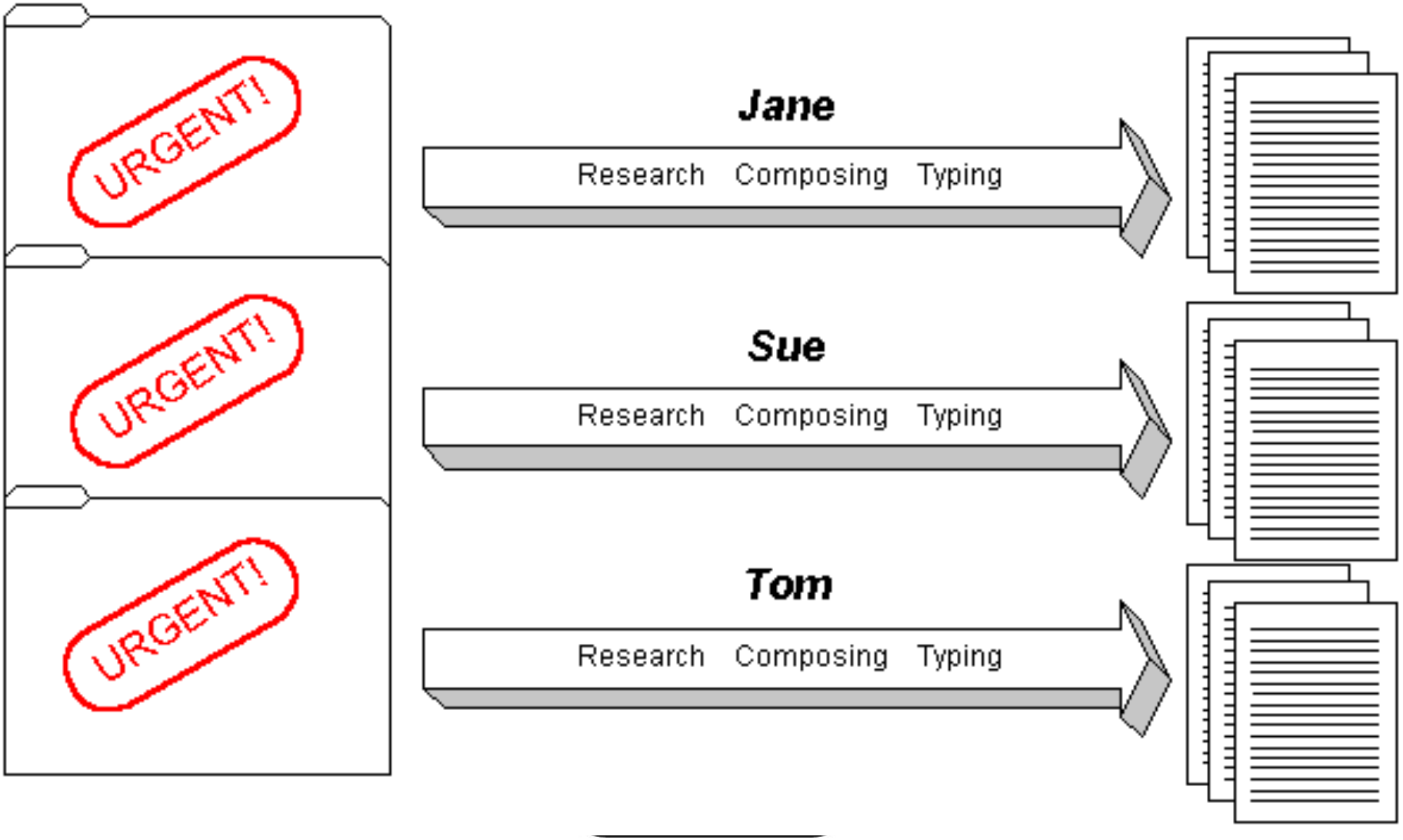


Jim Gray
Turing Award
B.S. Cal 1966
Ph.D. Cal 1969

Great Idea #3: Principle of Locality/ Memory Hierarchy



Great Idea #4: Parallelism



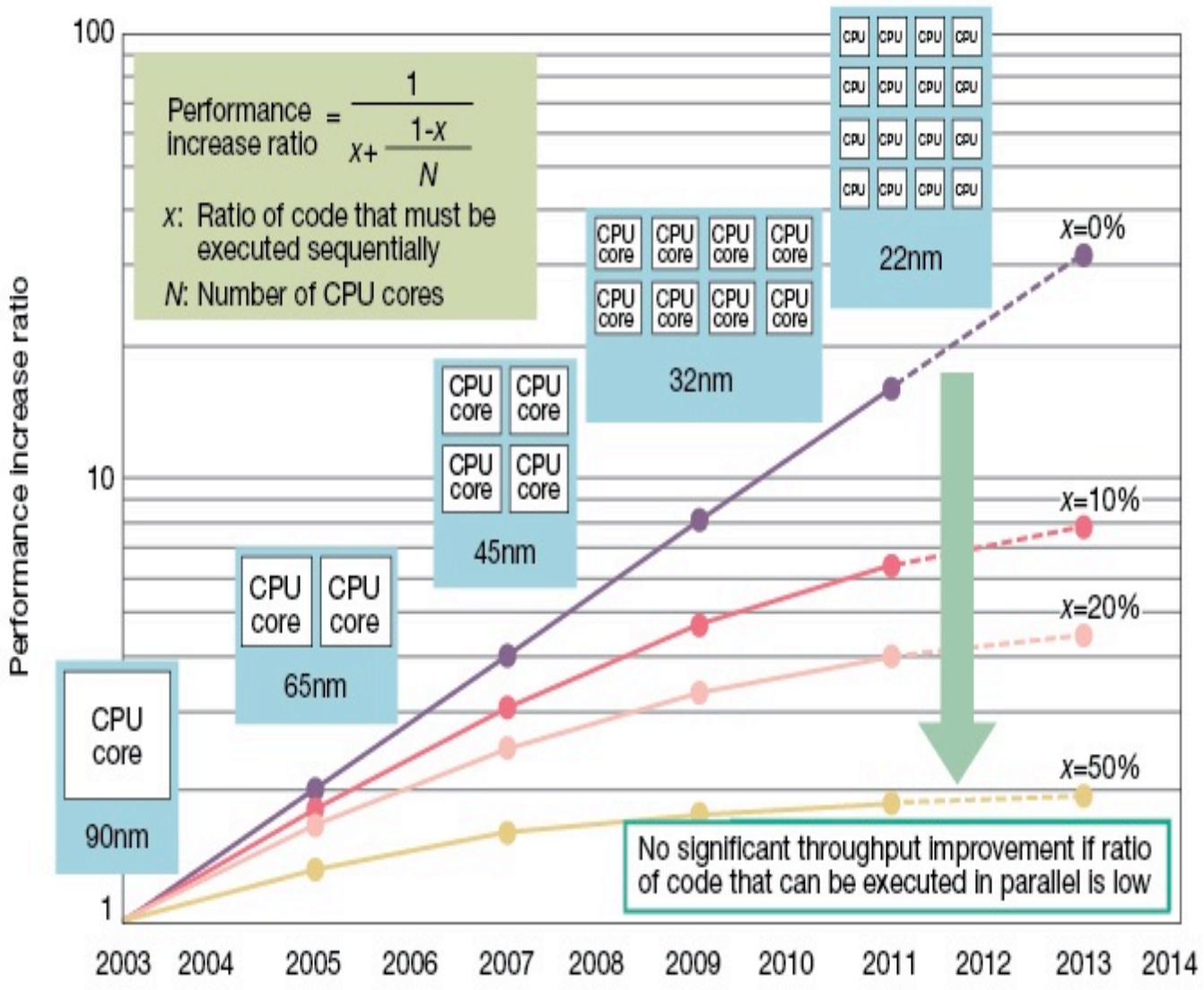


Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.

Gene Amdahl
Computer Pioneer

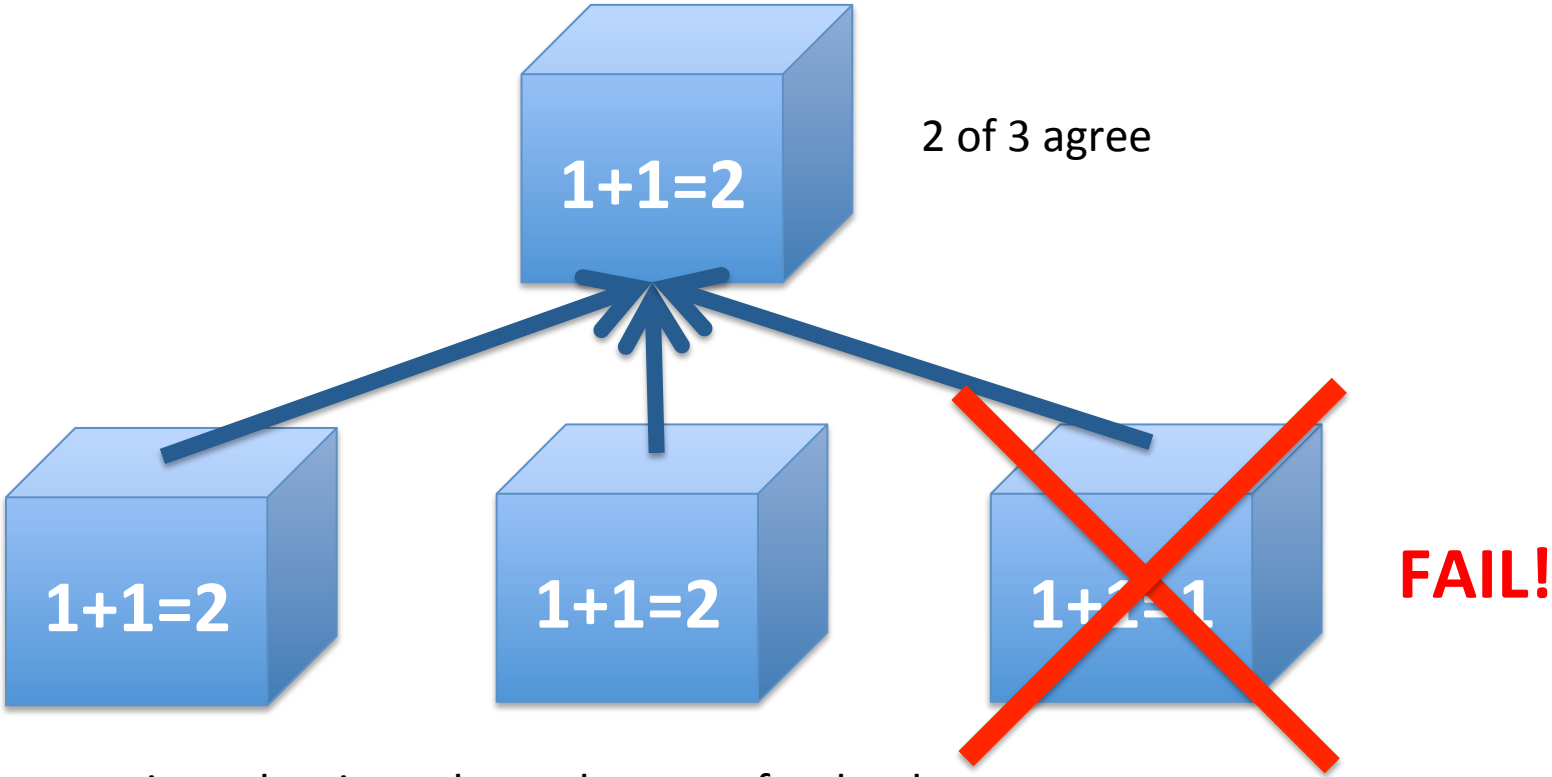
Caveat! Amdahl's Law

Great Idea #5: Performance Measurement and Improvement

- Matching application to underlying hardware to exploit:
  - Locality
  - Parallelism
  - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency
  - How long to set the problem up
  - How much faster does it execute once it gets going
  - It is all about *time to finish*

Great Idea #6: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail



Increasing transistor density reduces the cost of redundancy

Great Idea #6: Dependability via Redundancy

- Applies to everything from datacenters to storage to memory to instructors
  - Redundant datacenters so that can lose 1 datacenter but Internet service stays online
  - Redundant disks so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/RAID)
  - Redundant memory bits of so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

Number Representation

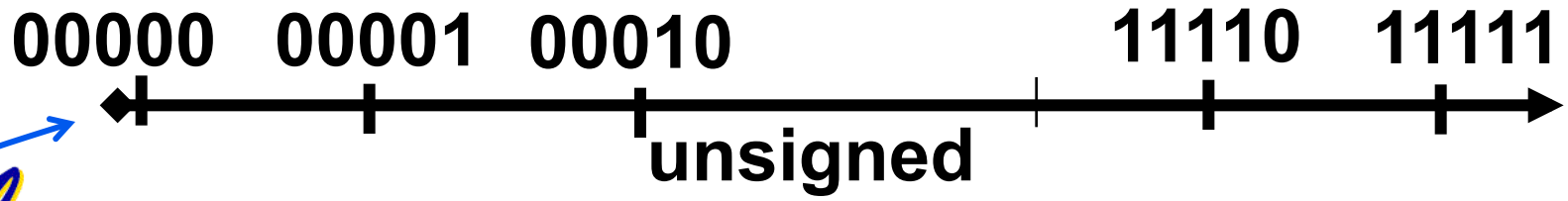
BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters ⇒ 5 bits ($2^5 = 32$)
  - upper/lower case + punctuation ⇒ 7 bits (in 8) ("ASCII")
  - standard code to cover all the world's languages ⇒ 8,16,32 bits ("Unicode") www.unicode.com

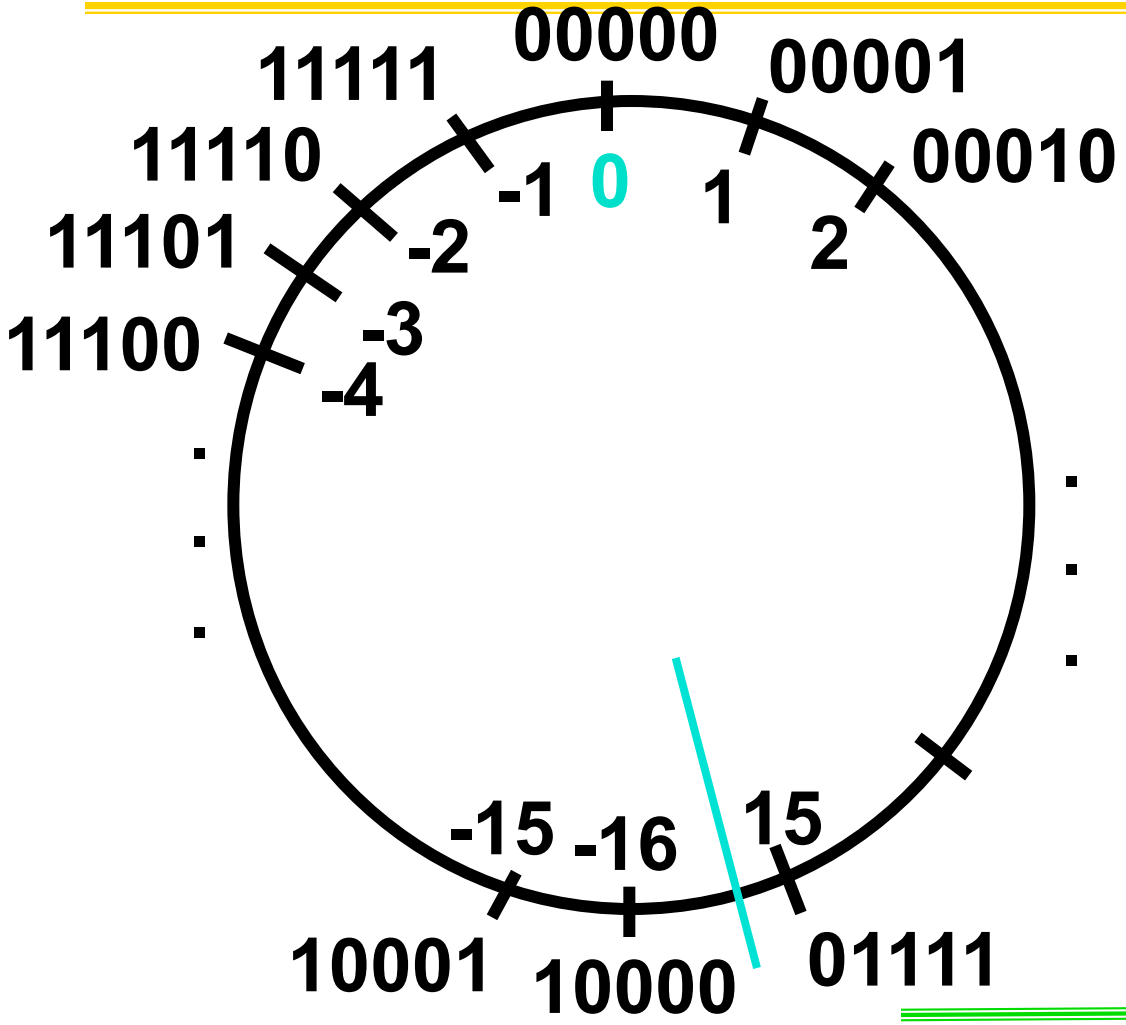
- Logical values?
  - 0 ⇒ False, 1 ⇒ True
- colors ? Ex: *Red (00)* *Green (01)* *Blue (11)*
- locations / addresses? commands?
- MEMORIZE: N bits ⇔ at most 2^N things

What if too big?

- Binary bit patterns above are simply representatives of numbers. Abstraction! Strictly speaking they are called "numerals".
- Numbers really have an ∞ number of digits
  - with almost all being same (00…0 or 11…1) except for a few of the rightmost digits
  - Just don't normally show leading digits
- If result of add (or -, *, /) cannot be represented by these rightmost HW bits, overflow is said to have occurred.



2's Complement Number "line": N = 5



- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?

Binary odometer

00000 00001 ... 01111

10000 ... 11110 11111

How best to represent -12.75?

a) 2s Complement (but shift binary pt)

b) Bias (but shift binary pt)

c) Combination of 2 encodings

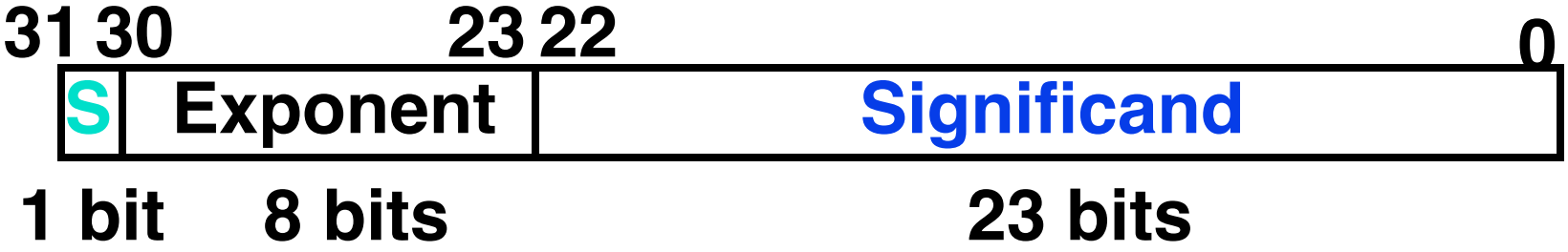
d) Combination of 3 encodings

e) We can't

Shifting binary point means "divide number by some power of 2. E.g., $11_{10} = 1011.0_2$ so $(11/4)_{10} = 2.75_{10} = 10.110_2$

Floating Point Representation (1/2)

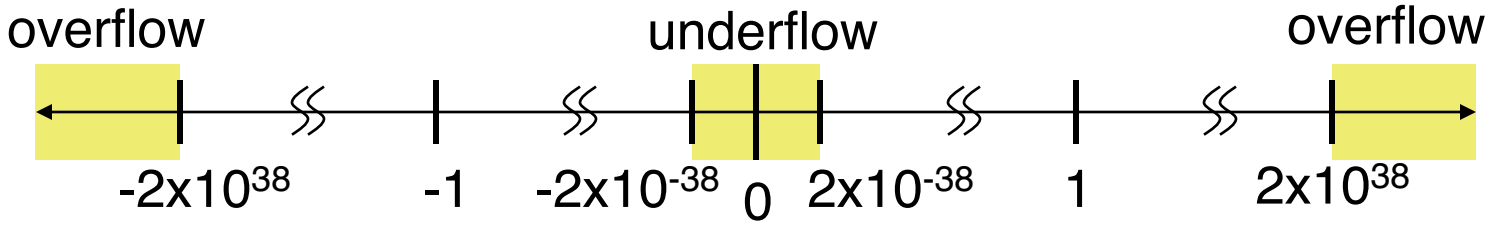
- Normal format: $+1.xxx\ldots x_{two} * 2^{yyy\ldots y_{two}}$
- Multiple of Word Size (32 bits)



- S represents Sign
  Exponent represents y's
  Significand represents x's
- Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

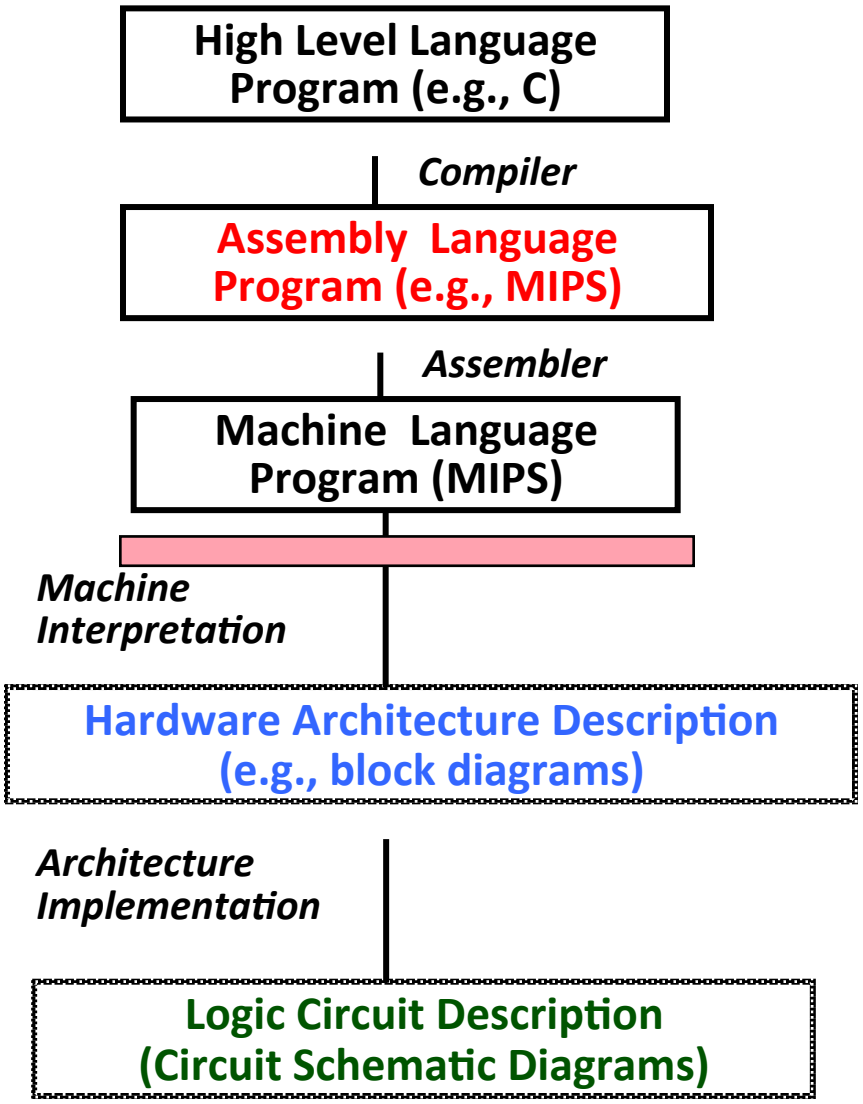
Floating Point Representation (2/2)

- **What if result too large?**
  - **(> 2.0×10^{38} , < -2.0×10^{38})**
  - **Overflow! ⇒ Exponent larger than represented in 8-bit Exponent field**
- **What if result too small?**
  - **(>0 & < 2.0×10^{-38} , <0 & > -2.0×10^{-38})**
  - **Underflow! ⇒ Negative exponent larger than represented in 8-bit Exponent field**



- **What would help reduce chances of overflow and/or underflow?**

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

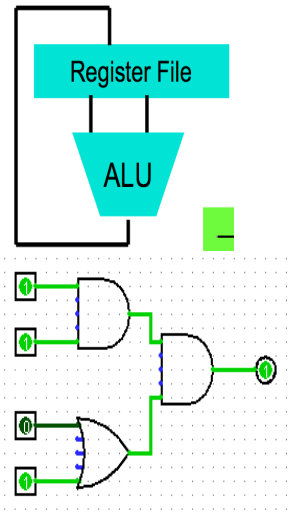



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



Overview of C

Compilation : Overview

C *compilers* take C and convert it into an architecture specific machine code (string of 1s and 0s).

- **Unlike Java which converts to architecture independent bytecode.**
- **Unlike most Python environments which interpret the code.**
- **These differ mainly in when your program is converted to machine instructions.**
- **For C, generally a 2 part process of compiling .c files to .o files, then linking the .o files into executables. Assembling is also done (but is hidden, i.e., done automatically, by default)**

Compilation : Advantages

- **Great run-time performance**: generally much faster than Python or Java for comparable code (because it optimizes for a given architecture)
- **OK compilation time**: enhancements in compilation procedure (`Makefiles`) allow only modified files to be recompiled

Compilation : Disadvantages

- All compiled files (including the executable) are architecture specific, depending on *both* the CPU type and the operating system.
- Executable must be rebuilt on each new system.
  - Called "porting your code" to a new architecture.
- The "change→compile→run [repeat]" iteration cycle is slow

C Syntax: `main`

- To get the main function to accept arguments, use this:

```
int main (int argc, char *argv[])
```

- What does this mean?
  - `argc` will contain the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:

```
unix% sort myFile
```

  - `argv` is a pointer to an array containing the arguments as strings (more on pointers later).

C Syntax: Variable Declarations

- Very similar to Java, but with a few minor but important differences
- A variable may be initialized in its declaration; if not, it holds garbage!
- Examples of declarations:
  - correct: {

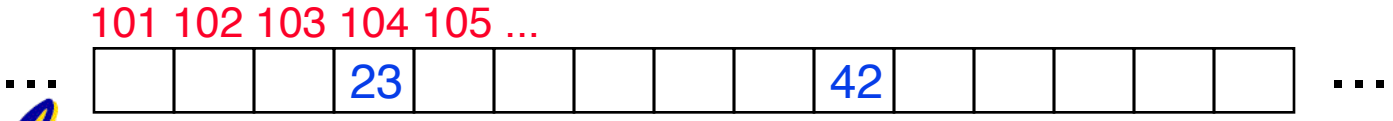
```
int a = 0, b = 10;

...
```

  - Incorrect:* `for (int i = 0; i < 10; i++)`

*C99 overcomes these limitations

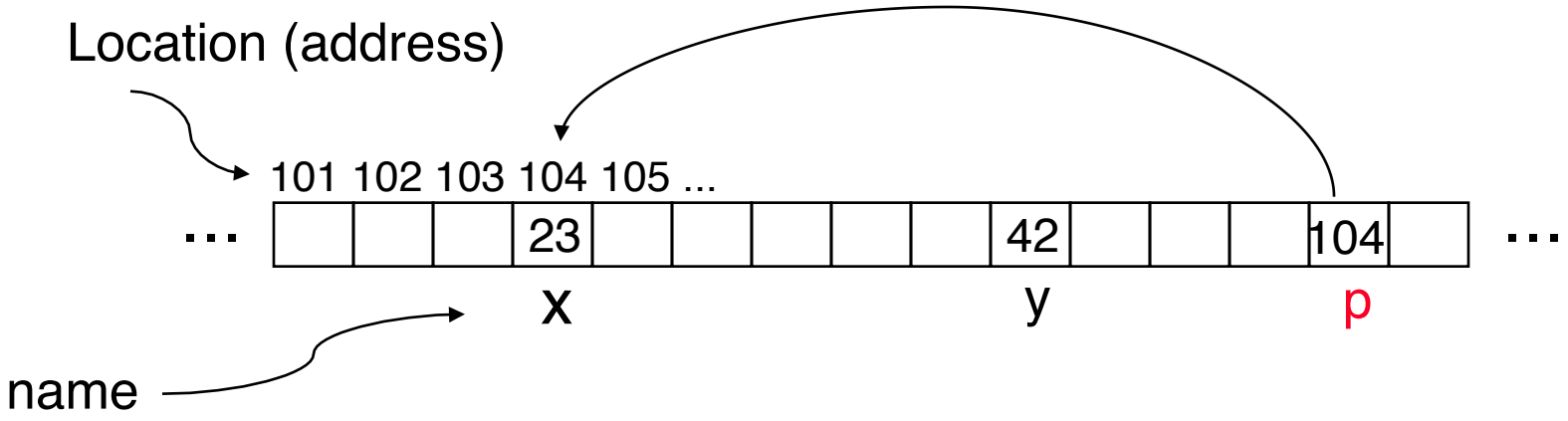
Address vs. Value

- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
  - Do you think they use signed or unsigned numbers? Negative address?!
- Don't confuse the address referring to a memory location with the value stored in that location.



Pointers

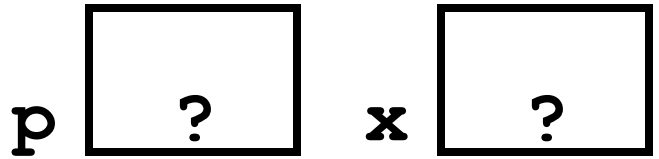
- **An address refers to a particular memory location. In other words, it points to a memory location.**
- **Pointer: A variable that contains the address of a variable.**



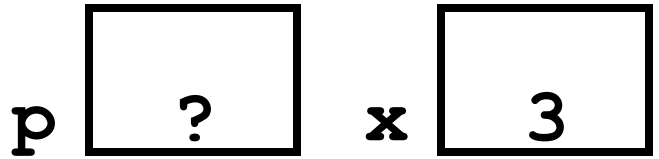
Pointers

- How to create a pointer:
  - & operator: get address of a variable

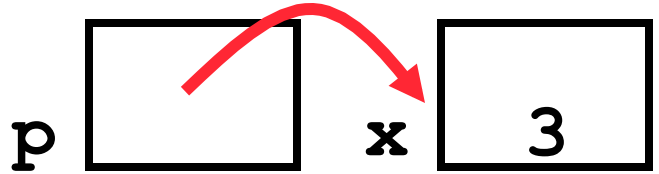
```
int *p, x;
```




```
x = 3;
```




```
p =&x;
```



- How get a value pointed to?
  - * "dereference operator": get value pointed to

```
printf("p points to %d\n",*p);
```

Pointers and Parameter Passing

- Java and C pass parameters "by value"
  - procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original

```
void addOne (int x) {
    x = x + 1;
}

int y = 3;

addOne(y);
```

y is still = 3

Pointers and Parameter Passing

- How to get a function to change a value?

```
void addOne (int *p) {
  *p = *p + 1;
}

int y = 3;


addOne(&y);
```

y is now = 4

Pointers

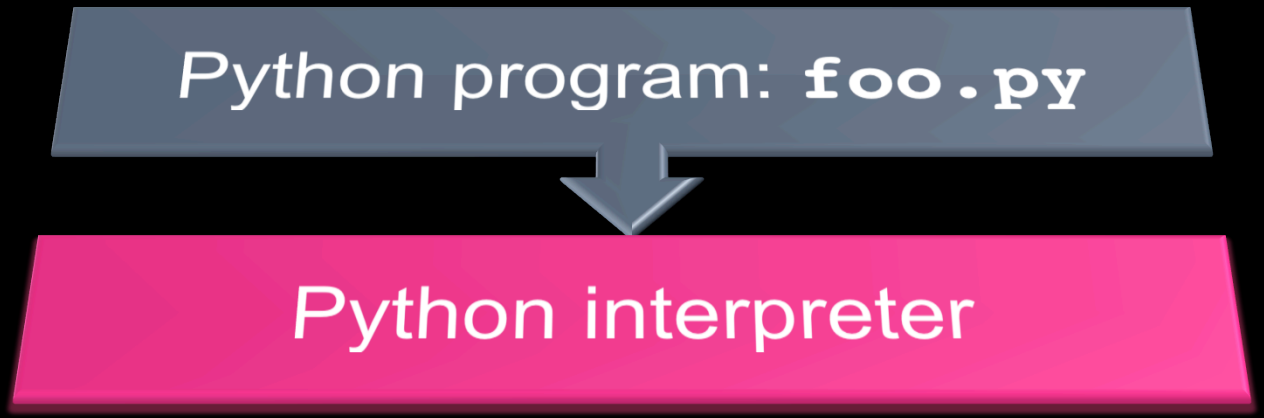
- Pointers are used to point to any data type (`int`, `char`, a `struct`, etc.).
- Normally a pointer can only point to one type (`int`, `char`, a `struct`, etc.).
  - `void *` is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs… and security issues… and a lot of other bad things!

C Syntax: True or False?

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (pointer: more on this later)
  - no such thing as a Boolean*
- What evaluates to TRUE in C?
  - everything else…
  - (same idea as in scheme: only `#f` is false, everything else is true!)

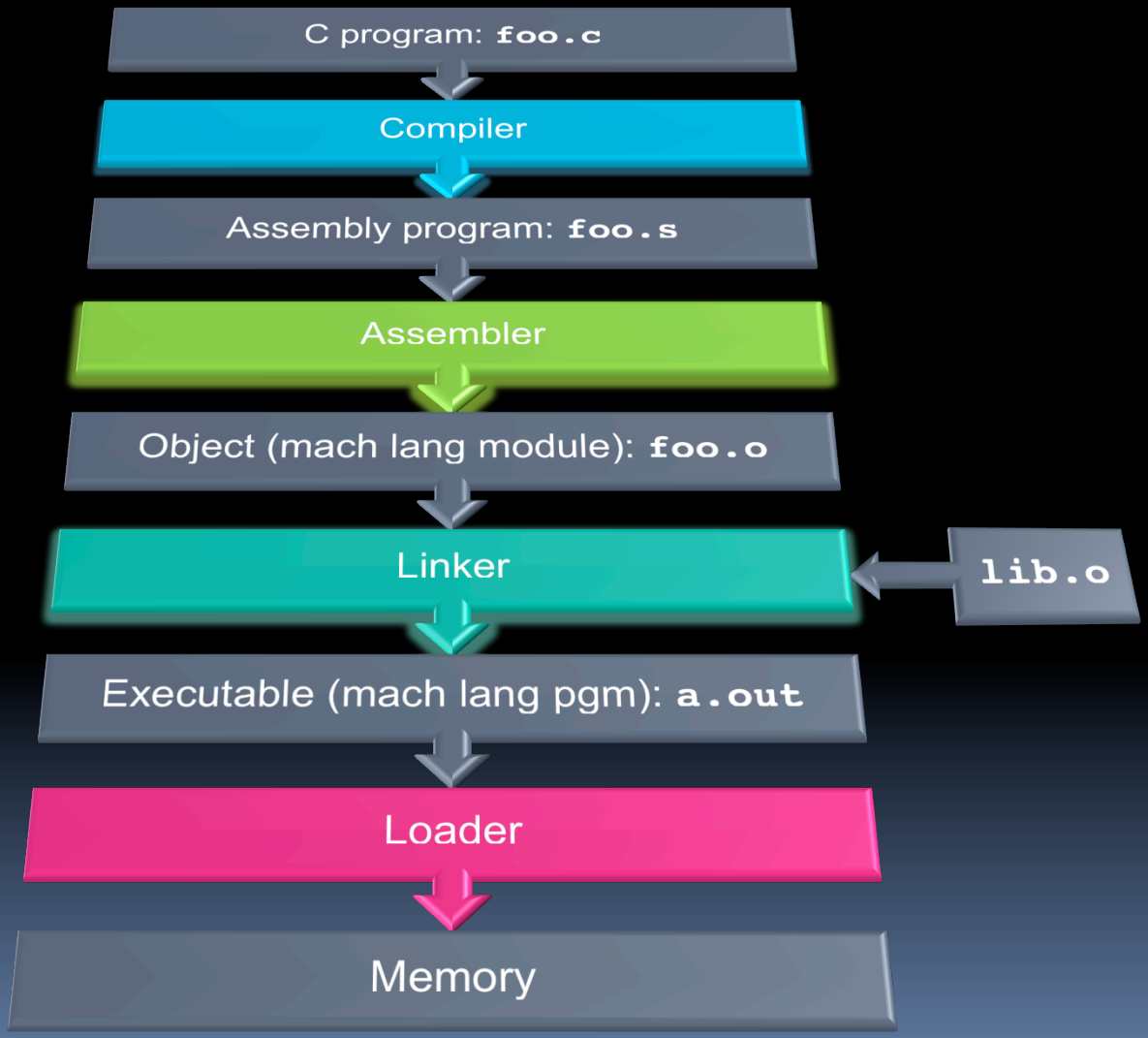
*Boolean types provided by C99's `stdbool.h`

Interpretation

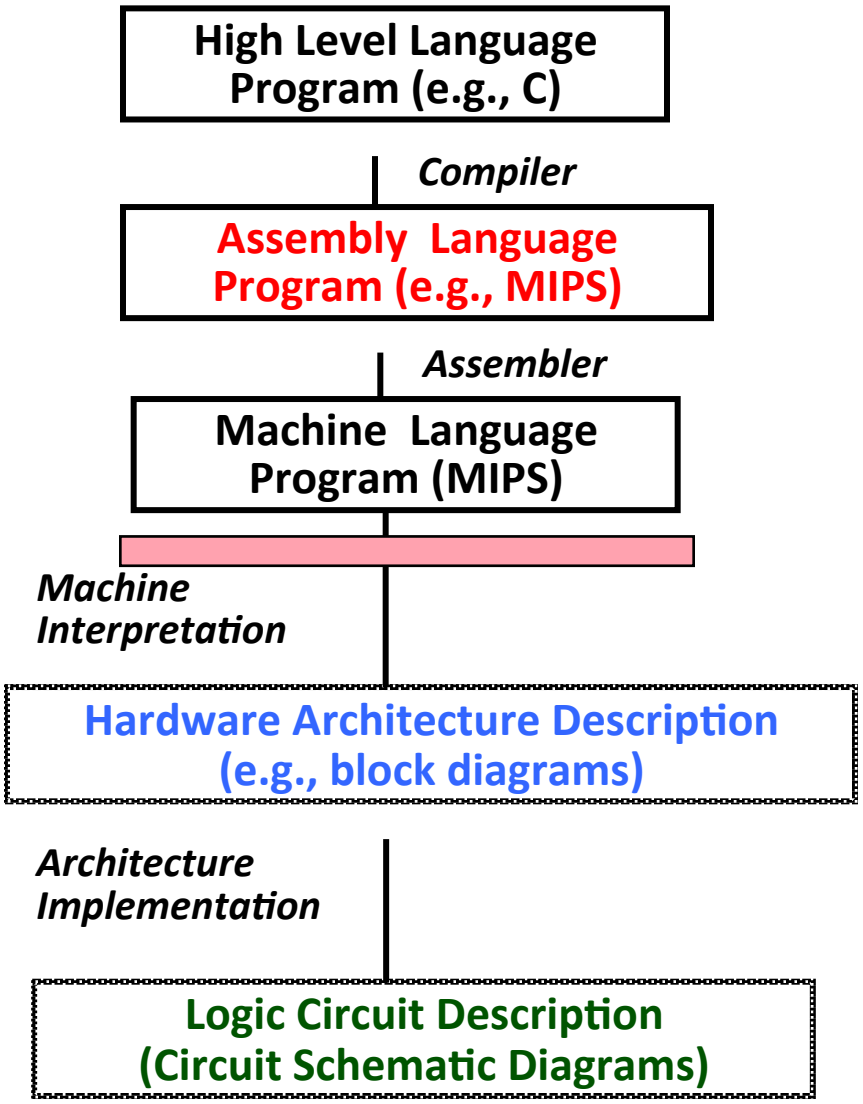


- Python interpreter is just a program that reads a python program and performs the functions of that python program.

Steps in compiling a C program



Great Idea #1: Abstraction (Levels of Representation/Interpretation)

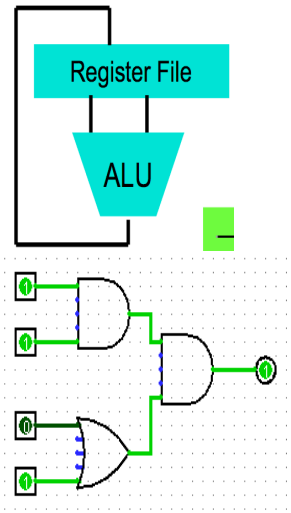



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



Assembly Language

Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture* (*ISA*).
  - Examples: Intel 80x86 (Pentium 4), IBM/ Motorola PowerPC (old Macintosh), MIPS, Intel IA64, ...

Garcia, Fall 2014 © UCB

Assembly Variables: Registers (1/4)

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are registers
  - limited number of special locations built directly into the hardware
  - operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 nano second - light travels 30cm in 1 ns!!!)

Assembly Variables: Registers (2/4)

- Drawback: Since registers are in hardware, there are a predetermined number of them
  - Solution: MIPS code must be very carefully put together to efficiently use registers
- 32 registers in MIPS
  - Why 32? Smaller is faster
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS

Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:
  - $0, $1, $2, … $30, $31

MIPS Addition and Subtraction (1/4)

- Syntax of Instructions:
  - One two, three, four
  - where:
  - 1) operation by name
  - 2) operand getting result (“destination”)
  - 3) 1st operand for operation (“source1”)
  - 4) 2nd operand for operation (“source2”)
- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

Addition and Subtraction of Integers (2/4)

- **Addition in Assembly**
  - Example: `add $s0,$s1,$s2` (in MIPS)
  - Equivalent to: a = b + c (in C)
  - where C variables ⇔ MIPS registers are: a ⇔ $s0, b ⇔ $s1, c ⇔ $s2
- **Subtraction in Assembly**
  - Example: `sub $s3,$s4,$s5` (in MIPS)
  - Equivalent to: d = e - f (in C)
  - where C variables ⇔ MIPS registers are: d ⇔ $s3, e ⇔ $s4, f ⇔ $s5

Garcia, Fall 2014 © UCB

Addition and Subtraction of Integers (3/4)

- How to do the following C statement?
- a = b + c + d - e;
- Break into multiple instructions
  - add $t0, $s1, $s2 *# temp = b + c*
  - add $t0, $t0, $s3 *# temp = temp + d*
  - sub $s0, $t0, $s4 *# a = temp - e*
- Notice: A single line of C may break up into several lines of MIPS.
- Notice: Everything after the hash mark on each line is ignored (comments)

Types of Branches

- **Branch – change of control flow**
- **Conditional Branch – change control flow depending on outcome of comparison**
  - **branch *if* equal (`beq`) or branch *if not* equal (`bne`)**
- **Unconditional Branch – always branch**
  - **a MIPS instruction for this*: jump* (`j`)**

Inequalities in MIPS

- Until now, we've only tested equalities (== and != in C). General programs need to test < and > as well.
- Introduce MIPS Inequality Instruction:

  "Set on Less Than"

  Syntax: `slt reg1,reg2,reg3`

  Meaning:
  ```
  if (reg2 < reg3)
      reg1 = 1;
  else reg1 = 0;
  ```

  "set" means "change to 1",
  "reset" means "change to 0".

R-Format Example (2/2)

- **MIPS Instruction:**

```
add    $8,$9,$10
```

Decimal number per field representation:

0	9	10	8	0	32

Binary number per field representation:

000000	01001	01010	01000	00000	100000

hex representation: $012A\ 4020_{hex}$

decimal representation: $19,546,144_{ten}$

Called a Machine Language Instruction

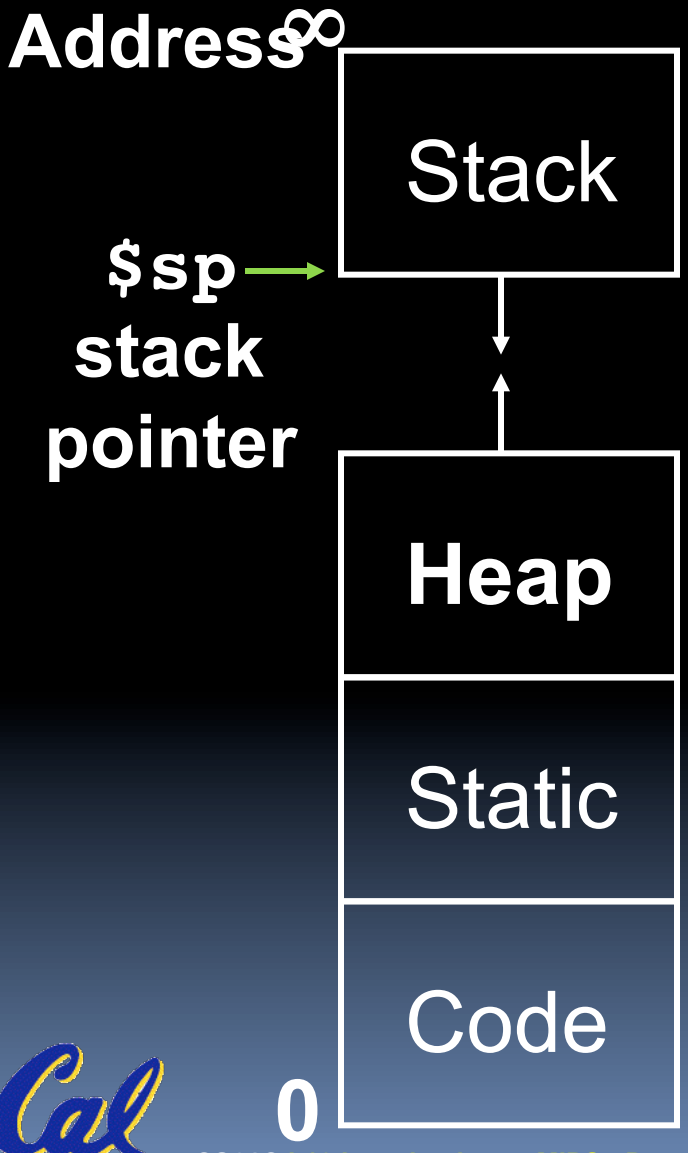
Review

- MIPS Machine Language Instruction:
  32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Branches use PC-relative addressing, Jumps use absolute addressing.
- Disassembly is simple and starts by decoding opcode field. (more next lecture)

C Memory Allocation



Space for local vars, saved procedure information

Explicitly created space, i.e., `malloc()`

Variables declared once per program; e.g., globals (doesn't change size)

Program (doesn't change size)

Function Call Bookkeeping

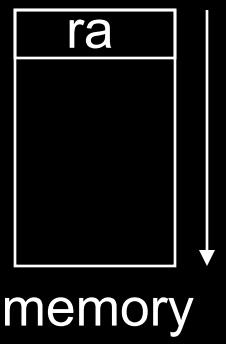
- Registers play a major role in keeping track of information for function calls.
- Register conventions:
  - Return address `$ra`
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`
  - Local variables `$s0, $s1, … , $s7`
- The stack is also used; more later.

Basic Structure of a Function

Prologue

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be
```

Body... (call other functions…)



Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp,$sp, framesize
jr $ra
```

Steps for Making a Procedure Call

1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. `jal` call
4. Restore values from stack.

MIPS Registers

The constant 0	$0	$zero
Reserved for Assembler	$1	$at
Return Values	$2-$3	$v0-$v1
Arguments	$4-$7	$a0-$a3
Temporary	$8-$15	$t0-$t7
Saved	$16-$23	$s0-$s7
More Temporary	$24-$25	$t8-$t9
Used by Kernel	$26-27	$k0-$k1
Global Pointer	$28	$gp
Stack Pointer	$29	$sp
Frame Pointer	$30	$fp
Return Address	$31	$ra

(From COD green insert)

Use names for registers -- code is clearer!

Putting it all together

Recursive Function Factorial

```
int fact (int n)
{
  if (n < 1) return (1);
    else return (n * fact(n-1));
}
```

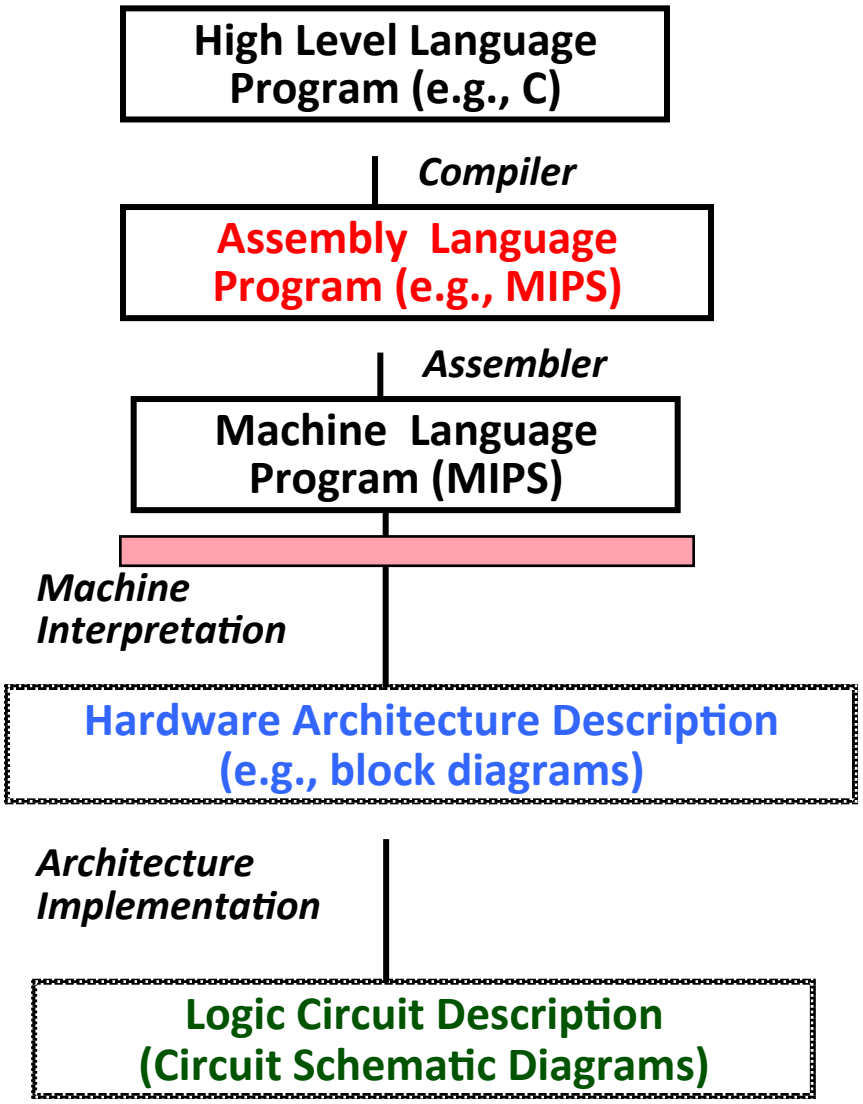
Recursive Function Factorial

```
Fact:
   # adjust stack for 2 items
   addi $sp,$sp,-8
   # save return address
   sw $ra, 4($sp)
   # save argument n
   sw $a0, 0($sp)
   # test for n < 1
   slti $t0,$a0,1
   # if n >= 1, go to L1
   beq $t0,$zero,L1
   # Then part (n==1) return 1
   addi $v0,$zero,1
   # pop 2 items off stack
   addi $sp,$sp,8
   # return to caller
   jr $ra
```

```
L1:
   # Else part (n >= 1)
   # arg. gets (n - 1)
   addi $a0,$a0,-1
   # call fact with (n - 1)
   jal fact
   # return from jal: restore n
   lw $a0, 0($sp)
   # restore return address
   lw $ra, 4($sp)
   # adjust sp to pop 2 items
   addi $sp, $sp,8
   # return n * fact (n - 1)
   mul $v0,$a0,$v0
   # return to the caller
   jr $ra
```

mul is a pseudo instruction

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

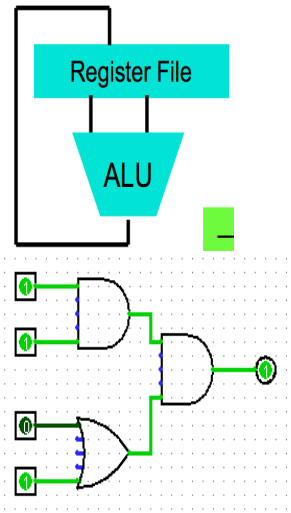



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



Circuit Description

Synchronous Digital Systems

Hardware of a processor, such as the MIPS, is an example of a Synchronous Digital System

Synchronous:

- All operations coordinated by a central clock
  - "Heartbeat" of the system!

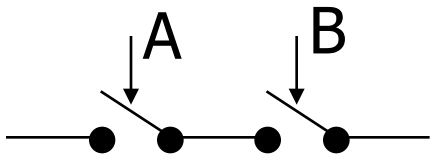
Digital:

- Represent all values by discrete values
- Two binary digits: 1 and 0
- Electrical signals are treated as 1's and 0's
  - 1 and 0 are complements of each other
- High /low voltage for true / false, 1 / 0

Switches (cont'd)

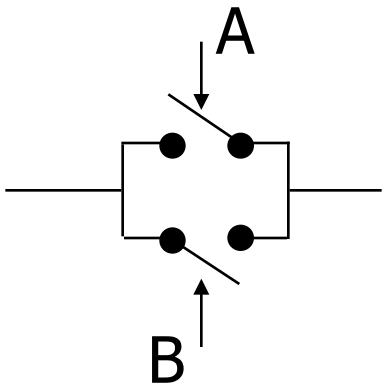
- Compose switches into more complex ones (Boolean functions):

AND



Z ≡ A and B

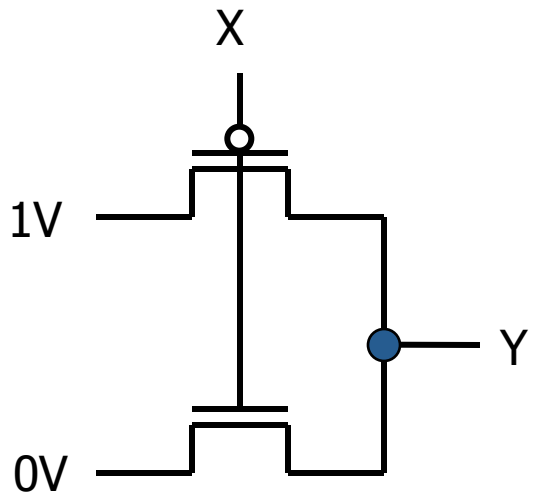
OR



Z ≡ A or B

CMOS Networks

p-channel transistor
closed when voltage at Gate is low
opens when:
voltage(Gate) > voltage (Threshold)



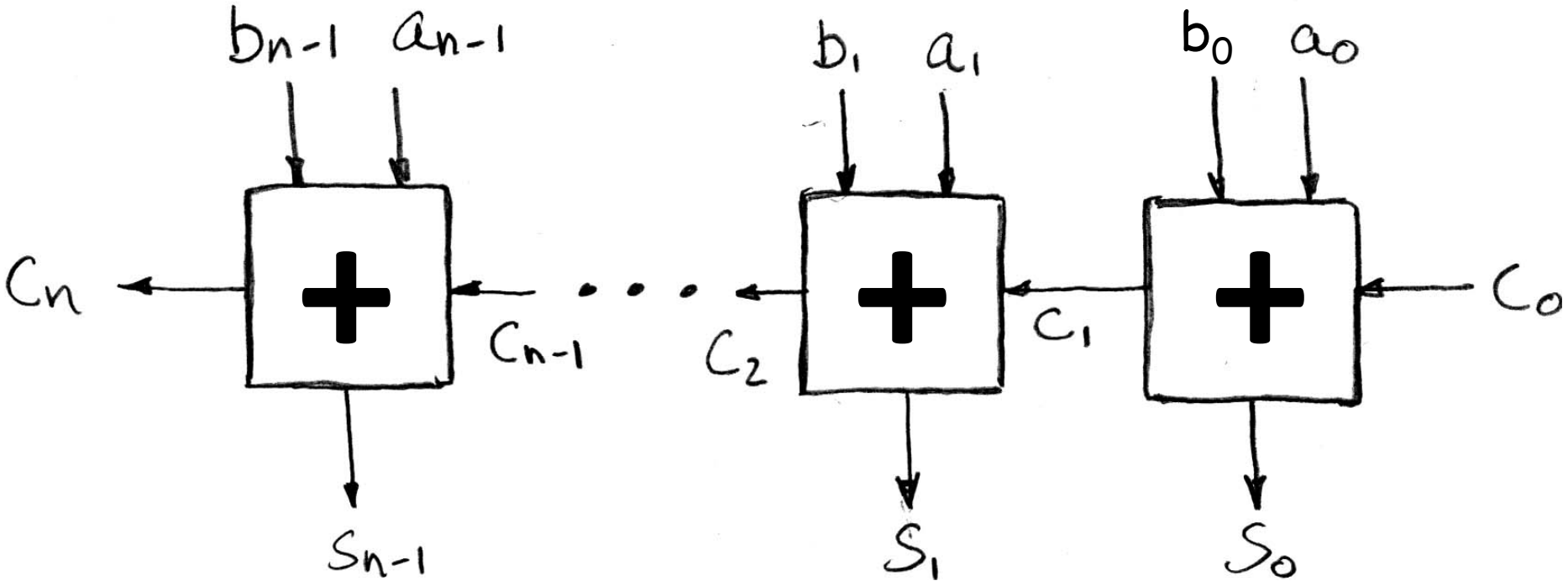
n-channel transitor
open when voltage at Gate is low
closes when:
voltage(Gate) > voltage (Threshold)

what is the relationship between x and y?

x	y
0 Volt (GND)	1 Volt (Vdd)
1 Volt (Vdd)	0 Volt (GND)

Called an *inverter* or *not gate*

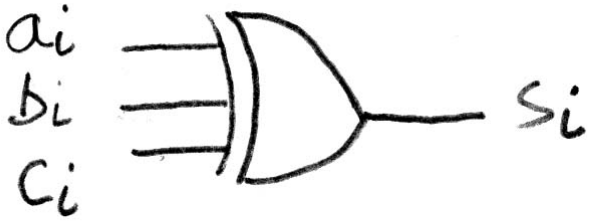
N 1-bit adders ⇒ 1 N-bit adder

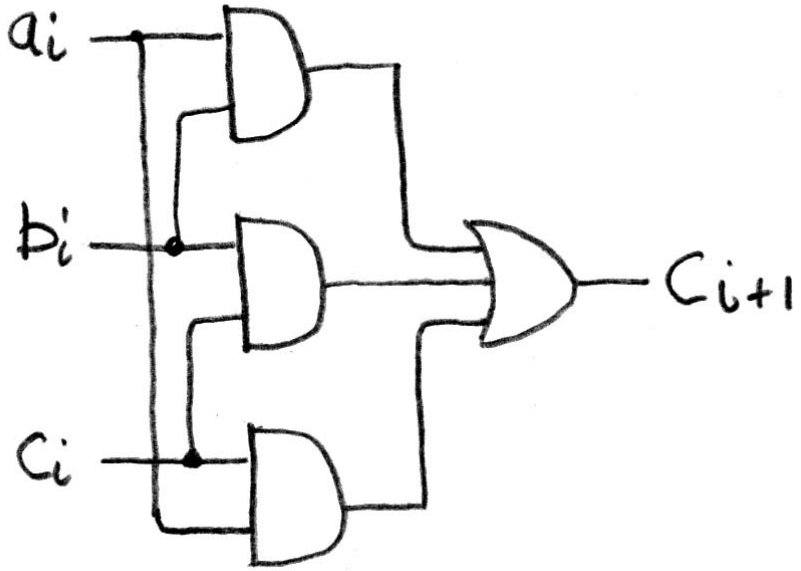


What about overflow?

Overflow = c_n?

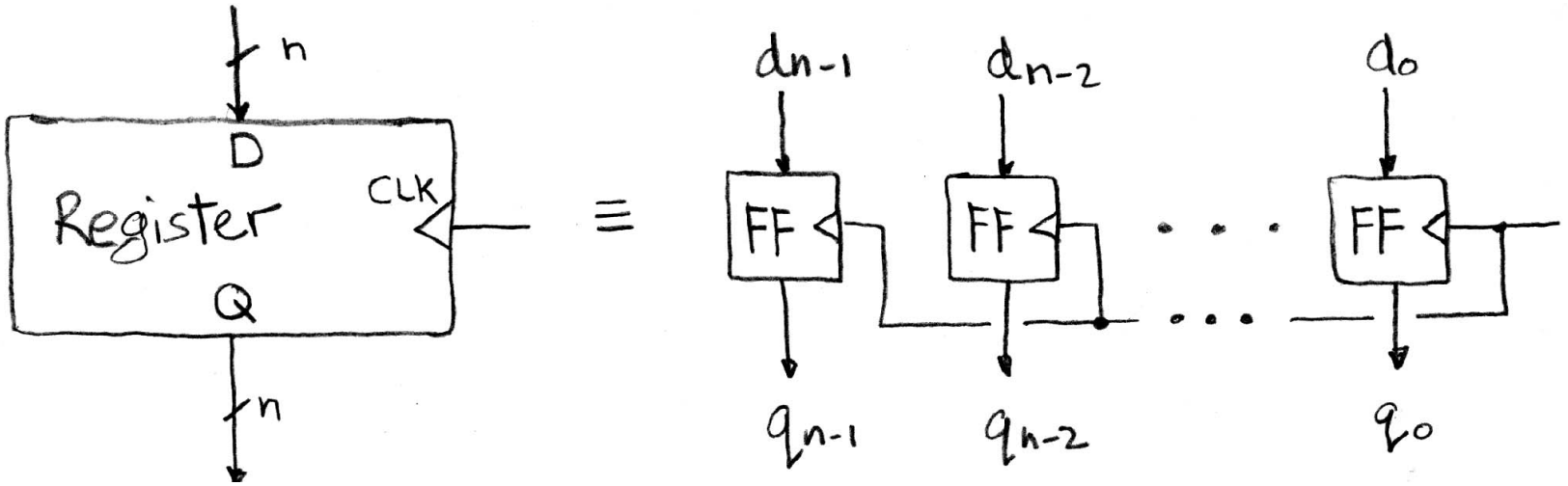
Adder/Subtractor – One-bit adder (2/2)





$$
\begin{aligned}
s_i &= \mathrm{XOR}(a_i, b_i, c_i) \\
c_{i+1} &= \mathrm{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i
\end{aligned}
$$

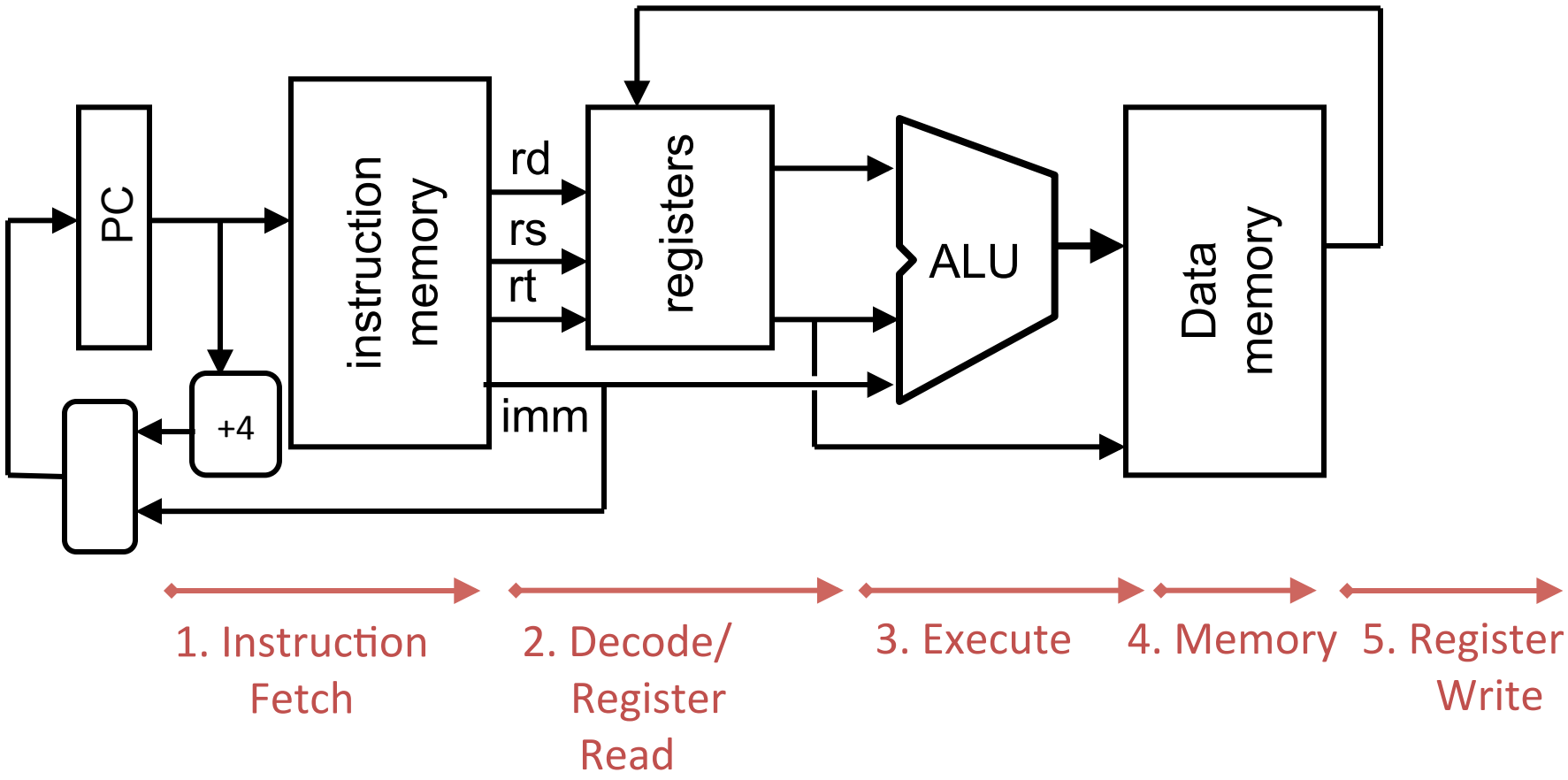
Register Internals



- n instances of a "Flip-Flop"
- Flip-flop name because the output flips and flops between 0 and 1
- D is "data input", Q is "data output"
- Also called "D-type Flip-Flop"

Datapath

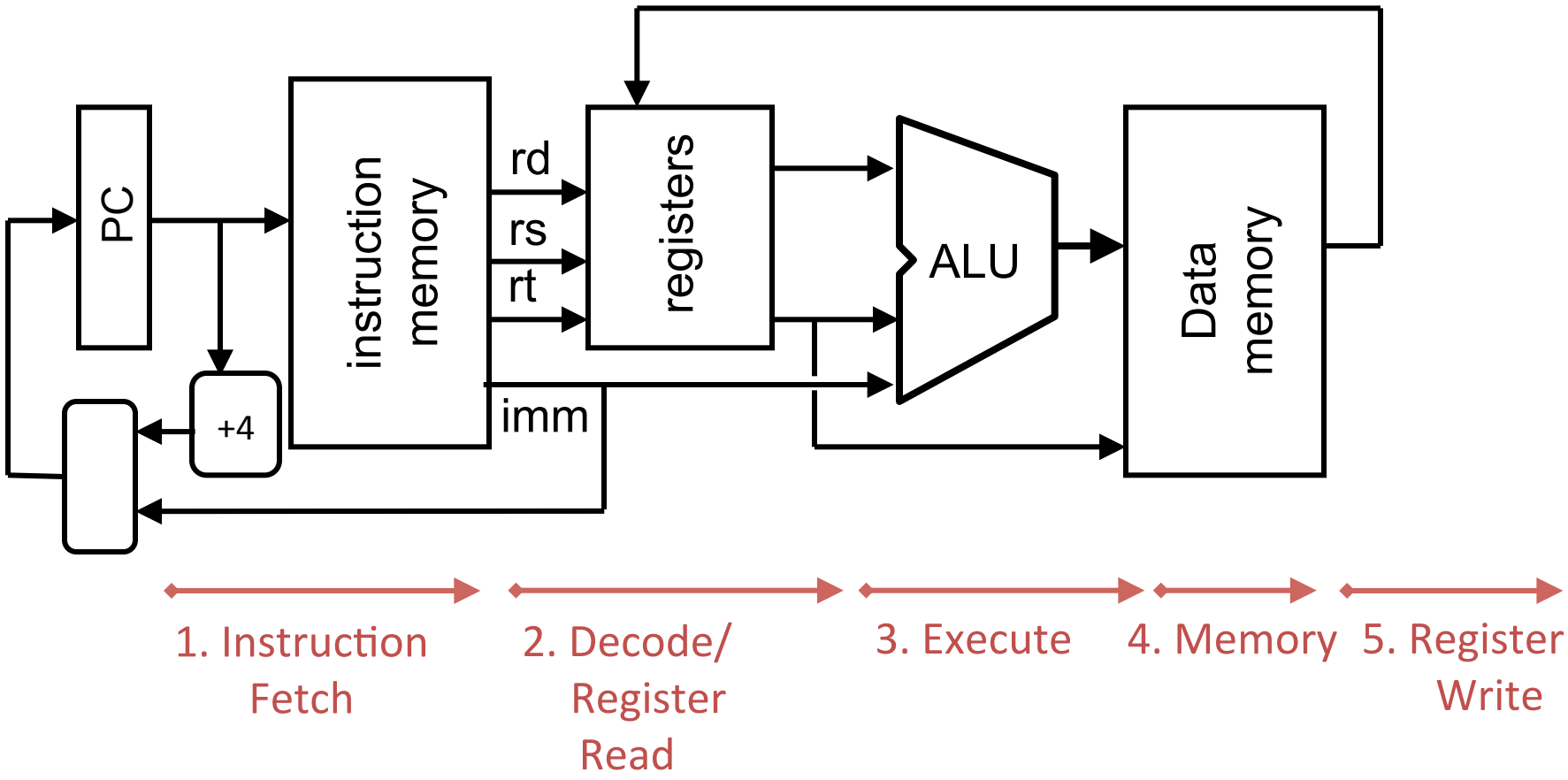
Stages of Execution on Datapath



Stages of Execution (1/5)

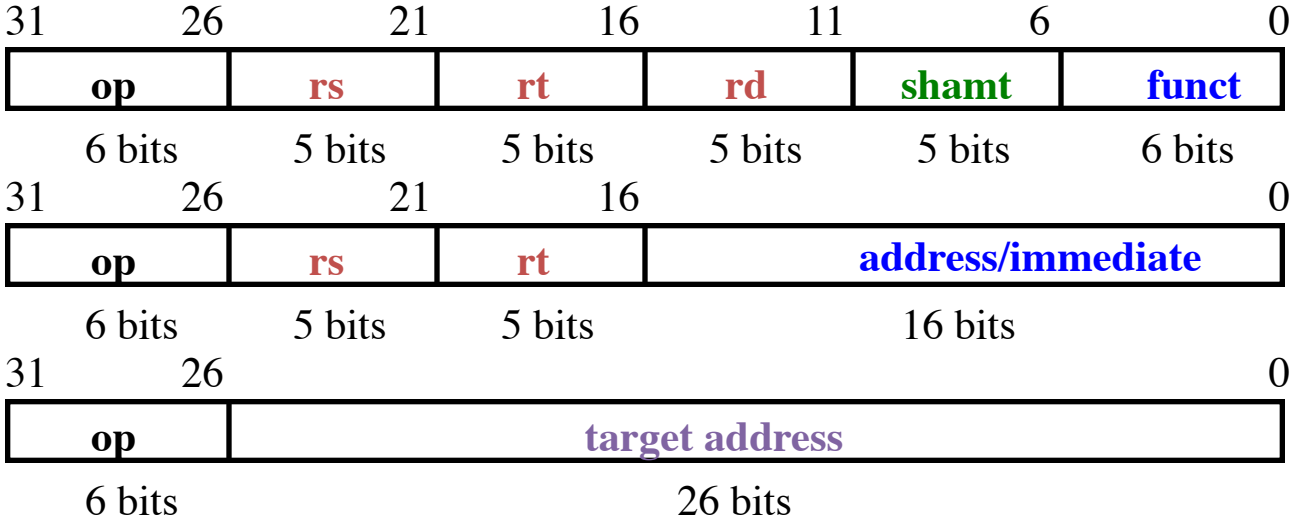
- There is a wide variety of MIPS instructions: so what general steps do they have in common?
- Stage 1: Instruction Fetch
  - no matter what the instruction, the 32-bit instruction word must first be fetched from memory (the cache-memory hierarchy)
  - also, this is where we Increment PC (that is, PC = PC + 4, to point to the next instruction: byte addressing so + 4)

Stages of Execution on Datapath



The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. 3 formats:
  - R-type
  - I-type
  - J-type

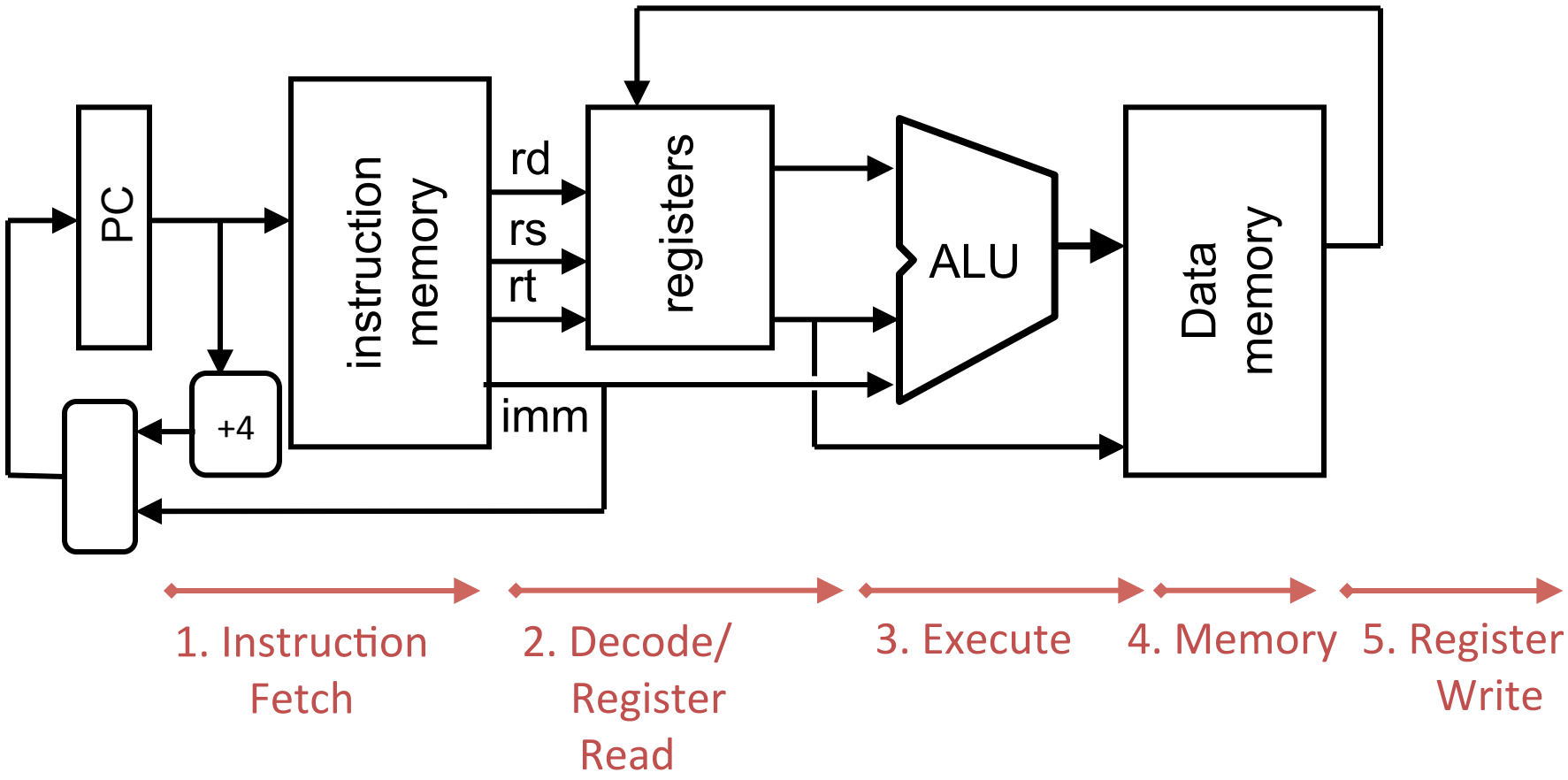


- The different fields are:
  - op: operation ("opcode") of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the "op" field
  - address / immediate: address offset or immediate value
  - target address: target address of jump instruction

Stages of Execution (2/5)

- Stage 2: Instruction Decode
  - upon fetching the instruction, we next gather data from the fields (decode all necessary instruction data)
  - first, read the opcode to determine instruction type and field lengths
  - second, read in data from all necessary registers
    - for add, read two registers
    - for addi, read one register
    - for jal, no reads necessary

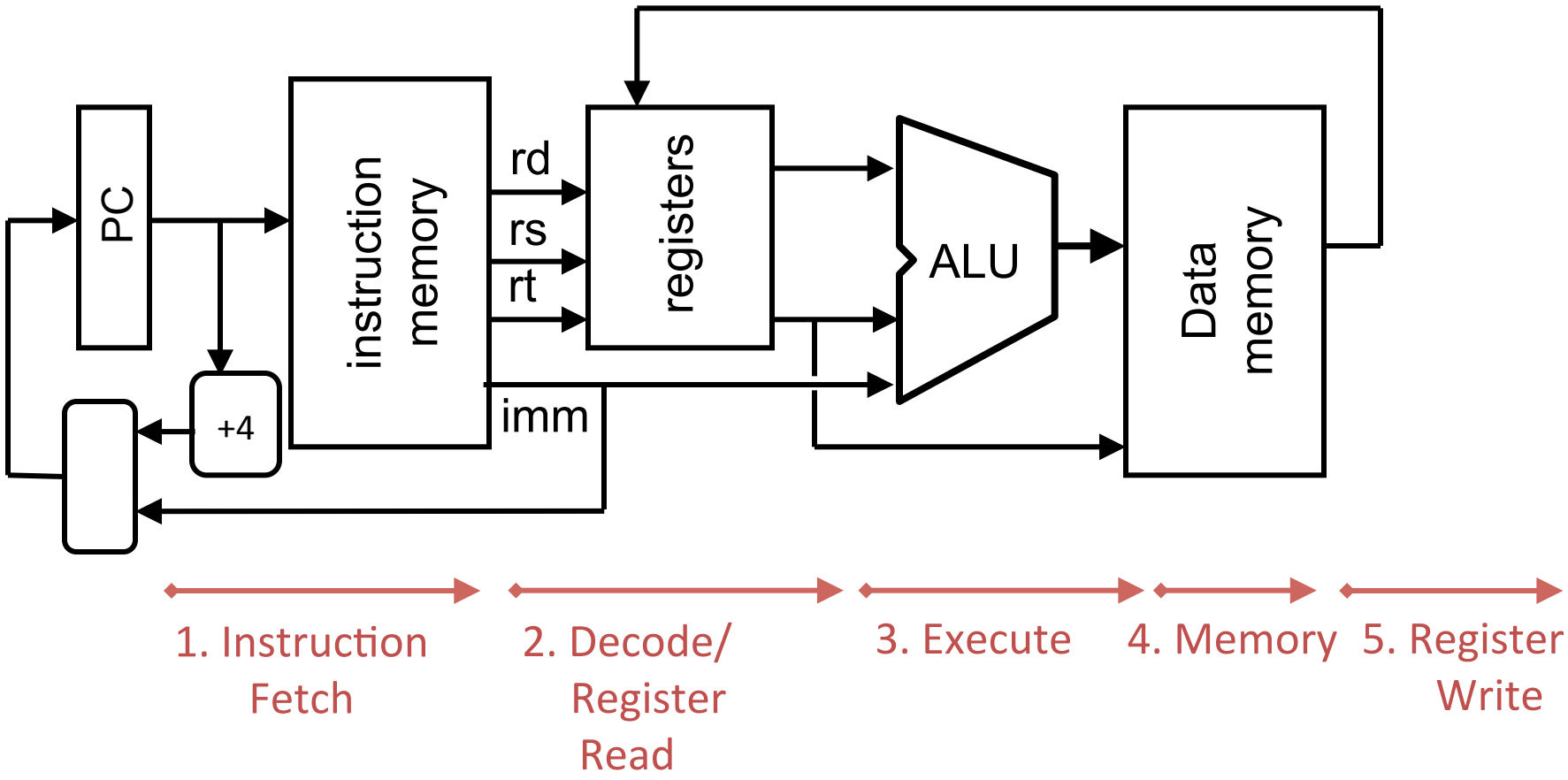
Stages of Execution on Datapath



Stages of Execution (3/5)

- Stage 3: ALU (Arithmetic-Logic Unit)
  - the real work of most instructions is done here: arithmetic (+, -, *, /), shifting, logic (&, |), comparisons (slt)
  - what about loads and stores?
    - lw $t0, 40($t1)
    - the address we are accessing in memory = the value in $t1 PLUS the value 40
    - so we do this addition in this stage

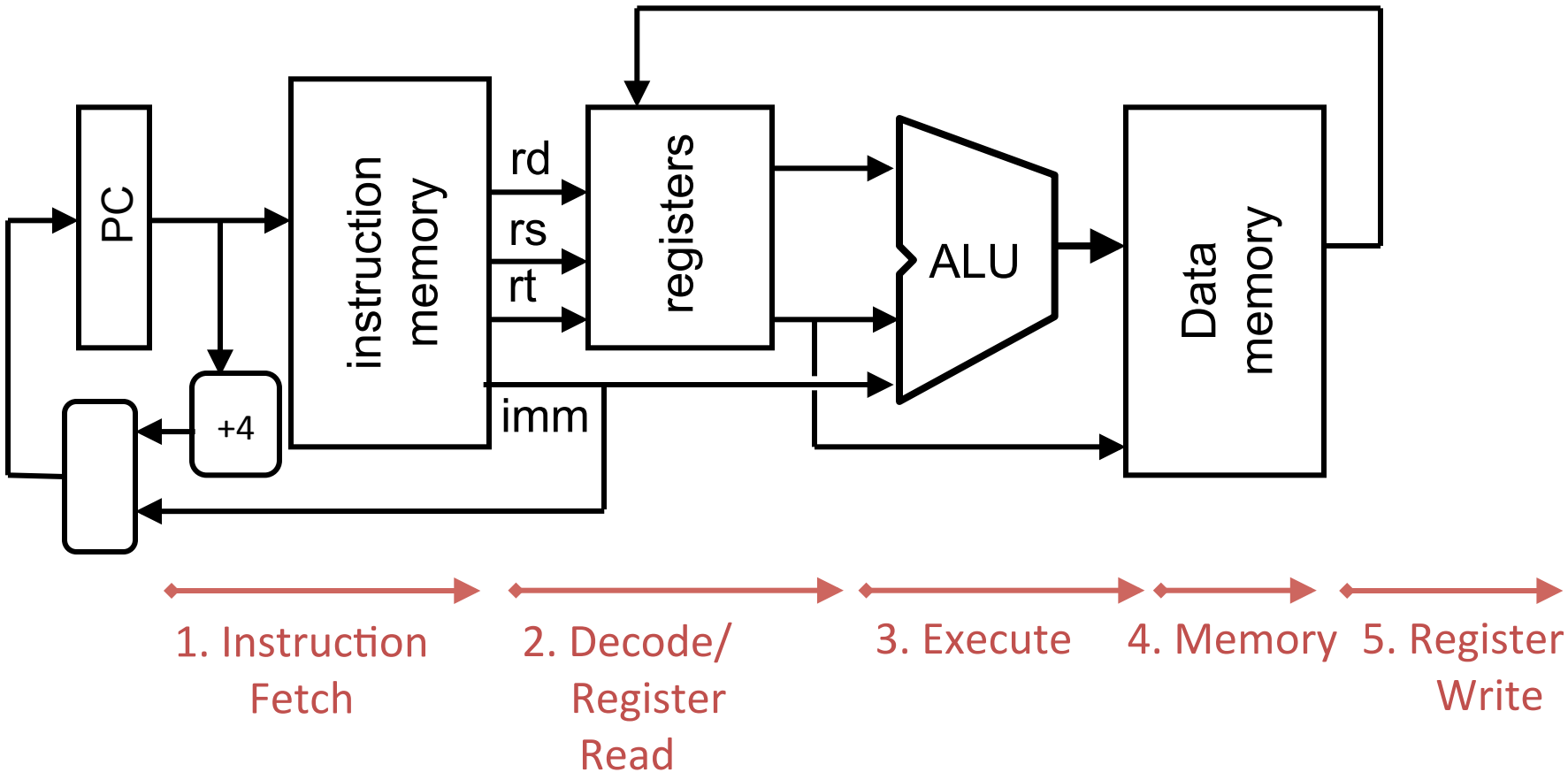
Stages of Execution on Datapath



Stages of Execution (4/5)

- Stage 4: Memory Access
  - actually only the load and store instructions do anything during this stage; the others remain idle during this stage or skip it all together
  - since these instructions have a unique step, we need this extra stage to account for them
  - as a result of the cache system, this stage is expected to be fast

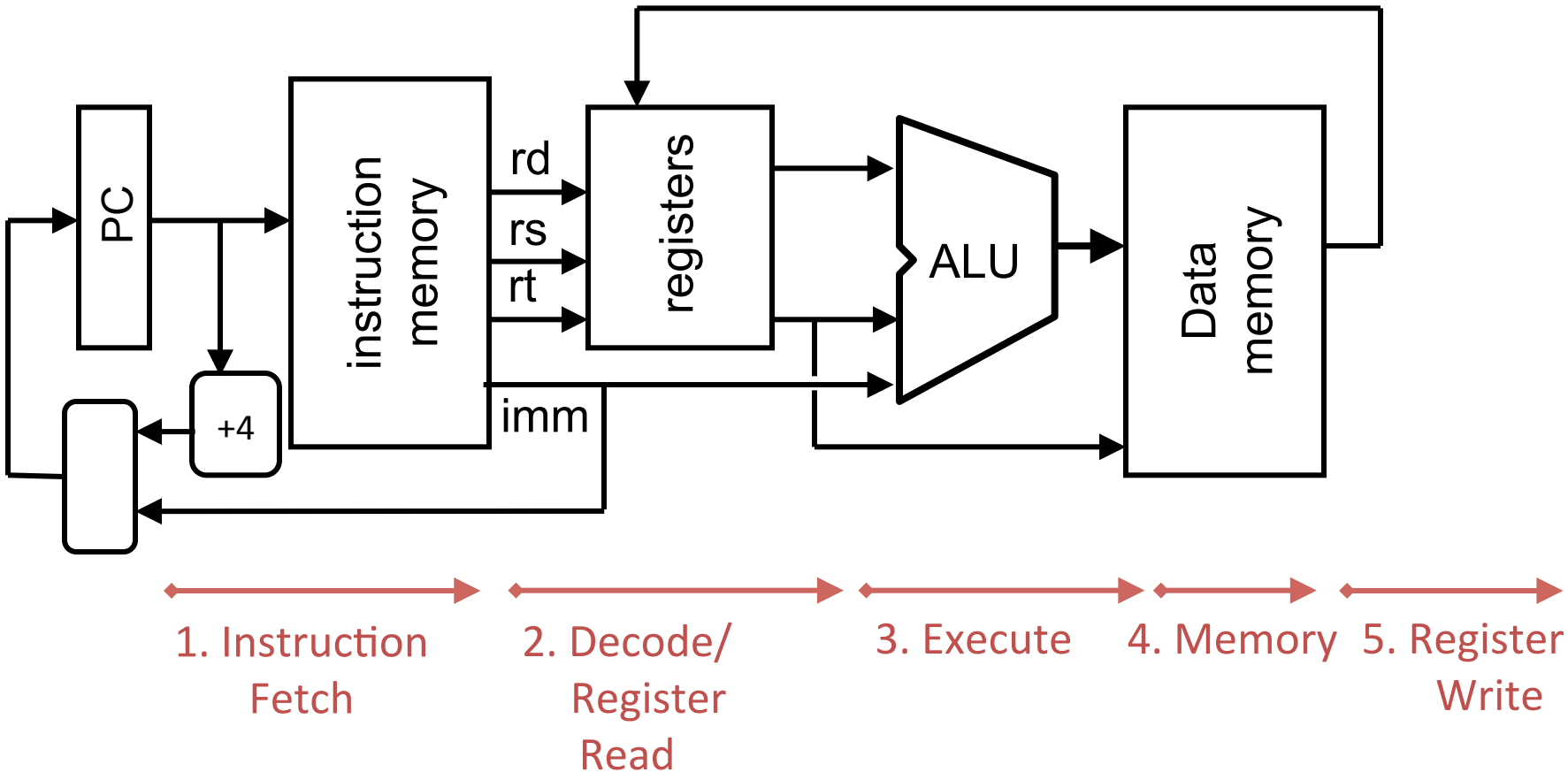
Stages of Execution on Datapath



Stages of Execution (5/5)

- Stage 5: Register Write
  - most instructions write the result of some computation into a register
  - examples: arithmetic, logical, shifts, loads, slt
  - what about stores, branches, jumps?
    - don't write anything into a register at the end
    - these remain idle during this fifth stage or skip it all together

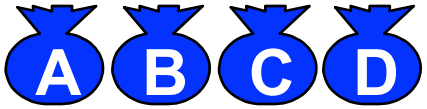
Stages of Execution on Datapath

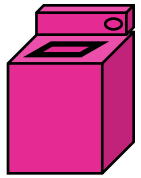


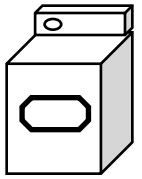
Pipelining and CISC

Gotta Do Laundry

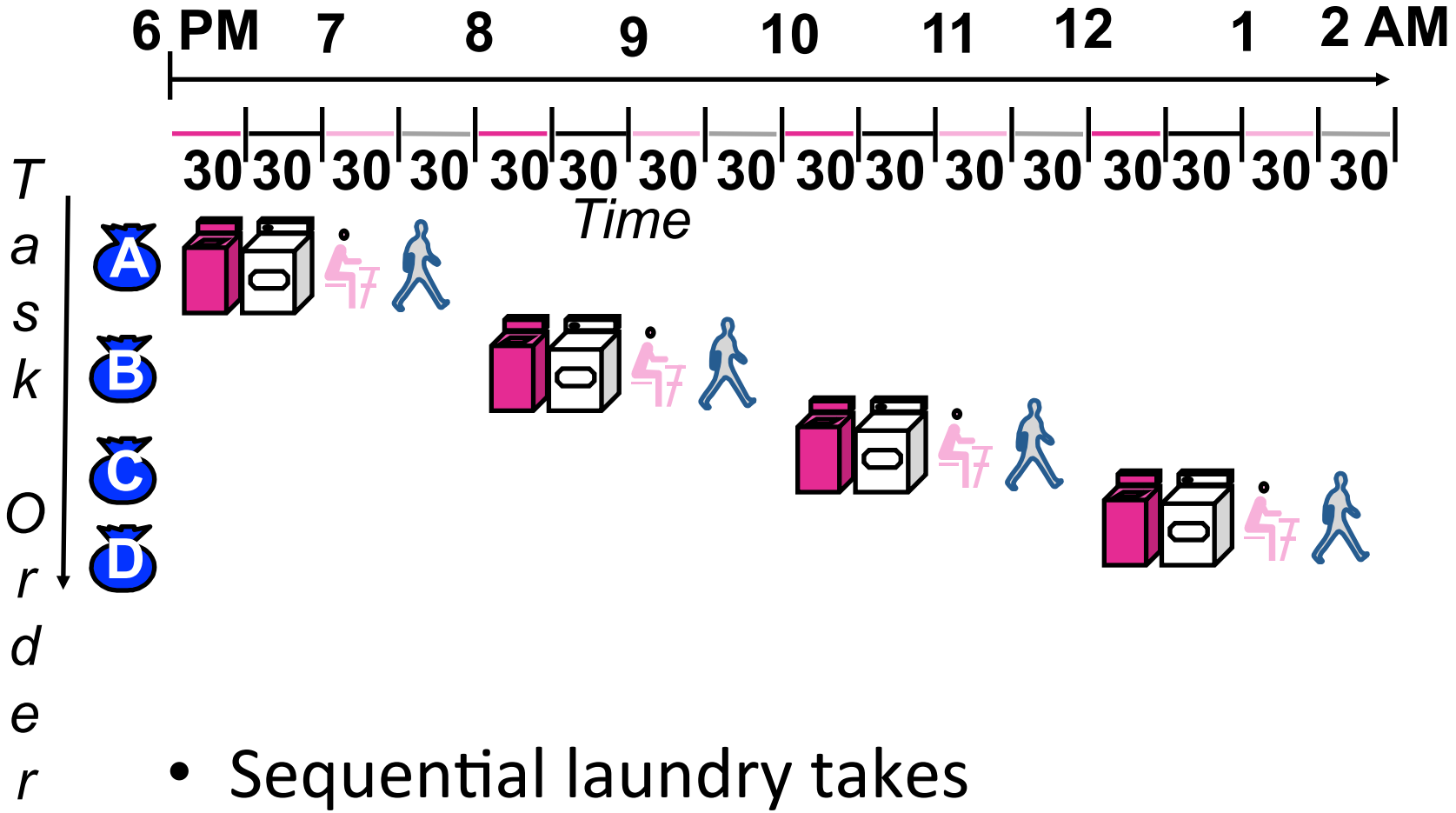
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - "Folder" takes 30 minutes
  - "Stasher" takes 30 minutes to put clothes into drawers





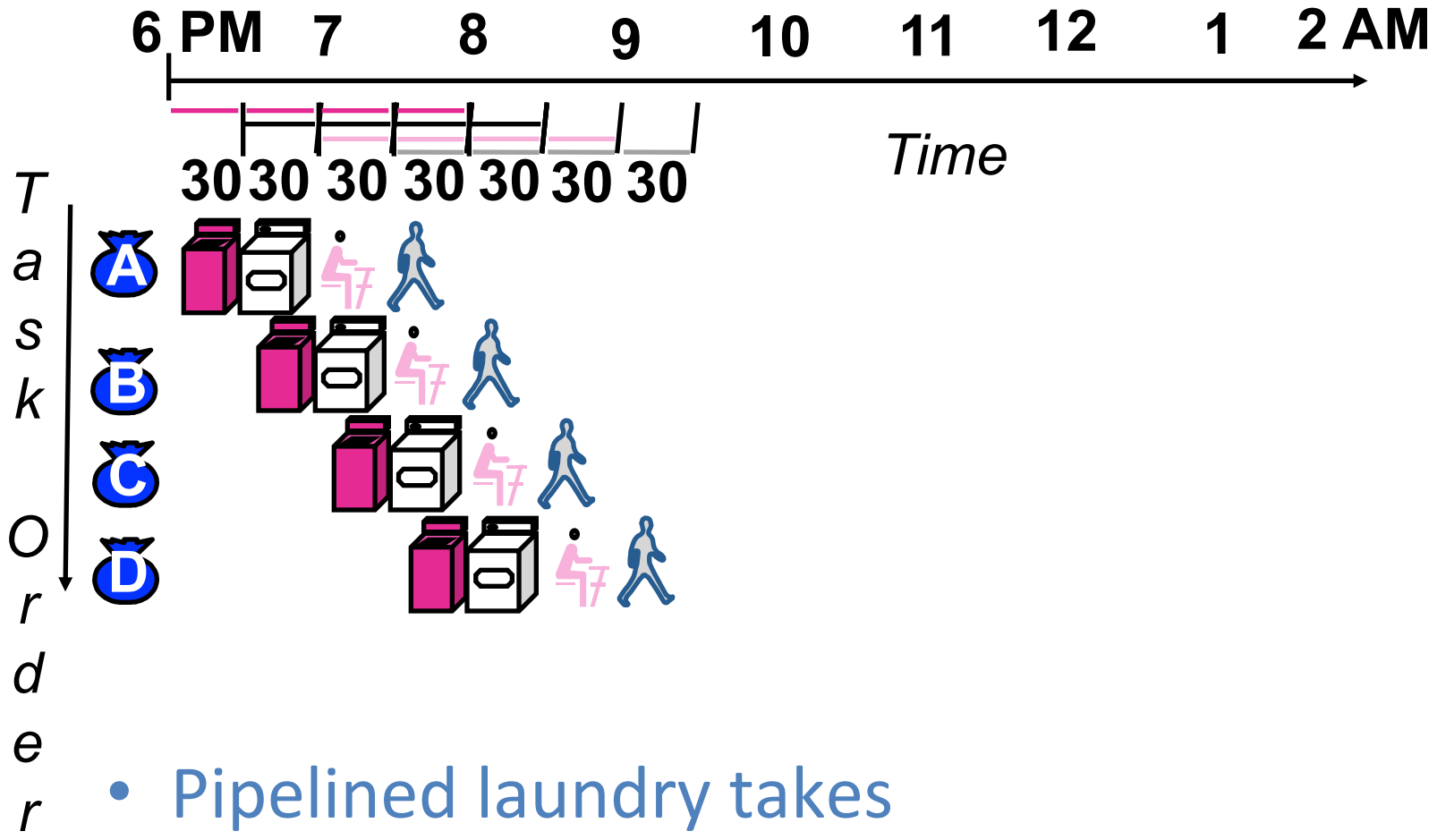


Sequential Laundry



- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry



- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) Structural hazard

- A required resource is busy (e.g. needed in multiple stages)

2) Data hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

3) Control hazard

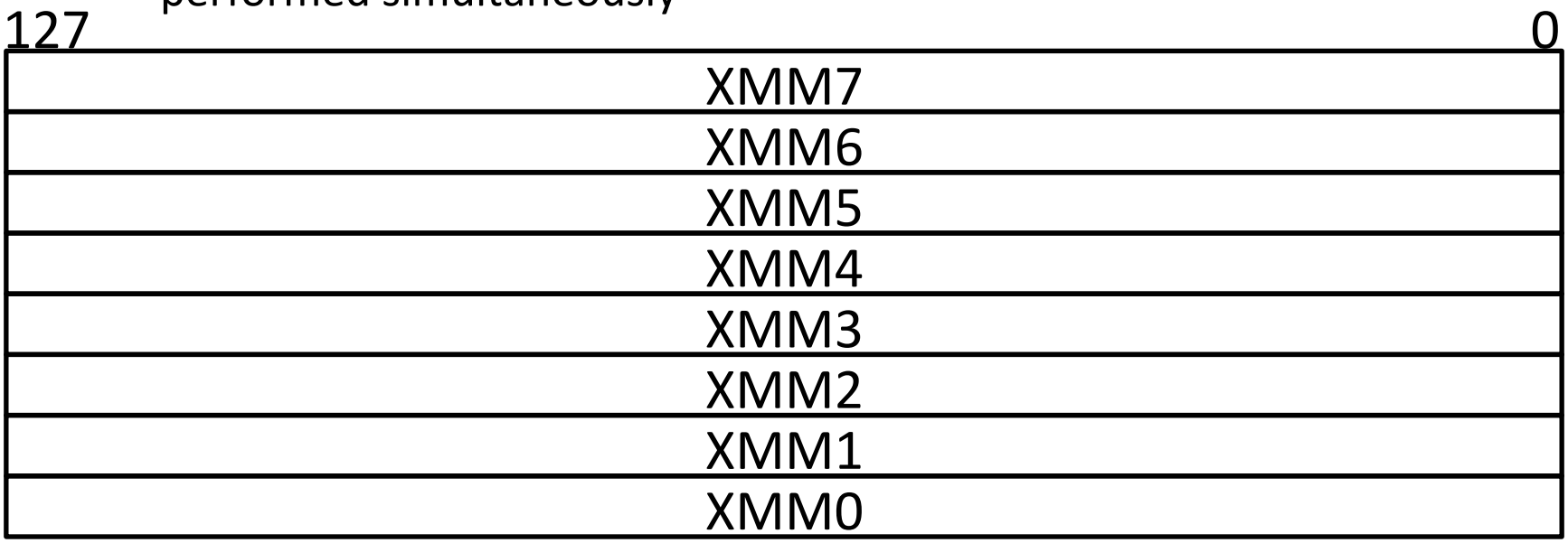
- Flow of execution depends on previous instruction

Great Idea #5: Performance Measurement and Improvement

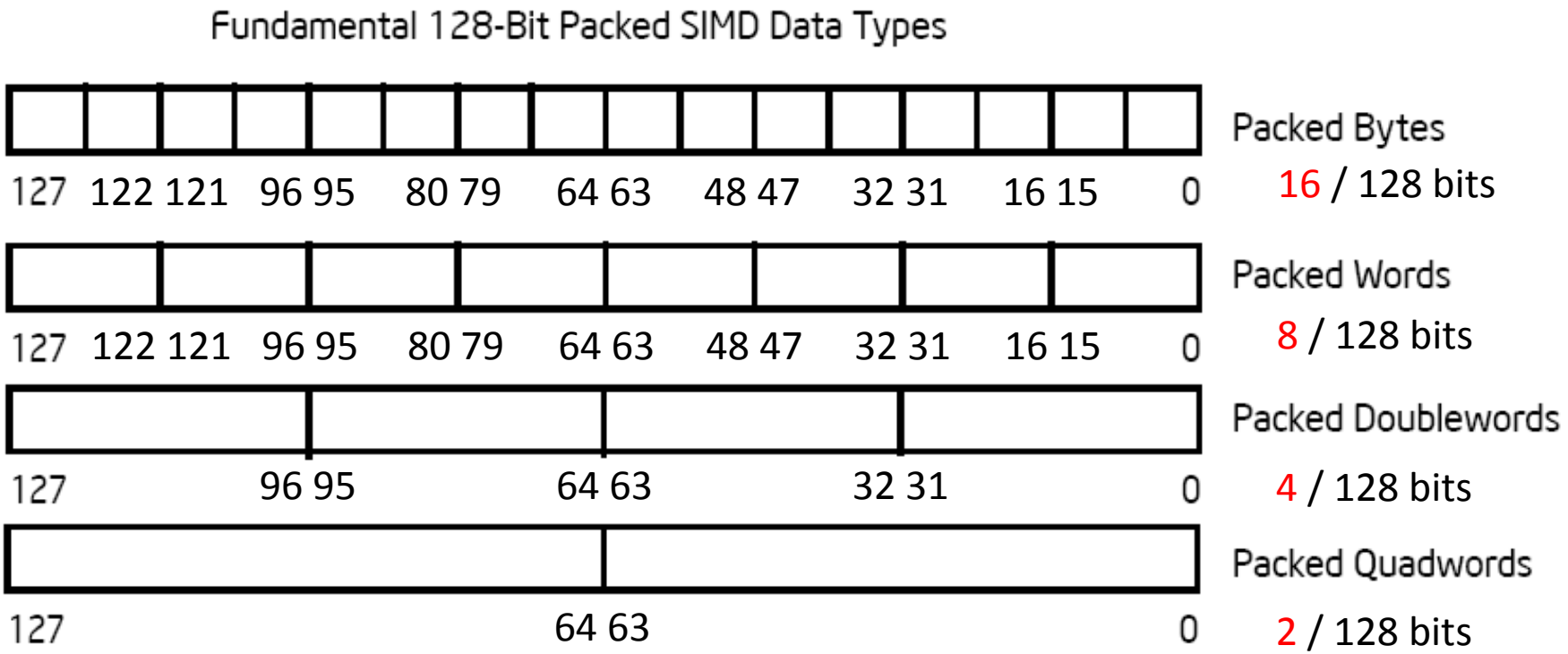
- Matching application to underlying hardware to exploit:
  - Locality
  - Parallelism
  - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency
  - How long to set the problem up
  - How much faster does it execute once it gets going
  - It is all about *time to finish*

XMM Registers

- Architecture extended with eight 128-bit data registers
  - 64-bit address architecture: available as 16 64-bit registers (XMM8 – XMM15)
  - e.g. 128-bit packed single-precision floating-point data type (doublewords), allows four single-precision operations to be performed simultaneously



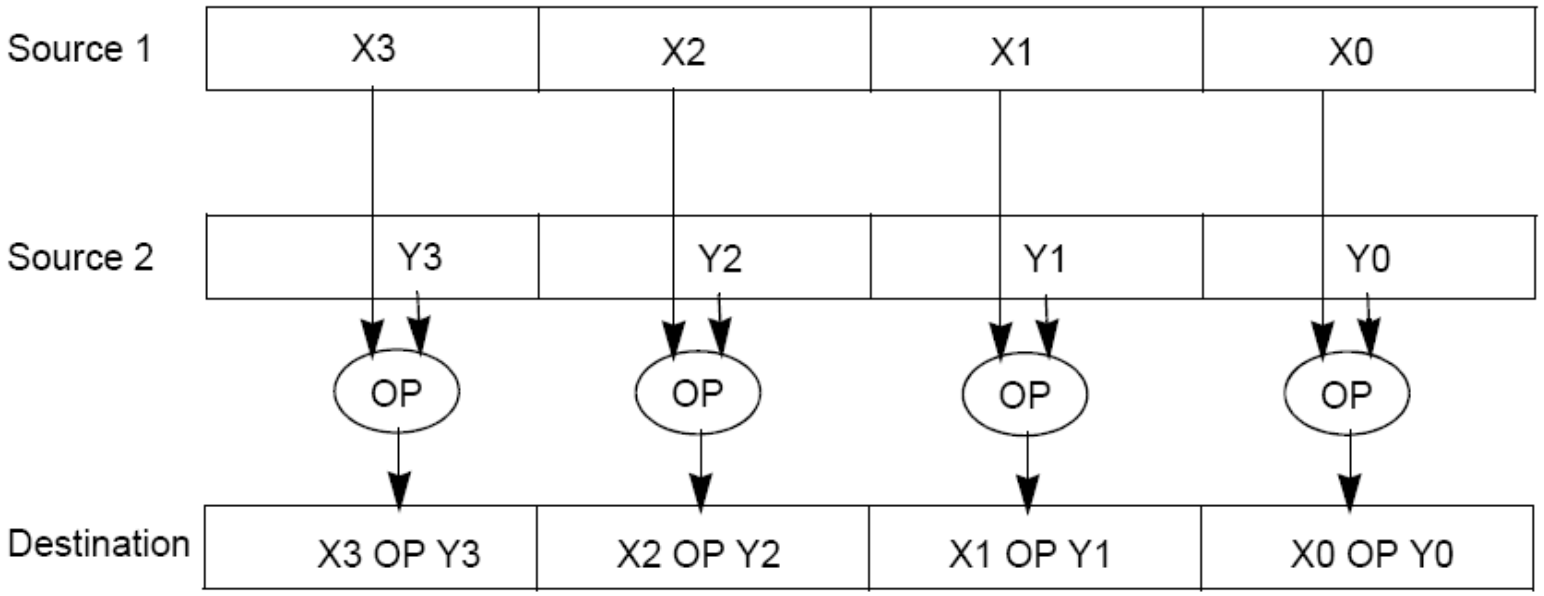
Intel Architecture SSE2+ 128-Bit SIMD Data Types



- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
  - Single precision FP: Double word (32 bits)
  - Double precision FP: Quad word (64 bits)

"Advanced Digital Media Boost"

- To improve performance, Intel's SIMD instructions
  - Fetch one instruction, do the work of multiple instructions
  - MMX (MultiMedia eXtension, Pentium II processor family)
  - *SSE (Streaming SIMD Extension, Pentium III and beyond)*



SSE Instruction Categories for Multimedia Support

Instruction category	Operands
Unsigned add/subtract	Eight 8-bit or Four 16-bit
Saturating add/subtract	Eight 8-bit or Four 16-bit
Max/min/minimum	Eight 8-bit or Four 16-bit
Average	Eight 8-bit or Four 16-bit
Shift right/left	Eight 8-bit or Four 16-bit

- Intel processors are CISC (complicated instrs)
- SSE-2+ supports wider data types to allow 16 × 8-bit and 8 × 16-bit operands

Loop Unrolling in C

- Instead of compiler doing loop unrolling, could do it yourself in C:

```
for(i=0; i<1000; i++)
  x[i] = x[i] + s;
```


```
for(i=0; i<1000; i=i+4) {
  x[i]   = x[i]   + s;
  x[i+1] = x[i+1] + s;
  x[i+2] = x[i+2] + s;
  x[i+3] = x[i+3] + s;
}
```

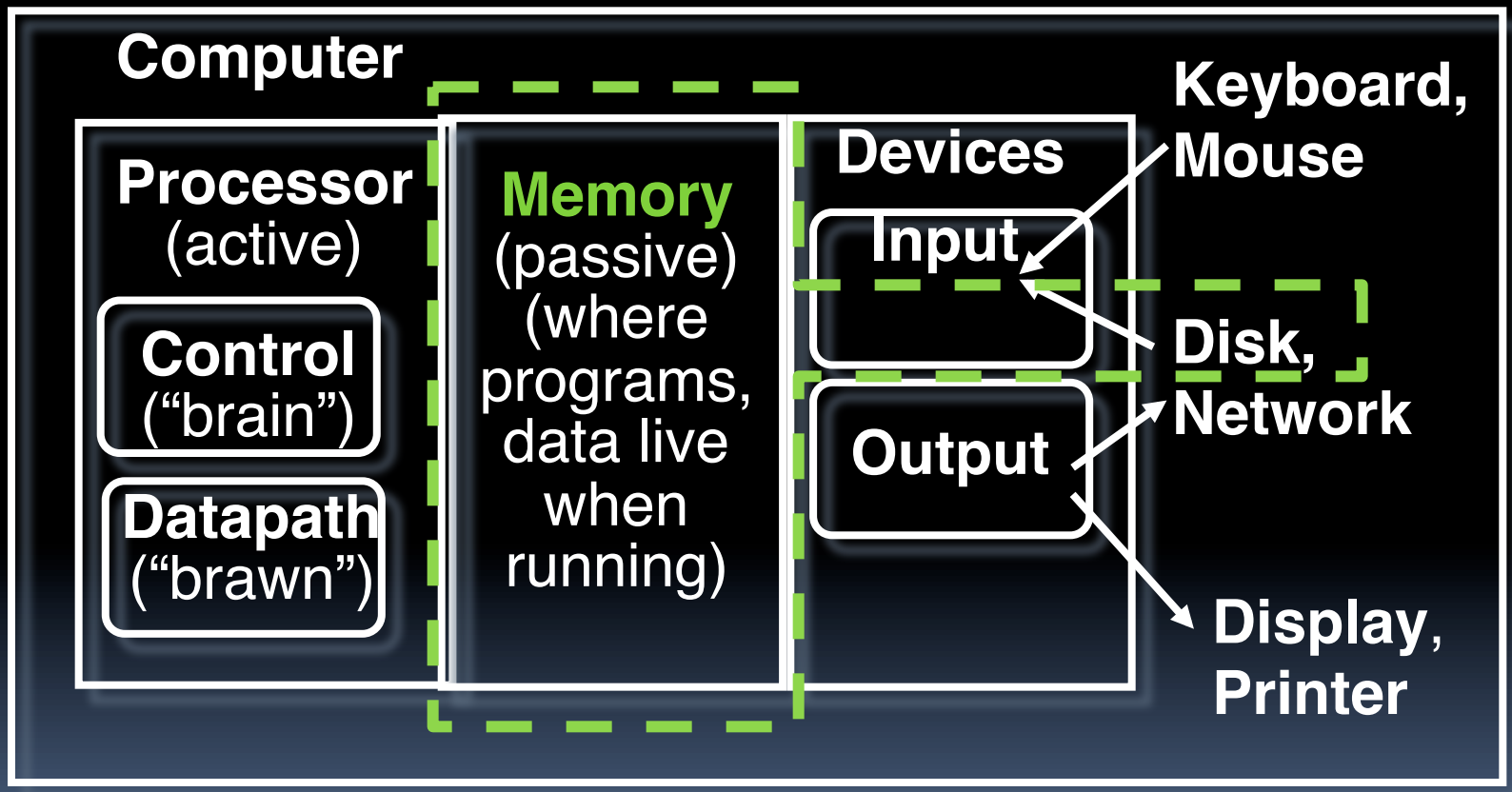
What is downside of doing this in C?

Caching

6 Great Ideas in Computer Architecture

1. Layers of Representation/Interpretation
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy

The Big Picture

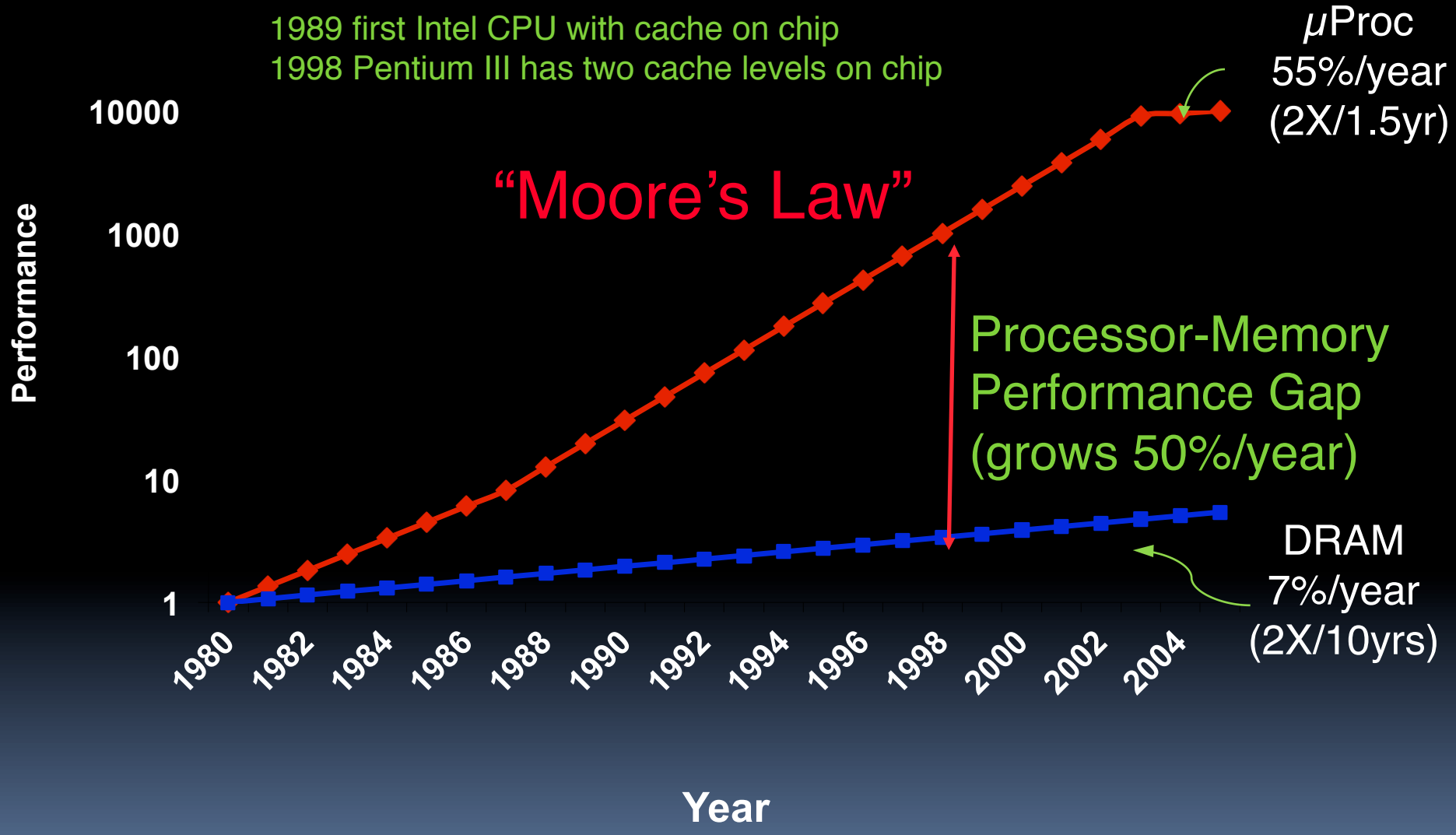


Memory Hierarchy

I.e., storage in computer systems

- Processor
  - holds data in register file (~100 Bytes)
  - Registers accessed on nanosecond timescale
- Memory (we'll call "main memory")
  - More capacity than registers (~Gbytes)
  - Access time ~50-100 ns
  - Hundreds of clock cycles per memory access?!
- Disk
  - HUGE capacity (virtually limitless)
  - VERY slow: runs ~milliseconds

Motivation : Processor-Memory Gap



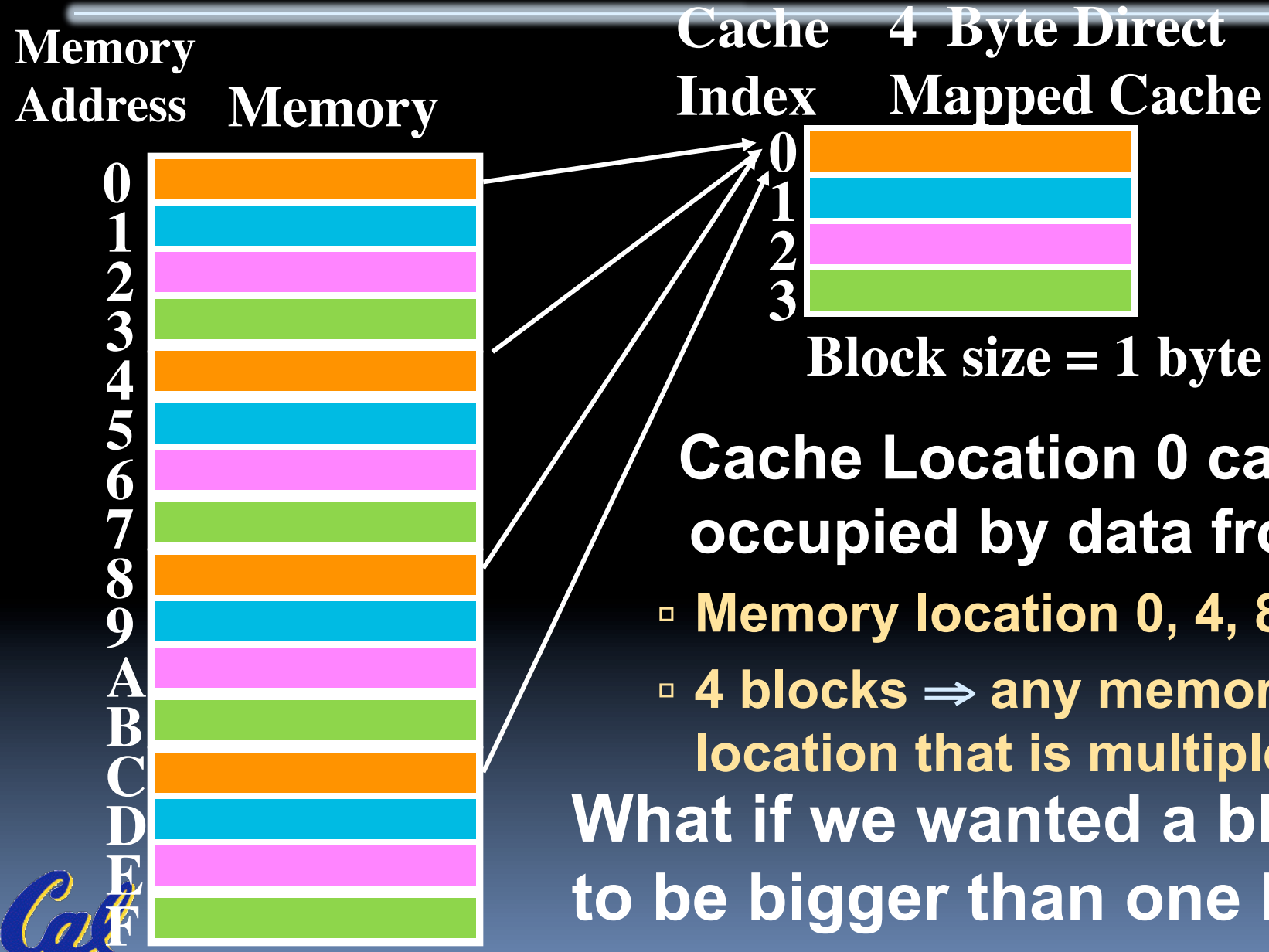
Memory Caching

- Mismatch between processor and memory speeds leads us to add a new level: a memory cache
- Implemented with same IC processing technology as the CPU (usually integrated on same chip): faster but more expensive than DRAM memory.
- Cache is a copy of a subset of main memory.
- Most processors have separate caches for instructions and data.

Direct-Mapped Cache (1/4)

- In a direct-mapped cache, each memory address is associated with one possible block within the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - Block is the unit of transfer between cache and memory

Direct-Mapped Cache (2/4)



Cache Location 0 can be occupied by data from:

- Memory location 0, 4, 8, ...
- 4 blocks ⇒ any memory location that is multiple of 4

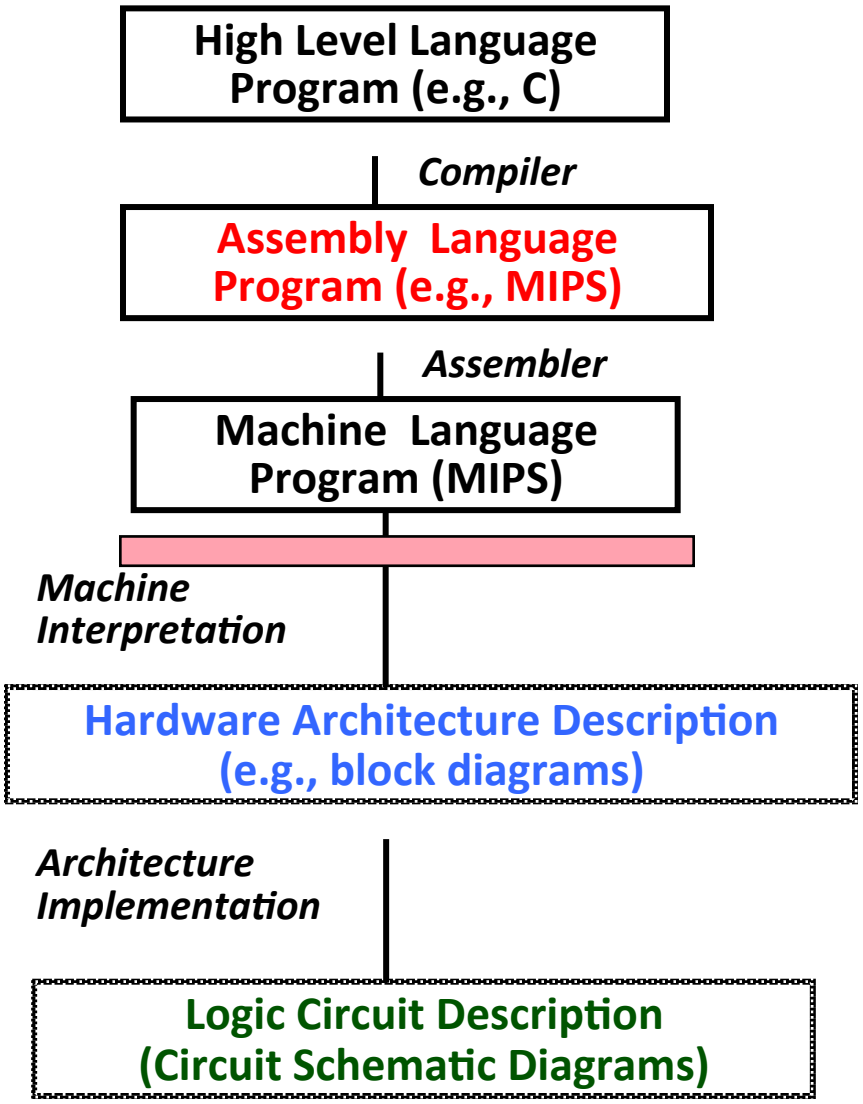
What if we wanted a block to be bigger than one byte?

Understanding Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st reference):
  - First access to block impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- *Conflict (collision):*
  - *Multiple memory locations mapped to the same cache location*
  - *Solution 1: increase cache size*
  - *Solution 2: increase associativity (may increase access time)*

Summary and Conclusion

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

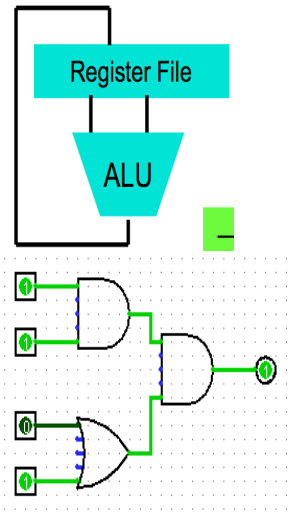



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

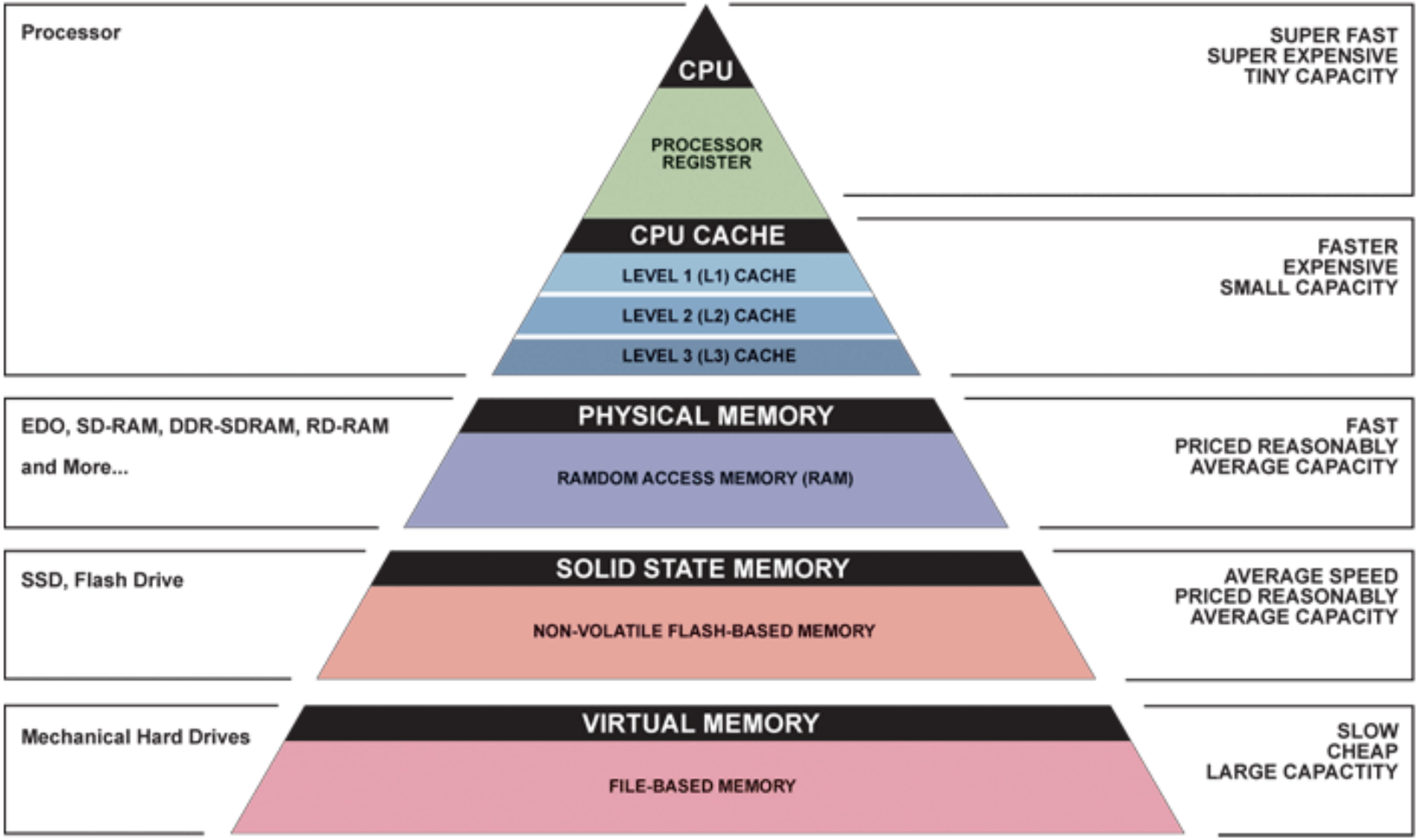
```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



Great Idea #3: Principle of Locality/ Memory Hierarchy



Great Idea #5: Performance Measurement and Improvement

- Matching application to underlying hardware to exploit:
  - Locality
  - Parallelism
  - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency
  - How long to set the problem up
  - How much faster does it execute once it gets going
  - It is all about *time to finish*

Questions?