UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 28: Course Review

Taylor Johnson

Announcements and Outline

- Student Feedback Survey (SFS)
  - Invitation by email *sent on Wednesday, November 19.*
  - *Also accessible via **Blackboard***
  - *MUST complete BEFORE Wednesday, December 3, 2014, 11pm*
  - ***PLEASE*** complete, ***very*** important for the university and your future classes
  - ***Note:*** university average and median ratings are ***~4.25+ out of 5.0***

University-wide statistics for all courses included in the survey for Spring 2013	*Mean*	**4.30**	**4.12**	**4.26**	**4.38**	**4.31**	**4.18**
	N	31,318	31,246	31,153	31,088	29,870	30,998
The instructor for this course:		… provided clearly defined expectations.	… used teaching methods to help me learn.	… encouraged me to take part in my own learning.	… was well prepared for each class meeting.	… was available outside of class.	…is one I would recommend to other students.

- Programming assignment 4 due 12/3 by midnight
- Course Review

Final Exam Details

- December 9, 2-4:30pm
- Closed book, no calculator
- Cheat sheet: one sheet of paper no larger than letter size (8.5”x11”), both sides
- Comprehensive: also review chapters tested in midterm
  - Midterm review slides: http://www.taylortjohnson.com/class/cse2312/f14/slides/cse2312_2014-10-07.pdf
- Slightly more focus on programming and the 2nd half of course material (e.g. sections we covered in chapters 3, 4, and 5)
- Practice Final Online later this week, will email when ready and also provide practice problems on using gdb, caches, floating point, etc.

Floating Point

IEEE 754 Floating-Point Format

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

IEEE 754 Example

- $n = sign * 2^e * f$
- 9 = b1.001 * 2^3 = 1.125 * 2^3 = 1.125 * 8 = 9
- Multiply by 2^3 is shift right by 3

Sign	Exponent	Fraction
0	1000 0010	00100000000000000000000

- e = exponent – 127 (biasing)
- f = 1.fraction

IEEE 754 Example

- $n = sign * 2^e * f$
- 5/4 = 1.25 = (-1)^0 * 2^0 * 1.25 = b1.01 = 1 + 1^-2

Sign	Exponent	Fraction
0	0111 1111	01000000000000000000000
+	127-127=0	1.25

- e = exponent – 127 (biasing)
- f = 1.fraction

IEEE 754 Example

- $n = sign * 2^e * f$
- -0.15625 = -5/32 = -1*b1.01 * 2^-3 = b0.00101
- Multiply by 2^-3 is shift left by 3

Sign	Exponent	Fraction
1	0111 1100	01000000000000000000000
-	124-127=-3	1.25

- e = exponent – 127 (biasing)
- f = 1.fraction
- -5/32 = -0.15625 = -1.25 / 2^3 = -1.25 / 8 = -5/(4*8)

ARM Floating Point

- Instructions prefixed with v, suffixed with, e.g., .f32
- Registers are s0 through s31 and d0 through d15

```
foperandA: .float 3.14
foperandB: .float 2.5
vldr.f32  s1, foperandA    @ s1 =
mem[foperandA]
vldr.f32  s1, foperandB    @ s2 =
mem[foperandB]
vadd.f32  s0, s1, s2
```

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - Exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Example

- Represent –0.75 in floating point (IEEE 754)
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - b1.1 = d1.5, and note 1.5 * ½ = 0.75
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + ***Bias***
    - Single: –1 + ***127*** = 126 = 01111110_2
    - Double: –1 + ***1023*** = 1022 = 01111111110_2
- Single: 10111111001000…00
- Double: 1011111111101000…00

$$n = sign * f * 2^e$$

Floating-Point Example

- What number is represented by the single-precision float

  1100000010100000000000000

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$

$$n = sign * f * 2^e$$

- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - \mathbf{127})}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round (***4 digits!***) and renormalize if necessary
  - 1.002×10^{2}

Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (i.e., 0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round (4 digits!) and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*
  - Cost hundreds of millions of dollars

Floating-Point Summary

- Floating-point
  - Decimal point moves due to exponents (bit shifting)
  - Positive / negative zeros
- Fixed-point
  - Decimal point remains at fixed point (e.g., after bit 8)
- Spacing between these numbers and real numbers

Combining C and Assembly and Compiler Optimizations

Compiling C

- How did we go from ASM to machine language?
  - Two-pass assembler
- How do we go from C to machine language?
  - Compilation
  - Can think of as generating ASM code, then assembling it (use –S option)
- Complication: optimizations
  - Any time you see the word “optimization” ask yourself, according to what metric?
    - Program Speed
    - Code Size
    - Energy
    - …

GCC Optimization Levels

- `-O`: Same as `-O1`
- `-O0`: do no optimization, the default if no optimization level is specified
- `-O1`: optimize
- `-O2`:optimise even more
- `-O3`: optimize the most
- `-Os`: Optimize for size (memory constrained devices)

Assembly Calls of C Functions

```
.globl _start

_start:
    mov sp, #0x12000        @ set up stack
    bl  c_function_0
    bl  c_function_1
    bl  c_function_2
    bl  c_function_3

iloop: b iloop
```

Most Basic Example

```
int c_function_0() {
    return 1;
}
```

Call via:

```
bl c_function_0
```

What assembly instructions make up `c_function_0`?

c_function_0 (with –O0)

```
10014: e52db004   push    {fp}
10018: e28db000   add     fp, sp, #0; fp = sp
1001c: e3a03005   mov     r3, #1
10020: e1a00003   mov     r0, r3
10024: e28bd000   add     sp, fp, #0 ; sp = fp
10028: e8bd0800   pop     {fp}
1002c: e12fff1e   bx      lr
```

c_function_0 (with –O1)

```
10014: e3a00001  mov     r0, #1
10018: e12fff1e  bx      lr
```

One Argument Example

```
int c_function_1(int x) {
    return 4*x;
}
```

Call via:

```
bl c_function_1
```

What assembly instructions make up `c_function_1`?

c_function_1 (with –O0)

```
10030: e52db004   push   {fp}
10034: e28db000   add fp, sp, #0 ; fp = sp
10038: e24dd00c   sub sp, sp, #12
1003c: e50b0008   str r0, [fp, #-8]
10040: e51b3008   ldr r3, [fp, #-8]
10044: e1a03103   lsl r3, r3, #2
10048: e1a00003   mov r0, r3
1004c: e28bd000   add sp, fp, #0 ; sp = fp
10050: e8bd0800   pop {fp}
10054: e12fff1e   bx  lr
```

c_function_1 (with –O1)

```
1001c: e1a00100  lsl    r0, r0, #2
10020: e12fff1e  bx     lr
```

lsl: logical shift left

Shift left by 2 == multiply by 4

One Argument Example with Conditional

```
int c_function_2(int x) {
    if (x <= 0) {
        return 1;
    }
    else {
        return x;
    }
}
```

c_function_2 (with –O0)

```
1005c: e52db004     push   {fp}          ; (str fp, [sp, #-4]!)
10060: e28db000     add    fp, sp, #0
10064: e24dd00c     sub    sp, sp, #12
10068: e50b0008     str    r0, [fp, #-8]
1006c: e51b3008     ldr    r3, [fp, #-8]
10070: e3530000     cmp    r3, #0
10074: ca000001     bgt    10080 <c_function_2+0x24>
10078: e3a03001     mov    r3, #1
1007c: ea000000     b      10084 <c_function_2+0x28>
10080: e51b3008     ldr    r3, [fp, #-8]
10084: e1a00003     mov    r0, r3
10088: e28bd000     add    sp, fp, #0
1008c: e8bd0800     pop    {fp}
10090: e12fff1e     bx     lr
```

c_function_2 (with –O2)

```
10028: e3500001  cmp r0, #1
1002c: b3a00001  movlt  r0, #1
10030: e12fff1e  bx lr
```

Loop Example

```
int c_function_3(int x) {
    int c;
    int f = x;

    for (c = x - 1; c > 0; c--) {
        f *= c;
    }
    return f;
}
```

c_function_3 (with –O1)

```
10034: e2403001  sub    r3, r0, #1
10038: e3530000  cmp    r3, #0
1003c: d12fff1e  bxle   lr
10040: e0000093  mul    r0, r3, r0
10044: e2533001  subs   r3, r3, #1
10048: 1afffffc  bne    10040
<c_function_3+0xc>
1004c: e12fff1e  bx lr
```

Compiler Optimization Summary

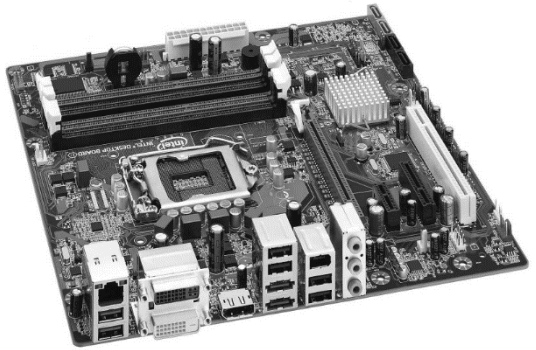
- First point: stack frames (frame pointer register, fp)
- Second point: often times it's safe to avoid using push/pop and the stack
- Easier when we manually ASM write code to just go ahead and use it (for safety and avoiding bugs), but the compiler as we've seen (when using optimization levels 1 and 2) will try to avoid the stack if it's safe to do so
  - Why?

Final Exam Review

Course Objective Overview

- Seen how computers really ***compute***
- Processor/memory organization: execution cycle, registers, memory accesses, caches
- Processor operation: pipeline, fetch-decode-execute
- Computer organization: memory, buses, I/O devices
- Assembly language programming: various architecture styles (CISC vs. RISC), register-to-register (ARM), etc.
- Saw more representations of information (machine language instructions, floating point, integers, signed vs. unsigned, Endianness, ASCII, Unicode, …)

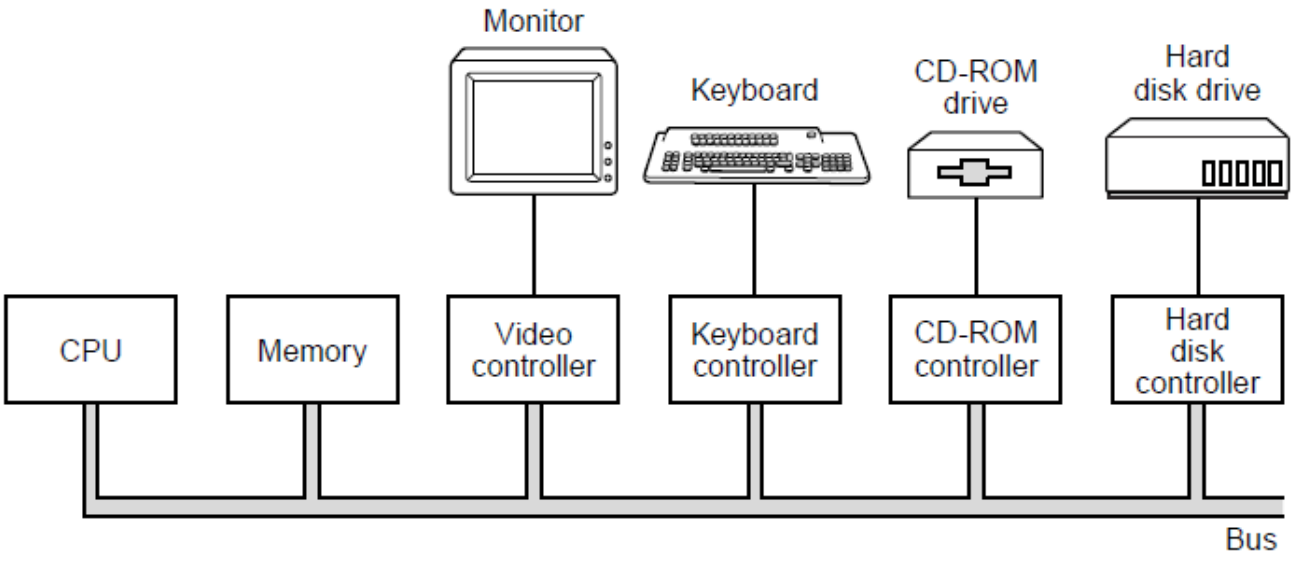
Digital Computers



- How do computers ***compute***?
- Machine for carrying out instructions
  - Program = sequence of instructions
- Instructions
  - Add numbers
  - Check if a number is zero
  - Copy data between different memory locations (addresses)
  - Represented as machine language (binary numbers of a certain length)
    - Example: $\overbrace{00}^{opcode}\ \overbrace{10}^{dest}\ \overbrace{01}^{src0}\ \overbrace{00}^{src1}$ on an 8-bit computer may mean:
      - Take numbers in registers 0 and 1 (special memory locations inside the processor) and add them together, putting their sum into register 2
      - That is, to this computer, 00100100 means $r2 = r1 + r0$
      - In assembly, this could be written: `add r2 r1 r0`
    - Question: for this same computer, what does 00000000 mean?
      - `add r0 r0 r0`, that is: $r0 = r0 + r0$

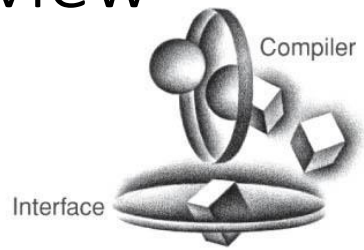
Computer System Overview

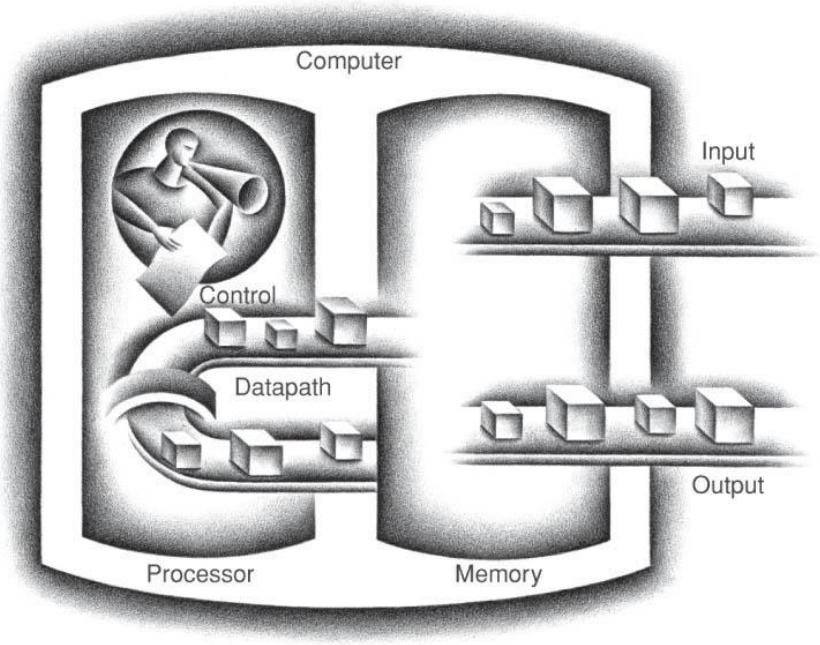
- CPU
  - Executes instructions
- Memory
  - Stores program and data
- Buses
  - Transfers data
- Storage
  - Permanent
- I/O devices
  - Input: keypad,
  - Output: printer, screen
  - Both (input and output), such as:
    - USB, network, Wifi, touch screen, hard drive



Computer Organization Overview

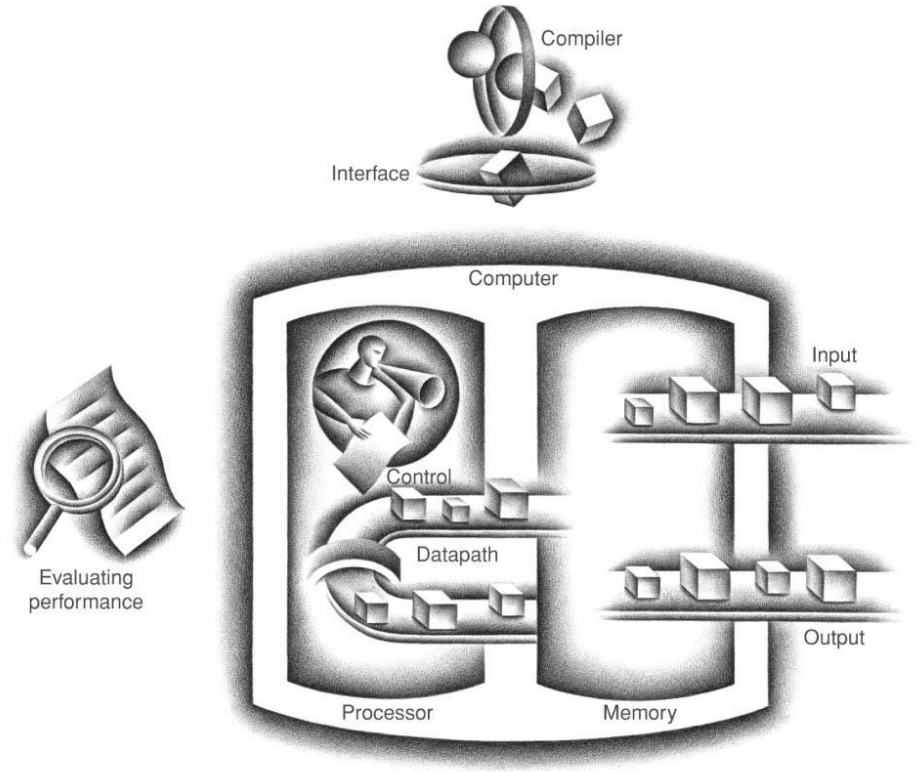
- ISA: hardware-software interface
- CPU
  - Executes instructions
- Memory
  - Stores programs and data
- Buses
  - Transfers data
- I/O devices
  - Input: keypad, mouse, touch, …
  - Output: printer, screen, …

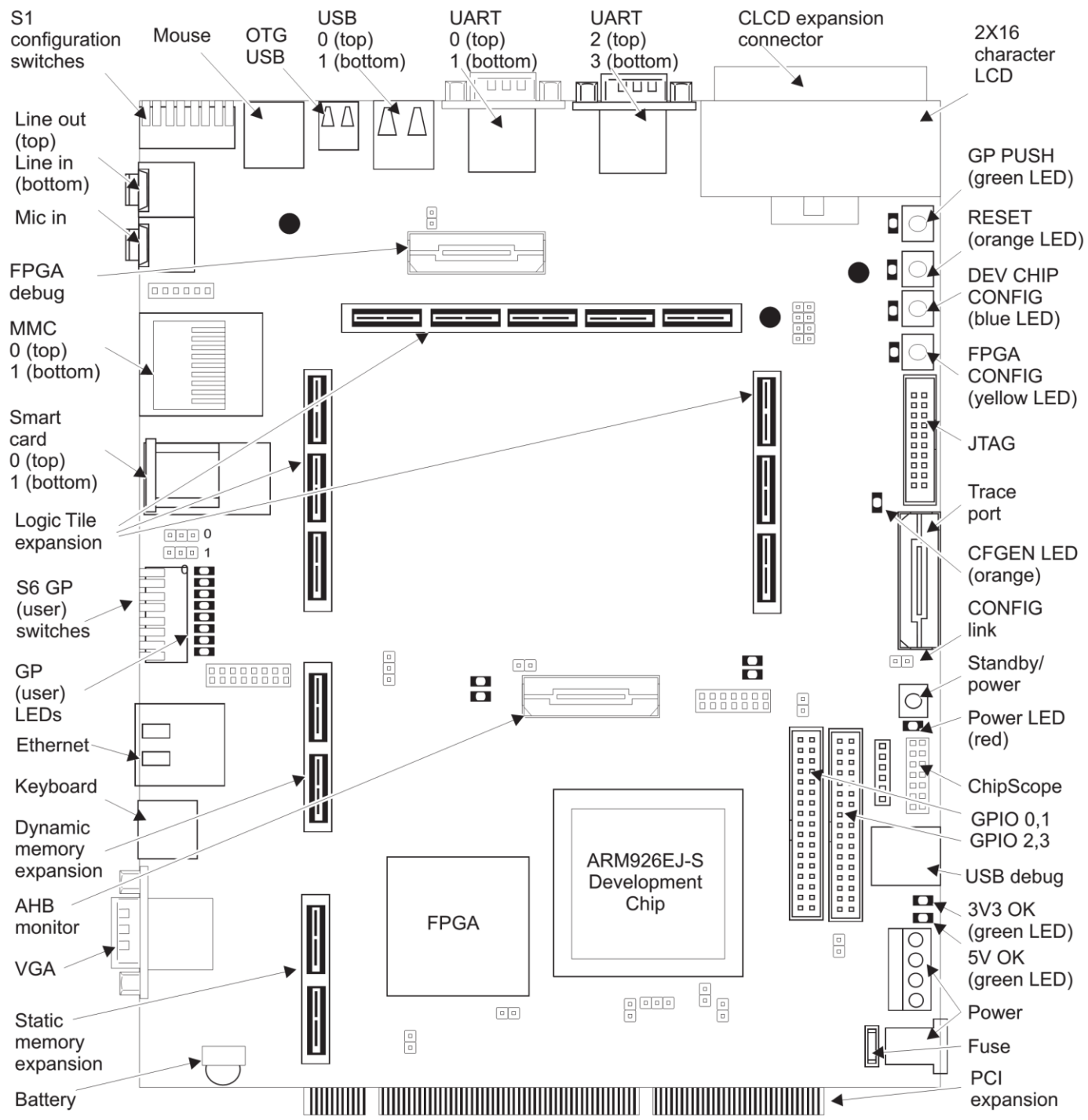




What Computer Have We Used this Semester?

- ARM Versatilepb computer
- Full computer!
  - Input
  - Output
  - Processor
  - Memory
  - Programs





This is a picture of the board for the ARM computer we've been using in QEMU!

[http://infocenter.arm.com/help/topic/com.arm.doc.dui0224i/DUI0224I_realview_platform_baseboard_for_arm926ej_s_ug.pdf]

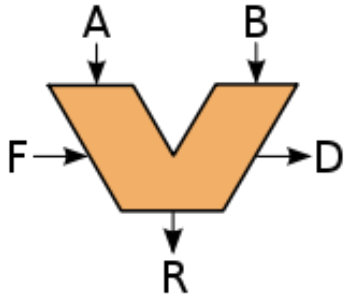
Review: Some Processor Components

CPU

Register File

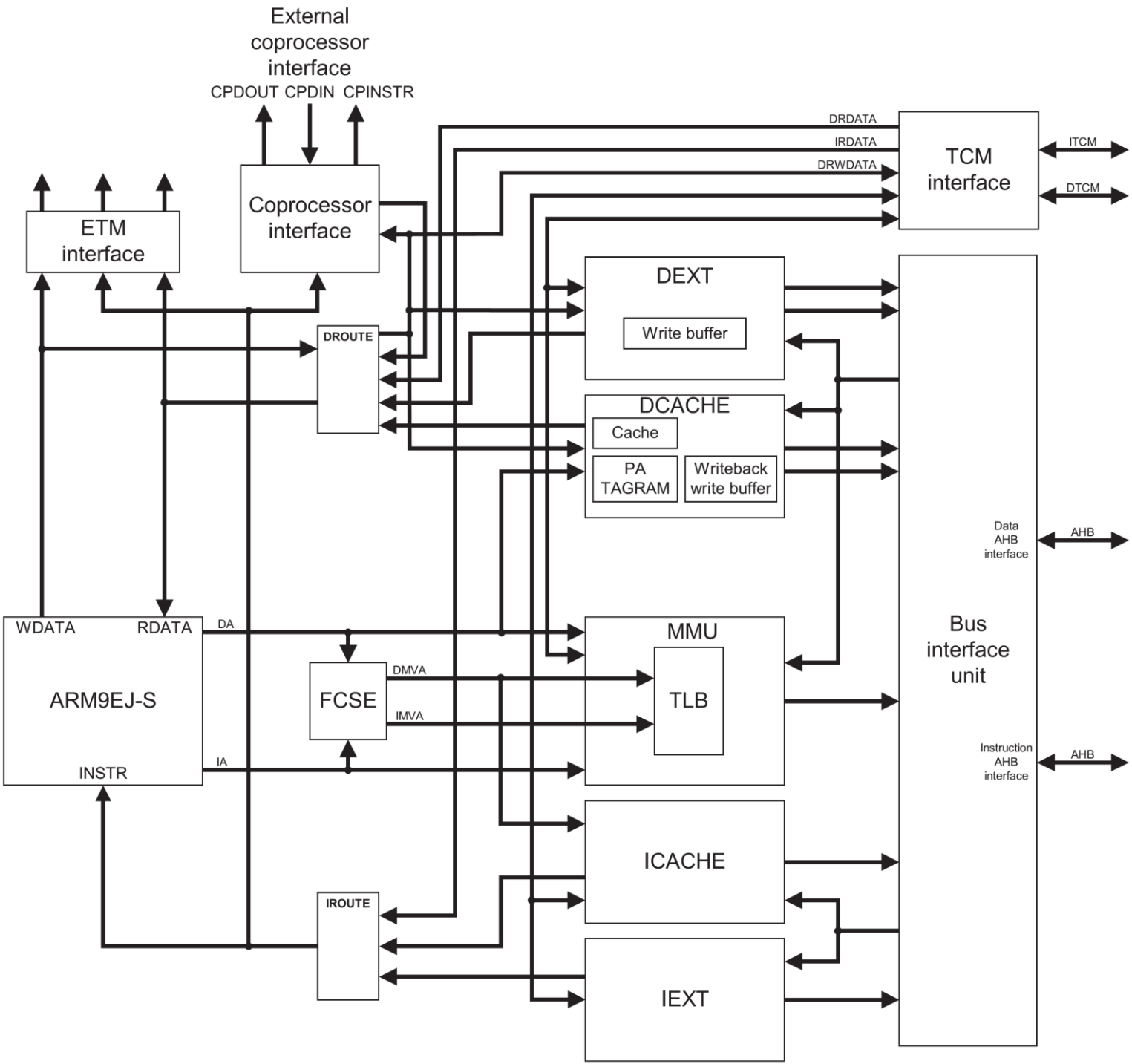
- Program Counter (PC)
- Instruction Register (IR)
- General Purpose Registers
  - Word size
  - Typically 16-32 of these
  - PC sometimes one of these
- Floating Point Registers

Arithmetic logic unit (ALU)



Other units (pipeline, MMU, caches, TLB, multicores, …)

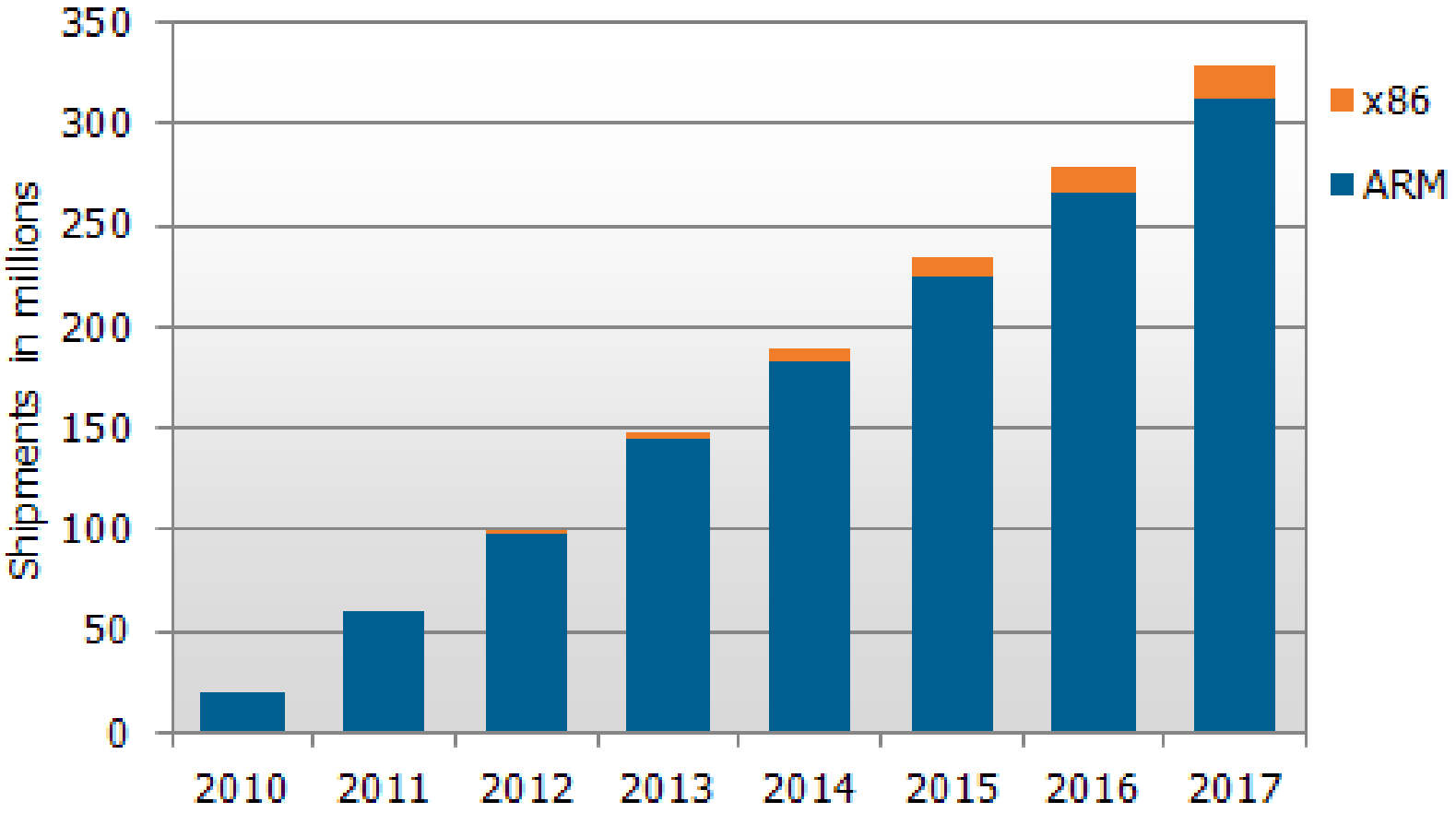
Floating Point Unit (FPU)



This is a block diagram of the CPU for the ARM computer we've been using in QEMU!

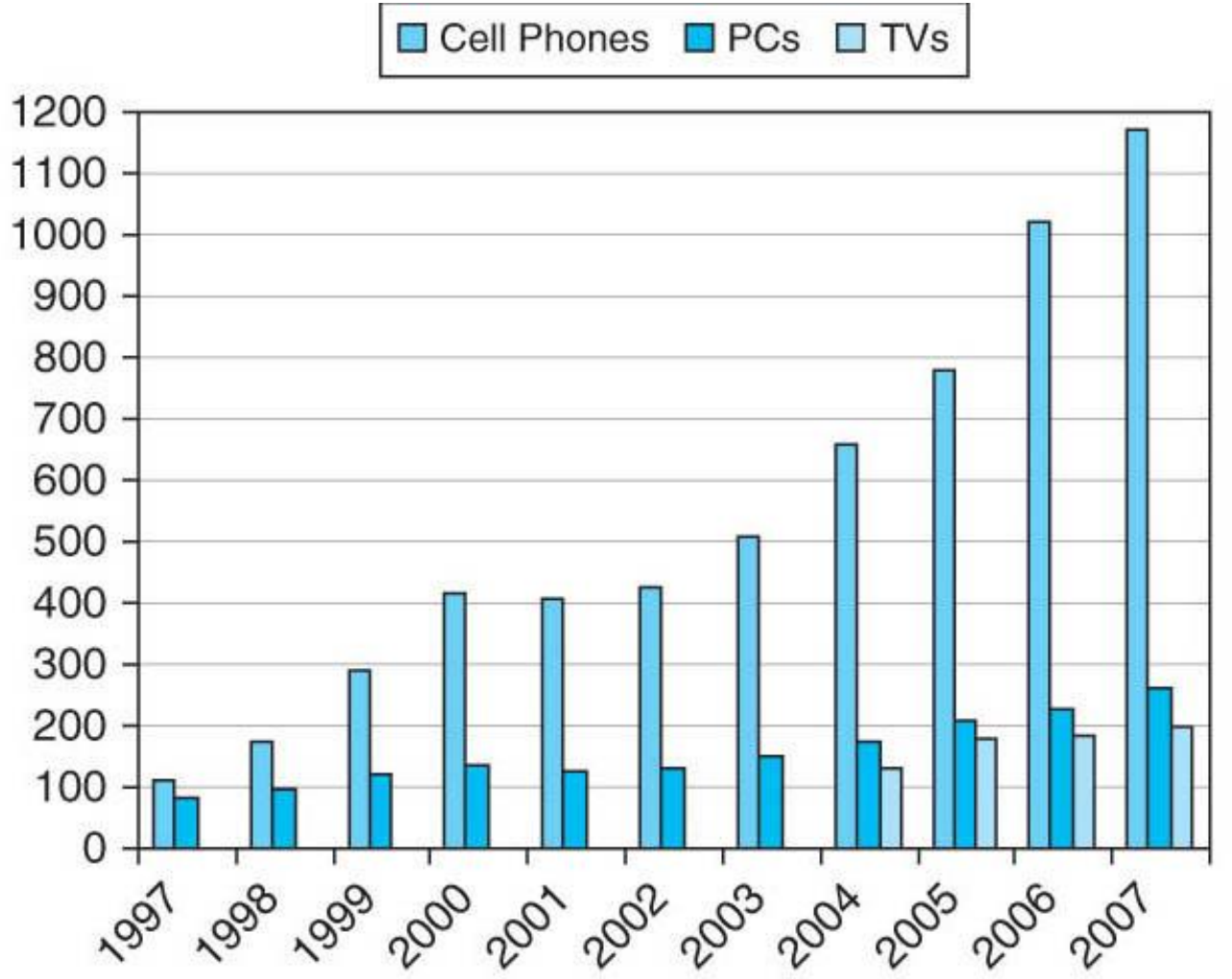
[http://www.atmel.com/Images/arm_926ejs_trm.pdf]

Why ARM?



http://www.displaysearch.com/cps/rde/xchg/displaysearch/hs.xsl/111024_tablet_pc_architectures_dominated_by_arm_and_ios.asp

Why ARM?


Cell Phones
PCs
TVs
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007

Why ARM?

- Easier to program

- RISC (reduced instruction set computing) vs. CISC (complex instruction set computing)
- RISC: ARM, MIPS, SPARC, Power, (i.e., lots of modern architectures), …
- CISC: x86, x86-64, lots of old architectures (PDP-11, VAX, …)
  - Note: modern x86 processors typically implemented internally as RISC (micro-instructions / microcode), but the programming interface is the same as x86

Review: ARM: Load/Store Architecture

- ARM is a load/store architecture
- This means that memory can only be accessed by load and store instructions
- All arguments for arithmetic and logical instructions must either:
  - Come from registers
  - Be constants specified within the instruction
    - (more examples of that later)
- This may not seem like a big deal to you, as you have not experienced the alternative
  - However, it makes life much easier
  - This is one reason why we chose ARM 7 for this course

ARM Arithmetic Instructions in Machine Language

Cond	F	I	Opcode	S	Rn	Rd	Operand2
1110	00	0	0100	0	0001	0101	0000 0000 0010
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits

- Example: `add r5, r1, r2`
- C equivalent: `r5 = r1 + r2`
- Machine language encoding above
- Opcode: 0100 means add (dependent on digital logic, some encoding)
- Rd: register destination operand. It gets the result of the operation
- Rn: first register source operand
- Operand2: second source operand
- I: Immediate. If I is 0, the second source operand is a register. If I is 1, the second source operand is a 12-bit immediate
- S: Set Condition Code
- Cond: Condition. Related to conditional branch instructions
- F: Instruction Format

Immediate/Literal Addressing

1. Immediate: ADD r2, r0, #5

cond	f	opcode	rn	rd	Immediate

- Operand comes from the instruction
- Example: 32-bit instruction to move 4 into R1
  - Result is R1 := 4

MOV R1 #4

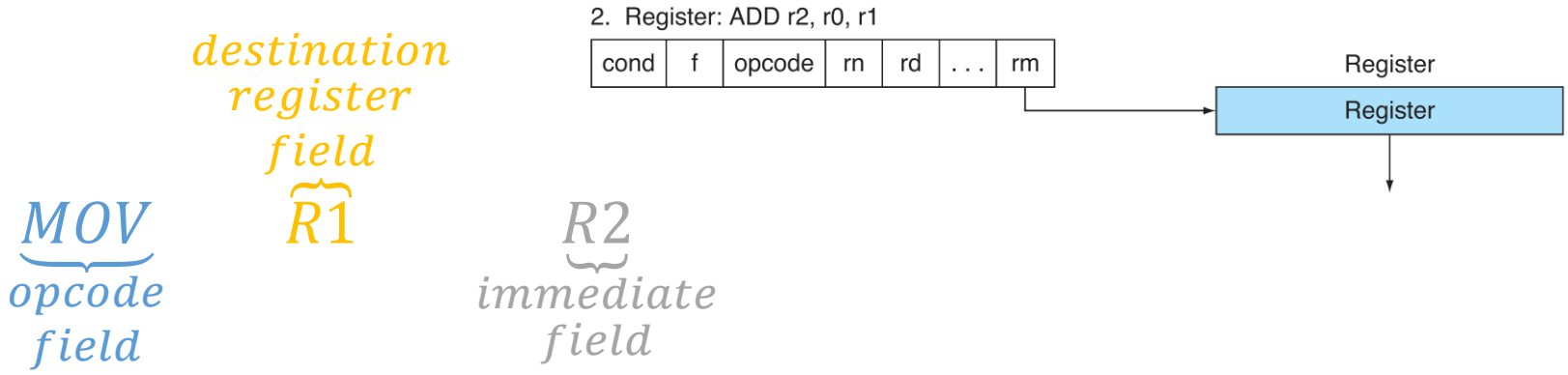
destination register field

MOV (*opcode field*) *R1* #4 (*immediate field*)

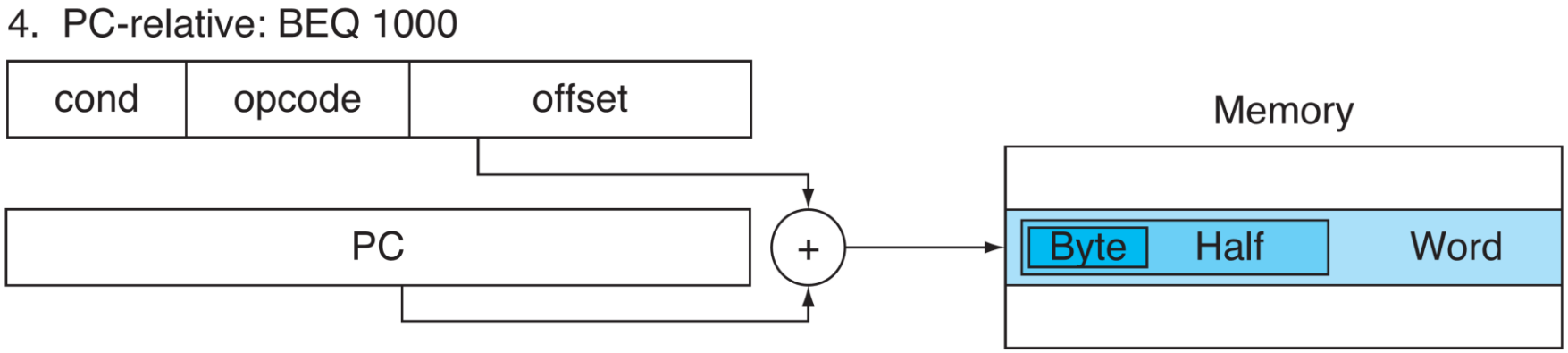
- Useful for specifying small integer constants (avoids extra memory access)
- Can only specify small constants (limited by size of immediate field)
  - ARM: typically 8-12-bits

Register/Register-Direct Addressing

- Operand(s) come(s) from register(s)
  - Seen this many times already: ADD R0 R1 R2 does R0 := R1 + R2
  - Also: MOV R1 R2: the destination operand is specified by its register address (Result is R1 := R2)

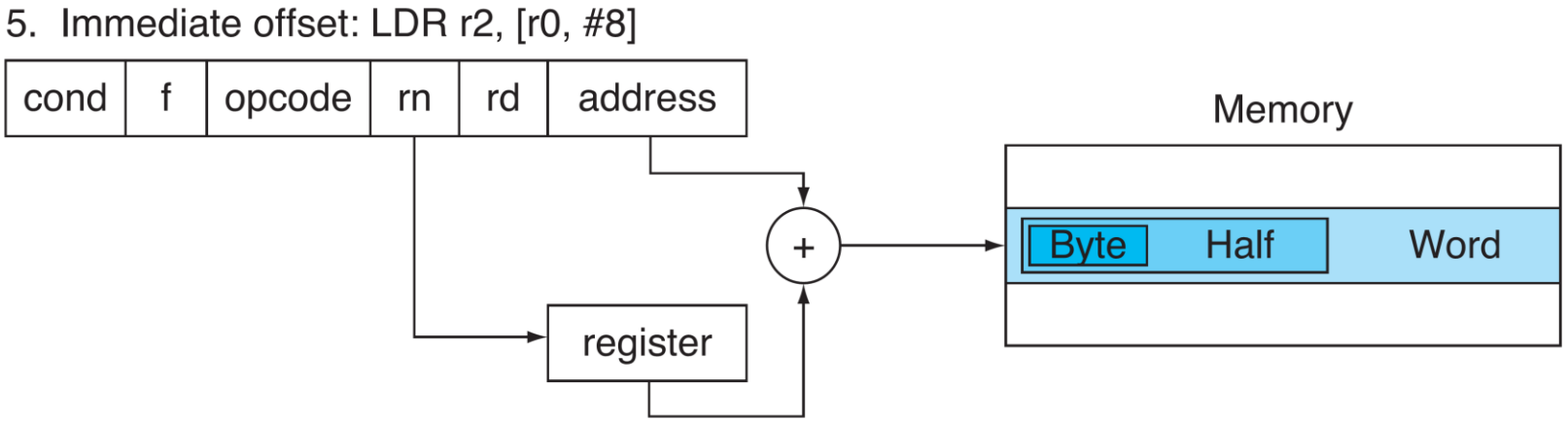


PC-Relative Indirect



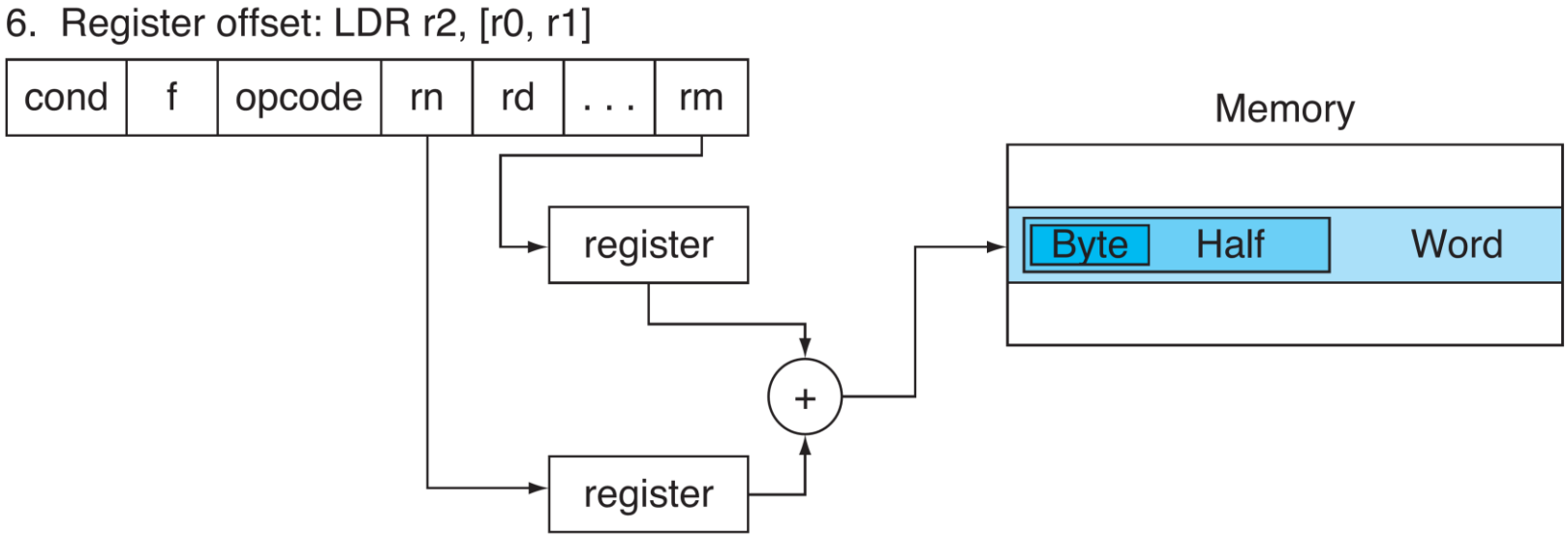
- Uses the current PC value with an immediate offset to determine the value
- Example: branch if equal to location PC + 1000
  - Updates PC = MEM[PC + 1000]
- Example: LDR r6, [PC]
  - Updates r6 = MEM[PC]

Indirect with Immediate Offset



- Uses a register value and an immediate offset
- Example: LDR r2, [r0, #8]
  - Updates r2 = MEM[r0 + #8]

Indirect Register Offset



- Uses a register value and another register value as an offset
- Example: LDR r2, [r0, r1]
  - Updates r2 = MEM[r0 + r1]

Making a Function

- Functions are easy to define and call in languages like C and Java
- In assembly, calling a function requires several steps
- This reflects that the CPU can do only a limited amount of work in a single step
- Note that, to correctly do a function call, both the caller (program/function making the function call) and the called function must do the right steps

Caller Steps

- **Step 1: Put arguments in the right place.**
- Specific machines use specific conventions.
  - "R0-R3 hold parameters to the procedure being called".
- So:
  - Argument 1 (if any) goes to r0.
  - Argument 2 (if any) goes to r1.
  - Argument 3 (if any) goes to r2.
  - Argument 4 (if any) goes to r3.
- If there are more arguments, they have to be placed in memory. We will worry about this case only if we encounter it.

Caller Steps

- **Step 2: branch to the first instruction of the function.**
  - Here, we typically use the bl instruction, not the b instruction.
  - The bl instruction, before branching, saves to register lr (the link register, aka r14) the return address.
  - The **return address** is the address of the instruction that should be executed when the function is done.
- **Step 3: after the function has returned, recover the return value, and use it.**
  - We will follow the convention that the return value goes to r0.
  - If there is a second return value, it goes to r1.

Called (callee) Function Steps

- Step 1: Do the preamble:
  - Allocate memory on the stack (more details in a bit).
  - Save to memory the return address. Why?
  - Save to memory all registers (except possibly for r0) that the function modifies. Why?
- Step 2: Do the main body of the function.
  - Assume arguments are in r0, r1, r2, r3.
  - This is where the actual work is done.
- Step 3: Do the wrap-up:
  - Store the return value (if any) on r0, and second return value (if any) on r1.
  - Retrieve from memory the return address. Why?
  - Retrieve from memory, and restore to registers, the original values of all registers that the function modified (except possibly for r0). Why?
  - Deallocate memory on the stack.
  - Branch to the return address.

Assembly Language Format

Label	Opcode	Operands	Comments
iloop:	add	r1,r1,#1	@ r1 := r1 + 1
	b	iloop	@ pc := iloop
val:	.byte	0x9F	@ put 0x96 at address val
s:	.asciz	"hello!"	@ put "hello!" at sequential addresses starting at address s

Representing Data

- Finite precision numbers
  - Unsigned integers
  - Signed integers
    - Two's complement
  - Word ints (32-bits) vs. longs/doubles (64-bits)
  - Rational numbers
    - Fixed point
    - Floating point
- Strings / character arrays
  - ASCII
  - Unicode

Review: Two's Complement Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1
  - Representation called one's complement

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

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

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

Review: Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits (also called a nibble or nybble) per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: 0xECA8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

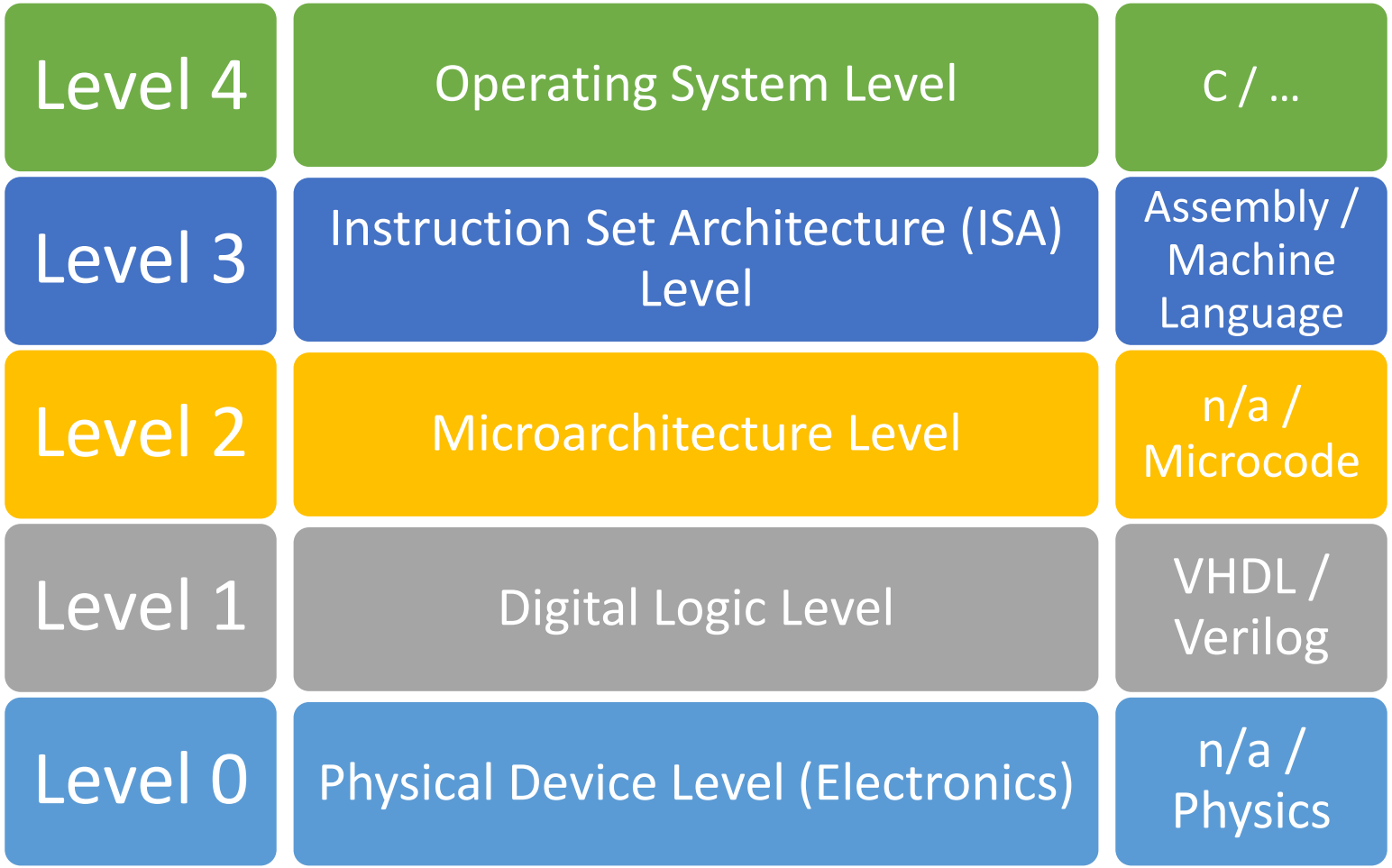
Review: Arithmetic Operations

- Add and subtract, three ***operands***
  - ***Operand:*** *quantity on which an operation is performed*
  - Two sources and one destination

```
add a, b, c  # a updated to b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

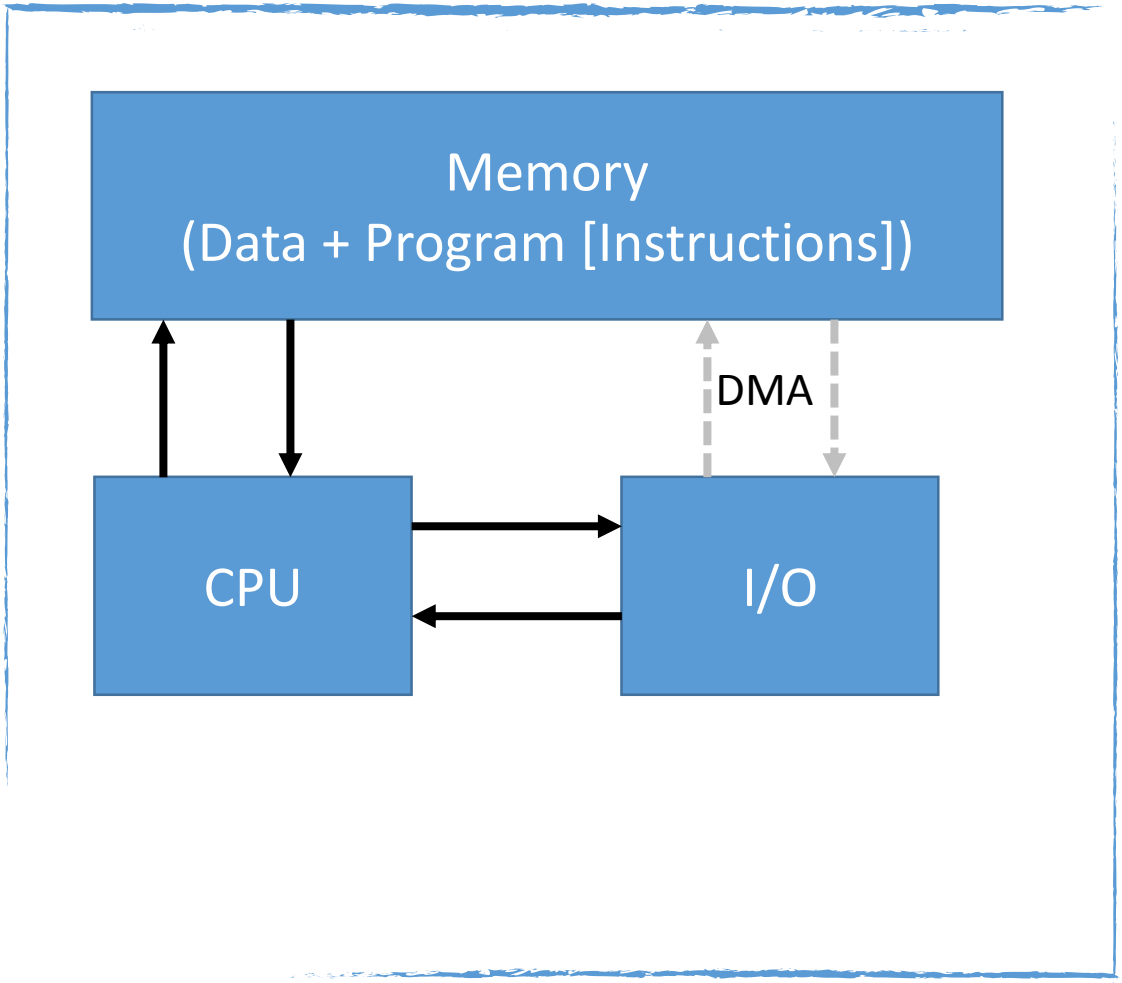
Multilevel Architectures



Processor (CPU) Components

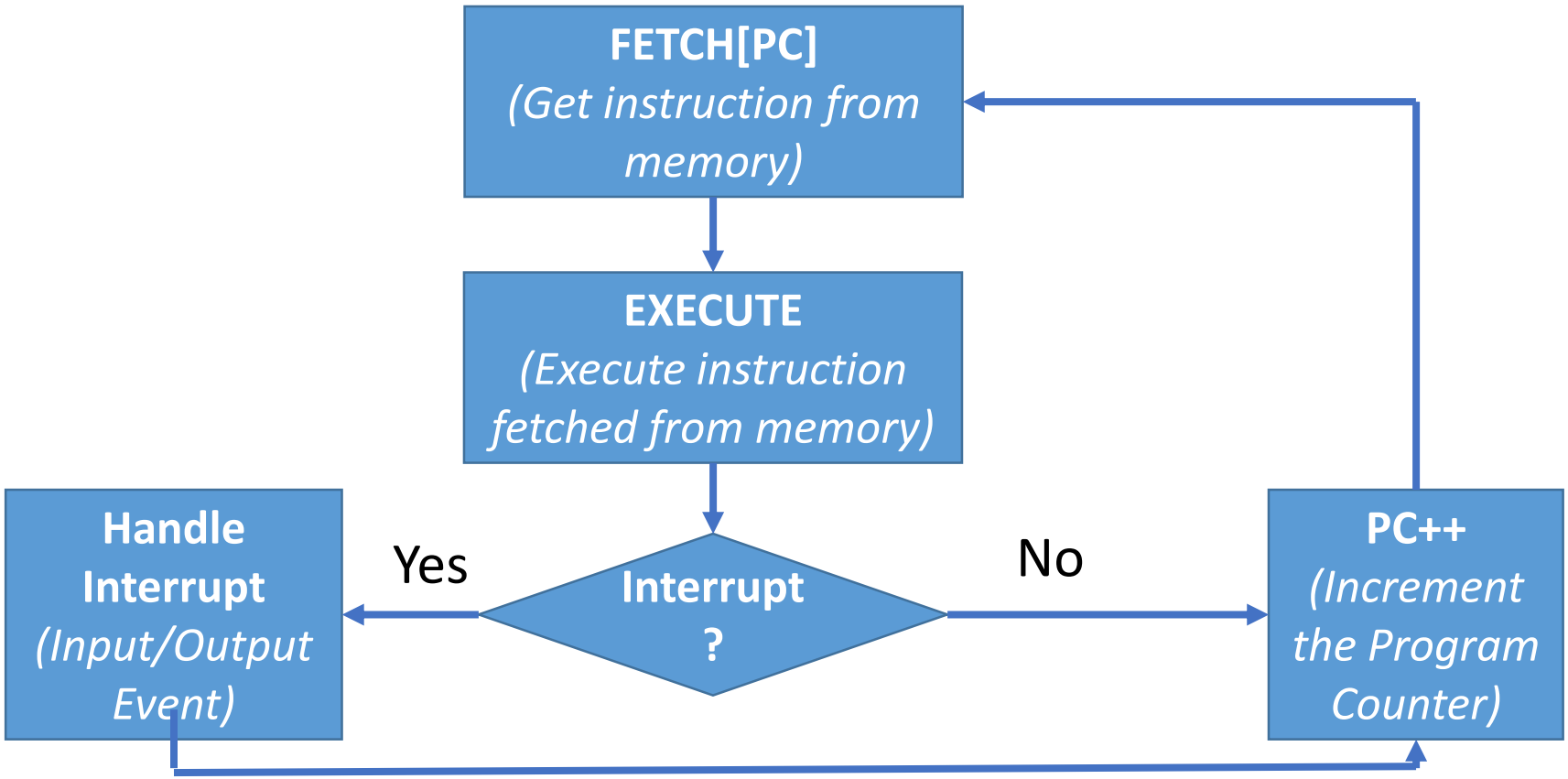
- Pipeline: stages (fetch, decode, execute)
- ALU: arithmetic logic unit
- MMU: memory management unit
  - TLB: translation lookaside buffer (cache for virtual memory)
- Cache (L1, L2, L3, …)
  - Caches for main memory
- Registers
  - Hold values for all ongoing computations (i.e., only can do computation on these values, otherwise first load/store)
- FPU: floating point unit

Von Neumann Architecture

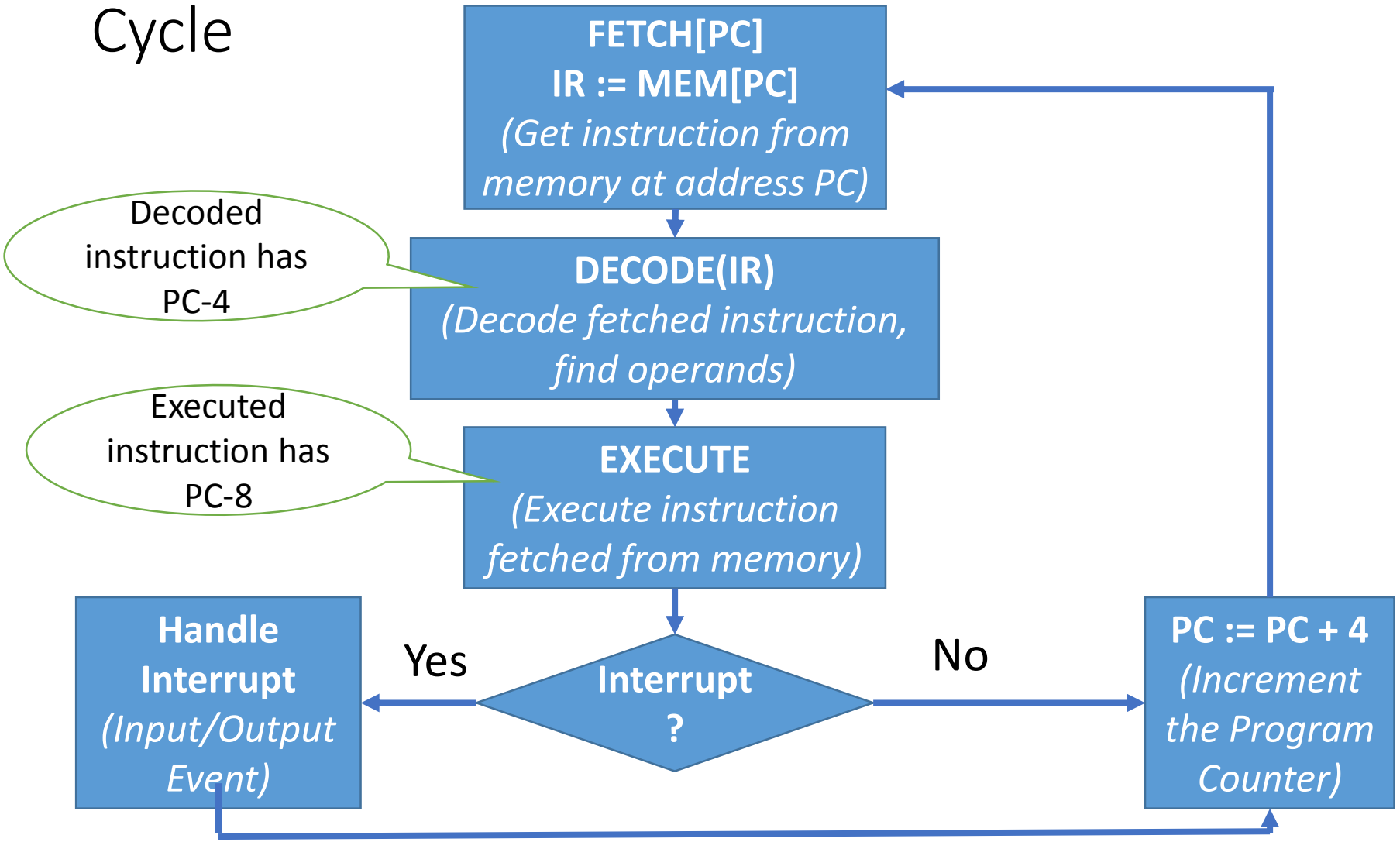


- Both data and program stored in memory
- Allows the computer to be “re-programmed”
- Input/output (I/O) goes through CPU
- I/O part is not representative of modern systems (direct memory access [DMA])
- Memory layout is representative of modern systems

Abstract Processor Execution Cycle



ARM 3-Stage Pipeline Processor Execution Cycle

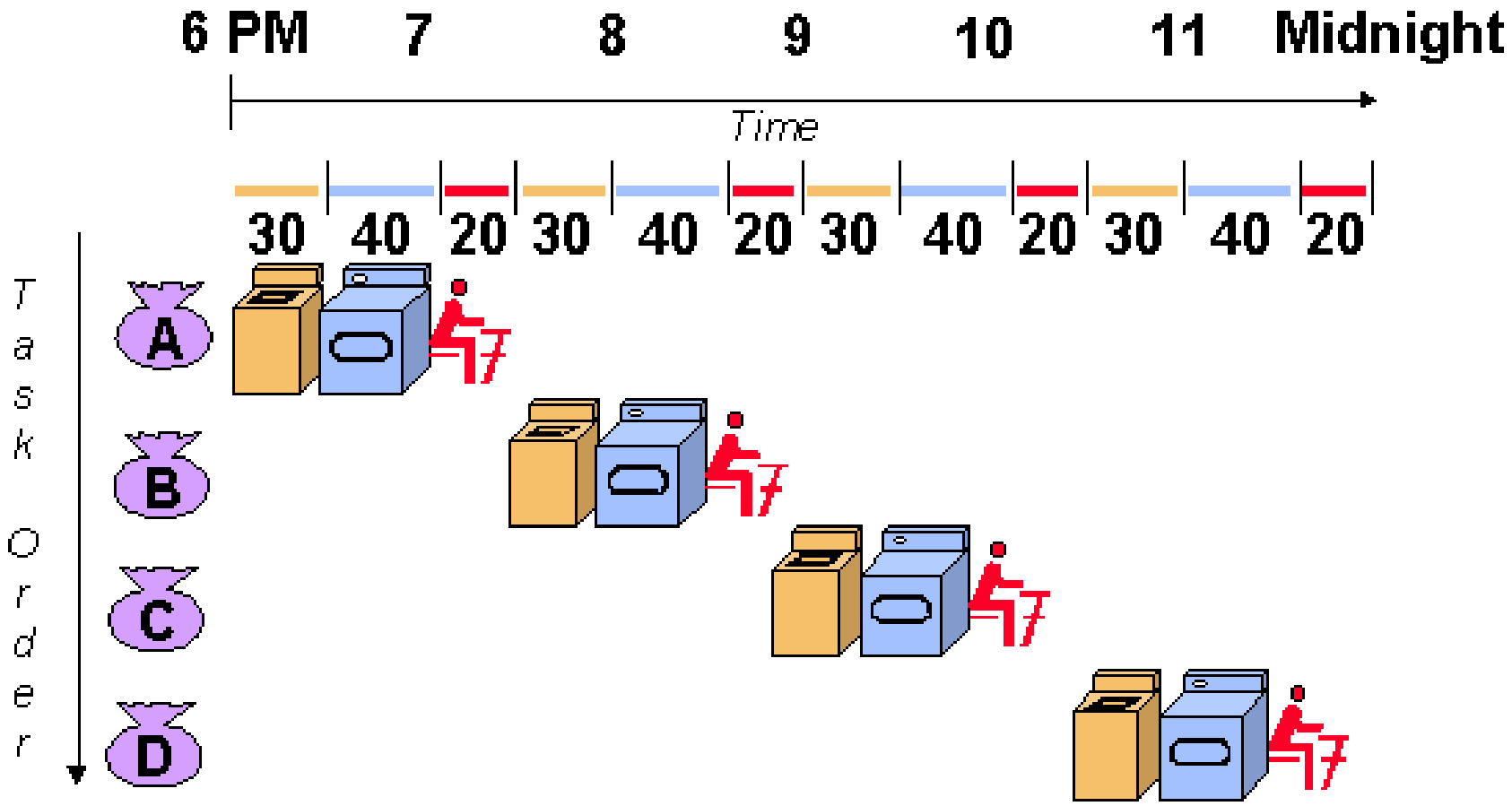


ARM 3 Stage Pipeline

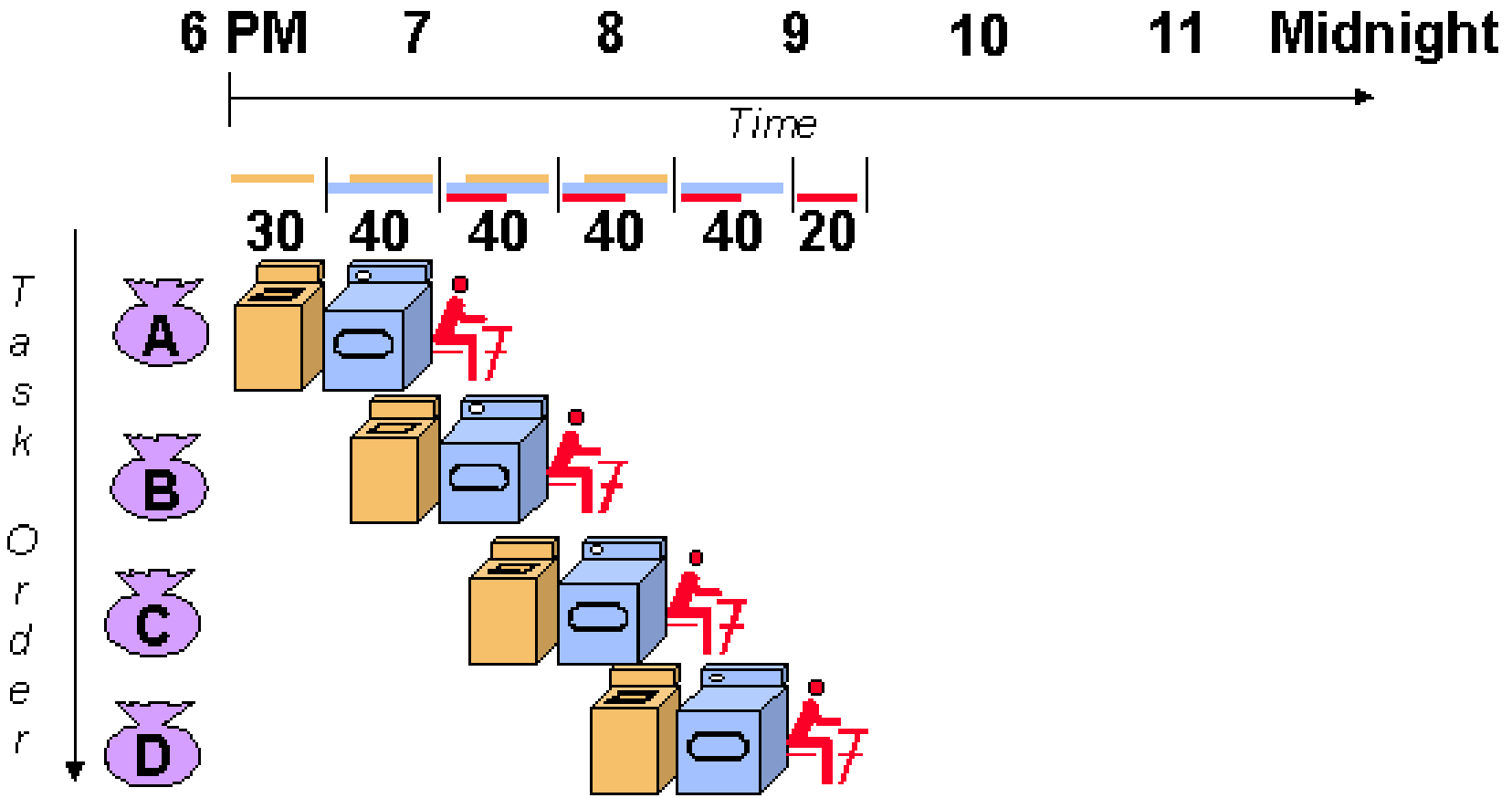
- Stages: fetch, decode, execute
- PC value = instruction being fetched
- PC – 4: instruction being decoded
- PC – 8: instruction being executed

- Beefier ARM variants use deeper pipelines (5 stages, 13 stages)

Un-Pipelined Laundry

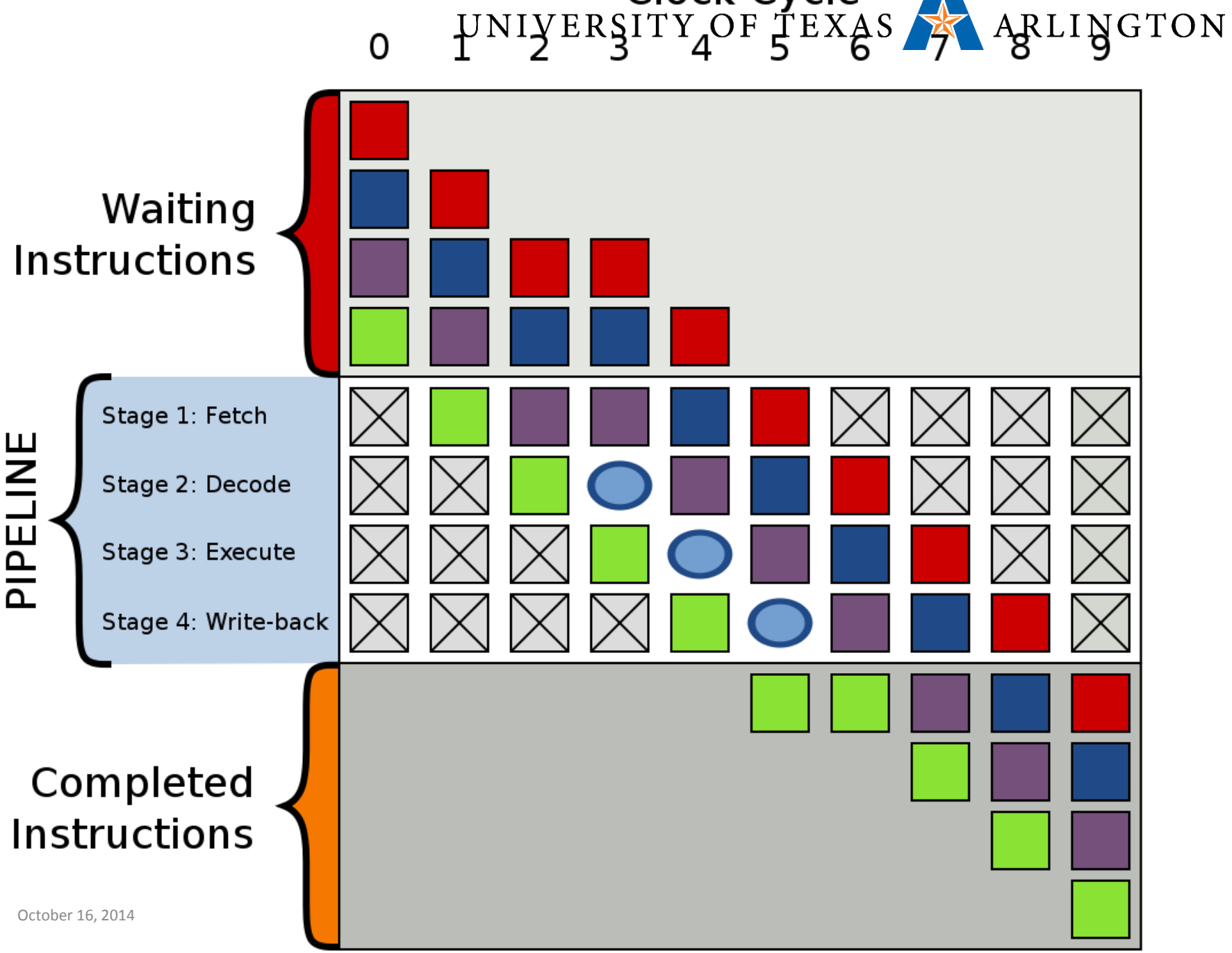


Pipelined Laundry

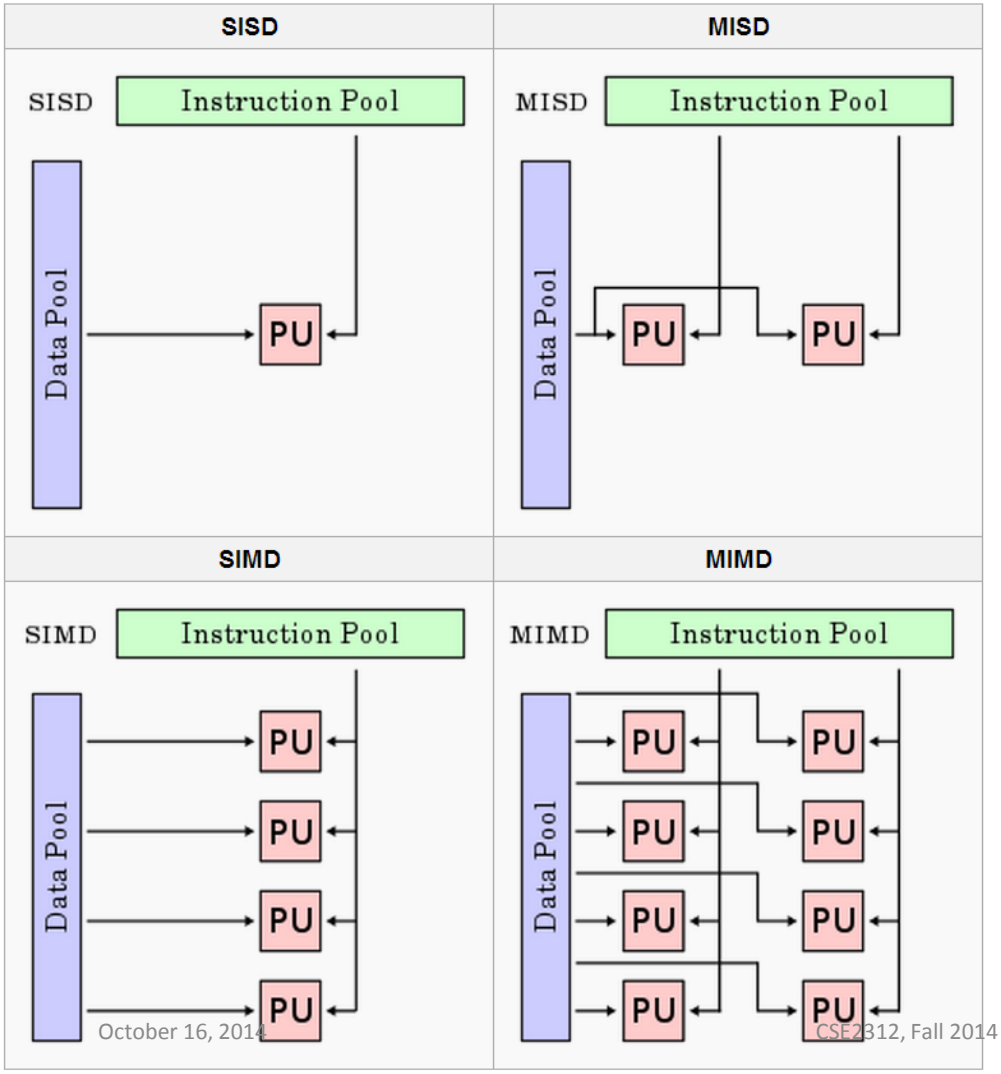


Why Pipelining?

- Consider a five-stage pipeline
  - Suppose 2ns for the cycle period
  - It takes 10ns for an instruction to progress all the way through pipeline
  - So, the machine runs at 100 MIPS?
  - Actual rate: 500 MIPS
- Pipelining
  - Tradeoff between latency and processor bandwidth
  - Latency: how long it takes to execute an instruction
  - Processor bandwidth: MIPS of the CPU
- Example
  - Suppose a complex instruction should take 10 ns, under perfect conditions, how many stage pipeline should we design to guarantee 500 MIPS?
  - Each pipeline stage should take: 1/500 MIPS = 2 ns
  - 10 ns/ 2ns =5 stages


Clock Cycle
0
1
2
3
4
5
6
7
8
9
Waiting Instructions
PIPELINE
Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back
Completed Instructions

Flynn's Taxonomy

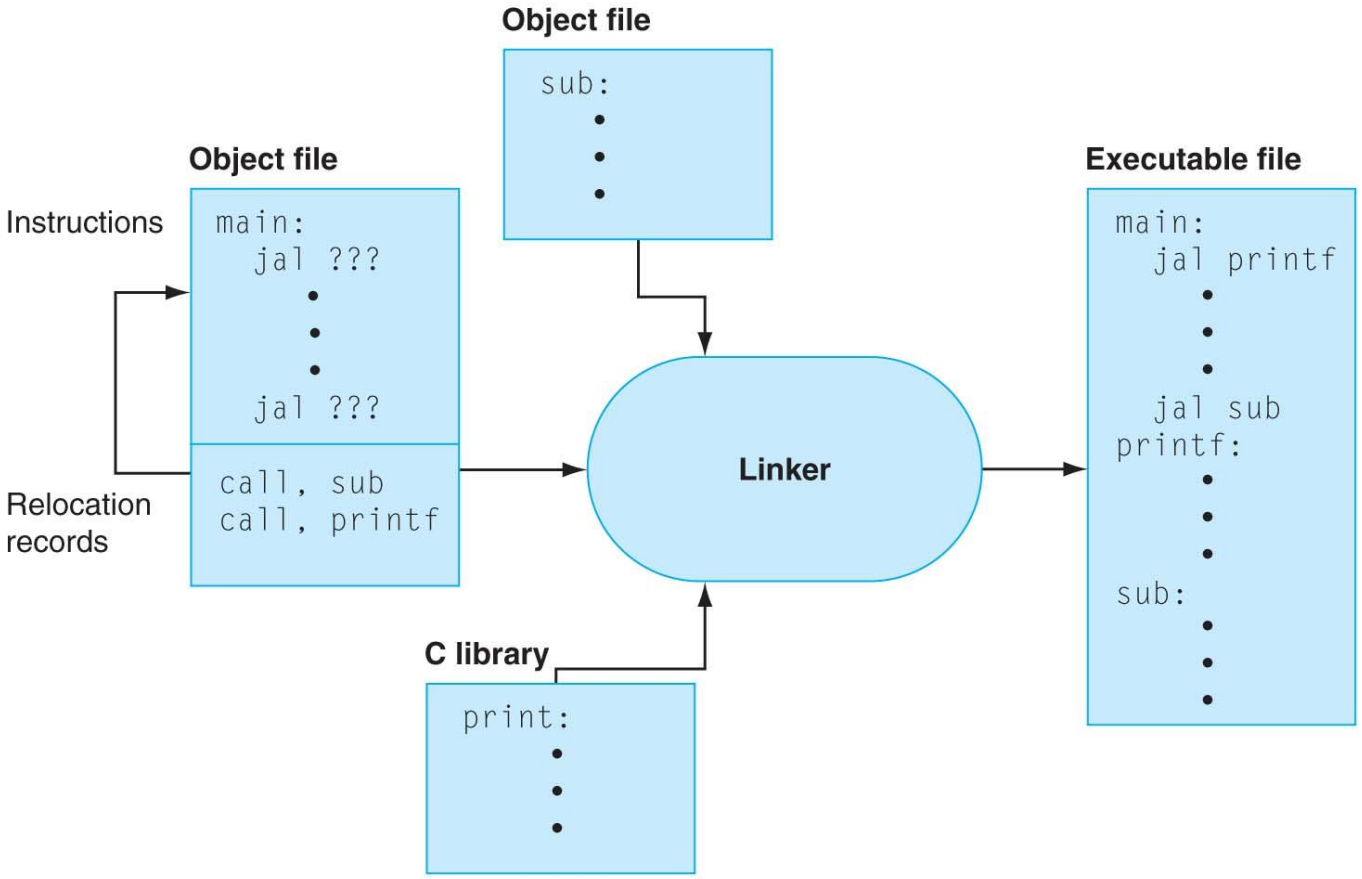


- SISD: Single Instruction, Single Data
  - Classical Von Neumann
- SIMD: Single Instruction, Multiple Data
  - GPUs
- MISD: Multiple Instruction, Single Data
  - More exotic: fault-tolerant computers using task replication (Space Shuttle flight control computers)
- MIMD: Multiple Instruction, Multiple Data
  - Multiprocessors, multicomputers, server farms, clusters, …

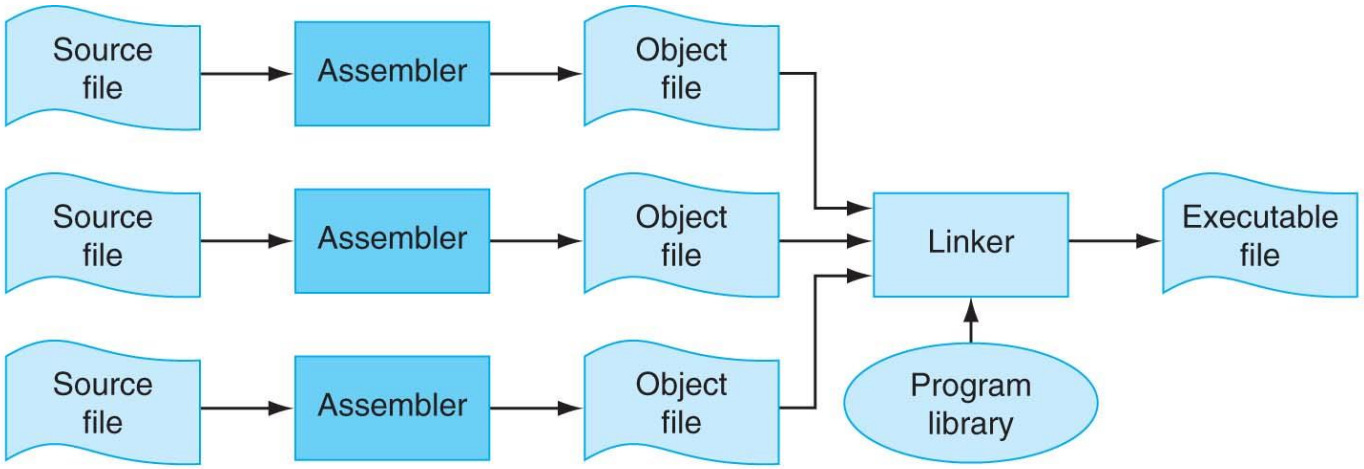
C to Assembly and Machine Language

- How did we go from ASM to machine language?
  - Two-pass assembler
- How do we go from C to machine language?
  - Compilation
  - Can think of as generating ASM code, then assembling
- Optimizations

Linker Process



Assembly Process



The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

QEMU

- Virtual machine: Quick-Emulator: http://www.qemu.org
- "QEMU is a generic and open source machine emulator and virtualizer."

- "When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance."

- "When used as a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, and S390 guests."

- ***QEMU runs like any other Linux process/program***

GDB Commands

- `b label`
  Sets a breakpoint at a specific label in your source code file. In practice, for some weird reason, the code actually breaks not at the label that you specify, but after executing the next line.
- `b line_number`
  Sets a breakpoint at a specific line in your source code file. In practice, for some weird reason, the code actually breaks not at the line that you specify, but at the line right after that.
- `c`
  Continues program execution until it hits the next breakpoint.
- `i r`
  Shows the contents of all registers, in both hexadecimal and decimal representations; short for `info registers`
- `list`
  Shows a list of instructions around the line of code that is being executed.
- `quit`
  This command quits the debugger, and exits GDB.
- `stepi`
  This command executes the next instruction.
- `set $register=val`
  `set $pc=0`
  This command updates a register to be equal to val, for example, to restart your program, set the PC to 0
- `monitor quit`
  Send the remote monitor (e.g., QEMU in our case) a command, in this case, tell QEMU to terminate; Call this before quiting gdb so that the QEMU process gets killed!

Instruction Set Architectures

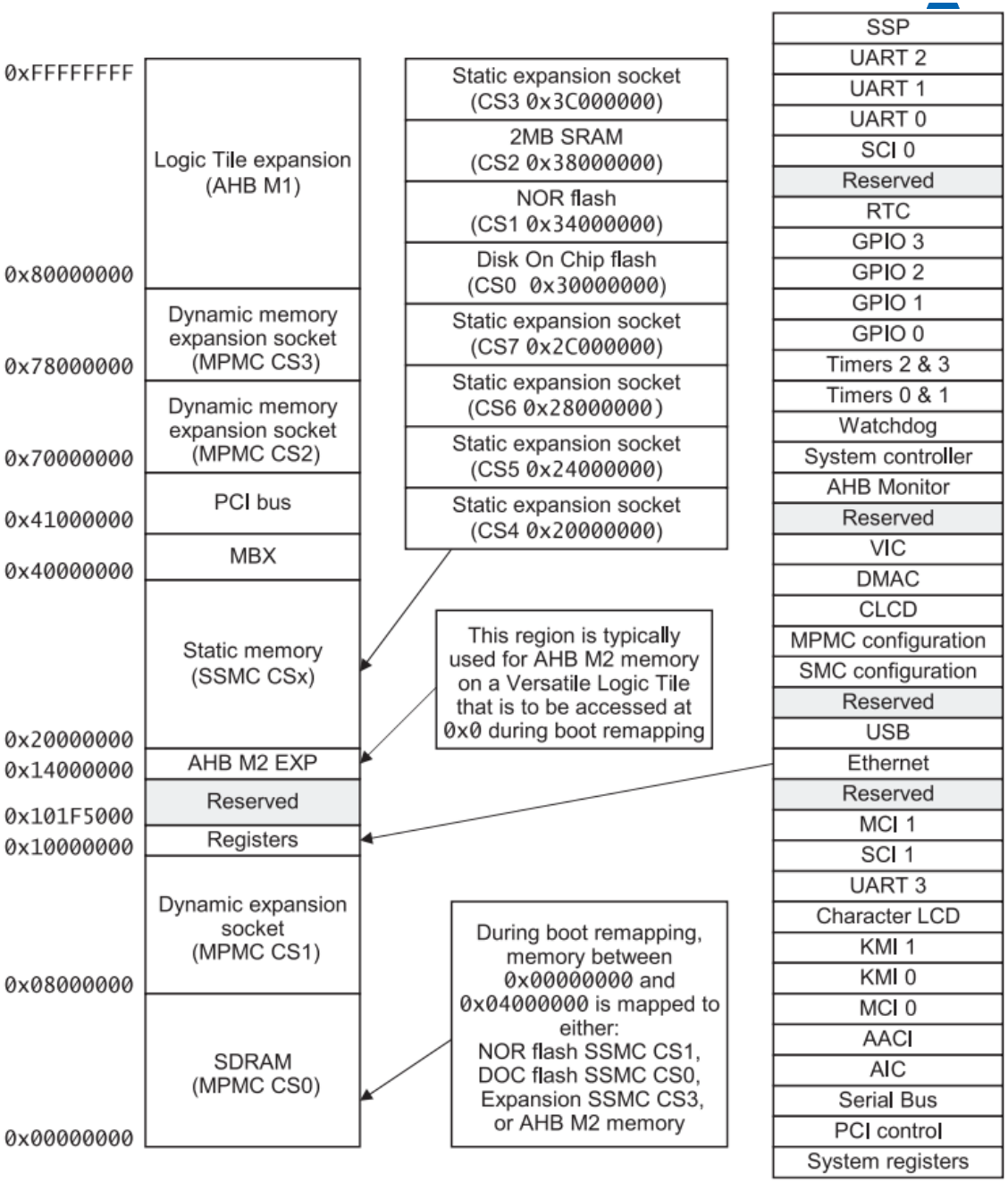
- Interface between software and hardware
- Examples: x86, x86-64, ARM, AVR, SPARC, ALPHA, MIPS
  - RISC vs. CISC
- High-level language to computer instructions
  - How do we execute a high-level language (e.g., C, Python, Java) using instructions the computer can understand?
    - Compilation (translation before execution)
    - Interpretation (translation-on-the-fly during execution)
  - What are examples of these processes?

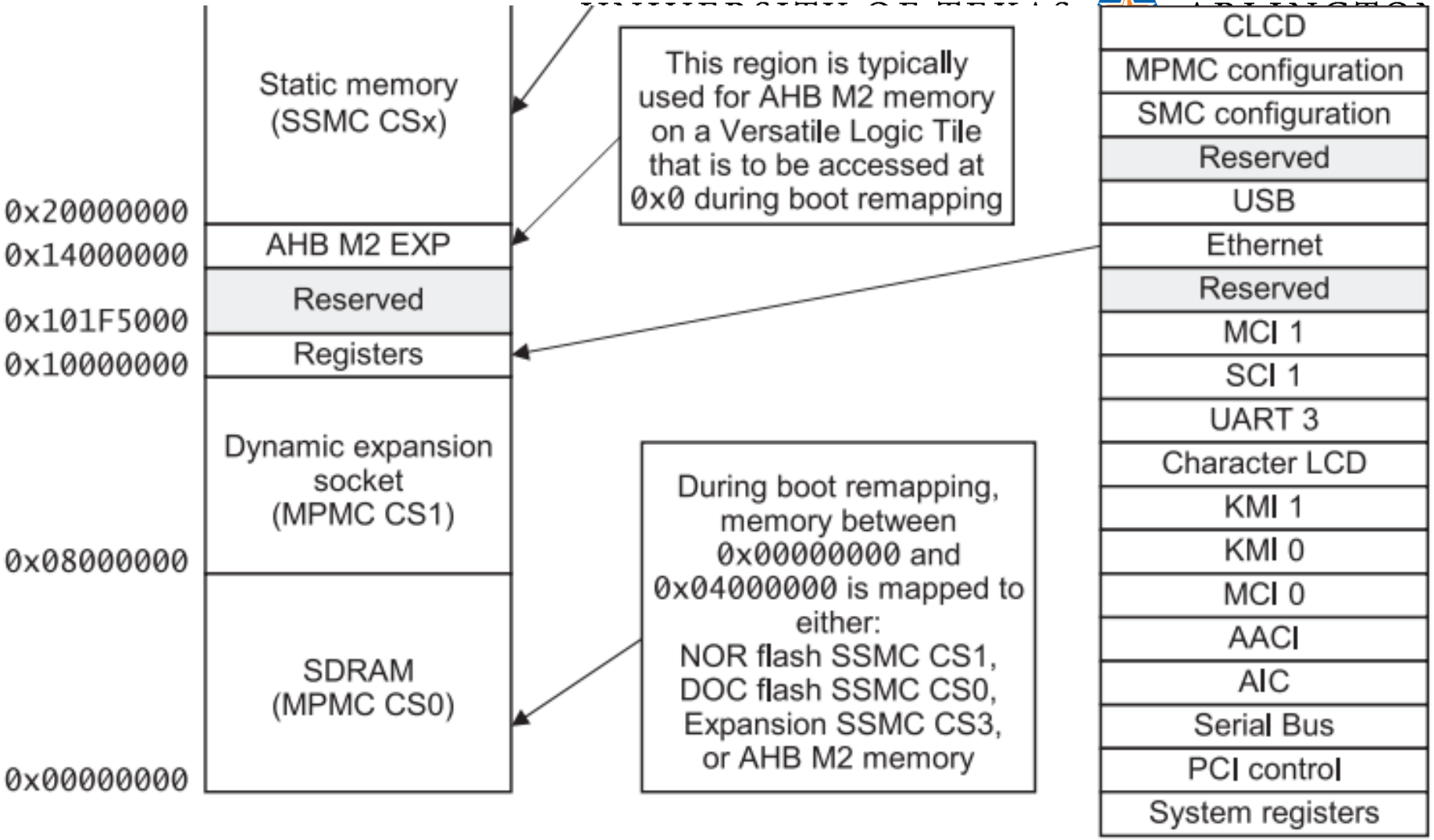
Memory-Mapped I/O

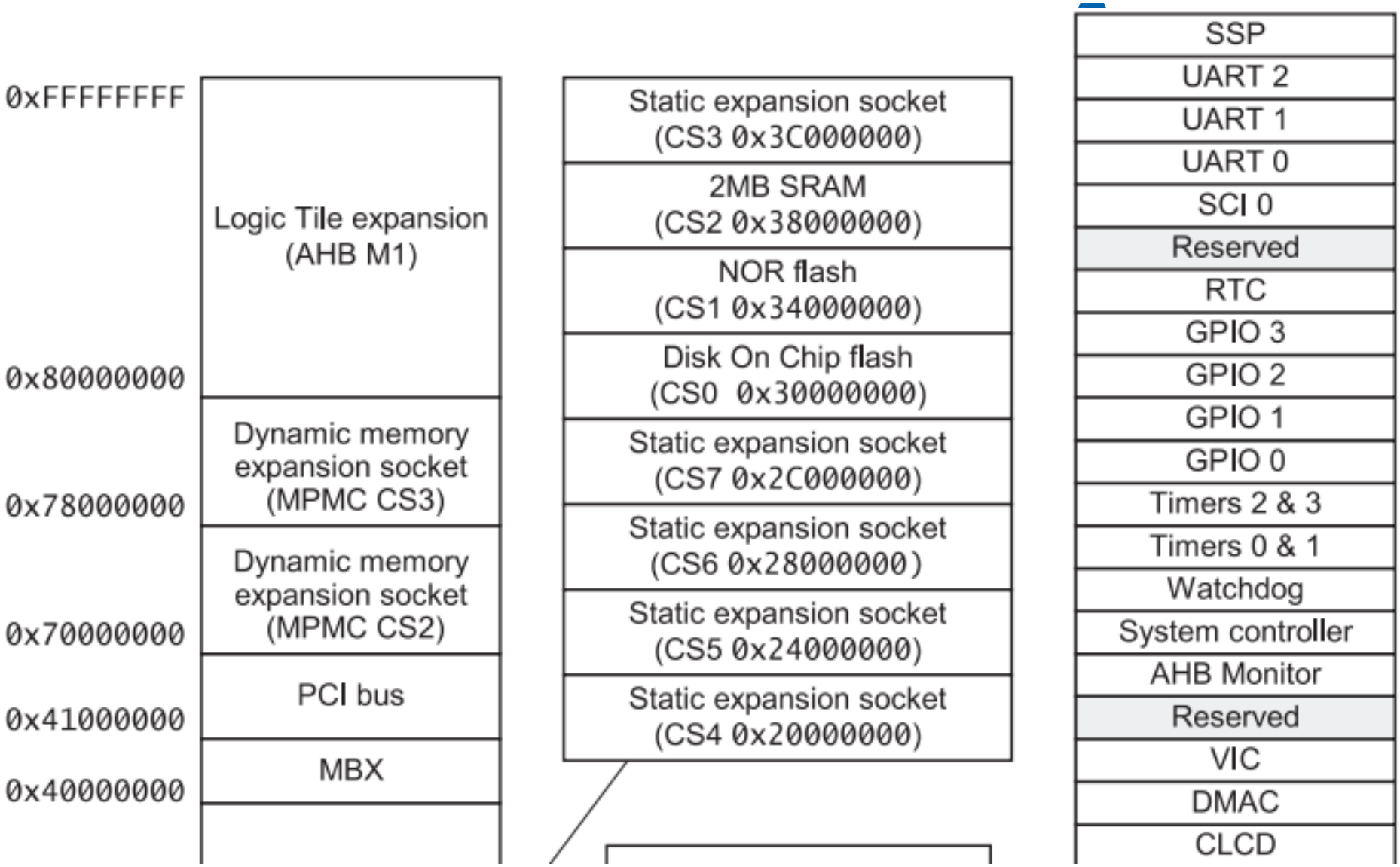
- Some of our original examples displayed output to console by writing to a special memory address

```
.equ    ADDR_UART0, 0x101f1000
ldr     r0,=ADDR_UART0@ r0 := 0x 101f 1000
mov     r2,#'a'            @ R2 := 'a'
str     r2,[r0]            @ MEM[r0] := r2
```

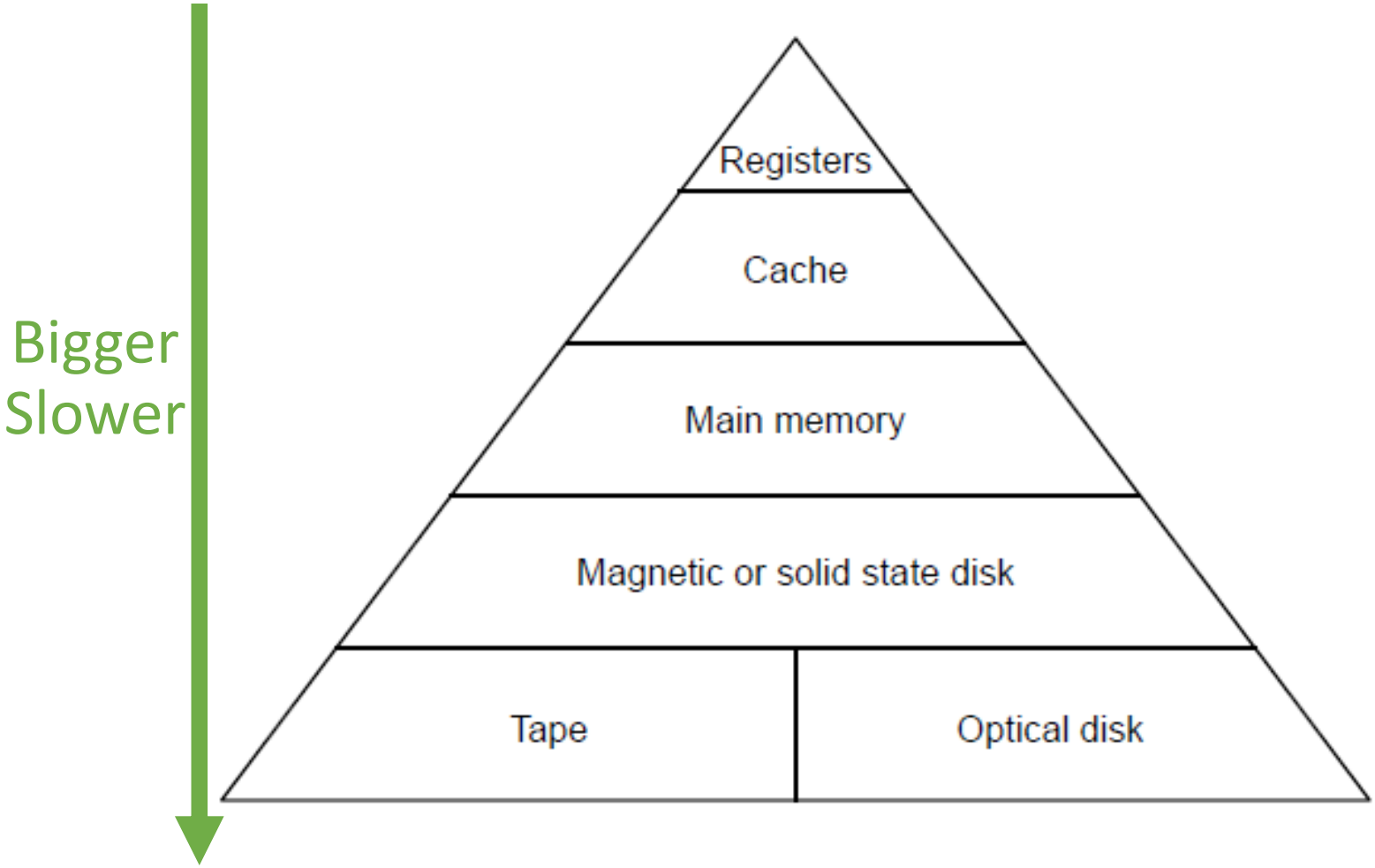
- How does this work? Memory Mapped I/O
  - Registers on peripheral devices (keyboards, monitors, network controllers, etc.) are addressable in same address space as main memory, and their values are mapped (i.e., readable / writeable at certain addresses)
  - How to read input values?
    - Polling vs. interrupts


0xFFFFFFFF
Logic Tile expansion (AHB M1)
0x80000000
Dynamic memory expansion socket (MPMC CS3)
0x78000000
Dynamic memory expansion socket (MPMC CS2)
0x70000000
PCI bus
0x41000000
MBX
0x40000000
Static memory (SSMC CSx)
0x20000000
AHB M2 EXP
0x14000000
Reserved
0x101F5000
Registers
0x10000000
Dynamic expansion socket (MPMC CS1)
0x08000000
SDRAM (MPMC CS0)
0x00000000
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping
During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either: NOR flash SSMC CS1, DOC flash SSMC CS0, Expansion SSMC CS3, or AHB M2 memory
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD
MPMC configuration
SMC configuration
Reserved
USB
Ethernet
Reserved
MCI 1
SCI 1
UART 3
Character LCD
KMI 1
KMI 0
MCI 0
AACI
AIC
Serial Bus
PCI control
System registers

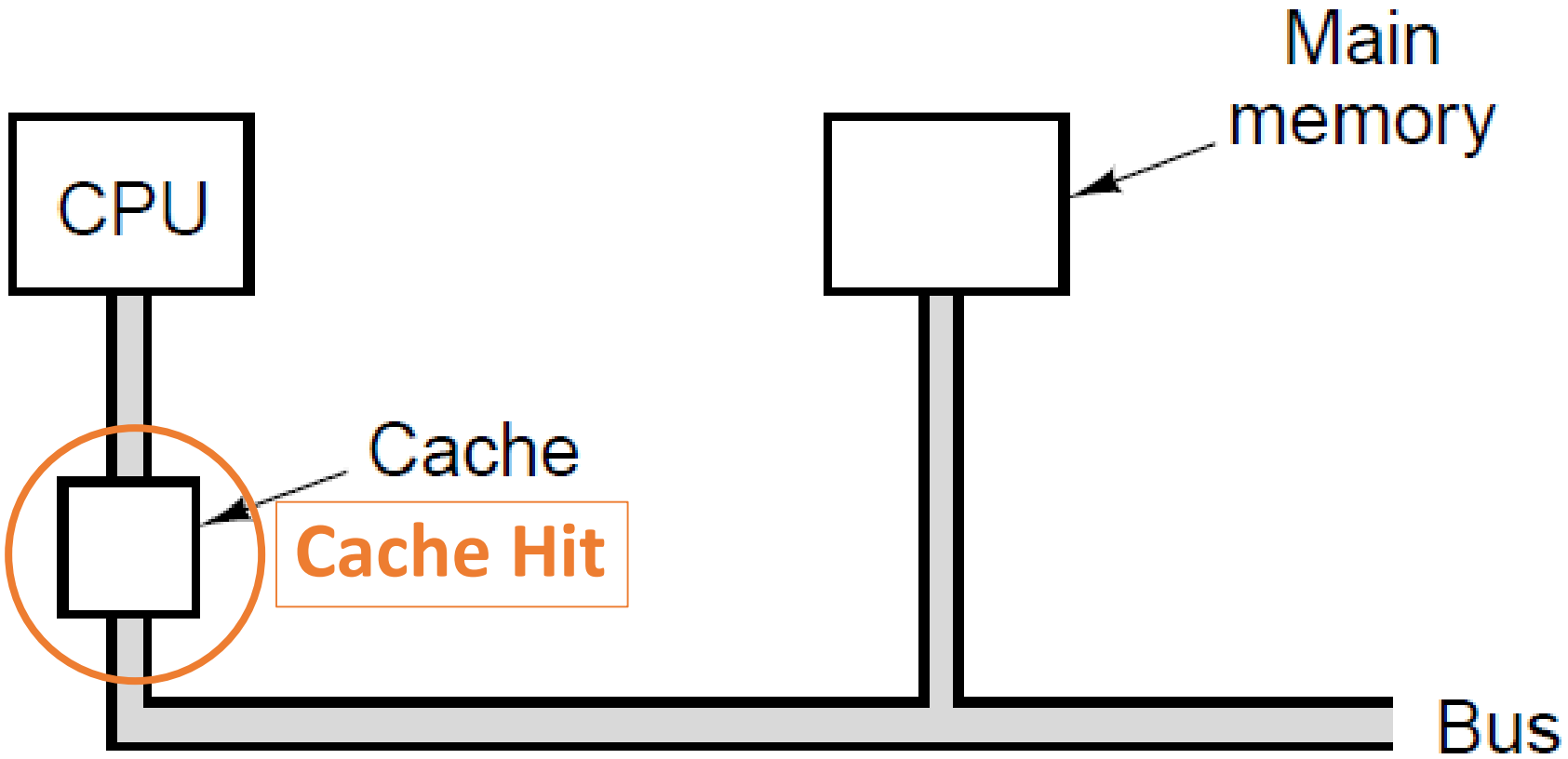

Static memory (SSMC CSx)
AHB M2 EXP
Reserved
Registers
Dynamic expansion socket (MPMC CS1)
SDRAM (MPMC CS0)
0x20000000
0x14000000
0x101F5000
0x10000000
0x08000000
0x00000000
This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping
During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either: NOR flash SSMC CS1, DOC flash SSMC CS0, Expansion SSMC CS3, or AHB M2 memory
CLCD
MPMC configuration
SMC configuration
Reserved
USB
Ethernet
Reserved
MCI 1
SCI 1
UART 3
Character LCD
KMI 1
KMI 0
MCI 0
AACI
AIC
Serial Bus
PCI control
System registers


0xFFFFFFFF
Logic Tile expansion (AHB M1)
0x80000000
Dynamic memory expansion socket (MPMC CS3)
0x78000000
Dynamic memory expansion socket (MPMC CS2)
0x70000000
PCI bus
0x41000000
MBX
0x40000000
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD

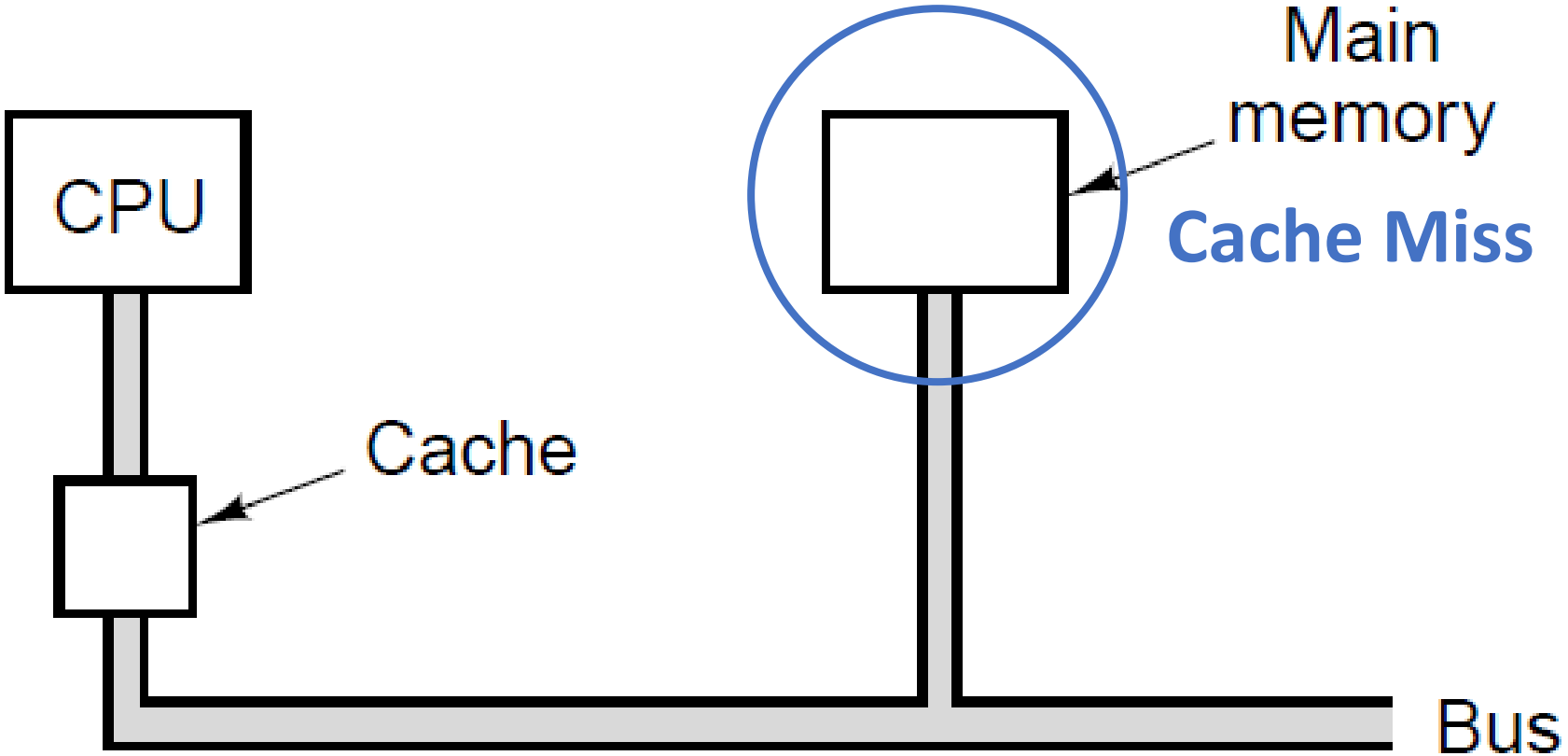
Memory Hierarchy



Cache Hit: find necessary data in cache



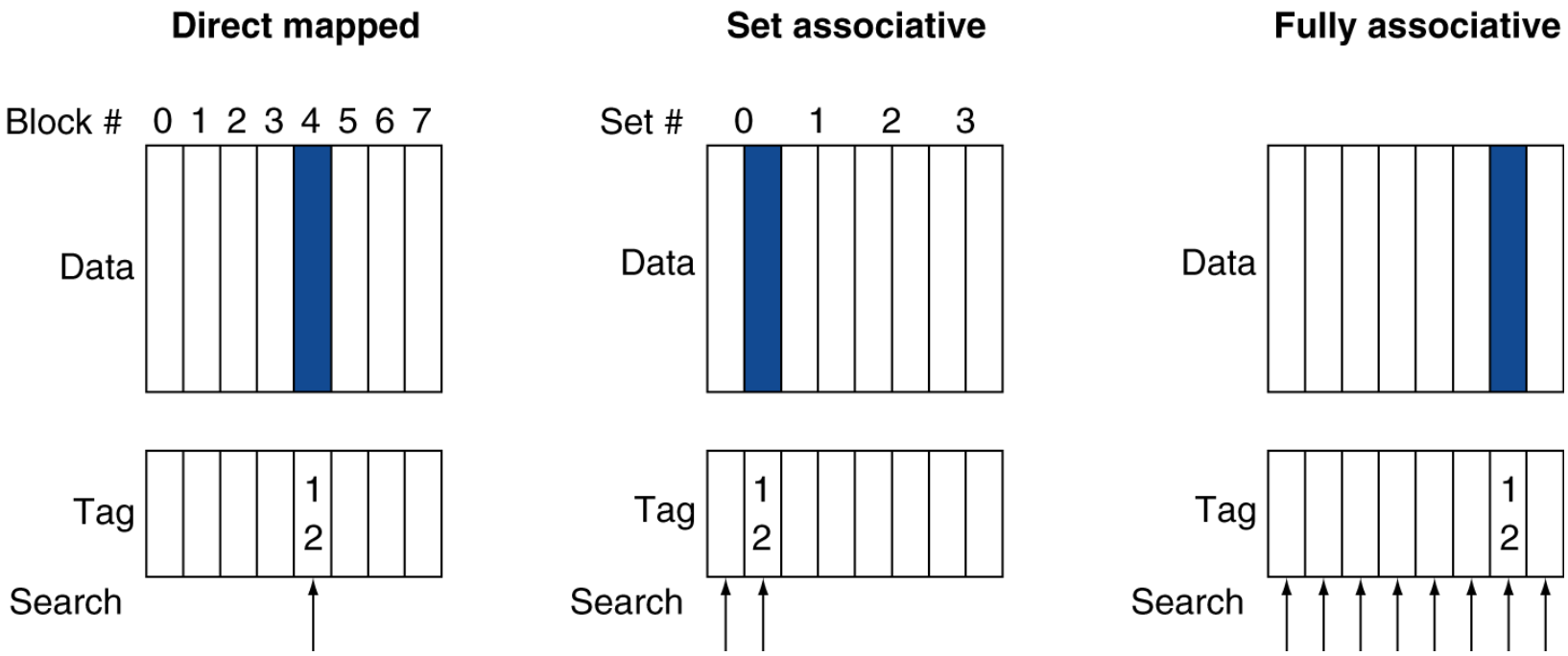
Cache Miss: have to get necessary data from main memory



Cache Terms

- Cache line: block of cells inside a cache
  - Usually store several words in a line (e.g., store 32 bytes on 32-bit word CPU)
- Cache hit: memory access finds value in cache
  - Antonym: cache miss: have to get it from main memory
- Spatial locality: likely we need data from addresses around one we're requesting (example: array operations)
- Mean access time: C + (1 – H) * M
  - C: cache access time
  - M: main memory access time (usually M >> C, e.g., M > 100 * C)
  - H: hit ratio: probability to find a value in the cache
  - miss ratio: 1 – H
- Time cost of cache miss: C + M memory access time

Associative Cache Example

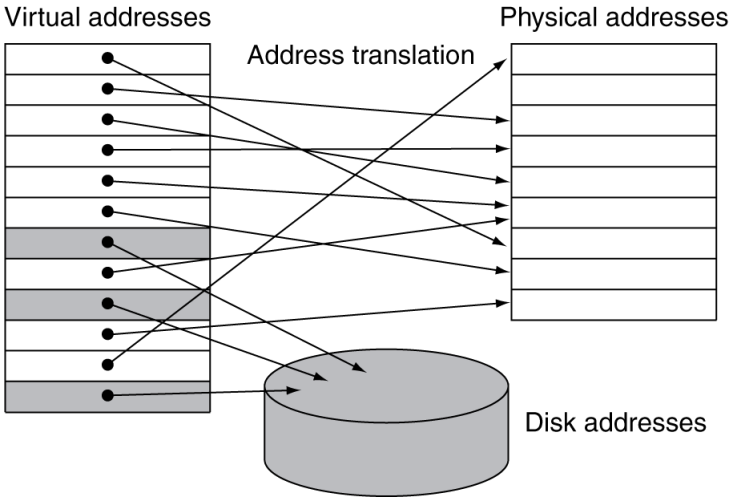


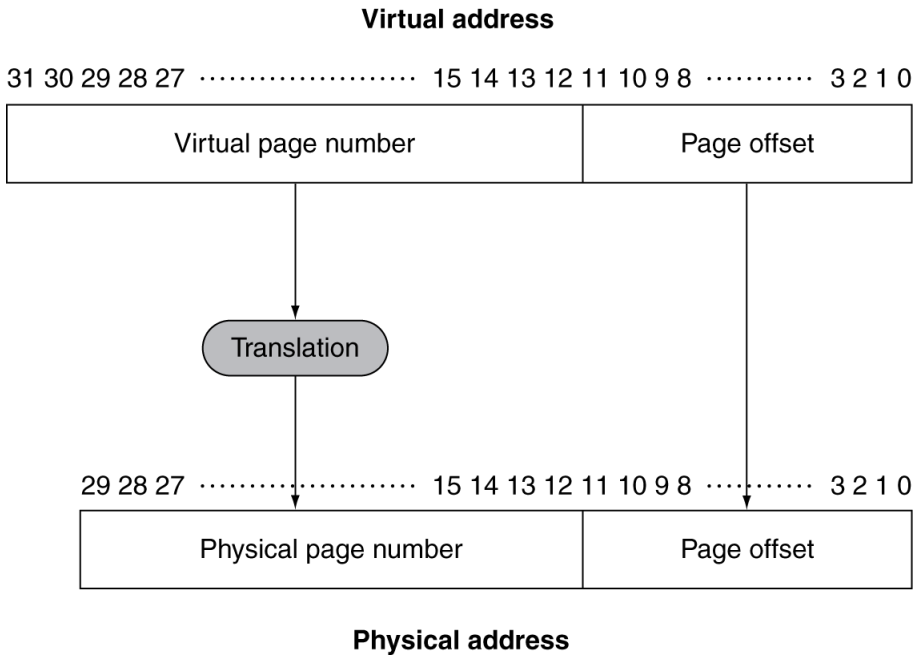
Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM “block” is called a page
  - VM translation “miss” is called a page fault

Address Translation

- Fixed-size pages (e.g., 4K)





Error Detection – Error Correction

- Memory data can get corrupted, due to things like:
  - Voltage spikes.
  - Cosmic rays.
- The goal in **error detection** is to come up with ways to tell if some data has been corrupted or not.
- The goal in **error correction** is to not only detect errors, but also be able to correct them.
- Both error detection and error correction work by attaching additional bits to each memory word.
- Fewer extra bits are needed for error detection, more for error correction.

Parity Bits - Examples

- Size of original word: $m = 8$.

Original Word (8 bits)	Number of 1s in Original Word	Codeword (9 bits): Original Word + Parity Bit
01101101	5	011011011
00110000	2	001100000
11100001	4	111000010
01011110	5	010111101

Some Questions You Should Be Able to Answer

1. How do computers compute?
2. What is a register? Where is it located? How many are there?
3. What is memory? What is a memory address / location?
4. What is the difference between a register and memory?
5. What is a cache? What is the memory hierarchy? Why does it exist?
6. What is translation (compilation)? What is interpretation?
7. How are translation and interpretation different?
8. Why do we use translators and/or interpreters?
9. If a multiply instruction is not available, how can it be created using loops and addition?
10. What is a virtual machine? What is QEMU?
11. What is sequential logic? How is it different than combinational logic?
12. How is a 32-bit processor different from a 64-bit processor?
13. How can you convert C code to assembly? How can you do this for example programs by hand?
14. How is performance evaluated? Why are benchmarks used?
15. How does the stack work? What do push and pop do? Why do we have the stack?
16. What is recursion? What is an iterative program?
17. What are the common addressing modes for ARM and how do you use them?
18. What is RISC vs. CISC? What is a Von Neumann architecture?
19. How can gdb be used to help you understand, write, and debug programs? ***Hint: know how to use this!***
20. What are the ALU, FPU, MMU, TLB, CPU, etc.? What are the main computer components?
21. What is ECC used for? How does parity work for detecting errors?

Summary

- Floating point (IEEE 754)
- Compiler optimizations
- Exam Review