Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 27: Floating Point (IEEE 754), Combining C and Assembly, and ARM Review

Taylor Johnson

Announcements and Outline

- Student Feedback Survey (SFS)
  - Invitation by email *sent on Wednesday, November 19.*
  - *MUST complete BEFORE Wednesday, December 3, 2014*
  - ***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 3 assigned, due tonight by midnight
- Programming assignment 4 assigned, due 12/2 by midnight
- Quiz 5 assigned, due by Monday 12/1 by midnight
- Floating Point
- ARM Architecture and Computer Organization Review

Floating Point

Representing Fractional Numbers

- Seen several ways to encode information using binary numbers
  - Unsigned integers as binary representation
  - Signed integers using two's complement
  - Letters using ASCII
  - Etc.
- How can we represent fractional (non-whole) numbers?
  - Fixed-point
  - Floating-point

Fixed-Point

- Suppose we have 16-bits to represent a fractional number
  - Use upper 8 bits to represent whole (integer) portion
  - Use lower 8 bits to represent fractional (non-whole) portion

Whole Part	Decimal Point (.)	Fractional Part
8 bits	.	8 bits
0010 0000	.	0000 0001
32	.	1/256
32	.	0.00390625

- Number of bits reserved for fractional part determines significance of each fractional part
- Here, we have 8 bits, so each fractional part is 1/256, since 2^8 = 256

Why Not Fixed-Point?

- Hard to represent very larger or very small numbers
- Smallest number representable using 64 bits, supposing we keep 32 bits for whole part and 32 bits for fractional part, is:

$$1/(2^{32}) = 0.00000000023283064365386962890625\ldots$$

- Largest number is still 2^32
- What if we need to represent larger or small numbers?
  - Utilize idea of significant digits
  - If a number is very large, a small deviation results in a small error
  - If a number if very small, a small deviation may result in a large error
  - Utilize relative (percentage) error as opposed to absolute error

Floating Point

- System for representing number where the range of expressible numbers if *independent* of the number of significant digits
- Represent number n in scientific notation:

$$n = f * 10^e$$

  - n: number being represented
  - f: fraction (mantissa)
  - e: positive or negative integer
- Examples
  - 3.14 = 0.314 * 10^1 = 3.14 * 10^0
  - 0.000001 = 0.1 * 10^-5 = 1.0 * 10^-6
  - 1941 = 0.1941 * 10^4 = 1.941 * 10^3

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

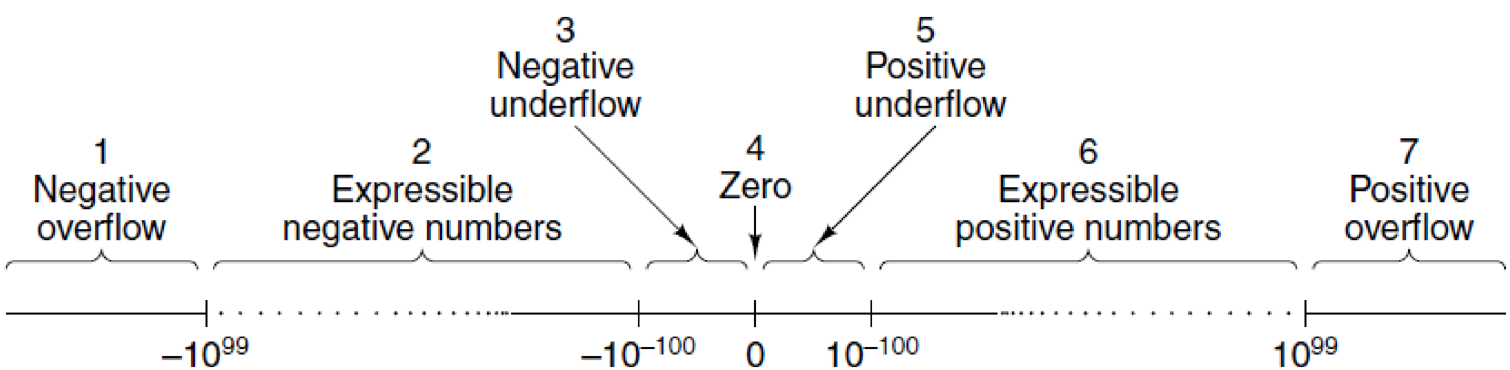
normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Real Number Line Regions

- Divided real number line into seven regions:
  - Large negative numbers less than -0.999×10^{99}
  - Negative between -0.999×10^{99} and -0.100×10^{-99}
  - Small negative, magnitudes less than 0.100×10^{-99}
  - Zero
  - Small positive, magnitudes less than 0.100×10^{-99}
  - Positive between 0.100×10^{-99} and 0.999×10^{99}
  - Large positive numbers greater than 0.999×10^{99}



Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE 754 Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: 1.0 ≤ |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

Expressible Numbers

- Approximate lower and upper bounds of expressible (unnormalized) floating-point decimal numbers

Digits in fraction	Digits in exponent	Lower bound	Upper bound
3	1	10^{-12}	10^{9}
3	2	10^{-102}	10^{99}
3	3	10^{-1002}	10^{999}
3	4	10^{-10002}	10^{9999}
4	1	10^{-13}	10^{9}
4	2	10^{-103}	10^{99}
4	3	10^{-1003}	10^{999}
4	4	10^{-10003}	10^{9999}
5	1	10^{-14}	10^{9}
5	2	10^{-104}	10^{99}
5	3	10^{-1004}	10^{999}
5	4	10^{-10004}	10^{9999}
10	3	10^{-1009}	10^{999}
20	3	10^{-1019}	10^{999}

Normalization

- Problem: many equivalent representation of same number using the exponent/fraction notation
- Example:
  - 0.5: exponent = -1, fraction = 5: $10^{-1} * 5 = 0.5$
  - 0.5: exponent = -2, fraction = 50: $10^{-2} * 50 = 0.5$
- Binary normalization
  - If leftmost bit is zero, shift all fractional bits left by one and decrease exponent by 1 (assuming no underflow)
  - Fraction with leftmost nonzero bit is normalized
- Benefit: only one normalized representation
  - Simplifies equality comparisons, etc.

Normalization in Binary

Example 1: Exponentiation to the base 2

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}

Unnormalized: $\underbrace{0}_{\text{Sign } +}\ \underbrace{1010100}_{\substack{\text{Excess 64} \\ \text{exponent is} \\ 84-64=20}}\ .\ \underbrace{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1}_{\substack{\text{Fraction is } 1\times 2^{-12} + 1\times 2^{-13} \\ +1\times 2^{-15} + 1\times 2^{-16}}}$ $= 2^{20}(1\times 2^{-12} + 1\times 2^{-13} + 1\times 2^{-15} + 1\times 2^{-16}) = 432$

To normalize, shift the fraction left 11 bits and subtract 11 from the exponent.

Normalized: $\underbrace{0}_{\text{Sign } +}\ \underbrace{1001001}_{\substack{\text{Excess 64} \\ \text{exponent is} \\ 73-64=9}}\ .\ \underbrace{1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0}_{\substack{\text{Fraction is } 1\times 2^{-1} + 1\times 2^{-2} \\ +1\times 2^{-4} + 1\times 2^{-5}}}$ $= 2^{9}(1\times 2^{-1} + 1\times 2^{-2} + 1\times 2^{-4} + 1\times 2^{-5}) = 432$

Normalization in Hex

Example 2: Exponentiation to the base 16

Unnormalized: $\underbrace{0}_{\substack{\text{Sign}\\ +}}\ \underbrace{1000101}_{\substack{\text{Excess 64}\\ \text{exponent is}\\ 69-64=5}}\ .\ \underbrace{\overset{16^{-1}}{0000}\ \overset{16^{-2}}{0000}\ \overset{16^{-3}}{0001}\ \overset{16^{-4}}{1011}}_{\text{Fraction is } 1\times 16^{-3} + B\times 16^{-4}}$ $= 16^5\,(1\times 16^{-3} + B\times 16^{-4}) = 432$

To normalize, shift the fraction left 2 hexadecimal digits, and subtract 2 from the exponent.

Normalized: $\underbrace{0}_{\substack{\text{Sign}\\ +}}\ \underbrace{1000011}_{\substack{\text{Excess 64}\\ \text{exponent is}\\ 67-64=3}}\ .\ \underbrace{0001\ 1011\ 0000\ 0000}_{\text{Fraction is } 1\times 16^{-1} + B\times 16^{-2}}$ $= 16^3\,(1\times 16^{-1} + B\times 16^{-2}) = 432$

IEEE Floating-Point Types

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits, total	32	64
Exponent system	Excess 127	Excess 1023
Exponent range	−126 to +127	−1022 to +1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

IEEE Numerical Types

Normalized	±	0 < Exp < Max	Any bit pattern
Denormalized	±	0	Any nonzero bit pattern
Zero	±	0	0
Infinity	±	1 1 1…1	0
Not a number	±	1 1 1…1	Any nonzero bit pattern

Sign bit

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 ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

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: 1011111101000…00
- Double: 1011111111101000…00

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

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

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

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

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

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

  $= -5.0$

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
```

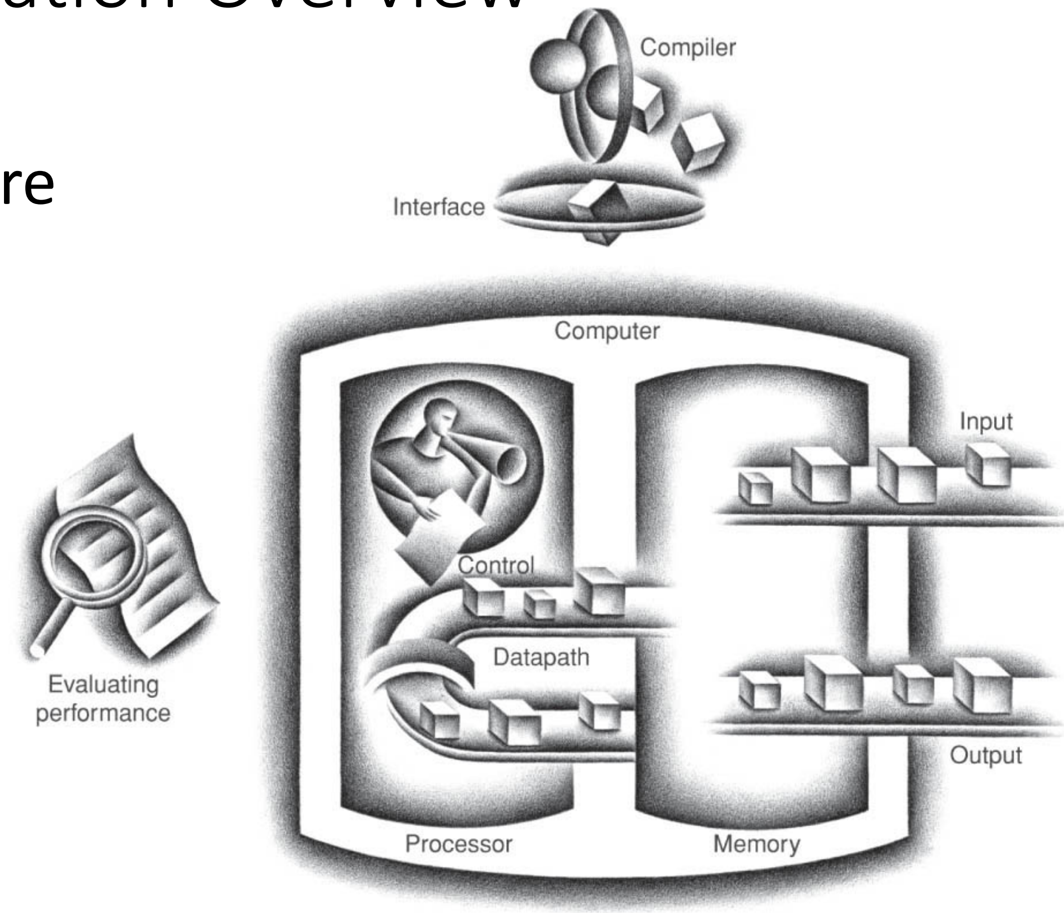
Compiler Optimization Summary

- 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?

ARM Architecture and Computer Organization Review

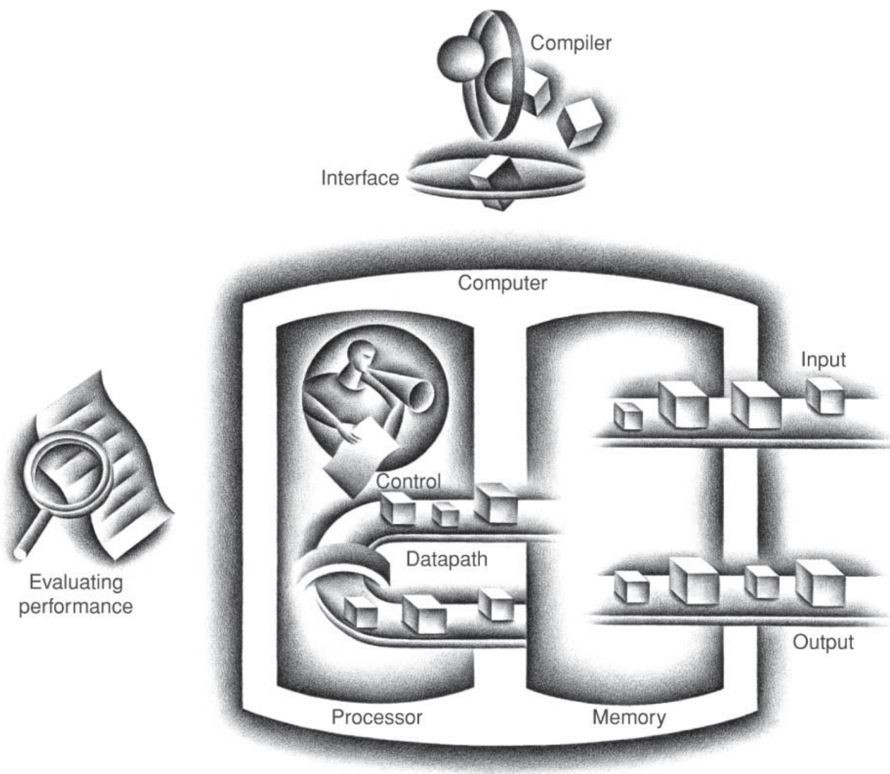
Computer Organization Overview

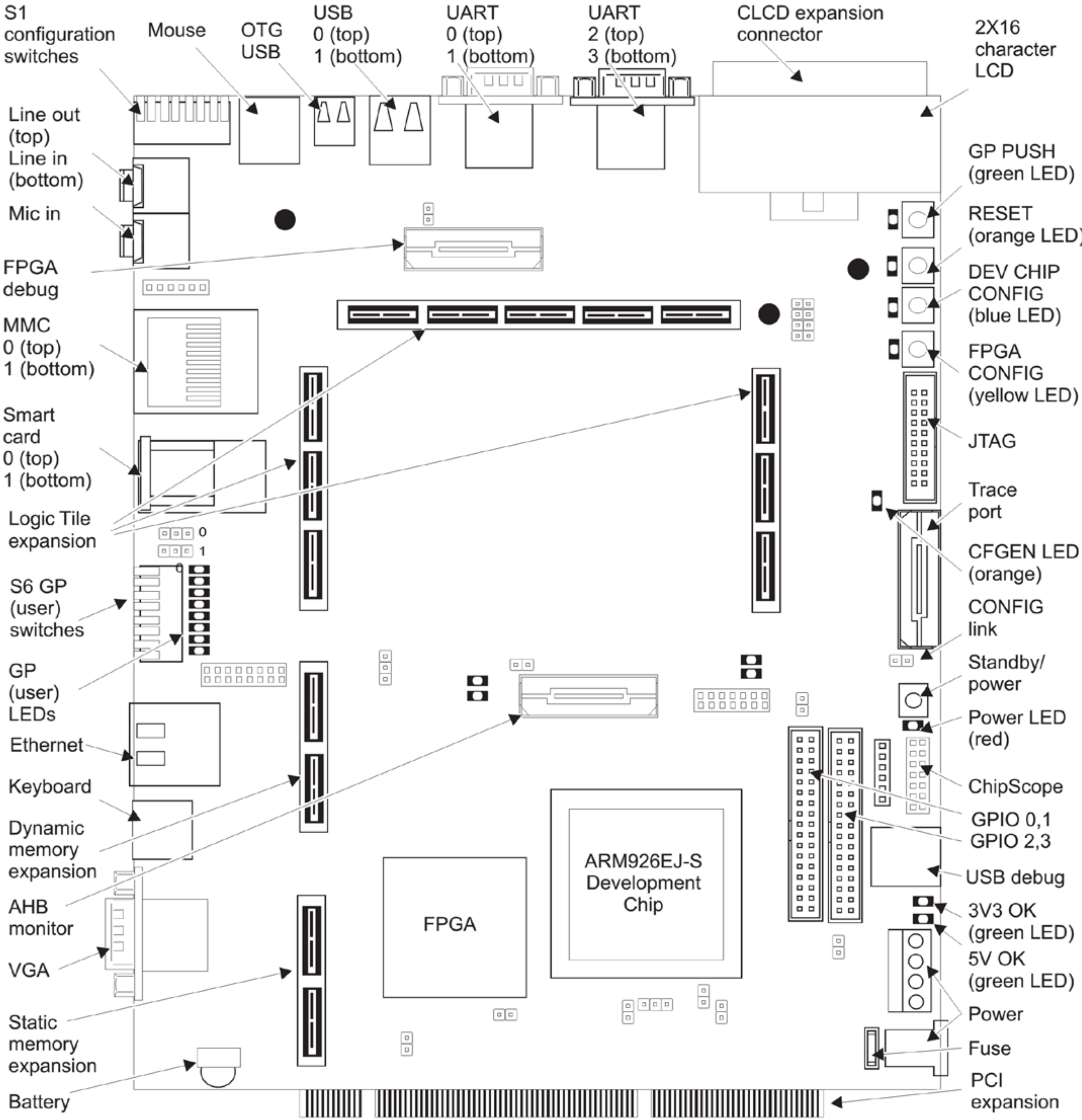
- ISA: hardware-software interface
- CPU
  - Executes instructions
- Memory
  - Stores programs and data
- Buses
  - Transfers data
- I/O devices
  - Input: keypad, mouse, touch, …
  - Output: printer, screen, …



What Computer Have We Used this Semester?

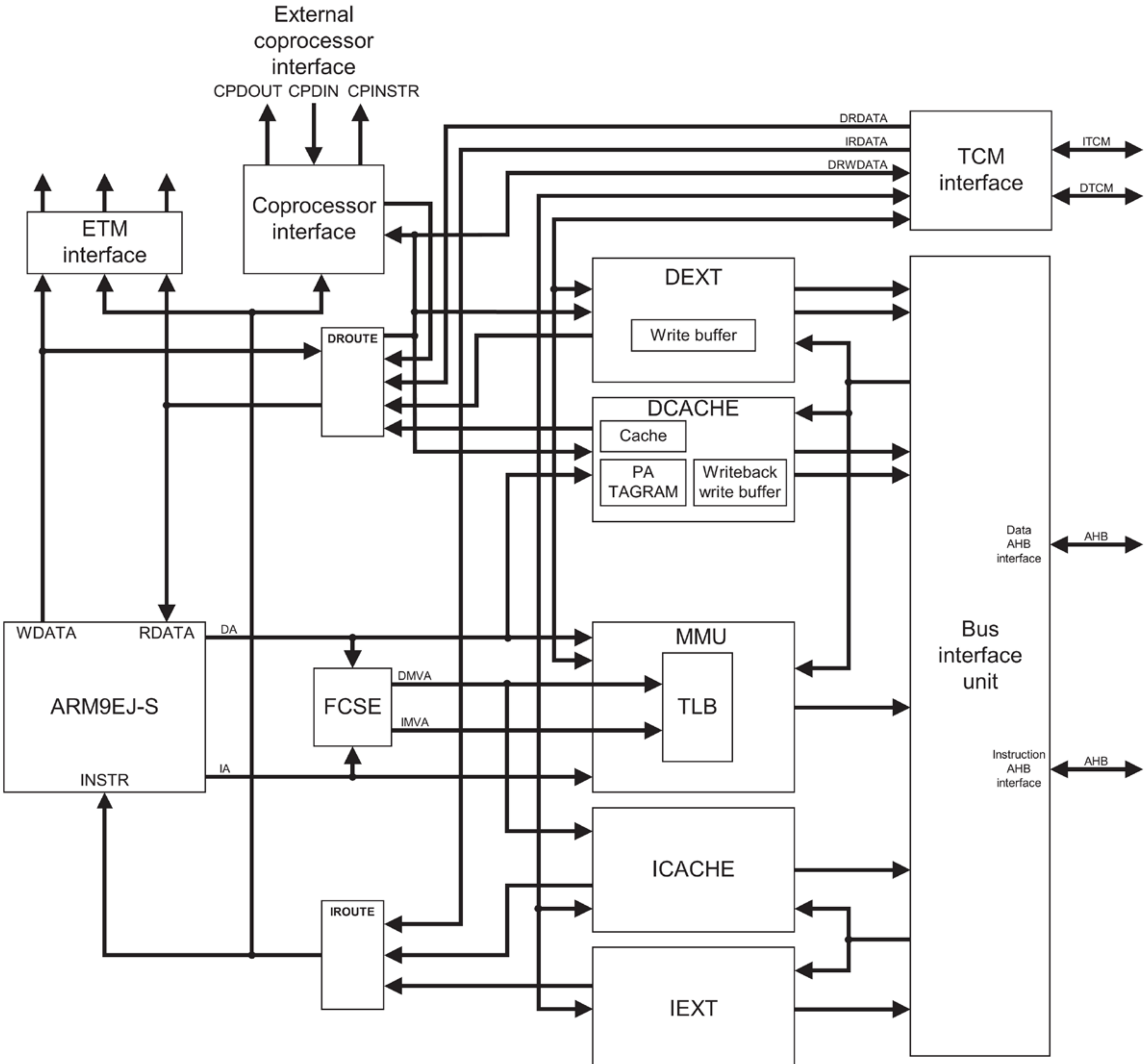
- ARM Versatilepb computer
- Full computer!
  - Input
  - Output
  - Processor
  - Memory
  - Programs





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

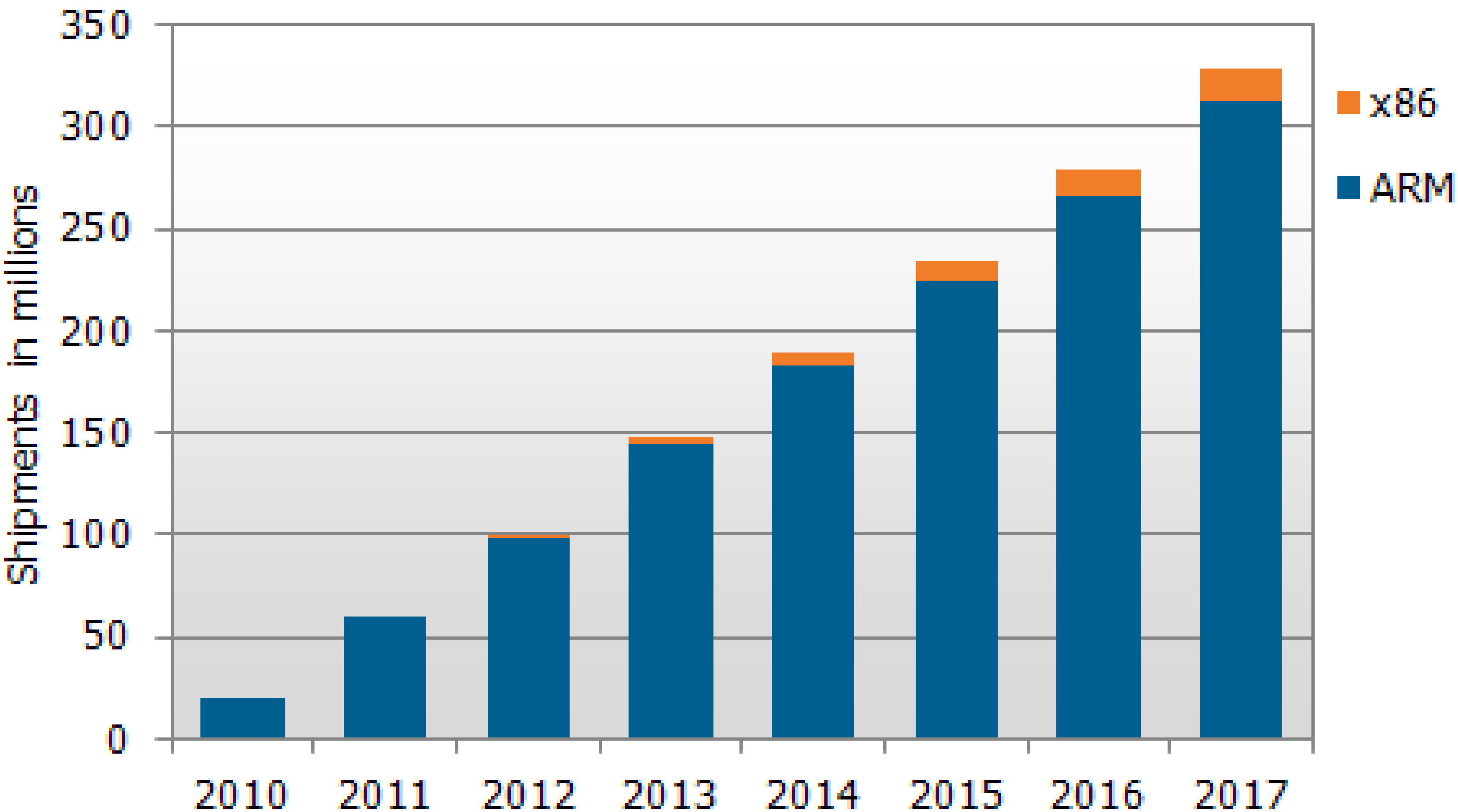
[http://infocenter.arm.com/help/topic/com.arm.doc.dui0224i/DUI0224I_realview_platform_baseboard_for_arm926ej_s_ug.pdf]



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

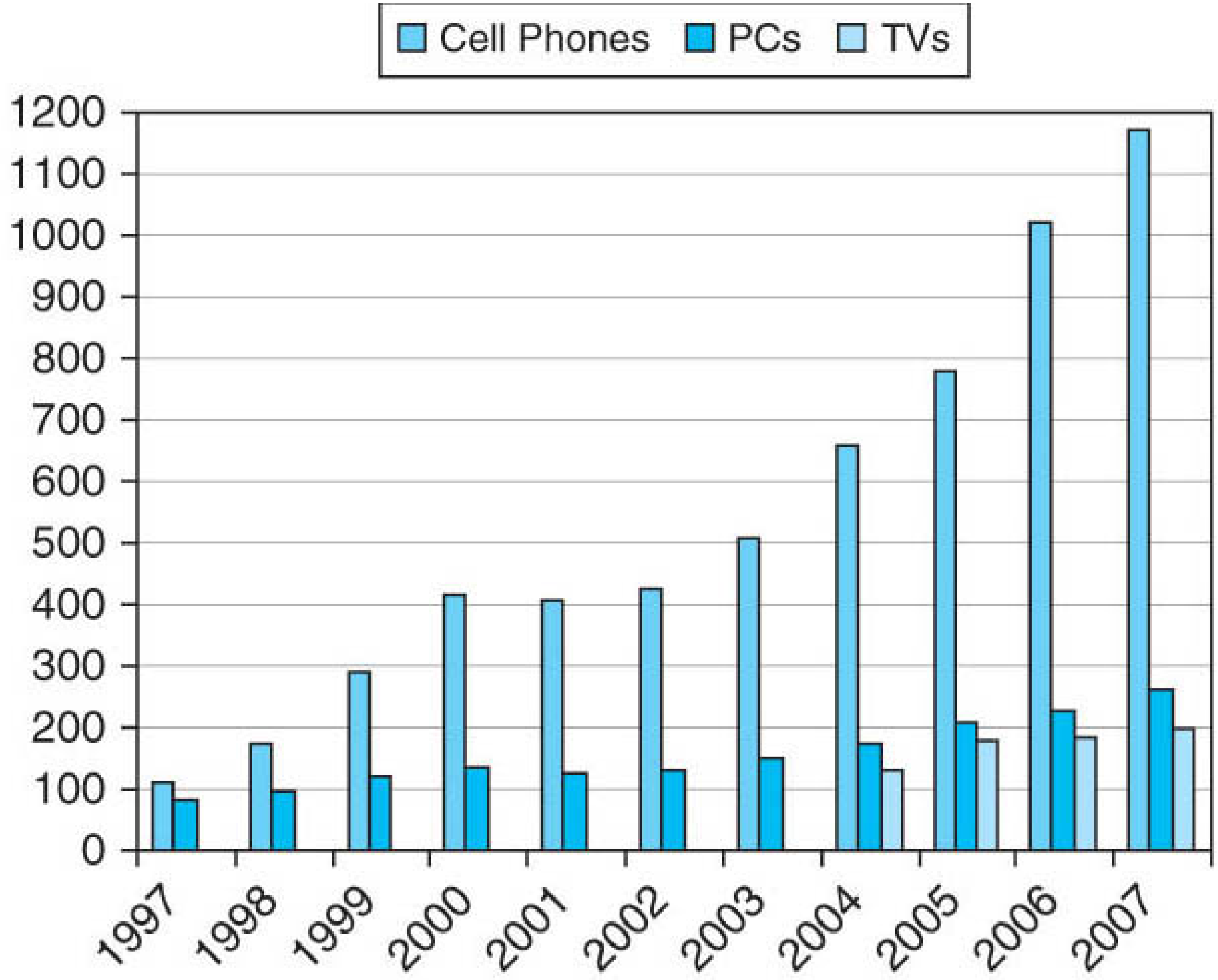
[http://www.atmel.com/Images/arm_926ejs_trm.pdf]

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

Course Objective Overview

- Seen how computers really ***compute***
- Processor/memory organization: execution cycle, registers, memory accesses
- Processor operation: pipeline
- Computer organization: memory, buses, I/O devices
- Assembly language programming: various architecture styles (stack-based), register-to-register (ARM), etc.
- Saw more representations of data (floating point, integers)

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

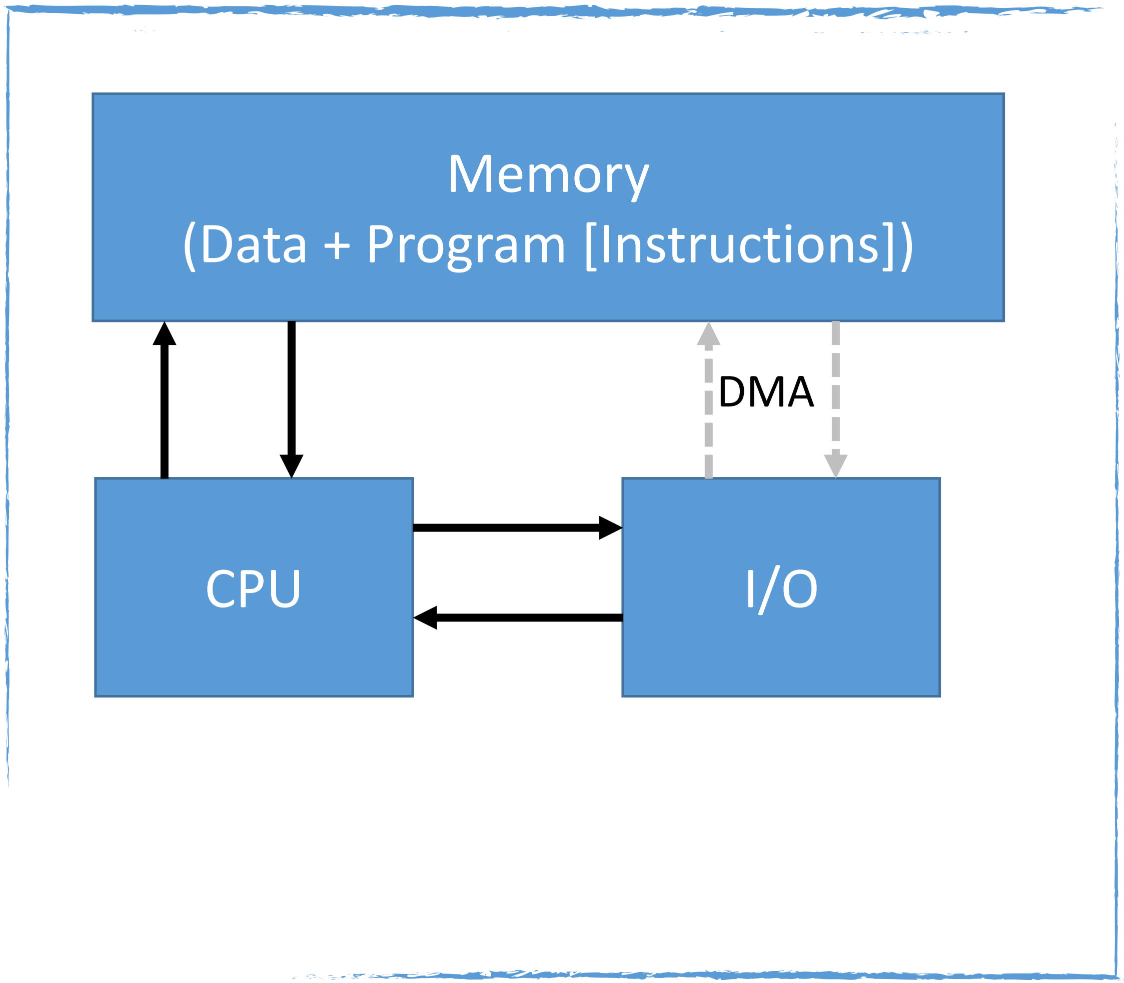
Multilevel Architectures

Level 4	Operating System Level	C / …
Level 3	Instruction Set Architecture (ISA) Level	Assembly / Machine Language
Level 2	Microarchitecture Level	n/a / Microcode
Level 1	Digital Logic Level	VHDL / Verilog
Level 0	Physical Device Level (Electronics)	n/a / Physics

Processor (CPU) Components

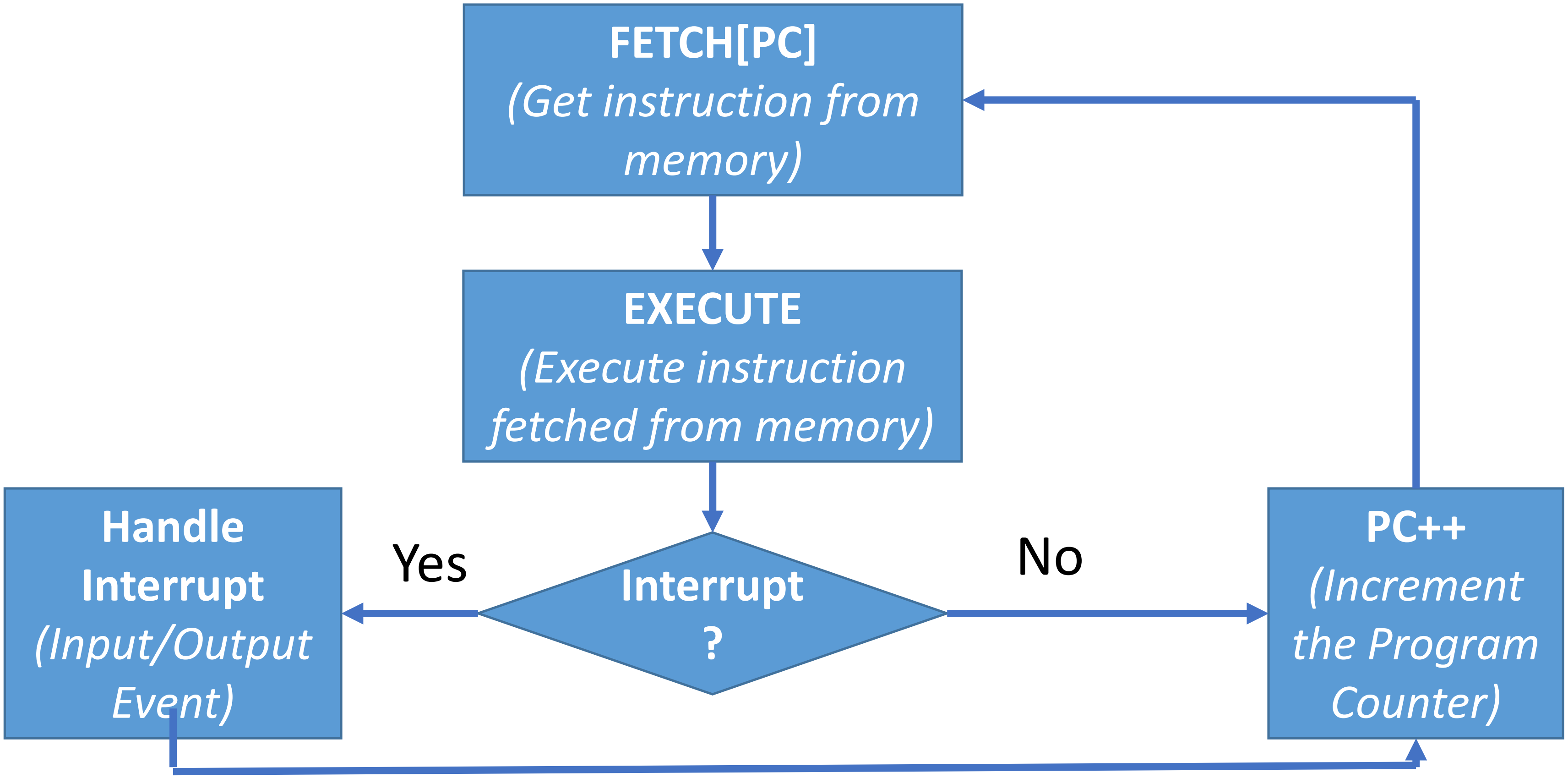
- Pipeline: stages (fetch, decode, execute)
- ALU: arithmetic logic unit
- MMU: memory management unit
  - TLB: translation lookaside buffer (cache for virtual memory)
- Cache (L1, L2, L3, …)
  - Caches for main memory
- Registers
  - Hold values for all ongoing computations (i.e., only can do computation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

Von Neumann Architecture

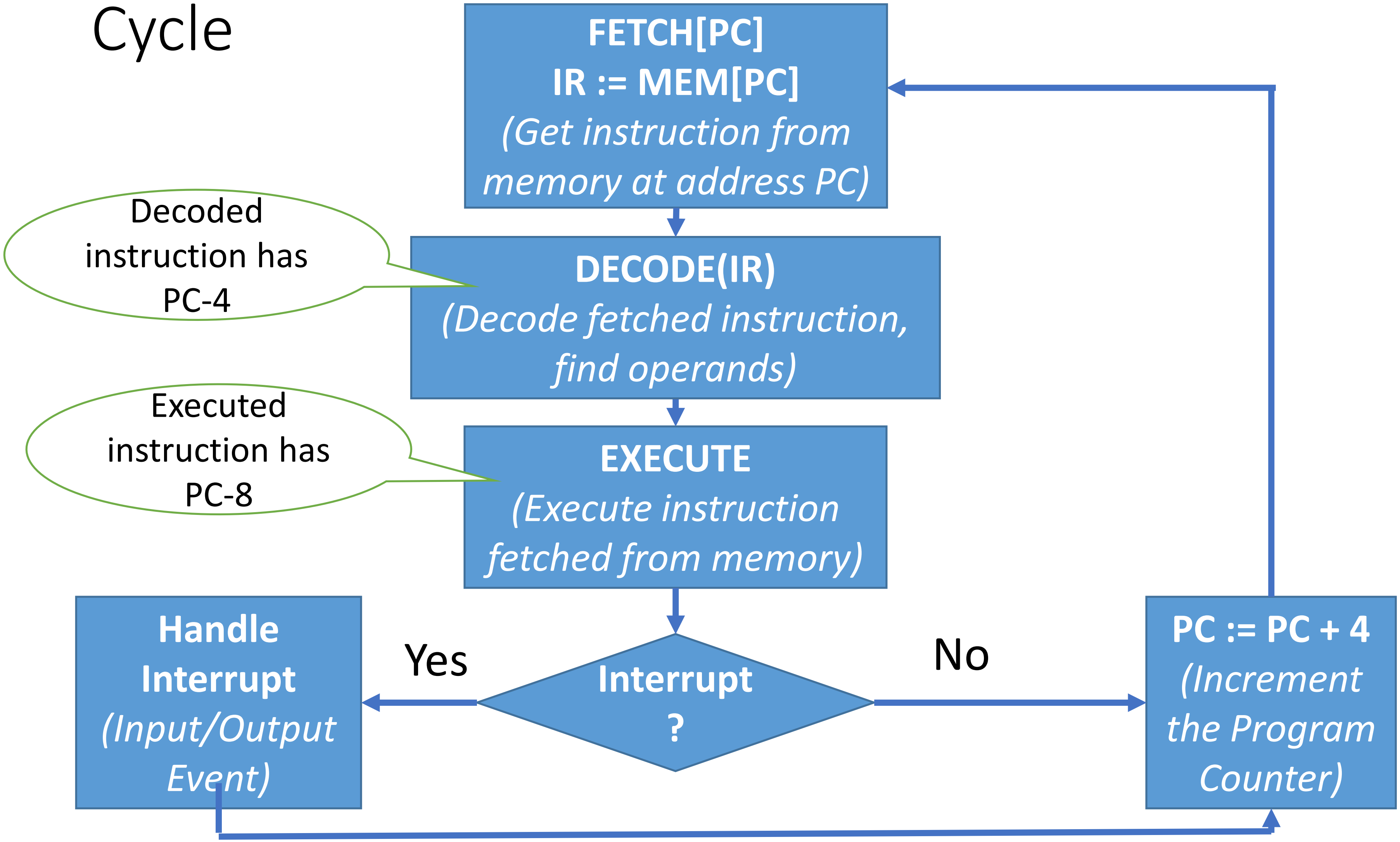


- Both data and program stored in memory
- Allows the computer to be “re-programmed”
- Input/output (I/O) goes through CPU
- I/O part is not representative of modern systems (direct memory access [DMA])
- Memory layout is representative of modern systems

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)

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

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?

Some Questions You Should Be Able to Answer

1. What is a register? Where is it located? How many are there?
2. What is memory? What is a memory address / location?
3. What is the difference between a register and memory?
4. What is translation (compilation)? What is interpretation?
5. How are translation and interpretation different?
6. Why do we use translators and/or interpreters?
7. If a multiply instruction is not available, how can it be created using loops and addition?
8. What is a virtual machine?
9. What is sequential logic? How is it different than combinational logic?
10. How is a 32-bit processor different from a 64-bit processor?

Summary

- Floating point (IEEE 754)
- Compiler optimizations
- More Exam Review Next Time