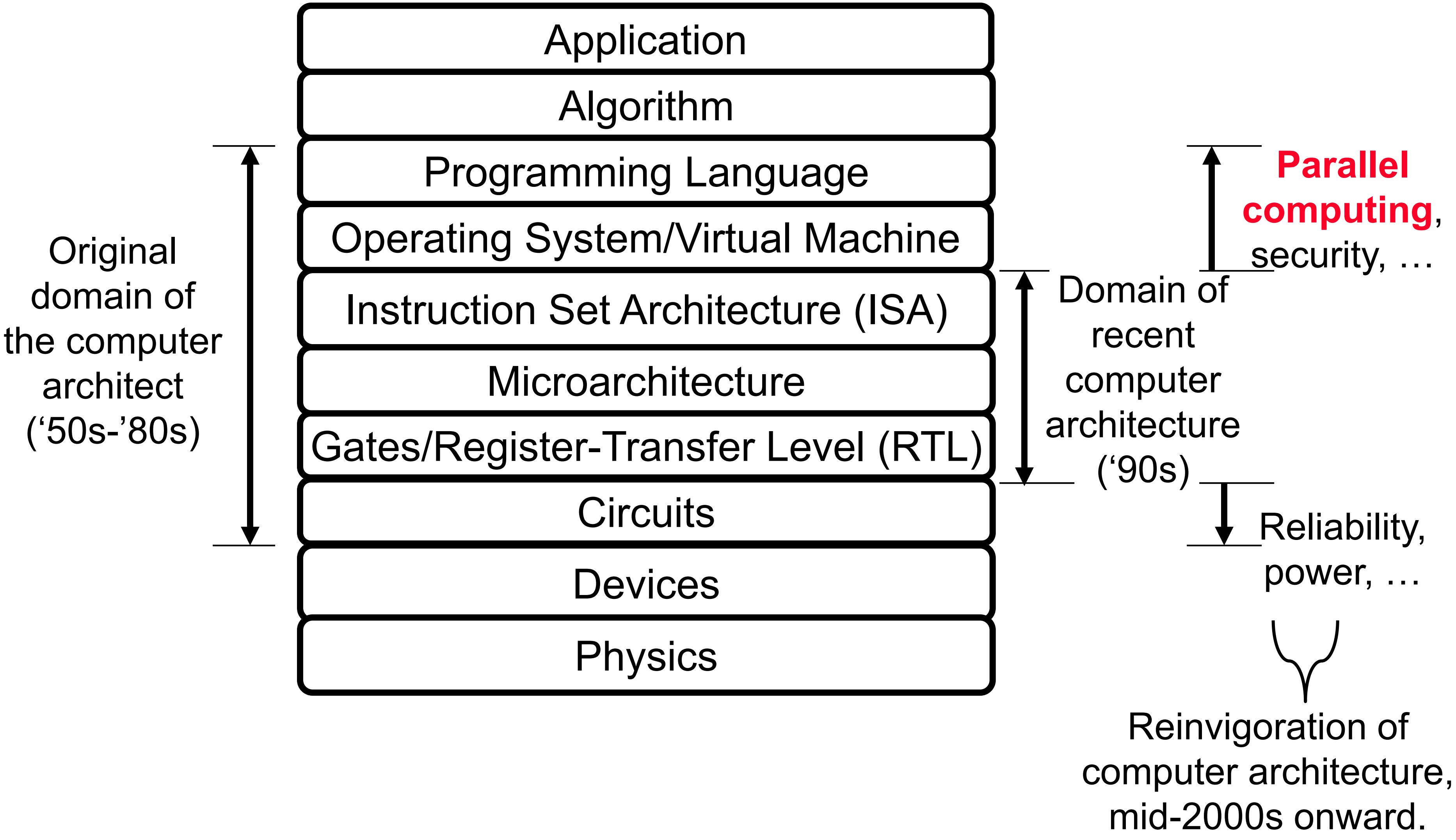
CPT_S 260 Intro to Computer Architecture
Lecture 18

Exam 1 Review
February 23, 2021

Ganapati Bhat
School of Electrical Engineering and Computer Science
Washington State University

Abstraction Layers in Modern Systems



Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- **Instruction Count for a program**
  - Determined by program, ISA and compiler
- **Average cycles per instruction**
  - Determined by CPU hardware
  - If different instructions have different CPI
    - » Average CPI affected by instruction mix

CPI in More Detail

- **If different instruction classes take different numbers of cycles**

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- **Weighted average CPI**

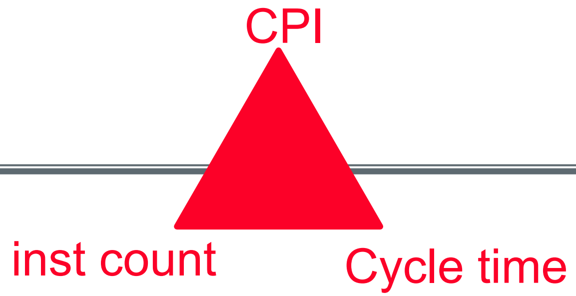
$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Performance Summary

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- **Performance depends on**
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Computer Performance



$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization		X	X
Technology			X

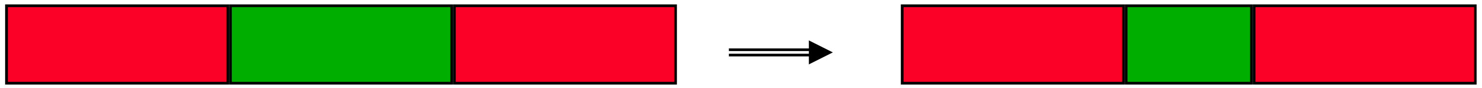
Amdahl’s Law

- **How do we increase performance?**
  - Utilize parallelism
  - Principle of locality
  - Focus on the common case
- **Amdahl’s law provides a method to quantify speedup**

$$Speedup_{overall} = \frac{t_{old}}{t_{new}} = \frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{speedup_{enhanced}}}$$

- **Best achievable speedup is**

$$Speedup_{maximum} = \frac{1}{1 - fraction_{enhanced}}$$



Numbering Systems

- A number system of a specific base (radix) uses numbers from 0 to that base-1
- Numbers can be computed to decimal through the sum of the weighted digits-

$$\boldsymbol{Number} = \sum_{i=0}^{n} base^{i} * digit$$

	Binary	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Symbols	{0,1}	{0,1,2,3,4,5,6,7}	{0,1,2,3,4,5,6,7,8,9}	{0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F}

Binary → Decimal

Compose a series of base-2 terms from a binary number by identifying the position of every '1' and expressing 1's as $2^{position}$. Read binary numbers from right to left and index the rightmost position as zero.

<u>Example</u>: Convert 11001001_2 to decimal

(1	1	0	0	1	0	0	$1)_2$
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$$

$$128 + 64 + 8 + 1$$

$$(201)_{10}$$

Binary ↔ Hexadecimal

Partition binary digits right-to-left in groups of four.
Convert each group to one hexadecimal symbol.

Example: Convert 1000101010_2 to hexadecimal

(0010	0010	$1010)_2$
2	2	A

$$\mathbf{2*16^2 + 2*16^1 + A*16^0}$$

$$\mathbf{(22A)_{16}}$$

Hexadecimal number	Binary-coded hexadecimal
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

2’s-Complement Signed Integers

- Given an n-bit number:

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits, 2’s complement range is:
  - –2,147,483,648 to +2,147,483,647

Signed Negation in 2's Complement

- **Complement and add 1**
  - Complement means 1 → 0, 0 → 1

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

Overflow

- **The addition of two numbers with the same sign or subtraction of two numbers with different sign may cause overflow**
- **The condition wherein a result cannot be represented in allocated memory**
- **An overflow condition can be detected by observing the carry into the sign bit and carry out of the sign bit**

carries:	0	1	carries:	1	0
+70	0	1000110	−70	1	0111010
+80	0	1010000	−80	1	0110000
+150	1	0010110	−150	0	1101010

Conversion for Numbers with Fractions

- **In real mathematical operation, we have numbers with fractions**
  - Float and Double numbers in programing languages
- **We should take three steps:**
  - Convert the Integer Part (The same as integer numbers)
  - Convert the Fraction Part
  - Join the two results with a radix point

Fractional Part in Binary Format

- Repeatedly multiply the fraction by 2 and save the resulting integer digits. The digits for the binary number are the 0,1 in order of their computation.

Normalized Numbers

- **A number in scientific notation that has no leading 0s is called a *normalized number***
- **Example:**
  - $1.0_{ten} \times 10^{-9}$ is in **normalized** scientific notation,
  - $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ **are not**
- **Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation**

$$\mathbf{1.0_2 \times 2^{-1}}$$

Binary point

IEEE 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 ⇒ non-negative, 1 ⇒ 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

Arithmetic Operations

- **Add and subtract, three operands**
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

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

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

- We see what are these variables!

Register Operands

- **Arithmetic instructions use register operands**
- **MIPS has a 32 × 32-bit register file**
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- **Assembler names**
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- ***Design Principle 2:* Smaller is faster**
  - c.f. main memory: millions of locations

Register Operand Example

- **C code:**

```
f = (g + h) - (i + j);
```

For instance, f, …, j stored in $s0, …, $s4

- **Compiled MIPS code:**

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operand Example #1

- **C code:**

```
g = h + A[8];
```

  – Assume that g is in \$s1, h in \$s2, base address of A in \$s3

- **Compiled MIPS code:**
  – Index 8 requires offset of 32
    » 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

Memory Operand Example #2

- **C code:**

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3

- **Compiled MIPS code:**
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Conditional Operations

- **Branch to a labeled instruction if a condition is true**
  - Otherwise, continue sequentially
- **`beq rs, rt, L1`**
  - if (rs == rt) branch to instruction labeled L1;
- **`bne rs, rt, L1`**
  - if (rs != rt) branch to instruction labeled L1;
- **`j L1`**
  - unconditional jump to instruction labeled L1

Compiling If Statements

- **C code:**

```
if (i==j) f = g+h;
else f = g-h;
```

  – f, g, … in $s0, $s1, …

- **Compiled MIPS code:**

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

Assembler calculates addresses