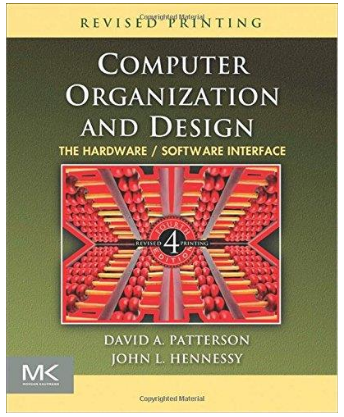
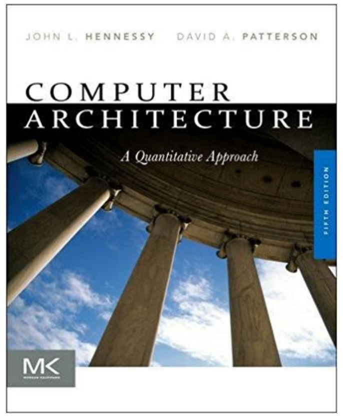
Computer Architecture
PhD Qualifying Exam
Review Session

November, 2017

Quincy FLINT

Resources


REVISED PRINTING
COMPUTER ORGANIZATION AND DESIGN
THE HARDWARE / SOFTWARE INTERFACE
DAVID A. PATTERSON
JOHN L. HENNESSY
MK



- Textbooks
  - Computer Organization and Design – Patterson, Hennessy
  - Computer Architecture – A Quantitative Approach – Patterson, Hennessy

- UF Past Practice Exams
  - https://www.ece.ufl.edu/content/phd-written-qualifying-exam-questions

- UF Exam Study Guide
  - https://www.ece.ufl.edu/sites/default/files/pictures/ComputerOrganization.pdf

- Exam Registration
  - https://gradadmissions.ece.ufl.edu/srs-servlet/examRegistration/phd

Important Dates

- **12/01/2017** – Qualifying exam registration closes (Friday, 5 pm)
- **12/??/2017** – Potential 2nd review session
- **01/??/2018** – Final review session
- **01/20/2018** – Qualifying exam (Saturday, time TBD)

Boolean Algebra

- **Algebra with 0’s and 1’s**
  - X + 0 = X
  - X + 1 = 1
  - X * 1 = X
  - X * 0 = 0
- **Idempotent Laws**
  - X + X = X
  - X * X = X
- **Complement Laws**
  - X + /X = 1
  - X * /X = 0

Boolean Algebra

- **Dual**:
  - 1 → 0
  - + → ×

  X + 0 = X → X × 1 = X

  X + 1 = 1 → X * 0 = 0

  X + /X = 1 → X * /X = 0

- **DeMorgan's Laws**
  - NOT(X+Y+Z) = NOT(X) * NOT(Y) * NOT(Z)
  - NOT(X*Y*Z) = NOT(X) + NOT(Y) + NOT(Z)

- **Consensus Theorem**
  - XY + **YZ** + \XZ = XY + \XZ

  Proof - example

Number Systems

- **Base 10**: 541 $= 5 \times 10^2 + 4 \times 10^1 + 1 \times 10^0$
  $= 5 \times 100 + 4 \times 10 + 1 \times 1$

- **Base 2:** 0101 $= 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  $= 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$

Conversion between systems

- **Decimal to Binary**
  1. Brute Force -- “Count it out”
  2. Divide by 2 – remainder becomes binary (least to most significant)

- **Decimal to Hex**
  1. Convert to Binary then groups of 4 bits
  2. Divide by 16 -- remainder becomes hex (least to most significant)

- **Decimal to Octal**
  1. Convert to Binary then groups of 3 bits
  2. Divide by 8 -- remainder becomes octal (least to most significant)

Boolean Arithmetic

- Think back to basic arithmetic in base 10

- Let’s just do some problems

Signed Number Representations

- **Signed Magnitude**:
  - MSB gives sign
  - 1000 0101 = -5
- **1’s Complement**:
  - if MSB is 1 – flip bits and apply minus sign
  - If MSB is 0 – do nothing, positive
  - 1111 1100 (flipped = 0000 0011) = -3
- **2’s Complement**:
  - if MSB is 1 – flip bits, add 1, and apply minus sign
  - 1111 1101 (flipped + 1 = 0000 0011) = -3

Alternate 2’s Complement Solution

- 2’s Complement:
  - 1011

1. Flip Bits: 0100
2. Add 1: 0101
3. Interpret: -5

- 2’s Complement:
  - 1011

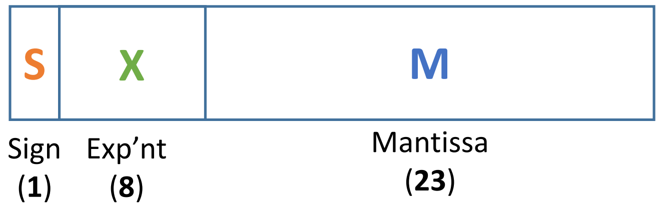
$$\begin{array}{cccccc} | & & | & | & & \\ -8 & + & 2 + 1 & & & = -5 \end{array}$$

Floating Point Numbers

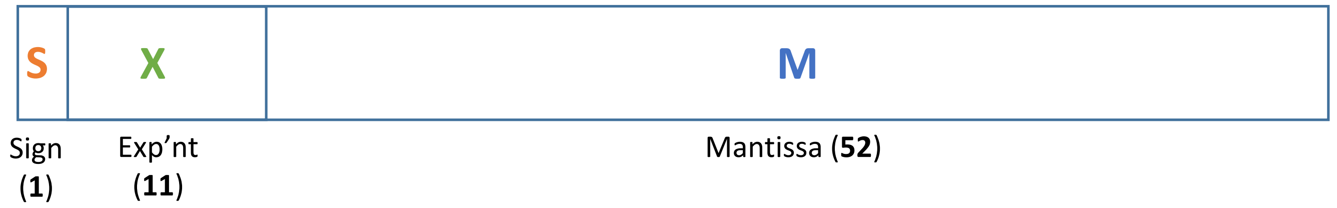
- Single Precision
  - [31][30 … 23][22 … 0]
  - Bias (B) = $2^{8-1} - 1 =$ **127**
- Double Precision
  - [63][62 … 52][51 … 0]
  - Bias (B) = $2^{11-1} - 1 =$ **1028**
- **Ex:** 0xC0C0 0000 to decimal
- **Q**: Precision and Range?

$$\textbf{FP Number} = (-1)^{S} * (1 + M) * 2^{X-B}$$

Single Precision Floating Point



Double Precision Floating Point



Floating Point Arithmetic

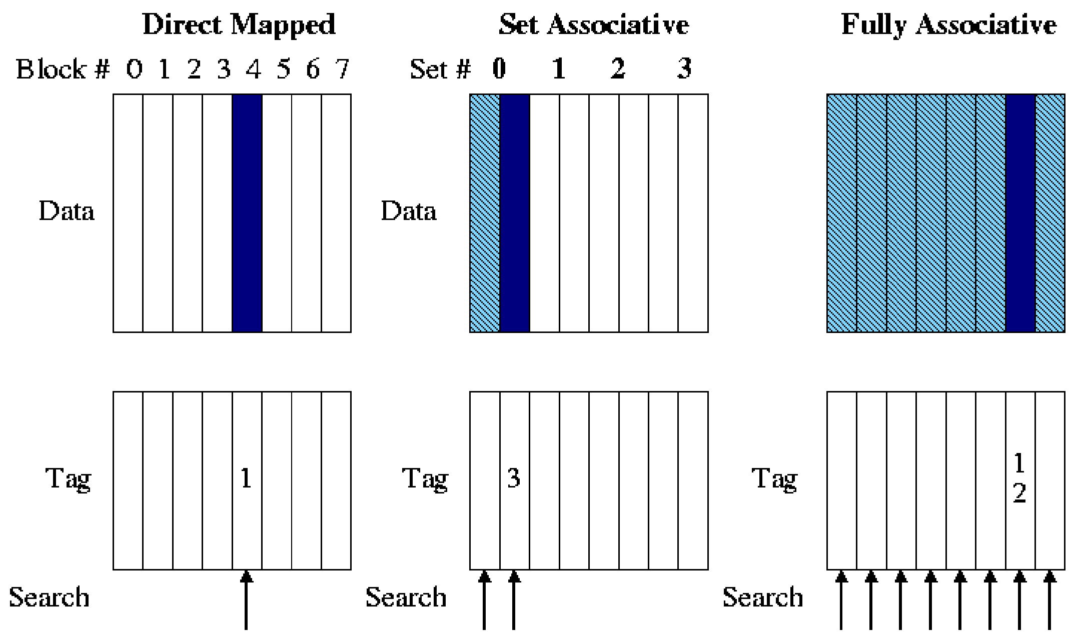
- Steps to perform FP *addition*
  1. Align radix
     - Calculate difference in exponent D = $X_> - X_<$
     - Choose same exponent, $X_< = X_>$
     - Align mantissa, shift "hidden bit" into $M_<$ by D [*de-normalize*]
  2. Perform operation
     - Keep same exponent
     - Add mantissa fields
  3. Re-Normalize

- **Example**: *add* 3E80 000 to 42C8 0000

Cache Basics

- Associativity:
  - 1-way [Direct Mapped]
  - N-way Set-Associative
  - All-the-way [Fully Associative]

- Finding a block in cache:
- Cache index = Ram Block Address mod Number Sets in Cache



Turquoise = entire set
Blue = selected block
Arrow = block placement

Direct Mapped Cache

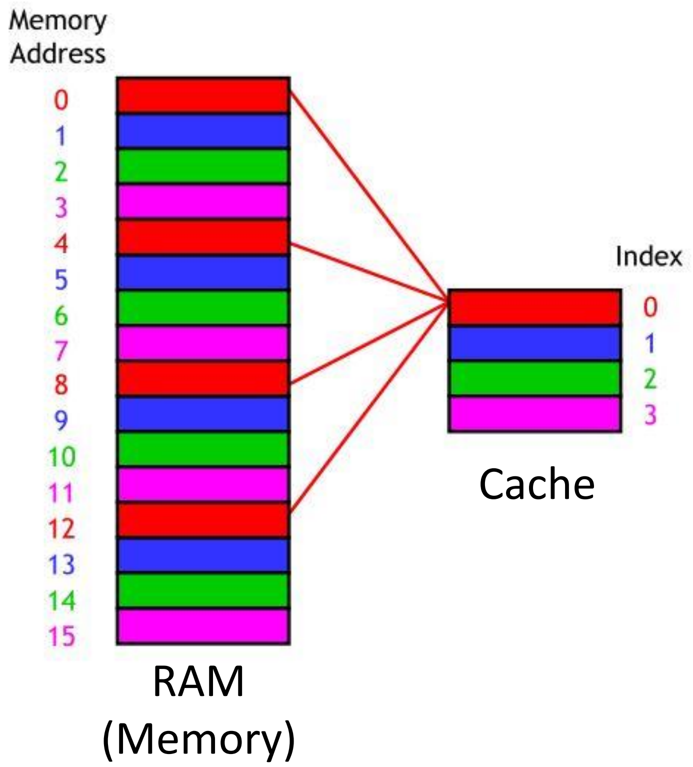
- RAM Size = M (Bytes)
- # Bits to address RAM = m = M log 2
- # RAM addresses = 2^m

- Cache Size = K (Bytes)
- # Bits to address cache = k = K log 2
- # Cache addresses = 2^k

- (RAM) Block Size = N (Blocks)
- # Bits to address blocks in cache = n = N log 2
- # Memory block in cache line = 2^n

- # (RAM) Blocks = $2^m/2^n = 2^{m-n}$

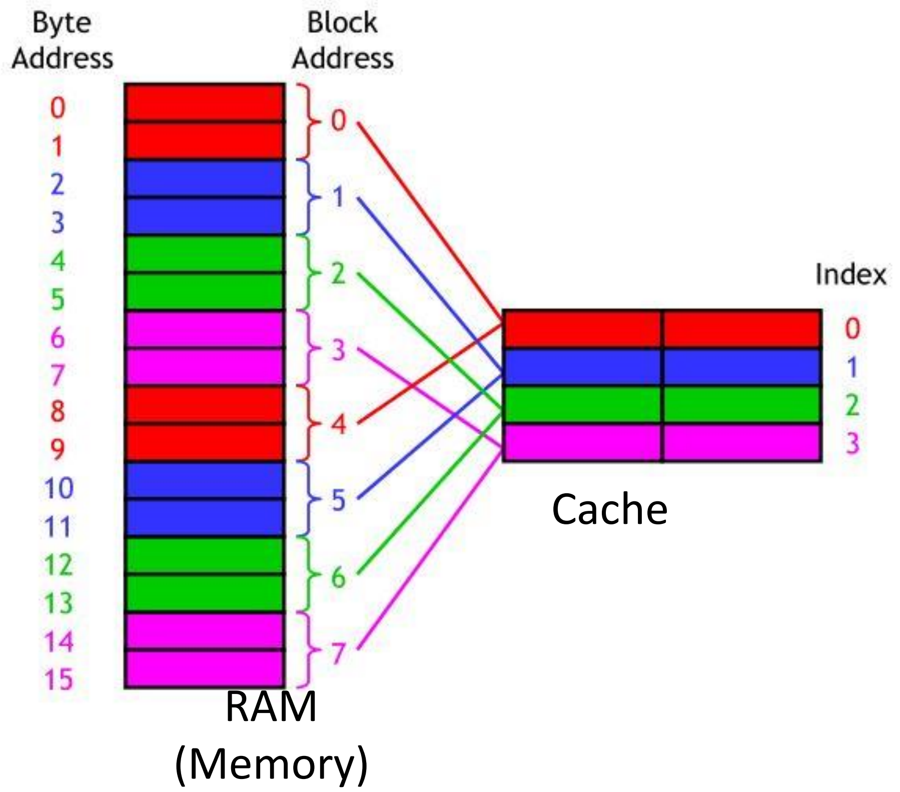
Example: 16-Byte RAM, 4-Byte Cache (Direct Mapped), 1-Byte Memory Blocks



Direct Mapped Cache + 2-Byte Blocks

- RAM Size = N = 16 (Bytes)
- # Bits to address RAM = n = 4
- Cache Size = M = 4 (Bytes)
- # Bits to address cache = m = 2
- (RAM) Block Size = K = 2 (blocks)
- # Bits to address blocks in cache = n = 1
- # (RAM) Blocks = $2^{4-1} = 2^3 = 8$

Example: 16-Byte RAM, 4-Byte Cache (Direct Mapped), 2-Byte Memory Blocks



M-Bit Address

- Offset = rem(block number / block size)
- Index = (memory set number) mod (cache size)
  = lowest k bits of block set address
- Tag = most significant bits of block set address not used by index

m-k-n bits	k bits	n bits
Tag (MSB of Block Address)	Cache Set Index	Block Offset

Virtual Memory

- VM solves 3 problems:
  - Not enough physical RAM
  - Data Fragmentation
  - Programs overwriting memory

- Virtual Memory…
  - Uses the hard drive like another layer of memory abstraction
  - Maps virtual addresses to physical addresses (*)

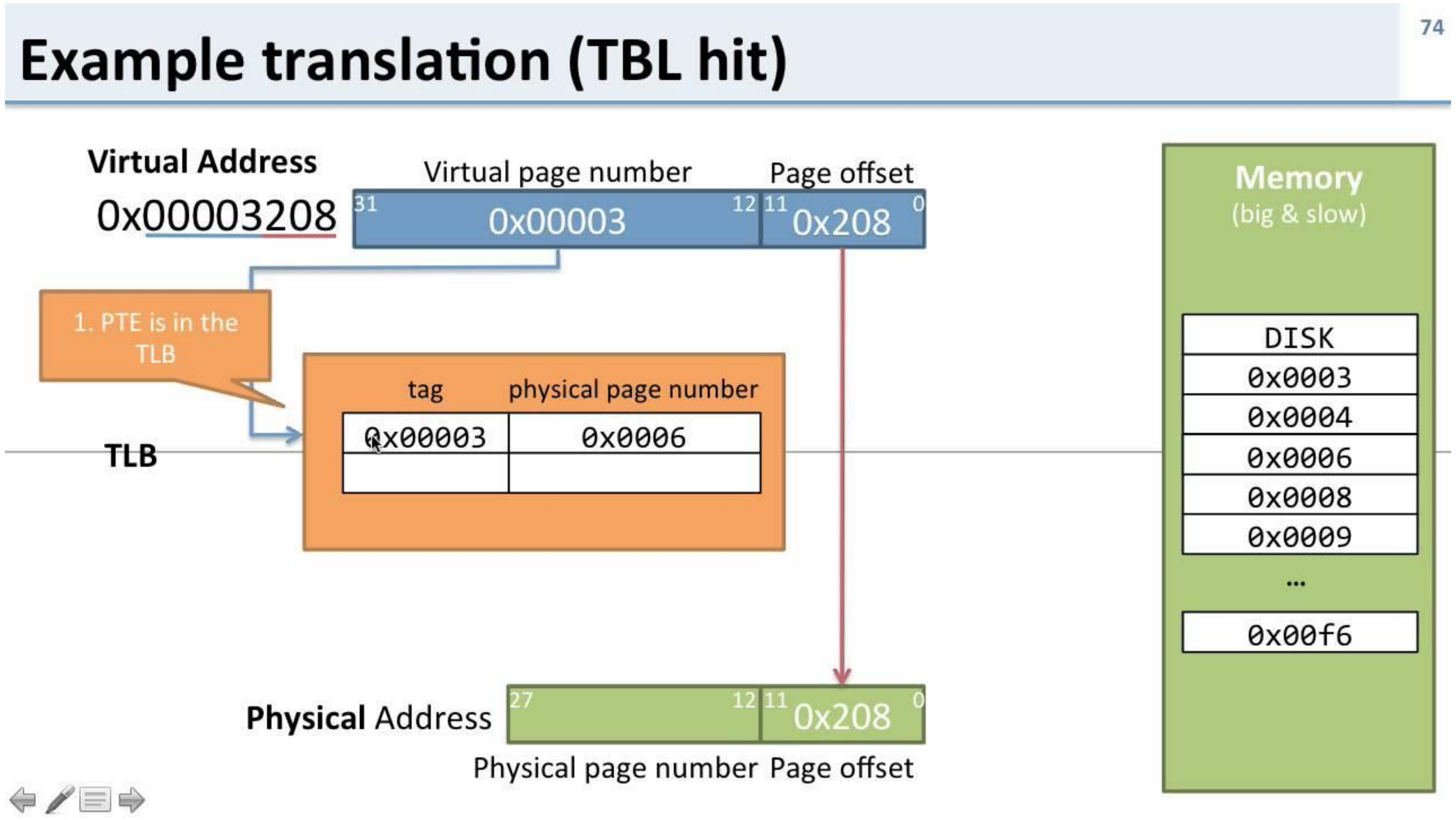
Virtual Memory Continued…

- ISA determines virtual address space (MIPS => 2^{32} bits)
- Physical Address space based on RAM

- Page Fault: when page table entry not in RAM we must fetch it
- Dirty Bit: VM does not write-through, instead dirty bit is set on writes

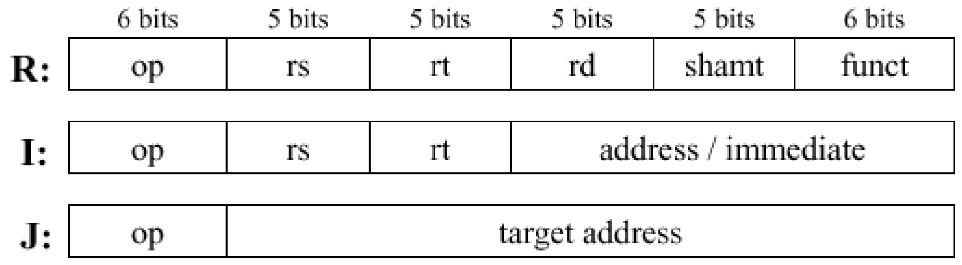
- How long does a page fault take?

Page Tables, Address Translation, TLB

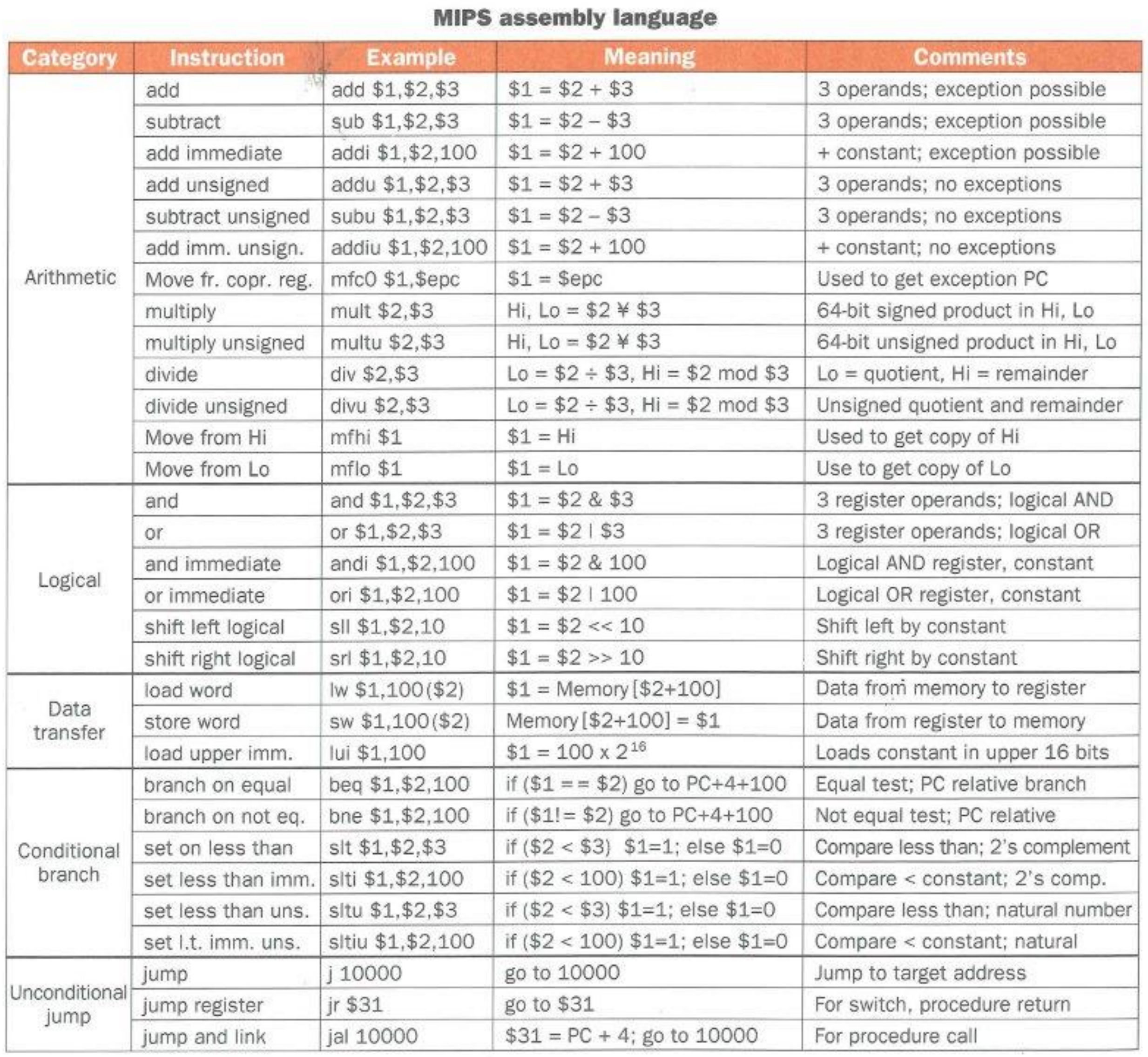


MIPS Instruction Set

- 32 bit (4 Byte) instructions
- 3 basic instruction types



op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions



MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; exception possible
	subtract	sub \$1,\$2,\$3	\$1 = \$2 – \$3	3 operands; exception possible
	add immediate	addi \$1,\$2,100	\$1 = \$2 + 100	+ constant; exception possible
	add unsigned	addu \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; no exceptions
	subtract unsigned	subu \$1,\$2,\$3	\$1 = \$2 – \$3	3 operands; no exceptions
	add imm. unsign.	addiu \$1,\$2,100	\$1 = \$2 + 100	+ constant; no exceptions
	Move fr. copr. reg.	mfc0 \$1,\$epc	\$1 = \$epc	Used to get exception PC
	multiply	mult \$2,\$3	Hi, Lo = \$2 ¥ \$3	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$2,\$3	Hi, Lo = \$2 ¥ \$3	64-bit unsigned product in Hi, Lo
	divide	div \$2,\$3	Lo = \$2 ÷ \$3, Hi = \$2 mod \$3	Lo = quotient, Hi = remainder
	divide unsigned	divu \$2,\$3	Lo = \$2 ÷ \$3, Hi = \$2 mod \$3	Unsigned quotient and remainder
	Move from Hi	mfhi \$1	\$1 = Hi	Used to get copy of Hi
	Move from Lo	mflo \$1	\$1 = Lo	Use to get copy of Lo
Logical	and	and \$1,\$2,\$3	\$1 = \$2 & \$3	3 register operands; logical AND
	or	or \$1,\$2,\$3	\$1 = \$2 | \$3	3 register operands; logical OR
	and immediate	andi \$1,\$2,100	\$1 = \$2 & 100	Logical AND register, constant
	or immediate	ori \$1,\$2,100	\$1 = \$2 | 100	Logical OR register, constant
	shift left logical	sll \$1,\$2,10	\$1 = \$2 << 10	Shift left by constant
	shift right logical	srl \$1,\$2,10	\$1 = \$2 >> 10	Shift right by constant
Data transfer	load word	lw \$1,100(\$2)	\$1 = Memory[\$2+100]	Data from memory to register
	store word	sw \$1,100(\$2)	Memory[\$2+100] = \$1	Data from register to memory
	load upper imm.	lui \$1,100	\$1 = 100 x 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$1,\$2,100	if (\$1 = = \$2) go to PC+4+100	Equal test; PC relative branch
	branch on not eq.	bne \$1,\$2,100	if (\$1!= \$2) go to PC+4+100	Not equal test; PC relative
	set on less than	slt \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0	Compare less than; 2's complement
	set less than imm.	slti \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0	Compare < constant; 2's comp.
	set less than uns.	sltu \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0	Compare less than; natural number
	set l.t. imm. uns.	sltiu \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0	Compare < constant; natural
Unconditional jump	jump	j 10000	go to 10000	Jump to target address
	jump register	jr \$31	go to \$31	For switch, procedure return
	jump and link	jal 10000	\$31 = PC + 4; go to 10000	For procedure call

Addressing Modes [MIPS]

- Register Addressing (direct)
  - add $t0, $t1, $t2
  - PC <= R[s]   (program counter gets contents of register s)
- Base Addressing (indirect)
  - Load and Store instructions
  - lw $rt, offset_value($rs)
  - ADDR <= R[s] + sign_extend(offset_value)
- Immediate Addressing
  - addi $t1, $t0, immediate
  - PC <= R[s] + immediate
- PC-Relative Addressing (branch instructions)
  - I-type instructions
  - PC <= PC + sign_extend(SLL($IR_{15\text{-}0}$, 2))

If-Else Loop MIPS

# Pseudocode:

```
if (a < b + 3)
        a = a + 1
else
        a = a + 2
        b = b + a
```

# Register mappings:

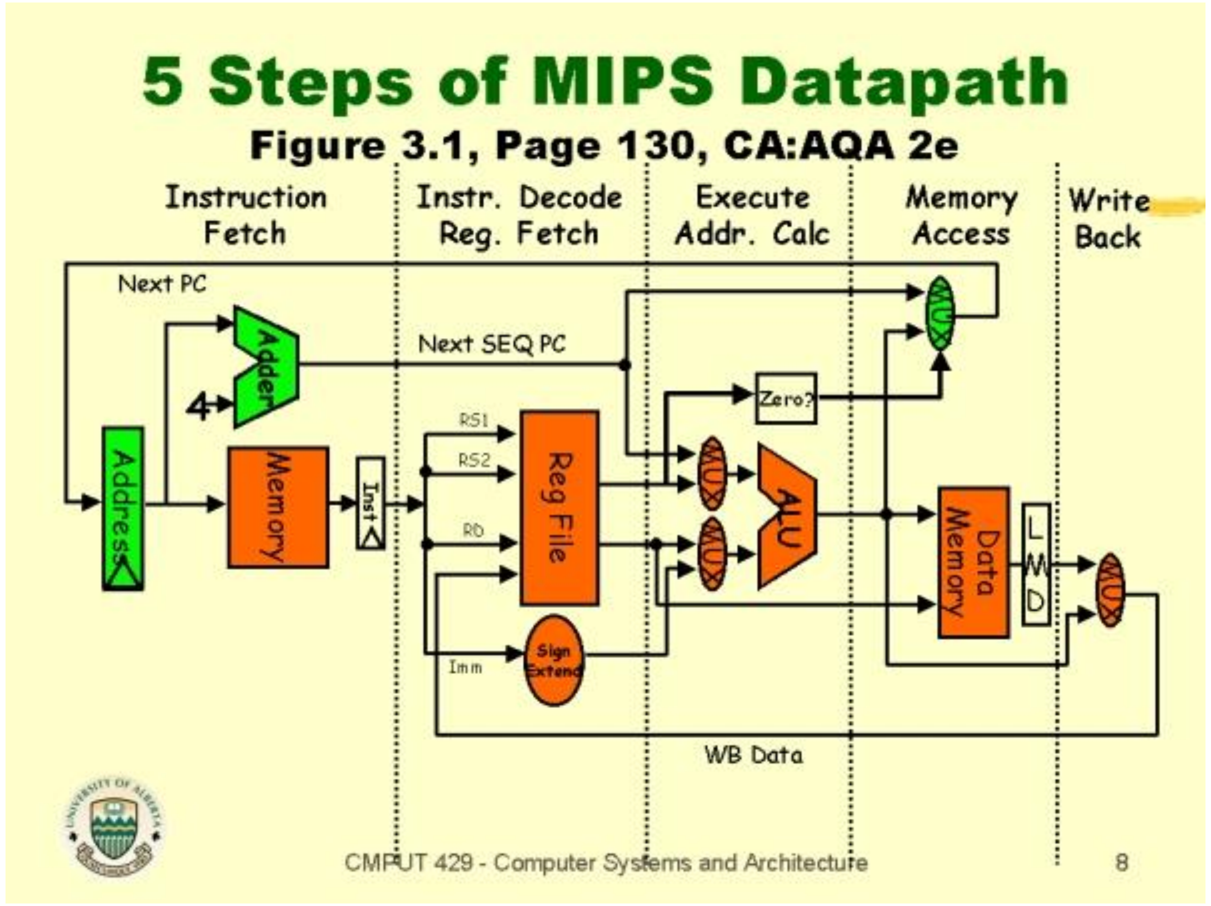
a: $t0, b: $t1

# Assembly Equivalent

```
        addi $t2, $t1, 3       # tmp = b + 3
        blt  $t0, $t2, THEN    # if (a < tmp)
        addi $t0, $t0, 2       # (else case) a=a+2
        j END
THEN:   addi $t0, $t0, 1       # (then case) a=a+1
END:    add  $t1, $t1, $t0     # b = b + a
```

MIPS Datapath

- 5 stage pipeline example
  - Fetch, Read/decode, ALU, Memory (optional), Write



Data Dependencies (and Hazards)

- RAW [Flow, True] Dependency
- WAW [Output, False] Dependency
- WAR [Anti-, False] Dependency
- Dependence: A property of the *program*
- Hazard: A property of the *pipeline*
  - Occurs when dependence causes incorrect execution

Identifying Dependencies

- **Example**: identify dependencies in following code

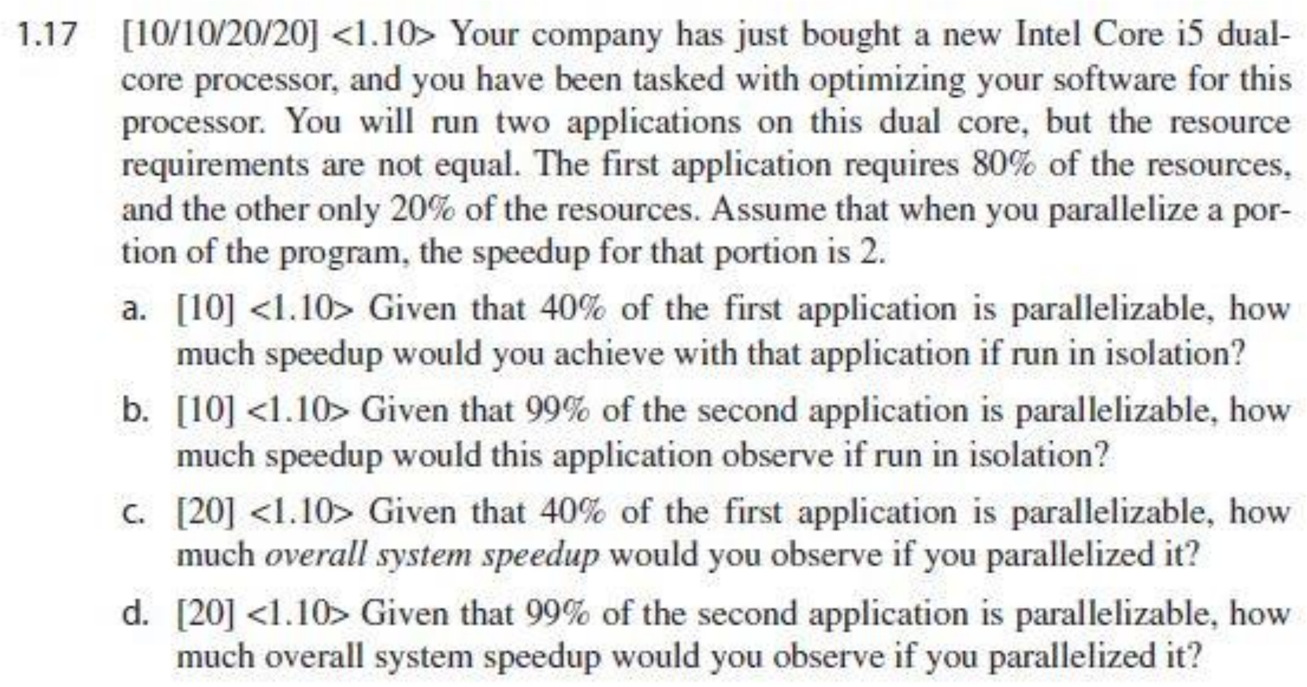
```
ADD R1, R2, R3
SUB R7, R1, R8
MUL R1, R5, R6
```

- Control Hazards: We must *flush* pipeline
- Data Hazards: We can *stall* pipeline or *forward* instructions

Amdahl's Law

- **Amdahl's Law**: $\text{Speedup}_{\text{new}} = \frac{1}{(1 - Fraction_{enhanced}) + {}^{Fraction_{enhanced}}/_{Speedup_{enhanced}}}$
  - $\boldsymbol{Fraction_{enhanced}}$ is *"the fraction of the computation time in the original computer that can be converted to take advantage of the enhancement"* (<= 1)
  - $\boldsymbol{Speedup_{enhanced}}$ is *"the improvement gained by the enhanced execution mode – how much faster the task would run if the enhanced mode were used for the entire program"* (> 1)
- **Example**: Patterson Hennessy Problem

PH Chapter 1 – Problem 17



1.17 [10/10/20/20] <1.10> Your company has just bought a new Intel Core i5 dual-core processor, and you have been tasked with optimizing your software for this processor. You will run two applications on this dual core, but the resource requirements are not equal. The first application requires 80% of the resources, and the other only 20% of the resources. Assume that when you parallelize a portion of the program, the speedup for that portion is 2.

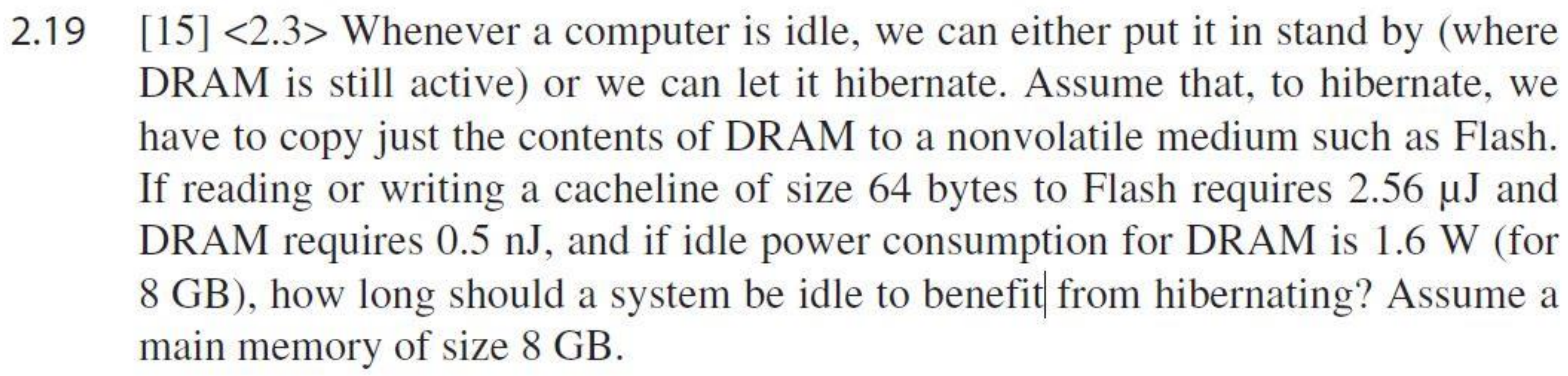
a. [10] <1.10> Given that 40% of the first application is parallelizable, how much speedup would you achieve with that application if run in isolation?

b. [10] <1.10> Given that 99% of the second application is parallelizable, how much speedup would this application observe if run in isolation?

c. [20] <1.10> Given that 40% of the first application is parallelizable, how much *overall system speedup* would you observe if you parallelized it?

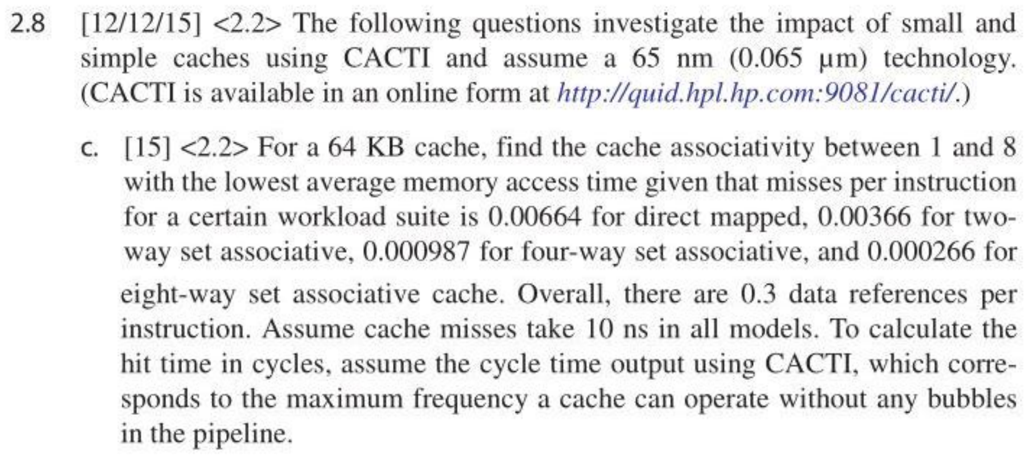
d. [20] <1.10> Given that 99% of the second application is parallelizable, how much overall system speedup would you observe if you parallelized it?

PH Chapter 2 – Problem 19



2.19 [15] <2.3> Whenever a computer is idle, we can either put it in stand by (where DRAM is still active) or we can let it hibernate. Assume that, to hibernate, we have to copy just the contents of DRAM to a nonvolatile medium such as Flash. If reading or writing a cacheline of size 64 bytes to Flash requires 2.56 μJ and DRAM requires 0.5 nJ, and if idle power consumption for DRAM is 1.6 W (for 8 GB), how long should a system be idle to benefit from hibernating? Assume a main memory of size 8 GB.

PH Chapter 2 – Problem 19



2.8 [12/12/15] <2.2> The following questions investigate the impact of small and simple caches using CACTI and assume a 65 nm (0.065 µm) technology. (CACTI is available in an online form at *http://quid.hpl.hp.com:9081/cacti/*.)

c. [15] <2.2> For a 64 KB cache, find the cache associativity between 1 and 8 with the lowest average memory access time given that misses per instruction for a certain workload suite is 0.00664 for direct mapped, 0.00366 for two-way set associative, 0.000987 for four-way set associative, and 0.000266 for eight-way set associative cache. Overall, there are 0.3 data references per instruction. Assume cache misses take 10 ns in all models. To calculate the hit time in cycles, assume the cycle time output using CACTI, which corresponds to the maximum frequency a cache can operate without any bubbles in the pipeline.

Previously Calculated for (a)

Associativity	Access Time	Cycle Time
1-Way	0.863 ns	0.504 ns
2-Way	1.121 ns	0.509 ns
4-Way	1.371 ns	0.829 ns
8-Way	2.035 ns	0.790 ns

Equations:
Avg. Access Time = (Hit% × Hit Time) + (Miss% × Miss Penalty)
Hit Time = Access Time / Cycle Time [Cycles]
Miss % = Misses per Instruction / References per Instruction
Hit % = 1 - Miss %
Miss Penalty = Cache Miss Time / Cycle Time [Cycles]