CMPEN 431
Computer Architecture
Fall 2018

Review Slides Part 1
(Focus: Chapter 2, “Instructions: Language of the Computer”)

Jack Sampson(www.cse.psu.edu/~sampson)

[Slides adapted from work by Mary Jane Irwin, in turn adapted from *Computer Organization and Design, Revised 4th Edition*, Patterson & Hennessy, © 2011, Morgan Kaufmann]

Where to find more on these topics

- Reading
  - MIPS ISA Review, PH, Chapter 2
  - MIPS ALU Review, PH, Chapter 3
  - Caching and Memory Hierarchy Review, PH Chapter 5

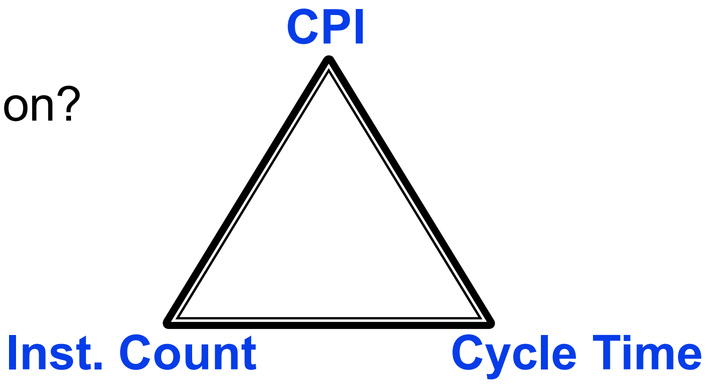
- Slides
  - Slower-paced review in Lectures 2 through 5 of Fall 2016 slides on CANVAS

- Review = FAST-PACED
  - … but still stop me if you have questions!
  - Make use of instructor and assistant OH to shore up any areas you're not up to speed on
  - Arithmetic review will be particularly fast. If you've forgotten your discrete arithmetic from CMPEN 270, please review on your own as we're focusing the in-class time on higher-level topics!

Recall: Evaluating ISAs

- Design-time metrics
  - Can it be implemented, at what cost (design, fabrication, test, packaging), with what power, with what reliability?
  - Can it be programmed? Ease of compilation?
- Static Metrics
  - How many bytes does the program occupy in memory?
- Dynamic Metrics
  - How many instructions are executed? How many bytes does the processor fetch to execute the program?
  - How many clocks are required per instruction?
  - How "lean" (fast) a clock is practical?

Best Metric: Time to execute the program!
depends on the instructions set, the processor organization, and compilation techniques.



RISC vs CISC

- RISC = Reduced Instruction Set Computer
  - MIPS, SPARC, PowerPC, ARM (Cortex), etc.
- CISC = Complex Instruction Set Computer
  - X86 is the only surviving example
- Goals in the 1980s – reduce design time, faster/smaller implementation, ISA processor/compiler co-design
- ISAs are **now** measured by how well **compilers** use them, not by how well or how easily assembly language programmers use them
- There are (or, at least, it's believed there are) many old and useful programs that only exist as machine code, so supporting old ISAs has economic value

MIPS (RISC) Design Principles – Part 1

- ❑ Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
    - fixed size instructions, small number of instruction formats (three for MIPS), opcode in a fixed location (the first 6 bits for MIPS), etc.
- ❑ Smaller is faster
  - Smaller ISA reduces design and implementation costs (and power?), chip sizes, etc.
  - Faster
    - limited instruction set, load-store architecture
      - http://www.arm.com/products/processors/instruction-set-architectures/index.php
    - limited number of registers in RF
    - limited number of memory addressing modes
      - Memory address = register value + constant

Below “Program” level in the abstraction stack

- High-level language program (in C)

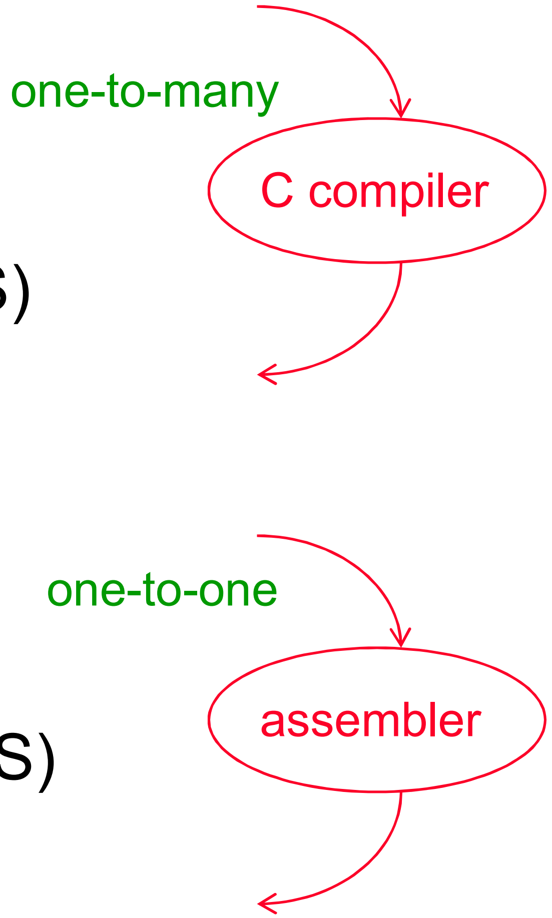
```
swap (int v[], int k)
(int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
)
```

- Assembly language program (for MIPS)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

- Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
 . . .
```

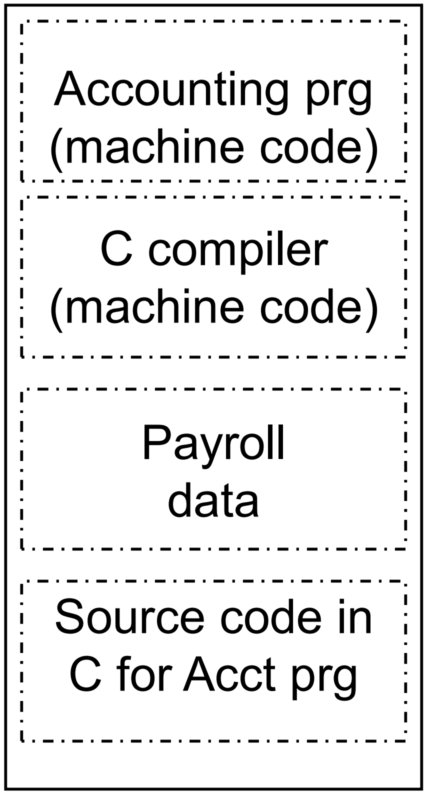


Two Key Principles of Machine Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data

2. Programs are stored in alterable memory (that can be read or written to just like data)

Memory



Accounting prg (machine code)

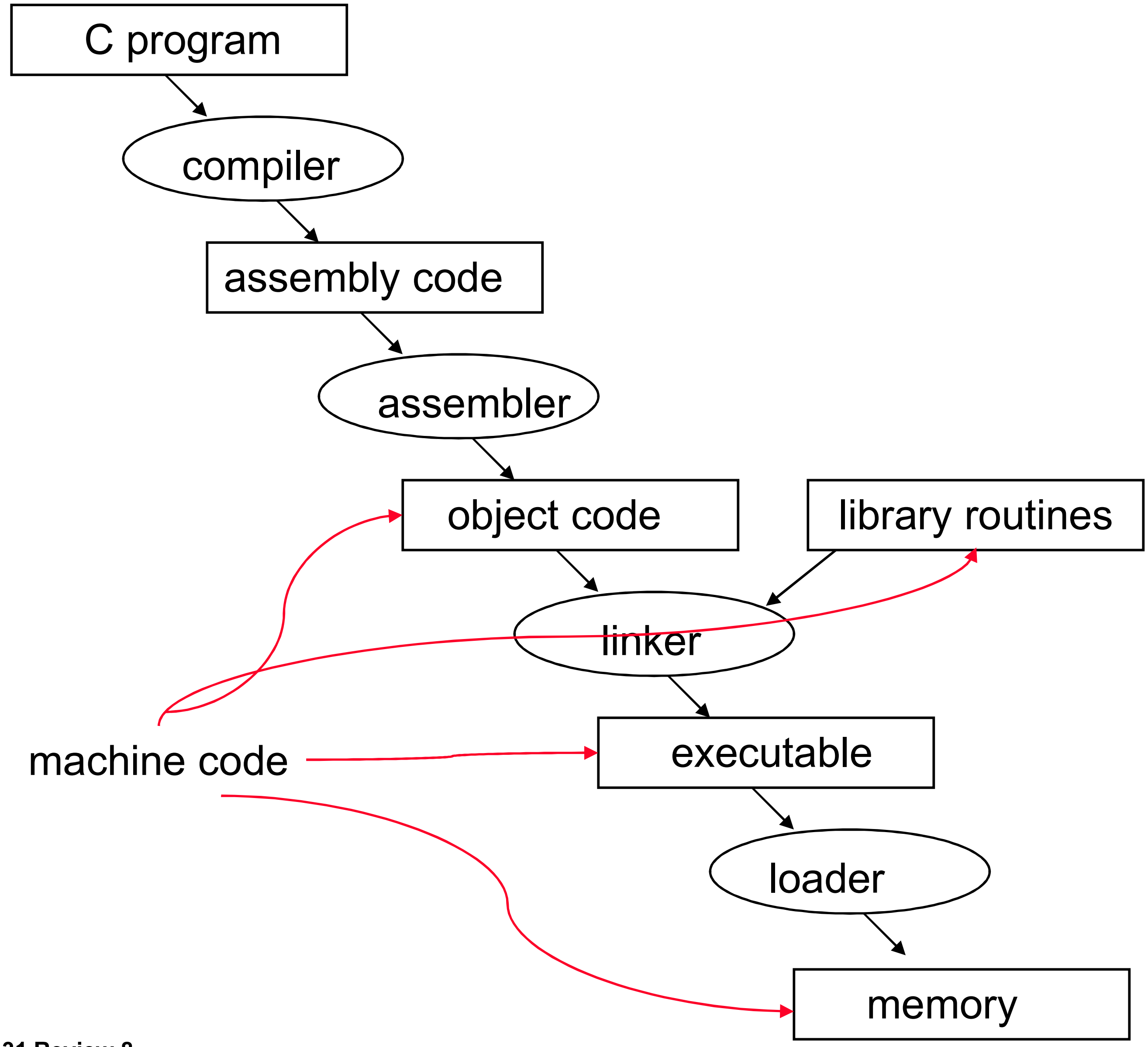
C compiler (machine code)

Payroll data

Source code in C for Acct prg

- Stored-program concept
  - Programs can be shipped as files of binary numbers – binary compatibility
  - Computers can inherit ready-made software provided they are compatible with an existing ISA – this led the industry to align around a small number of ISAs+ABIs

The C Code Translation Hierarchy



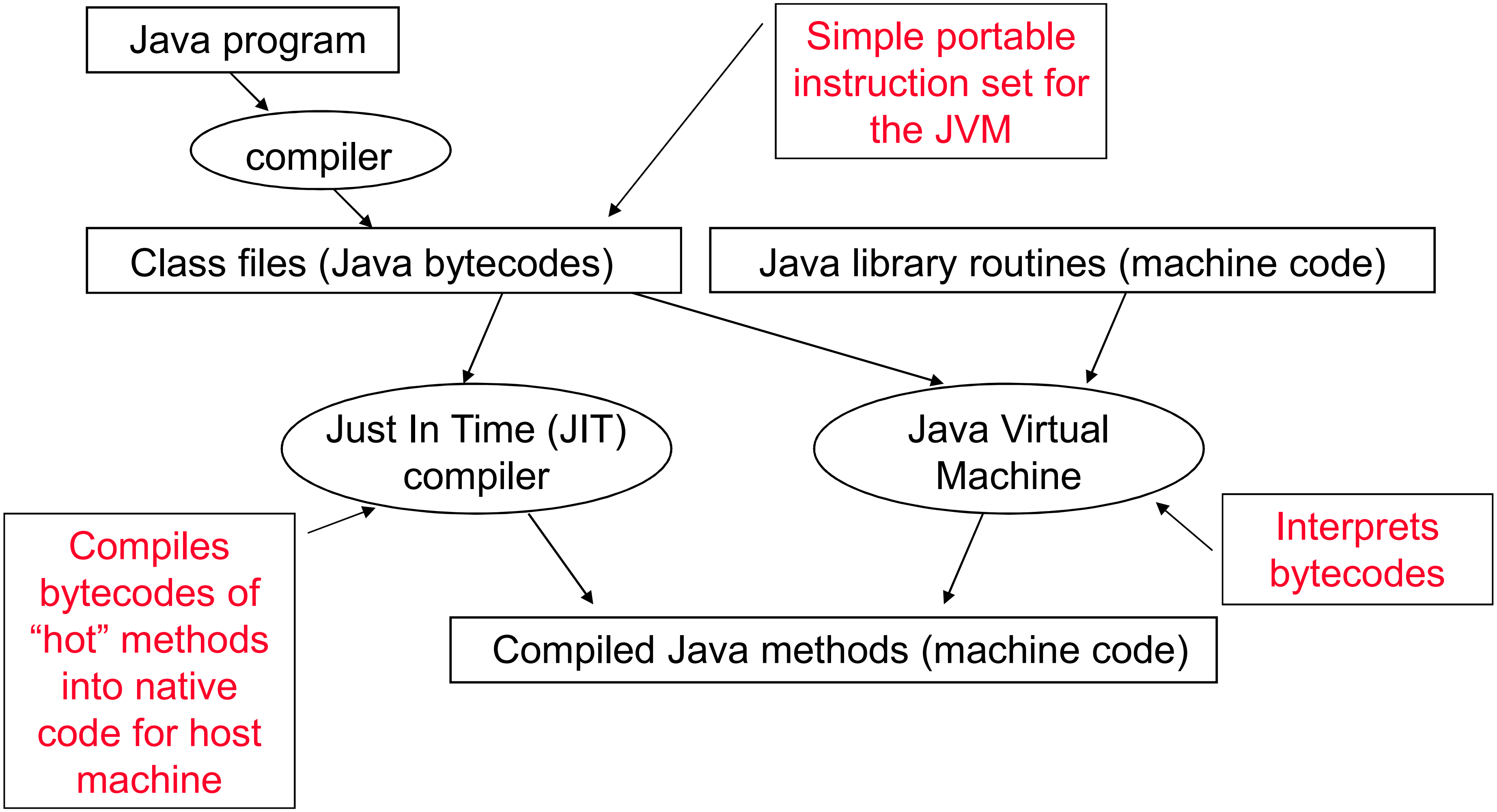
Compiler Benefits

- Comparing performance for bubble (exchange) sort
  - To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 GHz clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

gcc opt	Relative performance	Clock cycles (M)	Instr count (M)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (proc mig)	2.41	65,747	44,993	1.46

- The unoptimized code has the best CPI, the O1 version has the lowest instruction count, but the O3 version is the fastest. Why?

Aside: The Java Code Translation Hierarchy


Java program
compiler
Simple portable instruction set for the JVM
Class files (Java bytecodes)
Java library routines (machine code)
Just In Time (JIT) compiler
Java Virtual Machine
Compiles bytecodes of “hot” methods into native code for host machine
Interprets bytecodes
Compiled Java methods (machine code)

Sorting in C versus Java

- Comparing performance for two sort algorithms in C and Java
  - The JVM/JIT is Sun/Hotspot version 1.3.1/1.3.1

	Method	Opt	Bubble	Quick	Speedup quick vs bubble
			Relative performance		
C	Compiler	None	1.00	1.00	2468
C	Compiler	O1	2.37	1.50	1562
C	Compiler	O2	2.38	1.50	1555
C	Compiler	O3	2.41	1.91	1955
Java	Interpreter		0.12	0.05	1050
Java	JIT compiler		2.13	0.29	338

- Observations?

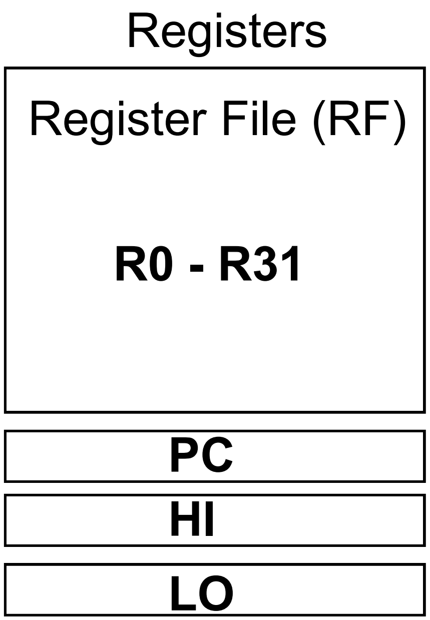
Review: The Fetch/Execute Cycle

- Memory stores both instruction and data (object code and data bits … just bits)

1. Instruction is fetched from memory at the address indicated by the Program Counter (PC)
2. Control unit decodes the instruction, generates signals to other components so the instruction can be executed
    1. Data is read from the RF or, if necessary, from memory
    2. Datapath executes the instruction as directed by the Control
    3. Data is written to the RF of, if necessary, to memory
3. Control updates the PC which specifies the next instruction to fetch and then execute

MIPS-32 ISA

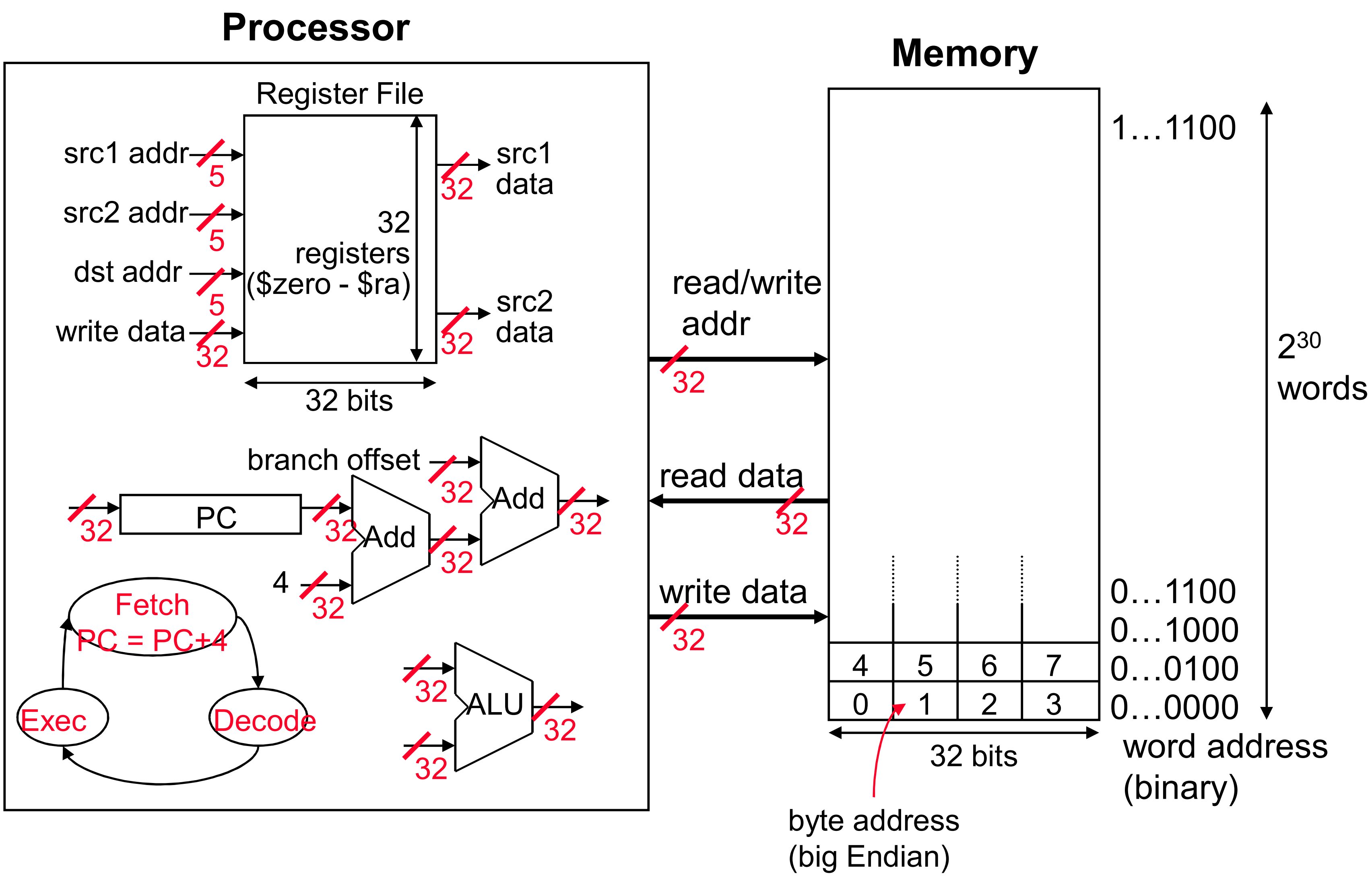
- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide

op	**rs**	**rt**	**rd**	**sa**	**funct**	R format
op	**rs**	**rt**	**immediate**			I format
op	**jump target**					J format

MIPS Organization



Processor

Register File

src1 addr 5

src2 addr 5

dst addr 5

write data 32

32 registers ($zero - $ra)

src1 data 32

src2 data 32

32 bits

branch offset 32

PC 32

Add 32

4 32

Add 32

32

Fetch PC = PC+4

Decode

Exec

ALU 32 32 32

Memory

read/write addr 32

read data 32

write data 32

1…1100

2^{30} words

0…1100

0…1000

0…0100

0…0000

4 5 6 7

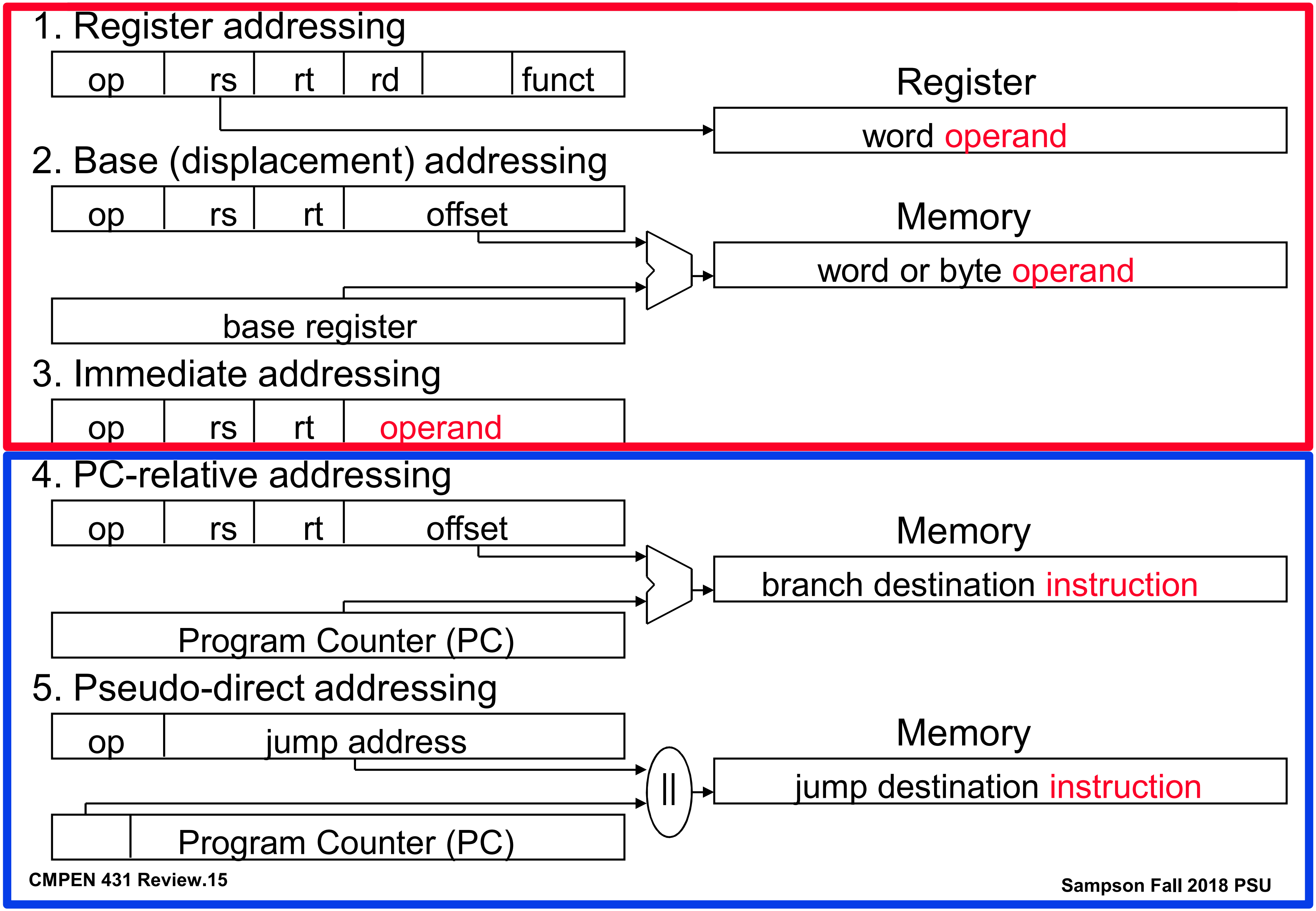
0 1 2 3

32 bits

word address (binary)

byte address (big Endian)

Review: MIPS Addressing Modes Illustrated



1. Register addressing

op | rs | rt | rd | | funct

Register

word operand

2. Base (displacement) addressing

op | rs | rt | offset

Memory

word or byte operand

base register

3. Immediate addressing

op | rs | rt | operand

4. PC-relative addressing

op | rs | rt | offset

Memory

branch destination instruction

Program Counter (PC)

5. Pseudo-direct addressing

op | jump address

Memory

jump destination instruction

Program Counter (PC)

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

  - Each arithmetic instruction performs one operation
  - Each specifies exactly three operands that are all contained in the datapath's RF (`$t0,$s1,$s2`)

destination ← source1 op source2

  - Instruction Format (R format)

0	17	18	8	0	0x22
`alu`	`$s1`	`$s2`	`$t0`	`unused`	`sub`

MIPS Instruction Fields

- MIPS fields given names to make them easier to refer to

op	rs	rt	rd	shamt	funct

op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result’s destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS (RISC) Design Principles – Part 2

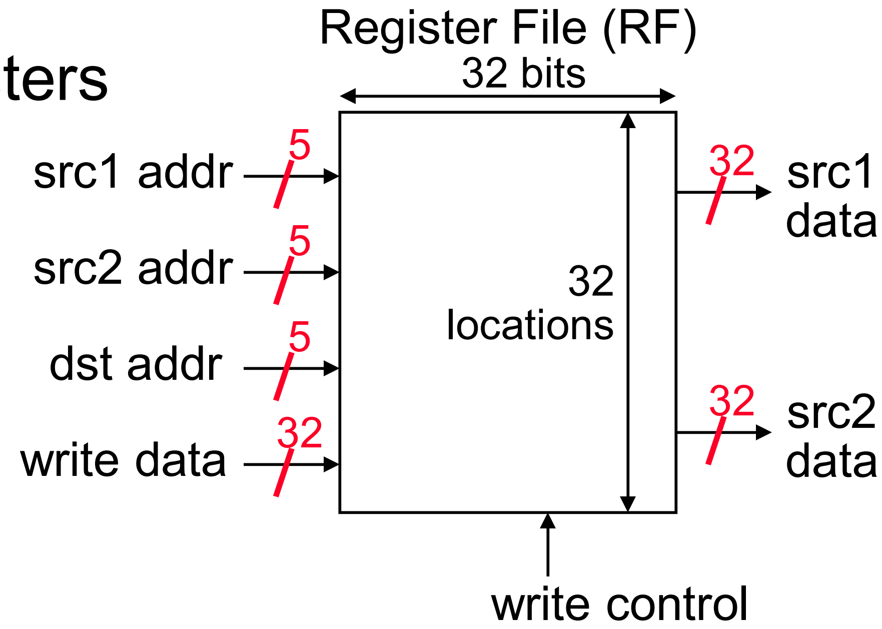
- Make the common case fast

COMMON CASE FAST

  - Find the biggest impact on performance
    - E.g., accessing registers is fast, memory is slow
  - Which are the “common cases”? Are they the same for all programs? Will they be the same in the future?
    - arithmetic operands in the RF (load-store machine)
    - allow instructions to contain immediate operands (small constants), otherwise have to bring the constants in from memory, store them in the RF, and access them from there
- Good design demands good compromises
  - Evaluate the many options, determine their impact on performance (CPI?, IC?, clock rate?), make a reasonable choice that doesn’t limit future extensions
    - three instruction formats, as similar as possible
    - only two branch instructions (`beq`, `bne`) with a way to do many more with the `slt` “set up” instruction

MIPS Register File (RF)

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port



- Registers are
  - Faster than main memory
    - But RFs with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Increasing number of read/write ports impacts speed quadratically
  - Improves code density (a register is named with fewer bits than a memory location)
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack

Aside: MIPS Register Convention (ABI)

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

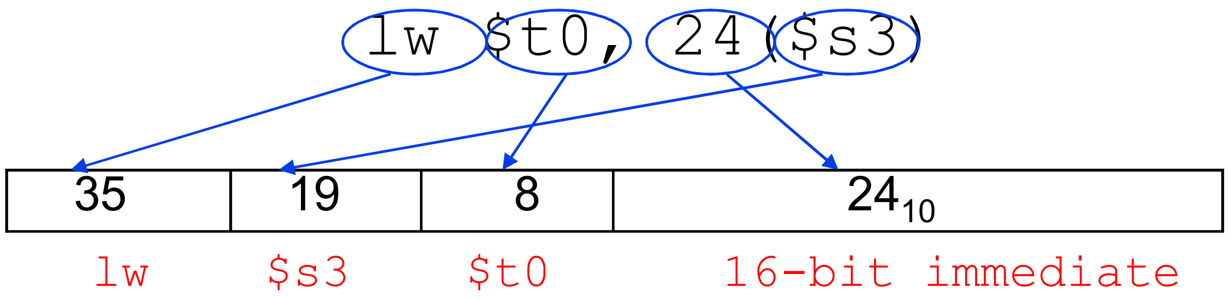
```
lw    $t0, 4($s3)   #load word from memory
sw    $t0, 8($s3)   #store word to  memory
```

- The data is loaded into (`lw`) or stored from (`sw`) a register in the register file – a 5 bit address

- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the sign-extended offset value
  - The offset is a 16-bit 2's complement number, so access is limited to memory locations within a region of ±2^{13} (8,192) words or ±2^{15} (32,768) bytes of the address in the base register

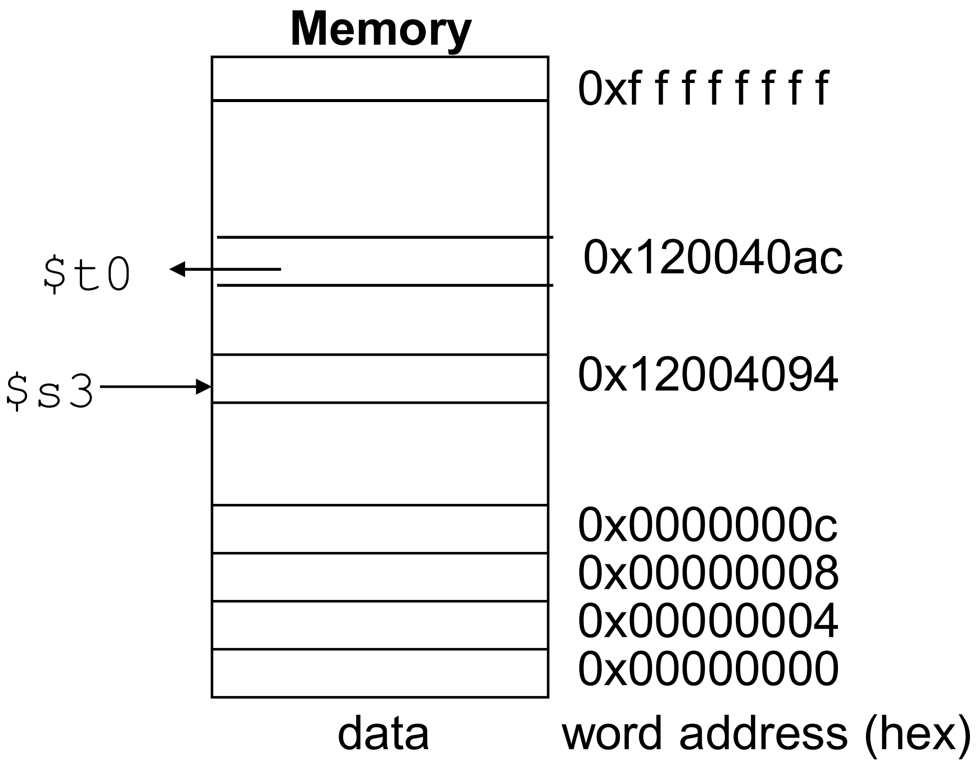
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac

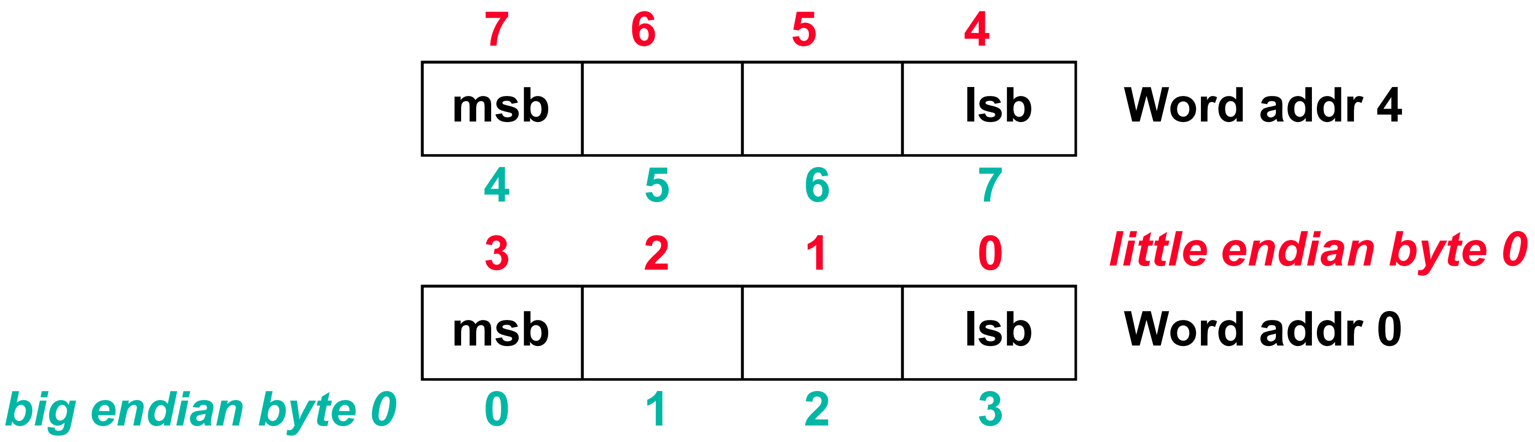


Byte Addresses

- Since 8-bit bytes are so useful, most architectures support addressing individual bytes in memory
  - Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha, ARM



	7	6	5	4	
	msb			lsb	**Word addr 4**
	4	5	6	7	
	3	2	1	0	***little endian byte 0***
	msb			lsb	**Word addr 0**
big endian byte 0	0	1	2	3	

Aside: Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)   #load byte from memory
sb    $t0, 6($s3)   #store byte to  memory
```

sb	$s3	$t0	6
0x28	19	8	16 bit offset

- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

MIPS Immediate Instructions

- Small constants are used often in typical code

COMMON CASE FAST

- Possible approaches?
  1. put “typical constants” in memory and load them into the RF
  2. create hard-wired registers (like $zero) for constants like 1
  3. have special instructions that contain constants !

```
addi  $sp, $sp, 4      #$sp = $sp + 4
slti  $t0, $s2, 15     #$t0 = 1 if $s2<15
                       # otherwise $t0 = 0
```

- Machine format (I format):

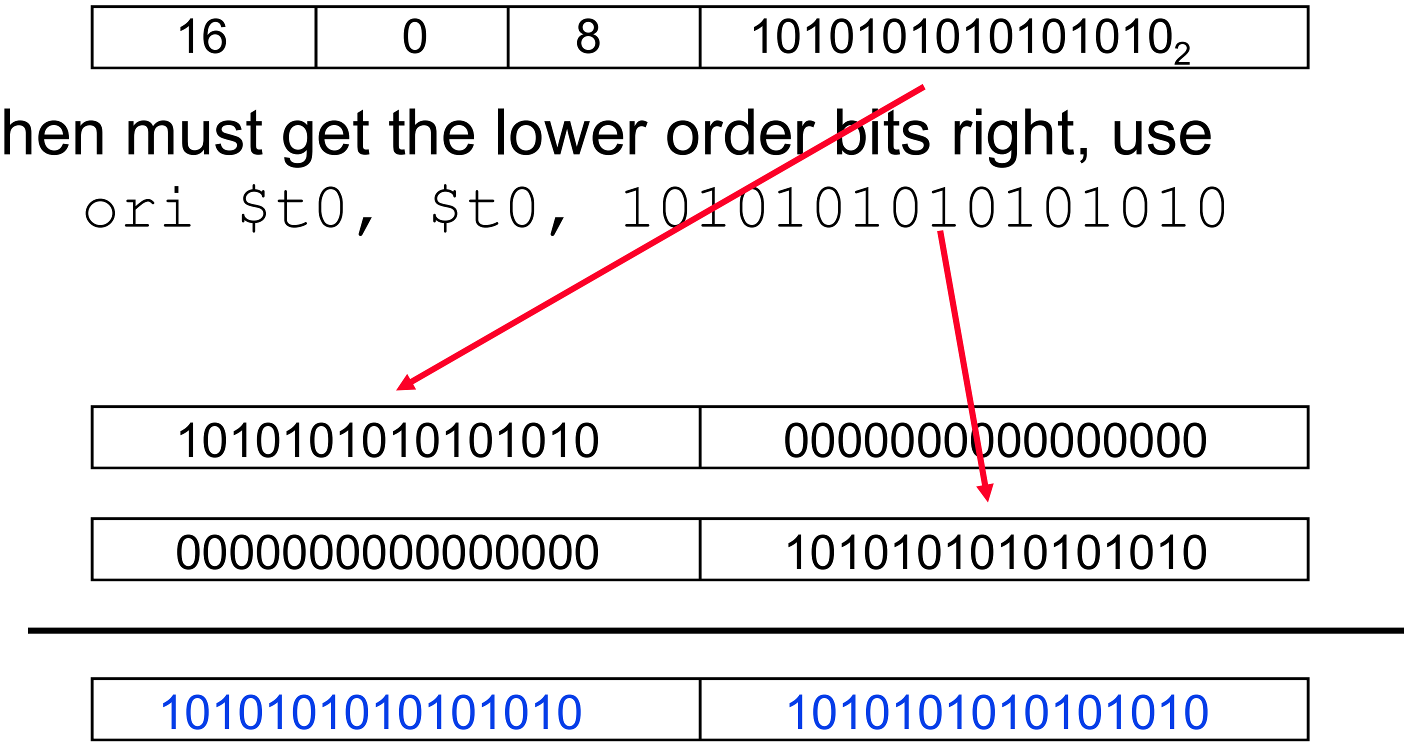
slti	$s2	$t0	15
0x0a	18	8	0x0f

- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $+2^{15}–1$ to -2^{15}

Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```



16	0	8	1010101010101010_2

- Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010
1010101010101010	1010101010101010

MIPS Shift Operations

- Need operations to pack and unpack 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8    #$t2 = $s0 << 8 bits
srl $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```

- Instruction Format (R format)

0		16	10	8	0x00
alu	unused	$s0	$t2	shamt	sll

- Such shifts are called logical because they fill with zeros
  - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 – 1$ or 31 bit positions

MIPS Logical Operations

- There are a number of bit-wise logical operations in the MIPS ISA

```
and $t0, $t1, $t2  #$t0 = $t1 & $t2

or  $t0, $t1, $t2  #$t0 = $t1 | $t2

nor $t0, $t1, $t2  #$t0 = not($t1 | $t2)
```

- Instruction Format (R format)

0	9	10	8	0	0x24
alu	$t1	$t2	$t0	unused	and

```
andi $t0, $t1, 0xFF00   #$t0 = $t1 & ff00

ori  $t0, $t1, 0xFF00   #$t0 = $t1 | ff00
```

- Instruction Format (I format)

0x0D	9	8	0xFF00

MIPS Control Flow Instructions

- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

- Ex: `if (i==j) h = i + j;`

```
            bne $s0, $s1, Lbl1
            add $s3, $s0, $s1
Lbl1:       ...
```

- Instruction Format (I format):

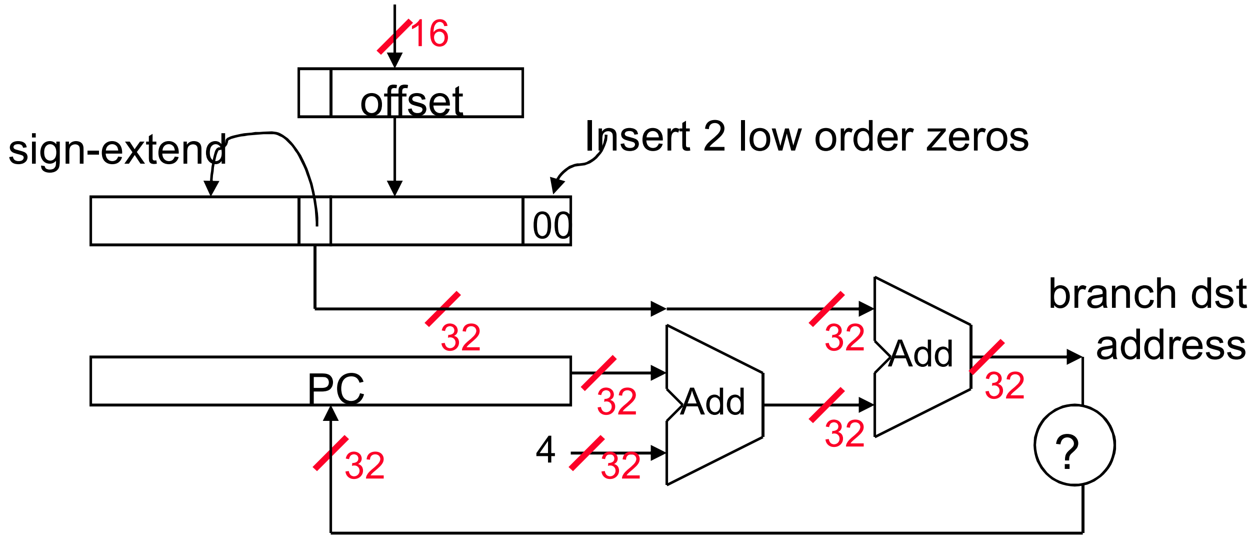
0x05	16	17	16 bit offset
bne	$s0	$s1	16-bit value

- How is the branch destination address specified?

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - Which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it is holding the address of the next instruction when the branch executes

from the low order 16 bits of the branch instruction



  - limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway

In Support of Branch Instructions

- We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt`
- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1    then
                      # $t0 = 1         else
                      # $t0 = 0
```

- Instruction format (R format):

0	16	17	8		0x2A

- Alternate versions of `slt`

```
slti $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

Aside: More Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than `blt $s1, $s2, Label`

```
slt  $at, $s1, $s2      #$at set to 1 if
bne  $at, $zero, Label  #$s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register (`$at`)

Aside: Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?

- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
          beq  $s0, $s1, L1
```

becomes

```
          bne  $s0, $s1, L2
          j    L1
     L2:
```

Other Control Flow Instructions

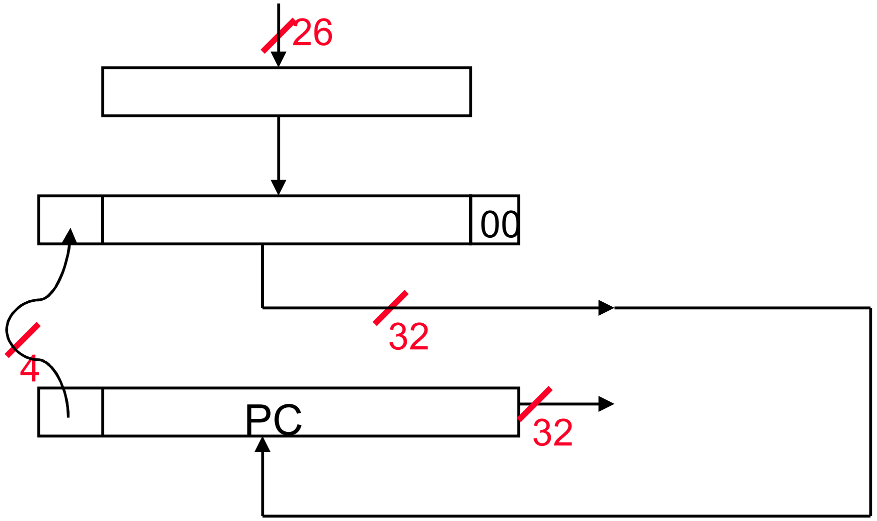
- MIPS also has an unconditional branch instruction or jump instruction:

```
j  label          #go to label
```

- Instruction Format (J Format):

0x02	26-bit address

from the low order 26 bits of the jump instruction



Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal   ProcedureAddress     #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (J format):

0x03	26 bit address

- Then can do procedure return with a jump register instr

```
jr    $ra                 #return
```

- Instruction format (R format):

0	31				0x08

Six Steps in Execution of a Procedure

- Recall the distinction from HLLs
  - parameters – names used when the function is written
  - arguments – values provided when the function is called
- Low-level languages like assembler will associate parameters with registers and memory locations, and arguments with the contents of those registers and memory locations

1. The main routine (caller) evaluates the function argument expressions and places argument values where the procedure (callee) can access them
   - `$a0` - `$a3`: four argument registers
   - Save previous values in those registers if necessary
   - Additional compiler-assigned space on the run-time stack

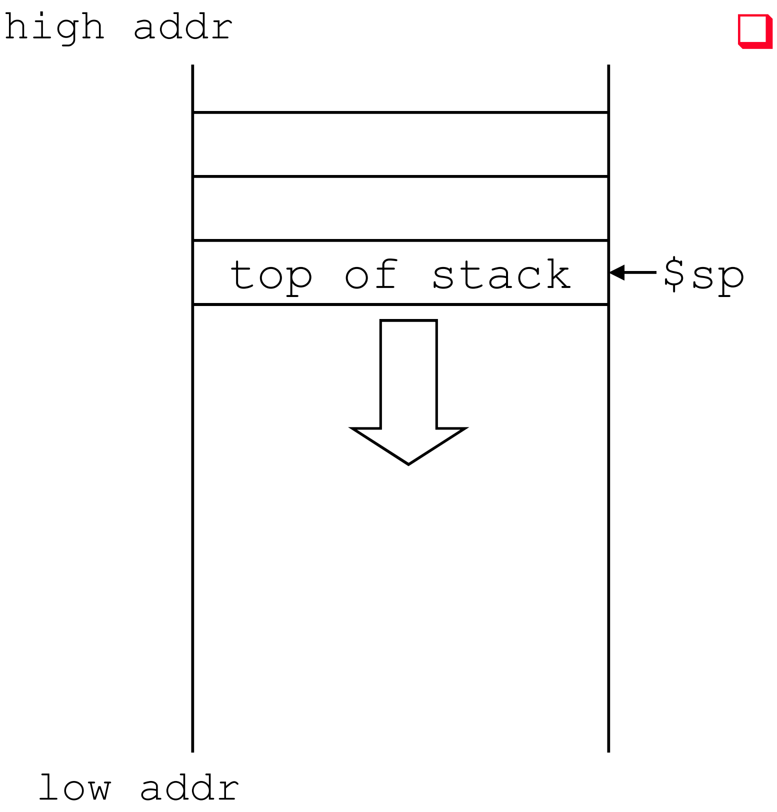
Six Steps in Execution of a Procedure, con’t

2. Caller transfers control to the callee
   - `jal` instruction, writes return address (PC + 4) to `$ra`
3. Callee acquires the more storage resources if needed
   - More registers, temporary space on the run-time stack, heap
4. Callee performs the desired task and places the result value in a place where the caller can access it
   - `$v0` - `$v1`: two value registers
   - Additional compiler-assigned space on the run-time stack
5. Callee prepares to return control to the caller
   - Restores previous register values (if necessary), releases temporary space on the run-time stack (adjust `$sp`)
6. Callee returns control to the caller
   - `$jr` instruction using `$ra`

- The caller continues execution after the function call

Aside: Spilling Registers

- What if the callee needs to use more registers than allocated to argument and return values?
  - callee uses a stack – a last-in-first-out queue



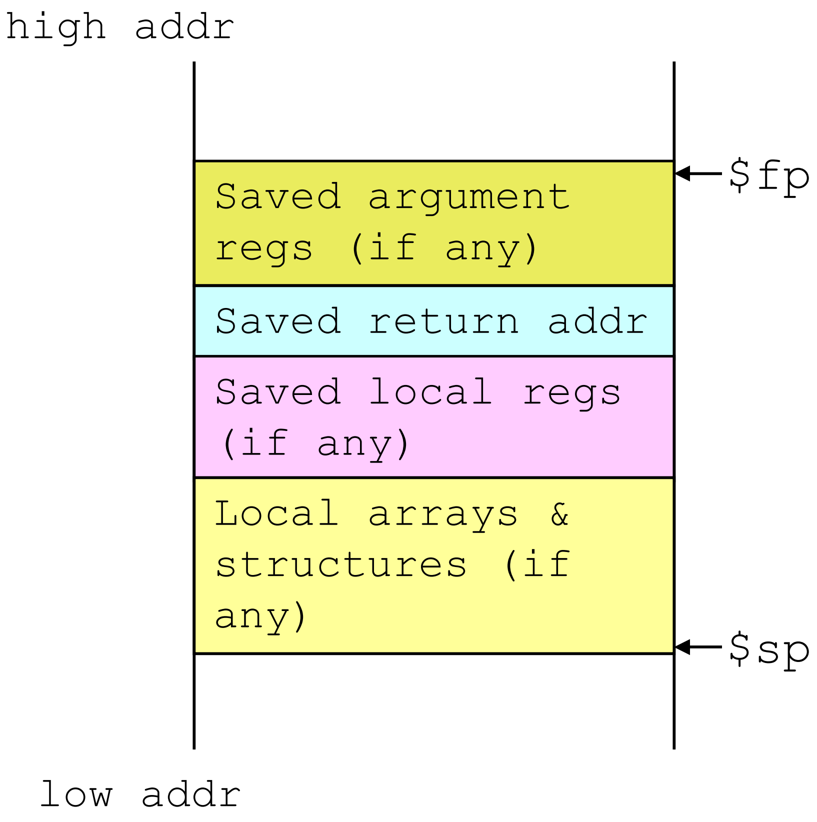
- One of the general registers, $sp ($29), is used to address the stack (which “grows” from high address to low address)
  - add data onto the stack – push

    $sp = $sp – 4
    data on stack at new $sp

  - remove data from the stack – pop

    data from stack at $sp
    $sp = $sp + 4

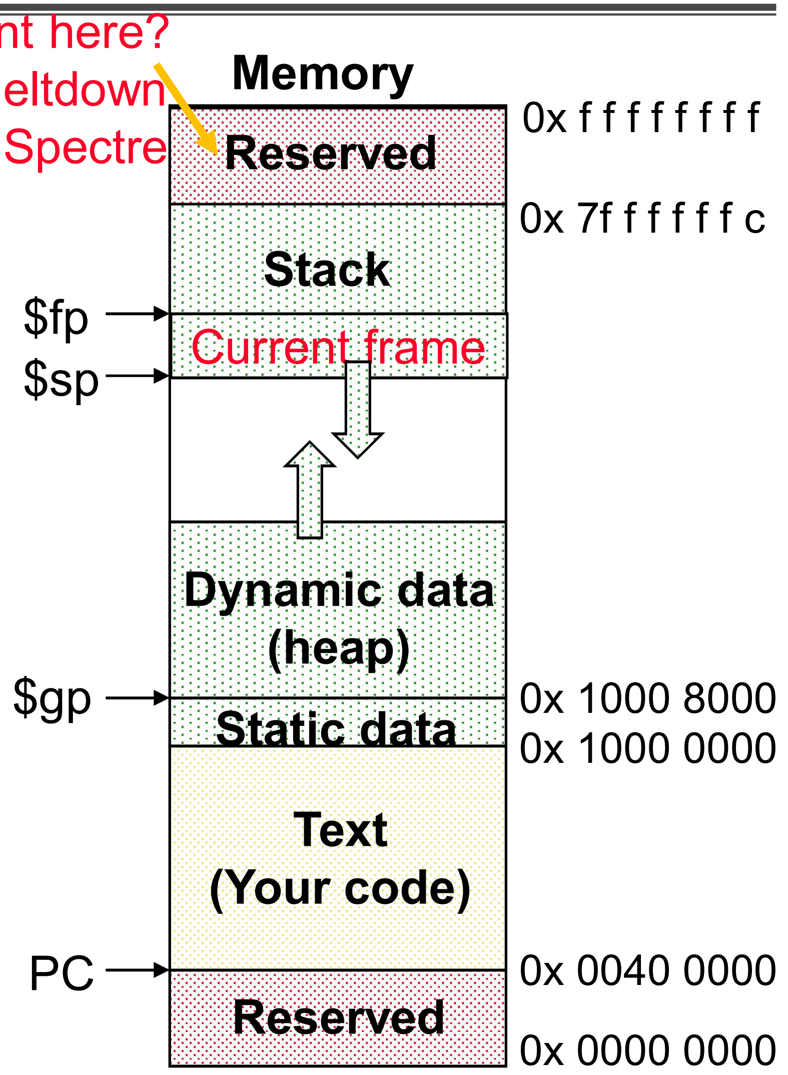
Aside: Allocating Space on the Stack


high addr
Saved argument regs (if any)
Saved return addr
Saved local regs (if any)
Local arrays & structures (if any)
$fp
$sp
low addr

- The segment of the stack containing a procedure's saved registers and local variables is its procedure frame (aka activation record)
  - The frame pointer (`$fp`) points to the first word of the frame of a procedure – providing a stable "base" register for the procedure
    - `$fp` is initialized using `$sp` on a call and `$sp` is restored using `$fp` on a return
    - `$fp` is unchanged during the procedure's execution
    - `$sp` could change even without calling another procedure, if we need more space on the stack

Aside: Allocating Space on the Heap

- Static data segment for constants and other staticially-allocated variables (e.g., globally defined arrays)
  - `$gp` = global pointer; never changes
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
  - In C, allocate space on the heap with `malloc()` and deallocate it with `free()`



For Later: Atomic Exchange Instructions

- Hardware support for synchronization mechanisms
  - Avoid data races where the results of the program can change depending on the relative ordering of events
  - Two memory accesses from different threads/cores to the same memory (cache) location, and at least one is a write – which goes first?
- Atomic exchange (atomic swap, atomic read/write)
  - Interchange a value in a register with a value in memory **atomically**, i.e., as one indivisible operation
  - Logically requires both a memory read and a memory write in a single, uninterruptable instruction. An alternative is to have a pair of specially configured instructions where no other access to the location is allowed between the read and the write.

```
ll   $t1, 0($s1)           #load linked

sc   $t0, 0($s1)           #store conditional
```

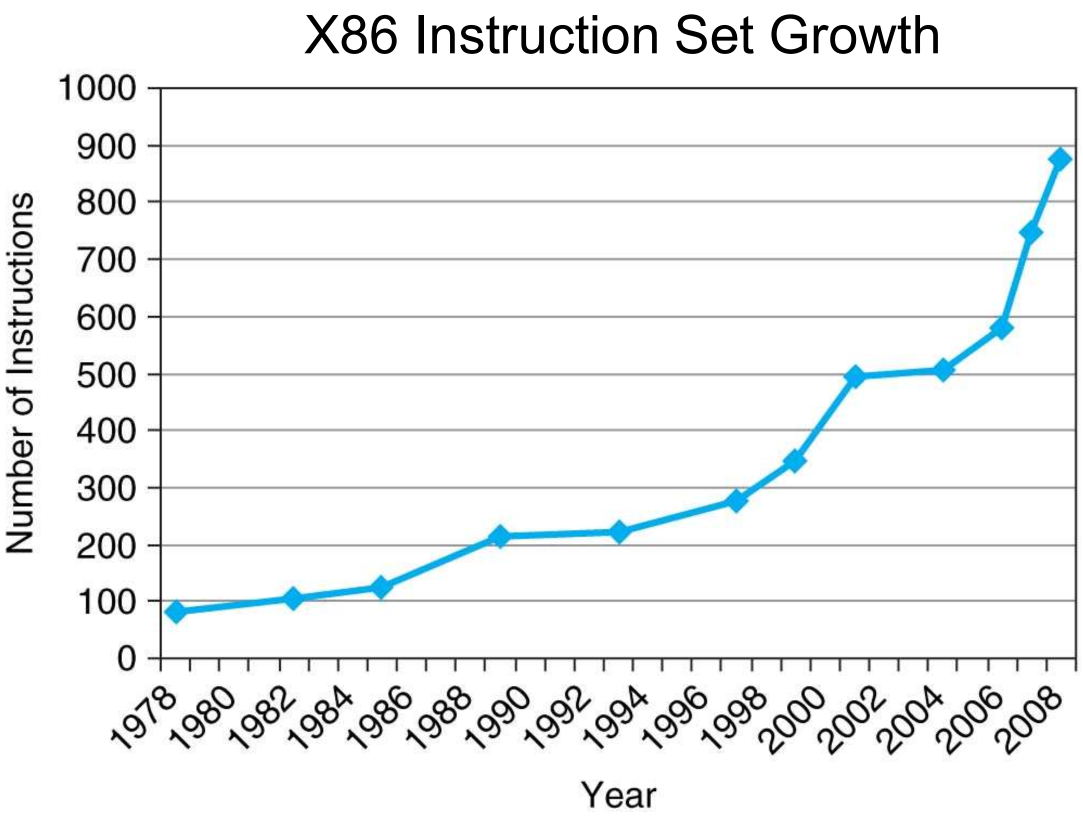
MIPS Instruction Classes Distribution

- Frequency of MIPS instruction classes for SPEC2006

Instruction Class	**Frequency**	
	SPECint	SPECfp
Arithmetic	16%	48%
Data transfer	35%	36%
Logical	12%	4%
Cond. Branch	34%	8%
Jump	2%	~0%

Fallacies and Pitfalls

- Fallacies:
  - More powerful instructions mean higher performance
  - Write in assembly language for highest performance
  - Binary compatibility means successful ISAs (e.g., x86) don't change



MIPS Arithmetic Support Review

MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

  `add, addi, addiu, addu`

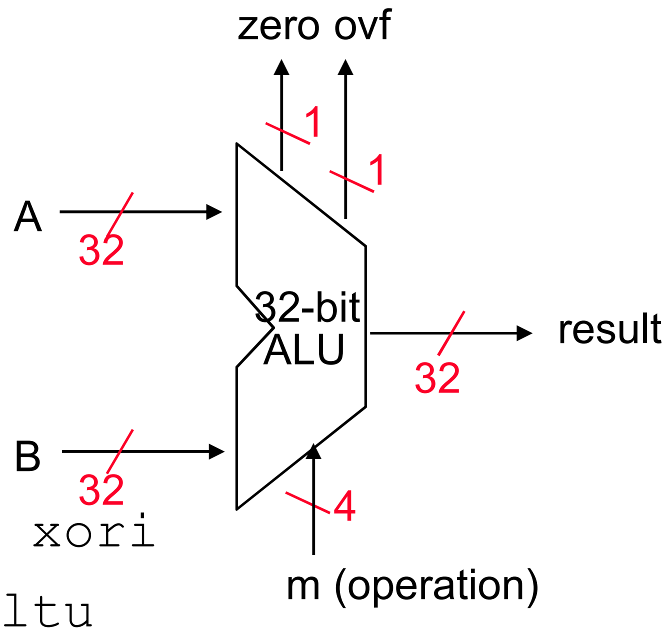
  `sub, subu`

  `mult, multu, div, divu`

  `sqrt`

  `and, andi, nor, or, ori, xor, xori`

  `beq, bne, slt, slti, sltiu, sltu`



- With special handling for
  - sign extend – `addi, addiu, slti, sltiu`
  - zero extend – `andi, ori, xori`
  - overflow detection – `add, addi, sub`
  - overflow detection in software – `mult, div`

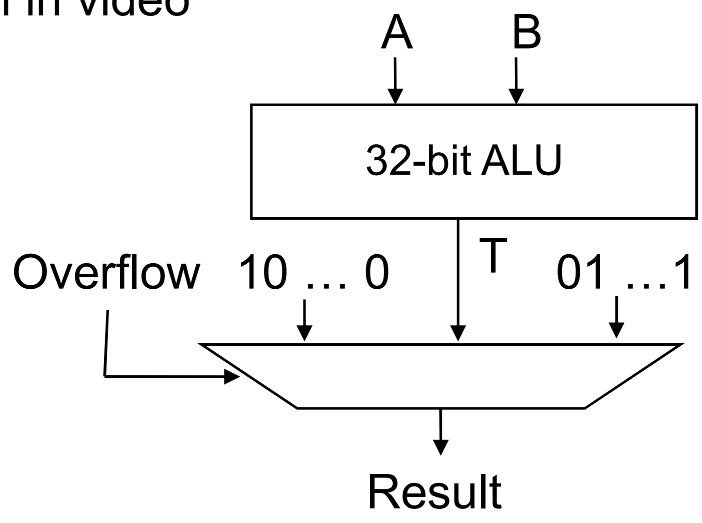
Arithmetic for Multimedia Operations

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use our 32-bit adder, with a “partitioned “ carry chain
    - Operate on 4×8-bit, 2×16-bit, or 1×32-bit vectors
  - SIMD (single-instruction, multiple-data), data-level parallelism

- Saturating operations
  - On overflow, result is largest representable signed value
    - E.g., clipping in audio, saturation in video

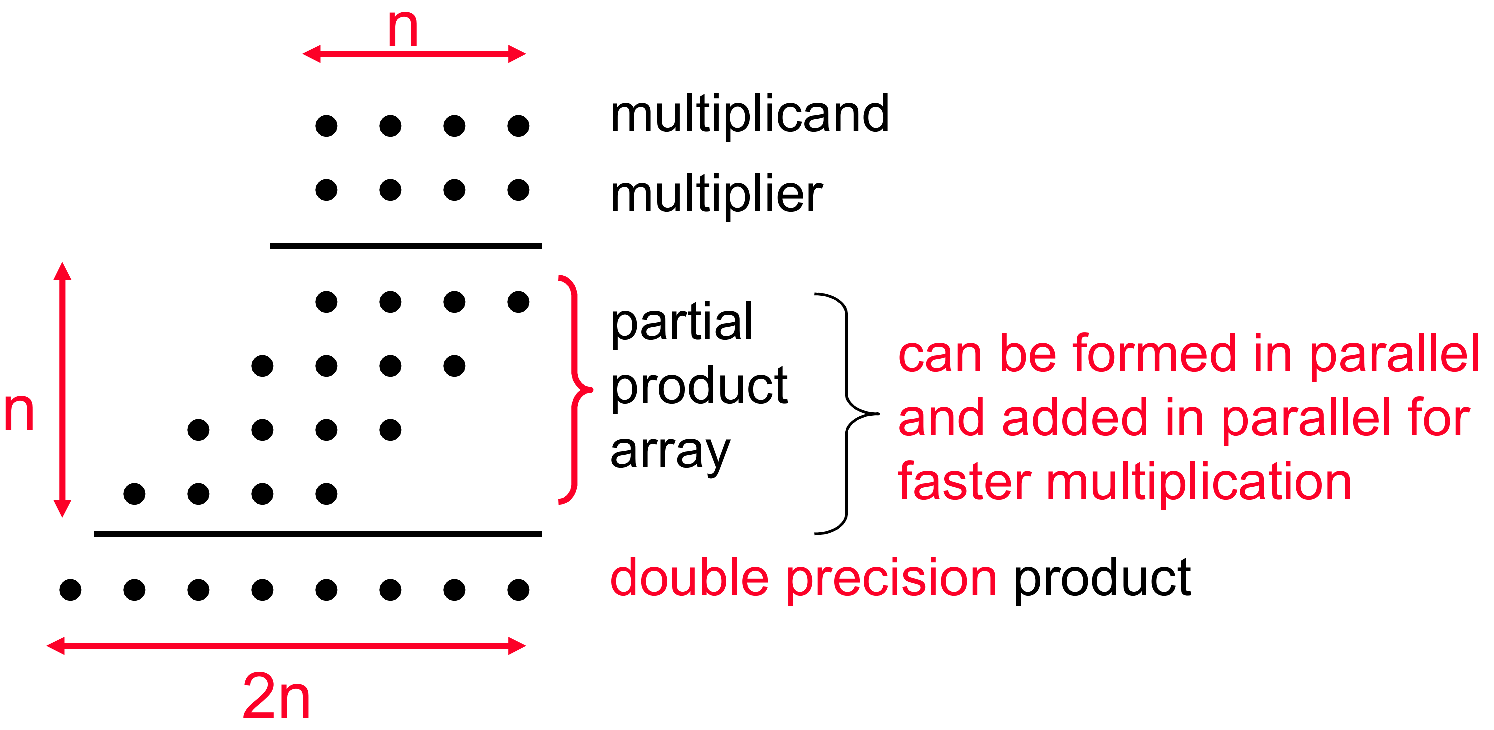
$$\text{Overflow}_{pos} = T_{31} \ \& \ !A_{31} \ \& \ !B_{31}$$

$$\text{Overflow}_{neg} = !T_{31} \ \& \ A_{31} \ \& \ B_{31}$$



Multiply

- Binary multiplication is just a *bunch* of right shifts and adds



"If war were arithmetic, the mathematicians would rule the world."

MIPS Multiply Instruction

- Multiply (`mult` and `multu`) produces a double precision product

```
mult    $s0, $s1        # hi||lo = $s0 * $s1
```

0	16	17	0	0	0x18
alu	$s0	$s1	unused	unused	mult

  - Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

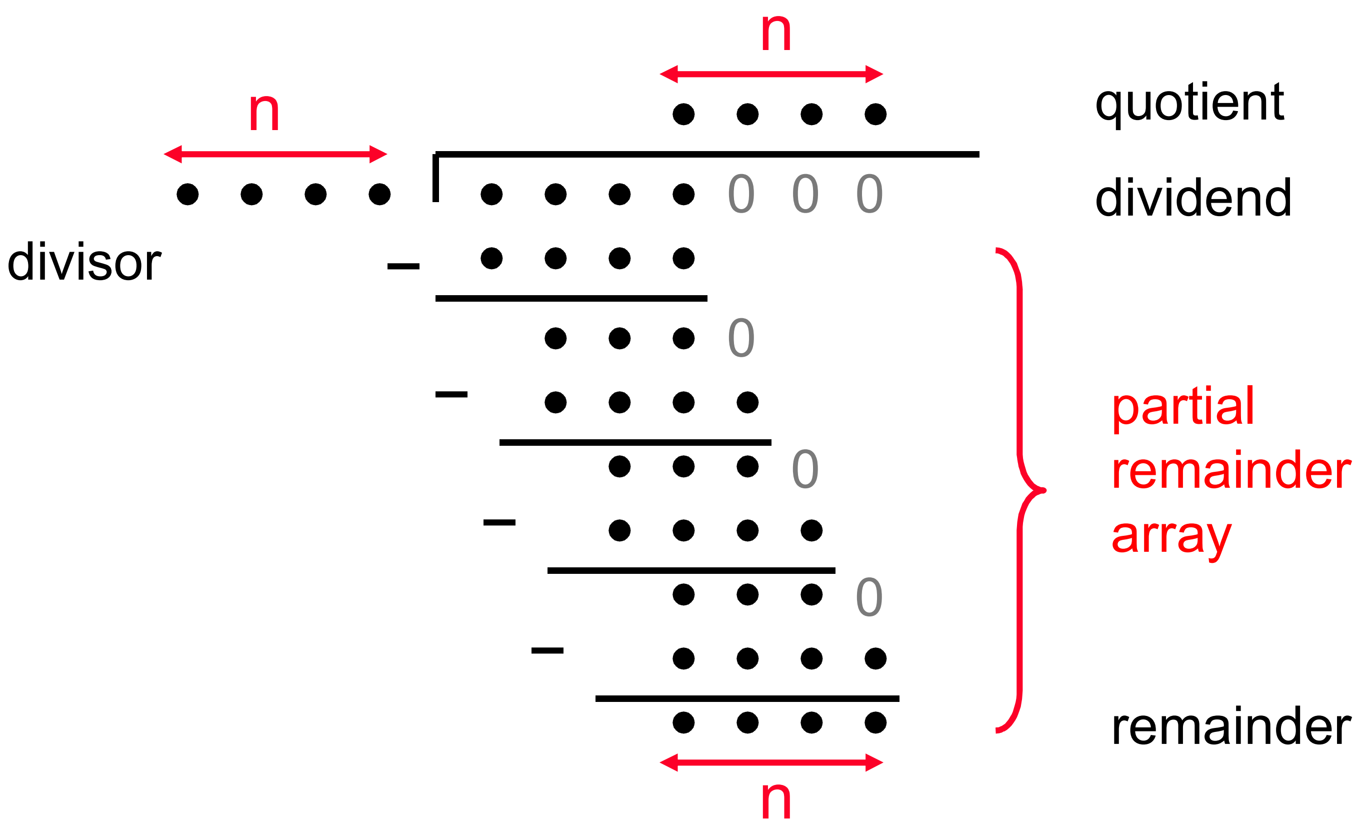
- What is the speed of our add and right shift (serial) 4-bit multiplier assuming a RCA is used?

- Thus, multiplies are usually done by fast, dedicated hardware and are much more complex (and slower) than adders

Division

- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts

dividend = quotient x divisor + remainder



MIPS Divide Instruction

- Divide (`div` and `divu`) generates the reminder in `hi` and the quotient in `lo`

```
div    $s0, $s1          # lo = $s0 / $s1
                         # hi = $s0 mod $s1
```

0	16	17	0	0	0x1A
alu	$s0	$s1	unused	unused	div

- Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- Speed? Hardware costs?

- As with multiply, divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.

Representing Big (and Small) Numbers

- What if we want to encode the approx. age of the earth?

  4,600,000,000 or 4.6×10^9

  or the weight in kg of one a.m.u. (atomic mass unit)

  0.00000000000000000000000000166 or 1.6×10^{-27}

  There is no way* we can encode either of the above in a 32-bit integer.

- Floating point representation $(-1)^{\text{sign}} \times F \times 2^E$
  - **Still** have to fit everything in 32 bits (single precision)

s	E (exponent)	F (fraction)
1 bit	8 bits	23 bits

  - The base (2, *not* 10) is **hardwired** in the design of the FPALU
  - More bits in the fraction (F) or the exponent (E) is a trade-off between precision (roughly, the accuracy of the number) and range (roughly, the magnitude of the number)

Aside: Precision vs. Accuracy

- Precision and accuracy are not the same concept.
- Precision is a quality of one number on its own – how many bits or digits do we retain?
  - How many do you trust? is a question for data collection, statistical analysis, numerical analysis, and hardware design.
- Accuracy is a quality of two numbers – how close "right" are they?
  - When discussing accuracy, one of the two numbers is assumed to be "right". For example, 4.5678 is precise to 5 digits, but as an approximation to π it is only accurate to one-half digit, since (4.5678 – 3.1416) / 3.1416 = 0.45397
  - When discussing precision, you could say that a number is accurate to within ½ unit in the last place. Why?

IEEE 754 FP Standard

s	E (exponent)	F (fraction)
1 bit	8 bits	23 bits

- Most (all?) computers these days conform to the IEEE 754 floating point standard $(-1)^{sign} \times (1+F) \times 2^{E-bias}$
  - With formats for both single and double precision
  - F is stored in normalized sign magnitude format
    - **Normalized means that before the result is put into the FPRF, F is shifted (left) and E decremented (once for every left bit shift) until the msb of F is a 1 (so there is no need to store it!) – called the hidden bit**
    - **So when a value is read from the FPRF, the hidden bit of 1 has to be restored before performing computation**
  - To simplify sorting FP numbers, E comes before F in the word and E is represented in excess (biased) notation where the bias is 127 (1023 for double precision) so the *most negative* exponent is **0**0000001 = 2^{1-127} = 2^{-126} and the *most positive* exponent is **1**1111110 = $2^{254-127}$ = 2^{+127} (exponents 00000000 and 11111111 are reserved for special uses)

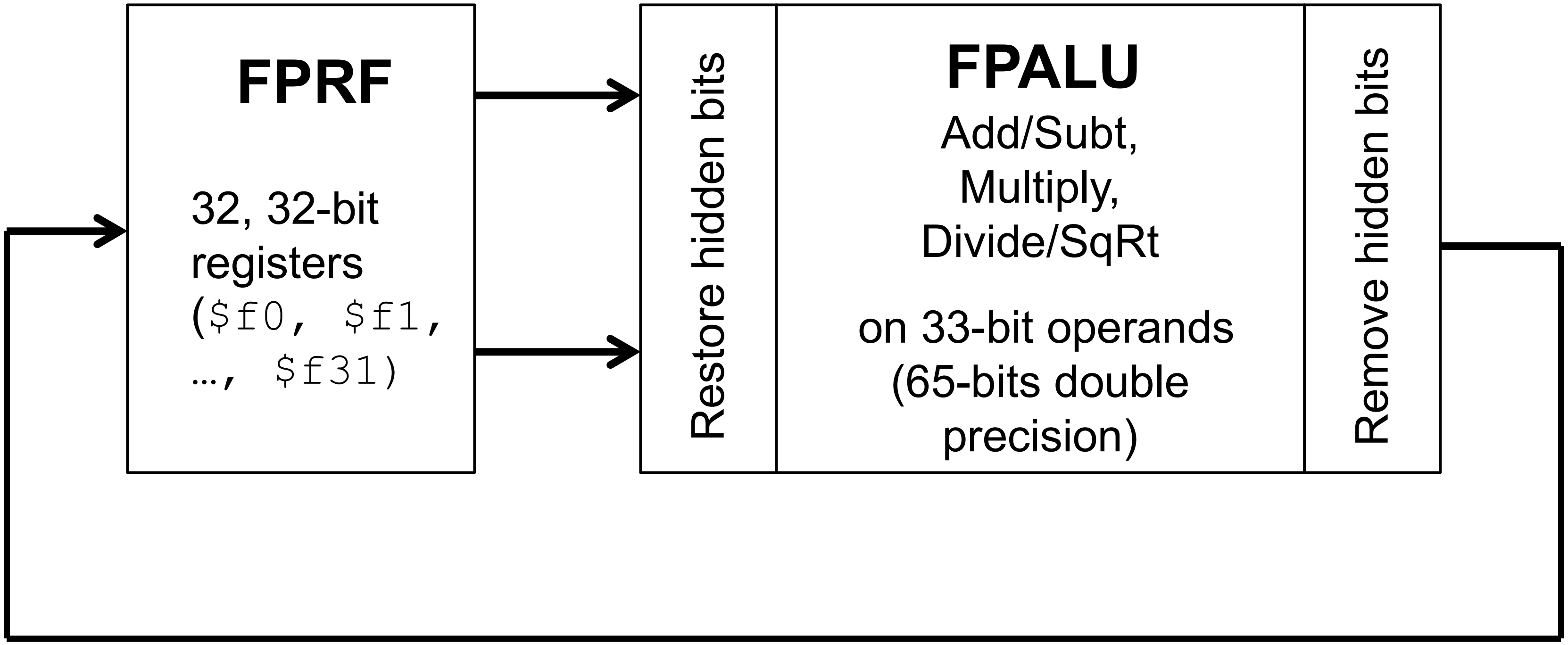
IEEE 754 FP Standard Encoding

- Special encodings are used to represent unusual events
  - ± infinity for division by zero
  - NAN (not a number) for the results of invalid operations such as 0/0
  - True zero is the bit string all zero

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0000 0000	0	0000 … 0000	0	true zero (0)
0000 0000	nonzero	0000 … 0000	nonzero	± denormalized number
0000 0001 to 1111 1110	anything	0000 … 0001 to 1111 … 1110	anything	± floating point number
1111 1111	± 0	1111 … 1111	± 0	± infinity
1111 1111	nonzero	1111 … 1111	nonzero	not a number (NaN)

Floating Point Arithmetic Hardware

- Floating Point Register File (FPRF) and floating point ALU (FPALU)



MIPS Floating Point Instructions

- MIPS has a separate Floating Point RF (`$f0, $f1, …, $f31`) (whose registers are used in *pairs* for double precision values) with special instructions to load to and store from them

```
lwcl  $f1,54($s2)   #$f1 = Memory[$s2+54]
swcl  $f1,58($s4)   #Memory[$s4+58] = $f1
```

- And supports IEEE 754 single

```
add.s $f2,$f4,$f6  #$f2 = $f4 + $f6
```

and double precision operations

```
add.d $f2,$f4,$f6  #$f2||$f3 =
                          $f4||$f5 + $f6||$f7
```

similarly for `sub.s, sub.d, mul.s, mul.d, div.s, div.d`

MIPS Floating Point Instructions, Con’t

- And floating point single precision comparison operations

```
c.x.s $f2,$f4       #if($f2 < $f4) cond=1;
                        else cond=0
```

where `x` may be `eq, neq, lt, le, gt, ge`

and double precision comparison operations

```
c.x.d $f2,$f4      #$f2||$f3 < $f4||$f5
             cond=1; else cond=0
```

- And floating point branch operations

```
bclt  25             #if(cond==1)
   go to PC+4+25
bclf  25      #if(cond==0)
   go to PC+4+25
```

Frequency of Common MIPS Instructions

- Only included those with >3% and >1%

	SPECint	SPECfp
addu	5.2%	3.5%
addiu	9.0%	7.2%
or	4.0%	1.2%
sll	4.4%	1.9%
lui	3.3%	0.5%
lw	18.6%	5.8%
sw	7.6%	2.0%
lbu	3.7%	0.1%
beq	8.6%	2.2%
bne	8.4%	1.4%
slt	9.9%	2.3%
slti	3.1%	0.3%
sltu	3.4%	0.8%

	SPECint	SPECfp
add.d	0.0%	10.6%
sub.d	0.0%	4.9%
mul.d	0.0%	15.0%
add.s	0.0%	1.5%
sub.s	0.0%	1.8%
mul.s	0.0%	2.4%
l.d	0.0%	17.5%
s.d	0.0%	4.9%
l.s	0.0%	4.2%
s.s	0.0%	1.1%
lhu	1.3%	0.0%

Pitfalls and Fallacies

- *Fallacy*: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as integer division by a power of 2.
  - True for unsigned integers, not true for 2's complement integers
    - E.g., -5 ÷ 4 → 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
- *Pitfall*: Floating-point addition is not associative.
  - Does c + (a + b) = (c + a) + b ?
    - If c = $-1.5_{10} \times 10^{38}$, a = $1.5_{10} \times 10^{38}$, and b = 1, then NO - left way gives 0.0 and right way gives 1.0 due to limited precision limitations
- *Fallacy*: Parallel execution strategies that work for integer data types also work for FP data types
  - *Order* of arithmetic operations is important in FP (see Pitfall above)
    - For more take CmpSc 451, 454, 455

CMPEN 431
Computer Architecture
Fall 2018

Review Slide Deck Part 2

Focus: Datapath & Memory Hierarchy

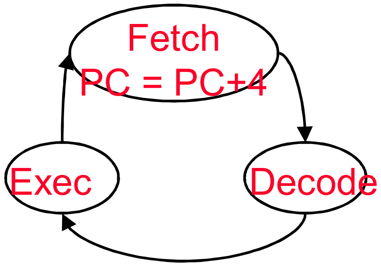
Jack Sampson(www.cse.psu.edu/~sampson)

[Slides adapted from work by Mary Jane Irwin, in turn adapted from *Computer Organization and Design, Revised 4th Edition*,

Patterson & Hennessy, © 2011, Morgan Kaufmann and 5th edition, © 2014]

The Processor: Datapath & Control

- Our implementation of the MIPS ISA is simplified
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction

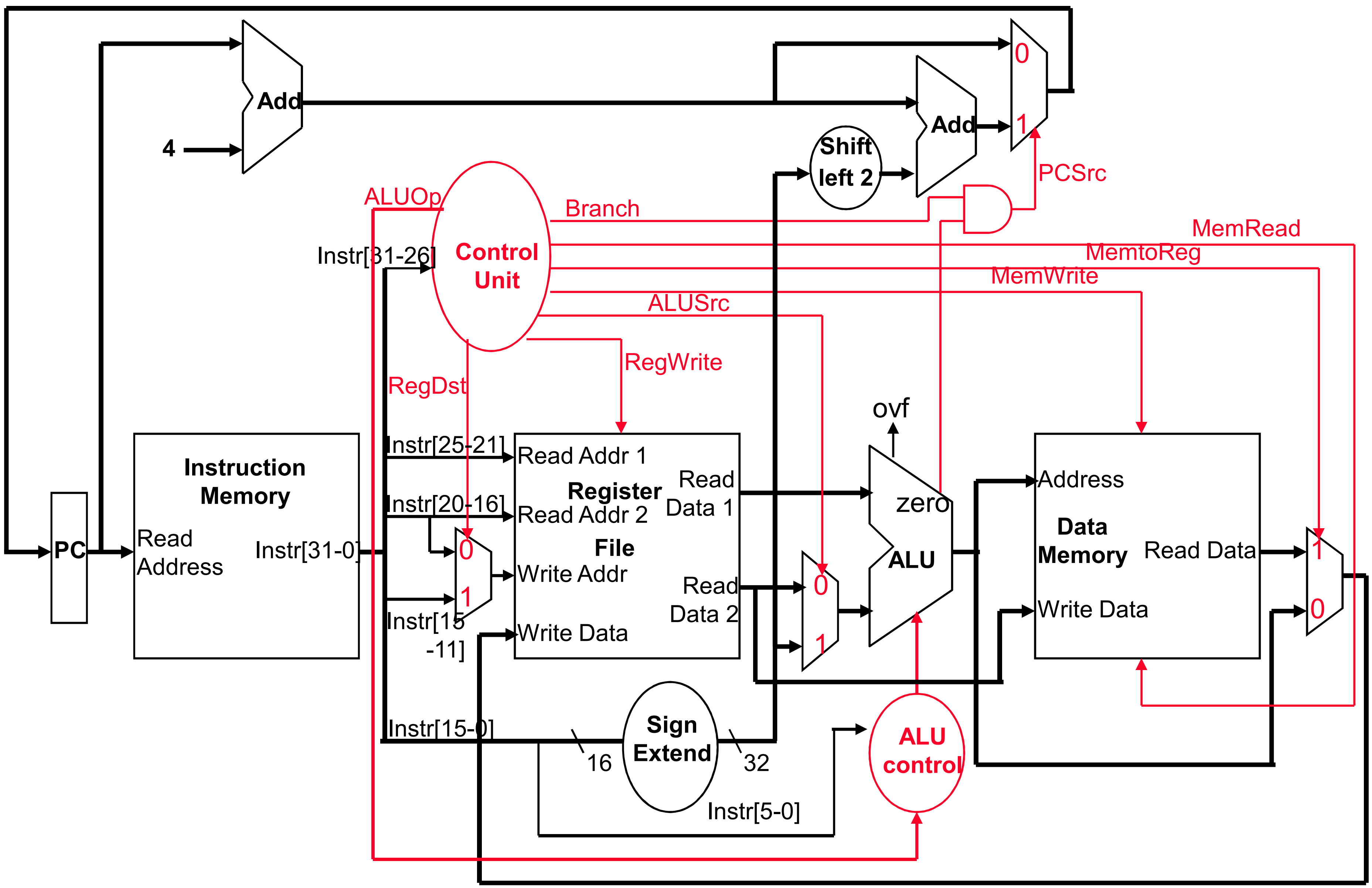


- All (integer) instructions (except `j`) use the (integer) ALU after reading the registers

Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the RF and Data Memory
- Cycle time is determined by length of the longest path

Single Cycle Datapath with Control Unit



Instruction Critical Paths

What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:

- Instruction and Data Memory (200 ps)
- ALU and adders (200 ps)
- Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500
jump	200					200

How Can We Make It Faster?

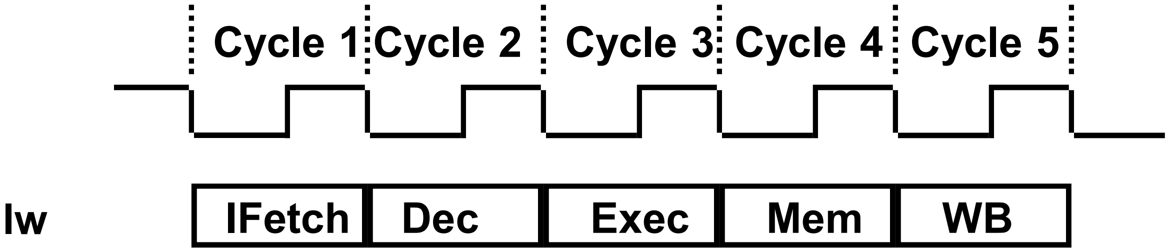
- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – (all?) modern processors are pipelined for performance
  - Remember *the* performance equation:

CPU time = CPI * CC * IC

- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
  - A five stage pipeline is nearly five times faster because the CC is nearly five times faster

- Fetch (and execute) more than one instruction at a time
  - Superscalar processing – stay tuned

The Five Stages of Load Instruction



IFetch: Instruction Fetch and Update PC

Dec: Registers Fetch and Instruction Decode

Exec: Execute R-type; calculate memory address

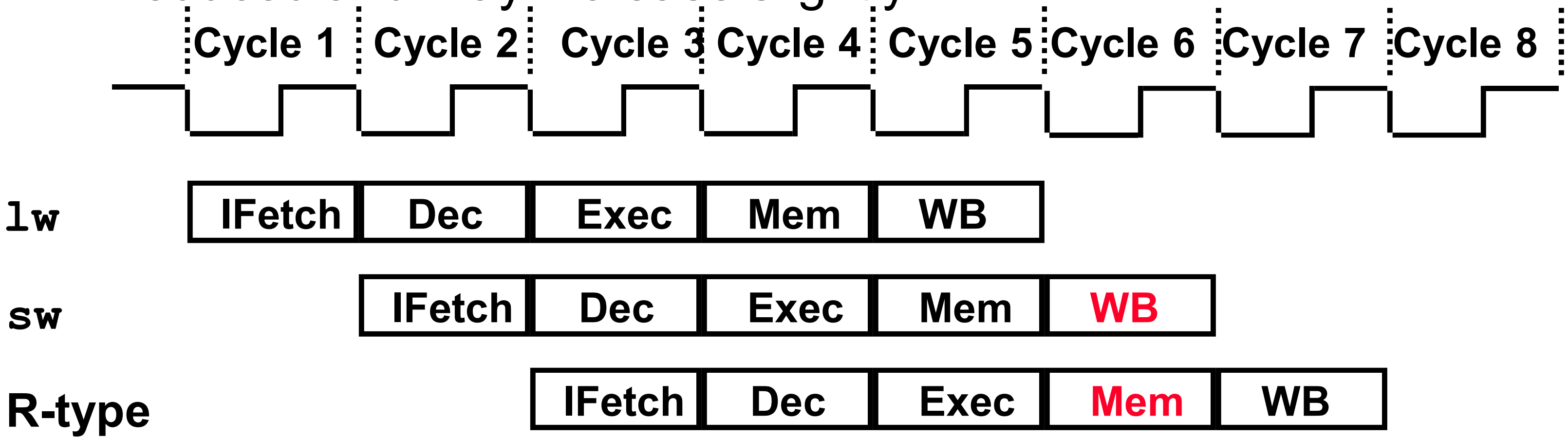
Mem: Read/write the data from/to the Data Memory

WB: Write the result data into the register file

- Single cycle – each stage is used once in each cycle
  - One active instruction per cycle (a looonnng cycle)
- Pipelined – each stage is used in each cycle
  - Multiple active instructions per cycle (a short cycle)

A Pipelined MIPS Processor

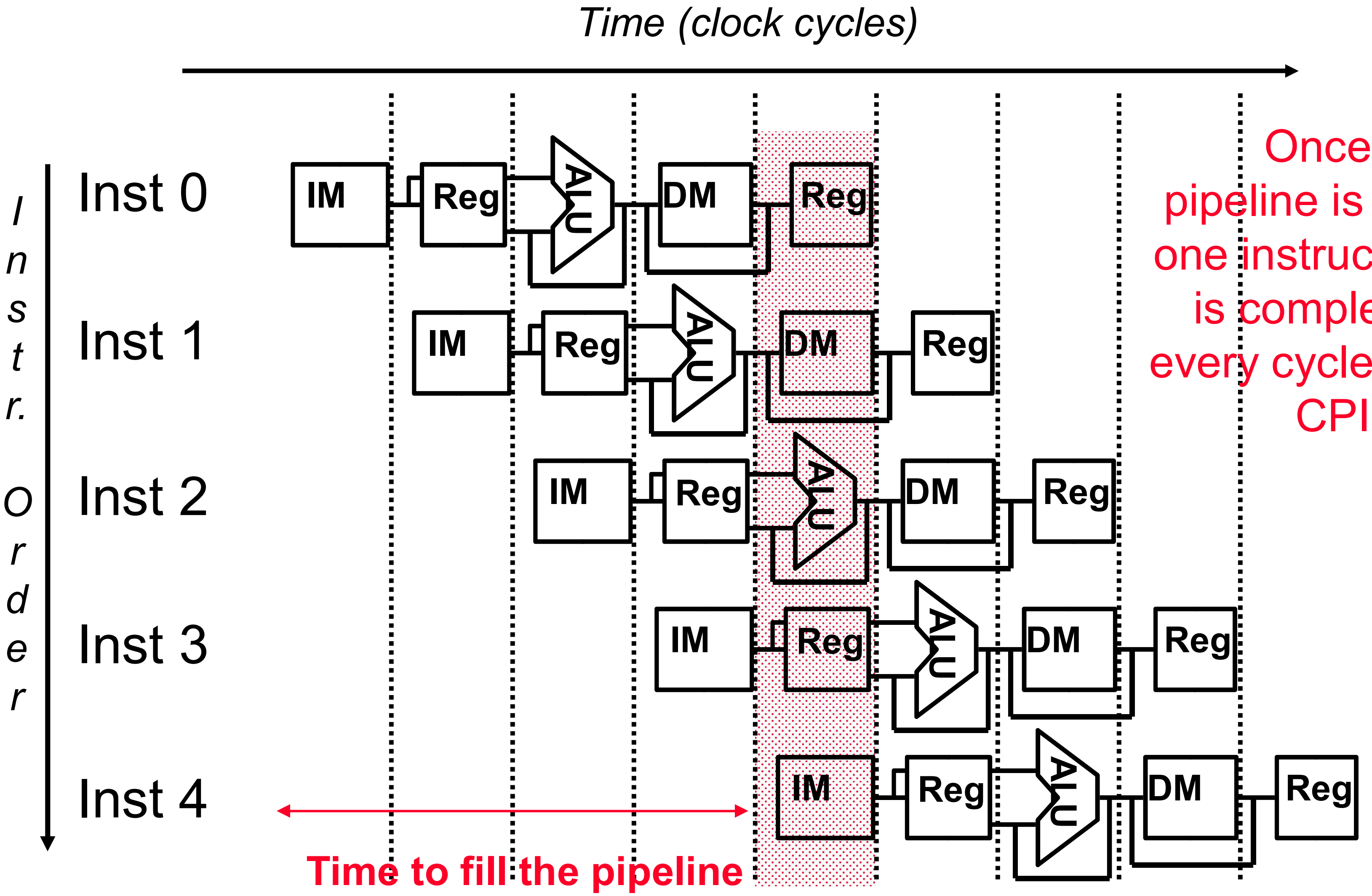
- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced and may increase slightly



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
`lw`	IFetch	Dec	Exec	Mem	WB			
`sw`		IFetch	Dec	Exec	Mem	WB		
R-type			IFetch	Dec	Exec	Mem	WB	

- clock cycle (pipeline stage time) is limited by the **slowest** stage
  - for some stages don't need the whole clock cycle (e.g., WB)
- for some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction)

Why Pipeline? For Performance!



Once the pipeline is full, one instruction is completed every cycle, so CPI = 1

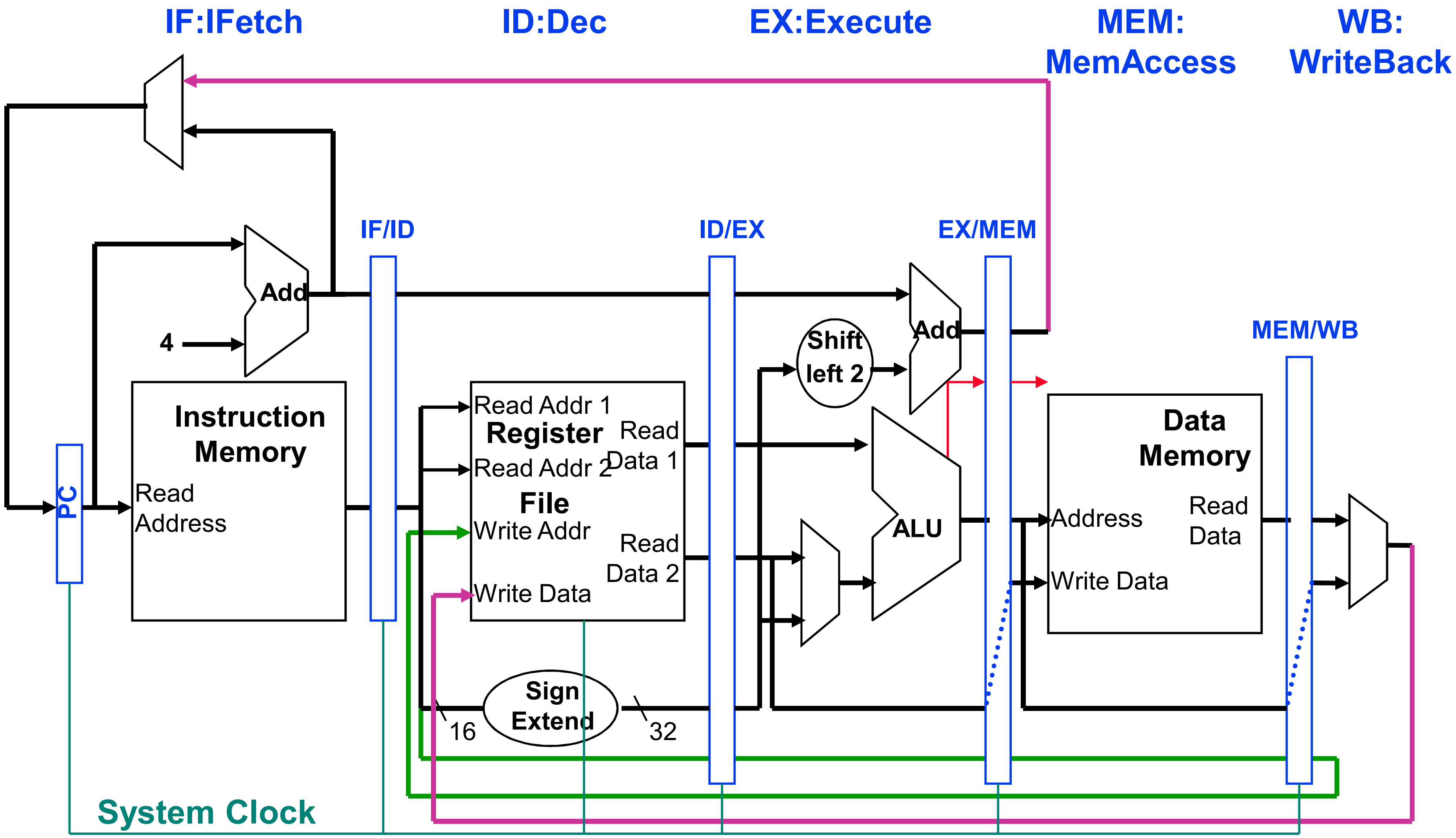
Time to fill the pipeline

Pipelining the MIPS ISA – What Makes It Easy?

- All instructions are the same length (32 bits)
  - can fetch in the 1st stage and decode in the 2nd stage
- Only a few instruction formats (three) with symmetry across formats
  - can begin reading register file in 2nd stage (before instruction is fully decoded) even if it turns out we don’t need it
- Memory operations occur only in loads and stores
  - can use the execute stage to calculate memory addresses
- Each instruction writes at most one result (i.e., changes the machine state) and does it in the last few pipeline stages (MEM or WB)
- Operands must be aligned in memory so a single data transfer takes only one data memory access

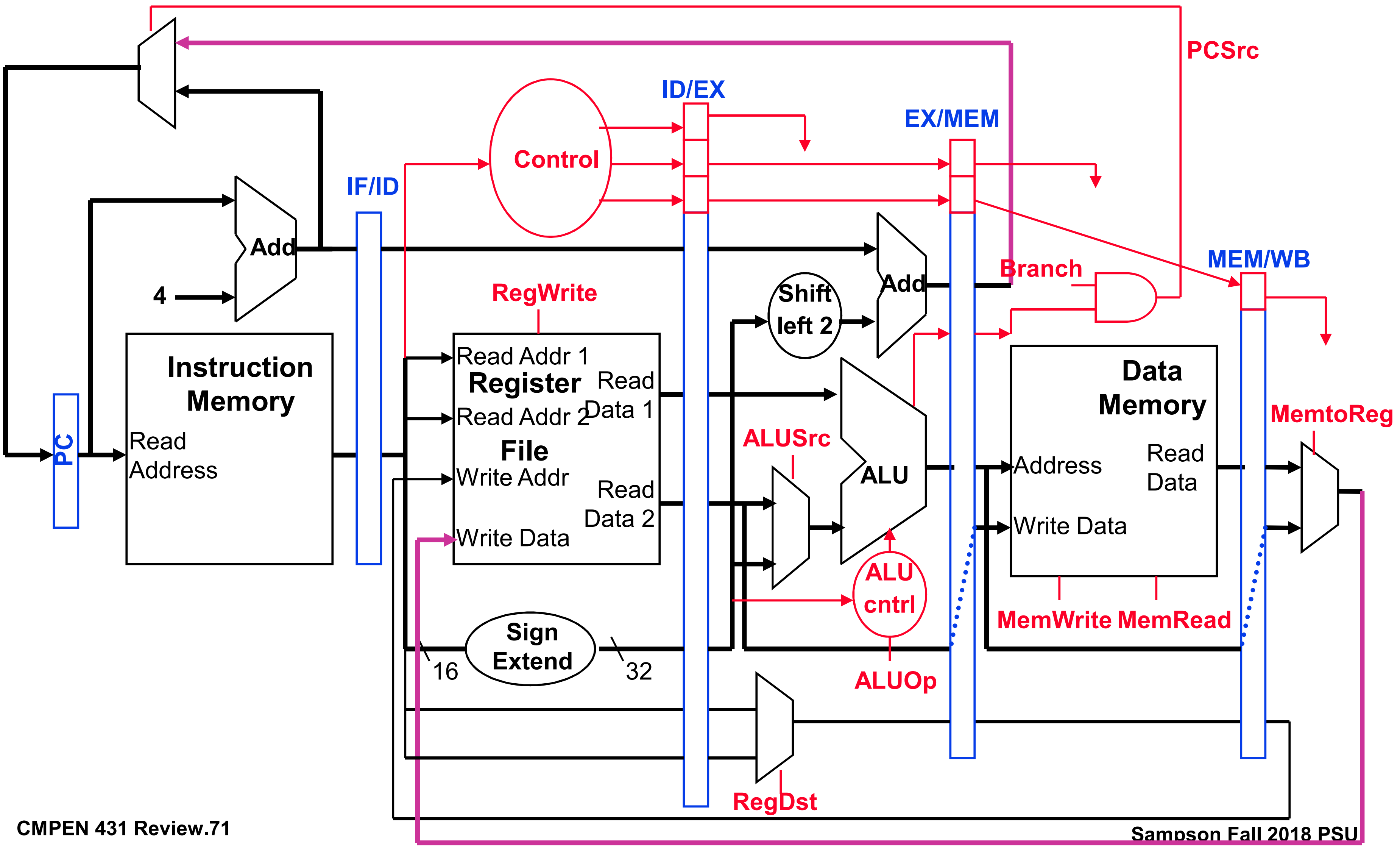
MIPS Pipeline Datapath Additions/Mods

- State registers between each pipeline stage to isolate them

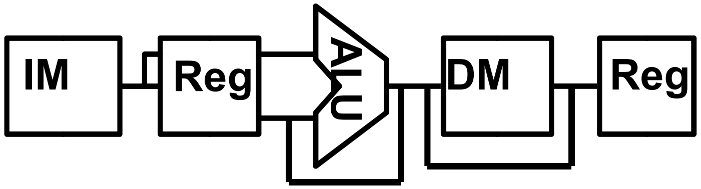


MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages



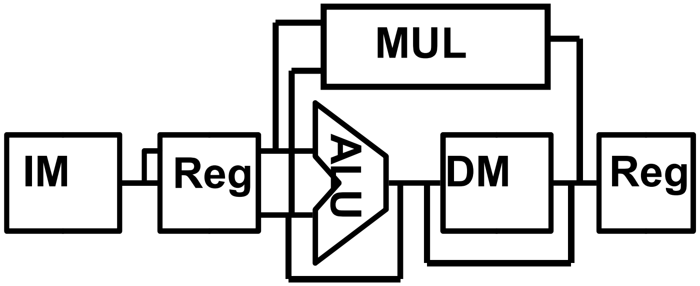
Graphically Representing MIPS Pipeline



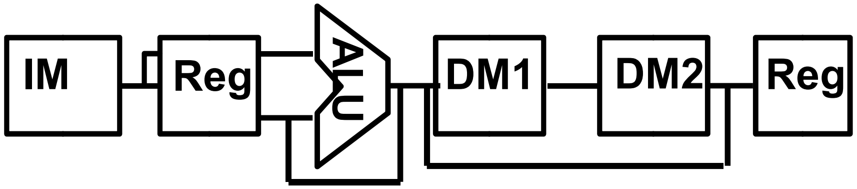
- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?

Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)

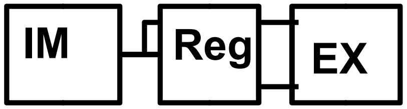


- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



Other Sample Pipeline Alternatives

- ARM7

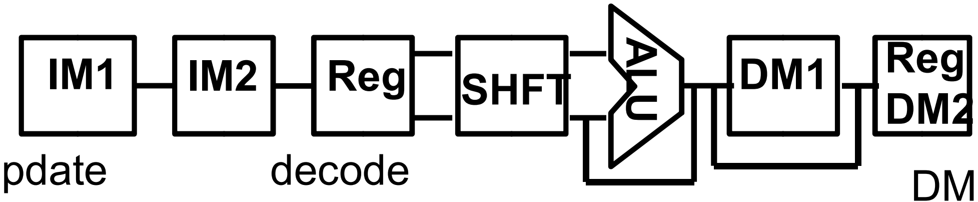


PC update
IM access

decode
reg
access

ALU op
DM access
shift/rotate
commit result
(write back)

- Intel's XScale



PC update
BTB access
start IM access

IM access

decode
reg 1 access

shift/rotate
reg 2 access

ALU op

start DM access
exception

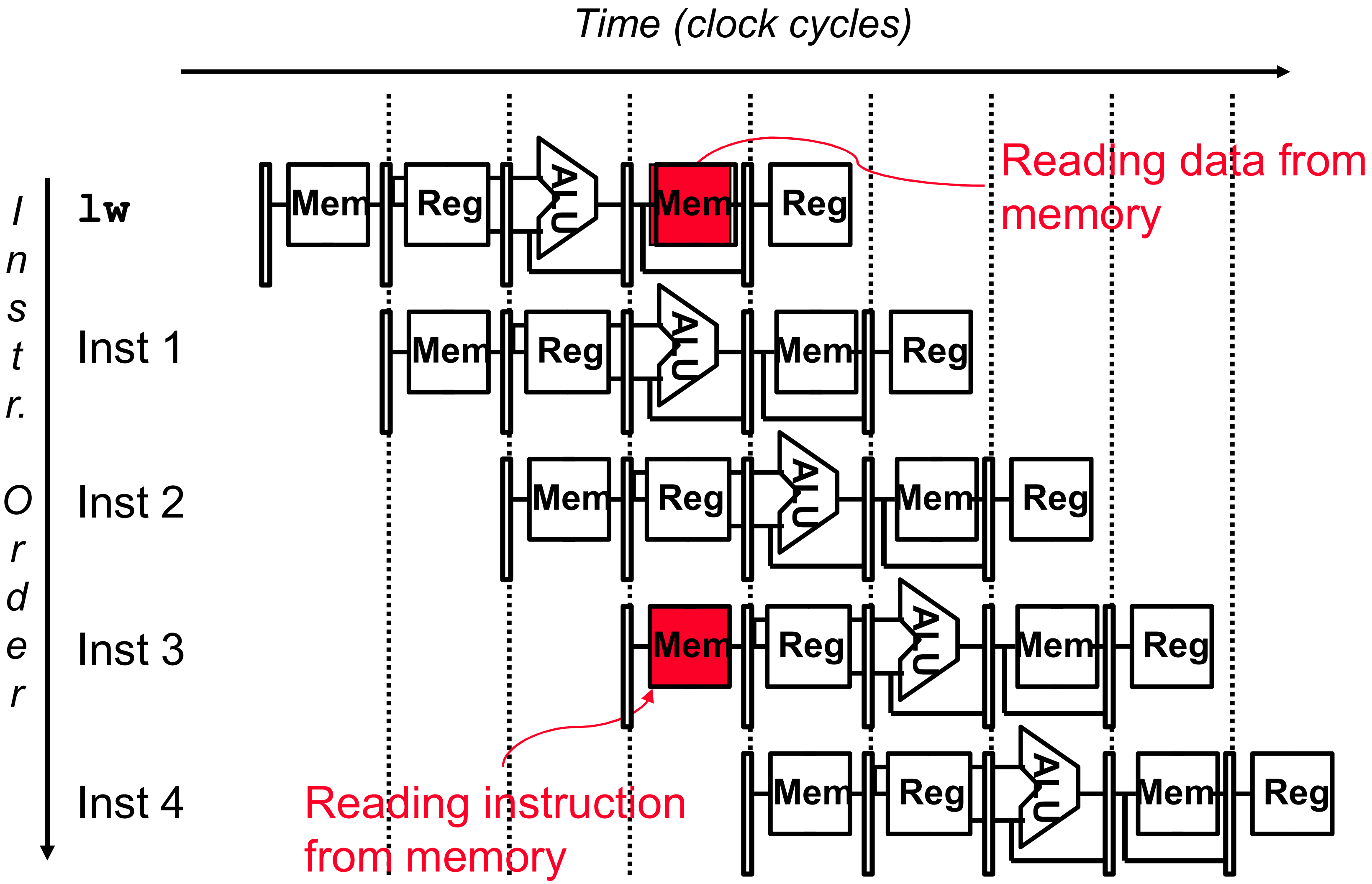
DM write
reg write

Pipelining - What Makes it Hard ?

- Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction’s source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions

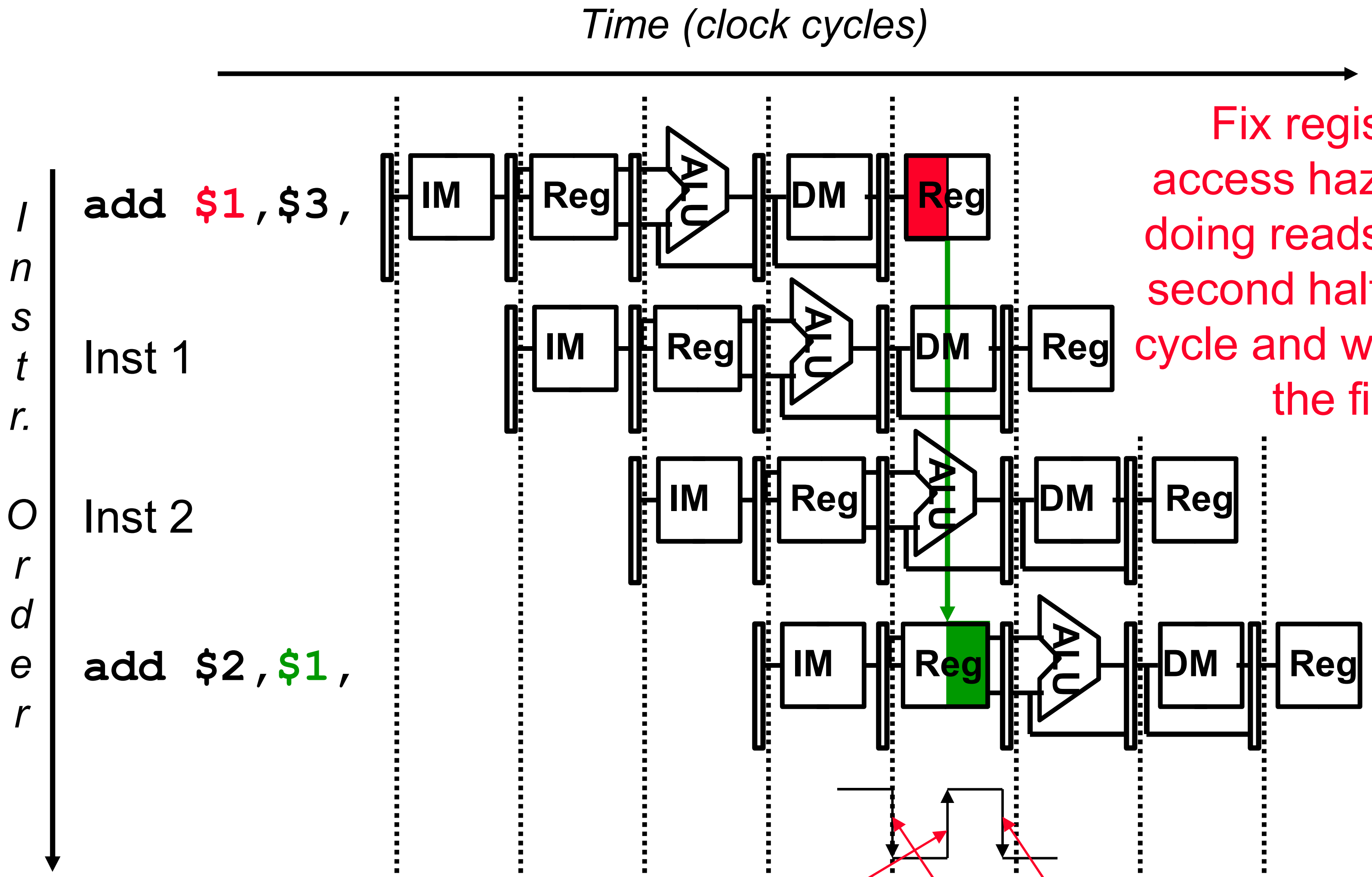
- Pipeline hardware control must detect the hazard and then take action to resolve hazard

A Single Memory Would Be a Structural Hazard



- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?



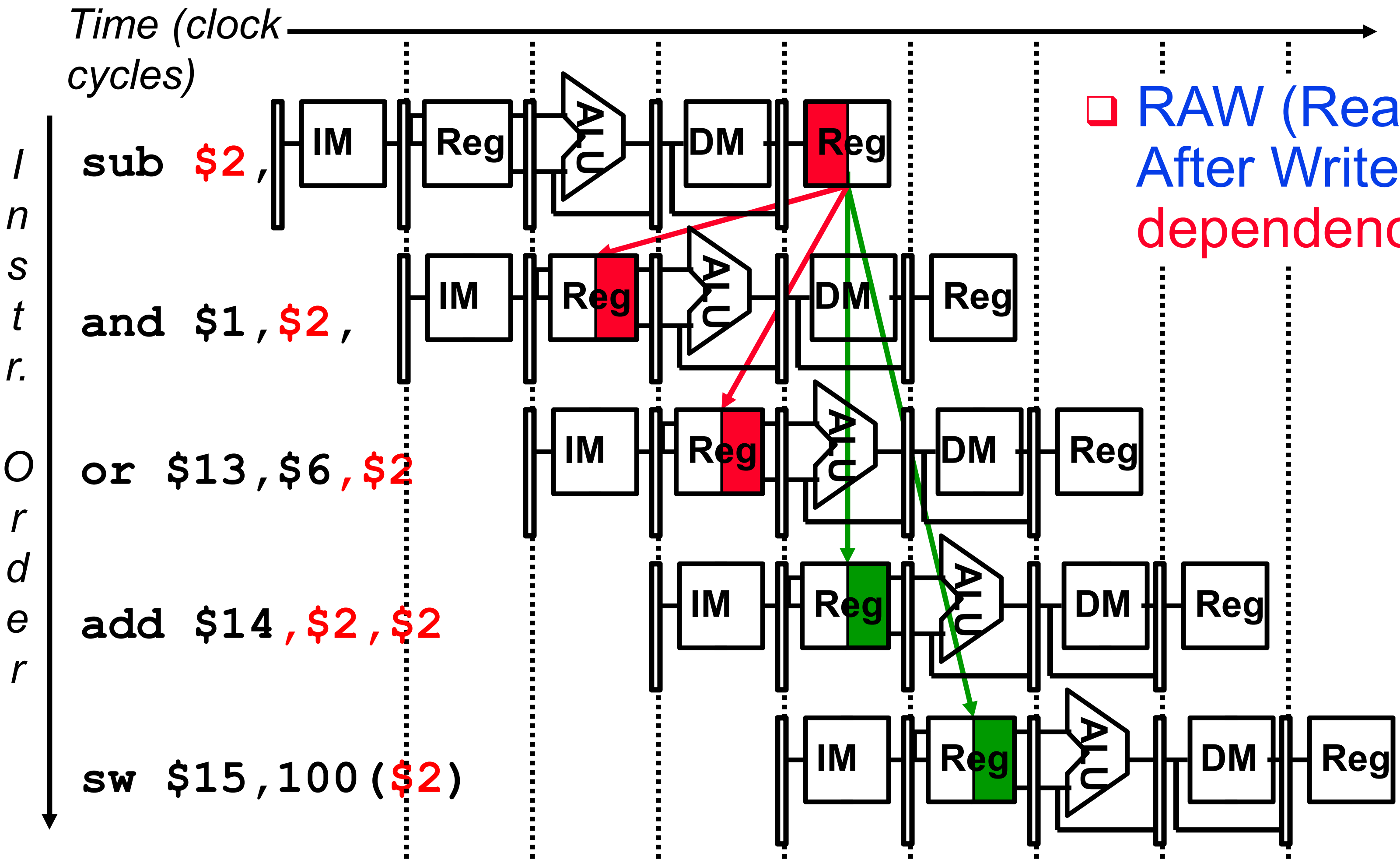
Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

clock edge that controls register writing

clock edge that controls loading of pipeline state registers

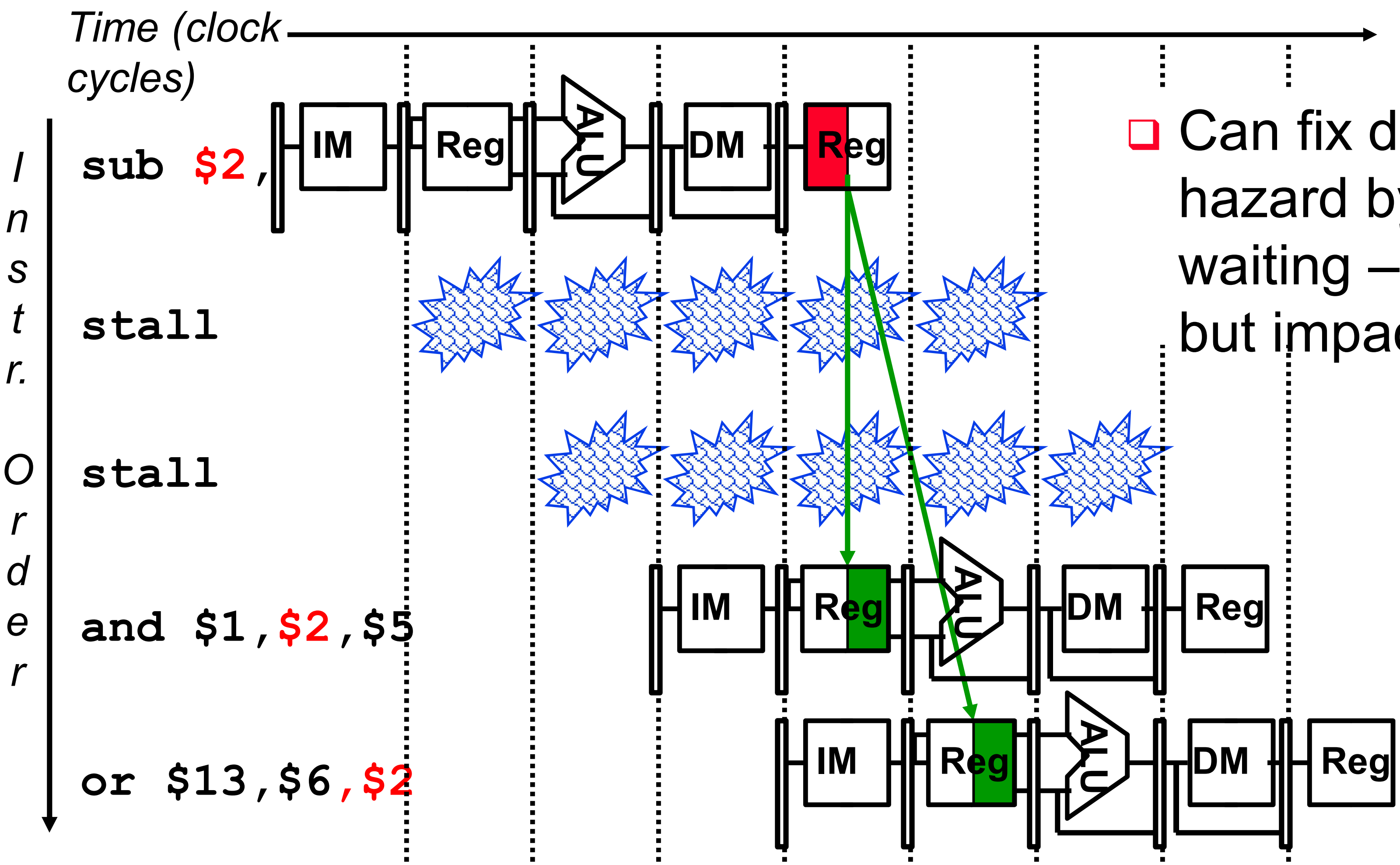
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards



- RAW (Read After Write) data dependency

One Way to “Fix” a Data Hazard



Time (clock cycles)

Instr. Order

sub $2,

stall

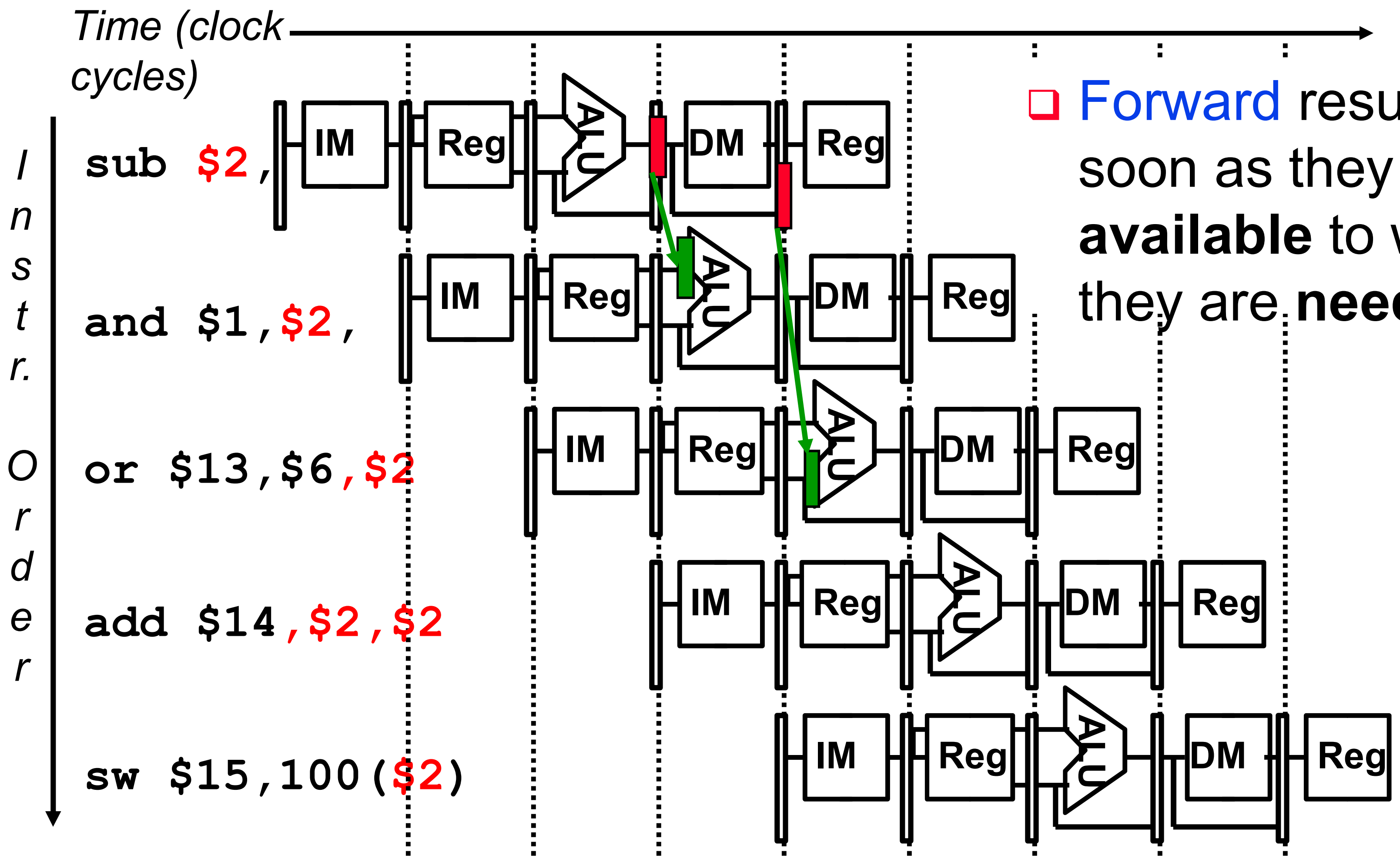
stall

and $1,$2,$5

or $13,$6,$2

- ❑ Can fix data hazard by waiting – stall – but impacts CPI

Another Way to “Fix” a Data Hazard



- Forward results as soon as they are **available** to where they are **needed**

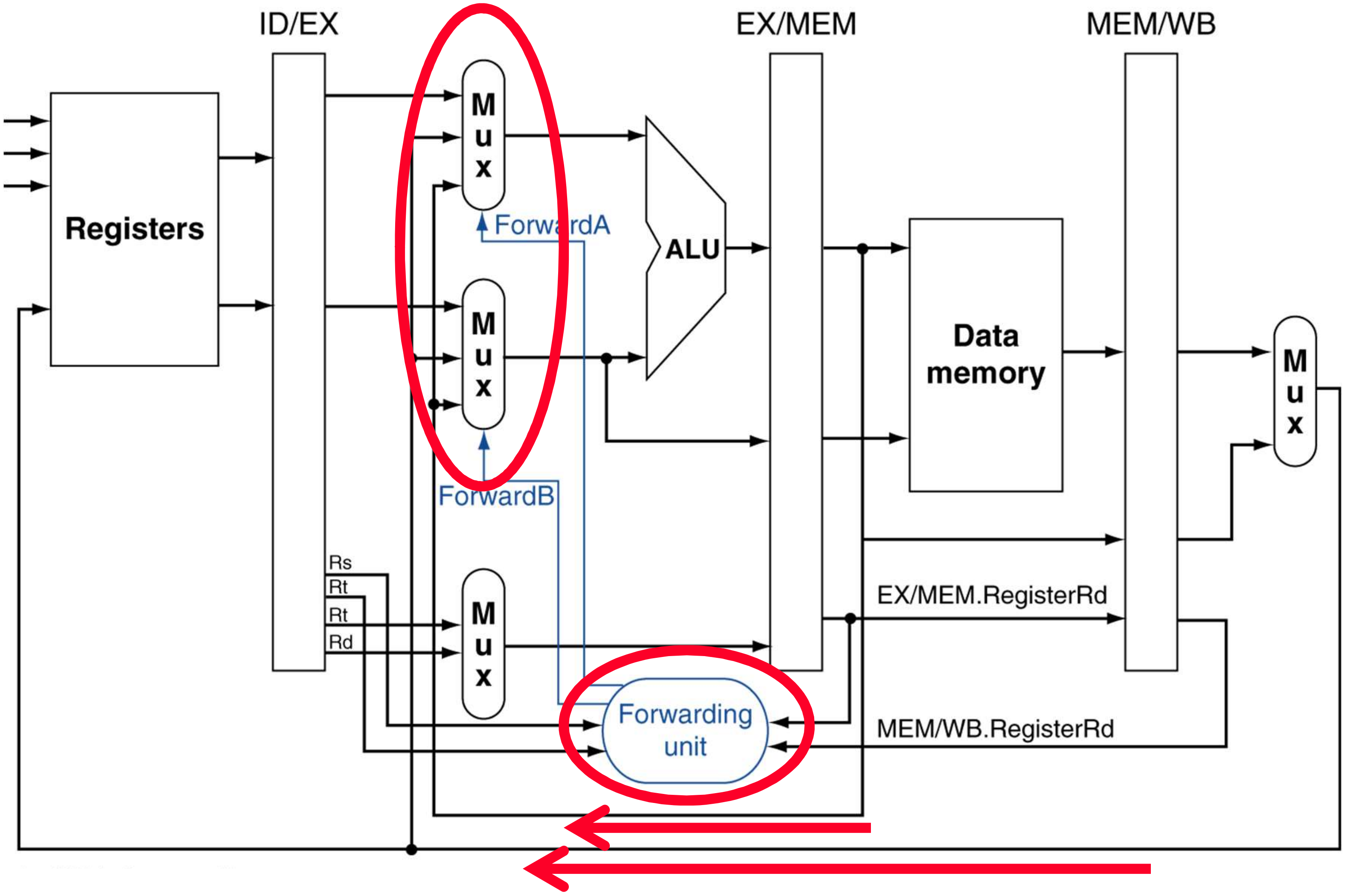
EX/MEM forwarding to ALU_A

MEM/WB forwarding to ALU_B

Data Forwarding (aka Bypassing)

- Take the result from a downstream pipeline state register that holds the needed data that cycle and forward it to the functional units (e.g., the ALU) that need that data that cycle
- For ALU functional unit: the inputs can come from other pipeline registers than just ID/EX by
  - adding multiplexors to the inputs of the ALU
  - connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX's stage Rs and Rt ALU mux inputs
  - adding the proper control hardware to control the new muxes
- Other functional units may need similar forwarding logic (e.g., the DM)
- With forwarding can achieve a CPI of 1 even in the presence of data dependencies

Forwarding Logic



Data Forwarding Control Conditions

1. EX/MEM Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

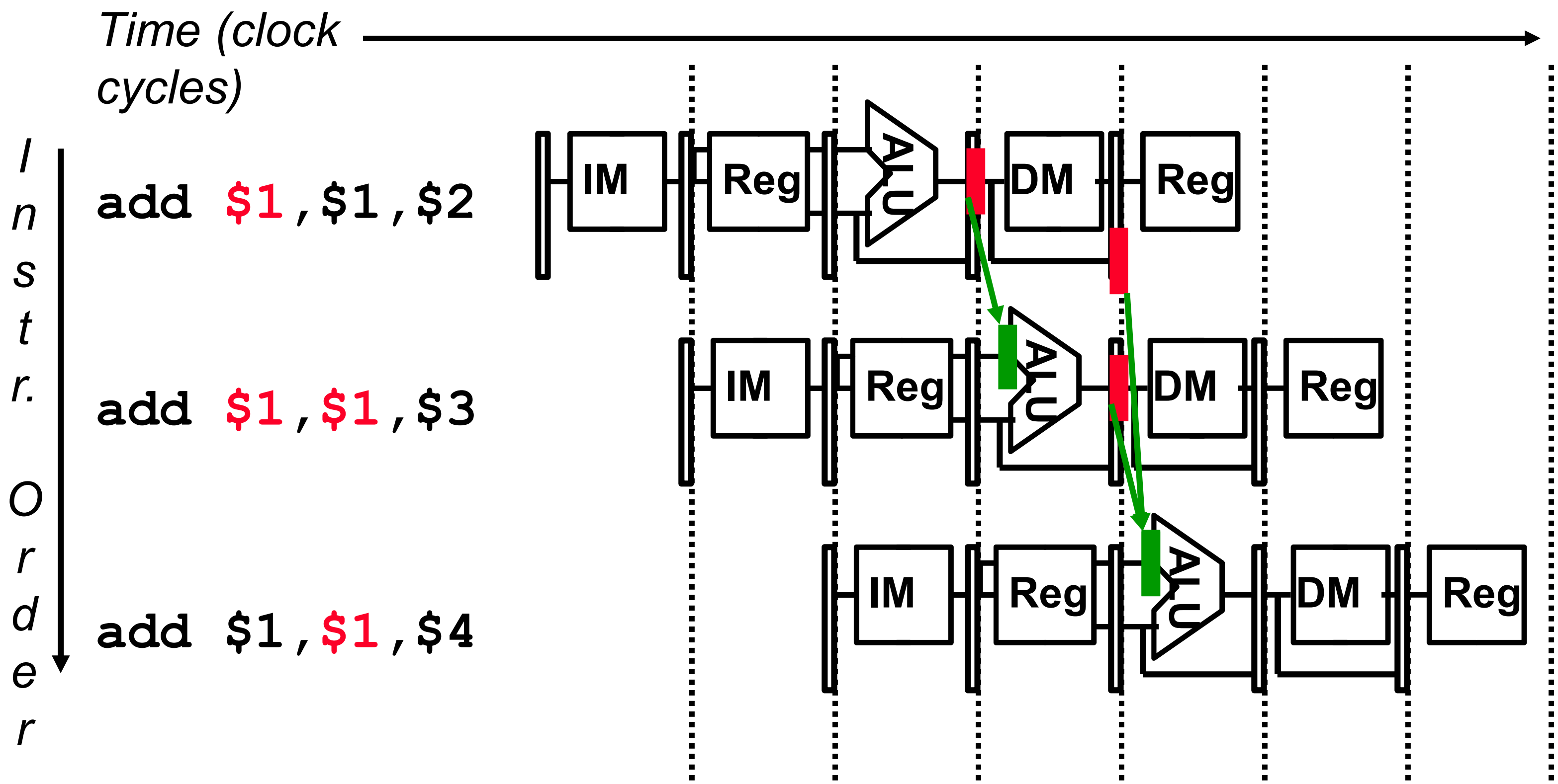
2. MEM/WB Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the second previous instr. to either input of the ALU

Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?



Time (clock cycles)

Instr. Order

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

Corrected Data Forwarding Control Conditions

1. EX/MEM Forward Unit:

```
if (EX/MEM.RegWrite
…
```

Forwards the result from the previous instr. to either input of the ALU

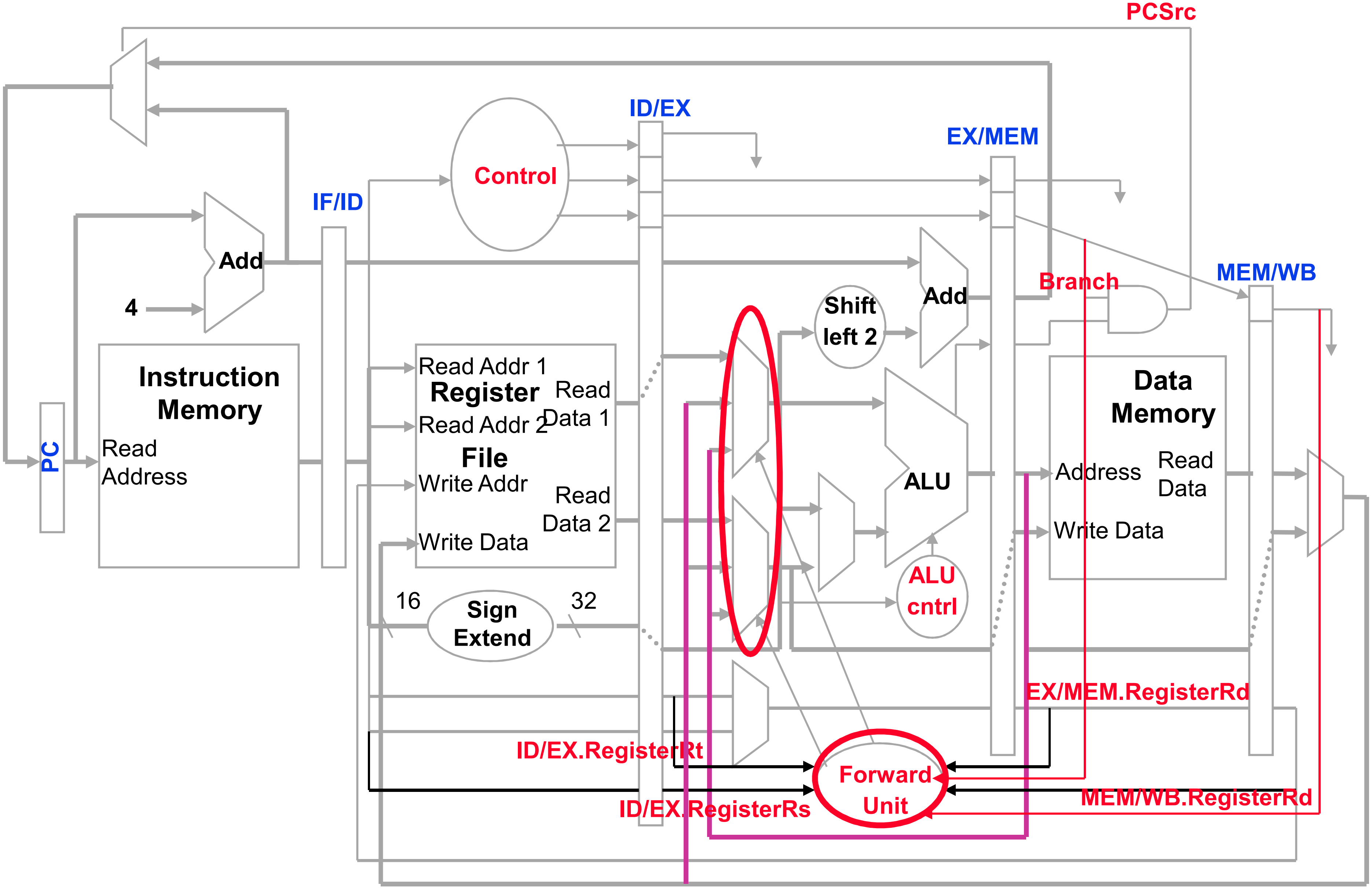
2. MEM/WB Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and !(EX/MEM.RegWrite and (EX/MEM RegisterRD != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01
```

Forwards the result from the previous or second previous instr. to either input of the ALU

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and !(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

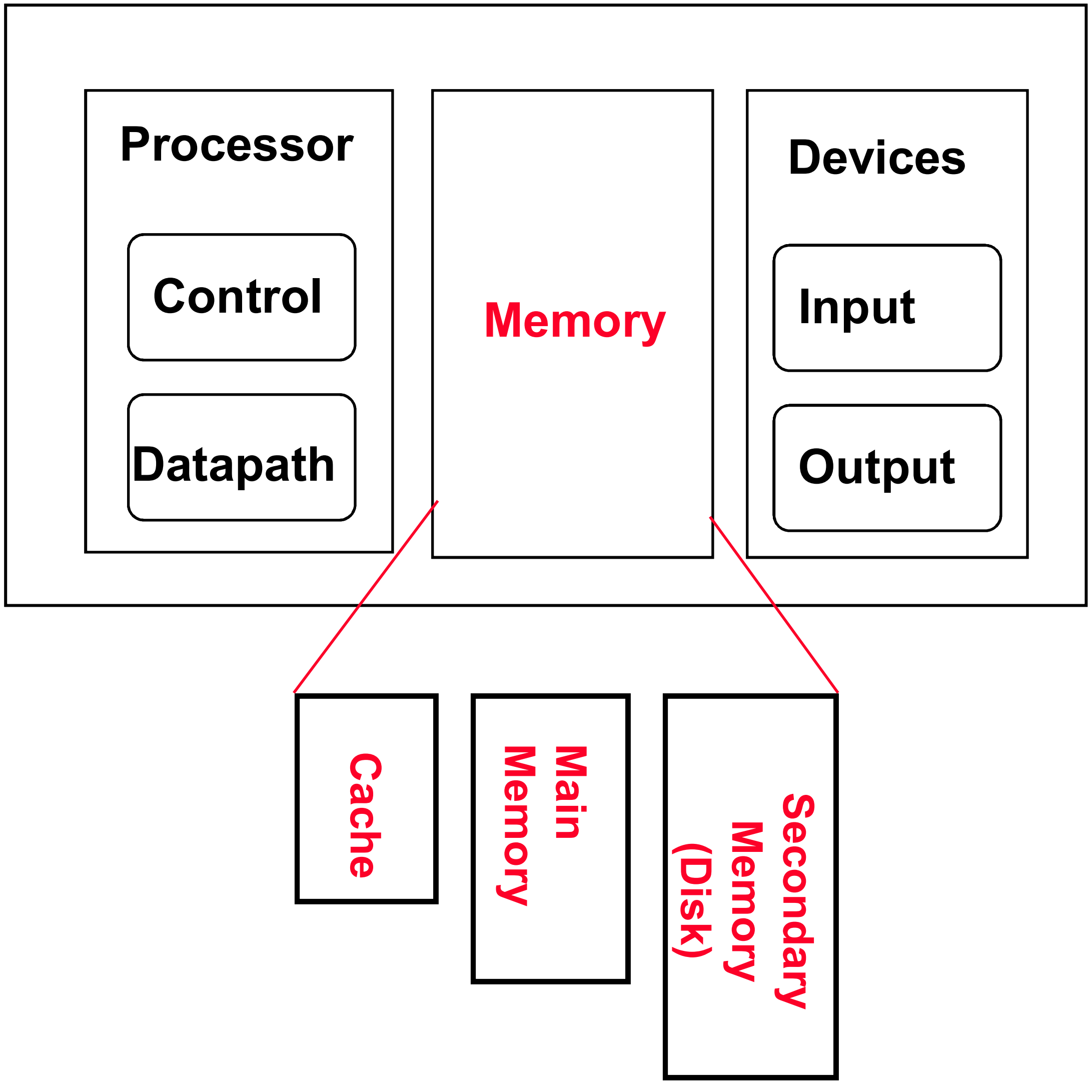
Datapath with Forwarding Hardware



Summary

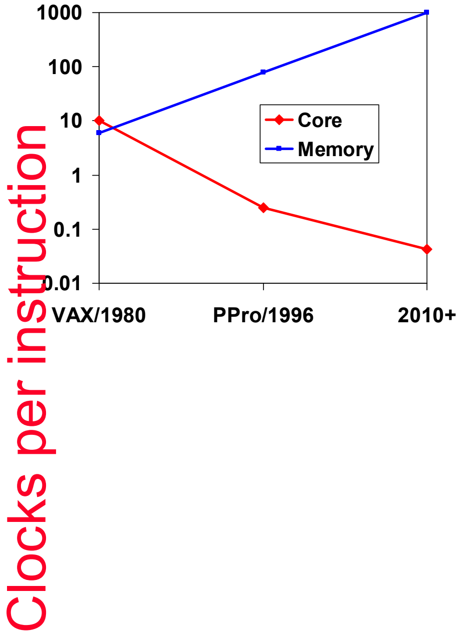
- All modern day processors use pipelining
- Pipelining doesn't help the latency of any single instruction, it helps the throughput of the entire workload
- Potential speedup: a CPI of 1 and fast a CC
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to "fill" the pipeline and the time to "drain" it can impact speedup for deep pipelines and short code runs which occur when there are a lot of branch instructions
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)

Review: Major Components of a Computer



The “Memory Wall”

- Processor vs DRAM speed disparity continues to grow



Intel i7
(Haswell)
4 cycle L1
11-16 cycle L2
30-55 cycle L3
1600MHz DDR3

AMD FX
(Bulldozer)
3 cycle L1
~18 cycle L2
~65 cycle L3
1866MZ DDR3

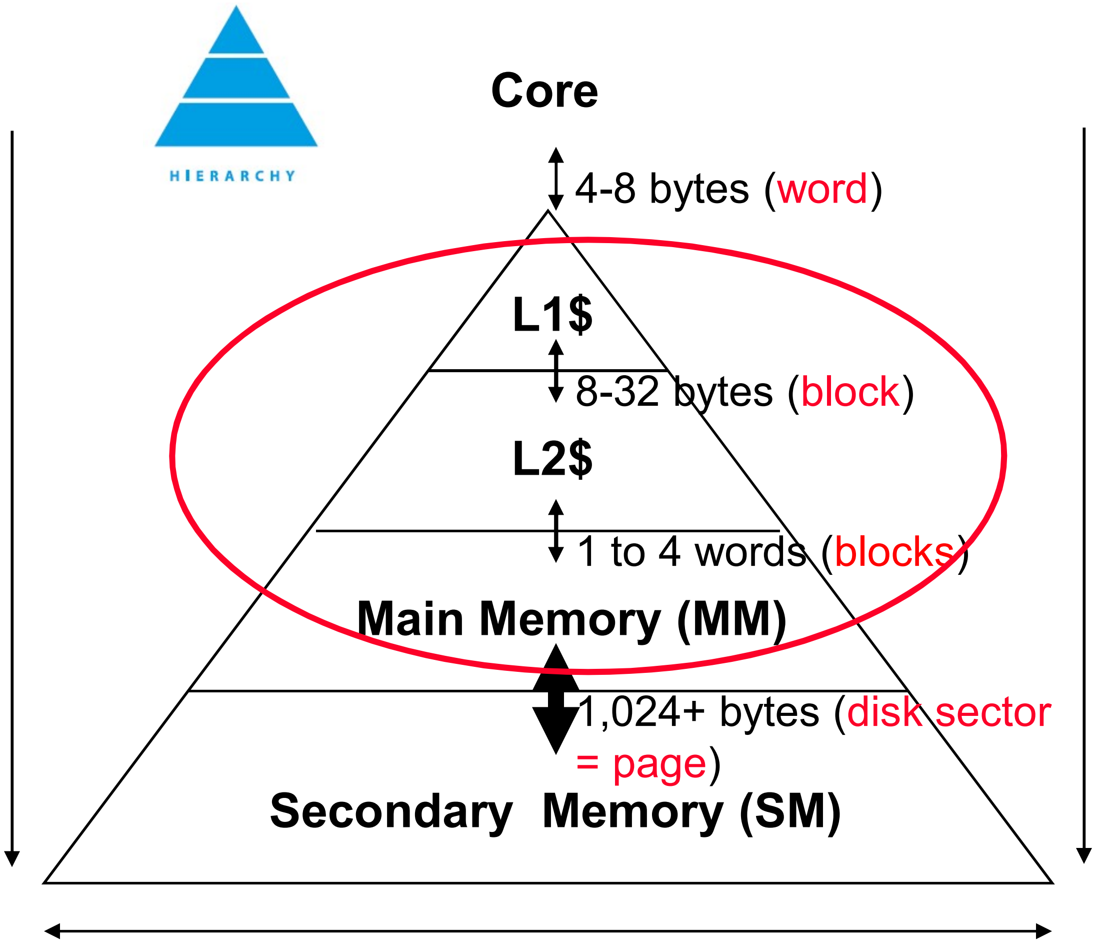
- Good memory hierarchy (cache) design is increasingly **important** to overall performance

The Memory Hierarchy Goal

- **Fact**: Large memories are slow and fast memories are small

- How do we create a memory that gives the illusion of being large, cheap and fast (most of the time)?
  - With hierarchy
  - With parallelism

Review: Characteristics of the Memory Hierarchy



Increasing distance from the core in access time

Inclusive caches – what is in L1$ is a subset of what is in L2$

Noninclusive caches – what is in L1$ is not a subset of what is in L2$

(Relative) size of the memory at each level

How is the Hierarchy Managed?

- registers ↔ memory
  - by compiler (programmer?)
- registers ↔ cache ↔ main memory
  - by the cache controller hardware
  - by the memory controller (MC) hardware
- main memory ↔ disks
  - by the operating system (virtual memory)
  - virtual to physical address mapping assisted by the hardware (TLB)
  - by the programmer (files)

Cache Basics

- Two questions to answer (in hardware):
  - Q1: How do we know if a data item is in the cache?
  - Q2: If it is, how do we find it?

- Direct mapped
  - Each memory block is mapped to exactly one block in the cache
    - lots of memory blocks must share a block in the cache
  - Address mapping (to answer Q2):

    (block address) modulo (# of blocks in the cache)

  - Have a tag associated with each cache block that contains the address information (the upper portion of the address) required to identify the block (to answer Q1)

The Memory Hierarchy: Terminology

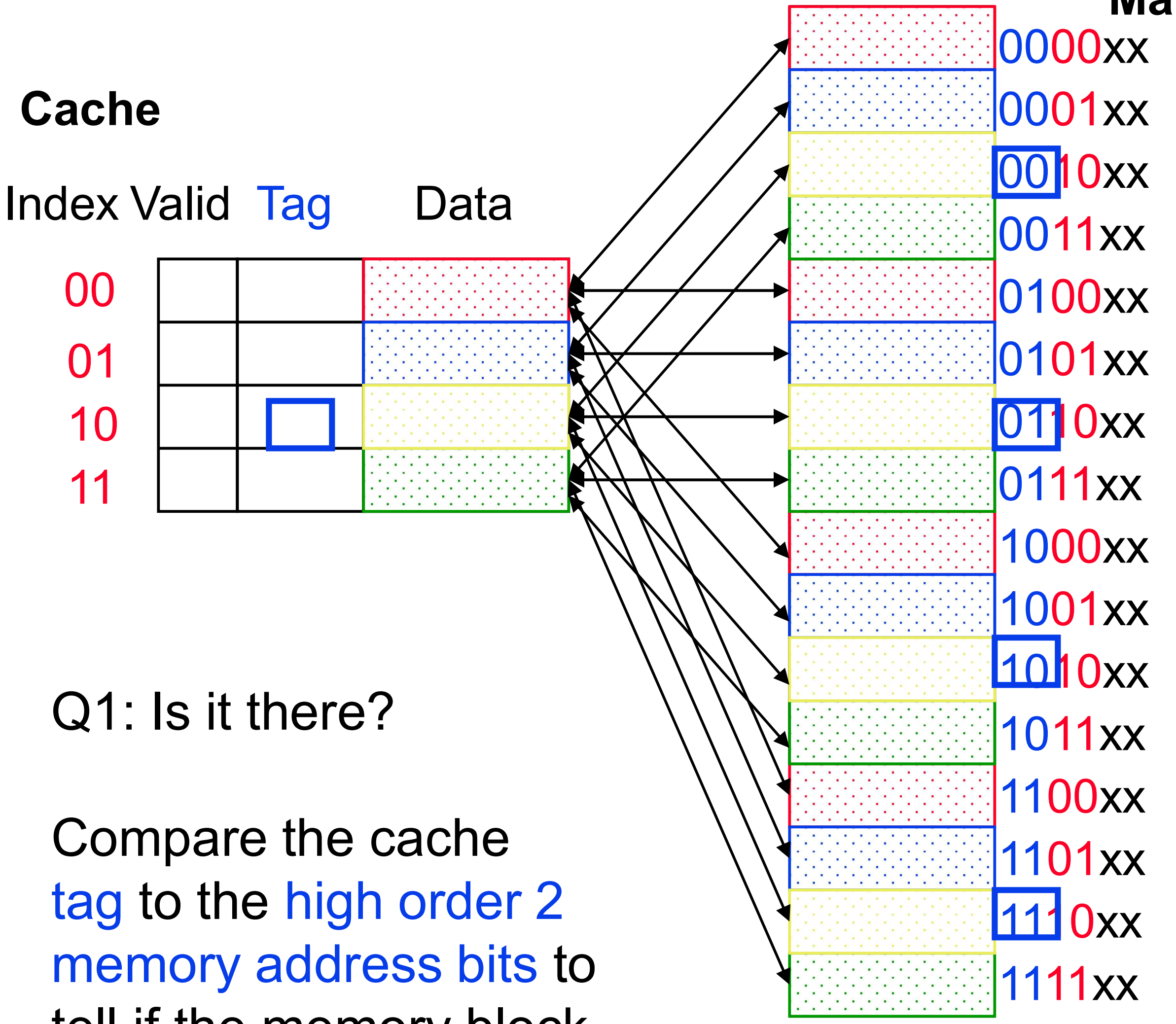
- Block (or line): the minimum unit of information that is present (or not) in a cache
- Hit Rate: the fraction of memory accesses found in a level of the memory hierarchy
  - Hit Time: Time to access that level which consists of

    Time to access the block + Time to determine hit/miss
- Miss Rate: the fraction of memory accesses *not* found in a level of the memory hierarchy ⇒ 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in that level with the corresponding block from a lower level which consists of

    Time to access the block in the lower level + Time to transmit that block to the level that experienced the miss + Time to insert the block in that level + Time to pass the block to the requestor

Hit Time << Miss Penalty

Caching: A Simple First Example



Main Memory

Cache

Index Valid Tag Data

00
01
10
11

0000xx
0001xx
0010xx
0011xx
0100xx
0101xx
0110xx
0111xx
1000xx
1001xx
1010xx
1011xx
1100xx
1101xx
1110xx
1111xx

One word blocks
Two low order bits define the byte in the word (32b words)

Q1: Is it there?

Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache

Q2: How do we find it?

Use next 2 low order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

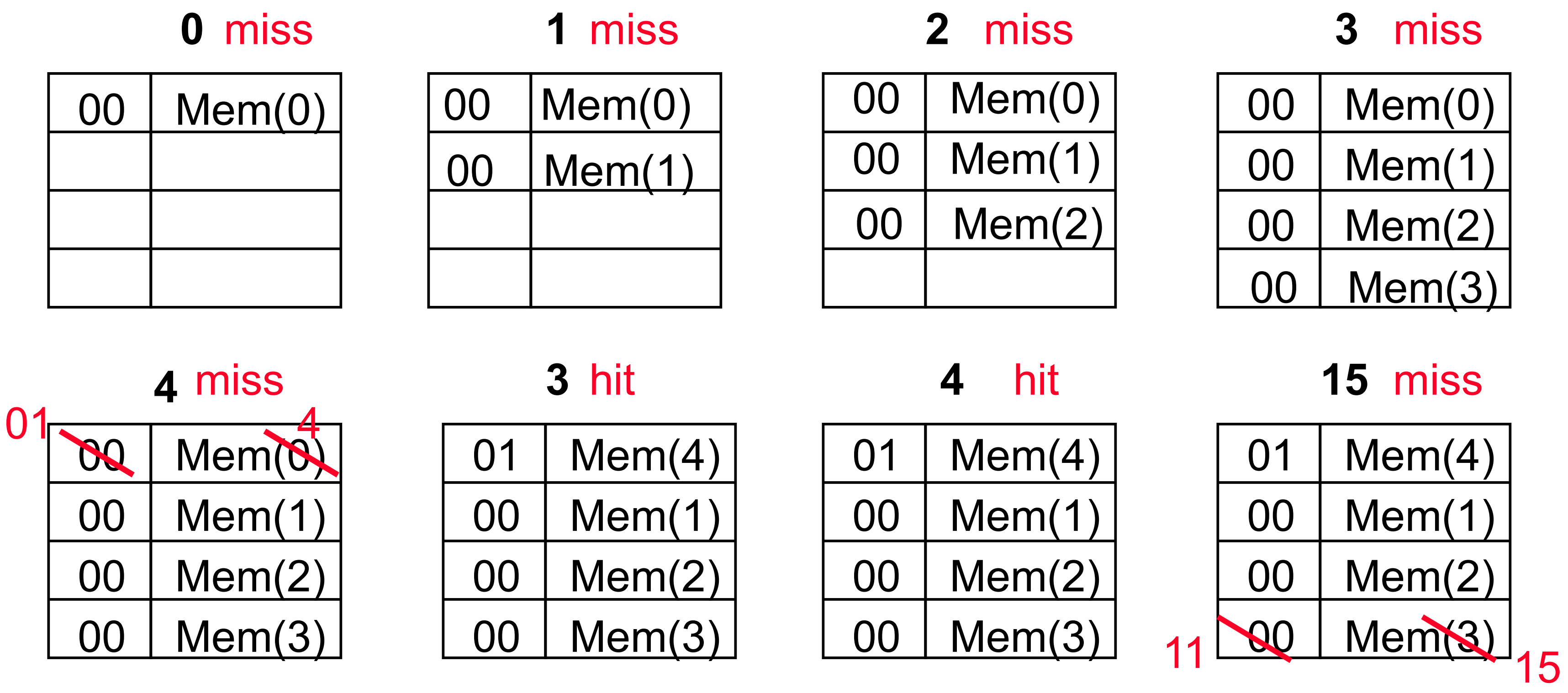
(block address) modulo (# of blocks in the cache)

Direct Mapped Cache

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15



0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

01 ~~00~~	~~Mem(0)~~ 4
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

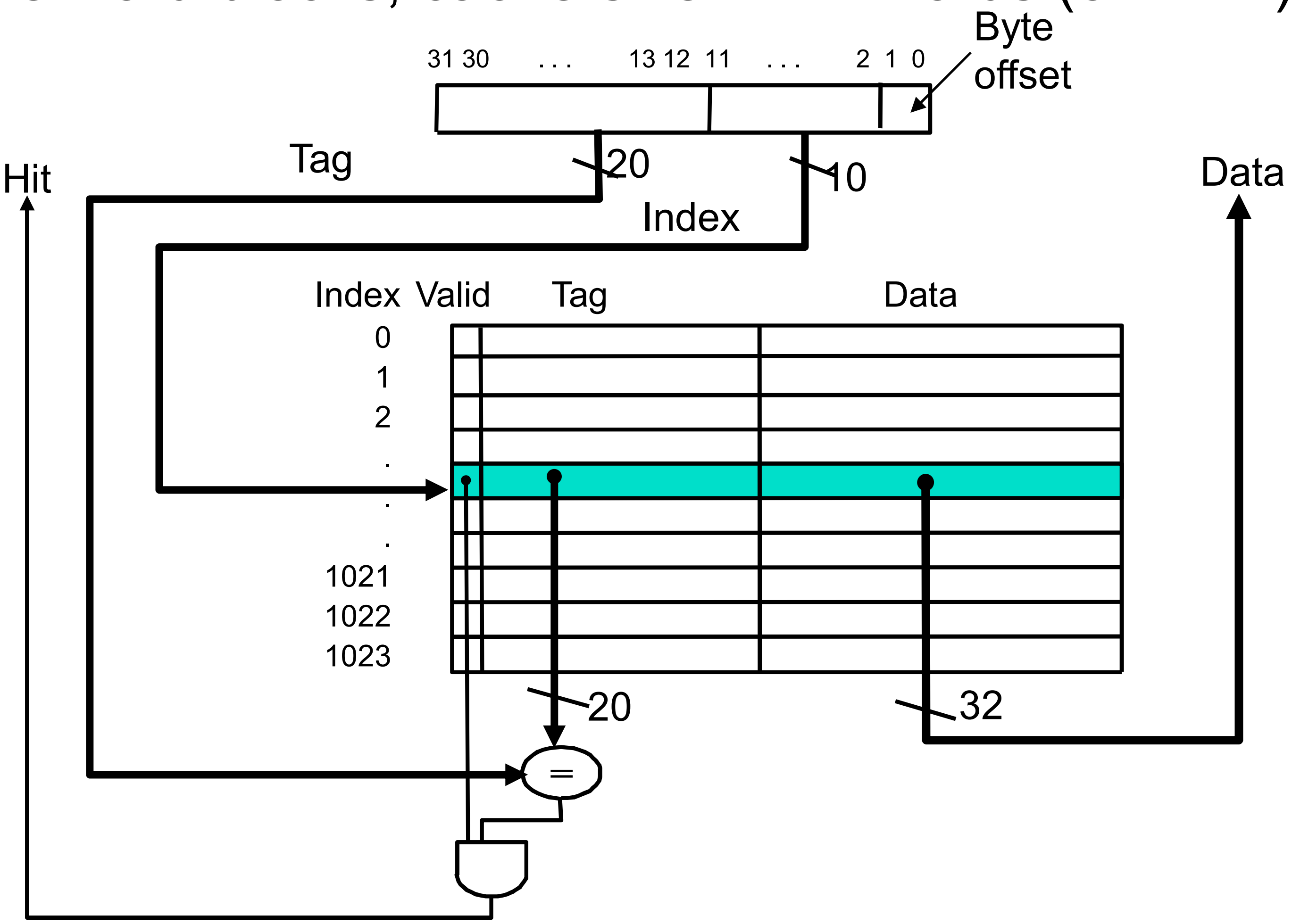
15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
11 ~~00~~	~~Mem(3)~~ 15

- 8 requests, 6 misses

MIPS Direct Mapped 4KiB Cache Example

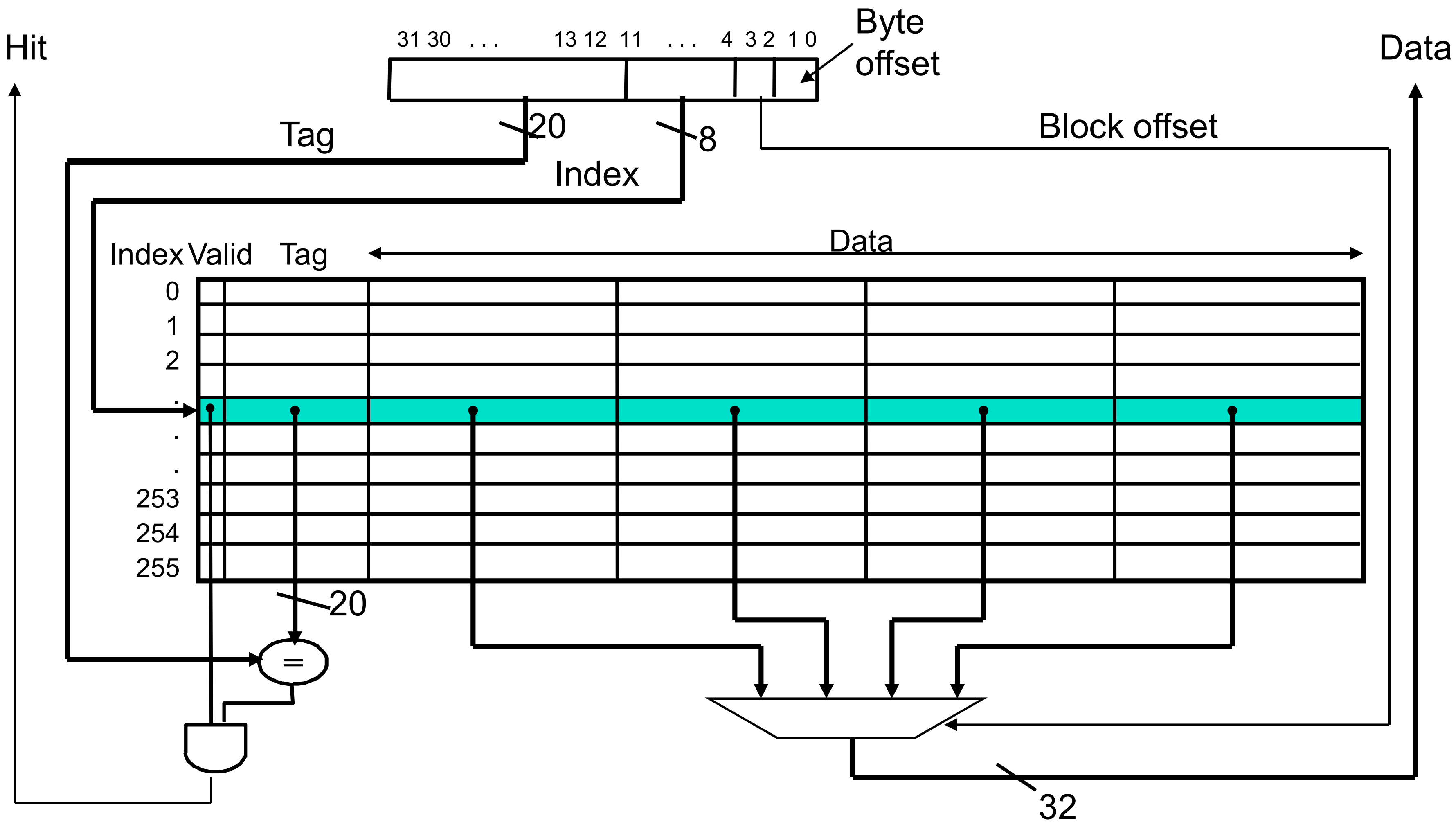
- One word blocks, cache size = 1Ki words (or 4KiB)



What kind of locality are we taking advantage of?

Multiword Block Direct Mapped 4KiB Cache

- Four words/block, cache size = 1Ki words



What kind of locality are we taking advantage of?

Taking Advantage of Spatial Locality

- ❑ Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

01 ~~00~~	~~Mem(1)~~ 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

01	Mem(5)	Mem(4)
11 ~~00~~	~~Mem(3)~~ 15	~~Mem(2)~~ 14

- 8 requests, 4 misses

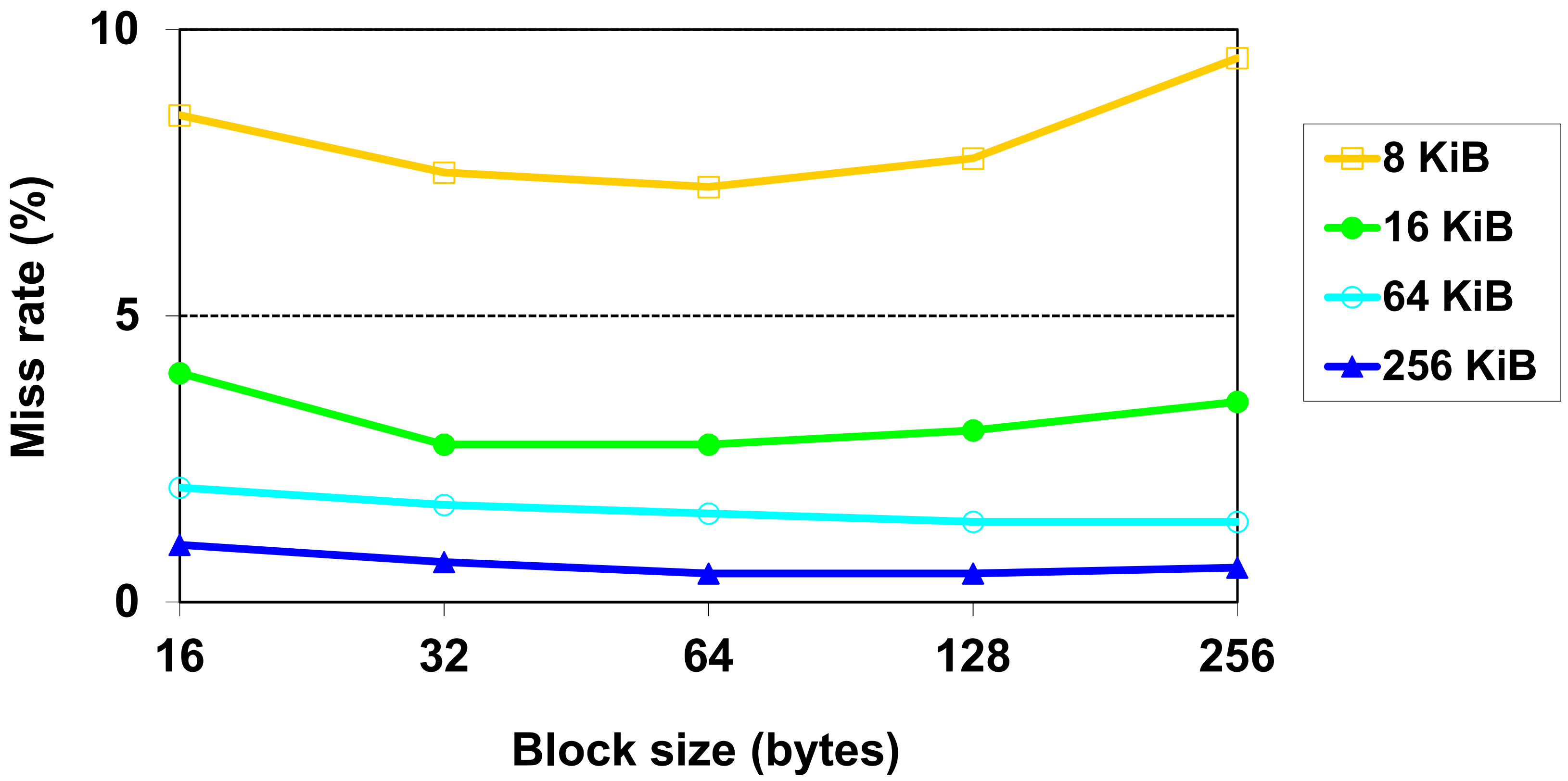
Cache Field Sizes

- The number of bits in a cache includes both the storage for data and for the tags
  - 32-bit byte address
  - A direct mapped cache with 2^n blocks has a *n* bits index
  - For a block size of 2^m words (2^{m+2} bytes), *m* bits are used to address the word within the block and 2 bits are used to address the byte within the word
- What is the size of the tag field? 32 – (n + m + 2)
- The total number of bits in a direct-mapped cache is then

  2^n x (block size + tag field size + valid field size)
- How many total bits are required for a direct mapped cache with 16KiB of data and 4-word blocks assuming a 32-bit address? 16KiB = 4KiW (2^12) 1024 blocks (2^10)

2^10 [4x32b data + (32-10-2-2)b tag +1b valid] = 147Kib = 18.375KiB

Miss Rate vs Block Size vs Cache Size

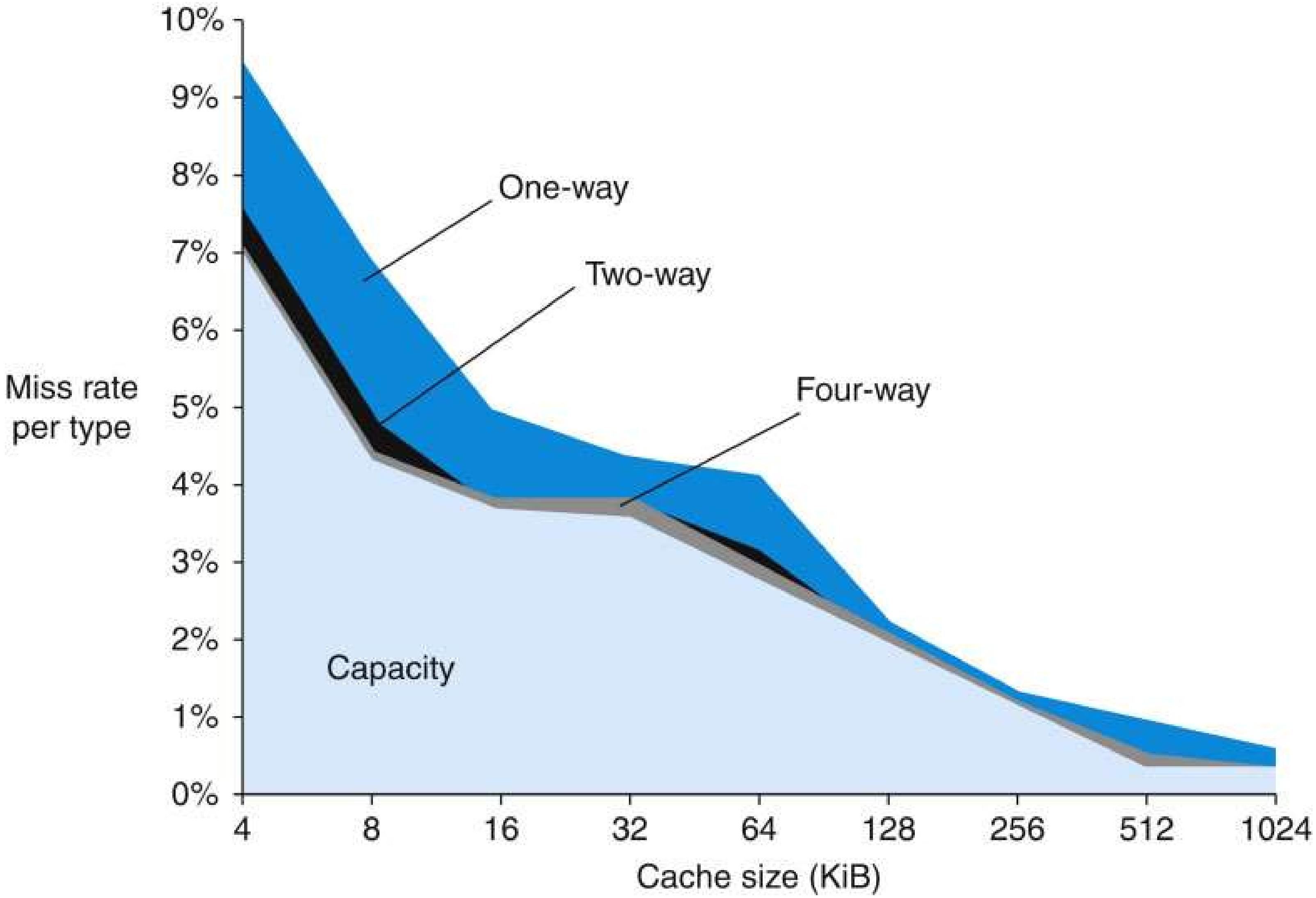


- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Sources of Cache Misses

- Compulsory (cold start or process migration, first reference):
  - First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant
  - 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 (stay tuned) (may increase access time)

Miss Rates per Cache Miss Type



Handling Cache Hits

- Read hits (I$ and D$)
  - this is what we want!

- Write hits (D$ only)
  - If we require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (write-through)
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer and stall only if the write buffer is full
  - If we allow cache and memory to be inconsistent
    - write the data only into the cache block (write-back the cache block to the next level in the memory hierarchy when that cache block is "evicted")
    - need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted – can use a write buffer to help "buffer" write-backs of dirty blocks

Handling Cache Misses (Single Word Blocks)

- Read misses (I$ and D$)
  - stall the pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), and send the requested word to the core, then let the pipeline resume
- Write misses (D$ only)
  1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the core to the cache, then let the pipeline resume
  2. Write allocate – just write the word (and its tag) into the cache (which may involve having to evict a dirty block if using a write-back cache), no need to check for cache hit, no need to stall
  3. No-write allocate – skip the cache write (but must invalidate that cache block since it will now hold stale data) and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full

Handling Cache Misses (Multiple Word Blocks)

- Read misses (I$ and D$)
  - Processed the same as for single word blocks – a miss returns the entire block from memory
  - Miss penalty grows as block size grows
    - Early restart – core resumes execution as soon as the requested word of the block is returned
    - Requested word first – requested word is transferred from the memory to the cache (and core) first
  - Nonblocking cache – allows the core to continue to access the cache while the cache is handling an earlier miss
- Write misses (D$ only)
  - If using write allocate must *first* fetch the block from memory and then write the word to the block (or could end up with a "garbled" block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block)

Measuring Cache Performance

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{CC}$$

$$= \text{IC} \times \underbrace{(\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles})}_{\text{CPI}_{\text{stall}}} \times \text{CC}$$

- Memory-stall cycles come from cache misses (a sum of read-stalls and write-stalls)

$$\text{Read-stall cycles} = \text{reads/program} \times \text{read miss rate} \times \text{read miss penalty}$$

$$\text{Write-stall cycles} = (\text{writes/program} \times \text{write miss rate} \times \text{write miss penalty}) + \text{write buffer stalls}$$

- For write-through caches, we can simplify this to

$$\text{Memory-stall cycles} = \text{accesses/program} \times \text{miss rate} \times \text{miss penalty}$$

Impacts of Cache Performance

- Relative cache penalty increases as core performance improves (faster clock rate and/or lower CPI)
  - The memory speed is unlikely to improve as fast as core cycle time. When calculating $\text{CPI}_{\text{stall}}$, the cache miss penalty is measured in *core* clock cycles needed to handle a miss
  - The lower the $\text{CPI}_{\text{ideal}}$, the more pronounced the impact of stalls

- A core with a $\text{CPI}_{\text{ideal}}$ of 2, a 100 cycle miss penalty, 36% load/store instr's, and 2% I$ and 4% D$ miss rates

$$\text{Memory-stall cycles} = 2\% \times 100 + 36\% \times 4\% \times 100 = 3.44$$

$$\text{So} \quad \text{CPI}_{\text{stalls}} = 2 + 3.44 = \mathbf{5.44}$$

more than twice the $\text{CPI}_{\text{ideal}}$!

Impacts of Cache Performance, Con't

- Relative cache penalty increases as core performance improves (lower CPI)
- What if the $\text{CPI}_{\text{ideal}}$ is reduced to 1? 0.5? 0.25?

$\text{CPI}_{\text{stall}}$ = 4.44 (up from 3.44/5.44 = 63% to 3.44/4.44 = 77%)

- What if the D$ miss rate went up 1%? 2%?

$\text{CPI}_{\text{stall}}$ = 2 + (2% x 100 + 36% x 5% x 100) = 5.80

- What if the core clock rate is doubled (doubling the miss penalty)?

$\text{CPI}_{\text{stall}}$ = 2 + (2% x 200 + 36% x 4% x 200) = 8.88 !!

Average Memory Access Time (AMAT)

- A larger cache will have a longer access time. An increase in hit time will likely add another stage to the pipeline. At some point the increase in hit time for a larger cache will overcome the improvement in hit rate leading to a decrease in performance.
- Average Memory Access Time (AMAT) is the average to access memory considering both hits and misses

AMAT = Time for a hit + Miss rate x Miss penalty

- What is the AMAT for a core with a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

AMAT = 1 + 0.02x50 = 2 cycles