CS 2340 - Computer Architecture

5 Arithmetic, Memory Operations

Dr. Alice Wang

The other day I told my friend 10 jokes about binary. Unfortunately, he didn't get either of them!

Review

Last Lecture

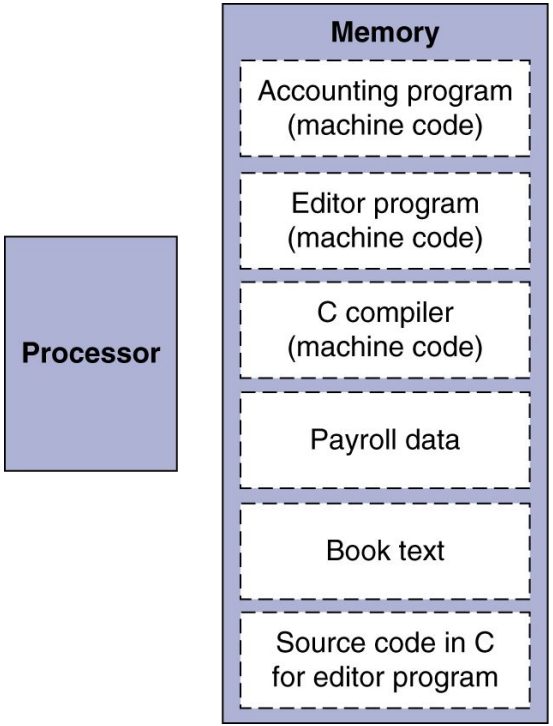
- Signed Number representation
  - Signed/Magnitude and 2s complement
- 2s complement is also a procedure
  - Changes positive numbers to negative, negative numbers to positive

This Lecture

- Back to MIPS programming

Review: Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data and stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

Review: Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **MIPS** architecture:
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many systems, including Sony Playstation, Nintendo 64 and Tesla Model S runs on MIPS computers

Once you've learned one architecture, it's easy to learn others

Review: MIPS Instructions (so far)

```
li $dst, immediate
la $dst, $symbol
lw $dst, offset($src)
sw $src, offset($dst)
add $dst, $src0, $src1
sub $dst, $src0, $src1
addi $dst, $src0, immediate
syscall
```

Today we will go deeper with arithmetic and memory operations

4 Architecture Design Principles

Computer Architecture (2340) underlying design principles, as articulated by Hennessy and Patterson:

1. **Simplicity favors regularity**
2. **Smaller is faster**
3. **Make the common case fast**
4. **Good design demands good compromises**

Operations

Definition of Operation

1. the fact or condition of functioning or being active.
2. an act of surgery performed on a patient.
3. a piece of organized and concerted activity involving a number of people, especially members of the armed forces or the police.
4. a process in which a number, quantity, expression, etc., is altered or manipulated according to formal rules, such as those of addition, multiplication, and differentiation.

Each instruction does one or more operations

Native Instructions vs Pseudo-Instructions

/Users/alicewang/MARS code/Lecture 2-Mars Demo.asm - MARS 4.5

File Edit Run Settings Tools Help

Click on the ?

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24020005	addiu $2,$0,5	15: li $v0, 5 # service...
	0x00400004	0x0000000c	syscall	16: syscall #...
	0x00400008	0x00028021	addu $16,$0,$2	17: move $s0, $v0 #...
	0x0040000c	0x24020001	addiu $2,$0,1	19: li $v0, 1 # service...
	0x00400010	0x00102021	addu $4,$0,$16	20: move $a0, $s0 #...
	0x00400014	0x0000000c	syscall	21: syscall #...
	0x00400018	0x24020004	addiu $2,$0,4	23: li $v0, 4 # service...
	0x0040001c	0x3c011001	lui $1,4097	24: la $a0, msg #...
	0x00400020	0x34240000	ori $4,$1,0	
	0x00400024	0x0000000c	syscall	25: syscall #...

Labels

Label	Address
Lecture 2-Mars Demo.asm	
msg	0x10010000
ArrayA	0x1001000c

Data Text

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	1819043144	1867980911	560229490	1	2	3	4	5
0x10010020	6	7	0	0	0	0	0	0
0x10010040	0	0	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0	0	0

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Registers

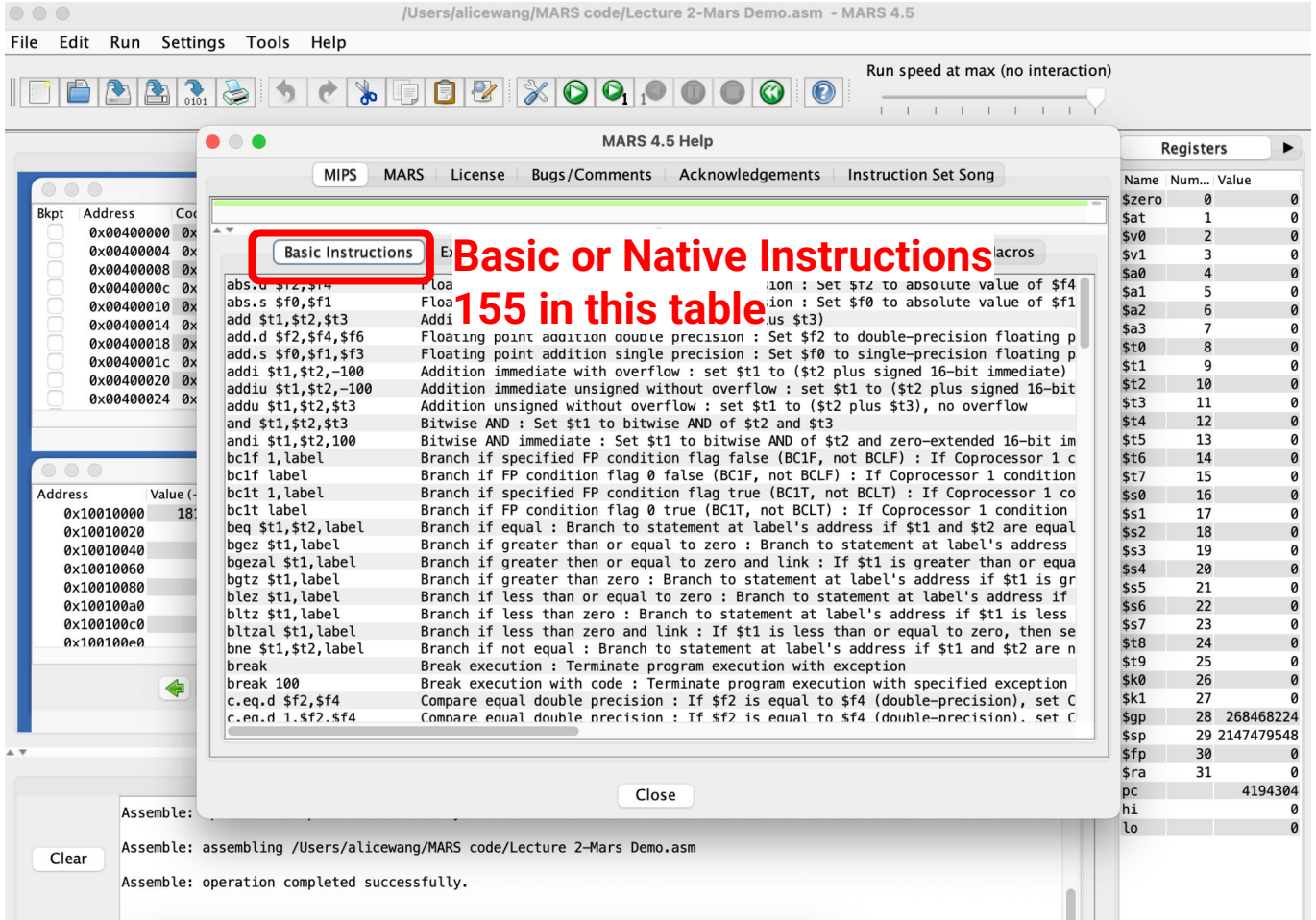
Name	Num...	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194304
hi		0
lo		0

Mars Messages Run I/O

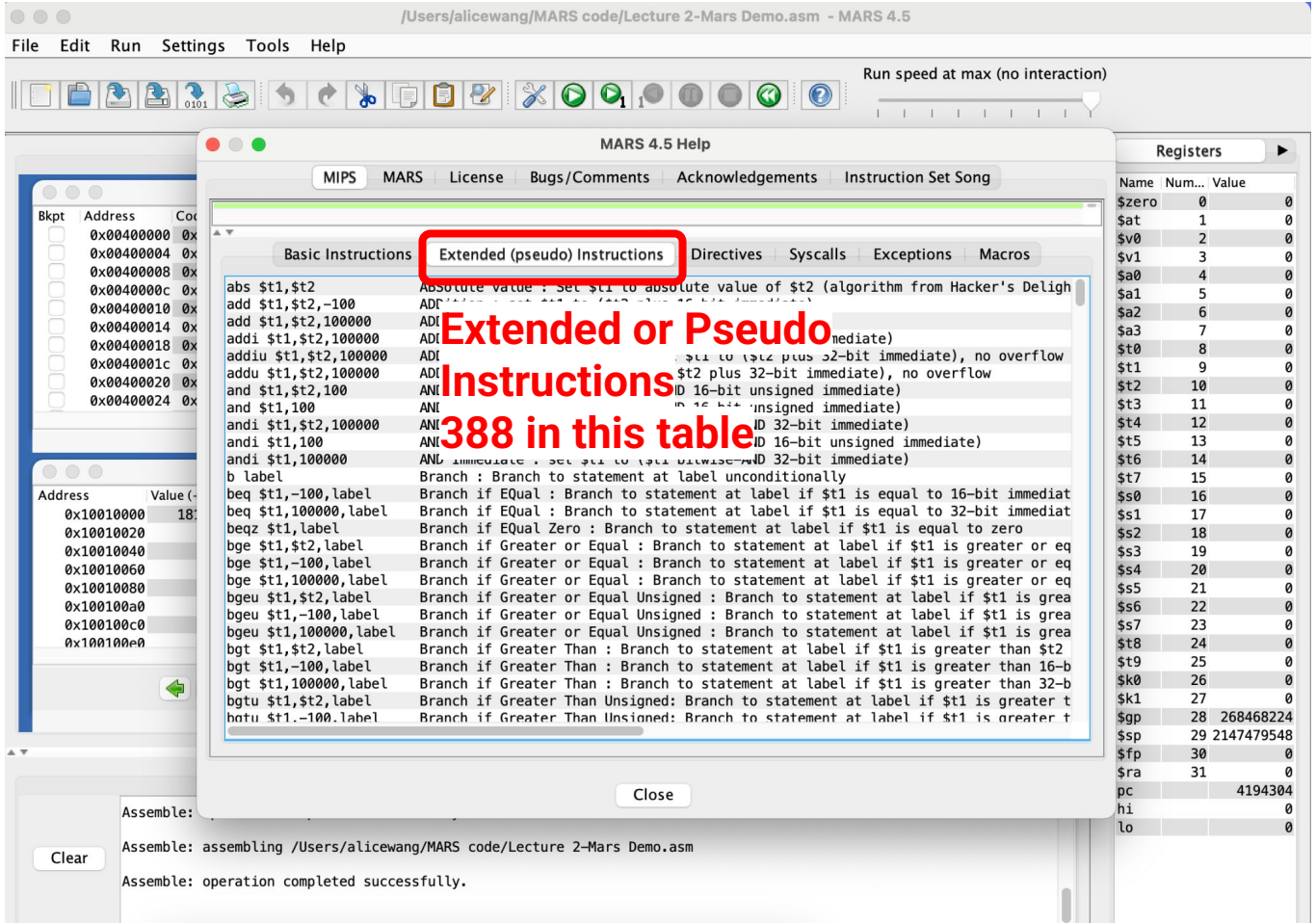
Clear

```
Assemble: operation completed successfully.

Assemble: assembling /Users/alicewang/MARS code/Lecture 2-Mars Demo.asm

Assemble: operation completed successfully.
```

Native Instructions vs Pseudo-Instructions



Native Instructions vs Pseudo-Instructions



Operands

- A computer operates on operands
- Operands are a physical location in computer
- Three kinds of Operands
  - Registers
  - Memory
  - Constants (also called *immediates*, stored in the instruction itself)

Review: Arithmetic Operations

- Examples: Add and Subtract
- Three operands
  - Two sources and one destination
  - `add a, b, c  # a gets b + c`
- All arithmetic operations have this form
- `c` can be a constant or immediate

- `add,sub,addi:` mnemonic (indicates operation to perform)
- `b, c:` source operands (on which the operation is performed)
- `a:` destination operand (to which the result is written)

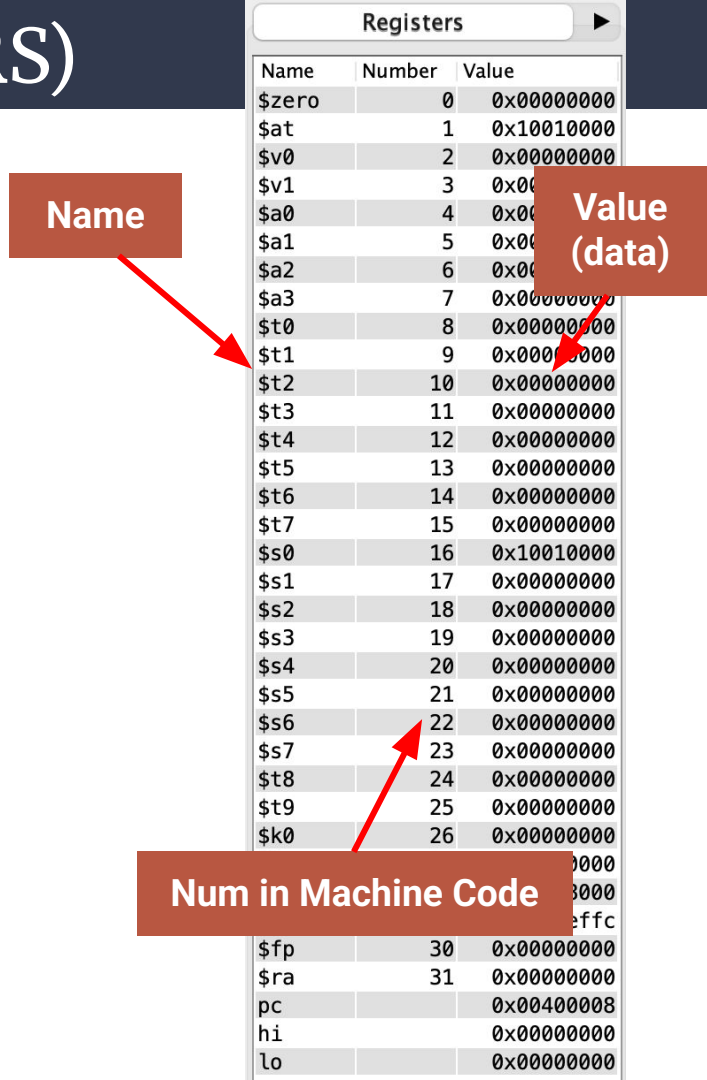
Design Principle #1

Simplicity favors regularity

- MIPS has a consistent instruction format with the same number of operands (two sources and one destination)
- Makes it easier to encode and handle in hardware

Review: Register Operands (MARS)

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - A “word” is 32-bits
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables



Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00
$a0	4	0x00
$a1	5	0x00
$a2	6	0x00
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
		0000
		3000
		effc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400008
hi		0x00000000
lo		0x00000000

Smaller is Faster

- MIPS includes only a small number of registers (32)
- MIPS is a “32-bit architecture” because it operates on 32-bit data
- Registers are faster than memory. The fewer there are the faster they can be accessed

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - Pseudo-instruction `mov` between registers uses $0

```
add $t2, $s1, $0
```

Interesting fact!

- There is no zero register in x86 ISA
- Dedicating a register to zero is surprisingly a large factor in simplifying the RISC-V ISA.

Arithmetic Example - My Turn

$s0 has g
$s1 has h
$s2 has i
$s3 has j
$s4 has f

Python code:

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

Compiled MIPS code:

```
add $t0, $s0, $s1  # temp t0 = g + h
add $t1, $s2, $s3  # temp t1 = i + j
sub $s4, $t0, $t1  # f = t0 - t1
```

Arithmetic Example – Your Turn

$s0 has a
$s1 has b
$s2 has ans1

- Python code:

```
ans1 = 3a -2b +32 # use a+a+a for 3a
```

- Compiled MIPS code:

```
add $t0, $s0, $s0     # temp t0 = a + a
add $t0, $t0, ___     # temp t0 = t0 + a
sub $t0, $t0, $s1     # temp t0 = t0 - b
sub $t0, $t0, $s1     # temp t0 = t0 - b
___ $s2, $t0, 32      # ans1 = t0 + 32
```

Make the common case fast

- MIPS is a ***reduced instruction set computer*** **(RISC)**, with a small number of commonly used simple instructions
- More complex instructions that are less commonly used execute with multiple simple instructions

Python Code

```
a = b + c - d
```

MIPS assembly code

```
add $t0, $s0, $s1   # t = b + c
sub $s2, $t0, $s3   # a = t - d
```

Unsigned Addition & Subtraction

C Code	MIPS assembly code
`a = b + c;`	`addu a, b, c`
`a = b + 6;`	`addiu a, b, 6`
`a = b - c;`	`subu a, b, c`

- Unsigned addition and subtraction are also native instructions
- Use these instructions if all of your numbers are positive (unsigned)
- Does not cause an <u>Overflow</u> Exception

What is Overflow?

Overflow happens when the result is > than the number of output bits allowed.

```
 0x7FFFFFFF        0111_1111_1111_1111_1111_1111_1111_1111
+0x00000001+  0000_0000_0000_0000_0000_0000_0000_0001
```

Adding 2 negative numbers should result in a negative number, but due to overflow the number becomes positive

Overflow exception in Mars

/Users/alicewang/MARS code/Lecture Code/Lecture 6-Mem Arith code.asm - MARS 4.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

Lecture 6-Mem Arith code.asm

```
27  # Overflow
28  li $s0, 0x7FFFFFFF
29  li $s1, 0x00000001
```

Line: 27 Column: 11 Show Line Numbers

```
Assemble: assembling /Users/alicewang/MARS code/Lec

Assemble: operation completed successfully.

Go: running Lecture 6-Mem Arith code.asm

Error in /Users/alicewang/MARS code/Lecture Code/Lecture 6-Mem Arith code.asm line 31: Runtime exception at 0x00400038: arithmetic overflow

Go: execution terminated with errors.
```

Clear

In Mars, an exception is thrown to alert the programmer that an overflow happened

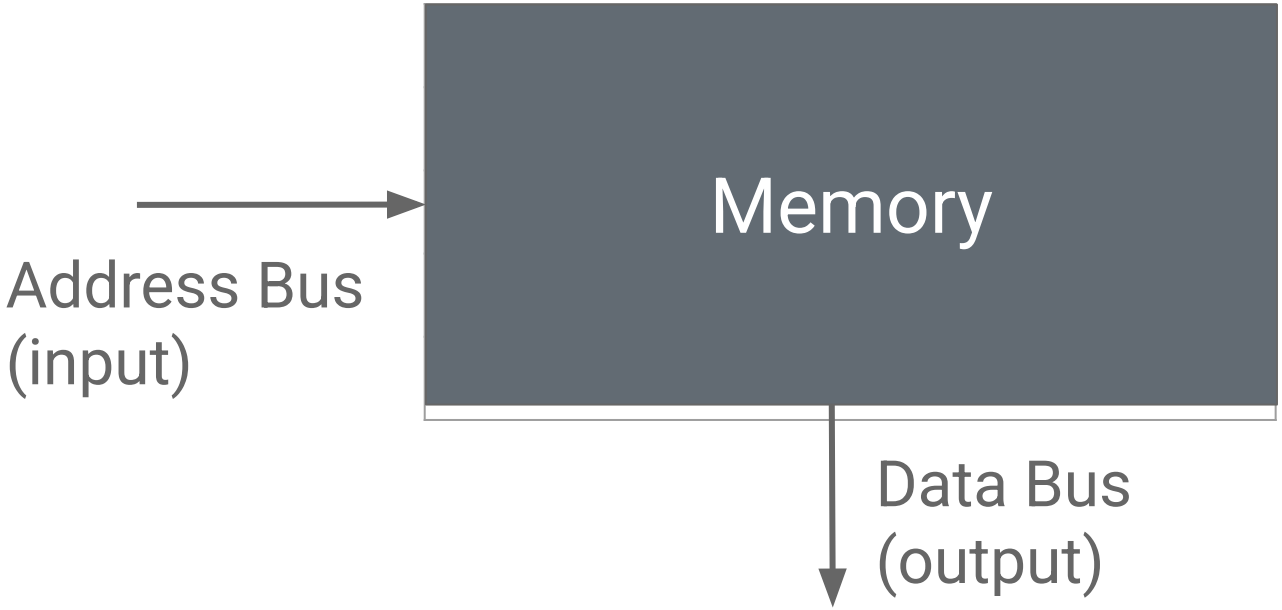
We will talk more about Exception handling in future lectures

Review: Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- Use memory when you more data than fits in 32 registers
  - Because you can store more data in memory
  - Downside - Large memories are slow
- Strategy : Commonly used variables are kept in registers
- MIPS can only do arithmetic or logical operations on registers
  - 2 step process to perform operations on memory data
  - Load base address (BA) to a register, Calculate address=BA+offset
    - Load values from Mem[Address] into Registers
    - Store result from Register to Mem[Address]

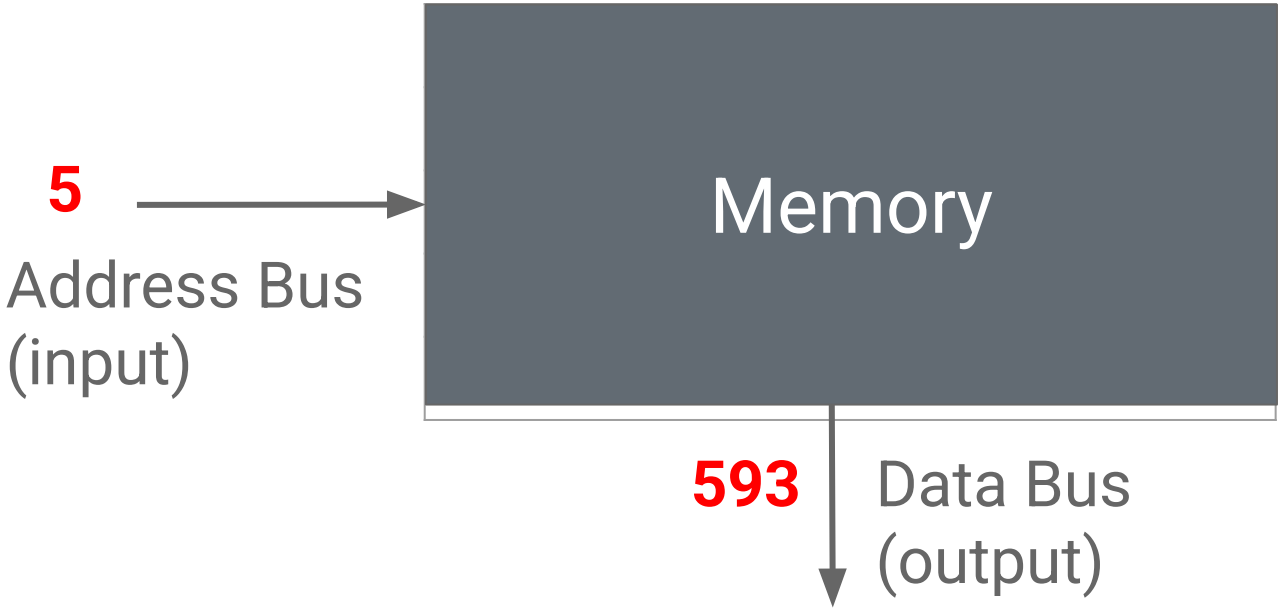
How does a memory work?

- If you want to read a value from memory, put the Address on the Address Bus. The Address points to a location in the memory.
- Then the value stored in the memory will appear on the Data Bus



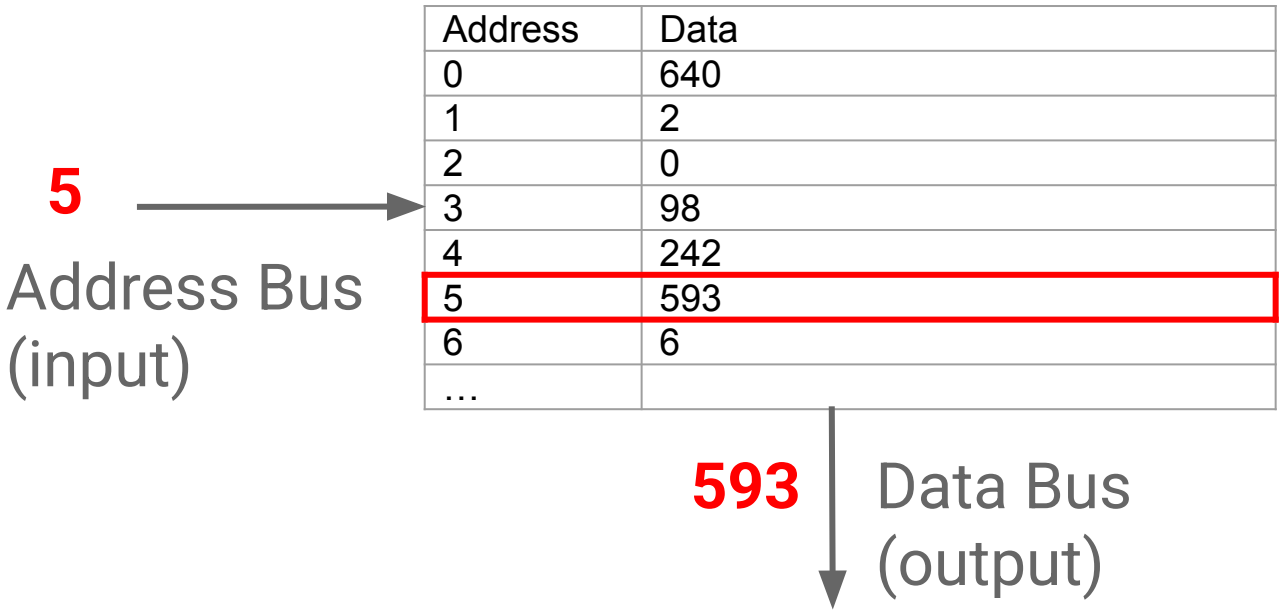
How does a memory work?

- Example: I want to read Mem[5] = 593
  - Put 5 into the Address Bus input
  - Read 593 on the Data Bus output



How does a memory work?

- Example: I want to read Mem[5] = 593
- What's under the hood? A lookup table



Address	Data
0	640
1	2
2	0
3	98
4	242
5	593
6	6
…	

How does a memory work? Your turn

- Example: I want to read Mem[3] = _____

3

Address Bus (input)

Address	Data
0	640
1	2
2	0
3	98
4	242
5	593
6	6
…	

Data Bus (output)

Data Memory (Mars)

Review of MARS

- Memory = Data Segment (Address, Value)

Mem[0x10010003] = _____

Data Segment

Address	Value (+0)	Value(+1)	Value(+2)	Value(+3)	Value(+4)	Value(+5)	Value(+6)	Value(+7)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010038	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Reading Word-Addressable Memory

- **Mnemonic:** *load word* (`lw`) - Memory read is called ***load***
- **Format:** `lw $s0, 5($t1)`

Assumes base address is already loaded into $t1

Step 1: Address calculation:

- add *base address* ($t1) to the *offset* (5)
- address = $t1 + 5

Step 2: Read from Memory to Register

- $s0 holds the value at address ($t1 + 5)

Remember! Load is reading from Memory to Register

Reading Word-Addressable Memory

Assembly code - My Turn

```
lw $s3, 1($s0)   # read memory to reg
```

Read a word of data at memory address $s0+1 into **$s3**

- **$s0** = _____
- address = **$s0** + 1 = _____
- **$s3** = _____ after load

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000

Data Segment

Address	Value (+0)	Value (+1)	Value (+2)	Value (+3)	Value (+4)	Value (+5)	Value (+6)	Value (+7)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010038	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Reading Word-Addressable Memory

Assembly code - My Turn

```
lw $s3, 1($s0)   # read memory to reg
```

Read a word of data at memory address $s0+1 into **$s3**

- **$s0** = 0x10010000
- address = **$s0** + 1 = 0x10010001
- **$s3** = 0xabcedf78 after load

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0xabcedf78
$s4	20	0x00000000
$s5	21	0x00000000

Data Segment

Address	Value (+0)	Value (+1)	Value (+2)	Value (+3)	Value (+4)	Value (+5)	Value (+6)	Value (+7)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010038	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Reading Word-Addressable Memory

Assembly code - Your Turn

```
lw $s4, 2($s0)  # read from memory to reg
```

Read a word of data at memory address $s0+2 into **$s4**

- **$s0** = _____
- address = **$s0** + 2 = _____
- **$s4** = _____ after load

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0xabcedf78
$s4	20	0x00000000
$s5	21	0x00000000

Data Segment

Address	Value (+0)	Value(+1)	Value(+2)	Value(+3)	Value(+4)	Value(+5)	Value(+6)	Value(+7)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010038	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Reading Word-Addressable Memory

Assembly code - Your Turn

```
lw $s4, 2($s0)  # read from memory to reg
```

Read a word of data at memory address $s0+2 into **$s4**

- **$s0** =
- address = **$s0** + 2 =
- **$s4** =

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0xabcedf78
$s4	20	0xf2f1ac07
$s5	21	0x00000000

Data Segment

Address	Value (+0)	Value(+1)	Value(+2)	Value(+3)	Value(+4)	Value(+5)	Value(+6)	Value(+7)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010038	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Writing Word-Addressable Memory

- **Mnemonic:** *store word* (`sw`) - Memory write is called ***store***
- **Format:** `sw $s0, 3($t1)`

Step 1: Address calculation:

- add *base address* ($t1) to the *offset* (3)
- address = $t1 + 3

Step 2: Write to Memory from Register

- Write to address ($t1 + 3) the value in $s0

Remember! Store is writing from Registers to Memory

Writing Word-Addressable Memory

Assembly code - My Turn

```
sw $t0, 1($s0)   # write from reg to memory
```

Write **$t0** into memory at address **$s0+1**

- **$s0** = _____
- address = **$s0** + 1 = _____
- Mem[_____] = _____

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000000f
$t1	9	0x00000041
$t2	10	0x000000dc
$t3	11	0xfffffff6
$t4	12	0x0000005c
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000

Data Segment

Address	Value (+0)	Value (+1)	Value (+2)	Value (+3)	Value (+4)	Value (+5)	Value (+6)	Value (+7)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Writing Word-Addressable Memory

Assembly code - My Turn

```
sw $t0, 1($s0)   # write from reg to memory
```

Write **$t0** into memory at address **$s0+1**

- **$s0** = 0x10010000
- address = **$s0** + 1 = 0x10010001
- Mem[0x10010001] = 0x0000000f

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000000f
$t1	9	0x00000041
$t2	10	0x000000dc
$t3	11	0xfffffff6
$t4	12	0x0000005c
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000

Data Segment

Address	Value (+0)	Value(+1)	Value(+2)	Value(+3)	Value(+4)	Value(+5)	Value(+6)	Value(+7)
0x10010000	0x40f30788	0x0000000f	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Writing Word-Addressable Memory

Assembly code - Your Turn

```
sw $t1, 4($s0)  # write from reg to memory
```

Write **$t1** into memory at address **$s0+4**

- **$s0** = _____
- address = **$s0** + 4 = _____
- Mem[_____] = _____

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000000f
$t1	9	0x00000041
$t2	10	0x000000dc
$t3	11	0xfffffff6
$t4	12	0x0000005c
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000

Data Segment

Address	Value (+0)	Value (+1)	Value (+2)	Value (+3)	Value (+4)	Value (+5)	Value (+6)	Value (+7)
0x10010000	0x40f30788	0x0000000f	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Writing Word-Addressable Memory

Assembly code - Your Turn

```
sw $t1, 4($s0)  # write from reg to memory
```

Write **$t1** into memory at address **$s0+4**

- **$s0** =
- address = **$s0** + 4 =
- Mem[_____] =

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000000f
$t1	9	0x00000041
$t2	10	0x000000dc
$t3	11	0xfffffff6
$t4	12	0x0000005c
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000

Data Segment

Address	Value (+0)	Value (+1)	Value (+2)	Value (+3)	Value (+4)	Value (+5)	Value (+6)	Value (+7)
0x10010000	0x40f30788	0x0000000f	0xf2f1ac07	0x01ee2844	0x00000041	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010028	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010030	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Question:

If a Word-Addressable memory has 32-bit entries

What kind of memory has 8-bit entries?

Byte-Addressable Memory

- Each data **`byte`** has unique address
  - Mem[0] = 0x88
  - Mem[1] = 0x07
  - o Mem[2] = 0x_____
- **MIPS is byte-addressed, not word-addressed**
- Load/store single bytes: load byte unsigned (**`lbu`**) and store byte (**`sb`**)

Data Segment **Word addressable vs**

Address	Value (+0)	Value (+1)	Value (+2)	Value (+3)	Value (+4)	Value (+5)	Value (+6)	Value (+7)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Data Segment **Byte addressable**

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Reading Byte-Addressable Memory

- MIPS cpu is 32-bit words, how do we calculate the address of a memory word for a byte-addressabile memory?
- 32-bit word = 4 bytes, so word address increments by 4
- For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 5 is _____ (0x_____ in hex)

Data Segment **Word addressable**

Address	Value (+0)	Value (+1)	Value (+2)	Value (+3)	Value (+4)	Value (+5)	Value (+6)	Value (+7)
0x10010000	0x40f30788	0xabcedf78	2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010008	0x00000000	0x00000000	0000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010010	0x00000000	0x00000000	0000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010018	0x00000000	0x00000000	0000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Data Segment **Byte addressable**

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Reading Byte-Addressable Memory

Assembly code - My Turn

```
lw $s3, 4($s0)  # read from memory to reg
```

If I want to read 0xabcedf78 from the memory, set offset to 4

- `$s0` = 0x10010000
- address = (`$s0` + 4) = 0x10010004
- `$s3` = 0xabcedf78 after load

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
lo		0x00000000

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Reading Byte-Addressable Memory

Assembly code - My Turn

```
lw $s3, 4($s0)  # read from memory to reg
```

If I want to read 0xabcedf78 from the memory, set offset to 4

- `$s0` = 0x10010000
- address = (`$s0` + 4) = 0x10010004
- `$s3` = 0xabcedf78 after load

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x10010000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0xabcedf78
$s4	20	0x00000000

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x40f30788	0xabcedf78	0xf2f1ac07	0x01ee2844	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

lo	0x00000000

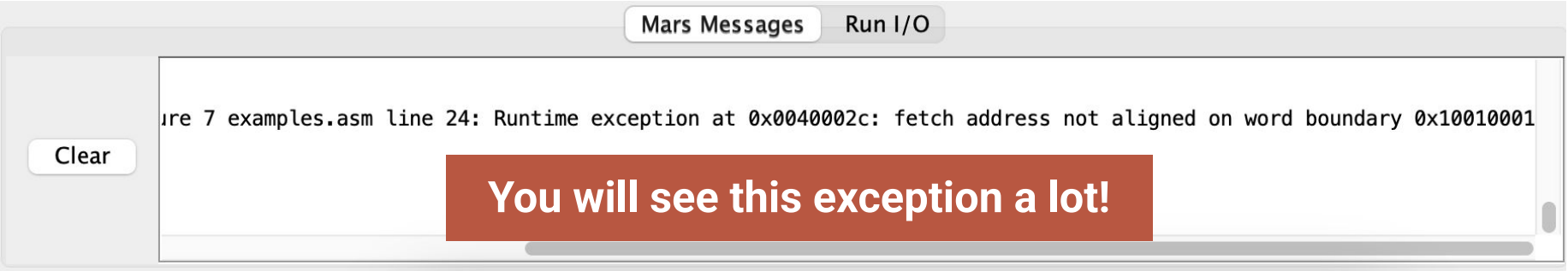
Fetch Address not aligned Exception

Assembly code - My Turn

```
lw $s4, 1($s0)   # write from reg to memory
```

Write reg value $t1 into memory at address $s0+1

- **$s0** = 0x10010000
- address = (**$s0** + 1) = 0x10010001 is not on a word boundary!



Memory alignment

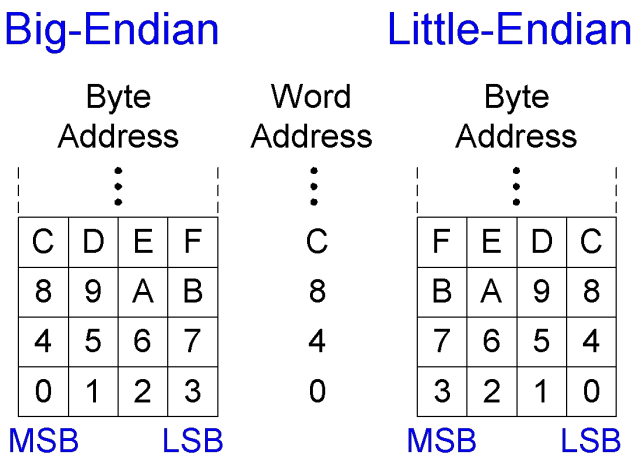
- Words (32-bit) are aligned in memory
  - load word (lw) address must be a multiple of 4
- Other Memory operations
  - load byte unsigned (`lbu`), store byte (`sb`)
  - load halfword unsigned (`lhu`), store halfword (`sh`)

DATA ALIGNMENT

<table>
<tr><td colspan="8">Double Word</td></tr>
<tr><td colspan="4">Word</td><td colspan="4">Word</td></tr>
<tr><td colspan="2">Halfword</td><td colspan="2">Halfword</td><td colspan="2">Halfword</td><td colspan="2">Halfword</td></tr>
<tr><td>Byte</td><td>Byte</td><td>Byte</td><td>Byte</td><td>Byte</td><td>Byte</td><td>Byte</td><td>Byte</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
</table>

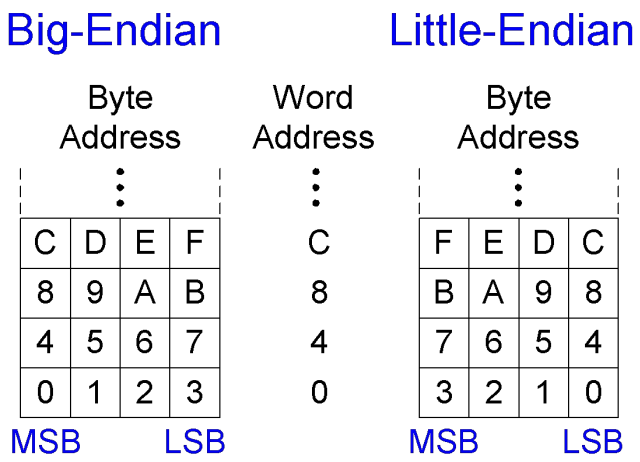
Big-Endian and Little-Endian Memory

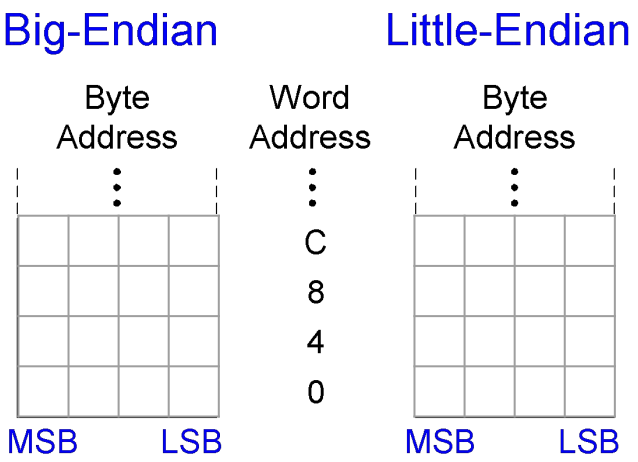
- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian



Big-Endian and Little-Endian Memory

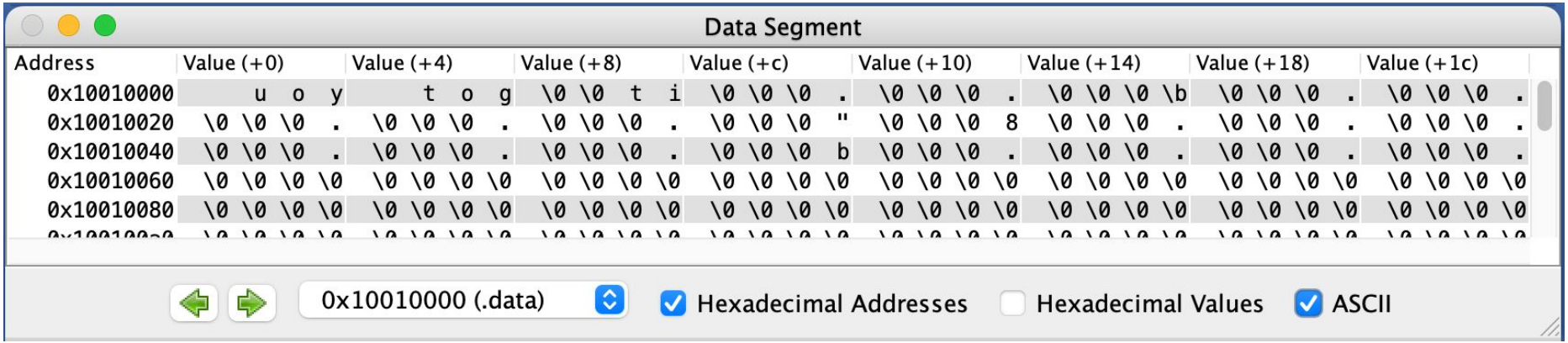
- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- How would the string “you got it” be stored in a big vs little endian system? (reminder: each character is one byte)





MARS data memory

- Here is the string “you got it” stored in MARS data memory
- MARS is little endian



Memory Operand Example – My Turn

- Python code:

```
A = np.array([1,2,3,4,5])

g = A[2]

A[4] = g
```

- Compiled MIPS code

```
.data
A: .word 1,2,3,4,5 # Array of Words

# variable g is $t0
.text
la $s3, A  # $s3 is the base address of A
lw $t0, ____ (_____) # Address = $s3+8
```

Memory Operand Example – My Turn

- Python code:

```
A = np.array([1,2,3,4,5])

g = A[2]

A[4] = g
```

- Compiled MIPS code

```
.data
A: .word 1,2,3,4,5 # Array of Words

# variable g is $t0
.text
la $s3, A  # $s3 is the base address of A
lw $t0, 8 ($s3) # Address = $s3+8
```

Memory Operand Example – Your Turn

- Python code:

```
A = np.array([1,2,3,4,5])

g = A[2]

A[4] = g
```

- Compiled MIPS code

```
.data
A: .word 1,2,3,4,5 # Array of Words

# variable g is $t0
.text
la $s3, A  # $s3 is the base address of A
lw $t0, 8 ($s3) # Address = $s3+8
__ $t0, __($s3)
```

Fill in the blanks

Memory and Arithmetic Example – My Turn

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	5	3	8	2	4	7	5	2
0x10010020	34	56	3	1	23	4	5	6
0x10010040	98	5	6	3	4	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0

0x10010000 (.data) ☑ Hexadecimal Addresses ☐ Hexadecimal Values ☐ ASCII

Registers | Coproc 1

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0x10010000
$t1	9	0
$t2	10	0
$t3	11	8
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0

```
lw $t3, 8($t0)
lw $t4, 20($t0)
lw $t5, 0($t0)
lw $t6, 12($t0)

add $t1, $t3, $t4
sub $t2, $t5, $t6
add $t1, $t1, $t2
addi $t1, $t1, 4

sw $t1, 4($t0)
```

Answer the questions:

1) What memory location is updated by this code?
2) What value is it updated to?

Here is the first instruction:
lw - load from Memory to Register
Address = $t0+8 = 0x10010008
$t3 = Mem[0x10010008] = 8

Memory and Arithmetic Example – Your Turn

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	5	3	8	2	4	7	5	2
0x10010020	34	56	3	1	23	4	5	6
0x10010040	98	5	6	3	4	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0

0x10010000 (.data) ☑ Hexadecimal Addresses ☐ Hexadecimal Values ☐ ASCII

Registers | Coproc 1

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0x10010000
$t1	9	0
$t2	10	0
$t3	11	8
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0

```
lw $t3, 8($t0)
lw $t4, 20($t0)
lw $t5, 0($t0)
lw $t6, 12($t0)

add $t1, $t3, $t4
sub $t2, $t5, $t6
add $t1, $t1, $t2
addi $t1, $t1, 4

sw $t1, 4($t0)
```

Answer the questions:

1) What memory location is updated by this code?
2) What value is it updated to?

I will give you 10min to work on the remaining instructions and answering the above questions.

Memory and Arithmetic Example – Your Turn

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	5	3	8	2	4	7	5	2
0x10010020	34	56	3	1	23	4	5	6
0x10010040	98	5	6	3	4	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0

0x10010000 (.data) ☑ Hexadecimal Addresses ☐ Hexadecimal Values ☐ ASCII

Registers | Coproc 1

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0x10010000
$t1	9	22
$t2	10	3
$t3	11	8
$t4	12	7
$t5	13	5
$t6	14	2
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0

```
lw $t3, 8($t0)
lw $t4, 20($t0)
lw $t5, 0($t0)
lw $t6, 12($t0)

add $t1, $t3, $t4
sub $t2, $t5, $t6
add $t1, $t1, $t2
addi $t1, $t1, 4

sw $t1, 4($t0)
```

$t3 = Mem[0x10010008] = 8

$t4 = Mem[_____]=_____

$t5 = Mem[_____]=_____

$t6 = Mem[_____]=_____

$t1 = $t3+$t4 = _____

$t2 = $t5-$t6 = _____

$t1 = $t1+$t2 = _____

$t1 = $t1+4 = _____

Mem[_____] = _____

Summary

- 4 Architecture Design Principles
- Operations and Operands
- Arithmetic Operations
- Memory Operations
  - Word- and Byte- addressable Memories
  - lw and sw, lbu and sb

Key things to remember:

- Load is Read Mem to Reg, Store is Write Mem from Reg
- MIPS is byte-addressed, A Word has 4 Bytes

Next lecture

Data Arrays and Conditional / Decision Operations