

```
loop: lw    $t3, 0($t0)
      lw    $t4, 4($t0)
      add   $t2, $t3, $t4
      sw    $t2, 8($t0)
      addi  $t0, $t0, 4
      addi  $t1, $t1, -1
      bgtz $t1, loop
```

Assembler

```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20fff9
```

# Intro to MIPS Assembly Language

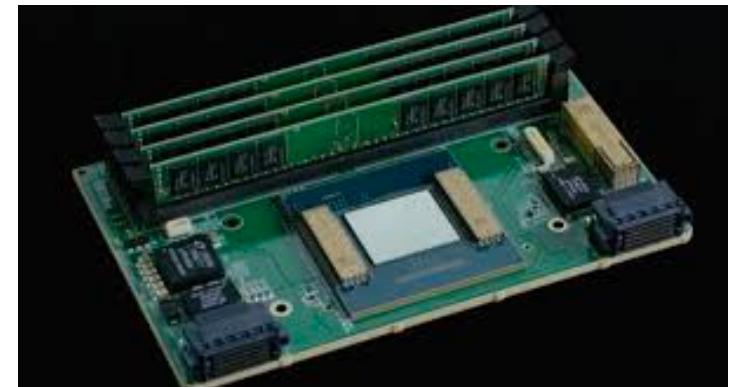
CS 64: Computer Organization and Design Logic

Lecture #4

Fall 2020

Ziad Matni, Ph.D.

Dept. of Computer Science, UCSB



# Lecture Outline

---

- MIPS core processing blocks
- Basic programming in assembly
- Intro to SPIM use

# Any Questions From Last Lecture?

---

# 5-Minute Pop Quiz!!!

---

**YOU MUST SHOW YOUR WORK!!!**

1. Calculate, give your answer in ***hexadecimal AND*** identify carry out (C) and overflow (V) bit values:

**(0xCE + 0xA9)**

2. Convert from binary to decimal **AND** to hexadecimal. Use any technique(s) you like:

**1011011**

# Answers...

---

1. Calculate, give your answer in hexadecimal, AND identify carry out (C) and overflow (V) bit values: (**0xCE + 0xA9**)

$$\begin{array}{r} \textcolor{purple}{1100 \ 1110} & \text{There is a carry out, so } \underline{\text{C = 1}} \\ + \textcolor{purple}{1010 \ 1001} & \text{There's overflow (why?), so } \underline{\text{V = 1}} \\ \hline = \textcolor{purple}{1 \ 0111 \ 0111} & = \textcolor{purple}{0x77} \end{array}$$

2. Convert from binary to decimal AND hexadecimal. Use any technique you like: **1011011**

$$\begin{aligned} &= \textcolor{red}{0101 \ 1011} = \textcolor{red}{0x5B} \quad (\textit{collect-the-bits method}) \\ &= \textcolor{red}{64 + 16 + 8 + 2 + 1} = \textcolor{red}{91} \quad (\textit{binary positional notation method}) \\ &\textbf{OR} \quad \textcolor{red}{0x5B} = \textcolor{red}{5 \times 16 + 11} = \textcolor{red}{80 + 11} = \textcolor{red}{91} \\ &\qquad\qquad\qquad (\textit{hex positional notation method}) \end{aligned}$$

# Code on MIPS

## Original

```
x = 5;  
y = 7;  
z = x + y;
```

## MIPS

```
li $t0, 5  
li $t1, 7  
add $t2, $t0, $t1
```

## Instruction Register

?

## Registers

\$t0: ?

\$t1: ?

\$t2: ?

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

?

## Program Counter

?

## Arithmetic Logic Unit

?

## Instruction Register

?

## Registers

\$t0: ?

\$t1: ?

\$t2: ?

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

## Program Counter

0

## Arithmetic Logic Unit

?

## Instruction Register

```
li $t0, 5
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

## Registers

```
$t0: ?
$t1: ?
$t2: ?
```

## Program Counter

```
0
```

## Arithmetic Logic Unit

```
?
```

## Instruction Register

```
li $t0, 5
```

## Registers

\$t0: 5

\$t1: ?

\$t2: ?

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

## Program Counter

0

## Arithmetic Logic Unit

?

## Instruction Register

```
li $t0, 5
```

## Registers

\$t0: 5

\$t1: ?

\$t2: ?

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

## Program Counter

4

## Arithmetic Logic Unit

0 + 4 = 4

## Instruction Register

```
li $t1, 7
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

## Registers

\$t0: 5

\$t1: ?

\$t2: ?

## Program Counter

4

## Arithmetic Logic Unit

?

## Instruction Register

```
li $t1, 7
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

## Registers

```
$t0: 5
$t1: 7
$t2: ?
```

## Program Counter

```
4
```

## Arithmetic Logic Unit

```
?
```

## Instruction Register

```
li $t1, 7
```

## Registers

```
$t0: 5  
$t1: 7  
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

## Memory

```
0: li $t0, 5  
4: li $t1, 7  
8: add $t2, $t0, $t1
```

## Program Counter

```
8
```

## Arithmetic Logic Unit

```
4 + 4 = 8
```

## Instruction Register

```
add $t2, $t0, $t1
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

## Registers

```
$t0: 5
$t1: 7
$t2: ?
```

## Program Counter

```
8
```

## Arithmetic Logic Unit

```
?
```

## Instruction Register

```
add $t2, $t0, $t1
```

## Registers

\$t0:	5
\$t1:	7
\$t2:	?

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

0:	li	\$t0,	5	
4:	li	\$t1,	7	
8:	add	\$t2,	\$t0,	\$t1

## Program Counter

8

## Arithmetic Logic Unit

5 + 7 = 12
------------

## Instruction Register

```
add $t2, $t0, $t1
```

## Registers

\$t0: 5

\$t1: 7

\$t2: 12

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes  
(more on this later).

## Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t2, $t0, $t1
```

## Program Counter

8

## Arithmetic Logic Unit

5 + 7 = 12

# Adding More Functionality

---

- Ok, so I know how to add 2 numbers in MIPS.
  - Wow
- What about: display results???? *Yes, that's kinda important...*
- What would this entail?
  - Engaging with Input / Output part of the computer
  - i.e. talking to devices
- So we need a way to tell
  - the operating system to kick in

**Q: What usually handles this?**

**A: the operating system**

# Also, Where's My MIPS Computer???

---

- You're not getting one.
- Who needs hardware when “cutting edge” software can do the job?!?!?!?
- We will be *EMULATING* a MIPS processor using software on our Macs/Windows/Linux machines.
- Hence... ***SPIM***... **The MIPS Emulator!**
  - Something funny about that name...

# Talking to the OS

---

- We are going to be running on MIPS *emulator* called **SPIM**
- We're not actually running our commands on an actual MIPS (hardware) processor!!
  - ...we're letting software *pretend* it's hardware...
  - ...so, in other words... we're “faking it”
- Ok, so how might we print something onto *std.out*?

# SPIM Routines

---

- MIPS features a **syscall** instruction, which triggers a ***software interrupt***, or ***exception***
- Outside of an emulator (i.e. in the real world), these instructions **pause the program** and tell the OS to go do something with I/O
- Inside the emulator, it tells the emulator to go ***emulate*** something with I/O

# syscall

---

- So we have the OS/emulator's attention, but how does it know what we want?
- The OS/emulator has access to the CPU registers
- We put special values (codes) in the registers to indicate what we want
  - These are codes that can't be used for anything else, so they're understood to be just for syscall
  - So... is there a "code book"????

Yes! All CPUs come with manuals.  
For us, we have the [MIPS Ref. Card](#)

# syscall Interaction Setup

---

You will need:

- System call code
  - Usually placed in \$v0
- Argument
  - Usually placed in \$a0

# (Finally) Printing an Integer

---

- For SPIM, if register **\$v0** contains **1** and then we issue a **syscall**, then SPIM will *print whatever integer is stored in register \$a0*  
**← this is a specific rule using a specific code**
  - Note: \$v0 is used for other stuff as well – more on that later...
  - When \$v0=1, syscall is *expecting* an integer!
- Other values put into **\$v0** indicate other types of I/O calls to **syscall**  
Examples:
  - \$v0 = 3 means **double (or the mem address of one)** in \$a0
  - \$v0 = 4 means **string (or the mem address of one)** in \$a0
  - \$v0 = 5 means **get user input from std input** and place in \$v0
  - We'll explore some of these later, but check **MIPS ref card** for all of them

# (Finally) Printing an Integer

- Remember, the usual syntax to load immediate a value into a register is:

**li <register>, <value>**

- You can also move (**copy**) the value of one register into another too!

**move <to register>, <from register>**

Example:      move \$a0, \$t0 # PUTS THE VALUE IN REG. \$t0 INTO REG. \$a0

To make sure that the register **\$a0** has the value of what you want to print out (let's say it's in another register, like **\$t0**), use the **move** command:

# Augmenting with Printing

---

```
# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print the integer that's in $t3
# to std.output
li $v0, 1
move $a0, $t3
syscall
```

# Program Files for MIPS Assembly

---

- The files have to be text
- Typical file extension type is **.asm**
- To leave comments,  
use **#** at the start of the line

# We're Not Quite Done Yet!

## Exiting an Assembly Program in SPIM

---

- If you are using SPIM, then you need to say *when you are done as well*
  - Most HLL programs do this for you automatically
- How is this done?
  - Issue a `syscall` with a special value in `$v0 = 10` (decimal)

# Augmenting with Exiting

```
.text      # We always have to have this starting line  
  
# Main program  
li $t0, 5  
li $t1, 7  
add $t3, $t0, $t1  
  
# Print to std.output  
li $v0, 1  
move $a0, $t3  
syscall  
  
# End program  
li $v0, 10  
syscall
```

# Let's Run This Program Already! Using SPIM

---

- We'll call it **simpleadd.asm**
- Run it on CSIL as: `$ spim -f simpleadd.asm`



- We'll also run other arithmetic programs and explain them as we go along
  - TAKE NOTES!

# YOUR TO-DOS

---

- Do readings!
  - Check syllabus for details!
- Get to Assignment #2
  - You have to submit it into ***Gradescope as 2 parts***
    - PDF with answers to questions + Program (in C/C++)
  - Due on **Tuesday 1/21, by 11:59:59 PM**

