

# Week 8, part B:

## ALU Instructions



# Arithmetic instructions

Instruction	Opcode/Function	Syntax	Operation
<b>add</b>	100000	\$d, \$s, \$t	\$d = \$s + \$t
<b>addu</b>	100001	\$d, \$s, \$t	\$d = \$s + \$t
<b>addi</b>	001000	\$t, \$s, i	\$t = \$s + SE(i)
<b>addiu</b>	001001	\$t, \$s, i	\$t = \$s + SE(i)
<b>div</b>	011010	\$s, \$t	lo = \$s / \$t; hi = \$s % \$t
<b>divu</b>	011011	\$s, \$t	lo = \$s / \$t; hi = \$s % \$t
<b>mult</b>	011000	\$s, \$t	hi:lo = \$s * \$t
<b>multu</b>	011001	\$s, \$t	hi:lo = \$s * \$t
<b>sub</b>	100010	\$d, \$s, \$t	\$d = \$s - \$t
<b>subu</b>	100011	\$d, \$s, \$t	\$d = \$s - \$t

**Notes:** "hi" and "lo" refer to the HI and LO registers (see register slide).  
"SE" = "sign extend".



# R-type vs I-type arithmetic

## R-Type

- `add, addu`
- `div, divu`
- `mult, multu`
- `sub, subu`

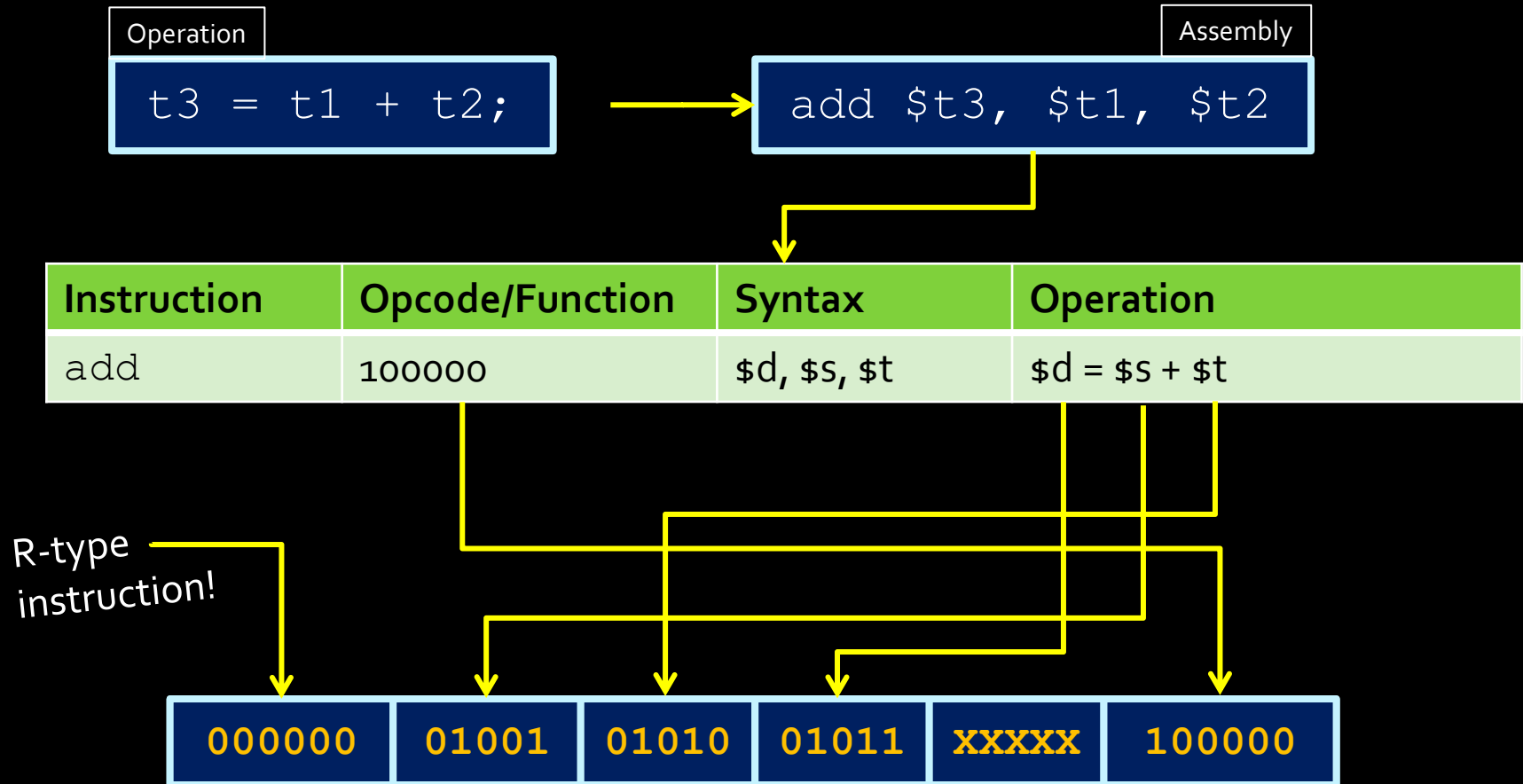
## I-Type

- `addi`
- `addiu`

- In general, most instructions are R-type (meaning all operands are registers) and some are I-type (meaning they use an immediate/constant value in their operation).
- Can you recognize which of the following are R-type and I-type instructions? (Hint: “i” for “immediate”)



# Assembly → Machine Code



Although we specify “don’t care” bits as X values, in practice the assembler generally sets them to zero



# Unsigned Instructions

- What is the difference between **add** or **addu**?
  - Both do exactly same thing! Add numbers.
- “**u**” stands for “**unsigned**”
  - Causes a “trap” (a.k.a exception) if there is overflow
  - Stops execution of current code.
  - **addu** ignores this overflow
- **mult** and **multu** are not the same!
  - Slight difference in operation. Use the right one!
  - Neither check for overflow.



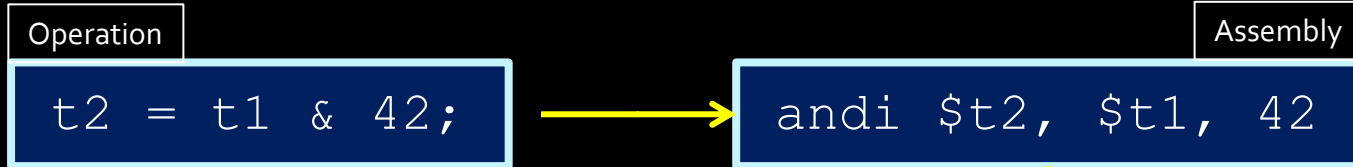
# Logical instructions

Instruction	Opcode/Function	Syntax	Operation
and	100100	\$d, \$s, \$t	$\$d = \$s \& \$t$
andi	001100	\$t, \$s, i	$\$t = \$s \& \text{ZE}(i)$
nor	100111	\$d, \$s, \$t	$\$d = \sim(\$s \mid \$t)$
or	100101	\$d, \$s, \$t	$\$d = \$s \mid \$t$
ori	001101	\$t, \$s, i	$\$t = \$s \mid \text{ZE}(i)$
xor	100110	\$d, \$s, \$t	$\$d = \$s \wedge \$t$
xori	001110	\$t, \$s, i	$\$t = \$s \wedge \text{ZE}(i)$

Note: ZE = zero extend (pad upper bits with 0 value).



# Assembly → Machine Code II



Instruction	Opcode/Function	Syntax	Operation
andi	001100	\$t, \$s, i	\$t = \$s & ZE(i)

I-type  
instruction!



# Shift instructions

Instruction	Opcode/Function	Syntax	Operation
sll	000000	\$d, \$t, a	\$d = \$t << a
sllv	000100	\$d, \$t, \$s	\$d = \$t << \$s
sra	000011	\$d, \$t, a	\$d = \$t >> a
srav	000111	\$d, \$t, \$s	\$d = \$t >> \$s
srl	000010	\$d, \$t, a	\$d = \$t >>> a
srlv	000110	\$d, \$t, \$s	\$d = \$t >>> \$s

- Order is **\$d, \$t, \$s** or **\$d, \$t, a** (not \$d, \$s, \$t as before!)
- srl = “shift right logical”
- sra = “shift right arithmetic”.
- The “v” denotes a variable number of bits, specified by \$s.
- a is **shift amount**, and is stored in **shamt** when encoding the R-type machine code instructions.





# Data movement instructions

Instruction	Opcode/Function	Syntax	Operation
mfhi	010000	\$d	\$d = hi
mflo	010010	\$d	\$d = lo
mthi	010001	\$s	hi = \$s
mtlo	010011	\$s	lo = \$s

- These are instructions for operating on the HI and LO registers described earlier (for multiplication and division)



# lui – load upper immediate

Instruction	Opcode/Function	Syntax	Operation
lui	001111	\$t, i	\$t = i << 16

- Load 16-bit immediate into upper half of the register.
- The lower 16 bits of the register are set to zero.

iiiiiiiiiiiiiiii0000000000000000





# ALU instructions in RISC

## R type

- add, div, mult, sub
- addu, divu, multu, subu
- or, and, nor, xor

## I type

- addi
- addiu
- andi, ori, xori

- Most ALU instructions are R-type instructions.
  - The six-digit codes in the tables are therefore the function codes (opcodes and function codes)
  - Except the few I-type instructions (addi, andi, ori, etc.)



# ALU instructions in RISC

- Not all R-type instructions have an I-type equivalent.
  - We have `addi` but not `subi`
  - We have `ori` but not `nori`
  - `div` but not `divi`
- RISC principle: an operation doesn't need an instruction if it can be performed through multiple existing operations.
  - `addi $t0, -1` → “`subi`” `$t0, 1`
  - `addi + div` → “`divi`”



# Pseudoinstructions

- Pseudo instructions **look** like assembly instructions...
- ...but don't have a dedicated machine code instruction.
- Provided by the assembler
  - Mapping ASM to machine code is more like a many-to-one mapping...
- If a temporary register is needed, use **\$at**



# Pseudoinstructions

- Move data from \$t4 to \$t5?

- `move $t5,$t4` →

```
add $t5,$t4,$zero
```

- Multiply and store in \$s1?

- `mul $s1, $t4, $t5` →

```
mult $t4,$t5  
mflo $s1
```

- Load a 32-bit immediate?

- `li $s0, 0x1234ABCD` →

```
lui $s0,$s0,0x1234  
ori $s0,$s0,0xABCD
```



# Time to write our first assembly program



# Making an assembly program

- Assembly language programs typically have structure similar to **simple** Python or C programs:
  - They set aside registers to store data.
  - They have sections of instructions that manipulate this data.
- It is always good to decide at the beginning which registers will be used for what purpose!
  - More on this later 😊





# Compute $\text{result} = a^2 + 2b + 10$

- Set up values in registers
  - ▣  $a \rightarrow \$t0, b \rightarrow \$t1$
- **temp = 10**
- $\text{temp} = \text{temp} + b$
- $\text{temp} = \text{temp} + b$  (again!)
- **result = a\*a**
- $\text{result} = \text{result} + \text{temp}$

```
addi $t0, $zero, 7  
addi $t1, $zero, 9
```

```
addi $t6, $zero, 10
```

```
add $t6, $t6, $t1  
add $t6, $t6, $t1
```

```
mult $t0, $t0  
mflo $t4
```

```
add $t4, $t4, $t6
```



# Formatting Assembly Code

- Start file with `.text`
  - (we'll see other options later)
- Follow this with:
- `.globl main`
  - Makes the main label visible to the OS
- `main:`
  - Tells OS which line of code should run first.
- Write instructions, up to 3 columns per line
  - `label: <instr> <params> # comments`
  - Labels and comments as needed
- At the end of the program, tell the OS to finish:  
`li $v0, 10`  
`syscall`

```
.text

.globl main
main:
    <code>

    li $v0, 10
    syscall
```



# Compute the following result:  $r = a^2 + 2b + 10$   
.text

.globl main

```
main:  addi $t0, $zero, 7    # set a=7 for testing
        addi $t1, $zero, 9    # set b=9 for testing
# $t0 will be a,    $t1 will be b,    $t5 will be r
# $t6 will be temp
        addi $t6, $zero, 10    # add 10 to r
        add $t6, $t6, $t1      # then add b
        add $t6, $t6, $t1      # then add b again
        mult $t0, $t0          # multiply a * a
        mflo $t4               # move the low result of a^2
                                # into the register for r
        add $t4, $t4, $t6      # add the temporary value
                                # (2b + 10) to the result

        addi $v0, $zero, 10    # end program
        syscall
```



# How can we run this?

- We don't have a MIPS CPU handy.
- We'll use a **simulator** instead.
- A program that simulates the operation of the MIPS CPU on your own computer,



# Simulating MIPS

## The MARS Simulator

C:\Dropbox\UofT\Teaching\B58 Winter 2021\Material\W8\code\week8\_intro.asm - MARS 4.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

**Text Segment**

Bkpt	Address	Code	Basic
	0x00400000	0x20080007	addi \$s,\$0,0x00000007 11: main: addi \$t0,\$zero, 7 # set a=7 for testing
	0x00400004	0x20090009	addi \$s,\$0,0x00000009 12: addi \$t1,\$zero, 9 # set b=9 for testing
	0x00400008	0x200e000a	addi \$t4,\$0,0x0000000a 14: addi \$t6,\$zero, 10 # add 10 to r
	0x0040000c	0x01c97020	add \$t4,\$t4,\$s 15: add \$t6,\$t6,\$t1 # then add b
	0x00400010	0x01c97020	add \$t4,\$t4,\$s 16: add \$t6,\$t6,\$t1 # then add b again
	0x00400014	0x01080018	mult \$s,\$s 17: mult \$t0,\$t0 # now we need to multiply a * a
	0x00400018	0x00006012	mflo \$t2 18: mflo \$t4 # move the low result of a^2 into the low register of r
	0x0040001c	0x00006010	mflo \$t3 19: mflo \$t5 # move the high result of a^2 into the high register of r
	0x00400020	0x018e6020	add \$t2,\$t2,\$t4 20: add \$t4,\$t4,\$t6 # add the temporary value (2b + 10) to the low register of r
	0x00400024	0x2002000a	addi \$t2,\$0,0x0000000a 23: addi \$v0,\$zero, 10
	0x00400028	0x0000000c	syscall 24: syscall

**Data Segment**

Address	Value (+0)	Value (+4)	Value (+8)
0x10010000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000

**Registers** Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$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	0x00000000
\$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
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

**Mars Messages** Run I/O

Assembly: assembling C:\Dropbox\UofT\Teaching\B58 Winter 2021\Material\W8\code\week8\_intro.asm

Clear Assembly: operation completed successfully.

# MARS Simulator

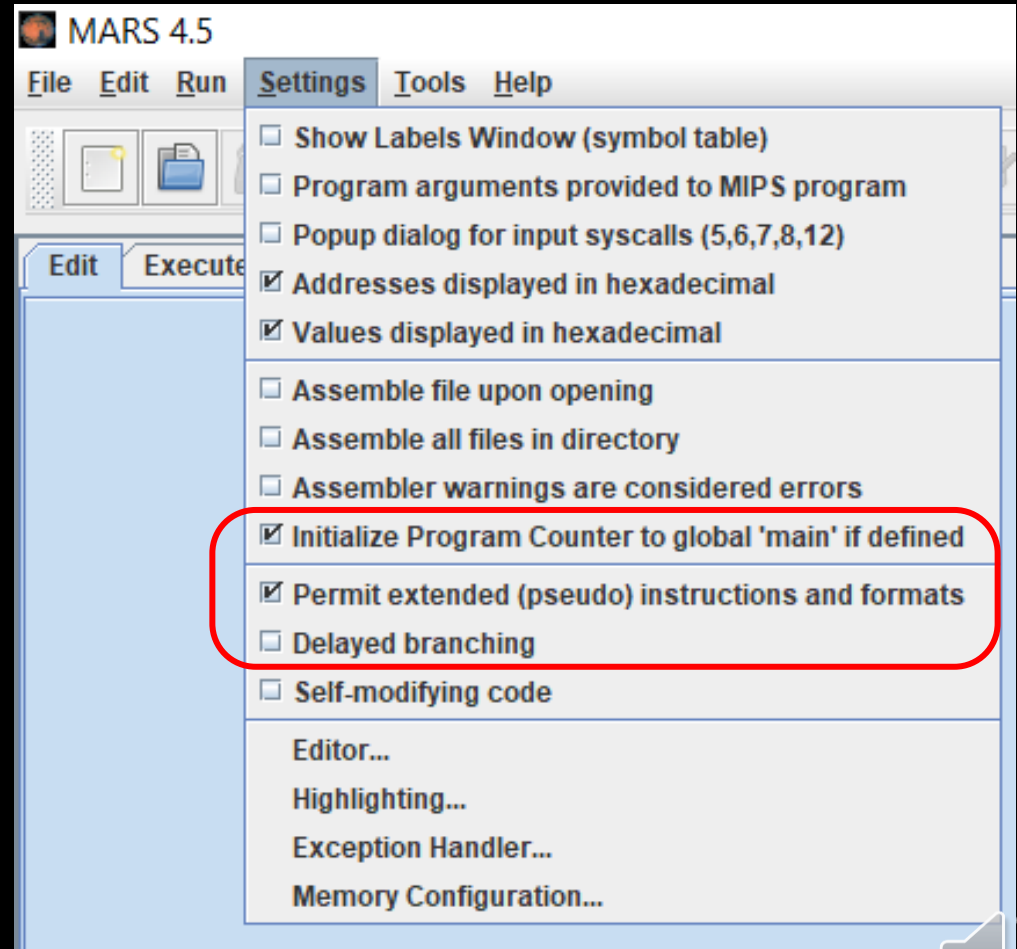
- MARS Simulator official site:
  - <http://courses.missouristate.edu/kenvollmar/mars/>
  - As with Logisim, you will need Java.
- Official version sometimes freezes during debugging.
- Download alternative MARS jar file from Quercus **module on assembly**



# MARS Settings

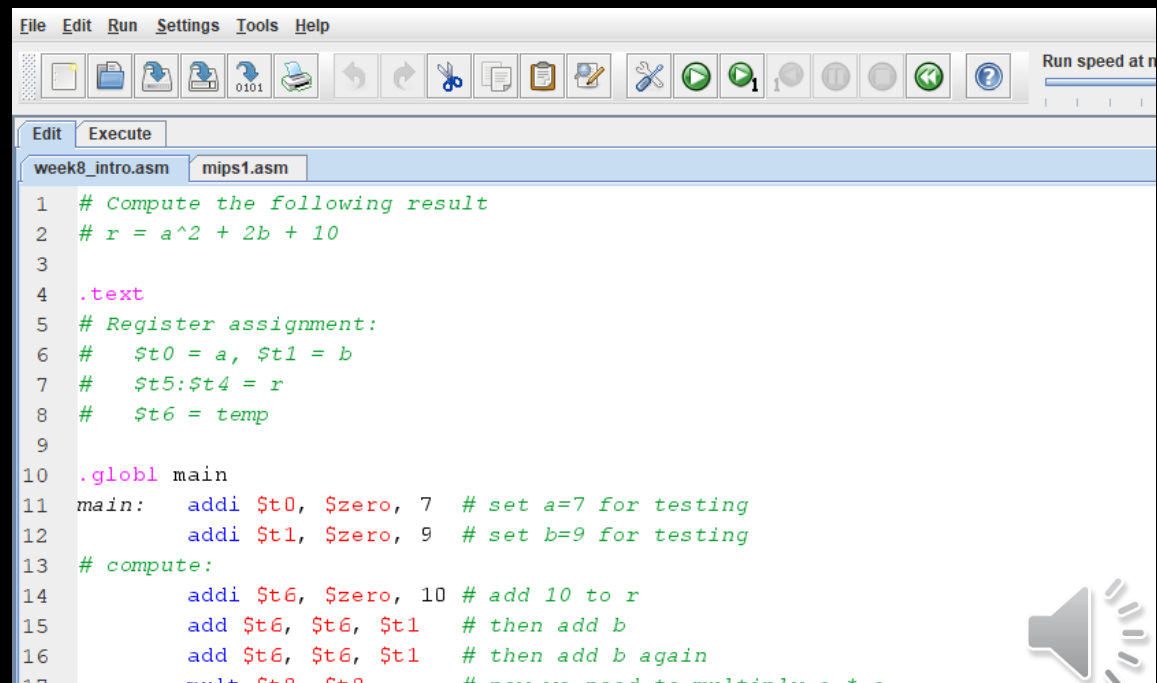
Make sure:

- delayed branching is turned **off**
- permit extended instruction **is on**
- Initialize program counter to global 'main' is **on**



# MARS HowTo

- MARS works like a simple IDE
  - Write assembly program in code editor
  - Save it to an .asm or .s file (doesn't matter)



The screenshot displays the MARS MIPS IDE interface. At the top is a menu bar with 'File', 'Edit', 'Run', 'Settings', 'Tools', and 'Help'. Below the menu is a toolbar with icons for file operations (new, open, save, print), editing (undo, redo, cut, copy, paste), and execution (run, step through, step over, step under, break, watch, help). The main window is titled 'mips1.asm' and contains the following assembly code:

```
1  # Compute the following result
2  # r = a^2 + 2b + 10
3
4  .text
5  # Register assignment:
6  # $t0 = a, $t1 = b
7  # $t5:$t4 = r
8  # $t6 = temp
9
10 .globl main
11 main:  addi $t0, $zero, 7  # set a=7 for testing
12        addi $t1, $zero, 9  # set b=9 for testing
13 # compute:
14        addi $t6, $zero, 10 # add 10 to r
15        add $t6, $t6, $t1   # then add b
16        add $t6, $t6, $t1   # then add b again
17        mult $t0, $t0
```

A speaker icon is visible in the bottom right corner of the window.



# MARS HowTo

- MARS works like a simple IDE
  - Assemble the program (F3 or Run→Assemble)
  - Mars will switch to execute view

Bkpt	Address	Code	Basic	
	0x00400000	0x20080007	addi \$8,\$0,0x00000007	11: main: addi \$t0, \$zero, 7 # set a=7 for testing
	0x00400004	0x20090009	addi \$9,\$0,0x00000009	12: addi \$t1, \$zero, 9 # set b=9 for testing
	0x00400008	0x200e000a	addi \$14,\$0,0x0000000a	14: addi \$t6, \$zero, 10 # add 10 to r
	0x0040000c	0x01c97020	add \$14,\$14,\$9	15: add \$t6, \$t6, \$t1 # then add b
	0x00400010	0x01c97020	add \$14,\$14,\$9	16: add \$t6, \$t6, \$t1 # then add b again
	0x00400014	0x01080018	mult \$8,\$8	17: mult \$t0, \$t0 # now we need to multiply a * a
	0x00400018	0x00006012	mflo \$12	18: mflo \$t4 # move the low result of a^2 into the low
	0x0040001c	0x00006810	mfhi \$13	19: mfhi \$t5 # move the high result of a^2 into the hi
	0x00400020	0x018e6020	add \$12,\$12,\$14	20: add \$t4, \$t4, \$t6 # add the temporary value (2b + 10) to th
	0x00400024	0x2002000a	addi \$2,\$0,0x0000000a	23: addi \$v0, \$zero, 10
	0x00400028	0x0000000c	syscall	24: syscall



# MARS HowTo

- MARS works like a simple IDE
  - Run the whole program (F5 or Run→Go)
  - Or execute line by line (F7 or Run→Step)

C:\Dropbox\UofT\teaching\B58 Winter 2021\Material\W8\code\week8\_intro.asm - MARS 4.5

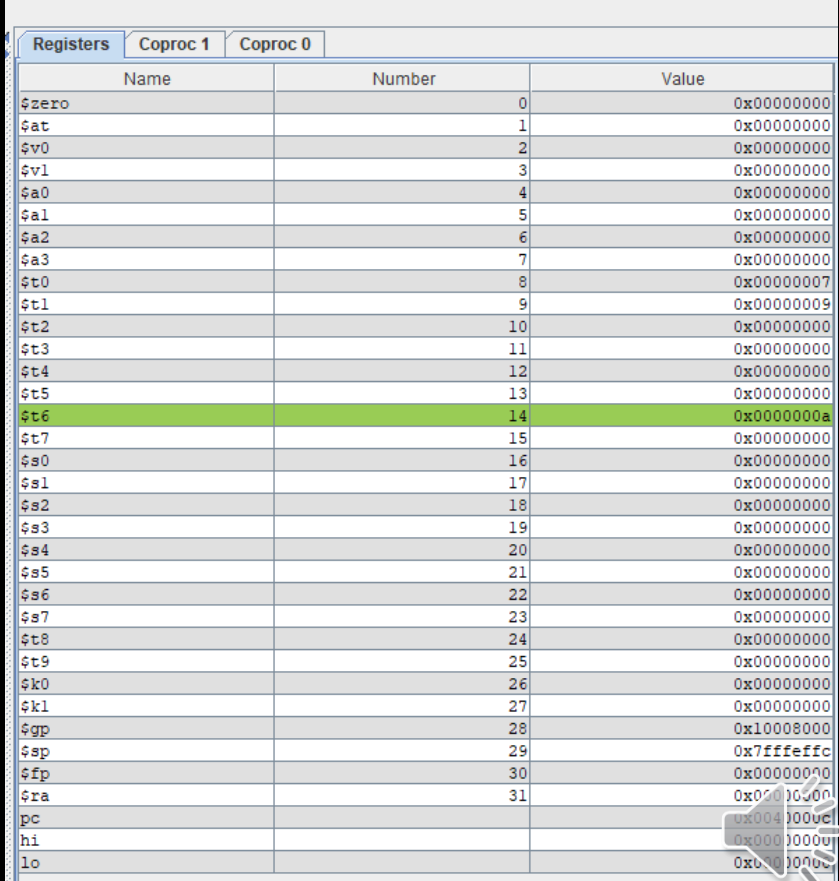
File Edit Run Settings Tools Help

Text Segment

Bkpt	Address	Code	Basic
	0x00400000	0x20080007	addi \$8,\$0,0x00000007 11: main: addi \$t0, \$zero, 7 # set a=7 for test
	0x00400004	0x20090009	addi \$9,\$0,0x00000009 12: addi \$t1, \$zero, 9 # set b=9 for test
	0x00400008	0x200e000a	addi \$14,\$0,0x0000000a 14: addi \$t6, \$zero, 10 # add 10 to r
	0x0040000c	0x01c97020	add \$14,\$14,\$9 15: add \$t6, \$t6, \$t1 # then add b
	0x00400010	0x01c97020	add \$14,\$14,\$9 16: add \$t6, \$t6, \$t1 # then add b again
	0x00400014	0x01080018	mult \$8,\$8 17: mult \$t0, \$t0 # now we need to r
	0x00400018	0x00006012	mflo \$12 18: mflo \$t4 # move the low res
	0x0040001c	0x00006810	mfhi \$13 19: mfhi \$t5 # move the high re
	0x00400020	0x018e6020	add \$12,\$12,\$14 20: add \$t4, \$t4, \$t6 # add the temporar
	0x00400024	0x2002000a	addi \$2,\$0,0x0000000a 23: addi \$v0, \$zero, 10
	0x00400028	0x0000000c	syscall 24: syscall

# MARS HowTo

- MARS works like a simple IDE
  - Check the register window to see what is going on.



Registers			
		Coproc 1	Coproc 0
Name	Number	Value	
\$zero	0	0x00000000	
\$at	1	0x00000000	
\$v0	2	0x00000000	
\$v1	3	0x00000000	
\$a0	4	0x00000000	
\$a1	5	0x00000000	
\$a2	6	0x00000000	
\$a3	7	0x00000000	
\$t0	8	0x00000007	
\$t1	9	0x00000009	
\$t2	10	0x00000000	
\$t3	11	0x00000000	
\$t4	12	0x00000000	
\$t5	13	0x00000000	
\$t6	14	0x0000000a	
\$t7	15	0x00000000	
\$s0	16	0x00000000	
\$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	
\$k1	27	0x00000000	
\$gp	28	0x10008000	
\$sp	29	0x7ffffc	
\$fp	30	0x00000000	
\$ra	31	0x00000000	
pc		0x00400000	
hi		0x00000000	
lo		0x00000000	

# Get MARS

- You need it for labs.
- And for practice.
- Code from lectures is available on OneDrive.
- Try to execute it on MARS.
- See how it works.
- Play with it yourself.
- Learn!



$$r = (2a + 5) * (7b)$$

```
.text
.globl main
# $t0 = a, $t1 = b, $t4 = r
# $t7 = left side, $t8 = right side
main: addi $t0, $zero, 7 # load up some values to test
      addi $t1, $zero, 9
# calculate left side
calc_left:  add $t7, $t0, $t0 # ls <- 2a
            addi $t7, $t7, 5  # ls <- ls + 5

# calculate right side
calc_right: addi $t8, $zero, 7 # rs <- 7
            mult $t8, $t1      # multiply 7 * b
            mflo $t8           # put result back into rs

# multiply left * right and put result into r
multiply:   mult $t7, $t8
            mflo $t4
```



# Implement $c = \max(a, b)$ ?

- Most code does not simply execute linearly from start to finish.
- For example, how would we implement:

```
if (a>b)
    c = a;
else
    c = b;
```

- Move to next part!

