Assembly Language and Computer Architecture Lab

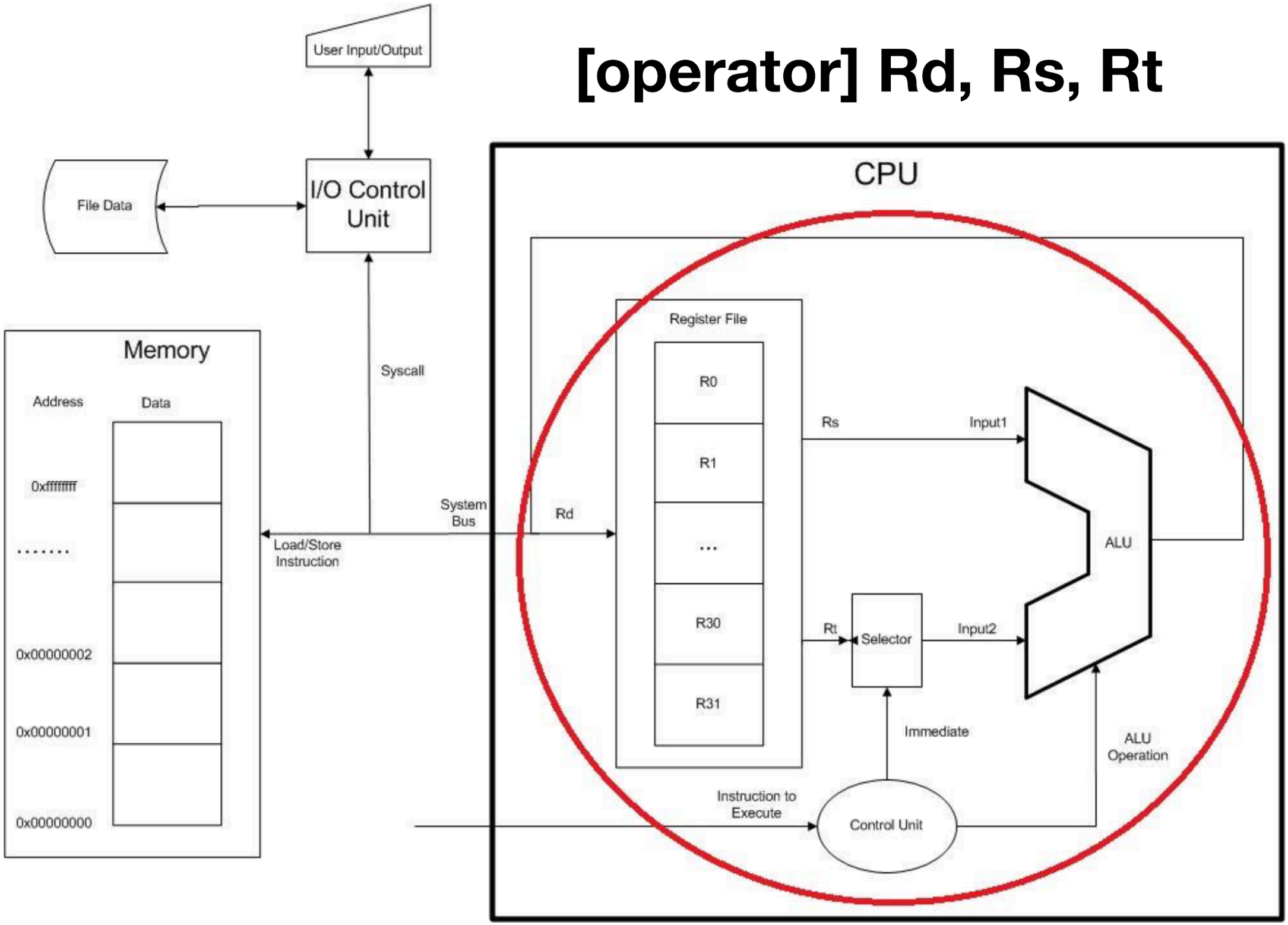
Nguyen Thi Thanh Nga
Dept. Computer engineering
School of Information and Communication Technology

Week 3

- 3-Address machines.
- The difference between real (or native) MIPS operators, and pseudo operators.
- Introductory pseudo code.
- MIPS arithmetic operators and how to use them
  - Addition
  - Subtraction
  - Multiplication
  - Division

3-Address machines

[operator] Rd, Rs, Rt



Types of operator

- Register operator (R)
- Immediate operator (I)
- Jump operator (J)

Register operator

- Syntax:

  [operator] R_d, R_s, R_t

- In which: **$R_d \leftarrow R_s$ [operator] R_t**
  - R_d: destination register, to write the result to
  - R_s: the first input register
  - R_t: the second input register

Immediate operator

- Syntax:

  [operator]i R_t, R_s, Immediate value

- In which: **$R_t \leftarrow R_s$ [operator]i Immediate value**

  - R_t: destination register, to write the result to
  - R_s: the first input register
  - Immediate value: the second input value

Arithmetic Operators

- Addition
- Subtraction
- Multiplication
- Division

Addition in MIPS assembly

- There are 4 real addition operators in MIPS assembly:

  - **add** operator
  - **addi** operator
  - **addu** operator
  - **addiu** operator

add operator

- Takes the value of the $\mathbf{R_s}$ and $\mathbf{R_t}$ registers containing integer numbers, adds the numbers, and stores the value back to the $\mathbf{R_d}$ register.

- The format and meaning are:

  format: **add R_d, R_s, R_t**

  meaning: $\mathbf{R_d \leftarrow R_s + R_t}$

addi operator

- Takes the value of **R_s**, adds the 16 bit immediate value in the instruction, and stores the result back in **R_t**.
- The format and meaning are:

format: **addi R_d, R_s, Immediate**

meaning: **$R_d \leftarrow R_s$ + Immediate**

addu operator

- The same as the **add** operator, except that the values in the registers are assumed to be unsigned, or whole, binary numbers.

- There are no negative values, so the values run from $0...2^{32}-1$

- The format and the meaning are the same as the add operator above:

  format: **addu R_d, R_s, R_t**

  meaning: **$R_d \leftarrow R_s + R_t$**

addiu operator

- The same as the **addi** operator, but again the numbers are assumed to be unsigned.

- The format and meaning are:

format: **addiu R_d, R_s, Immediate**

meaning: **$R_d \leftarrow R_s$ + Immediate**

Pseudo add operator

- Uses a 16 bit immediate value

- This is shorthand for the **add** operator to implement an **addi** operator. The same principal applies for the **addu** if an immediate value is used, and the operator is converted into an **addiu**.

- The format, meaning, and translation of this instruction is:

  format: **add R_t, R_s, Immediate**

  meaning: **$R_t \leftarrow R_s$ + Immediate**

  translation: **addi R_t, R_s, Immediate**

Pseudo add operators

- **add**, **addi**, **addu**, or **addiu** with a 32 bit immediate value.

- If an immediate instruction contains a number which value is more than 16 bit, the number must be loaded in two steps:

  - **1st step**: load the upper 16 bits of the number into a register using the Load Upper Immediate (**lui**) operator

  - **2nd step**: load the lower 16 bits using the using an Or Immediate (**ori**) operator

  The addition is then done using the R instruction **add** operator.

- Instruction **addi R_t, R_s, Immediate (32 bits)** would be translated to:

  lui $at, (upper 16 bits) Immediate #load upper 16 bits into $at

  ori $at, $at, (lower 16 bits) Immediate #load lower 16 bits into $at

  add R_t, R_s, $at

Program 3.1 Addition Example

```
1   # File: Program 3-1.asm
2   # Author: NTTNga
3   # Purpose: To illustrate some addition operators
4
5   # illustrate R format add operator
6   li $t1,100
7   li $t2,50
8   add $t0,$t1,$t2
9
10  # illustrate add with an immediate. Note that
11  # an add with a pseudo instruction translated
12  # into an addi instruction
13  addi $t0,$t0,50
14  add $t0,$t0,50
15
16  # using an unsign number. Note that the
17  # result is not what is expected
18  # for negative numbers.
19  addiu $t0,$t2,-100
20
21  # addition using a 32 immediate. Note that 5647123
22  # base 10 is 0x562b13
23  addi $t1,$t2,5647123
```

Program 3.1 Addition Example

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50
	0x00400010	0x21080032	addi $8,$8,0x00000032	14: add $t0, $t0, 50
	0x00400014	0x2548ff9c	addiu $8,$10,0xffff...	19: addiu $t0, $t2, -100
	0x00400018	0x3c010056	lui $1,0x00000056	23: addi $t1, $t2, 5647123
	0x0040001c	0x34212b13	ori $1,$1,0x00002b13	
	0x00400020	0x01414820	add $9,$10,$1	

- The first column is titled *Source*
- Contains the program exactly as entered.

Program 3.1 Addition Example

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50
	0x00400010	0x21080032	addi $8,$8,0x00000032	14: add $t0, $t0, 50
	0x00400014	0x2548ff9c	addiu $8,$10,0xffff...	19: addiu $t0, $t2, -100
	0x00400018	0x3c010056	lui $1,0x00000056	23: addi $t1, $t2, 5647123
	0x0040001c	0x34212b13	ori $1,$1,0x00002b13	
	0x00400020	0x01414820	add $9,$10,$1	

- The second column is titled *Basic*
- Contains the source code as it is given to the assembler.

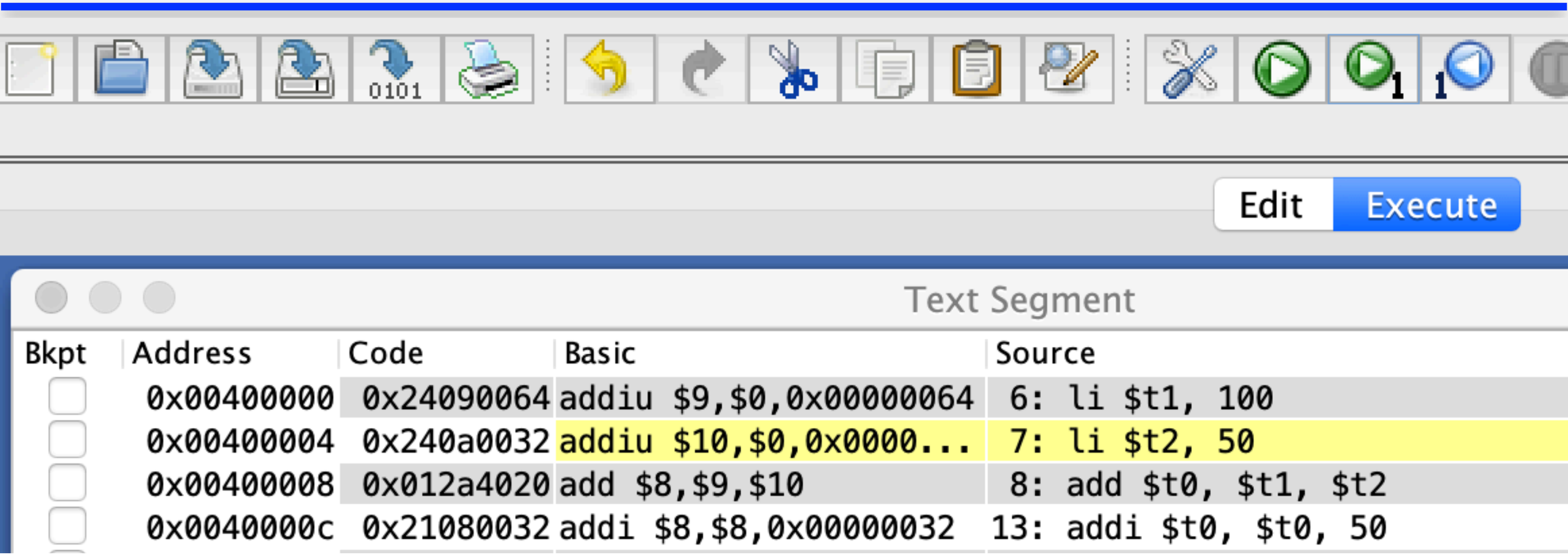
Program 3.1 Addition Example

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50
	0x00400010	0x21080032	addi $8,$8,0x00000032	14: add $t0, $t0, 50
	0x00400014	0x2548ff9c	addiu $8,$10,0xffff...	19: addiu $t0, $t2, -100
	0x00400018	0x3c010056	lui $1,0x00000056	23: addi $t1, $t2, 5647123
	0x0040001c	0x34212b13	ori $1,$1,0x00002b13	
	0x00400020	0x01414820	add $9,$10,$1	

- The first line of the program which is highlighted in yellow means the program is ready to execute at the first line in the program.

Program 3.1 Addition Example



- The program has executed the first line of the program, and is waiting to run at the second line of the program.

- As a result of running the first line, the register **$t1** (**$9**) has been updated to contain the value **100 (0x64)**.

$t1	9	0x00000064

Program 3.1 Addition Example

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
☐	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
☐	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
☐	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50

- Continue running the program
- The next line to execute will always be highlighted in yellow,
- The last register to be changed will be highlighted in green.

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	0x00000064
$t2	10	0x00000032
$t3	11	0x00000000

Program 3.1 Addition Example

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
☐	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
☐	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
☐	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50

- When line 7 is run, line 8 is yellow.
- **$t2** (**$10**) contains the value **0x32** (**5010**) and is highlighted in green.

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	0x00000064
$t2	10	0x00000032
$t3	11	0x00000000

Program 3.1 Addition Example

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50
	0x00400010	0x21080032	addi $8,$8,0x00000032	14: add $t0, $t0, 50
	0x00400014	0x2548ff9c	addiu $	

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	0x00000096
$t1	9	0x00000064

- After the addition at line 8, line 13 is highlighted in yellow
- Register **$t0** (**$8**) contains **0x96** (or **150_{10}**).

Program 3.1 Addition Example

Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
☐	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
☐	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
☐	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50
☐	0x00400010	0x21080032	addi $8,$8,0x00000032	14: add $t0, $t0, 50
☐	0x00400014	0x2548ff9c	addiu $8,$10,0xffff...	19: addiu $t0, $t2, -100
☐	0x00400018	0x3c010056	lui $1,0x00000056	23: addi $t1, $t2, 5647123

- Continue to step through the program until line 19 highlighted and ready to run.

- At this point register **$t0** has the value **0xfa (250_{10})**.

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	0x000000fa
$t1	9	0x00000064

Program 3.1 Addition Example

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24090064	addiu $9,$0,0x00000064	6: li $t1, 100
	0x00400004	0x240a0032	addiu $10,$0,0x0000...	7: li $t2, 50
	0x00400008	0x012a4020	add $8,$9,$10	8: add $t0, $t1, $t2
	0x0040000c	0x21080032	addi $8,$8,0x00000032	13: addi $t0, $t0, 50
	0x00400010	0x21080032	addi $8,$8,0x00000032	14: add $t0, $t0, 50
	0x00400014	0x2548ff9c	addiu $8,$10,0xffff...	19: addiu $t0, $t2, -100
	0x00400018	0x3c010056	lui $1,0x00000056	23: addi $t1, $t2, 5647123
	0x0040001c	0x34212b13	ori $1,	
	0x00400020	0x01414820	add $9,	

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	0xfffffffce
$t1	9	0x00000064

- Line 19 is executed
- The value in **$t0** changes from **0xfa** (**250**) to **0xffffffce** (-50_{10}), not **0x96** (150_{10}) as expected.

Subtraction in MIPS assembly

- Subtraction in MIPS assembly is similar to addition with one exception.

- The **sub** and **subu** behave like the **add** and **addu** operators.

- The **subi** and **subui** are not a real instructions. They are implemented as pseudo instructions.

sub operator

- Takes the value of the $\mathbf{R_s}$ and $\mathbf{R_t}$ registers containing integer numbers, does the operation, and stores the value back to the $\mathbf{R_d}$ register.

- The format and meaning are:

  format: **sub R_d, R_s, R_t**

  meaning: $\mathbf{R_d \leftarrow R_s - R_t}$

subi pseudo operator

- Takes the value of **R_s**, subtracts the 16 bit immediate value in the instruction, and stores the result back in **R_t**.

- The format, meaning, and translation are:

format: **subi R_d, R_s, Immediate**

meaning: **$R_d \leftarrow R_s$ - Immediate**

translation: **addi $at, $zero, Immediate**

sub R_t, R_s, $at

subu operator

- The same as the **addu** operator, the values in the registers are assumed to be unsigned, or whole, binary numbers.

- There are no negative values, so the values run from 0…2^{32}-1.

- The format and the meaning are:

  format: **subu R_d, R_s, R_t**

  meaning: **$R_d \leftarrow R_s - R_t$**

subiu pseudo operator

- The same as the **addiu** operator, the numbers are assumed to be unsigned.

- The format, meaning, and translation are:

format: **subiu R_d, R_s, Immediate**

meaning: **$R_d \leftarrow R_s$ - Immediate**

translation: **addi $at, $zero, Immediate**

subu R_t, R_s, $at

Multiplication in MIPS assembly

- Multiplication and division are more complicated than addition and subtraction, and require the use of two new, special purpose registers, the **hi** and **lo** registers.

- The result of a multiplication can require up to twice as many digits as the input values.

- The **hi** register being used to store the 32 bit larger part of the multiplication.

- The **lo** register being used to the store the 32 bit smaller part of the multiplication.

Multiplication in MIPS assembly

- **mult** operation:

```
mult $t1, $t2      #t1*t2

mflo $t0           #stores the result in $t0
```

Multiplies the value in **$t1** by the value in **$t2**, and stores the result in **$t0**.

- What happens if the result of the multiplication is too big to be stored in a single 32-bit register?

Comment

- For example 3*2=06 and 3*6=18

```
    0011
  * 0010
  ------
  0000 0110
```

the larger part of the answer is 0

→ No overflow

```
    0011
  * 0110
  ------
  0001 0010
```

the larger part of the answer is non-zero

→ Overflow

Comment

- For example 2*(-3)=-6 and 2*(-8)=-16

```
      0010                    0010
   *  1101                 *  1010
   ------                  ------
   1111  1010              1110 1110
```

the high 4-bits are 1111, which is the extension of the sign for -6

→ No overflow

the high 4-bits are 1110, since all 4 bits are not 1, they cannot be the sign extension of a negative number,

→ Overflow

Comment

- For example 2*(-6)

```
          0010
        * 1010
          ----
     1111 0010
```

- The high 4-bits are 1111, so it looks like there is not an overflow.

- However, the low 4 bits show a positive number, so 1111 indicates that the lowest 1 (the red one), is really part of the multiplication result, and not an extension of the sign. This result does show overflow.

Comment

- To show overflow in a result, the result contained in the **hi** register must match:

  - All 0's or all 1's, and
  - The high order (sign) bit of the **lo** register.

Multiplication operators

- **mult** operator
- **mflo** operator
- **mfhi** operator

- **mult** operator
- **mulo** pseudo operator

mult operator

- Multiplies the values of **R_s** and **R_t** and saves it in the **lo** and **hi** registers.

- The format and meaning are:

format: **mult R_s, R_t**

meaning: **[hi, lo] ← R_s * R_t**

mflo operator

- Moves the value from the **lo** register into the $\mathbf{R_d}$ register.

- The format and meaning are:

format: **mflo $\mathbf{R_d}$**

meaning: **$\mathbf{R_d} \leftarrow$ lo**

mfhi operator

- Moves the value from the **hi** register into the $\mathbf{R_d}$ register.

- The format and meaning are:

format: **mfhi $\mathbf{R_d}$**

meaning: **$\mathbf{R_d} \leftarrow$ hi**

mult operator

- Multiplies the values of **R_s** and **R_t** and stores them in **R_d**.

- The format and meaning are:

format: **mult R_d, R_s, R_t**

meaning: **$R_d \leftarrow R_s * R_t$**

mulo pseudo operator

- Multiples the values in $\mathbf{R_s}$ and $\mathbf{R_t}$, and stores them in $\mathbf{R_d}$, checking for overflow.

- If overflow occurs an exception is raised, and the program is halted with an error.

- The format and meaning:

  format: **mulo $\mathbf{R_d}$, $\mathbf{R_s}$, $\mathbf{R_t}$**

  meaning: $\mathbf{R_d} \leftarrow \mathbf{R_s} * \mathbf{R_t}$

Multiplication operator with immediate value

- Both the **mult** and **mulo** operators have an immediate pseudo operator implementation.

- The format, meaning, and translation are:

format: **mult R_d, R_s, Immediate**

meaning: **$R_d \leftarrow R_s$ * Immediate**

translation: **addi $\$R_t$, $zero, Immediate**

mult R_d, R_s, R_t

Multiplication operator with immediate value

- Both the **mult** and **mulo** operators have an immediate pseudo operator implementation.

- The format, meaning, and translation are:

format: **mulo R_d, R_s, Immediate**

meaning: **$R_d \leftarrow R_s$ * Immediate**

translation: **addi $\$R_t$, $zero, Immediate**

mulo R_d, R_s, R_t

Division in MIPS Assembly

- Division, like multiplication requires two registers to produce an answer.
- The **lo** register will contain the quotient
- The **hi** register will contain the remainder

Division operators

- div operator: 3 formats
- rem (remainder) operator: 2 formats

div operator format 1

- The only real format
- Divides $\mathbf{R_s}$ by $\mathbf{R_t}$ and stores the result in the [**hi**,**lo**] register pair with the quotient in the **lo** and the remainder in the **hi**.
- The format and meaning are:

  format: **div R_s, R_t**

  meaning: **[hi, lo] ← R_s/R_t**

div operator format 2

- A 3-address format pseudo instruction

- In addition to executing a division instruction, this pseudo instruction also checks for a zero divide using **bne** and **break** instructions.

- The format, meaning, and translation are:

format: **div R_d , R_s, R_t**

meaning: **[if R_t!=0] $R_d \leftarrow R_s/R_t$**

else break

div operator format 2

translation: **bne R_t, $zero, 0x00000001**

break

div R_s, R_t

mflo R_d

div operator format 3

- A pseudo instruction.
- The format, meaning, and translation are:

format: **div R_d, R_s, Immediate**

meaning: **$R_d \leftarrow R_s$/Immediate**

translation **addi $\$R_t$, $zero, Immediate**

div R_s, R_t

mflo R_d

rem operator format 1

- A pseudo instruction.
- The format, meaning, and translation are:

format: **rem R_d , R_s, R_t**

meaning: **[if R_t!=0] $R_d \leftarrow R_s \% R_t$**

else break

rem operator format 1

translation: **bne R_t, $zero, 0x00000001**

break

div R_s, R_t

mfhi R_d

rem operator format 2

- A pseudo instruction
- The format, meaning, and translation are:

format: **rem R_d, R_s, Immediate**

meaning: **$R_d \leftarrow R_s$/Immediate**

translation: **addi $\$R_t$, $zero, Immediate**

div R_s, R_t

mfhi R_d

Program 3.2 Remainder operator, even/odd number checker

```
1  # File: Program 3-2.asm
2  # Author: Charles W. Kann
3  # Purpose: To have a user enter a number,and print 0 if
4  # the number is even, 1 if the number is odd
5
6  .data
7  prompt: .asciiz "Enter your number: "
8  result: .asciiz "A result of 0 is even, 1 is odd: result = "
9
10 .text
11 .globl main      #Declare lable as global to
12                  #enable referencing from other file
13 main:
14    # Get input value
15    addi $v0,$zero,4      # Write Prompt
16    la $a0,prompt
17    syscall
18
19    addi $v0, $zero, 5    # Retrieve input
20    syscall
21    move $s0, $v0
22
23    # Check if odd or even
24    addi $t0,$zero,2      # Store 2 in $t0
25    div $t0,$s0,$t0       # Divide input by 2
26    mfhi $s1              # Save remainder in $s1
27
28    # Print output
29    addi $v0,$zero,4      # Print result string
30    la $a0,result
31    syscall
32
33    addi $v0,$zero,1      # Print result
34    move $a0,$s1
35    syscall
36
37    #Exit program
38    addi $v0,$zero,10
39    syscall
```

Solving arithmetic expressions in MIPS assembly

- Using the MIPS arithmetic operations covered so far, a program can be created to solve equations.

- For example the following pseudo code program, where the user is prompted for a value of x and the program prints out the results of the equation $5x^2 + 2x + 3$.

```
main
{
        int x = prompt("Enter a value for x: ");
        int y = 5 * x * x + 2 * x + 3;
        print("The result is: " + y);
}
```

Program 3.3 Solving the equation

```
1  # File: Program 3-3.asm
2  # Author: Charles Kann
3  # Purpose: To calculate the result of 5*x*x+2*x+3
4
5  .data
6  prompt: .asciiz "Hay nhap vao gia tri x: "
7  result: .asciiz "Ket qua la: "
8
9  .text
10 .globl main
11 main:
12         # Get input value, x
13         addi $v0,$zero,4
14         la $a0, prompt
15         syscall
16         addi $v0,$zero,5
17         syscall
18         move $s0,$v0
19
--
20         # Calculate the result of 5*x*x+2*x+3 and store it in $s1
21         mul $t0,$s0,$s0
22         mul $t0,$t0,5
23         mul $t1,$s0,2
24         add $t0,$t0,$t1
25         addi $s1,$t0,3
26
27         # Print output
28         addi $v0,$zero,4          # Print result string
29         la $a0,result
30         syscall
31         addi $v0,$zero,1          # Print result
32         move $a0,$s1
33         syscall
34
35         #Exit program
36         addi $v0,$zero,10
37         syscall
```

Program 3.4 Division and accuracy of an equation

- One thing to always keep in mind when using division with integers in any language is that the results are truncated.

- This can lead to errors and different answers depending on the order of evaluation of the terms in the equation.

- For example, most 5th graders knows that "(10/3) *3 = 10", as the 3's should cancel. However in integer arithmetic the result of "10/3 = 3", and so "(10/3) *3 = 9" (not 10).

- However if reversing the order of the operations it could be found that "(10*3) / 3 = 10". This is shown in the following program.

Program 3.4 Division and accuracy of an equation

```
1  # File:      Program3-4.asm
2  # Author:    Charles Kann
3  # Purpose:   To show the difference in result if
4  #            ordering of multiplication and division
5  #            are reversed.
6  .data
7        result1: .asciiz "\n(10/3)*3 = "
8        result2: .asciiz "\n(10*3)/3 = "
9
10 .text
11 .globl main
12 main:
13       addi $s0,$zero,10   #Store 10 and 3 in registers $s0 and $s1
14       addi $s1,$zero,3
15
16       div $s2,$s0,$s1     ` #Write out (10/3) * 3
17       mul $s2,$s2,$s1
18
19       addi $v0,$zero,4    #Print the result 1
20       la $a0,result1
21       syscall
22
23       addi $v0,$zero,1    #Print the result
24       move $a0,$s2
25       syscall
26
27       mul $s2,$s0,$s1     #Write out (10*3)/3
28       div $s2,$s2,$s1
29
30       addi $v0,$zero,4    #Print the result 2
31       la $a0,result2
32       syscall
33
34       addi $v0,$zero,1    #Print the result
35       move $a0,$s2
36       syscall
37
38       addi $v0,$zero,10   #Exit program
39       syscall
```

Assigment 3.1

Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 1
.text
      addi    $s0, $zero, 0x3007 # $s0 = 0 + 0x3007 = 0x3007 ;I-type
      add     $s0, $zero, $0     # $s0 = 0 + 0 = 0             ;R-type
```

Sau đó:

- Sử dụng công cụ gỡ rối, Debug, chạy từng lệnh và dừng lại,
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý
    - Sự thay đổi giá trị của thanh ghi $s0
    - Sự thay đổi giá trị của thanh ghi $pc
- Ở cửa số Text Segment, hãy so sánh mã máy của các lệnh trên với khuôn dạng lệnh để chứng tỏ các lệnh đó đúng như tập lệnh đã qui định
- Sửa lại lệnh lui như bên dưới. Chuyện gì xảy ra sau đó. Hãy giải thích

```
addi  $s0, $zero, 0x2110003d
```

Assigment 3.2

Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 4
.text
     #  Assign X, Y
      addi  $t1, $zero, 5     # X = $t1 = ?
      addi  $t2, $zero, -1    # Y = $t2 = ?
     #  Expression Z = 2X + Y
     add   $s0, $t1, $t1     # $s0 = $t1 + $t1 = X + X = 2X
     add   $s0, $s0, $t2     # $s0 = $s0 + $t2 = 2X + Y
```

Sau đó:

- Sử dụng công cụ gỡ rối, Debug, chạy từng lệnh và dừng lại,
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý
    - Sự thay đổi giá trị của các thanh ghi
    - Sau khi kết thúc chương trình, xem kết quả có đúng không?
- Ở cửa số Text Segment, xem các lệnh **addi** và cho biết điểm tương đồng với hợp ngữ và mã máy. Từ đó kiểm nghiệm với khuôn mẫu của kiểu lệnh I
- Ở cửa số Text Segment, chuyển mã máy của lệnh **add** sang hệ 2. Từ đó kiểm nghiệm với khuôn mẫu của kiểu lệnh R.

Assigment 3.3

Gõ chương trình sau vào công cụ MARS.

```
#Laboratory Exercise 2, Assignment 5
.text
    #  Assign X, Y
    addi  $t1, $zero, 4     # X = $t1 = ?
    addi  $t2, $zero, 5     # Y = $t2 = ?
    #  Expression Z = 3*XY
    mul   $s0, $t1, $t2     # HI-LO = $t1 * $t2 = X * Y  ; $s0 = LO
    mul   $s0, $s0, 3       # $s0 = $s0 * 3 = 3 * X * Y
    #  Z' = Z
    mflo  $s1
```

Sau đó:

- Biên dịch và quan sát các lệnh mã máy trong cửa sổ Text Segment. Giải thích điều bất thường?

- Sử dụng công cụ gỡ rối, Debug, chạy từng lệnh và dừng lại,
- Ở mỗi lệnh, quan sát cửa sổ Register và chú ý
    - Sự thay đổi giá trị của các thanh ghi, đặc biệt là Hi, Lo
    - Sau khi kết thúc chương trình, xem kết quả có đúng không?

Assigment 3.4

- Write programs to evaluate the following expressions. The user should enter the variables, and the program should print back an answer. Prompt the user for all variables in the expression, and print the results in a meaningful manner. **The results should be as accurate as possible.**

a) $5x + 3y + z$

b) $((5x + 3y + z) / 2) * 3$

c) $x^3+2x^2 +3x+4$

d) $(4x / 3) * y$

Assigment 3.5

- The ideal gas law allows the calculation of volume of a gas given the pressure (P), amount of the gas (n), and the temperature (T). The equation is:

$$V = nRT / P$$

- Since we only have used integer arithmetic, all numbers will be integer values with no decimal points. The constant R is 8.314 and will be specified as (8314/1000). This gives the same result.

- Implement the idea gas law program where the user is prompted for and enters values for n, T, and P, and V is calculated and printed out. Be careful to implement an accurate version of this program. Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

Assigment 3.6

- Correct the following programs.

Program 1

```
.text
main:
    li $v0, 4
    la $a0, result1
    syscall
    li $v0, 1
    li $a0, 4
    syscall

    li $v0, 4
    la $a0, result2
    syscall
    li $v0, 1
    li $a0, 8
    syscall

    addi $v0, $zero, 10 #Exit program
    syscall
.data
result1: .ascii "\nfirst value = "
result2: .ascii "\nsecond value = "
```

Assigment 3.7

- Correct the following programs.

Program 2

```
.text
main:
    li $v0, 4
    la $a0, result1
    syscall
    li $v0, 4
    li $a0, 4
    syscall

    li $v0, 4
    la $a0, result2
    syscall
    li $v0, 1
    li $a0, 8
    syscall

    addi $v0, $zero, 10 #Exit program
    syscall
.data
result1: .asciiz "\nfirst value = "
result2: .asciiz "\nsecond value = "
```

Assigment 3.8

- Correct the following programs.

Program 3

```
.text
main:
    li $v0, 4
    la $a0, result1
    syscall
    li $v0, 4
    li $a0, 4
    syscall

    li $v0, 4
    la $a0, result2
    syscall
    li $v0, 1
    li $a0, 8
    syscall

    addi $v0, $zero, 10 #Exit program
    syscall
result1: .ascii "\nfirst value = "
result2: .ascii "\nsecond value = "
```

End of week 3