3 Integer Arithmetic

3.1 Objectives

After completing this lab, you will:

- Get familiar with the basic MIPS integer arithmetic and logic instructions including:
  - Integer addition and subtraction instructions
  - Bitwise logic instructions
  - Shift instructions
- Learn some useful applications of these instructions.

3.2 Integer Add/Subtract Instructions

The MIPS add/subtract instructions are shown in Table 3.1 below. The R-type add/sub instructions have three registers where the first register is the destination register while the other two registers are the two registers to be added. Similarly the I-type add/sub instructions have the first register as the destination register, while the second register and the constant are the operands to be either added or subtracted.

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`addu $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`subu $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`addi $s1, $s2, 10`	`$s1 = $s2 + 10`
`addiu $s1, $s2, 10`	`$s1 = $s2 + 10`

Table 3.1: Add/Subtract Instructions

The difference between **`add/sub`** and **`addu/subu`** instructions is that in case of overflow occurrence, the **`add/sub`** instructions will cause an arithmetic exception and the result will not be written to the destination register. However, for the instructions **`addu/subu`**, overflow occurrence is ignored.

As an example of using the **add/sub** instructions, consider the translation of the expression:

```
f = (g+h) – (i+j)
```

Assuming that *f*, *g*, *h*, *i*, and *j* are allocated registers **$s0** thru **$s4**, the following assembly code performs the translation:

```
addu   $t0, $s1, $s2     # $t0 = g + h
addu   $t1, $s3, $s4     # $t1 = i + j
subu   $s0, $t0, $t1     # $s0 = f = (g+h)–(i+j)
```

To illustrate the use of the **addiu** instruction with constants, let us assume that variables **a**, **b**, and **c** are allocated in registers **$s0**, **$s1**, **$s2**. Then,

```
a = b + 5      is translated as:    addiu $s0, $s1, 5
c = b – 1      is translated as:    addiu $s2, $s1, -1
```

3.3 Logical Bitwise Instructions

The MIPS logical instructions are given in Table 3.2. These include the: **and**, **or**, **nor** and **xor** instructions. The operands for these instructions follow the same convention as the **add/sub** instructions. The immediate value for the **andi**, **ori**, and **xori** logical instructions is treated as unsigned constant, while it is treated as a signed constant for the **addi** and **addiu** instructions.

Instruction		Meaning
and	$s1, $s2, $s3	$s1 = $s2 & $s3
or	$s1, $s2, $s3	$s1 = $s2 | $s3
xor	$s1, $s2, $s3	$s1 = $s2 ^ $s3
nor	$s1, $s2, $s3	$s1 = ~($s2|$s3)
andi	$s1, $s2, 10	$s1 = $s2 & 10
ori	$s1, $s2, 10	$s1 = $s2 | 10
xori	$s1, $s2, 10	$s1 = $s2 ^ 10

Table 3.2: Logical Instructions

The truth tables of the **and**, **or**, **xor** and **nor** logical operations are given below:

x	*y*	*x* and *y*
0	0	0
0	1	0
1	0	0
1	1	1

x	*y*	*x* or *y*
0	0	0
0	1	1
1	0	1
1	1	1

x	*y*	*x* xor *y*
0	0	0
0	1	1
1	0	1
1	1	0

x	*y*	*x* nor *y*
0	0	1
0	1	0
1	0	0
1	1	0

The **and** instruction is used to clear bits: **x and 0** is equal to **0**. The **or** instruction is used to set bits: **x or 1** is equal to **1**. The **xor** instruction is used to toggle bits: **x xor 1** is equal to **not *x***. The **nor** instruction can be used as a **not** since **nor $s1,$s2,$s2** is equivalent to **not $s1,$s2**.

As an example of these instructions, assume that **$s1 = 0xabcd1234** and **$s2 = 0xffff0000**. Then, the following logical instructions produce the shown resulting values in registers **$s3** to **$s6**:

```
and  $s3,$s1,$s2      # $s3 = 0xabcd0000
or   $s4,$s1,$s2      # $s4 = 0xffff1234
xor  $s5,$s1,$s2      # $s5 = 0x54321234
nor  $s6,$s1,$s2      # $s6 = 0x0000edcb
```

The sample program to run this code is given below and the resulting register content after executing the program is shown in Figure 3.1.

```
.text
.globl main
main:                       # main program
   li  $s1, 0xabcd1234      # Pseudo instruction to initialize a register
   li  $s2, 0xffff0000
   and $s3,$s1,$s2          # $s3 = 0xabcd0000
   or  $s4,$s1,$s2          # $s4 = 0xffff1234
   xor $s5,$s1,$s2          # $s5 = 0x54321234
   nor $s6,$s1,$s2          # $s6 = 0x0000edcb

   li $v0, 10               # Exit program
   syscall
```

Arithmetic and logic instructions have many useful applications. For example, to convert a character in register **$s0** from lower case (i.e. **'a'** to **'z'**) to upper case (i.e. **'A'** to **'Z'**), we could use any of the following instructions:

```
subi $s0, $s0, 0x20      # ASCII code of 'a' = 0x61, of 'A' = 0x41
andi $s0, $s0, 0xfd
```

Similarly, to convert a character in register **$s0** from upper case to lower case, we could use any of the following instructions:

```
addi $s0, $s0, 0x20
ori $s0, $s0, 0x20
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0xffff0000
$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	0xabcd1234
$s2	18	0xffff0000
$s3	19	0xabcd0000
$s4	20	0xffff1234
$s5	21	0x54321234
$s6	22	0x0000edcb
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000

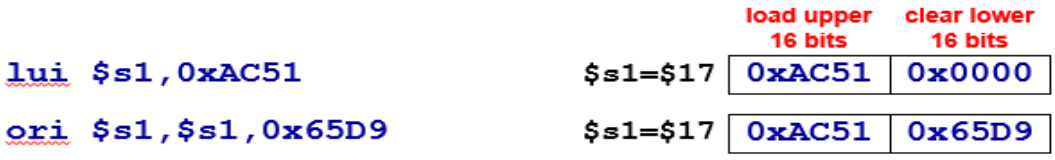
Figure 3.1: Register content after sample code execution.

To initialize the content of register **`$s0`** by a 16-bit constant **`k`** (i.e., having a value in the range 0 to 2^{15}-1), we can use any of the following instructions:

```
addi $s0, $0, k
ori  $s0, $0, k
```

However, to initialize a register with a 32-bit constant, we need to use the **`lui`** instruction.

Suppose that we want to initialize **`$s1`** with the constant **`0xAC5165D9`** (32-bit constant), then we can use the following two instructions:



This sequence of instructions is generated by the assembler when we use the pseudo instruction:

```
li $s1, 0xAC5165D9
```

3.4 Shift Instructions

The MIPS shift instructions are given in **Table 3.3**. The first operand of the shift instructions is the destination register, the second operand is the register to be shifted while the third operand specifies the amount of shift. The amount of shift can be specified as a constant value or it can be stored in a register. For the instructions **`sll, srl, sra,`** the shift amount is a 5-bit constant while for the instructions **`sllv, srlv, srav,`** the shift amount is variable and is stored in a register.

Instruction	Meaning
`sll $s1,$s2,10`	`$s1 = $s2 << 10`
`srl $s1,$s2,10`	`$s1 = $s2>>>10`
`sra $s1,$s2,10`	`$s1 = $s2 >> 10`
`sllv $s1,$s2,$s3`	`$s1 = $s2 << $s3`
`srlv $s1,$s2,$s3`	`$s1 = $s2>>>$s3`
`srav $s1,$s2,$s3`	`$s1 = $s2 >> $s3`

Table 3.3: Shift Instructions.

Shifting is to move all the bits in a register left or right. **`sll/srl`** mean shift left/right logical while **`sra`** means shift right arithmetic for which the sign-bit (rather than 0) is shifted from the left as illustrated in Figure 3.2.

As an example, let us assume that **`$s2 = 0xabcd1234`** and **`$s3 = 16`**. Then, the following shift instructions produce the shown values in **`$s1`**.

```
sll  $s1,$s2,8       $s1 = $s2<<8          $s1 = 0xcd123400
sra  $s1,$s2,4       $s1 = $s2>>4          $s1 = 0xfabcd123
srlv $s1,$s2,$s3     $s1 = $s2>>>$s3       $s1 = 0x0000abcd
```

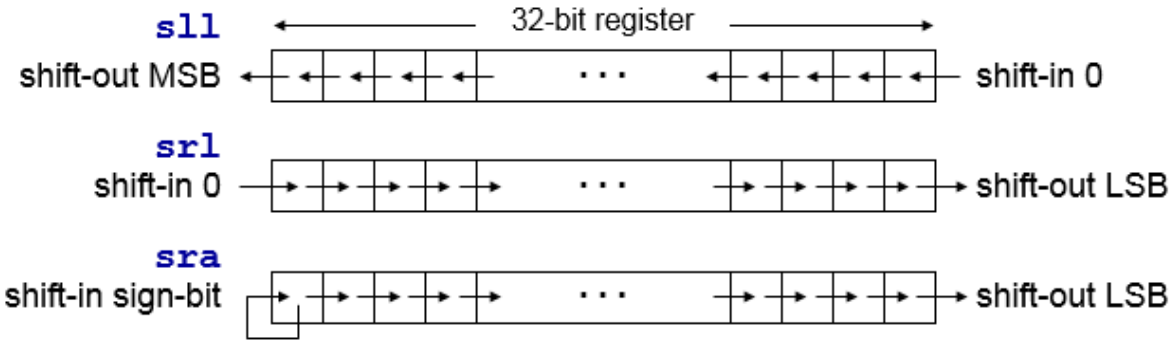


Figure 3.2: Illustration of shift instructions.

We can use shift instructions along with either addition or subtraction instructions to multiply the content of a register by a constant. Shift-left (**sll**) instruction can be used to perform multiplication when the multiplier is a power of **2**. You can factor any binary number into powers of **2**. For example, to multiply **$s1** by **36**, factor **36** into **(4 + 32)** and use distributive property of multiplication **$s2 = $s1*36 = $s1*(4 + 32) = $s1*4 + $s1*32**. Thus, this can be achieved by the following instructions:

```
sll    $t0, $s1, 2      # $t0 = $s1 * 4
sll    $t1, $s1, 5      # $t1 = $s1 * 32
addu   $s2, $t0, $t1    # $s2 = $s1 * 36
```

As another example, let us multiply the content of **$s1** by **31**. We can do that using the following instructions noting that **31=32-1**:

```
sll    $s2, $s1, 5      # $s2 = $s1 * 32
subu   $s2, $s2, $s1    # $s2 = $s1 * 31
```

We can also use the shift right instructions (**srl** and **sra**) to divide a number by a power of **2** constant. Shifting register **$s0** right by **n** bits divides its content by 2^n. For example, to divide an unsigned number in register **$s0** by **8**, we use the instruction **srl $s0, 3**. However, to divide a signed number in register **$s0** by **8**, we use the instruction **sra $s0, 3**.

3.5 In-Lab Tasks

1. Write a program to ask the user to enter two integers **A** and **B** and then display the result of computing the expression: **A + 2B - 5**.
2. Assume that **$s1 = 0x12345678** and **$s2 = 0xffff9a00**. Determine the content of registers **$s3** to **$s6** after executing the following instructions:

```
and $s3,$s1,$s2          # $s3 =
or  $s4,$s1,$s2          # $s4 =
xor $s5,$s1,$s2          # $s5 =
nor $s6,$s1,$s2          # $s6 =
```

Write a program to execute these instructions and verify the content of registers **$s3** to **$s6**.

3. Assume that **$s1 = 0x87654321**. Determine the content of registers **$s2** to **$s4** after executing the following instructions:

```
sll $s2,$s1, 16          # $s2 =
srl $s3,$s1, 8           # $s3 =
sra $s4,$s1, 12          # $s4 =
```

   Write a program to execute these instructions and verify the content of registers **$s2** to **$s4**.

4. Write a program that asks the user to enter an alphabetic character (either lower or upper case) and change the case of the character from lower to upper and from upper to lower and display it.

5. Write a program that asks the user to enter and integer number and read it. Then ask him to enter a bit position (between 0 and 31) and display the value of that bit.

6. Write a program that asks the user to enter a signed number and read it. Then display the content of multiplying this number by **24.5**.

7. Write a program that asks the user to enter an unsigned number and read it. Then swap the bits at odd positions with those at even positions and display the resulting number. For example, if the user enters the number **9**, which has binary representation of **1001**, then bit **0** is swapped with bit **1**, and bit **2** is swapped with bit **3**, resulting in the binary number **0110**. Thus, the program should display **6**.