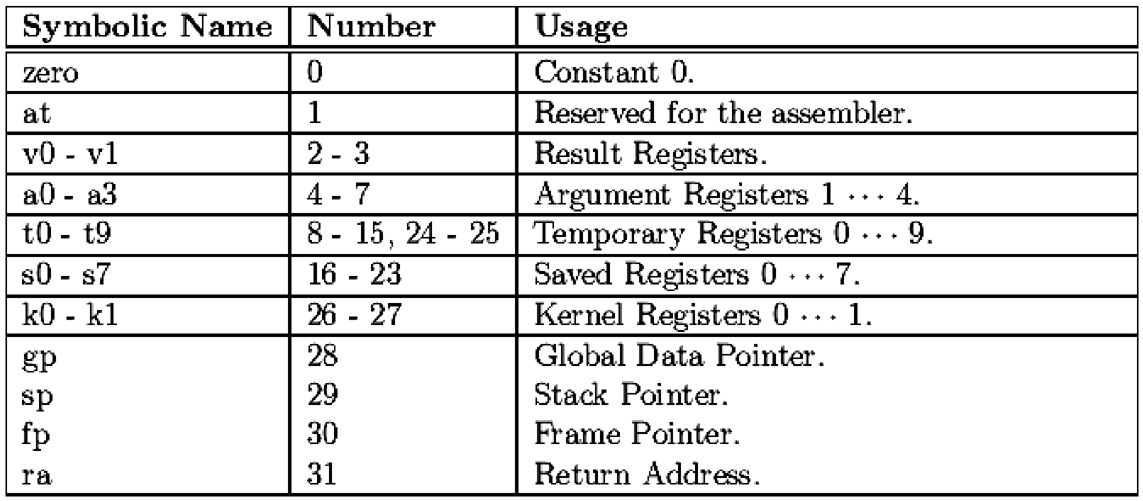
COMPUTER ORGANIZATION AND ARCHITECTURE Lab

Practice Tasks

Registers Details:



Symbolic Name	Number	Usage
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers 1 ⋯ 4.
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 ⋯ 9.
s0 - s7	16 - 23	Saved Registers 0 ⋯ 7.
k0 - k1	26 - 27	Kernel Registers 0 ⋯ 1.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

SPIM Assembler

Name	Parameters	Description
`.data`	*addr*	The following items are to be assembled into the data segment. By default, begin at the next available address in the data segment. If the optional argument *addr* is present, then begin at *addr*.
`.text`	*addr*	The following items are to be assembled into the text segment. By default, begin at the next available address in the text segment. If the optional argument *addr* is present, then begin at *addr*. In SPIM, the only items that can be assembled into the text segment are instructions and words (via the `.word` directive).
`.kdata`	*addr*	The kernel data segment. Like the data segment, but used by the Operating System.
`.ktext`	*addr*	The kernel text segment. Like the text segment, but used by the Operating System.
`.extern`	*sym size*	Declare as global the label *sym*, and declare that it is *size* bytes in length (this information can be used by the assembler).
`.globl`	*sym*	Declare as global the label *sym*.

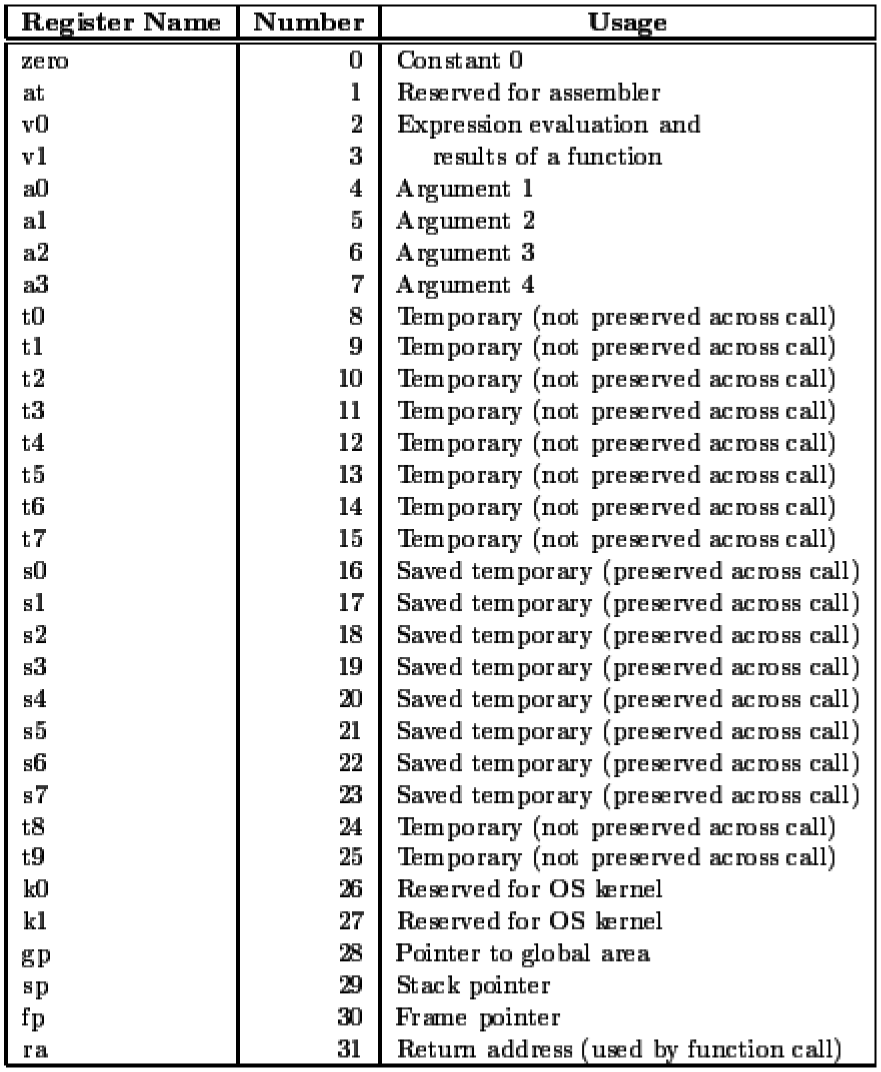
SPIM data directives:

Name	Parameters	Description
`.align`	n	Align the next item on the next 2^n-byte boundary. `.align 0` turns off automatic alignment.
`.ascii`	*str*	Assemble the given string in memory. Do not null-terminate.
`.asciiz`	*str*	Assemble the given string in memory. Do null-terminate.
`.byte`	*byte1* $\cdots$ *byteN*	Assemble the given bytes (8-bit integers).
`.half`	*half1* $\cdots$ *halfN*	Assemble the given halfwords (16-bit integers).
`.space`	*size*	Allocate n bytes of space in the current segment. In SPIM, this is only permitted in the data segment.
`.word`	*word1* $\cdots$ *wordN*	Assemble the given words (32-bit integers).

syscalls:

Service	Code	Arguments	Result
`print_int`	1	`$a0`	*none*
`print_float`	2	`$f12`	*none*
`print_double`	3	`$f12`	*none*
`print_string`	4	`$a0`	*none*
`read_int`	5	*none*	`$v0`
`read_float`	6	*none*	`$f0`
`read_double`	7	*none*	`$f0`
`read_string`	8	`$a0` (address), `$a1` (length)	*none*
`sbrk`	9	`$a0` (length)	`$v0`
`exit`	10	*none*	*none*

Registers detail:



Register Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and
v1	3	results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)
t3	11	Temporary (not preserved across call)
t4	12	Temporary (not preserved across call)
t5	13	Temporary (not preserved across call)
t6	14	Temporary (not preserved across call)
t7	15	Temporary (not preserved across call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved across call)
t9	25	Temporary (not preserved across call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

Data and Text Segment (.data and .text)

The string "Hello World" should not be part of the executable part of the program (which contains all of the instructions to execute), which is called the text segment of the program. Instead, the string should be part of the data used by the program, which is, by convention, stored in the data segment. The MIPS assembler allows the programmer to specify which segment to store each item in a program by the use of several assembler directives.

To put something in the data segment, all we need to do is to put a .data before we define it. Everything between a .data directive and the next .text directive (or the end of the file) is put into the data segment. Note that by default, the assembler starts in the text segment, which is why our earlier programs worked properly even though we didn’t explicitly mention which segment to use. In general, however, it is a good idea to include segment directives in your code, and we will do so from this point on.

We also need to know how to allocate space for and define a null-terminated string. In the MIPS assembler, this can be done with the .asciiz (ASCII, zero terminated string) directive. For a string that is not null-terminated, the .ascii directive can be used

```
# hello.asm-- A "Hello World" program.
# Registers used:
#       $v0             - syscall parameter and return value.
#       $a0             - syscall parameter-- the string to print.
        .text
main:
        la $a0, hello_msg               # load the addr of hello_msg into $a0.
        li $v0, 4                       # 4 is the print_string syscall.
        syscall                         # do the syscall.

        li $v0, 10                      # 10 is the exit syscall.
        syscall                         # do the syscall.

# Data for the program:
        .data
hello_msg:      .asciiz "Hello World\n"
# end hello.asm
```

Note that data in the data segment is assembled into adjacent locations. Therefore, there are many ways that we could have declared the string "Hello World\n" and gotten the same exact output. For example we could have written our string as:

```
                .data
hello_msg:      .ascii "Hello"          # The word "Hello"
                .ascii " "              # the space.


                .ascii "World"          # The word "World"
                .ascii "\n"             # A newline.
                .byte 0                 # a 0 byte.
```

If we were in a particularly cryptic mood, we could have also written it as:

```
                .data
hello_msg:      .byte 0x48              # hex for ASCII "H"
                .byte 0x65              # hex for ASCII "e"
                .byte 0x6C              # hex for ASCII "l"
                .byte 0x6C              # hex for ASCII "l"
                .byte 0x6F              # hex for ASCII "o"
                ...                     # and so on...
                .byte 0xA               # hex for ASCII newline
                .byte 0x0               # hex for ASCII NUL
```

You can use the .data and .text directives to organize the code and data in your programs in whatever is most stylistically appropriate. The example programs generally have the all of the .data items defined at the end of the program, but this is not necessary. For example, the following code will assemble to exactly the same program as our original hello.asm:

```
        .text                       # put things into the text segment...
main:
        .data                       # put things into the data segment...
hello_msg:              .asciiz "Hello World\n"
        .text # put things into the text segment...
        la      $a0, hello_msg      # load the addr of hello_msg into $a0.
        li      $v0, 4              # 4 is the print_string syscall.
        syscall                     # do the syscall.

        li      $v0, 10             # 10 is the exit syscall.
        syscall                     # do the syscall.
```

Conditional Execution: the larger Program

The next program that we will write will explore the problems of implementing conditional execution in MIPS assembler language. The actual program that we will write will read two numbers from the user, and print out the larger of the two.

One possible algorithm for this program is exactly the same as the one used by add2.asm, except that we're computing the maximum rather than the sum of two numbers. Therefore, we'll start by copying add2.asm, but replacing the add instruction with a placeholder comment:

```
        # larger.asm-- prints the larger of two numbers specified
        # at runtime by the user.
        # Registers used:
        #           $t0             - used to hold the first number.
        #           $t1             - used to hold the second number.
        #           $t2             - used to store the larger of $t1 and $t2.
                .text
main:
                ## Get first number from user, put into $t0.
                li      $v0, 5          # load syscall read_int into $v0.
                syscall                 # make the syscall.

                move    $t0, $v0        # move the number read into $t0.

                ## Get second number from user, put into $t1.
                li      $v0, 5          # load syscall read_int into $v0.
                syscall                 # make the syscall.

                move    $t1, $v0        # move the number read into $t1.
                ## put the larger of $t0 and $t1 into $t2.
                ## (placeholder comment)
                ## Print out $t2.
                move   $a0, $t2          # move the number to print into $a0.
                li        $v0, 1            # load syscall print_int into $v0.
                syscall                     # make the syscall.

                ## exit the program.
                li        $v0, 10           # syscall code 10 is for exit.
                syscall                     # make the syscall.
# end of larger.asm.
```

MIPS branching instructions allow the programmer to specify that execution should branch (or jump) to a location other than the next instruction. These instructions allow conditional execution to be implemented in assembler language. One of the branching instructions is bgt. The bgt instruction takes three arguments. The first two are numbers, and the last is a label. If the first number is larger than the second, then execution should continue at the label, otherwise it continues at the next instruction. The b instruction, on the other hand, simply branches to the given label.

These two instructions will allow us to do what we want. For example, we could replace the placeholder comment with the following:

```
                        # If $t0 > $t1, branch to t0_bigger,
        bgt  $t0, $t1, t0_bigger
        move        $t2, $t1        # otherwise, copy $t1 into $t2.
        b endif                     # and then branch to endif
t0_bigger:
        move        $t2, $t0        # copy $t0 into $t2
endif:
```

If $t0 is larger, then execution will branch to the t0_bigger label, where $t0 will be copied to $t2. If it is not, then the next instructions, which copy $t1 into $t2 and then branch to the endif label, will be executed.
This gives us the following program:

```
# larger.asm-- prints the larger of two numbers specified
# at runtime by the user.
# Registers used:
#       $t0         - used to hold the first number.
#       $t1         - used to hold the second number.
#       $t2         - used to store the larger of $t1 and $t2.
#       $v0         - syscall parameter and return value.
#       $a0         - syscall parameter.

        .text
main:
        ## Get first number from user, put into $t0.
        li          $v0, 5          # load syscall read_int into $v0.
```

```
            syscall                     # make the syscall.

            move        $t0, $v0        # move the number read into $t0.
            ## Get second number from user, put into $t1.
            li          $v0, 5          # load syscall read_int into $v0.
            syscall                     # make the syscall.

            move        $t1, $v0        # move the number read into $t1.
            ## put the larger of $t0 and $t1 into $t2.
            bgt         $t0, $t1, t0_bigger     # If $t0 > $t1, branch to t0_bigger,
            move        $t2, $t1                # otherwise, copy $t1 into $t2.
            b           endif                   # and then branch to endif
t0_bigger:
            move        $t2, $t0                # copy $t0 into $t2
endif:

            ## Print out $t2.
            move        $a0, $t2        # move the number to print into $a0.
            li          $v0, 1          # load syscall print_int into $v0.
            syscall                     # make the syscall.

            ## exit the program.
            li          $v0, 10         # syscall code 10 is for exit.
            syscall                     # make the syscall.
# end of larger.asm.
```

Looping: the *multiples* Program

The next program that we will write will read two numbers A and B, and print out multiples of A from A to A×B. Since we already know how to read in numbers and print them out, we won't bother to implement these steps here. We'll just leave these as comments for now.

Algorithm: The multiples program.

```
1. Get A from the user.
2. Get B from the user. If B = 0, terminate.
3. Set sentinel value S = A×B.
4. Set multiple m = A.
5. Loop:
        (a) Print m.
        (b) If m = = S, then go to the next step.
        (c) Otherwise, set m = m+A, and then repeat the loop.
6. Terminate
```

```
# multiples.asm-- takes two numbers A and B, and prints out
# all the multiples of A from A to A * B.
# If B <= 0, then no multiples are printed.
# Registers used:
```

```
#       $t0         - used to hold A.
#       $t1         - used to hold B.
#       $t2         - used to store S, the sentinel value A * B.
#       $t3         - used to store m, the current multiple of A.

        .text
main:
        ## read A into $t0, B into $t1 (omitted).


      blez    $t1, exit        # if B <= 0, exit.
      mul     $t2, $t0, $t1    # S = A * B.
      move    $t3, $t0         # m = A
loop:

      ## print out $t3 (omitted)
      beq     $t2, $t3, endloop      # if m == S, we're done.
      add     $t3, $t3, $t0          # otherwise, m = m + A.

      ## print a space (omitted)
      b       loop
endloop:

      ## exit (omitted)

# end of multiples.asm
```

SPIM Assembler Directives

Name	Parameters	Description
`.data`	*addr*	The following items are to be assembled into the data segment. By default, begin at the next available address in the data segment. If the optional argument *addr* is present, then begin at *addr*.
`.text`	*addr*	The following items are to be assembled into the text segment. By default, begin at the next available address in the text segment. If the optional argument *addr* is present, then begin at *addr*. In SPIM, the only items that can be assembled into the text segment are instructions and words (via the `.word` directive).
`.kdata`	*addr*	The kernel data segment. Like the data segment, but used by the Operating System.
`.ktext`	*addr*	The kernel text segment. Like the text segment, but used by the Operating System.
`.extern`	*sym size*	Declare as global the label *sym*, and declare that it is *size* bytes in length (this information can be used by the assembler).
`.globl`	*sym*	Declare as global the label *sym*.

SPIM data directives:

Name	Parameters	Description
`.align`	n	Align the next item on the next 2^n-byte boundary. `.align 0` turns off automatic alignment.
`.ascii`	*str*	Assemble the given string in memory. Do not null-terminate.
`.asciiz`	*str*	Assemble the given string in memory. Do null-terminate.
`.byte`	*byte1* $\cdots$ *byteN*	Assemble the given bytes (8-bit integers).
`.half`	*half1* $\cdots$ *halfN*	Assemble the given halfwords (16-bit integers).
`.space`	*size*	Allocate n bytes of space in the current segment. In SPIM, this is only permitted in the data segment.
`.word`	*word1* $\cdots$ *wordN*	Assemble the given words (32-bit integers).

Lab A1.asm

1. Create a directory to hold the files for this lab.

2. Launch your favorite editor and type the following program. Note that assembly language is free-form but it is a good idea to align the four fields (label, instruction, operand, comment) in order to enhance the program's readability:

   ```
           .text
   main:   #--------------------------------------
           addi   $t0, $0, 60       # t0 = 60
           addi   $t1, $0, 7        # t1 = 7
           add    $t2, $t0, $t1     # t2 = t0+t1
           #--------------------------------------
           addi   $v0, $0, 1        # service #1
           add    $a0, $0, $t2      # printInt
           syscall                  # do print
           #--------------------------------------
           jr     $ra               # return
   ```

   Save the program under the name: LabA1.asm.

3. Launch SPIM. On a window station, run QtSpim and open the program using the File menu. This will not only read the program but also assemble it (translate it to machine language). Hence, if any statement were mistyped, the error would be reported at this stage.

4. Run the program. Select Go from the Simulator menu (as a shortcut, you can click the Go icon in the toolbar or simply press F5). In both versions, a dialog window will pop to prompt for the starting set-up. Accept the defaults by clicking OK.

5. There are two variants of the add instruction: one adds two registers and stores the sum in a third. The other adds a register and an immediate (a constant), and hence the 'i' suffix. Note also that add is sometimes used as an assignment statement (by adding to $0).

6. One of the panes in the SPIM GUI shows the values of all registers. Verify that the value of $t0 is 60 (decimal) or 3c (hexadecimal). Verify also the values of $t1 and $t2.

7. The value of register $v0 seems odd: the program stored 1 in it yet the register pane reports 10 for it! It must be that SPIM ran additional instructions after executing our program. To confirm, let us single-step through the program: re-open the program in order to load a fresh copy and re-initialize SPIM. Instead of running, single-step by pressing F10 (QtSpim).

8. Each time you step, the statement highlighted in the text-segment pane is executed and the registers are updated. How many statements are executed before our program?

9. Step through our program and watch the register pane. What is the value of $v0 immediately after the execution of our program has ended?

LabA2.asm

1. Our program printed the output using service #1, which prints integers. Modify the program so it prints using service #11, which interprets and prints the contents of $a0 as a character. Save the modified program as: LabA2.asm. Run it and explain why the output became C.

LabA3.asm

1. Revert back to LabA2.asm and save it as LabA3.asm. Rather than hard-coding the numbers in our program, let us read one of them from the user by using service #5, readInt. Replace the statement:

   ```
   addi    $t0, $0, 60   # t0 = 60
   ```

   with:

   ```
   addi    $v0, $0, 5    # v0 = readInt
   syscall
   add     $t0, $0, $v0
   ```

   Notice that you must set $v0 to 5 prior to issuing syscall. Afterwards, the entered integer is returned to you in $v0. We copied the return to $t0 so that the rest of the program can remain unchanged. Run the program and enter 10. Do you obtain the expected output?

LabA3.asm

1. Save LabA2.asm as LabA3.asm then modify it so that it processes the two numbers as follows:

   ```
   if ($t0 = = $t1)
   {
     Print ($t0 + $t1);
   }
   else
   {
     Print ($t0 - $t1);
   ```

```
}
```

We can express this using a conditional transfer of control (a branch) and an unconditional one (a jump):

```
    if ($t0 = = $t1) branch to XX;
    print ($t0 - $t1);
    jump to YY;
XX: print ($t0 + $t1);
YY: rest of program
```

The first instruction in the above pseudo-code can be realized using:

```
     beq $t0, $t1, XX
```

The third instruction is realized using:

```
     j YY
```

Run the program and verify that it works as expected. Note that beq (branch-on-equal) is used for = = and bne (branch-on-not-equal) for !=.

LabA7.asm

1. Save LabA6.asm as LabA7.asm then modify it so that it processes the two numbers as follows:

   ```
   if ($t0 < $t1)
   {
     Print ($t0 + $t1);
   }
   else
   {
     Print ($t0 - $t1);
   }
   ```

   We can reduce a "less-than" test to a "not-equal" test by using the following instruction:

   ```
   slt $x, $y, $z
   ```

   It sets $x to 1 if $y < $z and sets it to zero otherwise. Using this "set-on-less-than" instruction, you can perform the above test using bne. Run the program and verify that it works as expected. Note that slt has an immediate variant (slti) that allows the third operand to be an immediate.

LabA8.asm

1. Start fresh and create the program LabA8.asm with the following body (between main and fini):

   ```
           addi   $v0, $0, 1
           add    $a0, $0, $0
   loop:   slti   $t9, $a0, 5
           beq    $t9, $0, fini
           syscall
           addi   $a0, $a0, 1
           j      loop
   ```

   It is important that you understand the program and attempt to predict its output before running. Save the program then run it and confirm or correct your prediction.

2. Replace the statement before last in the above program with:

   ```
   addi   $a0, $0, 1
   ```

   What will the output be in this case? Show that single stepping is ideally suited to debug such an error.

LabA9.asm

1. Start fresh and create the program LabA9.asm that operates as follows:

   ```
   int $s0 = 0;
   int $t0 = readInt ();
   for (int $t5 = 0; $t5 < $t0; $t5++)
   {
     $s0 = $s0 + $t5;
   }
   Print ($s0);
   ```

   Note that there is no looping construct; you use branches and jumps. You can, for example, re-think the above loop as follows:

   ```
           $t5 = 0;
   loop:   if (! $t5 < $t0) branch to done;
           $s0 = $s0 + $t5;
           $t5++;
           jump to loop;
   done:   print($s0);
   ```

Save the program then run it with 10 as input. Do you get 45 as output?

+++

TASKS

1. Create a directory to hold the files for this lab. Launch `spim` and make sure its register pane displays values in hex.
2. Launch your favorite editor and type the following fragment:

```
          .text
main: #--------------------
addi $t0, $0, 60
addi $t1, $0, 7
div $t0, $t1
mflo $a0
```

   The `div` instruction is equivalent to:

```
lo = $t0 / $t1;
hi = $t0 % $t1;
```

In other words, it sets `lo` register to the *quotient* and `hi` register to the *remainder* of dividing its two integer operands. Since these two registers are not programmable (i.e. cannot be referenced in a program), the two instructions: `mflo` (move-from-lo) and `mfhi` (move-from-hi) were added to the instruction set to facilitate copying the contents of `lo` and `hi` to programmable registers. Complete the above fragment by having it output `lo` and `hi`. Save the program under the name: `Lab4_1.s`. Run the program and verify that it works as expected.

3. Modify the program so that it also prints the product of the two integers. The instruction:

```
     mult $t0, $t1
```

   multiplies the two operands and store the 64-bit product in `hi-lo`, i.e. its most significant 32 bits in `hi` and its least-significant 32 bits in `lo`. Again, transfer these two pieces to general-purpose registers and print them.

4. Start fresh and create the program `Lab4_2.s` as follows:

```
     .text
main: #--------------------
addi $t0, $0, 60
srl $a0, $t0, 1 # a0 = t0 shifted right once
# print($a0);
# print(' ');
sll $a0, $t0, 1 # a0 = t0 shifted left once
# print($a0);
#--------------------
fini: jr $ra
```

Replace the comments by appropriate output statements. Run the program and examine its two outputs. Explain the output based on your understanding of how integers are represented in binary.

5. Set a breakpoint immediately after the `srl` instruction. When the program pauses, examine the values of `$t0` and `$a0` as shown in the register pane. Explain these values based on your understanding of how integers are represented in hex.
6. Change the shift amount from 1 to 2. Re-run the program and interpret its output.
7. Is the left shift sign sensitive? Does it correctly perform multiplication when the operand is negative?
8. Is the right shift sign sensitive? Does it correctly perform division when the operand is negative?

9. Write the program `Lab4_3.s` that reads an integer `x` from the user and outputs `18x` *without* using `mult`. Note that you can rewrite `18x` as a sum of two terms each of which multiplies `x` by a power of 2.
10. Replacing multiplications by shifts is often done in high-level languages by optimizing compilers. Why is shifting a register faster than multiplying it?

11. The program `Lab4_4.s` seeks to determine and output the value of bit #10 in `$t0`. Note that bits are numbered right-to-left starting with zero. It does so by isolating this bit using the following pseudo-code:
    ```
    a0 = t0 shifted left 21 times
    a0 = a0 shifted right 31 times
    ```
    Write this program so that it reads an integer into `$t0`; performs the above algorithm; and then outputs `$a0`. For example, if the input is 5000, the output should be 0, but if the input were 6000, the output would be 1.
12. Add the following fragment to `Lab4_4.s`:
    ```
    andi $a0, $t0, 1024
    srl $a0, $a0, 10
    syscall
    ```
    The number, 1024, is known as the **mask** and it has the following representation in binary:
    ```
    0000 0000 0000 0000 0000 0100 0000 0000
    ```

    Verify that this mask-based approach will also output the state of bit #10. Could we have written the mask hex?

13. The word "mask" is often used to describe something that blocks the view except for a hole through which one can see. Does the number 1024 behave like a mask in this sense?

14. We seek to write the program `Lab4_5.s` that clears bit #10 in `$t0`; i.e. sets it to zero. The idea is to read an integer into `$t0` and then AND it with the following mask:
    ```
    1111 1111 1111 1111 1111 1011 1111 1111
    ```

Unfortunately this does not work because the immediate in the instruction:

```
andi $t5, $t0, 0xfffffbff
```

has to have at most 16 bits. Hence, we construct this mask in stages as follows:

```
$t5 = 0xffff;
$t5 = $t5 shifted left 16 times;
$t5 = $t5 or 0xfbff;
```

Implement the program and run it. Use `ori` instead of `addi` to store a bit pattern in a register Verify that it works as expected. For example, if the input were 5000 then the output should be 5000 whereas if the input were 6000 then the output should be 4976.

15. The mask of the previous task could also be generated using the following algorithm:

    ```
    $t6 = 1024;
    $t6 = not $t6;
    ```

    There is no `not` instruction in MIPS but there is a `nor` (i.e. `not or`). Add code to `Lab4_5.s` so that it generates the mask using the new method. Are the two methods equivalent? You can simply inspect the register pane of `spim` and compare the contents of `$t5` and `$t6`.
16. The mask of the previous task could also be generated using the `lui` instruction. Split the immediate `0xfffffbff` into two pieces; send them separately in two different instructions; and then reconstitute the 32-bit immediate.

17. It is often said that exclusive-or is useful for detecting differences in bit values, i.e. detecting the combinations `01` and `10`. Justify this claim by examining the truth table of `xor`.
18. Write the program `Lab4_6.s` that flips bit #10 in `$t0`. Prepare an appropriate mask and note that when you `xor` a bit with 1, you get the negation of that bit, but when you `xor` with 0, you get the original bit unchanged.

NOTES

- The **shift amount** appears as an immediate in shift instructions but, in fact, it is represented in **five bits** as an unsigned integer. Hence, its range is [`0, 31`].
- The 16-bit immediate in non-shift logical instructions is treated as a 16-bit unsigned integer and is **zero extended** (*not* sign extended) to 32 bits. Hence, if you `andi` using `35000` as an immediate, you are `and`ing with the 32-bit sequence:

  ```
  0000 0000 0000 0000 1000 1000 1011 1000
  ```

  The value of the immediate in non-shift logical instructions is thus constrained to the 16-bit unsigned range, i.e. [$0,\ 2^{16} - 1$].