UMM AL-QURA UNIVERSITY

College of Computer and Information Systems

Computer Engineering Department

1403372

Computer Organization

Lab Manual

Student Name: _____

Student ID: _____

Section: _____ Group: _____

Session (Fall / Spring / Summer) _____

This page is intentionally left blank

TABLE OF CONTENTS

Prepared By:	Engr. Muhammad Asif Manzoor	2012
Revised By:	_____	_____
Reviewed By:	_____	_____
Approved By:	_____	_____

Umm Al-Qura University

Computer Engineering Department

LABORATORY SAFETY GUIDELINES

A. General Laboratory Safety Rules

1. Personal Safety

- Be familiar with the electrical and fire hazards associated with your workplace.
- Be as careful for the safety of others as for yourself. Think before you act.
- Be tidy and systematic.
- Avoid bulky, loose or trailing clothes. Avoid long loose hair.
- No one is allowed to enter in the lab area bare foot due to increased risk of electric shock.
- Remove metal bracelets, rings or watchstraps when working in the laboratories.
- Avoid working with wet hands and clothing.

2. Food, Beverages and Smoking

- Due to the increased risk of electric shock, no drinking of beverages, consumption or storage of any kind of food is allowed in the laboratory.
- Smoking is prohibited in all laboratories in all timings.

3. Soldering

- No one is allowed to do soldering in any of the computer engineering laboratories except the graduation project design laboratory.
- Anyone doing soldering in the graduation project design laboratory must wear appropriate apparel, socks, gloves, covered shoes and safety goggles to prevent the possibility of severe burns resulting from the splashing or dripping of hot liquefied solder into the face and eyes or on to the exposed skin on the chest, hands, legs, and feet.
- Students who are not so properly attired for these tasks will NOT be allowed to perform any type of soldering in the graduation project design laboratory.

4. Laboratory Operating Hours

- Students are never allowed to work alone in any lab area other than scheduled laboratory operating hours unless either a Lab T/A or Course Instructor is present inside that lab area.
- The laboratory operating hours for students are posted on the entrance doorway and on the notice board of computer engineering department.

5. Power Supply Related Safety

- Voltages above **50-VAC** or **120-VDC** are always dangerous.
- Extra precautions should be considered as voltage levels are increased.
- Never make any changes to circuits or mechanical layout without first isolating the circuit by switching off and removing connections to power supplies.

6. Laboratory Equipment

- Lab equipment may not be removed from the Computer Engineering lab areas without the permission of the Laboratory Supervisors.
- Laboratory bench equipment (except for some lab bench computers) must be turned off before closing down the lab area for the day.
- Never open (remove cover) of any equipment in the laboratories.
- Never "jump," disable, bypass or otherwise disengage any safety device or feature of any equipment in the laboratories.
- Laboratories shall be locked when unoccupied.

7. Waste Management Safety

- Know the correct handling, storage and disposal procedures for batteries, cell, capacitors, inductors and other high energy-storage devices.

8. Equipment Safety

- Before equipment is energized ensure, circuit connections and layout have been checked by a Teaching Assistant (TA) and all colleagues in your group has given their consent.
- Experiments left unattended should be isolated from the power supplies. If for a special reason, it must be left on, a barrier and a warning notice are required.
- Equipment found to be faulty in any way should be reported to the lab supervisor and taken out of service until inspected and declared safe.

9. Equipment Accessories

- Use extension cords only when necessary and only on a temporary basis.
- Request new outlets if your work requires equipment in an area without an outlet.
- Discard damaged cords, cords that become hot, or cords with exposed wiring.

B. Electrical and Fire Emergency Responses

1. Police, Fire or Medical Emergency

- Use the telephone located in the laboratory area and press **0-996** to notify police, fire, and ambulance for emergency help.
- Everyone present in the laboratory area shall be familiar with the locations and operation of safety and emergency equipment, including but not limited to, fire extinguishers, first aid kits, emergency power off system, fire alarm pull stations, and emergency telephones.

2. Electric Shock

- When someone suffers serious electrical shock, he may be knocked unconscious.
- If the victim is still in contact with electrical current, immediately turn off the electrical power source.
- If you cannot disconnect the power source, depress the Emergency Power Off switch.
- Do not touch a victim that is still in contact with a live power source; you could be electrocuted! Have someone call for emergency medical assistance immediately. Administer first-aid, as appropriate.

3. Electrical Fire

- If an electrical fire occurs, try to disconnect the electrical power source, if possible.
- If the fire is small and you are not in immediate danger; and if you have been properly trained in fighting fires, use the correct type of fire extinguisher to extinguish the fire.
- When in doubt, push in the Emergency Power Off button.
- **NEVER use water to extinguish an electrical fire**.

4. Emergency Power Off

- Every lab is equipped with an Emergency Power off System.
- When this switch is depressed, electrical power to the lab will shut off, except for lights.
- Only authorized personnel are permitted to reset power once the Emergency Power Off system has been engaged.

5. Building Evacuation in Emergency

- Everyone present in the laboratory should be familiar to emergency exits & way out plans.
- Use the nearest exit doorway from lab area closest to the stairwell to exit the building.
- Follow the Emergency Exit Signs posted in the hallways. Do not use elevators.
- Lab Teaching Assistants (T/As) or Instructor shall make sure all persons are out of the laboratory area and follow the directions posted at each doorway to the laboratory area.

The above general laboratory safety rules are designed to safeguard you and your co-workers, fellow students and colleagues and are a minimum requirement for individuals working in the computer engineering laboratories at Umm Al-Qura University, Makkah Al-Mukarramah. Specialized training and rules may apply depending on type and scope of activities involved.

This page is intentionally left blank

Lab 1: Introduction to MARS

MARS is a software simulator for the MIPS assembly language intended for educational use. We will explore the capabilities of MARS release 4.3.

MARS may be downloaded from www.cs.missouristate.edu/MARS.

Basic MARS Use

The example program is **`Fibonacci.asm`** to compute everyone's favorite number sequence.

1. Start MARS from the Start menu or desktop icon.

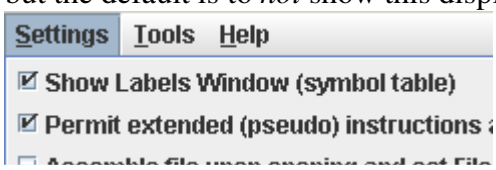
2. Use the menubar File…Open or the Open icon  to open Fibonacci.asm in the default folder. *(All icons have menubar equivalents; the remainder of these steps will use the icon whenever possible.)*

3. The provided assembly program is complete. Assemble the program using the icon 

4. Identify the location and values of the program's initialized data. Use the checkbox to toggle the display format between decimal and hexadecimal  .
   - The nineteen-element array **`fibs`** is initialized to zero, at addresses 0x10010000 … 0x10010048.
   - The data location **`size`** has value 19_{ten} at 0x1001004c.
   - The addresses 0x10010050 … 0x1001006c contain null-terminated ASCII strings.

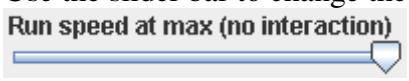
Use the checkbox to toggle the display format between decimal and hexadecimal,  .

5. Use the Settings menu to configure the MARS displays. The settings will be retained for the next MARS session.
   - The Labels display contains the addresses of the assembly code statements with a label, but the default is to *not* show this display. Select the checkbox from the Settings menu.



   - Select your preference for allowing pseudo-instructions (programmer-friendly instruction substitutions and shorthand).
   - Select your preference for assembling *only one* file, or *many* files together (all the files in the current folder). This feature is useful for subroutines contained in separate files, etc.
   - Select the startup display format of addresses and values (decimal or hexadecimal).

6. Locate the Registers display, which shows the 32 common MIPS registers. Other tabs in the Registers display show the floating-point registers (Coproc 1) and status codes (Coproc 0).

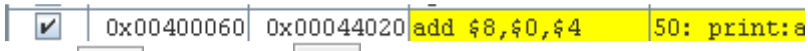
7. Use the slider bar to change the run speed to about 10 instructions per second.


This allows us to "watch the action" instead of the assembly program finishing directly.

8. Choose how you will execute the program:

   - The  icon runs the program to completion. Using this icon, you should observe the yellow highlight showing the program's progress and the values of the Fibonacci sequence appearing in the Data Segment display.
   - The  icon resets the program and simulator to initial values. Memory contents are those specified within the program, and register contents are generally zero.
   - The  icon is "single-step." Its complement is  , "single-step backwards" (undoes each operation).

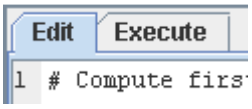
9. Observe the output of the program in the Run I/O display window:
   The Fibonacci numbers are:
   1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
   -- program is finished running --

10. Modify the contents of memory. (Modifying a register value is exactly the same.)
    - Set a breakpoint at the first instruction of the subroutine which prints results. Use the checkbox at the left of the instruction whose address is 0x00400060 = 4194400_{ten}.


    - Reset  and re-run  the program, which stops at the breakpoint.
    - Double-click in one of the memory locations containing the computed Fibonacci numbers. The cell will be highlighted and will accept keyboard entry, similar to a spreadsheet. Enter some noticeably different value, and use the Enter key or click outside the cell to indicate that the change is complete. *Example: Memory address 0x10010020 = 268501024 $_{ten}$ presently contains data 0x00000022 = 34 $_{ten}$.*
    - Click  to continue from the breakpoint. The program output includes your entered value instead of the computed Fibonacci number.

11. Open the Help  for information on MIPS instructions, pseudo instructions, directives, and syscalls.

12. Modify the program so that it prompts the user for the Fibonacci sequence length.

- Select the Edit tab in the upper right to return to the program editor. 

- The MIPS comment symbol is #. All characters on the line after the character # are ignored.
- Un-comment lines 12-19. The newly exposed program fragment will prompt the user for the length of the Fibonacci sequence to generate, in the range $2 \le x \le 19$. (The length of the sequence must be limited to the size of the declared space for result storage.)
- Determine the correct **`syscall`** parameter to perform "read integer" from the user, and insert the parameter at line The correct **`syscall`** parameter may be found at Help  … Syscall tab…read integer service. The completed line will have the form **`li $v0, 42`** (where in this case 42 is not the right answer).
- Reset  and re-run  the program. The program will stop at the breakpoint you inserted previously. Continue and finish with  .

Lab 2: Data Types, Registers and Program Structure

Data Types and Literals

Data types:

- Instructions are all 32 bits
- byte(8 bits), halfword (2 bytes), word (4 bytes)
- a character requires 1 byte of storage
- an integer requires 1 word (4 bytes) of storage

Literals:

- numbers entered as is. e.g. 4
- characters enclosed in single quotes. e.g. 'b'
- strings enclosed in double quotes. e.g. "A string"

Registers

- 32 general-purpose registers
- register preceded by $ in assembly language instruction
  two formats for addressing:
  - using register number e.g. $0 through $31
  - using equivalent names e.g. $t1, $sp
- special registers Lo and Hi used to store result of multiplication and division
  - not directly addressable; contents accessed with special instruction mfhi ("move from Hi") and mflo ("move from Lo")
- stack grows from high memory to low memory

Register Number	Alternative Name	Description
0	zero	the value 0
1	$at	(**a**ssembler **t**emporary) reserved by the assembler
2-3	$v0 - $v1	(**v**alues) from expression evaluation and function results
4-7	$a0 - $a3	(**a**rguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(**s**aved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting.

		Preserved across procedure calls
24-25	$t8 - $t9	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	**g**lobal **p**ointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	**s**tack **p**ointer Points to last location on the stack.
30	$s8/$fp	**s**aved value / **f**rame **p**ointer Preserved across procedure calls
31	$ra	**r**eturn **a**ddress

Program Structure

- just plain text file with data declarations, program code (name of file should end in suffix .asm to be used with SPIM simulator)
- data declaration section followed by program code section

Data Declarations

- placed in section of program identified with assembler directive **.data**
- declares variable names used in program; storage allocated in main memory (RAM)

Code

- placed in section of text identified with assembler directive **.text**
- contains program code (instructions)
- starting point for code e.g.ecution given label **main:**
- ending point of main code should use exit system call (see below under System Calls)

Comments

- anything following # on a line
  # This stuff would be considered a comment
- Template for a MIPS assembly language program:
- `# Comment giving name of program and description of function`
- `# Template.s`
- `# Bare-bones outline of MIPS assembly language program`
-
- `            .data       # variable declarations follow this line`
- `                        # ...`
-
- `            .text       # instructions follow this line`
-
- `main:                  # indicates start of code (first instruction to execute)`
- `                        # ...`
-
- `# End of program, leave a blank line afterwards to make SPIM happy`

Lab 3: Basic Instructions

Arithmetic Instructions

- most use 3 operands
- all operands are registers; no RAM or indirect addressing
- operand size is word (4 bytes)
- In all instructions below, `Src2` can either be a register or an immediate value (a 16 bit integer). The immediate forms of the instructions are only included for reference. The assembler will translate the more general form of an instruction (e.g., `add`) into the immediate form (e.g., `addi`) if the second argument is constant.

`add Rdest, Rsrc1, Src2` *Addition (with overflow)*
`addi Rdest, Rsrc1, Imm` *Addition Immediate (with overflow)*
`addu Rdest, Rsrc1, Src2` *Addition (without overflow)*
`addiu Rdest, Rsrc1, Imm` *Addition Immediate (without overflow)*

Put the sum of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

Similar to *add,* we have instruction for subtraction *'sub'*. It works in the same manner as addition instruction. Examples of addition and subtraction are:

```
        add     $t0,$t1,$t2     #  $t0 = $t1 + $t2;   add as signed (2's
complement) integers
        sub     $t2,$t3,$t4     #  $t2 = $t3 Ð $t4
        addi    $t2,$t3, 5      #  $t2 = $t3 + 5;   "add immediate" (no sub
immediate)
        addu    $t1,$t6,$t7     #  $t1 = $t6 + $t7;   add as unsigned integers
        subu    $t1,$t6,$t7     #  $t1 = $t6 + $t7;   subtract as unsigned
integers
```

Constant Manipulating Instructions

`li Rdest, imm` *Load Immediate*

This instruction is used to store an immediate value *imm* into the register *Rdest*. Example of *li* instruction can be:

li $t0, 5 ($t0 = 5)

```
Program: Addition and subtraction of two numbers
li      $t0, 4
li      $t1, 5

add     $t2, $t0, $t1
sub     $t3, $t1, $t0
```

First two instructions are used to store the values in registers $t0 and $t1. Line 3 will add these 2 registers and result will be stored in register $t2 and line 4 will subtract $t1, $t0 and the result will be stored in register $t3.

Lab 4: System Calls

Introduction

A number of system services, mainly for input and output, are available for use by your MIPS program. They are described in the table below.

MIPS register contents are not affected by a system call, except for result registers as specified in the table below.

How to use SYSCALL system services

Step 1. Load the service number in register $v0.
Step 2. Load argument values, if any, in $a0, $a1, $a2, or $f12 as specified.
Step 3. Issue the SYSCALL instruction.
Step 4. Retrieve return values, if any, from result registers as specified.

Table: System services.

Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_float	6		float (in `$f0`)
read_double	7		double (in `$f0`)
read_string	8	`$a0` = buffer, `$a1` = length	
sbrk	9	`$a0` = amount	address (in `$v0`)
exit	10		
print_character	11	`$a0` = character	
read_character	12		character (in `$v0`)
open	13	`$a0` = filename,	file descriptor (in `$v0`)
		`$a1` = flags, `$a2` = mode	

read	14	`$a0` = file descriptor,	bytes read (in `$v0`)
		`$a1` = buffer, `$a2` = count	
write	15	`$a0` = file descriptor,	bytes written (in `$v0`)
		`$a1` = buffer, `$a2` = count	
close	16	`$a0` = file descriptor	0 (in `$v0`)
exit2	17	`$a0` = value	

Termination of Program:

System call code '10' is used to terminate the program. To terminate the executing program, the value '10' is required to be copied to register $v0 and then the instruction *syscall* will be executed to perform the termination task.

```
li      $v0, 10
syscall
```

Integer Input:

System call code '5' is used to get an integer input. After calling the *syscall*, user has to input the number. This input is stored in the register $v0. Which has to be moved to some other register before proceeding forward.

```
li      $v0, 5
syscall

move $t0, $v0
```

Integer Output:

System call code '1' is used to display an integer value on the screen. The value must be stored in the register $a0 hence before calling *syscall*, we have to move the required value to the register $a0. In the below example, we want to display the value of register $t0, so we have moved the contents of register $t0 to the register $a0

```
move $a0, $t0
li      $v0, 5
```

syscall

move is data movement instruction. The number of operands required for the *move* instruction in two. One of them is source while the other is destination. The general syntax of *move* instruction is:

```
move Rdest, Rsrc
```

Where *Rdest* is the destination register while *Rsrc* is the source register and we are moving value from *Rsrc* to *Rdest*.

Lab 5: Branch Statements-I

In the MIPS assembly language, there are only two types of conditional branch instructions. This means you don't have to remember any great variety of special case branching mechanisms. One branches if two registers are equal, the other if they are not equal.

beq $value1, $value2, offset # if ($value1 == $value2) goto offset;

bne $value1, $value2, offset # if ($value1 != $value2) goto offset;

The third operand in the instruction is the offset. In MIPS, this is a 16 bit signed integer that represents where to continue if the comparison returns true. The offset represents how many instructions, counting from the instruction after the branch instruction, to pass over in order to get to the correct instruction. For example look at the code below:

0x4000:0000 add $t1, $t2, $t3

0x4000:0004 beq $t1, $t3, -2

0x4000:0008 sub $t1, $t1, $t3

In this code, the branch instruction would move up two instructions from the instruction after itself. This means it would branch from position 0x4000:0008 to position 0x4000:0000 and then continue evaluating the instructions in sequence. It works going in the opposite direction as well.

0x4000:000C bne $t1, $t3, 1

0x4000:0010 addi $t1, $t3, 20

0x4000:0014 addi $t3, $t3, -5

In the case above, the branch would go from position 0x4000:0010 to position 0x4000:0014 before continuing evaluation of the instruction in sequence. Notice that the total number of bytes skipped is found by multiplying offset by 4.

The C if and MIPS:

Comparing the C++ if expression with MIPS branch statements may help in writing code. Especially when you know how to "express" yourself in C++, but perhaps not as well in assembly language. First let's examine a simple if expression and break it up into different parts:

if (pred) consequent

If consists of both a predicate and a consequent. The predicate is itself a single expression that results in either a "true" or "false" value (non-zero or zero value). The consequent is a single statement expression, or multiple statement expressions surrounded by braces, '{' and '}'. We can make a similar construct using MIPS assembly code:

```
predicate: beq $s1, $s2, endif      # if ($s1 == $s2)
consequent: addi $s1, $s1, 1        # $s1++;
endif:                              # End of if-else portion
```

In the above code, if the condition is true, the processor will jumps to the *endif*. If the condition is false, the processor won't jumps, rather it will execute the very next line after the branch statement (in this case: the consequent portion).

Notice that we can divide our MIPS code into three regions, the predicate, the branch statement, and the consequent. The first of these regions is the predicate. Any number of statements that produce a zero or non-zero value in a register. The second region is the branch statement. If beq is used with $zero, a non-zero value would be true, and if bne is used with $zero, a zero value would be true. The third region is the consequent. This does whatever should be done if a true value results.

Something similar can be done to if statements with else statements in them.

```
Predicate: bne $s1, $s2, endif      # if ($s1 != $s2)
consequent: addi $s1, $s1, 1        # $s1++;
j endif                             # else
```

```
alternate: addi $s2, $s2, 1        # $s2++;
endif:                             # End of if-else portion
```

In the above piece of code, two registers $s1 and $s2 is compared. If $s1 is not equal to $s2 then register $s1 is incremented by '1'. Otherwise the value of register $s2 is incremented. After execution of *true* part, the code jumps to the *endif* statement; so that it doesn't execute the *false* part of this branch statement.

Notice the use of the jump instruction for MIPS. This instruction jumps without condition to the location given. The location is specified by a 28bit number. There is also an instruction that jumps without condition to the location given inside a register.

```
J location          # goto location;
jr $location        # goto $location;
```

Program: take two inputs from the user. Add these two if the first number is greater and subtract, if the second is greater.

Main:

```
        li      $v0, 5
        syscall
        move        $t0, $v0

        li      $v0, 5
        syscall
        move        $t1, $v0

        bne   $t0, $t1, else
        sub   $t2, $t1, $t0
        j       out
else:   add   $t2, $t1, $t0

out:    li      $v0, 10
        syscall
```

Lab 6: Branch Statements-II

We'll consider several jump instructions, which is used to implement certain branches that don't exist as instructions MIPS.

There are the list of instructions we'll look at.

- **beq** Branches if the quantities of two registers are equal.
- **bne** Branches if the quantities of two registers are NOT equal.
- **bgtz** Branches if a quantity in a register is greater than zero (quantity is 32 bit, 2C).
- **bgez** Branches if a quantity in a register is greater than or equal to zero (quantity is 32 bit, 2C).
- **bltz** Branches if a quantity in a register is less than zero (quantity is 32 bit, 2C).
- **blez** Branches if a quantity in a register is less than or equal to zero (quantity is 32 bit, 2C).
- **bgt** Branches if a quantity in a register is greater than a quantity in another register (quantity is 32 bit, 2C).
- **blt** Branches if a quantity in a register is less than a quantity in another register(quantity is 32 bit, 2C).
- **j** Jump to an address
- **jr** Jump to an address stored in a register
- **jal** Jump to an address, and store the return address in a register.
- **jalr** Jump to an address stored in a register, and store the return address in another register.

Here's how they would be written

```
beq   $rs, $rt, offset
bne   $rs, $rt, offset
bgez  $rs, offset
bgtz  $rs, offset
blez  $rs, offset
bltz  $rs, offset
bgt    $rs, $rt, offset
blt    $rs, $rt, offset
j     offset
jal   offset
```

```
jr     $rs
jalr   $rs
```

The offset is a 16-bit 2C value, except for **j** and **jal**, where they are 26 bits UB. **j** and **jal** are **J-type** instructions, while all the branch instructions are **I-type**. Notice that the instruction format (e.g., **R-type**, **I-type**) are formats. Just because an instruction is used for jumping (e.g., **beq**) does not mean it's a **J-type** instruction. It's the format that matters.

Semantics of Branch instructions

The branch instructions (i.e., **beq**, **bne**, **bgtz**, **bgez**, **bltz**, **blez**) all have 16 bit immediate values.

Program: Display the greater of two numbers

```
main:
        li      $v0, 5
        syscall

        move  $t0, $v0

        li      $v0, 5
        syscall

        move  $t1, $v0

        bgt     $t0, $t1, else
        move  $a0, $t1
        li      $v0, 1
        syscall
        j       out

else:   move  $a0, $t0
        li      $v0, 1
        syscall

out:    li      $v0, 10
        syscall
```

Lab 7: Loops

The loops are also implemented by using the branch statement (discussed in previous labs). All type of loops (for, while, and do-while) can be implement by using branch statements.

The While and MIPS:

The C++ while expression closely resembles the if expression. It has a predicate and something that happens continuously as long as the expression returns true.

```
predicate: bgt $s1, $s2, endwhile        # while ($s1 < $s2) {

consequent: addi $s1, $s1, 1             # $s1++;
j predicate                              # }
endwhile:                                # end of while loop
```

In the above code, if the condition is true ($s1 > $s2), the code will jumps to the *endwhile* and the loop will stop execution. On other hand, the loop will executes till the value of register $s1 is less than the value of register $s2. After incrementing the value of register $s1, the code takes an unconditional jump to the start of the loop.

The C++ do-while and MIPS

The C++ do expression resembles the while, except that it doesn't have a loop back statement, that's where the predicate and the branch statement both go to continue looping only if the predicate returns true.

```
doloop: addi $s1, $s1, 1                 # do { $s1++;
predicate: blt $s1, $s2, doloop          # }while ($s1 < $s2)
enddo:                                   # end of do-while loop
```

We have executed the body of loop without checking the condition. Condition will be checked after executing the body of loop and the execution of next iteration depends upon the result of this condition checking statement. Hence the body of the loop will be executed at least once whether the given condition is true or false.

The C++ for and MIPS

The for expression is like the while expression except it has two addition components to it. Not only does it have the consequent body which is evaluated continuously as long as the predicate returns a true value, it has initialization and next statements built into it. It would look as follows:

```
initialize: add $s1, $zero, $zero        # for ($s1 = 0,
addi $s2, $zero, 10                      # $s2 = 10;
predicate: bgt $s1, $s2, endfor          # $s1 < $2;

consequent: addi $s1, $s1, 1             # $s1 += 1;
# body of loop
j predicate                              # }
endfor:
```

Notice how it's form doesn't really change that much from the while loop. It's really just a construct in C to make code more compressed and readable.

Lab 8: Divide and Multiply Instruction

Divide Instruction:

The general syntax of instruction used for division of two numbers is

```
div  $t0,  $t1
```

In the above example, the register $t0 is divided by the register $t1. In this instruction, we don't mention the output register like we do in add and sub instruction. Two special registers are used for storing the results. These register are *lo* and *hi*. Hence *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. The use of *div* instruction is illustrated by the following example:

```
main:
addi $t0, $0, 60
addi $t1, $0, 7
div $t0, $t1
mflo $a0
```

Register $t0 is divided by register $t1 and the contents of register lo are moved to register $a0. These two register lo and hi can only be accessible through mflo and mfhi commands.

Multiply Instruction:

The general syntax of instruction used for multiply of two numbers is

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

In the above example, the register $t0 is multiplied by the register $t1. In this instruction, we don't mention the output register like we do in add and sub instruction. Two special registers are used for storing the results. These register are *lo* and *hi*. When we multiply

two 32-bit register, the size of resultant is 64-bit. Hence we require two 32-bit register to store the complete result of the multiplication process. So the lower 32-bits of the result are stored in the register lo while the higher 32-bits are stored in the register hi.

As discussed earlier, 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. The use of *mult* instruction is illustrated by the following example:

```
main:
addi $t0, $0, 60
addi $t1, $0, 7
mult $t0, $t1
mfhi $a0
```

Register $t0 is multiplied by register $t1 and the contents of register lo are moved to register $a0. These two register lo and hi can only be accessible through mflo and mfhi commands.

Data Movement Instructions:

The multiply and divide unit produces its result in two additional registers, hi and lo. These instructions move values to and from these registers. The multiply, divide, and remainder instructions described above are pseudo-instructions that make it appear as if this unit operates on the general registers and detect error conditions such as divide by zero or overflow.

`mfhi Rdest`	*Move From hi*
`mflo Rdest`	*Move From lo*

Program: Division of two numbers and displaying the result on screen

```
main: #----------------------
addi $t0, $0, 60
addi $t1, $0, 7
div $t0, $t1

mflo $a0
li $v0,1
syscall

mfhi $a1
li $v0,1
move $a0,$a1
syscall

#The div instruction is equivalent to:
#lo = $t0 / $t1;
#hi = $t0 % $t1;
```

Lab 9: Variables and Memory Access

Data Declarations

format for declarations:

name: storage_type value(s)

- create storage for variable of specified type with given name and specified value
- value(s) usually gives initial value(s); for storage type .space, gives number of spaces to be allocated

Note: labels always followed by colon (:)

example

```
var1:       .word   3       # create a single integer variable with initial value 3
array1:     .byte   'a','b' # create a 2-element character array with elements
initialized
                            #  to  a  and  b
array2:     .space  40      # allocate 40 consecutive bytes, with storage
uninitialized
                            #  could be used as a 40-element character array, or a
                            #  10-element integer array; a comment should
indicate which!
```

Load / Store Instructions

- RAM access only allowed with load and store instructions
- all other instructions use register operands

load:

```
lw      register_destination, RAM_source
```

#copy word (4 bytes) at source RAM location to destination register.

```
lb      register_destination, RAM_source
```

#copy byte at source RAM location to low-order byte of destination register,
and sign-e.g.tend to higher-order bytes

store word:

```
        sw      register_source, RAM_destination
```

#store word in source register into RAM destination

```
        sb      register_source, RAM_destination
```

#store byte (low-order) in source register into RAM destination

```
example:
        .data
var1:   .word   23              # declare storage for var1; initial value is 23

        .text
__start:
        lw      $t0, var1               # load contents of RAM location into
register $t0:  $t0 = var1
        li      $t1, 5          #  $t1 = 5   ("load immediate")
        sw      $t1, var1               # store contents of register $t1 into
RAM:  var1 = $t1
        done
```

Indirect and Based Addressing

- Used only with load and store instructions

load address:

```
        la      $t0, var1
```

- copy RAM address of var1 (presumably a label defined in the program) into register $t0

indirect addressing:

```
        lw      $t2, ($t0)
```

- load word at RAM address contained in $t0 into $t2

```
sw      $t2, ($t0)
```

- store word in register $t2 into RAM at address contained in $t0

based or indexed addressing:

```
lw      $t2, 4($t0)
```

- load word at RAM address ($t0+4) into register $t2
- "4" gives offset from address in register $t0

```
sw      $t2, -12($t0)
```

- store word in register $t2 into RAM at address ($t0 - 12)
- negative offsets are fine

Note: based addressing is especially useful for:

- arrays; access elements as offset from base address
- stacks; easy to access elements at offset from stack pointer or frame pointer

example

```
                .data
array1:         .space  12              #  declare 12 bytes of storage to hold
array of 3 integers
                .text
__start:        la      $t0, array1             #  load base address of array
into register $t0
                li      $t1, 5          #  $t1 = 5   ("load immediate")
                sw $t1, ($t0)           #  first array element set to 5;
indirect addressing
                li $t1, 13              #   $t1 = 13
                sw $t1, 4($t0)          #  second array element set to 13
                li $t1, -7              #   $t1 = -7
                sw $t1, 8($t0)          #  third array element set to -7
                done
```

Lab 10: Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type int in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, int for example, with a unique identifier.

Since arrays can store LOTS of data, and since we have only a small (~32) number of registers, it is infeasible to use the registers for long-term storage of the array data. Hence, arrays are stored in the Data Segment of a MIPS program. Fundamentally, there are three operations which one can perform on an array:

- Getting the data from an array cell, e.g, `x = list[i];`
- Storing data into an array cell, e.g. `list[i] = x;`
- Determining the length of an array, i.e. `list.length`.

For purposes of this step in the lab, you may assume that the length of the array is 10. (The worksheet asks about changing this, but you may hard code this value while writing the program.)

To access the data in the array requires that we know the address of the data and then use the load word (`lw`) or store word (`sw`) instructions. Words (which is how integers are stored) in MIPS take up 32 bits or 4 bytes. Therefore, if we have a declaration such as:

```
list: .word 3, 0, 1, 2, 6, -2, 4, 7, 3, 7
```

the address that is loaded by the instruction `la $t3, list` is the address of the first '3' in the list. The address of the '0' is 4 greater than that number, and the address of the '6' is 16 greater than that number.

The following snippet of code will place the value of `list[6]` into the `$t4`:

```
la $t3, list          # put address of list into $t3
li $t2, 6             # put the index into $t2
add $t2, $t2, $t2     # double the index
add $t2, $t2, $t2     # double the index again (now
```

```
4x)
    add $t1, $t2, $t3    # combine the two components of
the address
    lw $t4, 0($t1)       # get the value from the array
cell
```

If we wish to assign to the contents of `$t4` to `list[6]` instead, the last line would simply be:

```
     sw $t4, 0($t1)      # store the value into the array
cell
```

Program: Addition of two arrays

```
.data
A:          .word 3,4,5,6,7
B:          .word 1,2,3,4,5
C:          .word  0:5
.text
main:
        la      $s0, A
        la      $s1, B
        la      $s2, C

        li      $t5,0
        li      $t6,5

Loop:
        lw      $t1, ($s0)
        lw      $t2, ($s1)

        add     $t3, $t1, $t2

        sw      $t3, ($s2)

        add     $s0, $s0, 4
        add     $s1, $s1, 4
        add     $s2, $s2, 4

        add     $t5, $t5, 1
        beq     $t5, $t6, out

        j       Loop

out:
        li      $v0, 10
        syscall
```

Lab 11: Strings

Recall from C++/Java programming that each character in a text file is stored in one byte. In C++/Java, a variable of type "char" uses one byte. Such text is often represented is using ASCII. Previously, we saw the instructions lw and sw which take a load from or store a word to Memory. There are similar instructions for single bytes. lb loads a byte, and sb stores a byte. These functions are of I-format.

lb $s2, 3($s1)

sb $s2, -2($s1)

lb takes one byte from memory and puts it into a register. The upper 24 bits of the register are sign extended i.e. filled with whatever value is in the most significant bit (MSB) of that byte. There is also an unsigned version of load byte, namely lbu, which fills the upper 24 bits with 0's (regardless of the MSB).

sb copies the lower 8 bits of a register into some byte in memory. This instruction ignores the upper 24 bits of the word in the register.

Storage type (required for the variable declaration) for the string is either *.ascii* or *.asciiz.* Difference between these two is; in case of *.asciiz*, a string terminating character (null character) is automatically inserted at the end of the string while in case of *.ascii*, no string terminating character is added. The contents of the string are enclosed by quotes and the *syscall* for displaying the string on the screen require the value '4' in the register $v0 while the address of the string must be in the register $a0.

Program: Displaying the string

```
.data
str:    .asciiz         “hello world”

.text
main:

la      $a0, str
li      $v0, 4
syscall

li      $v0, 10
syscall
```

Program: Counting the number of characters in a string. Register $t1 will contain the result.

```
.data
str:    .asciiz         “hello world”

.text
main:
la      $t0, str
li      $t1, 0
lb      $t2, ($t0)

start:
beqz    $t2, endloop
add     $t1, $t1, 1
add     $t0, $t0, 1
lb      $t2, ($t0)
j start
endloop:
li      $v0, 10
syscall
```

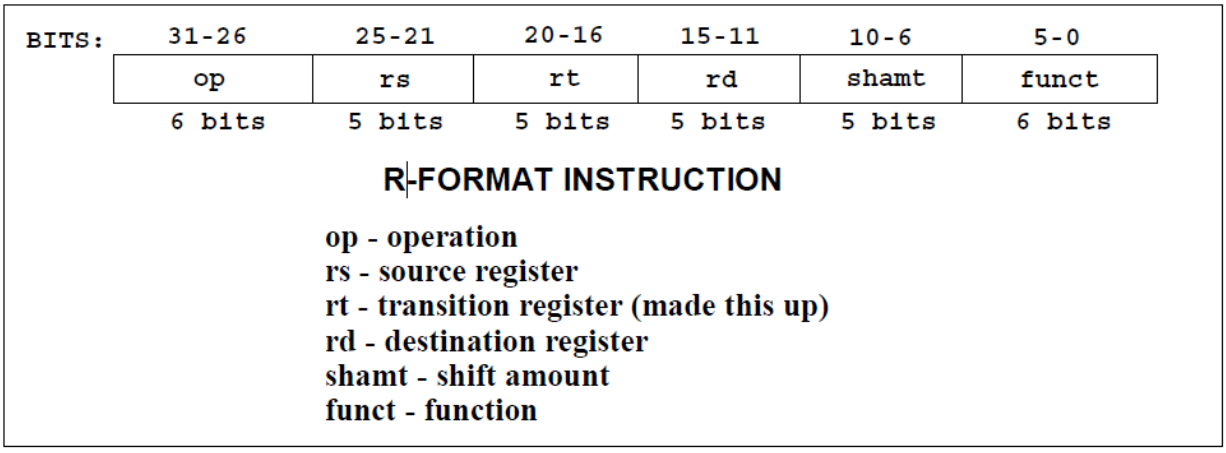
Lab 12: Formats of MIPS Instructions

What are formats?

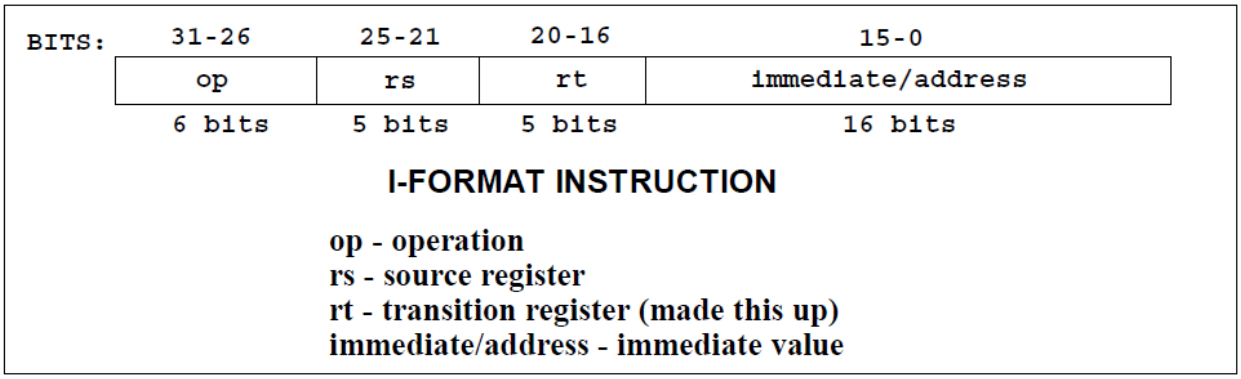
Formats are patterns that instructions fit so that the microprocessor can tell what to do. Rather than hard coding each possible instruction as a unique possibility1, the designers of MIPS decided to make instructions follow certain patters that allowed them to extract useful information out of the 32 bits and use it in a more general manner. One group of bits defines one action, another group of bits defines another action.

Format Types:

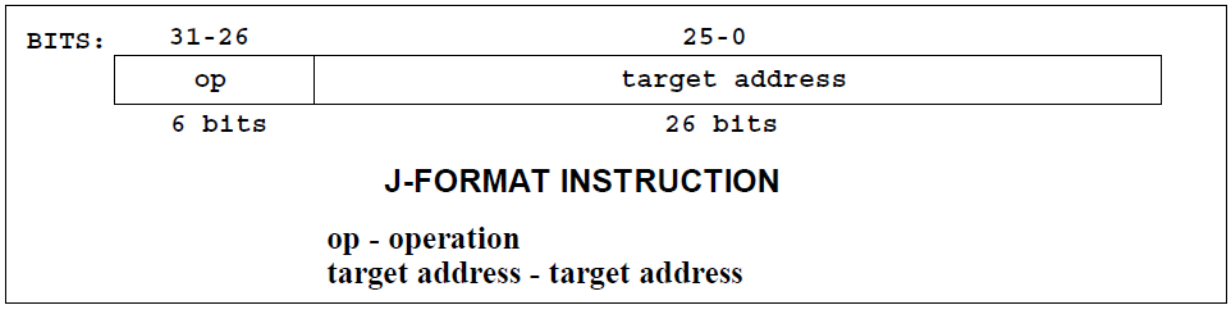
There are three types of instruction formats, R-format, I-format, and J-format. The R stands for a register-format instruction, an I stands for an immediate-format instruction, and a J stands for a jump-format instruction. The R-format instructions mainly handle only the register instructions because three of the fields in the format tell the MIPS processor what registers are being used.



The *operation field* is 6 bits and is found in all of the formats, in the exact same location as shown above. This field tells the microprocessor, not only what type of instruction is being evaluated, but what type of format is being used. In the R-format, the *source* and *transition register fields* tell the microprocessor which registers will be acting as source registers (registers left unchanged by the operation; only their value is used). The *destination register field*, only existing in the R-format, indicates which register will hold the final value of an operation. The *shift amount field* is used in shift operations to indicate how much to shift a given register. Notice that it is 5 bits, so it can shift 32 bits. The *function field*, possibly getting it's name from "math functions", is used in addition to the operation field for determining the type of R-format instruction.



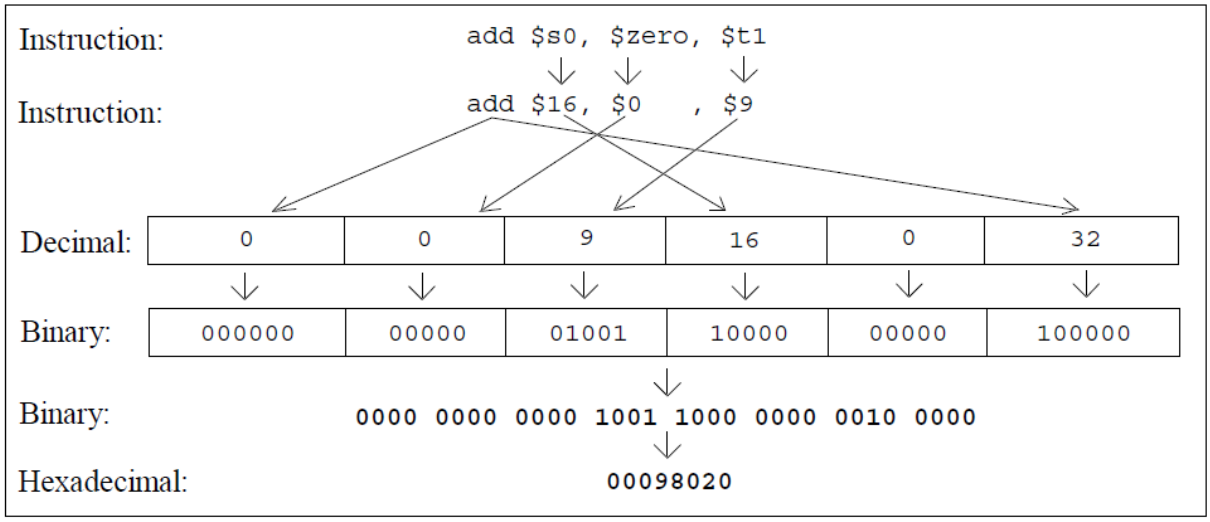
Whereas many instructions require registers to function, some can take 16 bit values imbedded within the instruction. This is used in MIPS whenever a constant value is encountered. In the I-format, the *transition register field* is used as both a destination register and a source register (hence it's transitional). The *immediate value field* contains a 16 bit value, which may or may not be signed, depending on the instruction. Any value larger than 16 bits must be loaded into a register using *lui* and another instruction, such as *ori*.



The J-format is only capable of operating given a constant value, an address. Some special J-format instructions *do* operate on registers, even though they aren't specified directly. The *target address field* only contains addresses. If an address in excess of a 28 bit value[3] is needed, a register can be used to hold the address.

Interpreting the values of formats

Each field in a format consists of a bunch of bits. These bits are read as numbers which define a how the instruction should be interpreted. For example, if you take an R-format instruction, the *rs*, *rt*, and *rd* fields are all 5 bit numbers representing the register being used. (Recall that there are only 32 registers and 5 bits can represent the numbers 0 through 31.) The *op*, *shamt*, and *funct* fields are all treated in the same way. For example what if we had the instruction: `add $s0, $zero, $t1`



Translating instructions from binary into assembly language is just as straight forward. Just reverse the arrows and go backwards.