



Under The Hood

Programming Club

Science and Technology Council

EndTerm Evaluation Report
24 July, 2022

Contents

1	PHASE I - BOOLEAN ALGEBRA	4
1.1	BINARY REPRESENTATION OF NUMBERS	4
1.1.1	Unsigned Integers	4
1.1.2	Signed Integers	4
1.2	BOOLEAN OPERATORS	4
1.2.1	Basic Boolean operators	4
1.2.2	Useful Properties of Boolean Operators	5
1.3	TRUTH TABLE	5
1.3.1	Truth Table Diagram	5
1.4	LOGIC GATES	6
1.4.1	AND Gate	6
1.4.2	OR Gate	6
1.4.3	NOT Gate	7
1.4.4	NAND Gate	7
1.4.5	NOR Gate	7
1.4.6	XOR Gate	8
1.4.7	XNOR Gate	8
1.4.8	Universal Gates	8
1.5	FUNCTIONS	8
1.6	K-MAPS	9
1.6.1	Simplifying K-Maps	9
1.6.2	Examples	9
1.7	ELECTRONIC DEVICES	10
1.7.1	Adder	10
1.7.2	Subtractor	11
1.7.3	MUX	11
1.7.4	Comparator	12
1.7.5	Clock	12
1.7.6	Memory	13
2	PHASE II - VERILOG PROGRAMMING	14
2.1	BASICS	14
2.1.1	Intro	14
2.1.2	Variable types	14
2.1.3	Timescale	14
2.2	VERILOG LANGUAGE FEATURES	14
2.2.1	Modules	14
2.2.2	Initial Block	15
2.2.3	Always Block	15
2.2.4	If-else Block	16
2.3	TESTING A VERILOG PROGRAM	16
2.3.1	Creating a Test bench	16
2.3.2	Running the simulation	17
2.4	FINITE STATE MACHINE(FSM)	18

2.4.1	Principle	18
2.4.2	An example: Elevator Control	18
3	PHASE III - MIPS ASSEMBLY LANGUAGE	19
3.1	Intro to MIPS architecture	19
3.2	General MIPS programme structure	20
3.3	Manipulating registers	21
3.3.1	Data Transfer	21
3.3.2	Logical	21
3.3.3	Arithmetic	21
3.4	Flow Control	22
3.4.1	Branching	22
3.4.2	Jump	22
3.5	An Example: To see so far and to see more	22
3.6	QTspim	24
3.7	Application: Assignment	24
4	PHASE IV - BUILD APPLICATIONS USING MIPS	25
4.1	Build Tic Tac Toe using MIPS:	25
4.1.1	Problem Statement	25
4.1.2	Requirements	25
4.1.3	Implementation	25
4.1.4	Results	26
4.1.5	Team	28
4.1.6	Example:	29
4.2	Build Scientific Calculator using MIPS:	30
4.2.1	Problem Statement:	30
4.2.2	Scientific Calculator:	30
4.2.3	Input-Output Interfacing:	30
4.2.4	LOG	32
4.2.5	GCD	32
4.2.6	LCM	33
4.2.7	POWER	34
4.2.8	ROOT	35
4.2.9	Members	36
4.3	Build Simple Calculator using MIPS	36
4.3.1	Problem Statement:	36
4.3.2	Introduction	37
4.3.3	Example	37
4.3.4	Instructions	38
4.3.5	Members	38
4.4	Build Date Converter Using MIPS	38
4.4.1	Introduction and Problem Statement	38
4.4.2	Algorithm and Working	39
4.4.3	Interface Screenshot	40
4.4.4	Members	41

1 PHASE I - BOOLEAN ALGEBRA

1.1 BINARY REPRESENTATION OF NUMBERS

1.1.1 Unsigned Integers

In binary representation, numbers are stored in bits where each bit can be either 0 or 1. Let X be any number and $a_0, a_1, a_2, \dots, a_{n-1}$ denote the n digits of an n -bit binary representation of X where a_0 is the LSB and a_{n-1} is the MSB, then decimal representation of X is given by:

$$X = a_0 2^0 + a_1 2^1 + a_2 2^2 + \dots + a_{n-1} 2^{n-1}$$

Note: An n -bit unsigned binary number can store 2^n values, from 0 to $2^n - 1$.

1.1.2 Signed Integers

In signed integers, 1-bit has to be reserved to store the sign. There are two major ways of storing signed integers in binary.

1's complement: Flip all the bits of a number to get the negative of that number.

Major Drawbacks with this method:

1. Zero has 2 different representations.
2. Only $2^n - 1$ numbers can be stored.

2's complement: Negative of a number is equal to its 1's complement + 1.

An easy way to find 2's complement of a number is that, starting from the LSB copy the bits until the first 1 is encountered, and flip every bit after that, The number we get is the 2's complement of the given number.

1.2 BOOLEAN OPERATORS

1.2.1 Basic Boolean operators

Boolean algebra mainly consists of 3 Boolean operators, which are AND, OR and NOT. Using these three operators, other operators like NAND, NOR, XOR and XNOR can be constructed.

1. AND operator: It is denoted by (\cdot) in boolean algebra. It takes two operands, if both the operands of AND are 1 then it gives 1 as output, otherwise it gives output 0.
2. OR operator: It is denoted by $(+)$ in boolean algebra. It takes two operands, if either or both the operands of OR are 1 then it gives 1 as output, otherwise, if both the operands are 0 then the output is 0.
3. NOT operator: It is denoted by $(A'$ or $\bar{A})$ in boolean algebra. It acts on only one operand and flips its value.

The XOR Operator is denoted by $A \oplus B$ and can be constructed from the above operators as $A \oplus B = A \cdot \bar{B} + \bar{A} \cdot B$ XOR operator has a unique property while checking equality that is $A \oplus B = 0 \iff A = B$

1.2.2 Useful Properties of Boolean Operators

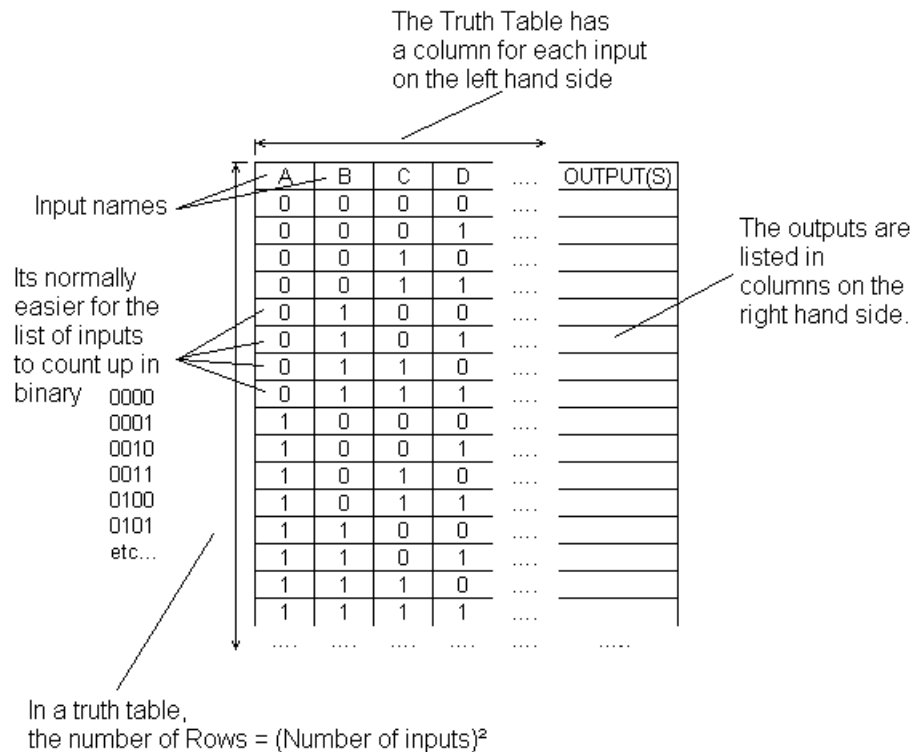
$$\begin{array}{lll}
 A.1 = A & A.0 = 0 & A + 1 = 1 \\
 A + 0 = A & A.\bar{A} = 0 & A + \bar{A} = 1 \\
 A.(B + C) = A.B + A.C & \overline{A + B} = \bar{A}.\bar{B} & \overline{A.B} = \bar{A} + \bar{B}
 \end{array}$$

1.3 TRUTH TABLE

Truth tables are used to help show the function of a logic gate. Truth tables help understand the behaviour of logic gates.

- They show how the input of a logic gate relate to its output.
- The gate inputs are shown in the left columns of the table with all the different possible input combinations. This is normally done by making the inputs count up in binary.
- The gate outputs are shown in the right hand side column.

1.3.1 Truth Table Diagram



1.4 LOGIC GATES

Boolean functions are practically implemented by using electronic gates (aka logic gates). Electronic gates require a power supply. Gate INPUTS are driven by voltages having two nominal values, e.g. 0V and 5V representing logic 0 and logic 1 respectively. The OUTPUT of a gate provides two nominal values of voltage only, e.g. 0V and 5V representing logic 0 and logic 1 respectively.

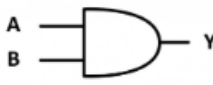
Digital systems are said to be constructed by using logic gates.

The basic gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations of these gates are described below with the aid of truth tables:

1.4.1 AND Gate

The AND gate is an electronic circuit that gives a high output (1) only if all its inputs are high. The AND gate is so named because, if 0 is called "false" and 1 is called "true," the gate acts in the same way as the logical "and" operator.


Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1


$$Y = A.B$$

A dot (\cdot) is used to show the AND operation i.e. $A.B$. This dot is sometimes omitted i.e. AB

1.4.2 OR Gate

Inputs		Output
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1


$$Y = A+B$$

The OR gate is an electronic circuit that gives a high output (1) if one or more of its inputs are high. A plus (+) is used to show the OR operation.

1.4.3 NOT Gate

Input	Output
A	Y
0	1
1	0

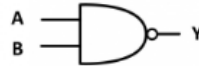


$$Y = \overline{A}$$

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an inverter. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs.

1.4.4 NAND Gate

Inputs		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



$$Y = \overline{A \cdot B}$$

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if any of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

1.4.5 NOR Gate

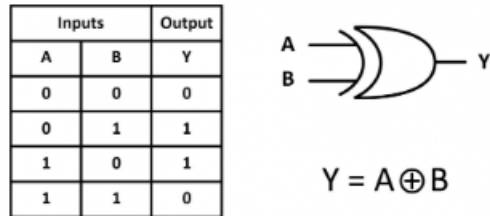
Inputs		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



$$Y = \overline{A + B}$$

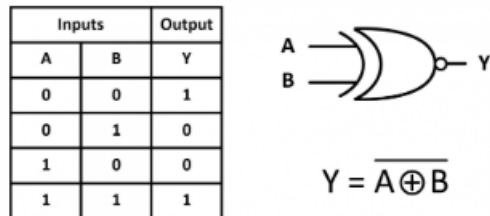
This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if any of the inputs are high. The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

1.4.6 XOR Gate



The 'Exclusive-OR' gate is a circuit which will give a high output if either, but not both, of its two inputs are high. An encircled plus sign (\oplus) is used to show the XOR operation.

1.4.7 XNOR Gate



The 'Exclusive-NOR' gate circuit does the opposite to the XOR gate. It will give a low output if either, but not both, of its two inputs are high. The symbol is an XOR gate with a small circle on the output. The small circle represents inversion.

1.4.8 Universal Gates

The NAND and NOR gates are called universal functions since with either one the AND and OR functions and NOT can be generated.

1.5 FUNCTIONS

Any truth table involving n variables with 2^n outputs can be expressed as a function having input, a combination of n bits mapping to particular output bit. This function involves those n Boolean variables in a logical expression. At times these expressions can be very complicated involving many complex relations between numerous variables, to simplify it we use *K-Maps*.

1.6 K-MAPS

In digital circuits, to find expression with minimum variables, we minimize and simplify Boolean expressions of 3, 4 variables easily using K-map (Karnaugh Maps). K-map is table like representation but it gives more information than TRUTH TABLE.

A boolean function can be represented in two forms (i) Sum of Product (SOP) and (ii) Product of Sum (POS).

1.6.1 Simplifying K-Maps

To simplify expression using K-Maps:

1. Select K-map according to the number of variables.
2. Identify minterms or maxterms as given in problem.
3. For SOP put 1's in blocks of K-map respective to the minterms (0's elsewhere).
4. For POS put 0's in blocks of K-map respective to the maxterms(1's elsewhere).
5. Make rectangular groups containing total terms in power of two like 2,4,8 ..(except 1) and try to cover as many elements as you can in one group.
6. From the groups made in step 5 find the product terms and sum them up for SOP form.

K-Map for 3 variables is as follows:

x	y	z	F
0	0	0	m0
0	0	1	m1
0	1	0	m2
0	1	1	m3
1	0	0	m4
1	0	1	m5
1	1	0	m6
1	1	1	m7

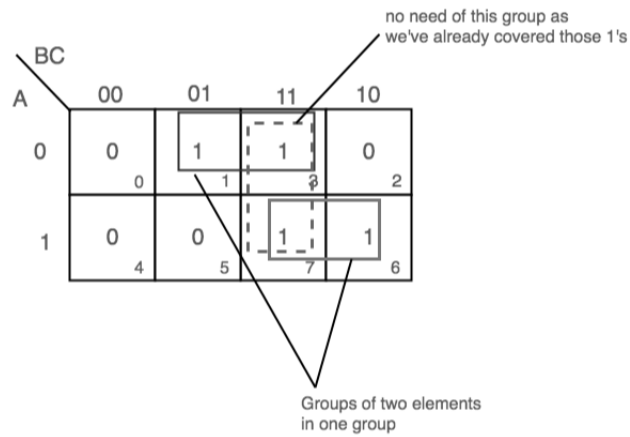
3-variable Truth table of F

		yz			
		y'z'	y'z	yz	yz'
x	x'	m0 x'y'z' 0	m1 x'y'z 1	m3 x'yz 3	m2 x'yz' 2
	y	m4 xy'z' 4	m5 xy'z 5	m7 xyz 7	m6 xyz' 6

3-Variable K-Map, minterm and cell position

1.6.2 Examples

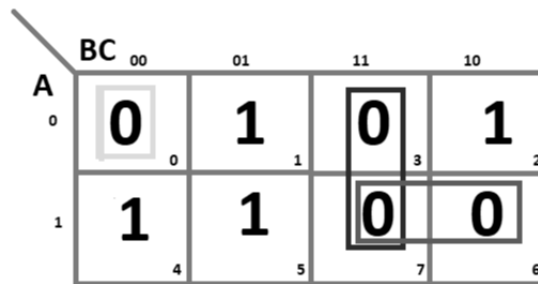
- (i) SOP form - $f(A, B, C) = \Sigma(1, 3, 6, 7)$



From the groups we get terms $\bar{A}C$ and AB . To get SOP form, we sum these product terms, and get the final expression:

$$\bar{A}C + AB$$

(ii) POS form - $f(A, B, C) = \Pi(0, 3, 6, 7)$



After finding terms from the groups and complimenting and adding them we get $\bar{A} + \bar{B}$, $\bar{B} + \bar{C}$, $A + B + C$. To get POS form, we will take product of these three terms:

$$(\bar{A} + \bar{B}).(\bar{B} + \bar{C}).(A + B + C)$$

1.7 ELECTRONIC DEVICES

1.7.1 Adder

An adder adds binary numbers and accounts for values carried in as well as out. A one-bit adder adds three one-bit numbers, often written as A, B, and

C_{in} ; A and B are the operands, and C_{in} is a bit carried in from the previous less-significant stage.

Output carry and sum typically represented by the signals C_{out} and S. Adder is implemented in the following way:

$$\begin{aligned} S &= A \oplus B \oplus C_{in} \\ C_{out} &= AB + C_{in}B + C_{in}A \end{aligned}$$

When performing addition of multi-bit numbers an adder is used for each i^{th} bit and the corresponding output carry of i-th adder becomes input carry for (i+1)th bit adder i.e

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_{in} \\ C_{i+1} &= A_i B_i + C_i B_i + C_i A_i \end{aligned}$$

where C_i represents input carry for adder of i-th bit. Initial carry for adder (c_0) is 0.

A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

1.7.2 Subtractor

Subtractors are usually implemented using the same approach as that of an adder, by using the standard *two's complement* notation, by providing an addition/subtraction selector to the initial carry and to invert the second operand.

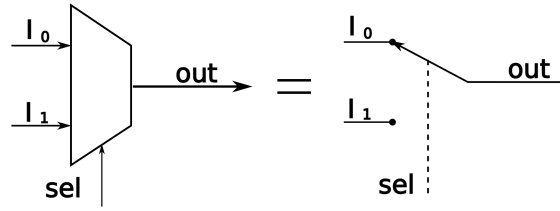
By definition of 2's complement

$$\begin{aligned} -B &= \bar{B} + 1 \\ A-B &= A + \bar{B} + 1 \end{aligned}$$

Addition/subtraction selector can be provided by giving A, ($B \oplus \text{op}$), and op as input, where it acts as adder when op = 0 and as subtractor when op = 1.

1.7.3 MUX

A MUX (we see, 2-to-1 multiplexer) consists of two inputs I_0 and I_1 , one selector input (sel, S) and one output *out*. Depending on the selector value, the output (out, Y) is either of the inputs (I_0 or I_1).



MUX can be equated to a controlled switch; a hardware level analog of if-else. Truth table is as follows

<i>sel</i>	<i>out</i>
0	$I(0)$
1	$I(1)$

and logical expression is

$$Y = I_0\bar{S} + I_1S$$

1.7.4 Comparator

A comparator takes two numbers as input and as the name suggests, it compares them. It will have three possible outputs: greater, equal, and smaller, and only one of them will be set to 1. The outputs depicts $G = 1$ when $A > B$, $E = 1$ when $A = B$ and $L = 1$ when $A < B$.

Eg:

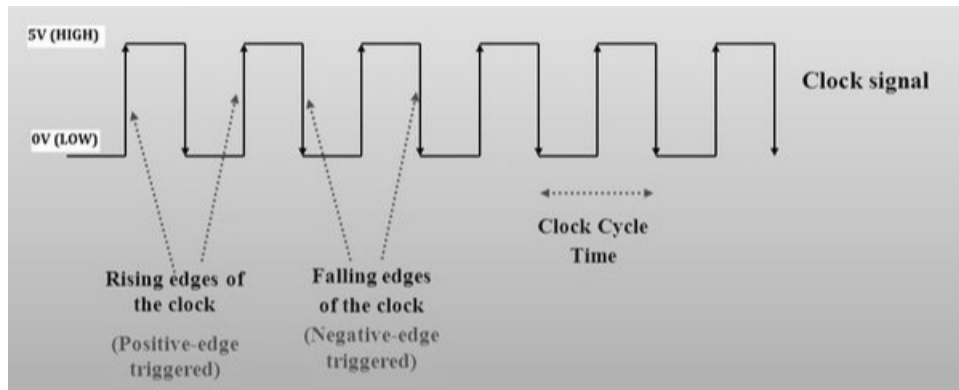
<i>A</i>	0	0	1	1
<i>B</i>	0	1	0	1
<i>G</i>	0	0	1	0
<i>E</i>	1	0	0	1
<i>L</i>	0	1	0	0

From this we observe,

$$\begin{aligned} G &= A \bar{B} \\ E &= A \odot B \\ L &= \bar{A}B \end{aligned}$$

1.7.5 Clock

Clock is used to bring a sense of chronology in computers. To do so, they require an external signal (an alternating boolean variable), apart from standard inputs to operate, such external signal is referred as clock or clock pulse. The idea of clock is to control the output even if the input is given.



- There are two levels, namely logic High and logic Low in clock signal. The signal stays at a *logic level*, either high 5V or low 0V, for an equal amount of time.
- In clock signal, two types of transitions occur, i.e. transition either from Low to High or High to Low. These transitions are called *edge*.
- The transition from low to high is called positive edge, and that from high to low is called negative edge. These transitions can be used for *triggering*.

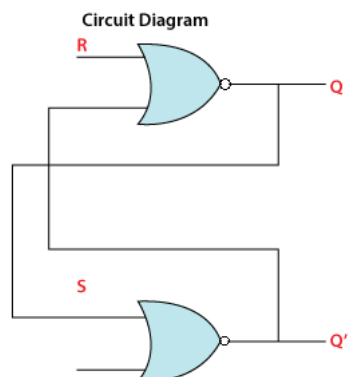
1.7.6 Memory

Latches and flip flops are elements to mimic memory.

1. LATCHES

Latches are *level triggered*.

SR Latch:



Truth table for SR Latch:

	S	R	Q_t	Q_{t+1}
<i>hold</i>	0	0	0	0
<i>hold</i>	0	0	1	1
<i>reset</i>	0	1	0	0
<i>reset</i>	0	1	1	0
<i>set</i>	1	0	1	1
<i>set</i>	1	0	1	1
<i>race</i>	1	1	0	?
<i>race</i>	1	1	1	?

2. FLIP-FLOPS

Flip-flops are *edge triggered*.

2 PHASE II - VERILOG PROGRAMMING

2.1 BASICS

2.1.1 Intro

Verilog is a Hardware Description Language(HDL) used for designing digital circuits, it has a modular structure and syntax more or less like C.

2.1.2 Variable types

A variable in verilog belongs to two types:

1. Net: This variable type is one which stores continuously driven signals of input or output in verilog. Among many examples of this data type is "wire" which we used in many programs.
2. Reg: This variable type retains the last value assigned to it and so used as a storage element.

2.1.3 Timescale

Typical format to declare an inclusion of timescale in verilog program is 'timescale <time_unit>/<time_precision>. Time unit can be 1ms,1ns etc. which gives a time unit to statements like #3 which means a delay by 3 time units. Time precision is the precision of the time clock, the least count.

2.2 VERILOG LANGUAGE FEATURES

2.2.1 Modules

Modules in verilog are same as functions in any higher level language like C in its working. The format of a module in verilog is as follows:

```

module module_name(list_of_ports);
    io declarations
    parallel statements
    ...
    ...
endmodule

```

A verilog program consists of multiple modules which perform different functions. A typical example of a module(AND gate) would be:

```

module and_Gate(a,b,c);
    input a,b;
    output c;
    assign c = a&b;
endmodule

```

Here all parameters are by default 1-bit numbers. One can easily define a variable to be a vector of multiple bits for say a 16 bit variable can be declared as var_type[15:0]var_name. Bits are stored from MSB: var_name[0] to LSB: var_name[15].

Here we see a new keyword "assign". This keyword is used to represent continuous assignment where the left variable is updated whenever the expression on the right changes. Here the LHS variable is usually a wire and RHS can contain both reg and a wire.

2.2.2 Initial Block

It is a block inside a module used for a specific function of running the code whatever is inside it only once the program is run. **Every such code block has a begin and end statement inside which statements are written:**

```

initial begin
    ...
    statements
    ...
end

```

2.2.3 Always Block

Another useful paradigm in verilog programming is an Always block which runs every time the given condition is satisfied. The condition being typically a sensitivity statement which checks for changes in multiple parameters passed into that statement:

```

always @(parameters) begin

```



```

...
    statements
...
end

```

2.2.4 If-else Block

In verilog if-else statements are declared just like C:

```

if (conditional) begin
    operations;
end else begin
    operations;
end

```

Once we are done with making a program using above features in verilog, we need to simulate the system and verify its operations just like running a program in a high level language by providing inputs. For this we need to create a test bench and run it to see the outputs.

2.3 TESTING A VERILOG PROGRAM

2.3.1 Creating a Test bench

A test bench is yet another module inside which we call another module which we want to test, In verilog calling a module inside another module is called instantiation which actually keeps a copy of that module inside it. Then inside an "initial-block" we pass desired inputs and simulate the outputs to readable format using different system tasks like "monitor" and "display". Monitor function is like a smart `printf` statement which only outputs if any one of the parameters changes while display is just like a `printf` statement. A typical example of a test bench is shown below:

```

module tb;

    reg a, B;
    wire C;
    andMod ans(a, B, C);

    initial begin
        $monitor("a = %b, B = %b, C = %b", a, B,C);
        a = 0;
        B = 0;
        #2;

        repeat(4) begin
            a = $random;
            B = $random;
            #15;
        end
    end
endmodule

```

Here the repeat block does the same thing the specified no. of times and the random statement assigns the variable a value randomly. Note how each block is in between a begin and end statement.

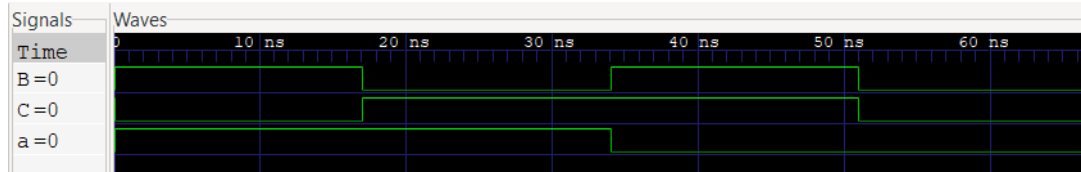
2.3.2 Running the simulation

Now commands iverilog -o name file.v and vvp name gives the output on the terminal. This output can be simulated in gtkwave by adding two lines:

```
$dumpfile("file_name.vcd");
```

```
$dumpvars(0, tb);
```

inside the initial block of test bench. This will create a .vcd file in same directory, now u can type in terminal gtkwave file_name.vcd to open the gtkwave and see the simulation there which typically looks like this:



2.4 FINITE STATE MACHINE(FSM)

2.4.1 Principle

FSM is a reactive system whose response to a particular stimulus (a signal, or a piece of input) is not the same on every occasion, depending on its current “state”. A Finite State Machine is defined by $(\Sigma, S, S_0, \delta, F)$, where:

1. Σ is the input alphabet (a finite, non-empty set of symbols).
2. S is a finite, non-empty set of states.
3. s_0 is an initial state, an element of S .
4. δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$
5. F is the set of final states, a (possibly empty) subset of S .
6. O is the set (possibly empty) of outputs

The Finite State Machine class keeps track of the current state, and the list of valid state transitions. You define each transition by specifying :

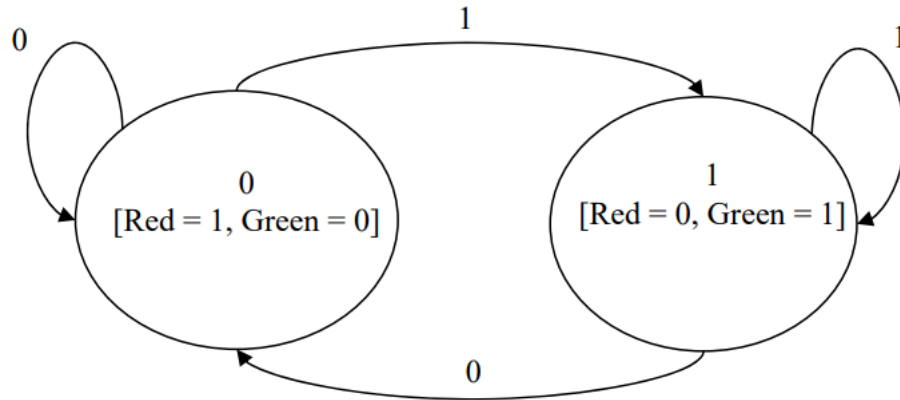
1. FromState - the starting state for this transition
2. ToState - the end state for this transition
3. condition - a callable which when it returns True means this transition is valid.
4. callback - an optional callable function which is invoked when this transition is executed.

2.4.2 An example: Elevator Control

Following example will provide a better picture of FSM:

Step 1: Describing the machine: In this example, we’ll be designing a controller for an elevator. The elevator can be at one of two floors: Ground or First. There is one button that controls the elevator, and it has two values: Up or Down. Also, there are two lights in the elevator that indicate the current floor: Red for Ground, and Green for First. At each time step, the controller checks the current floor and current input, changes floors and lights in the obvious way.

Step 2: Draw FSM diagram.



Step 3: Write the truth table.

CurrentState	Input	NextState	Red	Green
0	0	0	1	0
0	1	1	1	0
1	0	0	0	1
1	1	1	0	1

3 PHASE III - MIPS ASSEMBLY LANGUAGE

3.1 Intro to MIPS architecture

1. Literals: In the MIPS architecture, literals represent all numbers (e.g. 5), characters enclosed in single quotes (e.g. 'x') and strings enclosed in double quotes (e.g. "Qtspin").
2. Registers: MIPS architecture uses 32 registers. Each register is preceded by '\$' in the instruction. You can address these registers in two ways. Either use the register's number (from 0 to 31), or the register's name (for example, \$t1).

Name	Register	Usage
\$zero	\$0	the constant value 0
\$v0-\$v1	\$2-\$3	values for results and expression evaluation
\$a0-\$a3	\$4-\$7	arguments
\$t0-\$t7	\$8-\$15	temporaries
\$s0-\$s7	\$16-\$23	saved
\$t8-\$t9	\$24-\$25	more temporaries
\$gp	\$28	global pointer
\$sp	\$29	stack pointer
\$fp	\$30	frame pointer
\$ra	\$31	return address

3.2 General MIPS programme structure

A program created using the MIPS assembly language has two parts. They are the data declaration section and the code section.

Data section : The data section of the program is identified with the assembler directive `.data`. In this part all the variables to be used in the program are created and defined.

A typical example of variable definition is:

```
var1: .word 5 i.e.
name: .storage_type value(s)
```

Code Section : The code section of the program is the part of the program in which the instructions to be executed by the program are written. It is placed in the section of the program identified with the assembler directive `.text`. The starting point for the code section of the program is marked with the label “main” and the ending point for the code section of the program is marked with an exit system call. This section of a MIPS assembly language program typically involves the manipulation of registers and the performance of arithmetic operations.

Using the data section, writing the code section and manipulating the registers we are able to write programmes in MIPS assembly language as we see in following sections.

3.3 Manipulating registers

Several mips instructions in the code section help us to manipulate desired registers and achieve end results.

3.3.1 Data Transfer

Following are instructions used for data transfer i.e. loading values into register from memory and saving values in memory, with their implementations and descriptions:

Data transfer	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	Memory[\$s2+20]=\$s1;\$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	\$s1 = 20 * 2 ¹⁶	Loads constant in upper 16 bits

3.3.2 Logical

Following are instructions used for Logical operations between different values stored in different registers with their implementations and descriptions:

Logical	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2 \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	\$s1 = \$s2 20	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant

3.3.3 Arithmetic

Following are instructions used for Arithmetic operations with their implementations and descriptions:

Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three register operands
	add immediate	addi \$s1,\$s2,20	\$s1 = \$s2 + 20	Used to add constants

3.4 Flow Control

Like in any higher language like C, control flow in MIPS assembly language is done by using branching and looping techniques which are implemented in MIPS in a slightly different way using following instructions.

3.4.1 Branching

Each instruction consists of a keyword which executes a specific logical operation between registers and under a condition satisfied branches to a specific label mentioned, i.e. jumps to the label and executes code starting from that label.

Conditional branch	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned

3.4.2 Jump

Following instructions provide the iterative feature like of a for loop in C and the functionality of return statement.

Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

3.5 An Example: To see so far and to see more

There are several new things to notice in the following example assembly code for Printing the sum of first N natural numbers.

```

        .text
        .globl main
main:
    # Print msg1
    li    $v0,4          # print_string syscall code = 4
    la    $a0, msg1
    syscall

    # Get N from user and save
    li    $v0,5          # read_int syscall code = 5
    syscall
    move   $t0,$v0        # syscall results returned in $v0

    # Initialize registers
    li    $t1, 0          # initialize counter (i)
    li    $t2, 0          # initialize sum

loop:   # Main loop body
    addi   $t1, $t1, 1    # i = i + 1
    add    $t2, $t2, $t1  # sum = sum + i
    beq    $t0, $t1, exit # if i = N, continue
    j      loop

exit:   # Exit routine - print msg2
    li    $v0, 4          # print_string syscall code = 4
    la    $a0, msg2
    syscall
    # Print sum
    li    $v0,1          # print_int syscall code = 4
    move   $a0, $t2
    syscall
    # Print newline
    li    $v0,4          # print_string syscall code = 4
    la    $a0, lf
    syscall
    li    $v0,10         # exit
    syscall

        .data
msg1:   .asciiz "Please enter the input (N)? "
msg2:   .asciiz "Sum = "
lf:     .asciiz "\n"

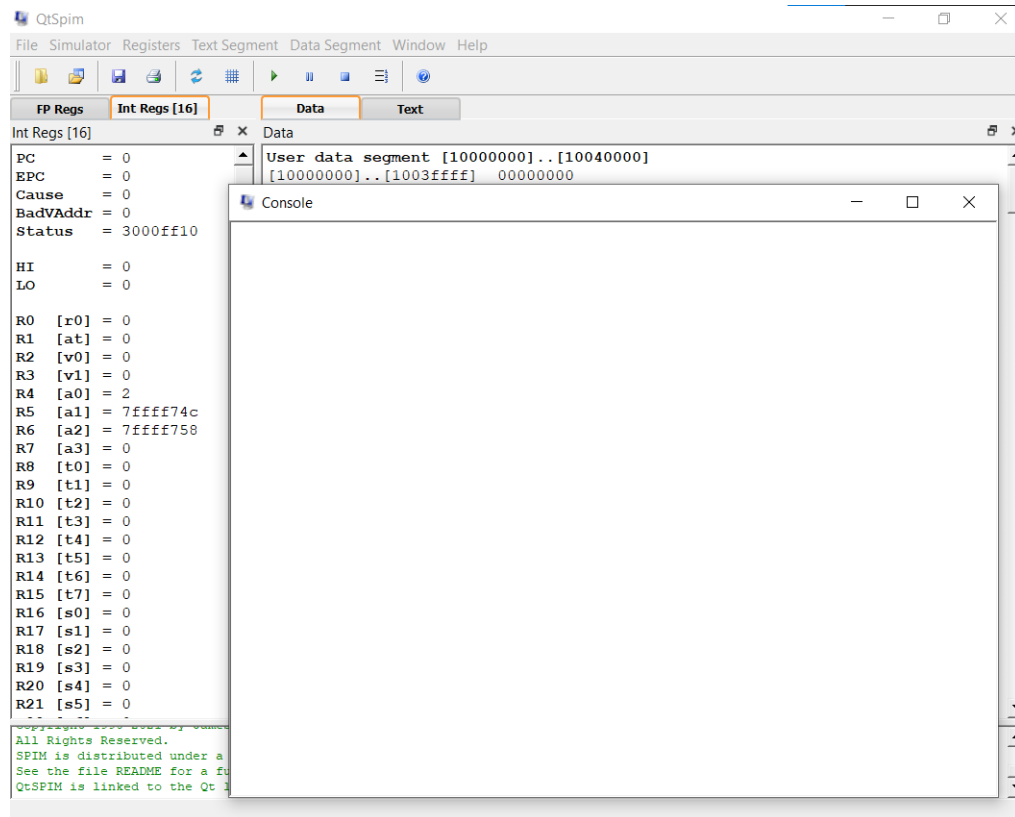
```

`syscall` is an instruction that triggers a system call that causes control to be transferred from user space to kernel space where the system call is handled. We have different codes(integers) for different instructions which are loaded into `$v0` before making a `syscall`. For example loading a value of 10 in register `$v0` results in execution of `exit` at `syscall`.

Now that we have written an assembly code, we need to see its execution and `QTspim` is a tool which helps us in the same.

3.6 QTspim

QTspim is a tool(a processor simulator) used to run a MIPS assembly language code. We can load our .asm or.s file into the tool and see the execution of the code in the console window. Here is how the tool looks:



On the left we can see of the registers that we have seen theoretically which are used for their functions and some extra features.

3.7 Application: Assignment

In assignment of this phase we were asked to apply knowledge of MIPS architecture to implement following three tasks:

1. GCD
Given two non-negative integers we were asked to give their GCD. The Euclidean algorithm of finding GCD was implemented using conditionals and loops in MIPS assembly language.
2. Iterative Fibonacci
Given a non-negative integer $1 < n < 40$ we were asked to calculate $F(n)$

where $F(0) = 0$ and $F(1) = 1$. This was to be implemented using loops(i.e. using j).

3. Recursive Fibonacci

Given a non-negative integer $1 < n < 40$ we were asked to calculate $F(n)$ where $F(0) = 0$ and $F(1) = 1$. This was to be implemented using recursion(i.e. using jal).

4 PHASE IV - BUILD APPLICATIONS USING MIPS

4.1 Build Tic Tac Toe using MIPS:

4.1.1 Problem Statement

The final phase of the project required us to implement a **Tic Tac Toe** game in **MIPS Assembly Language**. Creation of user interface and the algorithm for finding the *best computer move* were needed to be coded.

The interface contain the **current status of the board** like:

```
- - -  
0 X -  
0 - -
```

User always has the first turn. And the user can continue his turn by entering the desired position as input:

```
1 2 3  
4 5 6  
7 8 9
```

At the end, the game tells **who won**.

4.1.2 Requirements

The project was to develop the well known 'tic-tac-toe' game using MIPS Assembly language.

The game is to be played between the **User and Computer** with user playing the first turn.

4.1.3 Implementation

An I/O was developed that let's the user choose where he/she wants to place their token and shows the current situation of board.

The code also validates that the user input is a valid and empty location on board, if not, it prompts the user to try again. Additionally, we have written the code in such a way that the computer is now **unbeatable**.

We realised that the user's responses throughout the game can be represented as a single integer, which can be stored in a register. (This single integer can be thought of as the state of the game as it defines the whole game)

We have put this observation to use by making a 'state' vs 'optimal computer solution' relationship.

We also analysed that the no. of different states can go upto 1332.

Even though it is a huge number, it was still feasible to make all the possible states algorithmically and find the optimal response for each state.

Once this relationship is established, the only tasks left for the computer are -

1. Incorporating the user's response and generate a new state on each turn of the game
2. Lookup the current state in the list of possible states and implement its corresponding response
3. Check whether the game is a victory for a computer or a draw (as mentioned earlier victory of user is not possible)

Incorporating these things, our final solution was ready if perfectly working condition.

4.1.4 Results

We accomplished the task of developing the Tic-Tac-Toe game.

The general workflow of the game looks as follows:

1. The computer presents the instruction to user

```
|-----|
|  Tic-Tac-Toe  |
|-----|

Instructions:
1. You play as X and Computer plays as O, first turn is yours
2. The position on the grid are marked as :
   1 2 3
   4 5 6
   7 8 9
3. On your turn, enter the position no. where you want to place 'X'

Game starts!

-- --
-- --
-- --
```

2. It prompts the user for a position to place 'X' at

```

Enter a position to put 'X': 1
Computer's response: Position 5
X _ _
_ O _
_ _ _

```

```

Enter a position to put 'X': |

```

3. Validates the user's input

```

Enter a position to put 'X': 3
Computer's response: Position 5
  _ _ X
  _ O _
  _ _ _

```

```

Enter a position to put 'X': 5
ERROR! That's an invalid position, try again!

```

```

Enter a position to put 'X':

```

4. Finds the optimal response to current game
5. Prints the updated grid
6. Checks for victory / draw

```

Enter a position to put 'X': 9

X X O
O O X
X O X

It's a DRAW
Thanks for playing. Hope you enjoyed the game.
|

```

7. At this point there are 3 possibilities:

- (a) It's a victory for computer: The computer tells the user that Computer has won
- (b) It's a draw: The computer declares the game as draw when all the cells are filled

```

Enter a position to put 'X': 9

X X O
O O X
X O X

It's a DRAW
Thanks for playing. Hope you enjoyed the game.
|

```

- (c) Continued with the next turn for user: The computer moves back to Step 2 and follows the execution from that step.

4.1.5 Team

qtsimp

Lavesb Gupta
 Anuj Singhal
 Ujjawal Dubey
 Rajat Gattani
 Aayush Aggarwal

4.1.6 Example:

A full game run is shown below:

```
-----
|   Tic-Tac-Toe   |
|-----|

Instructions:
1. You play as X and Computer plays as O, first turn is yours
2. The position on the grid are marked as :
   1 2 3
   4 5 6
   7 8 9
3. On your turn, enter the position no. where you want to place 'X'

Game starts!

- - -
- - -
- - -

Enter a position to put 'X': 9
Computer's response: Position 5

- - -
-  O -
-  _ X
- - -

Enter a position to put 'X': 1
Computer's response: Position 2
X O -
-  O -
-  _ X

Enter a position to put 'X': 2
ERROR! That's an invalid position, try again!

Enter a position to put 'X': 3
Computer's response: Position 6
X O X
-  O O
-  _ X

Enter a position to put 'X': 7
Computer's response: Position 4
X O X
O O O
X _ X

WINNER: Computer
Thanks for playing. Hope you enjoyed the game.
```

4.2 Build Scientific Calculator using MIPS:

4.2.1 Problem Statement:

Take two positive integers as input and a function name as input-

pow, log, gcd, lcm, root

a^b , $\log_b a$, $\gcd(a, b)$, $\text{lcm}(a, b)$, $a^{\{1/b\}}$

All of these are supposed to return integer outputs, so, take floor in log and root. Pow might lead to overflow, so calculate it modulo 4999 (a prime).

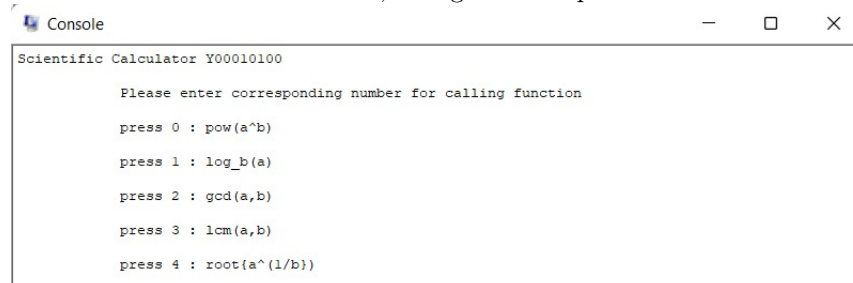
4.2.2 Scientific Calculator:

We have developed a Scientific calculator which can do the following operations on the two numbers a and b given in the input:

- 1) a^b modulo 4999
- 2) logarithm of "a" to the base "b"
- 3) gcd of a,b
- 4) lcm of a,b
- 5) a power $1/b$

4.2.3 Input-Output Interfacing:

As soon as the calculator is run, a msg0 shows up in the console-

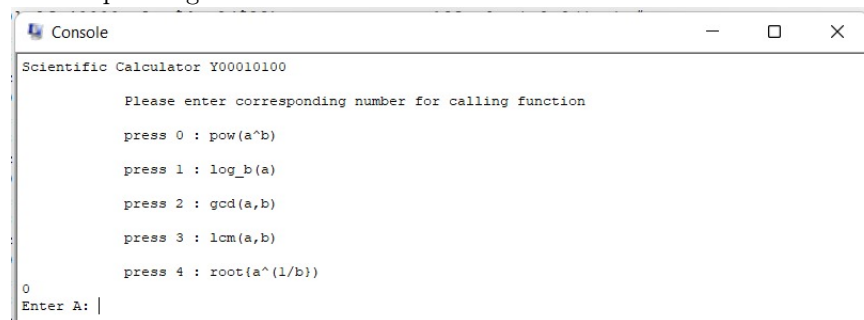


```
Console
Scientific Calculator Y00010100

Please enter corresponding number for calling function

press 0 : pow(a^b)
press 1 : log_b(a)
press 2 : gcd(a,b)
press 3 : lcm(a,b)
press 4 : root(a^(1/b))
```

Then we have to instruct the calculator to perform the desired operation by giving a number corresponding to the operation as the input. After that, msg1 shows up asking to Enter A



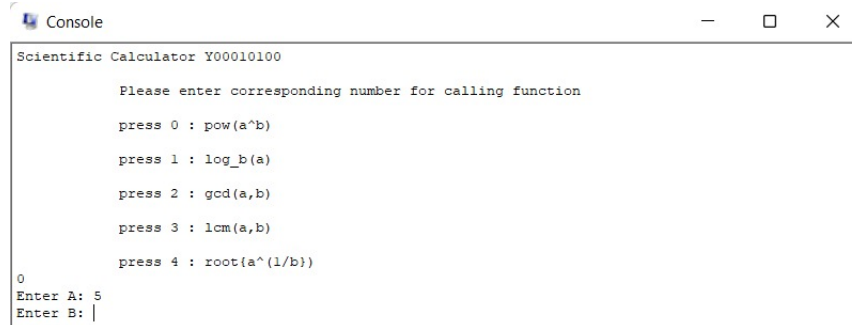
```
Console
Scientific Calculator Y00010100

Please enter corresponding number for calling function

press 0 : pow(a^b)
press 1 : log_b(a)
press 2 : gcd(a,b)
press 3 : lcm(a,b)
press 4 : root(a^(1/b))

0
Enter A: |
```

After entering A, msg2 shows up asking to enter B.



```

Scientific Calculator Y00010100

Please enter corresponding number for calling function

press 0 : pow(a^b)

press 1 : log_b(a)

press 2 : gcd(a,b)

press 3 : lcm(a,b)

press 4 : root(a^(1/b))

0
Enter A: 5
Enter B: |

```

After entering B, the following messages show up in the console depending on the operation we choose to operate on the two numbers.

```

#-----
.data
msg0:
    .asciiiz "Scientific Calculator Y00010100\n"
    .asciiiz "Please enter corresponding number for calling function\n"
    .asciiiz "press 0 : pow(a^b)\n"
    .asciiiz "press 1 : log_b(a)\n"
    .asciiiz "press 2 : gcd(a,b)\n"
    .asciiiz "press 3 : lcm(a,b)\n"
    .asciiiz "press 4 : root(a^(1/b))\n"

msg1:
    .asciiiz "Enter A: "

msg2:
    .asciiiz "Enter B: "

msg3:
    .asciiiz "Invalid Input \n"

msg4:
    .asciiiz "\n"

msg_log1:
    .asciiiz "Log_B(A) = "

msg_log2:
    .asciiiz "Log_B(A) = 0"

msg_gcd1:
    .asciiiz "GCD(A,B) = "

msg_lcm1:
    .asciiiz "LCM(A,B) = "

msg_root:
    .asciiiz "A^(1/B) = "

msg_pow:
    .asciiiz "A^B = "

msg_ty:
    .asciiiz "\n-----Thank You-----\n"

```

These Messages are explained individually in the subsequent part where each function is separately discussed.

end function:

This is the exiting point for the calculator. After the operation desired by the

user is completed and the result is printed, all operations jump to the end function. This function prints a thank you message stored in msg_ty data variable, and ends the program.

Below is the description of implementations of various functions:

4.2.4 LOG

log function:

The log function is the starting point for calculating the logarithm of A to base B, denoted as LOG_B(A). We start by initialising the result stored in register \$t3 equal to zero. Since we want the floor value of the logarithm, so we check if the base B is greater than A and then jump to the log_end2 function. We create a \$t4 register of integer value 1. We check if B¹, then we jump to the log_end3 function to print invalid output. We then jump into the log_loop function.

log_loop function:

This function divide A(\$t0) with B(\$t1) and we store the quotient in A(\$t0) then we increment the result(\$t3) by 1.

$$\log_b(a) = 1 + \log_b(a/b)$$

Then we check if \$t0 is less than or equal to one, then we jump to the print_log function; else, jump to the log_loop function to repeat the loop. print_log function: We simply print the prompt "Log_B(A) = " then print the result stored in register \$t3. After printing the newline, we jump to the end function to exit the calculator.

log_end2 function:

We print the prompt "Log_B(A) = 0" and jump to end function

log_end3 function:

We print the prompt "Invalid Input " and jump to the end function

4.2.5 GCD

gcd function:

The gcd function is the starting point for calculating the GCD of A to base B, denoted as GCD(A, B). We start by checking if \$t0 is greater than \$t1, then we jump to the main2_gcd function for swapping; else, we proceed to loop_gcd.

loop_gcd function:

This function is a loop in which \$t1 is divided by \$t0, and the quotient is stored in \$lo and modulo in \$hi. Then \$t1 is assigned to a value equal to \$t0. And \$t0 is assigned the modulo value of the \$hi. Now we check if \$t0 is equal to zero, then we proceed to the print_gcd function; else, repeat the loop by jumping to the loop_gcd function.

$$\text{GCD}(A,B) = \text{GCD}(B, A\%B)$$

main2_gcd function:

This function is just for swapping \$t0 and \$t1 using a third temporary register \$t3. Then we jump to loop1_gcd.

loop1_gcd function:

This is exactly the same as the loop_gcd function, except that we jump to the loop1_gcd function to repeat the loop.

print_gcd function:

We simply print the prompt "GCD(A, B) = " then print the result stored in register \$t1. After printing the newline, we jump to the end function to exit the calculator.

4.2.6 LCM

lcm function:

The lcm function is the starting point for calculating the LCM of A to base B, denoted as LCM(A, B). We start by storing values of A and B in registers *t4* and *t5* for further use in the final expression of LCM. Then we check if *t0* is greater than *t1*, then we jump to the main2_lcm function for swapping; else, we proceed to loop_lcm.

$$\text{LCM}(A,B) = (A*B)/\text{GCD}(A,B)$$

loop_lcm function:

This function is a loop in which \$t1 is divided by \$t0, and the quotient is stored in \$lo and modulo in \$hi. Then \$t1 is assigned to a value equal to \$t0. And \$t0 is assigned the modulo value of the \$hi. Now we check if \$t0 is equal to zero, then we proceed to the convert_lcm function; else, repeat the loop by jumping to the loop_lcm function.

$$\text{GCD}(A,B) = \text{GCD}(B, A)$$

main2_lcm function:

This function is just for swapping \$t0 and \$t1 using a third temporary register \$t3. Then we jump to loop1_lcm.

`loop1_lcm` function:

This is exactly the same as the `loop_lcm` function, except that we jump to the `loop1_gcd` function to repeat the loop.

`convert_lcm` function:

We store the multiplication result of `$t4` and `$t5` and store it in register `$t6`. Then we divide `$t6` by GCD of `$t4` and `$t5` stored in register `$t1` and store the quotient value in `$t0`.

`print_lcm` function:

We simply print the prompt "LCM(A, B) = " then print the result stored in register `$t0`. After printing the newline, we jump to the end function to exit the calculator

4.2.7 POWER

`pow` function:

The `pow` function is the starting point for calculating the result when one positive integer (a) is raised to another (b). The result is calculated modulo 4999, which is a prime number. This is done to avoid overflow, as exponentiation easily results in large numbers. In the `pow` function, we start by initializing the result to be the same as the input a, which is a raised to 1. As one is the minimum input for b, the minimum result is a itself. This is stored in register `$t2`. Next, we store the value 4999 in the register `$t3`, to use in the modulo function. We then jump into the `pow_mod_a` function

`pow_mod_a` function:

This function takes the input a and calculates a mod 4999. To prevent overflows, we are always keeping our numbers below 4999. We are making use of the following rule

$$ab \bmod n = ((a \bmod n) * (b \bmod n)) \bmod n$$

In the context of our function for exponentiation, this simplifies to

$$(a^{t-1} * a) \bmod n = (((a^{t-1}) \bmod n) * (a \bmod n)) \bmod n$$

So we have to calculate modulo 4999 of the current result and the input number when repeatedly multiplying to get a raised to b. The `pow_mod_a` function does the latter of these. It calculates the modulo 4999 of the input. We do this by repeated subtraction. We keep subtracting 4999 from the input as long as it is greater than 4999. This finally results in a mod 4999. The result is stored in `$t0`.

This needs to be done only once. Once we have the result, we can move on to raising this number to the power of b, which is handled by the `pow_loop` function.

`pow_loop` function:

In this function, we raise the input (modulo 4999) to the power of b, by repeatedly multiplying it with itself. We keep track of the current power decrementing the value of b stored in \$t1. When the value in this register equals 1, we have raised the input to the power of b. The intermediate and final results are stored in the register \$t2. After each iteration, we must take modulo 4999 of the result, as discussed above. For this, we make use of the `pow_mod_ab` function. If the result is already less than 4999, we simply proceed to the next iteration. When the raising is done, we jump to the `pow_exitloop` function to print the results.

`pow_mod_ab` function:

This function uses the same logic as the `pow_mod_a` function, to calculate modulo 4999 of the intermediate result of the raising loop, using repeated subtraction of 4999 from the value stored in 4999. Once the result is less than 4999, we jump back to the raising loop (`pow_loop`).

`pow_exitloop`:

The final result of $(ab \bmod 4999)$ is available in the register \$t2. We simply print the text “ $A^B =$ “ and the final result, after which we jump to the end function to exit the calculator.

4.2.8 ROOT

`root` function

This function is the starting point for the calculation of $A^{1/B}$, or the Bth root of A. Since we need only the integer answer, we do not have to calculate a very exact value of the root using methods such as the Newton Raphson method. Instead, we can simply increment an integer (guess) starting from 1 and see when guess raised to the power of b goes above a. The integer guess-1 will then be the answer to our problem. In the `root` function we start by storing the guess in the register \$t2, initialized to 1. We also store a copy of b in \$t3 and a copy of guess in \$t4, for using in `raise_to_b`, to which we jump to next. We will use this to count down to 1 when raising the guesses to the power of b, using repeated multiplication. For this we initialize \$t4 to store guess raised to one, and jump to the `raise_to_b` function.

`raise_to_b` function:

This function raises the value stored in \$t4 to the power of b. We have a copy of b in \$t3, which we will decrement to one to keep track of the no of iterations and thereby the current power of the intermediate results of the loop. In each iteration, we multiply the current value in \$t4 and store the value in the same register. When the value in \$t3 becomes one, raising is done and we jump to `raise_done`.

raise_done function:

In this function, we compare the calculated value, guess raised to b with the input a. if input is larger, we increment guess and repeat the same process, for which we jump back to raise_to_b. Before this, we reset the values stored in \$t3 and \$t4 to b and the incremented guess respectively.

If the calculated value is larger than the input, this means that we have already passed the exact value of $A(1/B)$. Since our guesses are integers, we know that the exact answer has to be in [guess-1, guess), the floor of which is always guess-1. So we have found our answer. We can jump to the found function to output the results.

found function:

This function simply prints the decorator text " $A(1/B) =$ " and the final result stored in \$t2. After this, we jump to the end function to exit the calculator.

4.2.9 Members

Team Y20: Hisham Hadi T, Satyender Kumar Yadav, Ramtej Penta

4.3 Build Simple Calculator using MIPS

4.3.1 Problem Statement:

Initially, take a signed integer as input, then alternatively take a character and an integer as input. The characters can be $+/-/*\%$

Until you get $;$ you'll use do the "character" operation between your current answer and the current input. Once you get $;$ output the answer.

Say the expression is-

```
5
+
9
*
2
/
3
*
-6
=
```

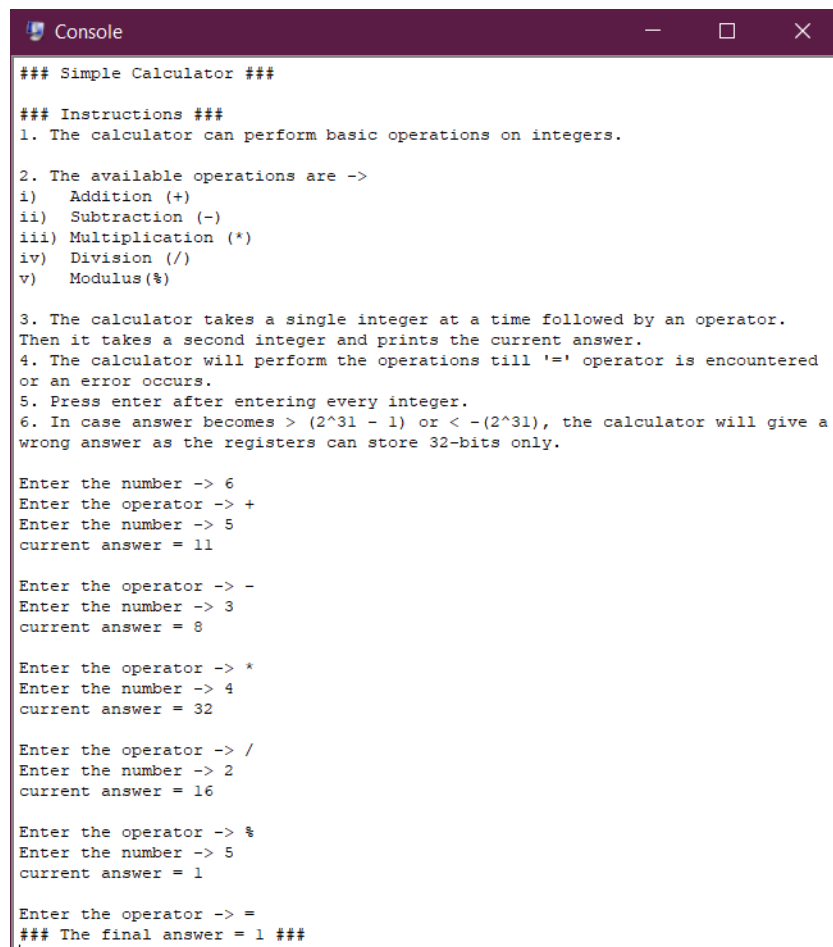
Then your current answer will be: 5 then 14 then 28 then 9 then -54

4.3.2 Introduction

We have implemented a simple calculator using MIPS assembly language. The calculator can perform addition, subtraction, multiplication, division and modulus. It will take a single input at a time, and output the answer at each step, hence evaluating the expression from left to right. It will go on taking instructions till the '=' operator is encountered. Once '=' operator occurs, the calculator prints the final answer.

4.3.3 Example

Here is an example involving all the operations -



```
### Simple Calculator ###

### Instructions ###
1. The calculator can perform basic operations on integers.

2. The available operations are ->
i) Addition (+)
ii) Subtraction (-)
iii) Multiplication (*)
iv) Division (/)
v) Modulus(%)

3. The calculator takes a single integer at a time followed by an operator.
Then it takes a second integer and prints the current answer.
4. The calculator will perform the operations till '=' operator is encountered
or an error occurs.
5. Press enter after entering every integer.
6. In case answer becomes > (2^31 - 1) or < -(2^31), the calculator will give a
wrong answer as the registers can store 32-bits only.

Enter the number -> 6
Enter the operator -> +
Enter the number -> 5
current answer = 11

Enter the operator -> -
Enter the number -> 3
current answer = 8

Enter the operator -> *
Enter the number -> 4
current answer = 32

Enter the operator -> /
Enter the number -> 2
current answer = 16

Enter the operator -> %
Enter the number -> 5
current answer = 1

Enter the operator -> =
### The final answer = 1 ###
```

Figure 1: Example: $6 + 5 - 3 * 4 / 2 \% 5$

In the above example the calculator evaluates the expression $6+5-3*4/2\%5$

from left to right and then prints the final answer i.e. 1.

4.3.4 Instructions

Here are some instructions for using the calculator -

1. The calculator can perform basic operations on integers.
2. The available operations are addition (+), subtraction (-), multiplication (*), division (/) and modulus (%).
3. The calculator takes a single integer at a time followed by an operator. Then it takes a second integer and prints the current answer.
4. The calculator will perform the operations till '=' operator is encountered or an error occurs (like division by 0).
5. Press enter after entering every integer.
6. In case answer becomes $> (2^{31} - 1)$ or $< -(2^{31})$, the calculator will give a wrong answer as the registers can store 32-bits only.

4.3.5 Members

Surviving with Google

Suvrat Pal

Madhur Basnal

P. Lokesh Nayak

4.4 Build Date Converter Using MIPS

4.4.1 Introduction and Problem Statement

The final phase of the project required us to implement a Date Converter using MIPS Assembly Language.

It has the following two functions

1. To convert the date into a required format(eg - DD/MM/YYYY)
2. To check what was the weekday on that date

The program takes 3 inputs initially - Date, Month and Year. Then you have to choose the required format and then it asks you if you want to know the weekday or not. There is also a restart option at the end so that you can add another date.

4.4.2 Algorithm and Working

First the programme has been coded such that the interface asks the user to input the date, the month and the year of their choice in numbers. Then further it would ask the user the format in which they want the date printed (out of the 5 possible formats). The code till here uses `li` (load immediate), `la` (load address) and `move` commands.

Inputting the format number will trigger a basic jump operation, which is the MIPS assembly word for an (if-else) block. All the 5 formats have been hard coded, and -1 is consecutively added to the register storing the format number and after each addition its equality with zero is checked to decide which format statement the code jumps to.

Next the `$t3` is made to store the value 1,2,3 and so on successively and checked at each point to see when it will be equal to the register storing the month to jump to the correct month block to store the month in words.

Now to calculate the day (as in monday,tuesday etc.) first the difference between the current date and 1st January, 1970 is calculated. For that, we are checking if `current year == 1970`. If its true we then look at the month and take care of the number of days in each month and add the number of days.

For example if the date is 08/07/1970. The number of days is $31 + 28 + 31 + 30 + 31 + 30 + 8 = 189$

The days of each month is hardcoded.

Now to calculate the date we have to find the modulo 7 of the number of days.

$189 \equiv 07$ Thus the day will be a **Monday**.

If the current year is different than 1970, could be greater or smaller, the algorithm additionally calculates the difference in days due to the difference in years. For this additional care needs to be taken to calculate the 365/366 days for a normal year/leap year. . For that we iterate over each year until the difference between the year we are currently iterating on and the given year is 0. And for each year that we have iterated on we check the conditions of a leap year. A year is a leap year if

$$\begin{aligned} \text{year} &\equiv 04 \\ \text{year} &\equiv 0100 \\ \text{year} &\equiv 0400 \end{aligned}$$

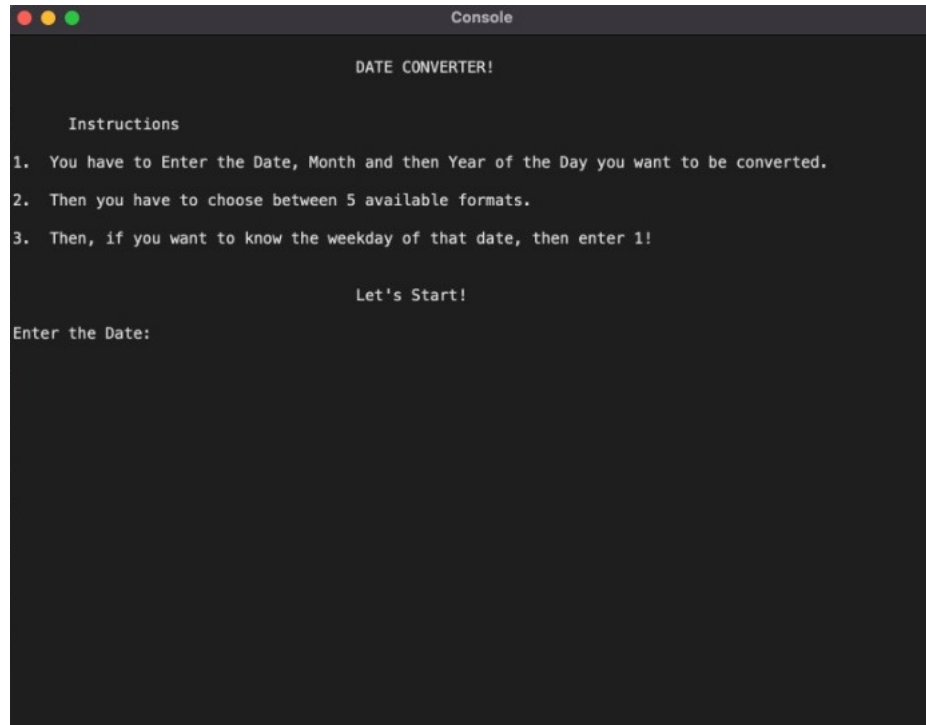
or if

$$\begin{aligned} \text{year} &\equiv 04 \\ \text{year} &\not\equiv 0100 \end{aligned}$$

The rest of the process of calculating the difference in days and then further calculation of which day it will be on that date is the same as the previous example.

Also at the end pressing 1 will jump to the restart statement and we can start all over again

4.4.3 Interface Screenshot



Initial interface screen.

```
Console

DATE CONVERTER!

Instructions

1. You have to Enter the Date, Month and then Year of the Day you want to be converted.
2. Then you have to choose between 5 available formats.
3. Then, if you want to know the weekday of that date, then enter 1!

Let's Start!

Enter the Date: 1
Enter the Month: 1
Enter the Year: 2000
Please enter a number corresponding to the format you want your input in(like 1 for first format, 2
for second, etc.)
1: DD/MM/YYYY
2: MM/DD/YYYY
3: DD/MM/'YY
4: MM/DD/'YY
5: Date(in numbers) Month(in words) Year(in numbers)
5
1 January 2000
Do you want to know what day was on that date, then press 1 else 0: 1
Saturday
Enter 1 to restart else enter 0:
```

Interface for date 01/01/2000

4.4.4 Members

Rishi, Abir, Dhruv, Vishant

5 MENTORS MENTEES

MENTORS

Aditya Tanwar

Akhil Agrawal

Documentation

Lavesh Gupta

Ujjawal Dubey

Anuj Singhal

Rajat Gattani

Presentation

Dhruv

Rishi Poonia

MENTEES

Team: qtsimp

Anuj

Ujjawal Dubey

Lavesh Gupta

Rajat Gattani

Aayush Agrawal

Team: VRAD

Dhruv

Rishi Poonia

Vishant Bhadana

Abhir Rajbongshi

Team: Y00010100

Ramtej Penta

Hisham Hadi T

Satender Kumar Yadav

Team: Surviving with google

Madhur Bansal

Survat Pal

Lokesh Nayak