Spring 2018

SAN JOSÉ STATE UNIVERSITY

Department of Electrical Engineering

EE-275- ADVANCED COMPUTER ARCHITECTURE

MINI-PROJECT REPORT # 2

MIPS ISA DOT PRODUCT PROGRAMMING WITH "FORWARD CHAINING"

Submitted to –

Prof. Chang Choo

Director of AI FPGA/DSP Hardware Laboratory

Submitted by –

Shivang Singh, MSEE

010939851

-- April 16th, 2018 –

TABLE OF CONTENTS

ABSTRACT

The main objective of this project is the learn the designing and implementation of a program which utilizes MIPS ISA along with Dot Product program which also uses Forward Chaining. So, starting with the abstract we can try to understand how the process works in MIPS ISA program when we use the feature of forward chaining into it.

Present era of SOC's contains analog, digital and mixed signal components housing all on the same chip. In such complex environment, a processor plays a vital and vivid role. As we can notice that the computer architecture industry is shrinking at a rapid rate to sub-micrometer technology node, that leads to a huge scope and space for undesirable hazards in processors [1]. And these hazards are extremely dangerous and can ultimately lead to disturbances in area, power and timing which can deviate requirements from the standard desired qualities. Among multiple types of RISC CPU architecture, MIPS is one of the most successful microprocessor and we are focused on improving its performance in this project through Verilog code by using *Dot Product* and *Forward Chaining* mechanism.

A Reduced Instruction Set Computer (RISC) is a microprocessor that is designed to perform a small set of instructions operating at higher speed. The project deals with the concepts of computer architecture, understanding the flow of data and control, enhancing the Verilog coding skills, strong understanding of digital logic and using this to design a functional architecture. It incorporates terminologies, techniques and skills gained in the class. Also, to deal the hazard issues, Forward Chains and Hazard detection units are included.

INTRODUCTION

In the very broad field of computer architecture, when several companies have occupied the market, they use one similar thing for MIPS architecture. It is one of the most commonly and most popular technology, being employed in today's technological world.

MIPS stands for Microprocessor without Interlocked Pipeline Stages and is a reduced instruction set computer (RISC) instruction set. It was developed 33 years ago in the year 1985, by MIPS Technology, a company which was known as MIPS Computer Systems earlier [2]. Nowadays there are multiple versions of MIPS, which includes MIPS I, II, III, IV and V; which also has several sub-divisions within inside it.

As stated above, MIPS processors are widely used in embedded systems, which includes residential gateways and routers. Earlier MIPS architecture was solely designed for general computational purposes.

Discussing about the architecture of MIPS ISA set, it currently supports up to 4 coprocessors. In its terminology, CP0 is called the System Control Coprocessor, CP1 is an optional Floating point unit (FPU), CP2 and CP3 are optional implementation defined coprocessors. A very common example utilizing MIPS architecture and its components are the PlayStation video game console, in which they are used to accelerate the processing of 3-Dimentional computer graphics, for obtaining a better and good output along with better graphic conditions of the 3D images and scenarios.

MIPS microprocessor with RISC architecture have been used widely in the networking equipment, communication equipment and even in entertainment industry and other fields. Cisco routers, IBM network printer, 4K,5K,8K and 9K series of HP laser printer and Sony's PlayStations and the list goes on, where the MIPS microprocessor is utilized successfully to enhance the performance and other desirable tasks.

Also along the time lane, the MIPS architecture has been improved greatly, to make it more adapting and more great output oriented with context of industrial usage; ranging from earlier MIPS16/ MIPS32 architecture to today's MIPS 64 architecture.

HISTORY

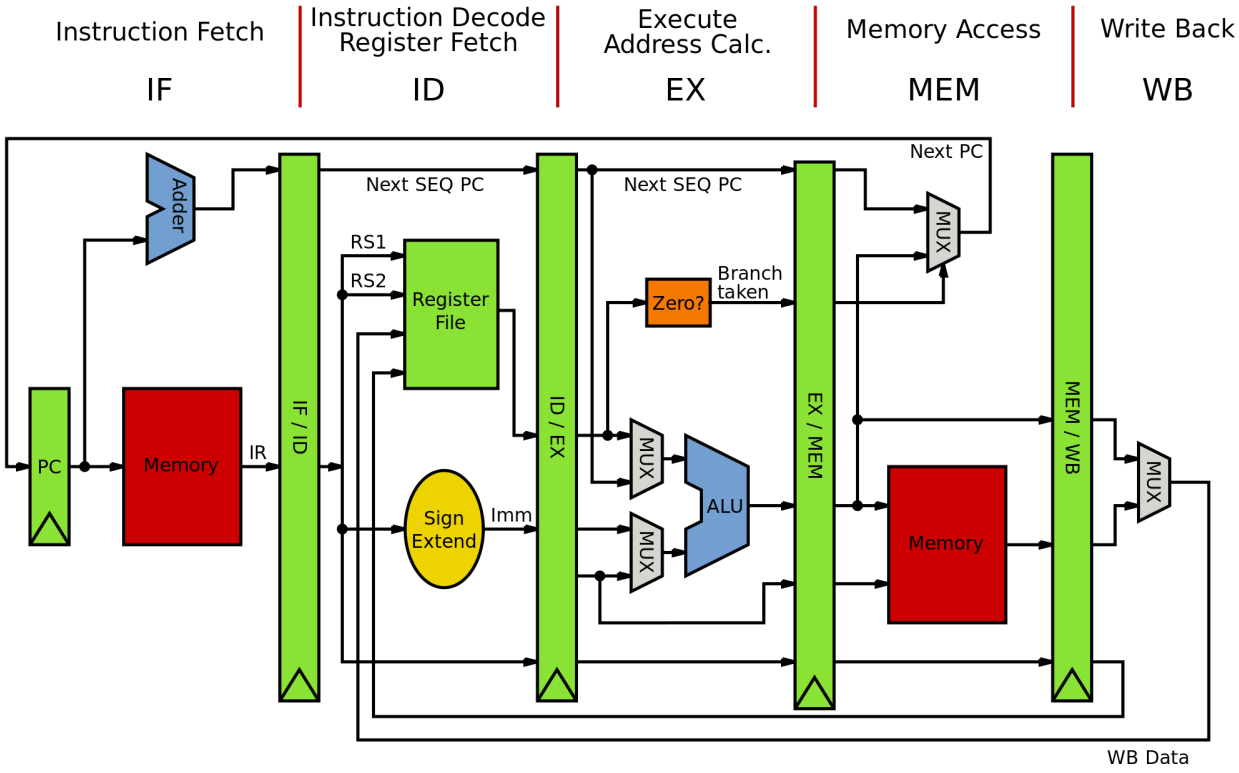
The history of MIPS architecture ranges backs to 33 years in the year 1985. It was developed by a company known as MIPS Technology, which was earlier known as MIPS Computer Systems. The very first version of the MIPS architecture was designed for company's initial R2000 microprocessor. They both were introduced in the same year. Later when MIPS II was introduced, MIPS was renamed to MIPS I, to make a differentiation between the version number and sequence.

It accepts three different types of instruction formats: R, I and J. Here every instruction starts with a 6-bit opcode. Along with the opcode, they also specify the three registers; which includes a shift amount field; and a function field; I-type instructions gives two registers and a 16-bit immediate value and J-type instructions follow an opcode which has a 26-bit jump target. All these three formats can be illustrated in a tabular format as below-

Type of Instruction	**Format (in Bits)**					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

DESIGN

The designing of the MIPS data path is stated as below; and the 5-staged pipelined MIPS data path is demonstrated below; along with all the steps are stated below [3]–



There are 5 stages in this RISC architecture and are stated one by one below –

1. Instruction Fetch (IF) –

In this stage PC is sent to instruction memory and the current instruction is fetched from memory as well update the PC to next in sequence by adding 4 to the PC (PC = PC+4).

2. Instruction Decode (ID) -

Instruction Decode is the second stage of MIPS pipeline. In this stage, the instruction gets decoded and the registers are read as specified in instruction. If there is a branch

instruction, the registers are compared as they are read. Moreover, Sign extends the offset field if it is needed. Compute the possible branch target address. Decoding can be done in parallel with reading the registers since the register specifiers at a fixed location; this is called as 'fixed field decoding'

3. Execute (EX) -

In this stage, ALU operations based on the instruction type. In terms of memory instructions, it adds base address and offset to acquire effective address. For register –register operations, as per the ALU – opcode it performs addition, subtraction as it is needed. It also performs operation for register –immediate ALU instructions.

4. Memory access (MEM) -

In this stage, load and store instructions are being performed. If it is a load instruction, then it reads an effective address from the memory and in the case of store instruction it writes the data from register in to memory.

5. Write Back (WB) -

This is the last stage and it performs register – register ALU instruction or LOAD instruction to write the result in to register file (at ID stage), to check whether it comes through load instruction or from ALU when it is a case of ALU instruction [4].

The table for instruction opcodes and their working and functions is given as below-

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

In case of Forwarding, the data hazard is detected in the following cases –

The data hazard is detected when-

1a.) EX/MEM.RegisterRd = ID/EX.RegisterRs

1b.) EX/MEM.RegisterRd = ID/EX.RegisterRt

2a.) MEM/WB.RegisterRd = ID/EX.RegisterRs

2b.) MEM/WB.RegisterRd = ID/EX.RegisterRt

; where Rs and Rd are Source, destination registers.

The conditions for detecting hazards and the control signals to resolve them:

1. *EX hazard:*

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

2. *MEM hazard:*

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01[1]

SIMULATION (Theory of Operation)

The simulation is done successfully with the proof of completion of the task shown by the following simulation waveforms. The code is written here, alongside the test cases which I tried in the simulation in VHDL, one by one.

The **SIMULATION RESULTS** for **DOT PRODUCT** is performed and the code is stated below. The Dot Product is done by the function:

Dot Product = $\Sigma A\,(i) * (i)$;

where A(i) and B(i) are Signed and Unsigned Numbers respectively.

The program is given as below –

```
movia r2, AVECTOR          /* Register r2 is a pointer to vector A */
movia r3, BVECTOR          /* Register r3 is a pointer to vector B */
movia r4, N
ldw r4, 0(r4)              /* Register r4 is used as the counter for loop iterations */
add r5, r0, r0             /* Register r5 is used to accumulate the product */
LOOP: ldw r6, 0(r2)        /* Load the next element of vector A */
ldw r7, 0(r3)              /* Load the next element of vector B */
mul r8, r6, r7             /* Compute the product of next pair of elements */
add r5, r5, r8             /* Add to the sum */
addi r2, r2, 4             /* Increment the pointer to vector A *////
addi r3, r3, 4             /* Increment the pointer to vector B */
subi r4, r4, 1             /* Decrement the counter */
```

```
bgt r4, r0, LOOP                /* Loop again if not finished */
stw r5, DOT_PRODUCT(r0)         /* Store the result in memory */
STOP: br STOP
```

The step by step pipelined execution is explained in the figure below.

- Pipelined execution of first instruction- ori r0, r2, AVECTOR - 32'h00801914
- The first three "ori" instructions are executed as shown in the figure 4. The registers 2, 3 and 4 are pointing to memory locations 100,200 and 50 respectively. i.e. R2 contains **100**, R3 contains **200**, and R4 contains **50**
- The data stored in the memory locations 100, 200 and 50 for Vector-A, Vector-B and N are loaded on to registers R6, R7 and R9 respectively. i.e. R6 contains **5**, R2 contains **2** and R9 contains **5**.
- The data loaded from memory is multiplied. But since the read register of multiply instruction is dependent on write register of previous load instruction, it causes data hazards (RAW). To resolve this hazard, we need to apply **forwarding**.
- ***Forwarding Unit -*** ldw r6, 0(r2) and ldw r7, 0(r3) instructions are followed by mul r8, r6, r7 instruction. Hence Forwarding methods are applied to directly fetch the value of R6 and R7 from memory and ALU stage respectively. This unit is present in ALU execution stage.
- ***Hazard Detection Unit -*** For the load instruction to fetch data from data memory, one more clock cycle should be stalled for the next dependent instruction to get its dependent register value. i.e the mul instruction should be stalled for one cycle, so that the value register r7 is available in the memory stage. Hazard detection unit is present in Decode stage.

- The multiplied value in r8 is added to r5. Then, the memory location values in register 2 and register 3 are incremented.
- The value of 'n' is decremented.
- The operation is repeated till the value of 'n' is less than zero. i.e. 6 iterations are carried out.
- The branch operation (blt) checks whether the Register 0 value is less than value of 'n'.

Branch Taken -

Branching is detected in pipeline in Instruction decode stage. Hence if branch is taken, the program counter value will be set to the instruction corresponding to 'Loop' value and the next instruction which is fetched before the decision should be flushed out. Therefore, one clock cycle is stalled after every Branch taken condition. The next PC value is **(PC+1) + Branch Address**.

The branch taken condition is explained in the figure 10 below.

- The value of RegDataA is zero and value of RegDataB is 5, RegDataA is **less than** RegDataB. Hence branch is taken. The branch control signal is set 1.
- When the branch control signal is set to one, The PC value changes to PC value corresponding to the value of loop. The present PC value is **'0D'** and it is changed to **'05'**.
- As soon as Branch is taken, flush signal is set to 1. The instruction which is already fetched before branch taken i.e. 32'h01404B15 is flushed out in instruction fetch stage and instead 32'h00000000 (stall) is sent to the corresponding stages.
- After the stall operation, the instruction corresponding to Loop, i.e. ldw r6, 0(r2) is executed.

- The loop is executed until the value of 'n' is less than zero. The multiplication of vector A and B is stored in register 8.
- The result is calculated by accumulating the sums of all the multiplication results as shown in the figure 11 below.

Branch Not Taken -

- When the register0 value i.e. Zero is **not less than** value of **'n'**, the branch is not taken.
- Next instruction is executed i.e. stw r5, DOT_PRODUCT (r0). In this program, the value of '**DOT_PRODUCT** 'is chosen as 300.
- Hence the result is stored in memory location 300 when Memory write signal is high.

PERFORMANCE COMPARISON

The performance of Vector Dot Product is amazing and the number of clock cycles increases linearly with the value of n. The results are as below :

For n=6, # clk cycles→ 68 ;

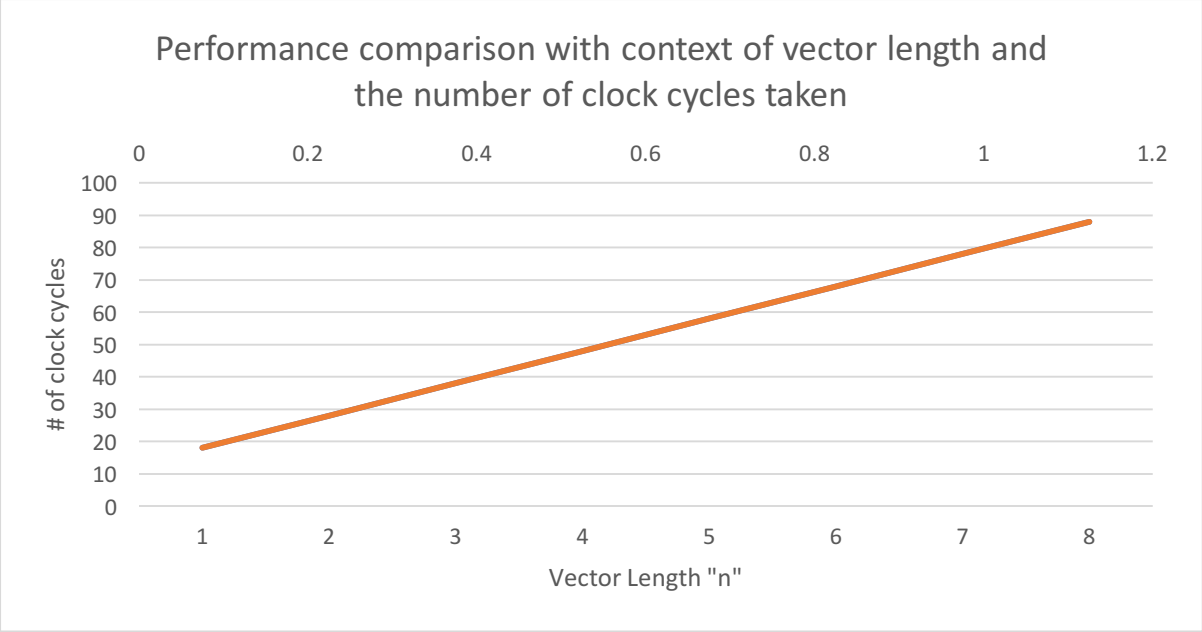
For n=5, # clk cycles→ 58;

For n=4, # clk cycles→ 48;

For n=3, # clk cycles→ 38;

For n=2, # clk cycles→ 28;

For n=1, # clk cycles→ 18.



CONCULSION

In our project, we have **successfully** demonstrated the 5 – stage MIPS ISA which executed dot product program. Several data hazard and control hazards are also successfully resolved. Designing hazard detection unit to overcome the data dependencies was critical task and it was implemented successfully. The project involved wide variety of logics to consider during the design. The performance evaluation results showed that the number of clock cycles taken to execute 'Dot Product' program is linearly proportional to the value of n. Forwarding and Hazard detection units not only helped to solve data dependency problems, but also improved the performance of the processor; i.e. number of clock cycles to execute any program will be less than the execution without Forwarding and Hazard detection units.

We have successfully achieved the goals for our project by reducing the time spent on the process of addition of two inputs. Also, we have successfully reduced the clock cycles during which the second input is considered for the processing. This is done with the help of pipelining and we have understood the concept of pipelining and implemented it in our project.

REFERENCES

1. https://ieeexplore.ieee.org/document/5228236/

2. https://en.wikipedia.org/wiki/MIPS_architecture

3.https://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/MIPS_Architecture_%28Pipelined%29.svg/2000px-MIPS_Architecture_%28Pipelined%29.svg.png

4. Computer Organization and Design: The Hardware/Software Interface, Fourth Edition by John L. Hennessy & David A. Patterson

APPENDIX

```
/ 1st Stage of Pipeline: // ************* Fetch Stage*************

module fetch(clk_51,rst_51,mra_i_51,mrd_i_51,PC_in_51,NPC_51,IR_51,PC_sel_51);
input clk_51,rst_51;
input PC_sel_51;                    // used to select new pc line or not
input [31:0] mrd_i_51;              // memory instruction read
input [31:0] PC_in_51;              // PC data initiated
output reg [31:0] NPC_51;           // After execution Next PC will be stored in this reg.
output reg [31:0]mra_i_51;
output [31:0] IR_51;                // Instruction of 32 bit is stored in IR register
reg [31:0] PC;                      //PC will have current instruction memory location.

// Instructions from code memory will be stored in IR re
assign IR_51 = mrd_i_51;

// On reset pc will be have value 0 so the execution will start from first memory location.

  always @(posedge clk_51 or posedge rst_51)
  begin
  if(rst_51)
  begin
    PC = 0;
  end
  else
  begin
    PC = PC_sel_51 ? PC_in_51 : PC;
    mra_i_51 = rst_51 ? 32'bz :PC;
   // Each of the instruction are at 4 bit difference to each other
 PC = PC + 4;
    NPC_51 = PC_sel_51 ? PC_in_51 : PC;
    //$display("Instruction in the first stage:  %b",IR_51);
  end
  end
endmodule



// 2nd Stage of Pipeline: // *************Decode Stage************* //
This stage will decode all the instructions and
//Get the opcode out which will control whole the execution Module

decode(clk_51,rst_51,IR_51,PC_51,a_51,b_51,NPC_51,PC_sel_51,sign_ext_51,opcode_51,src_reg_51,dest_
re                              g_51,targ_reg_51,r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15,r16,r17,r18,r19,
r20,r21,r22,r23,r24,r25,r26,r27,r28,r29,r30,r31);

input clk_51,rst_51;
input [31:0] IR_51;
input [31:0] PC_51;
output reg [31:0] a_51,b_51;
output reg [31:0] NPC_51;
output reg PC_sel_51;
output reg [31:0] sign_ext_51;
```

```
output [5:0] opcode_51;
output [5:0] src_reg_51;
output [5:0] dest_reg_51;
output [5:0] targ_reg_51;

// These registers are from register bank which will store all execution val
// These 32 registers will require 5 memory locations.
input [31:0] r0,r1,r2,r3,r4,r5,r6,r7,r8,r9;
input [31:0] r10,r11,r12,r13,r14,r15,r16,r17,r18,r19;
input [31:0] r20,r21,r22,r23,r24,r25,r26,r27,r28,r29;
input [31:0] r30,r31;

// 5 wire pair is used to access each of the registers
wire [5:0] acce;

//. Each of the regs are stored with specific parameter which defines it's //memory locations

parameter R0 =  5'b00000;
parameter R1 =  5'b00001;
parameter R2 =  5'b00010;
parameter R3 =  5'b00011;
parameter R4 =  5'b00100;
parameter R5 =  5'b00101;
parameter R6 =  5'b00110;
parameter R7 =  5'b00111;
parameter R8 =  5'b01000;
parameter R9 =  5'b01001;
parameter R10 = 5'b01010;
parameter R11 = 5'b01011;
parameter R12 = 5'b01100;
parameter R13 = 5'b01101;
parameter R14 = 5'b01110;
parameter R15 = 5'b01111;
parameter R16 = 5'b10000;
parameter R17 = 5'b10001;
parameter R18 = 5'b10010;
parameter R19 = 5'b10011;
parameter R20 = 5'b10100;
parameter R21 = 5'b10101;
parameter R22 = 5'b10110;
parameter R23 = 5'b10111;
parameter R24 = 5'b11000;
parameter R25 = 5'b11001;
parameter R26 = 5'b11010;
parameter R27 = 5'b11011;
parameter R28 = 5'b11100;
parameter R29 = 5'b11101;
parameter R30 = 5'b11110;
parameter R31 = 5'b11111;

// Each of the MIPS instructions are have predefined 6-bit hex code //which is written below

parameter ADD  = 6'b110001;// for adding two regs and stores value reg
parameter LDW  = 6'b010111;// load data from memory to reg
parameter MUL  = 6'b100111;// multiply two regs
parameter BLT  = 6'b010110;// it will branch if less than other one
parameter STW  = 6'b010101;// stores data from reg to mem
parameter BR   = 6'b000110;// unconditional branch
parameter ADDI = 6'b000100;// addition of immediate data to reg
```

```
parameter BEQ  = 6'b100110;// Branch if the values are equal
parameter NOPE = 6'b111111;// special instruction which do no operation

assign opcode_51 = rst_51 ? 6'b000000 : IR_51[5:0];
assign src_reg_51 = rst_51 ? 5'b000000 : IR_51[31:27];
assign dest_reg_51 = rst_51 ? 5'b000000 : IR_51[26:22];
assign targ_reg_51 = rst_51 ? 5'b000000 : IR_51[21:17];
assign j = rst_51 ? 5'b000000 : IR_51[16:11];

always @(*)
begin
  if(rst_51)
  begin
    NPC_51 = 0;
    PC_sel_51 = 0;
    sign_ext_51 = 0;
    a_51= 0;
    b_51= 0;
  end

// According to each individual instructions each of the temp regs assigned values.

  else
  begin
  PC_sel_51 = 1'b0;
    case(opcode_51)
      ADD:
      begin

// Value for the source reg stored in src
        case(src_reg_51)
          R0 : a_51 = r0;
          R1 : a_51 = r1;
          R2 : a_51 = r2;
          R3 : a_51 = r3;
          R4 : a_51 = r4;
          R5 : a_51 = r5;
          R6 : a_51 = r6;
          R7 : a_51 = r7;
          R8 : a_51 = r8;
          R9 : a_51 = r9;
          R10 : a_51 = r10;
          R11 : a_51 = r11;
          R12 : a_51 = r12;
          R13 : a_51 = r13;
          R14 : a_51 = r14;
          R15 : a_51 = r15;
          R16 : a_51 = r16;
          R17 : a_51 = r17;
          R18 : a_51 = r18;
          R19 : a_51 = r19;
          R20 : a_51 = r20;
          R21 : a_51 = r21;
          R22 : a_51 = r22;
          R23 : a_51 = r23;
          R24 : a_51 = r24;
          R25 : a_51 = r25;
          R26 : a_51 = r26;
          R27 : a_51 = r27;
```

```
          R28 : a_51 = r28;
          R29 : a_51 = r29;
          R30 : a_51 = r30;
          R31 : a_51 = r31;
        endcase


// Destination which is stored in b

case(dest_reg_51)
          R0 : b_51 = r0;
          R1 : b_51 = r1;
          R2 : b_51 = r2;
          R3 : b_51 = r3;
          R4 : b_51 = r4;
          R5 : b_51 = r5;
          R6 : b_51 = r6;
          R7 : b_51 = r7;
          R8 : b_51 = r8;
          R9 : b_51 = r9;
          R10 : b_51 = r10;
          R11 : b_51 = r11;
          R12 : b_51 = r12;
          R13 : b_51 = r13;
          R14 : b_51 = r14;
          R15 : b_51 = r15;
          R16 : b_51 = r16;
          R17 : b_51 = r17;
          R18 : b_51 = r18;
          R19 : b_51 = r19;
          R20 : b_51 = r20;
          R21 : b_51 = r21;
          R22 : b_51 = r22;
          R23 : b_51 = r23;
          R24 : b_51 = r24;
          R25 : b_51 = r25;
          R26 : b_51 = r26;
          R27 : b_51 = r27;
          R28 : b_51 = r28;
          R29 : b_51 = r29;
          R30 : b_51 = r30;
          R31 : b_51 = r31;
        endcase
      end
      LDW:
      begin
        case(src_reg_51)
          R0 : a_51 = r0;
          R1 : a_51 = r1;
          R2 : a_51 = r2;
          R3 : a_51 = r3;
          R4 : a_51 = r4;
          R5 : a_51 = r5;
          R6 : a_51 = r6;
          R7 : a_51 = r7;
          R8 : a_51 = r8;
          R9 : a_51 = r9;
          R10 : a_51 = r10;
          R11 : a_51 = r11;
```

```
    R12 : a_51 = r12;
    R13 : a_51 = r13;
    R14 : a_51 = r14;
    R15 : a_51 = r15;
    R16 : a_51 = r16;
    R17 : a_51 = r17;
    R18 : a_51 = r18;
    R19 : a_51 = r19;
    R20 : a_51 = r20;
    R21 : a_51 = r21;
    R22 : a_51 = r22;
    R23 : a_51 = r23;
    R24 : a_51 = r24;
    R25 : a_51 = r25;
    R26 : a_51 = r26;
    R27 : a_51 = r27;
    R28 : a_51 = r28;
    R29 : a_51 = r29;
    R30 : a_51 = r30;
    R31 : a_51 = r31;
  endcase
  sign_ext_51 = IR_51[21] ? {16'hFFFF,IR_51[21:6]} : {16'h0000,IR_51[21:6]};
end
MUL:
begin
  case(src_reg_51)
    R0 : a_51 = r0;
    R1 : a_51 = r1;
    R2 : a_51 = r2;
    R3 : a_51 = r3;
    R4 : a_51 = r4;
    R5 : a_51 = r5;
    R6 : a_51 = r6;
    R7 : a_51 = r7;
    R8 : a_51 = r8;
    R9 : a_51 = r9;
    R10 : a_51 = r10;
    R11 : a_51 = r11;
    R12 : a_51 = r12;
    R13 : a_51 = r13;
    R14 : a_51 = r14;
    R15 : a_51 = r15;
    R16 : a_51 = r16;
    R17 : a_51 = r17;
    R18 : a_51 = r18;
    R19 : a_51 = r19;
    R20 : a_51 = r20;
    R21 : a_51 = r21;
    R22 : a_51 = r22;
    R23 : a_51 = r23;
    R24 : a_51 = r24;
    R25 : a_51 = r25;
    R26 : a_51 = r26;
    R27 : a_51 = r27;
    R28 : a_51 = r28;
    R29 : a_51 = r29;
    R30 : a_51 = r30;
    R31 : a_51 = r31;
  endcase
```

```
  case(dest_reg_51)
    R0 : b_51 = r0;
    R1 : b_51 = r1;
    R2 : b_51 = r2;
    R3 : b_51 = r3;
    R4 : b_51 = r4;
    R5 : b_51 = r5;
    R6 : b_51 = r6;
    R7 : b_51 = r7;
    R8 : b_51 = r8;
    R9 : b_51 = r9;
    R10 : b_51 = r10;
    R11 : b_51 = r11;
    R12 : b_51 = r12;
    R13 : b_51 = r13;
    R14 : b_51 = r14;
    R15 : b_51 = r15;
    R16 : b_51 = r16;
    R17 : b_51 = r17;
    R18 : b_51 = r18;
    R19 : b_51 = r19;
    R20 : b_51 = r20;
    R21 : b_51 = r21;
    R22 : b_51 = r22;
    R23 : b_51 = r23;
    R24 : b_51 = r24;
    R25 : b_51 = r25;
    R26 : b_51 = r26;
    R27 : b_51 = r27;
    R28 : b_51 = r28;
    R29 : b_51 = r29;
    R30 : b_51 = r30;
    R31 : b_51 = r31;
  endcase
end
BLT:
begin
  case(src_reg_51)
    R0 : a_51 = r0;
    R1 : a_51 = r1;
    R2 : a_51 = r2;
    R3 : a_51 = r3;
    R4 : a_51 = r4;
    R5 : a_51 = r5;
    R6 : a_51 = r6;
    R7 : a_51 = r7;
    R8 : a_51 = r8;
    R9 : a_51 = r9;
    R10 : a_51 = r10;
    R11 : a_51 = r11;
    R12 : a_51 = r12;
    R13 : a_51 = r13;
    R14 : a_51 = r14;
    R15 : a_51 = r15;
    R16 : a_51 = r16;
    R17 : a_51 = r17;
    R18 : a_51 = r18;
    R19 : a_51 = r19;
    R20 : a_51 = r20;
```

```
    R21 : a_51 = r21;
    R22 : a_51 = r22;
    R23 : a_51 = r23;
    R24 : a_51 = r24;
    R25 : a_51 = r25;
    R26 : a_51 = r26;
    R27 : a_51 = r27;
    R28 : a_51 = r28;
    R29 : a_51 = r29;
    R30 : a_51 = r30;
    R31 : a_51 = r31;
  endcase
  case(dest_reg_51)
    R0 : b_51 = r0;
    R1 : b_51 = r1;
    R2 : b_51 = r2;
    R3 : b_51 = r3;
    R4 : b_51 = r4;
    R5 : b_51 = r5;
    R6 : b_51 = r6;
    R7 : b_51 = r7;
    R8 : b_51 = r8;
    R9 : b_51 = r9;
    R10 : b_51 = r10;
    R11 : b_51 = r11;
    R12 : b_51 = r12;
    R13 : b_51 = r13;
    R14 : b_51 = r14;
    R15 : b_51 = r15;
    R16 : b_51 = r16;
    R17 : b_51 = r17;
    R18 : b_51 = r18;
    R19 : b_51 = r19;
    R20 : b_51 = r20;
    R21 : b_51 = r21;
    R22 : b_51 = r22;
    R23 : b_51 = r23;
    R24 : b_51 = r24;
    R25 : b_51 = r25;
    R26 : b_51 = r26;
    R27 : b_51 = r27;
    R28 : b_51 = r28;
    R29 : b_51 = r29;
    R30 : b_51 = r30;
    R31 : b_51 = r31;
  endcase
  sign_ext_51 = IR_51[21] ? {16'hFFFF,IR_51[21:6]} : {16'h0000,IR_51[21:6]};
  NPC_51 = (a_51<b_51) ? (PC_51 - sign_ext_51) : PC_51;
  PC_sel_51 = 1'b1;
end
STW:
begin
  case(src_reg_51)
    R0 : a_51 = r0;
    R1 : a_51 = r1;
    R2 : a_51 = r2;
    R3 : a_51 = r3;
    R4 : a_51 = r4;
    R5 : a_51 = r5;
```

```
  R6 : a_51 = r6;
  R7 : a_51 = r7;
  R8 : a_51 = r8;
  R9 : a_51 = r9;
  R10 : a_51 = r10;
  R11 : a_51 = r11;
  R12 : a_51 = r12;
  R13 : a_51 = r13;
  R14 : a_51 = r14;
  R15 : a_51 = r15;
  R16 : a_51 = r16;
  R17 : a_51 = r17;
  R18 : a_51 = r18;
  R19 : a_51 = r19;
  R20 : a_51 = r20;
  R21 : a_51 = r21;
  R22 : a_51 = r22;
  R23 : a_51 = r23;
  R24 : a_51 = r24;
  R25 : a_51 = r25;
  R26 : a_51 = r26;
  R27 : a_51 = r27;
  R28 : a_51 = r28;
  R29 : a_51 = r29;
  R30 : a_51 = r30;
  R31 : a_51 = r31;
endcase
case(dest_reg_51)
  R0 : b_51 = r0;
  R1 : b_51 = r1;
  R2 : b_51 = r2;
  R3 : b_51 = r3;
  R4 : b_51 = r4;
  R5 : b_51 = r5;
  R6 : b_51 = r6;
  R7 : b_51 = r7;
  R8 : b_51 = r8;
  R9 : b_51 = r9;
  R10 : b_51 = r10;
  R11 : b_51 = r11;
  R12 : b_51 = r12;
  R13 : b_51 = r13;
  R14 : b_51 = r14;
  R15 : b_51 = r15;
  R16 : b_51 = r16;
  R17 : b_51 = r17;
  R18 : b_51 = r18;
  R19 : b_51 = r19;
  R20 : b_51 = r20;
  R21 : b_51 = r21;
  R22 : b_51 = r22;
  R23 : b_51 = r23;
  R24 : b_51 = r24;
  R25 : b_51 = r25;
  R26 : b_51 = r26;
  R27 : b_51 = r27;
  R28 : b_51 = r28;
  R29 : b_51 = r29;
  R30 : b_51 = r30;
```

```
    R31 : b_51 = r31;
  endcase
  sign_ext_51 = IR_51[21] ? {16'hFFFF,IR_51[21:6]} : {16'h0000,IR_51[21:6]};
end
BR:
begin
  sign_ext_51 = IR_51[21] ? {16'hFFFF,IR_51[21:6]} : {16'h0000,IR_51[21:6]};
  NPC_51 = PC_51 - sign_ext_51;
  PC_sel_51 = 1'b1;
end
ADDI:
begin
  case(src_reg_51)
    R0 : a_51 = r0;
    R1 : a_51 = r1;
    R2 : a_51 = r2;
    R3 : a_51 = r3;
    R4 : a_51 = r4;
    R5 : a_51 = r5;
    R6 : a_51 = r6;
    R7 : a_51 = r7;
    R8 : a_51 = r8;
    R9 : a_51 = r9;
    R10 : a_51 = r10;
    R11 : a_51 = r11;
    R12 : a_51 = r12;
    R13 : a_51 = r13;
    R14 : a_51 = r14;
    R15 : a_51 = r15;
    R16 : a_51 = r16;
    R17 : a_51 = r17;
    R18 : a_51 = r18;
    R19 : a_51 = r19;
    R20 : a_51 = r20;
    R21 : a_51 = r21;
    R22 : a_51 = r22;
    R23 : a_51 = r23;
    R24 : a_51 = r24;
    R25 : a_51 = r25;
    R26 : a_51 = r26;
    R27 : a_51 = r27;
    R28 : a_51 = r28;
    R29 : a_51 = r29;
    R30 : a_51 = r30;
    R31 : a_51 = r31;
  endcase
  sign_ext_51 = IR_51[21] ? {16'hFFFF,IR_51[21:6]} : {16'h0000,IR_51[21:6]};
end
BEQ :
begin
  case(src_reg_51)
    R0 : a_51 = r0;
    R1 : a_51 = r1;
    R2 : a_51 = r2;
    R3 : a_51 = r3;
    R4 : a_51 = r4;
    R5 : a_51 = r5;
    R6 : a_51 = r6;
    R7 : a_51 = r7;
```

```
  R8 : a_51 = r8;
  R9 : a_51 = r9;
  R10 : a_51 = r10;
  R11 : a_51 = r11;
  R12 : a_51 = r12;
  R13 : a_51 = r13;
  R14 : a_51 = r14;
  R15 : a_51 = r15;
  R16 : a_51 = r16;
  R17 : a_51 = r17;
  R18 : a_51 = r18;
  R19 : a_51 = r19;
  R20 : a_51 = r20;
  R21 : a_51 = r21;
  R22 : a_51 = r22;
  R23 : a_51 = r23;
  R24 : a_51 = r24;
  R25 : a_51 = r25;
  R26 : a_51 = r26;
  R27 : a_51 = r27;
  R28 : a_51 = r28;
  R29 : a_51 = r29;
  R30 : a_51 = r30;
  R31 : a_51 = r31;
endcase
case(dest_reg_51)
  R0 : b_51 = r0;
  R1 : b_51 = r1;
  R2 : b_51 = r2;
  R3 : b_51 = r3;
  R4 : b_51 = r4;
  R5 : b_51 = r5;
  R6 : b_51 = r6;
  R7 : b_51 = r7;
  R8 : b_51 = r8;
  R9 : b_51 = r9;
  R10 : b_51 = r10;
  R11 : b_51 = r11;
  R12 : b_51 = r12;
  R13 : b_51 = r13;
  R14 : b_51 = r14;
  R15 : b_51 = r15;
  R16 : b_51 = r16;
  R17 : b_51 = r17;
  R18 : b_51 = r18;
  R19 : b_51 = r19;
  R20 : b_51 = r20;
  R21 : b_51 = r21;
  R22 : b_51 = r22;
  R23 : b_51 = r23;
  R24 : b_51 = r24;
  R25 : b_51 = r25;
  R26 : b_51 = r26;
  R27 : b_51 = r27;
  R28 : b_51 = r28;
  R29 : b_51 = r29;
  R30 : b_51 = r30;
  R31 : b_51 = r31;
endcase
```

```
      sign_ext_51 = IR_51[21] ? {16'hFFFF,IR_51[21:6]} : {16'h0000,IR_51[21:6]};
      NPC_51 = (a_51==b_51) ? (PC_51 - sign_ext_51) : PC_51;
      PC_sel_51 = (a_51==b_51) ? 1'b1 : 1'b0;
     End

// Nope is special instruction which is used to put memory stall cycles
// It will avoid possible hazards

     NOPE:
     begin
     end
   endcase
  end
  end
endmodule

// During this stage acc to instruction execution takes place.



// 3rd Stage of Pipeline: // ************* Execution Stage*************

module
execute(clk_51,rst_51,opcode_51,a_51,b_51,sign_ext_51,alu_out_51,alu_src_51,opcode_out_51,src_reg_51,
dest_reg_51, targ_reg_51,src_reg_out_51,dest_reg_out_51,targ_reg_out_51);

input clk_51,rst_51;
input [5:0] opcode_51;
input [5:0] src_reg_51,dest_reg_51,targ_reg_51;
input [31:0] a_51,b_51,sign_ext_51;
output [5:0] opcode_out_51;
output reg [31:0] alu_out_51;
output [31:0] alu_src_51;
output [5:0] src_reg_out_51,dest_reg_out_51,targ_reg_out_51;
// If reset is found assigning value to 0

assign opcode_out_51      =  rst_51 ? 0 : opcode_51;
assign src_reg_51_out     =  rst_51 ? 0 : src_reg_51;
assign dest_reg_out_51    =  rst_51 ? 0 : dest_reg_51;
assign targ_reg_out_51    =  rst_51 ? 0 : targ_reg_51;
assign alu_src_51         =  rst_51 ? 0 : b_51;

  always @ (*)
  begin
    #0;
    if(rst_51)
    begin
     alu_out_51 = 0;
    end
    else
    begin
      case(opcode_51)

// According to various inst various execution takes place.

        ADD : alu_out_51 = a_51 + b_51;
        LDW : alu_out_51 = a_51 + sign_ext_51;
        MUL : alu_out_51 = a_51 * b_51;
        STW : alu_out_51 = a_51 + sign_ext_51;
```

```
      ADDI: alu_out_51 = a_51 + sign_ext_51;
      SUBI: alu_out_51 = a_51 - sign_ext_51;
      CALL: alu_out_51 = b_51;
     endcase
   end
  end
endmodule


// 4th Stage of Pipeline: // ************* Memory write ************* // During this stage memory will be accessed and data transfer from reg to mem // take place

module
memory_write(clk_51,rst_51,opcode_51,src_reg_51,dest_reg_51,targ_reg_51,alu_out_51,alu_src_51,mem_ou
t1_51,mem
_out2_51,opcode_out_51,src_reg_out_51,dest_reg_out_51,targ_reg_out_51,mwr_d_51,mwd_d_51,mwa_d_51
,mra_d_51,                              mrd_d_51,r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15,r16,r17,r18,r19,
r20,r21,r22,r23,r24,r25,r26,r27,r28,r29,r30,r31);

input clk_51,rst_51;
input [5:0] opcode_51;
input [5:0] src_reg_51,dest_reg_51,targ_reg_51;
input [31:0] alu_out_51,alu_src_51;
input [31:0] mrd_d_51;

output reg [31:0]mem_out1_51;
output [31:0] mem_out2_51;
output [5:0] opcode_out_51;
output [5:0] src_reg_out_51,dest_reg_out_51,targ_reg_out_51;
output reg [31:0] mra_d_51;
output reg [31:0] mwd_d_51,mwa_d_51;
output reg mwr_d_51;
// Each regs are intiated
input [31:0] r0,r1,r2,r3,r4,r5,r6,r7,r8,r9;
// Each regs are intiated
input [31:0] r10,r11,r12,r13,r14,r15,r16,r17,r18,r19;
// Each regs are intiated
input [31:0] r20,r21,r22,r23,r24,r25,r26,r27,r28,r29;
input [31:0] r30,r31;

assign opcode_out_51   =  rst_51 ? 5'b0 : opcode_51;
assign src_reg_out_51  =  rst_51 ? 5'b0 : src_reg_51;
assign dest_reg_out_51 =  rst_51 ? 5'b0 : dest_reg_51;
assign targ_reg_out_51 =  rst_51 ? 5'b0 : targ_reg_51;
assign mem_out2_51     =  rst_51 ? 5'b0 : alu_src_51;

  always @(*)
  begin
  if(rst_51)
  begin
    mem_out1_51 = 0;
  end
  else
  begin
    mwr_d_51 = 1'b0;
    case(opcode_51)
      LDW :
      begin
        mra_d_51 = alu_out_51;
```

```
      mem_out1_51 = mrd_d_51;
    end
    STW:
    begin
      mwd_d_51 = alu_src_51;
      mwa_d_51 = alu_out_51;
      mwr_d_51 = 1'b1;
    end
    default : mem_out1_51 = alu_out_51;
   endcase
  end
  end
endmodule



// During this stage data from the memory or reg will be transferred back to reg file.

// 5th Stage of Pipeline: // ************* Write Back Stage*************

module
rw(clk_51,rst_51,opcode_51,src_reg_51,dest_reg_51,targ_reg_51,mem_out1_51,mem_out2_51,r0,r1,r2,r3,r4,r
5,r6,r7,r8,r 9,r10,r11,r12,r13,r14,r15,r16,r17,r18,r19, r20,r21,r22,r23,r24,r25,r26,r27,r28,r29,r30,r31);

input clk_51,rst_51;
input [5:0] opcode_51;
input [5:0] src_reg_51,dest_reg_51,targ_reg_51;
input [31:0] mem_out1_51,mem_out2_51;
// initiating 32 bit regs
output reg [31:0] r0,r1,r2,r3,r4,r5,r6,r7,r8,r9;
output reg [31:0] r10,r11,r12,r13,r14,r15,r16,r17,r18,r19;
output reg [31:0] r20,r21,r22,r23,r24,r25,r26,r27,r28,r29;
output reg [31:0] r30,r31;
// if reset in set to 1 all the values will be transferred to 0
initial
begin
  r0 = 0;
  r1 = 0;
  r2 = 0;
  r3 = 0;
  r4 = 3;
  r5 = 0;
  r6 = 0;
  r7 = 0;
  r8 = 0;
  r9 = 0;
  r10 = 0;
  r11 = 0;
  r12 = 0;
  r13 = 0;
  r14 = 0;
  r15 = 0;
  r16 = 0;
  r17 = 0;
  r18 = 0;
  r19 = 0;
  r20 = 0;
  r21 = 0;
```

```
  r22 = 0;
  r23 = 0;
  r24 = 0;
  r25 = 0;
  r26 = 0;
  r27 = 0;
  r28 = 0;
  r29 = 32'h00000051;
  r30 = 150;
  r31 = 190;
end
  always @(*)
  begin
    case(opcode_51)
      ADD:
      begin
        case(targ_reg_51)
          R0 : r0 = mem_out1_51;
          R1 : r1 = mem_out1_51;
          R2 : r2 = mem_out1_51;
          R3 : r3 = mem_out1_51;
          R4 : r4 = mem_out1_51;
          R5 : r5 = mem_out1_51;
          R6 : r6 = mem_out1_51;
          R7 : r7 = mem_out1_51;
          R8 : r8 = mem_out1_51;
          R9 : r9 = mem_out1_51;
          R10 : r10 = mem_out1_51;
          R11 : r11 = mem_out1_51;
          R12 : r12 = mem_out1_51;
          R13 : r13 = mem_out1_51;
          R14 : r14 = mem_out1_51;
          R15 : r15 = mem_out1_51;
          R16 : r16 = mem_out1_51;
          R17 : r17 = mem_out1_51;
          R18 : r18 = mem_out1_51;
          R19 : r19 = mem_out1_51;
          R20 : r20 = mem_out1_51;
          R21 : r21 = mem_out1_51;
          R22 : r22 = mem_out1_51;
          R23 : r23 = mem_out1_51;
          R24 : r24 = mem_out1_51;
          R25 : r25 = mem_out1_51;
          R26 : r26 = mem_out1_51;
          R27 : r27 = mem_out1_51;
          R28 : r28 = mem_out1_51;
          R29 : r29 = mem_out1_51;
          R30 : r30 = mem_out1_51;
          R31 : r31 = mem_out1_51;
        endcase
      end
      LDW:
      begin
        case(dest_reg_51)
          R0 : r0 = mem_out1_51;
          R1 : r1 = mem_out1_51;
          R2 : r2 = mem_out1_51;
          R3 : r3 = mem_out1_51;
          R4 : r4 = mem_out1_51;
```

```
    R5 : r5 = mem_out1_51;
    R6 : r6 = mem_out1_51;
    R7 : r7 = mem_out1_51;
    R8 : r8 = mem_out1_51;
    R9 : r9 = mem_out1_51;
    R10 : r10 = mem_out1_51;
    R11 : r11 = mem_out1_51;
    R12 : r12 = mem_out1_51;
    R13 : r13 = mem_out1_51;
    R14 : r14 = mem_out1_51;
    R15 : r15 = mem_out1_51;
    R16 : r16 = mem_out1_51;
    R17 : r17 = mem_out1_51;
    R18 : r18 = mem_out1_51;
    R19 : r19 = mem_out1_51;
    R20 : r20 = mem_out1_51;
    R21 : r21 = mem_out1_51;
    R22 : r22 = mem_out1_51;
    R23 : r23 = mem_out1_51;
    R24 : r24 = mem_out1_51;
    R25 : r25 = mem_out1_51;
    R26 : r26 = mem_out1_51;
    R27 : r27 = mem_out1_51;
    R28 : r28 = mem_out1_51;
    R29 : r29 = mem_out1_51;
    R30 : r30 = mem_out1_51;
    R31 : r31 = mem_out1_51;
  endcase
end
MUL:
begin
  case(targ_reg_51)
    R0 : r0 = mem_out1_51;
    R1 : r1 = mem_out1_51;
    R2 : r2 = mem_out1_51;
    R3 : r3 = mem_out1_51;
    R4 : r4 = mem_out1_51;
    R5 : r5 = mem_out1_51;
    R6 : r6 = mem_out1_51;
    R7 : r7 = mem_out1_51;
    R8 : r8 = mem_out1_51;
    R9 : r9 = mem_out1_51;
    R10 : r10 = mem_out1_51;
    R11 : r11 = mem_out1_51;
    R12 : r12 = mem_out1_51;
    R13 : r13 = mem_out1_51;
    R14 : r14 = mem_out1_51;
    R15 : r15 = mem_out1_51;
    R16 : r16 = mem_out1_51;
    R17 : r17 = mem_out1_51;
    R18 : r18 = mem_out1_51;
    R19 : r19 = mem_out1_51;
    R20 : r20 = mem_out1_51;
    R21 : r21 = mem_out1_51;
    R22 : r22 = mem_out1_51;
    R23 : r23 = mem_out1_51;
    R24 : r24 = mem_out1_51;
    R25 : r25 = mem_out1_51;
    R26 : r26 = mem_out1_51;
```

```
    R27 : r27 = mem_out1_51;
    R28 : r28 = mem_out1_51;
    R29 : r29 = mem_out1_51;
    R30 : r30 = mem_out1_51;
    R31 : r31 = mem_out1_51;
  endcase
end
ADDI:
begin
  case(dest_reg_51)
    R0 : r0 = mem_out1_51;
    R1 : r1 = mem_out1_51;
    R2 : r2 = mem_out1_51;
    R3 : r3 = mem_out1_51;
    R4 : r4 = mem_out1_51;
    R5 : r5 = mem_out1_51;
    R6 : r6 = mem_out1_51;
    R7 : r7 = mem_out1_51;
    R8 : r8 = mem_out1_51;
    R9 : r9 = mem_out1_51;
    R10 : r10 = mem_out1_51;
    R11 : r11 = mem_out1_51;
    R12 : r12 = mem_out1_51;
    R13 : r13 = mem_out1_51;
    R14 : r14 = mem_out1_51;
    R15 : r15 = mem_out1_51;
    R16 : r16 = mem_out1_51;
    R17 : r17 = mem_out1_51;
    R18 : r18 = mem_out1_51;
    R19 : r19 = mem_out1_51;
    R20 : r20 = mem_out1_51;
    R21 : r21 = mem_out1_51;
    R22 : r22 = mem_out1_51;
    R23 : r23 = mem_out1_51;
    R24 : r24 = mem_out1_51;
    R25 : r25 = mem_out1_51;
    R26 : r26 = mem_out1_51;
    R27 : r27 = mem_out1_51;
    R28 : r28 = mem_out1_51;
    R29 : r29 = mem_out1_51;
    R30 : r30 = mem_out1_51;
    R31 : r31 = mem_out1_51;
  endcase
end
CALL :
begin
  r31 = mem_out2_51;
end
SUBI:
begin
  case(dest_reg_51)
    R0 : r0 = mem_out1_51;
    R1 : r1 = mem_out1_51;
    R2 : r2 = mem_out1_51;
    R3 : r3 = mem_out1_51;
    R4 : r4 = mem_out1_51;
    R5 : r5 = mem_out1_51;
    R6 : r6 = mem_out1_51;
    R7 : r7 = mem_out1_51;
```

```
        R8 : r8 = mem_out1_51;
        R9 : r9 = mem_out1_51;
        R10 : r10 = mem_out1_51;
        R11 : r11 = mem_out1_51;
        R12 : r12 = mem_out1_51;
        R13 : r13 = mem_out1_51;
        R14 : r14 = mem_out1_51;
        R15 : r15 = mem_out1_51;
        R16 : r16 = mem_out1_51;
        R17 : r17 = mem_out1_51;
        R18 : r18 = mem_out1_51;
        R19 : r19 = mem_out1_51;
        R20 : r20 = mem_out1_51;
        R21 : r21 = mem_out1_51;
        R22 : r22 = mem_out1_51;
        R23 : r23 = mem_out1_51;
        R24 : r24 = mem_out1_51;
        R25 : r25 = mem_out1_51;
        R26 : r26 = mem_out1_51;
        R27 : r27 = mem_out1_51;
        R28 : r28 = mem_out1_51;
        R29 : r29 = mem_out1_51;
        R30 : r30 = mem_out1_51;
        R31 : r31 = mem_out1_51;
      endcase
    end
  endcase
 end
endmodule

// Combining all these modules together:

`include "fetch.v"
`include "decode.v"
`include "execute.v"
`include "mw.v"
`include "wb.v"

module
proc(clk_51,rst_51,mrd_i_51,mwd_i_51,mra_i_51,mwa_i_51,mwr_i_51,mrd_d_51,mwd_d_51,mra_d_51,mwa_
d_51,mwr_d_51);

input clk_51,rst_51;
input [31:0] mrd_i_51,mrd_d_51;              //// memory access to read data
output [31:0] mwd_i_51,mwd_d_51;             //// memory access to  write data
output [31:0] mra_i_51,mra_d_51;             //// memory access to  read address
output [31:0] mwa_i_51,mwa_d_51;             //// memory access to  write address
output mwr_i_51,mwr_d_51;                    //// memory access to  write enable

// Intiating 32 bit wires
wire [31:0] r0,r1,r2,r3,r4,r5,r6,r7,r8,r9;


// Intiating 32 bit wires
wire [31:0] r10,r11,r12,r13,r14,r15,r16,r17,r18,r19;

// Intiating 32 bit wires
wire [31:0] r20,r21,r22,r23,r24,r25,r26,r27,r28,r29;
```

```
// Intiating 32 bit wires
wire [31:0] r30,r31;

// Transferring data from one to another stages:

wire [31:0] PC_in_fd_51;
wire [31:0] NPC_fd_51;
wire [31:0] IR_fd_51;
wire PC_sel_fd_51;
wire [31:0] sign_ext_de_51;
wire [5:0]  opcode_de_51;
wire [5:0]  src_reg_de_51;
wire [5:0]  dest_reg_de_51;
wire [5:0]  targ_reg_de_51;
wire [31:0] a_de_51;
wire [31:0] b_de_51;
wire [31:0] alu_out_em_51;
wire [31:0] alu_src_em_51;
wire [5:0]  opcode_out_em_51;
wire [5:0]  src_reg_out_em_51;
wire [5:0]  dest_reg_out_em_51;
wire [5:0]  targ_reg_out_em_51;
wire [31:0] mem_out1_mr_51;
wire [31:0] mem_out2_mr_51;
wire [5:0]  opcode_out_mr_51;
wire [5:0]  src_reg_out_mr_51;
wire [5:0]  dest_reg_out_mr_51;
wire [5:0]  targ_reg_out_mr_51;

// Till the clock arrives this regs stores the values till that time

reg [31:0] NPC_fd_f_51;
reg PC_sel_fd_f_51;
reg PC_in_fd_f_51;
reg [31:0] IR_fd_f_51;
reg [31:0] a_de_f_51;
reg [31:0] b_de_f_51;
reg [31:0] sign_ext_de_f_51,a_de_f_51,b_de_f_51;
reg [5:0]  opcode_de_f_51,src_reg_de_f_51,dest_reg_de_f_51,targ_reg_de_f_51;
reg [31:0] alu_out_em_f_51,alu_src_em_f_51;
reg [5:0]  opcode_out_em_f_51,src_reg_out_em_f_51,dest_reg_out_em_f_51,targ_reg_out_em_f_51;
reg [31:0] mem_out1_mr_f_51,mem_out2_mr_f_51,mem_out1_mr_ff_51;
reg [5:0]  opcode_out_mr_f_51,src_reg_out_mr_f_51,dest_reg_out_mr_f_51,targ_reg_out_mr_f_51;
reg [31:0] a_de_m_51,b_de_m_51;
wire [31:0] alu_out_em_m_51,alu_src_em_m_51;

parameter ADD  = 6'b110001;
parameter LDW  = 6'b010111;
parameter MUL  = 6'b100111;
parameter BLT  = 6'b010110;
parameter STW  = 6'b010101;
parameter ADDI = 6'b000100;
parameter SUBI = 6'b011111;
parameter NOPE = 6'b111111;

initial
begin
  a_de_m_51 = 0;
  b_de_m_51 = 0;
```

```
end

// initializing all values to zero,
always @(posedge clk_51 or posedge rst_51)
begin
  if(rst_51)
  begin
    NPC_fd_f_51 <= 0;
    PC_sel_fd_f_51 <= 0;
    PC_in_fd_f_51 <= 0;
    IR_fd_f_51 <= 0;
    sign_ext_de_f_51 <= 0;
    opcode_de_f_51   <= 0;
    src_reg_de_f_51  <= 0;
    dest_reg_de_f_51 <= 0;
    targ_reg_de_f_51 <= 0;
    a_de_f_51        <= 0;
    b_de_f_51        <= 0;
    alu_out_em_f_51      <= 0;
    alu_src_em_f_51      <= 0;
    opcode_out_em_f_51   <= 0;
    src_reg_out_em_f_51  <= 0;
    dest_reg_out_em_f_51 <= 0;
    targ_reg_out_em_f_51 <= 0;
    mem_out1_mr_f_51     <= 0;
    mem_out2_mr_f_51     <= 0;
    opcode_out_mr_f_51   <= 0;
    src_reg_out_mr_f_51  <= 0;
    dest_reg_out_mr_f_51 <= 0;
    targ_reg_out_mr_f_51 <= 0;
    mem_out1_mr_ff_51    <= 0;
  end
  else

// at every clock data will be transferred to the next stage:

  begin
    NPC_fd_f_51 <= NPC_fd_51;
    PC_sel_fd_f_51 <= PC_sel_fd_51;
    PC_in_fd_f_51 <= PC_in_fd_51;
    IR_fd_f_51 <= IR_fd_51;
    sign_ext_de_f_51 <= sign_ext_de_51;
    opcode_de_f_51   <= opcode_de_51;
    src_reg_de_f_51  <= src_reg_de_51;
    dest_reg_de_f_51 <= dest_reg_de_51;
    targ_reg_de_f_51 <= targ_reg_de_51;
    a_de_f_51        <= a_de_51;
    b_de_f_51        <= b_de_51;
    alu_out_em_f_51      <= alu_out_em_51;
    alu_src_em_f_51      <= alu_src_em_51;
    opcode_out_em_f_51   <= opcode_out_em_51;
    src_reg_out_em_f_51  <= src_reg_out_em_51;
    dest_reg_out_em_f_51 <= dest_reg_out_em_51;
    targ_reg_out_em_f_51 <= targ_reg_out_em_51;
    mem_out1_mr_f_51     <=  mem_out1_mr_51;
    mem_out2_mr_f_51     <=  mem_out2_mr_51;
    opcode_out_mr_f_51   <=  opcode_out_mr_51;
    src_reg_out_mr_f_51  <=  src_reg_out_mr_51;
    dest_reg_out_mr_f_51 <=  dest_reg_out_mr_51;
```

```
   targ_reg_out_mr_f_51 <=  targ_reg_out_mr_51;
   mem_out1_mr_ff_51    <=  mem_out1_mr_f_51;
  end
end

// initiating each and every modules to the main module

fetch
f(.clk_51(clk_51),.rst_51(rst_51),.mra_i_51(mra_i_51),.mrd_i_51(mrd_i_51),.PC_in_51(PC_in_fd_51),.NPC_51
(NPC_fd_51),.IR_51 (IR_fd_51),.PC_sel_51(PC_sel_fd_51));

decode
d(.clk_51(clk_51),.rst_51(rst_51),.IR_51(IR_fd_f_51),.PC_51(NPC_fd_f_51),.a_51(a_de_51),.b_51(b_de_51),.
NPC_51(PC_in_fd_51
),.PC_sel_51(PC_sel_fd_51),.sign_ext_51(sign_ext_de_51),.opcode_51(opcode_de_51),.src_reg_51(src_reg_
de_51),.dest_reg_51(des
t_reg_de_51),.targ_reg_51(targ_reg_de_51),.r0(r0),.r1(r1),.r2(r2),.r3(r3),.r4(r4),.r5(r5),.r6(r6),.r7(r7),.r8(r8),.r9(r
9),.r10(r10),.r11(r11),                .r12(r12),.r13(r13),.r14(r14),.r15(r15),.r16(r16),.r17(r17),.r18(r18),.r19(r19),
.r20(r20),.r21(r21),.r22(r22),.r23(r23),.r24(r24),.r25(r25),.r26(r26),.r27(r27),.r28(r28),.r29(r29),.r30(r30),.r31(r31)
);

execute
e(.clk_51(clk_51),.rst_51(rst_51),.opcode_51(opcode_de_f_51),.a_51(a_de_m_51),.b_51(b_de_m_51),.sign_e
xt_51(sign_ext_de_f_4
0),.alu_out_51(alu_out_em_51),.alu_src_51(alu_src_em_51),.opcode_out_51(opcode_out_em_51),.src_reg_5
1(src_reg_de_f_51),.des
t_reg_51(dest_reg_de_f_51),.targ_reg_51(targ_reg_de_f_51),.src_reg_out_51(src_reg_out_em_51),.dest_reg_
out_51(dest_reg_out_e m_51),.targ_reg_out_51(targ_reg_out_em_51));

memory_write
m(.clk_51(clk_51),.rst_51(rst_51),.opcode_51(opcode_out_em_f_51),.src_reg_51(src_reg_out_em_f_51),.dest
_reg_51(dest_reg_out_
em_f_51),.targ_reg_51(targ_reg_out_em_f_51),.alu_out_51(alu_out_em_f_51),.alu_src_51(alu_src_em_f_51),.
mem_out1_51(mem_o
ut1_mr_51),.mem_out2_51(mem_out2_mr_51),.opcode_out_51(opcode_out_mr_51),.src_reg_out_51(src_reg
_out_mr_51),.dest_reg_
out_51(dest_reg_out_mr_51),.targ_reg_out_51(targ_reg_out_mr_51),.mwr_d_51(mwr_d_51),.mwd_d_51(mwd
_d_51),.mwa_d_51(m
wa_d_51),.mra_d_51(mra_d_51),.mrd_d_51(mrd_d_51),.r0(r0),.r1(r1),.r2(r2),.r3(r3),.r4(r4),.r5(r5),.r6(r6),.r7(r7),
.r8(r8),.r9(r9),.r10(r1       0),.r11(r11),.r12(r12),.r13(r13),.r14(r14),.r15(r15),.r16(r16),.r17(r17),.r18(r18),.r19(r19),
.r20(r20),.r21(r21),.r22(r22),.r23(r23),.r24(r24),.r25(r25),.r26(r26),.r27(r27),.r28(r28),.r29(r29),.r30(r30),.r31(r31)
);

rw
r(.clk_51(clk_51),.rst_51(rst_51),.opcode_51(opcode_out_mr_f_51),.src_reg_51(src_reg_out_mr_f_51),.dest_r
eg_51(dest_reg_out_m
r_f_51),.targ_reg_51(targ_reg_out_mr_f_51),.mem_out1_51(mem_out1_mr_f_51),.mem_out2_51(mem_out2_
mr_f_51),.r0(r0),.r1(r1
),.r2(r2),.r3(r3),.r4(r4),.r5(r5),.r6(r6),.r7(r7),.r8(r8),.r9(r9),.r10(r10),.r11(r11),.r12(r12),.r13(r13),.r14(r14),.r15(r15)
,.r16(r16),.r17(r17),                                                                    .r18(r18),.r19(r19),
.r20(r20),.r21(r21),.r22(r22),.r23(r23),.r24(r24),.r25(r25),.r26(r26),.r27(r27),.r28(r28),.r29(r29),.r30(r30),.r31(r31)
);

always @(*)
begin

  a_de_m_51 = a_de_f_51;
  b_de_m_51 = b_de_f_51;
```

```
// decoding opcode and doing operation accordingly

  case(opcode_out_em_f_51)
    CALL:
    begin
      case(opcode_de_f_51)
        ADD:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
          b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
        end
        MUL:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
          b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
        end
        ADDI:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
        end
        SUBI:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
        end
        LDW:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
        end
        STW:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
          b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
        end
        BNE:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
          b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
        end
      endcase
    end
    ADD:
    begin
      case(opcode_de_f_51)
        ADD:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
          b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
        end
        MUL:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
          b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
        end
        ADDI:
        begin
          a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
        end
        SUBI:
        begin
```

```
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    LDW:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    STW:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    BNE:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
  endcase
end
MUL:
begin
  case(opcode_de_f_51)
    ADD:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    MUL:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    ADDI:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    SUBI:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    LDW:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    STW:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    BNE:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
  endcase
end
ADDI:
begin
  case(opcode_de_f_51)
    ADD:
```

```
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (dest_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    MUL:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (dest_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    ADDI:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    SUBI:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    LDW:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    STW:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (dest_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    BNE:
    begin
      a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
  endcase
end
SUBI:
begin
  case(opcode_de_f_51)
    ADD:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (dest_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    MUL:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
      b_de_m_51 = (dest_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
    end
    ADDI:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    SUBI:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    LDW:
    begin
      a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
    end
    STW:
```

```
      begin
        a_de_m_51 = (dest_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
        b_de_m_51 = (dest_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
      end
      BNE:
      begin
        a_de_m_51 = (targ_reg_out_em_f_51 == src_reg_de_f_51 ) ? alu_out_em_f_51 : a_de_f_51;
        b_de_m_51 = (targ_reg_out_em_f_51 == dest_reg_de_f_51) ? alu_out_em_f_51 : b_de_f_51;
      end
    endcase
  end
  LDW:
  begin
  end
endcase

case(opcode_out_mr_f_51)
  ADD:
  begin
    case(opcode_de_f_51)
      ADD:
      begin
        a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
        b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
      end
      MUL:
      begin
        a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
        b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
      end
      ADDI:
      begin
        a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      end
      SUBI:
      begin
        a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      end
      LDW:
      begin
        a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      end
      STW:
      begin
        a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
        b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
      end
      BNE:
      begin
        a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
        b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
      end
    endcase
  end
  MUL:
  begin
    case(opcode_de_f_51)
      ADD:
      begin
```

```
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    MUL:
    begin
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    ADDI:
    begin
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    SUBI:
    begin
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    LDW:
    begin
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    STW:
    begin
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    BNE:
    begin
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
  endcase
end
ADDI:
begin
  case(opcode_de_f_51)
    ADD:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    MUL:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    ADDI:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    SUBI:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    LDW:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    STW:
    begin
```

```
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    BNE:
    begin
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
  endcase
end
SUBI:
begin
  case(opcode_de_f_51)
    ADD:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    MUL:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    ADDI:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    SUBI:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    LDW:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
    end
    STW:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    BNE:
    begin
      a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
  endcase
end
LDW:
begin
  case(opcode_de_f_51)
    ADD:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
    end
    MUL:
    begin
      a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
```

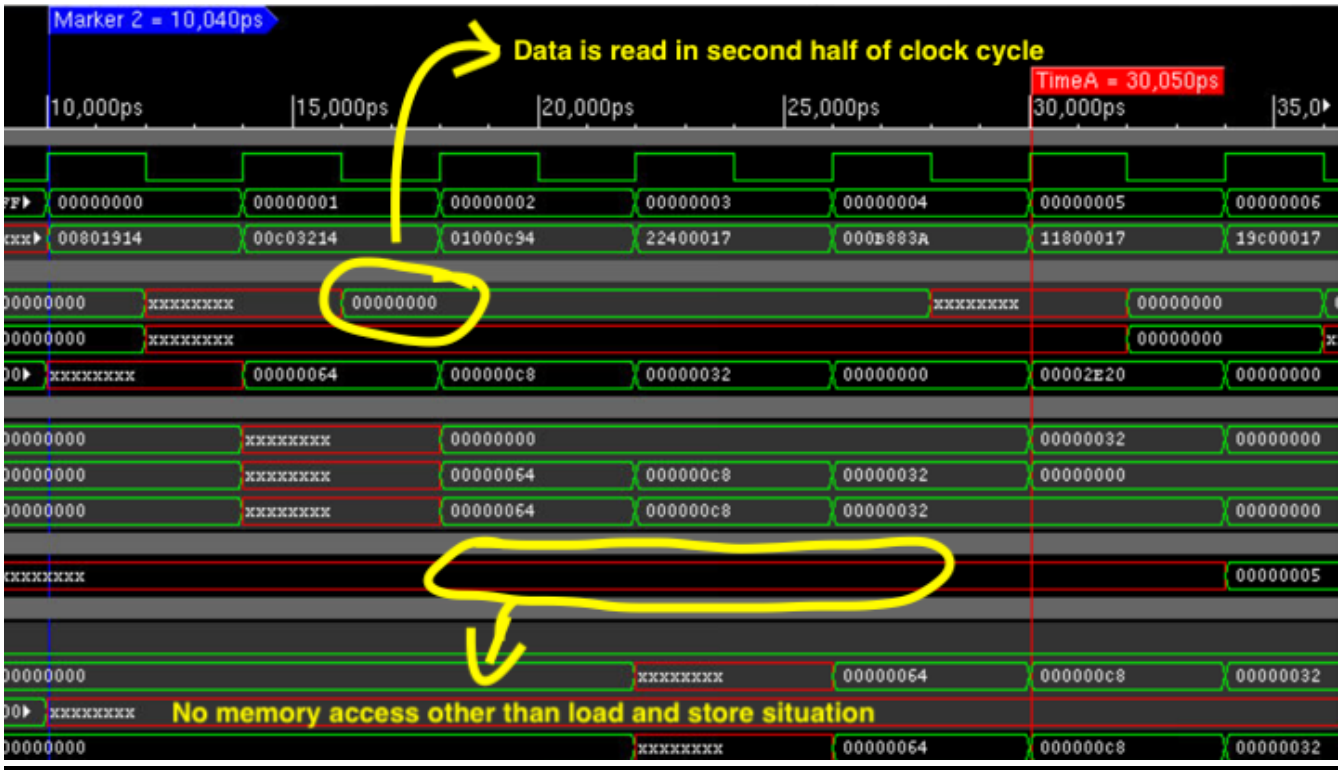
```
      end
      ADDI:
      begin
        a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      end
      SUBI:
      begin
        a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      end
      LDW:
      begin
        a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
      end
      STW:
      begin
        a_de_m_51 = (dest_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
        b_de_m_51 = (dest_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
      end
      BNE:
      begin
        a_de_m_51 = (targ_reg_out_mr_f_51 == src_reg_de_f_51 ) ? mem_out1_mr_f_51 : a_de_m_51;
        b_de_m_51 = (targ_reg_out_mr_f_51 == dest_reg_de_f_51) ? mem_out1_mr_f_51 : b_de_m_51;
      end
     endcase
   end
  endcase
end
endmodule


//Data memory

// Now making a Data memory where enrolment will be stored. It stores the data.

module data_mem_51(input clk_51,input [31:0] waddr_51,
   input [31:0] wdata_51, input write_51,
   input [31:0] raddr_51, output [31:0] rdata_51);

reg [31:0] mem_51[0:2000];
reg [31:0] rdata_51x;
wire [31:0] w0,w1,w2;
initial
begin

// Writing SJSU ID in the mem_51ory locations:
// SJSU ID: 010939851
// For 9 Digits 9 Location will be required
   mem_51[26] = 32'h00000000;    // Digit: 0(Decimal) = = = 0(Hex)
   mem_51[30] = 32'h00000001;    // Digit: 1(Decimal) = = = 1(Hex)
   mem_51[34] = 32'h00000001;    // Digit: 1(Decimal) = = = 1(Hex)
   mem_51[38] = 32'h00000004;    // Digit: 4(Decimal) = = = 4(Hex)
   mem_51[42] = 32'h00000008;    // Digit: 8(Decimal) = = = 8(Hex)
   mem_51[46] = 32'h00000007;    // Digit: 7(Decimal) = = = 7(Hex)
   mem_51[50] = 32'h00000008;    // Digit: 8(Decimal) = = = 8(Hex)
   mem_51[54] = 32'h00000004;    // Digit: 4(Decimal) = = = 4(Hex)
   mem_51[58] = 32'h00000000;    // Digit: 0(Decimal) = = = 0(Hex)

// Writing double of SJSU ID in the memory locations:
```

```
// SJSU ID:   0 1 0 9 3 9 8 5 1
// Double of it: 0 2 0 18 6 18 16 10 2

// For 9 Digits 9 Location will be required
    mem_51[60] = 32'h00000000; // Digit: 0(Decimal)  = = =  0(Hex)
    mem_51[64] = 32'h00000002; // Digit: 2(Decimal)  = = =  2(Hex)
    mem_51[68] = 32'h00000002; // Digit: 2(Decimal)  = = =  2(Hex)
    mem_51[72] = 32'h00000008; // Digit: 8(Decimal)  = = =  8(Hex)
    mem_51[76] = 32'h00000010; // Digit: 16(Decimal) = = =  10(Hex)
    mem_51[80] = 32'h0000000E; // Digit: 14(Decimal) = = =  E(Hex)
    mem_51[84] = 32'h00000010; // Digit: 16(Decimal) = = =  10(Hex)
    mem_51[88] = 32'h00000008; // Digit: 8(Decimal)  = = =  8(Hex)
    mem_51[92] = 32'h00000000; // Digit: 0(Decimal)  = = =  0(Hex)
end
assign rdata_51 = rdata_51x;
always @(*) begin
  rdata_51x <=  mem_51[raddr_51];
end
always @(posedge(clk_51)) begin
  if(write_51) begin
    mem_51[waddr_51]<=#2 wdata_51;
  end
//  $monitor (" mem_51 ----->>>> %d  at time %t",mem_51[7],$realtime);
end
endmodule


//Instruction memory
// Making instruction memory, where instruction will be store.

module ins_mem_51(input clk_51,input [31:0] waddr_51,
    input [31:0] wdata_51, input write_51,
    input [31:0] raddr_51, output [31:0] rdata_51);
reg [31:0] mem_51[0:2000];
reg [31:0] rdatax_51;
wire [31:0] w0,w1,w2;
initial
begin

   mem_51[0] =    32'h0000003F;          //// nope
   mem_51[4] =    32'h0000003F;          //// nope
   mem_51[8] =    32'h00800684;          //// addi r2,avector,r0
   mem_51[12] =   32'h00C00F04;          //// addi r3,bvector,r0
   mem_51[16] =   32'h01000284;          //// r4,N,r0
   mem_51[20] =   32'h0000003F;          //// nope(ldw r4,0,r4)
   mem_51[24] =   32'h000A0031;          //// add r5,r0,r0
   mem_51[28] =   32'h11800017;          //// ldw r6,0,r2      LOOP STARTS FROM HERE

   mem_51[32] =   32'h19C00017;          //// ldw r7,0,r3
   mem_51[36] =   32'h0000003F;          //// nope
   mem_51[51] =   32'h39900027;          //// mul r8,r6,r7
   mem_51[44] =   32'h414A0031;          //// add r5,r5,r8
   mem_51[48] =   32'h10800104;          //// addi r2,r2,4
   mem_51[52] =   32'h18C00104;          //// addi r3,r3,4
   mem_51[56] =   32'h2100005F;          //// subi r4,r4,1
   mem_51[60] =   32'h01000716;          //// blt r0,r4,loop
   mem_51[64] =   32'h0000003F;          //// nope
   mem_51[68] =   32'h01510115;          //// stw r5,dotproduct(r0)
   mem_51[72] =   32'h00000106;          //// br stop              Loop stops here
```

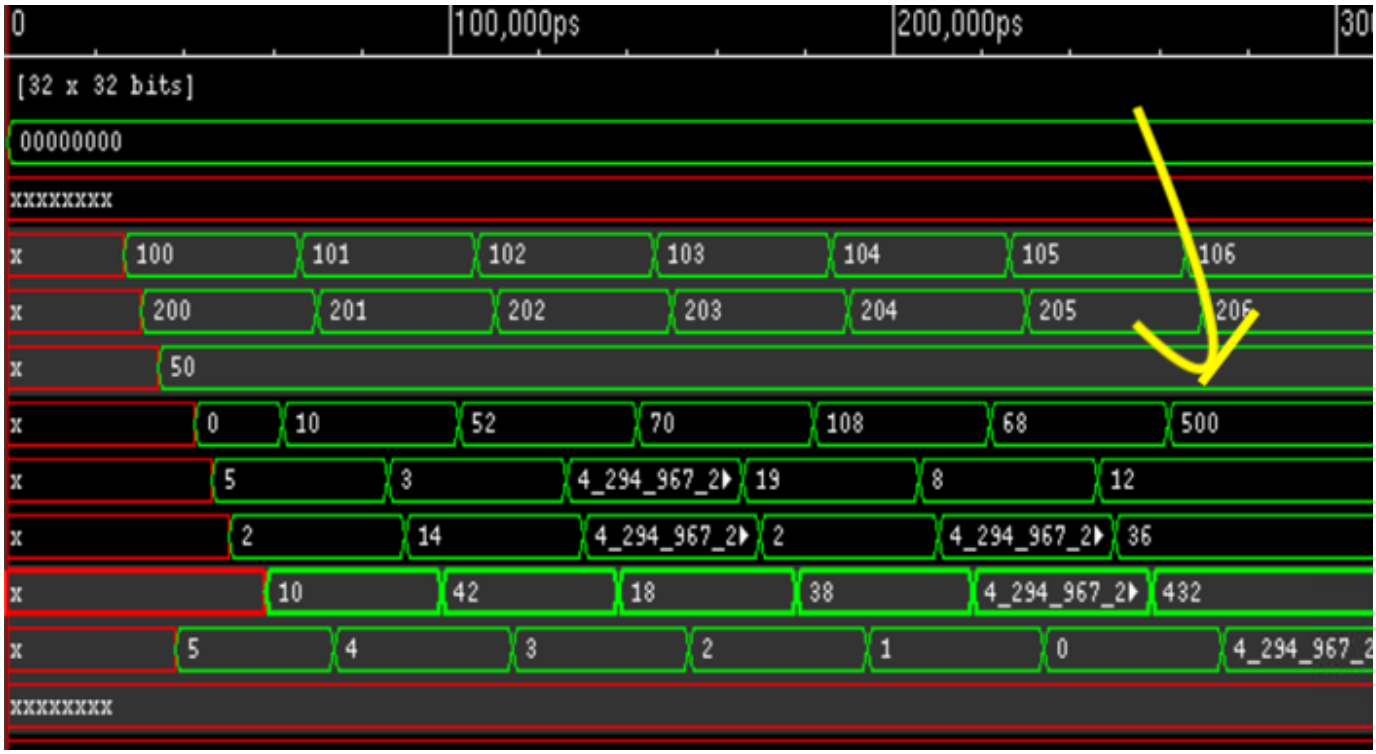
```
end
assign rdata_51 = rdatax_51;
always @(*) begin
  rdatax_51 <=  mem_51[raddr_51];
end
always @(posedge(clk_51)) begin
  if(write_51) begin
    mem_51[waddr_51]<=#2 wdata_51;
  end
end
endmodule // Test bench:

// Test Bench for the given instructions:

`include "mem_code.v"
`include "ins_mem_51.v"
`include "combine_proce.v"
module proc_tb();
reg clk_51,rst_51;
wire [31:0] mrd_d_51,mrd_i_51;
wire [31:0] mwd_d_51,mwd_i_51;
wire [31:0] mwa_d_51,mwa_i_51;
wire [31:0] mra_d_51,mra_i_51;
wire        mwr_d_51,mwr_i_51;

//proc DUT(clk,rst,mrd_i,mwd_i,mra_i,mwa_i,mwr_i,mrd_d,mwd_d,mra_d,mwa_d,mwr_d);

proc
DUT(clk_51,rst_51,mrd_i_51,mwd_i_51,mra_i_51,mwa_i_51,mwr_i_51,mrd_d_51,mwd_d_51,mra_d_51,mwa_
d_51,mwr_d_51);
ins_mem_51
m0(.clk_51(clk_51),.waddr_51(mwa_i_51),.wdata_51(mwd_i_51),.write_51(mwr_i_51),.raddr_51(mra_i_51),.rd
ata_51(mrd_i_51));

data_mem_51
m1(.clk_51(clk_51),.waddr_51(mwa_d_51),.wdata_51(mwd_d_51),.write_51(mwr_d_51),.raddr_51(mra_d_51),
.rdata_51(mrd_d_51) );
initial
begin
  clk_51 = 1'b0;
  forever #5 clk_51 = ~clk_51;
end

initial
// Reset signal intiated.
begin
  #2 rst_51 = 1'b1;
  #5 rst_51 = 1'b0;
end

initial
begin
  $dumpfile ("proc.vcd");
  $dumpvars (0,proc_tb);
  #2500 $finish();
end
endmodule
```

SIMULATION RESULTS

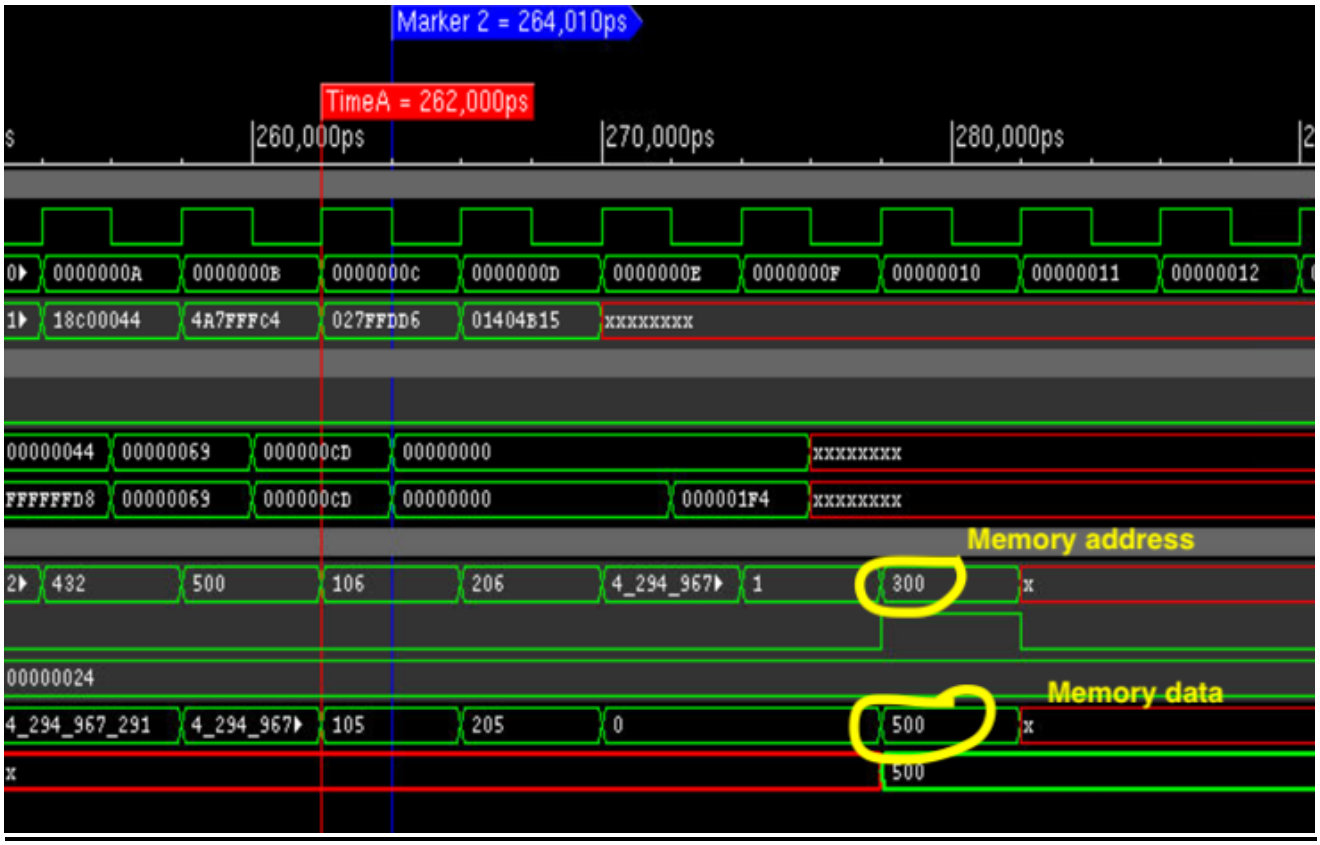
Step1:



Step 2:


0
100,000ps
200,000ps
[32 x 32 bits]
00000000
xxxxxxxx
x 100 101 102 103 104 105 106
x 200 201 202 203 204 205
x 50
x 0 10 52 70 108 68 500
x 5 3 4_294_967_2 19 8 12
x 2 14 4_294_967_2 2 4_294_967_2 36
x 10 42 18 38 4_294_967_2 432
x 5 4 3 2 1 0
xxxxxxxx

Step 3:


Marker 2 = 264,010ps
TimeA = 262,000ps
260,000ps
270,000ps
280,000ps
Memory address
Memory data