MULTICYCLE PROCESSOR DESIGN IN VERILOG

BSCpE V-GF (A.Y. 2019-2020)

Computer Engineering Department

Southern Luzon State University

Lucban, Quezon

ALU

Quillope, Rodann Apollo T. (Leader)

Camelon, Andrea Lee B.

De Asis, Novalyn Keith S.

De La Cruz, Charra Joy S.

Del Rosario, Jelmer N.

Rescober, Gladys Ann A.

CONTROL UNIT

Cabriga, Adrian M. (Leader)

Betito, Ryan Joshua R.

Cabrera, Aniceto Jerwin V.

De Silva, Mark Vincent F.

Imperial, Randy O.

Obmerga, Carl Adrian J.

MEMORY

Ursolino, Seth Howell M. (Leader)

Aco, Klark Ashram A.

Felicidario, Kay Ann R.

Malabayabas, Marrence L.

Morillo, Arlan John

Reyes, Lorie Mae J.

REGISTER FILE

Bojilador, Charmayne Mari S. (Leader)

Cabral, Jomarie A.

Evangelista, Dan Mark John

Pajilan, Cesar P.

Rañada, Marize Valery V.

Villaverde, Paolo C.

DATAPATH

Cuadillera, Frex C. (Leader)

Alvarez, Jhon Milbert C.

Casao, Gabriel P.

Daliva, Adrian Kent B.

Pawang, Emmanuel Z.

Yanoria, Mark N.

I. LEARNING OUTCOME

At the end of the project, the students must be able to:

- Define a processor and its components.
- Distinguish between a single-cycle and multicycle processor.
- Demonstrate how a processor works step-by-step.

II. OBJECTIVES

The project aims to attain the following:

1. Design a complete multi cycle processor.
2. Write the design of each component of the processor, i.e. ALU, control unit, register.
3. Simulate the design using testbenches.

III. INTRODUCTION

Computer architecture is the implementation of multiple designs that describe the methods and rules used in a computer system. As the start of a new age of electronics from the late 50s, each computer design is unique from each other. Programs are only created and used for a specific function. This was a drawback later realized when the start of the era of microprocessors and transistors came and major languages were unable to standardize their numeric behavior because decimal computers had groups of users too large to alienate. By early 1970, the term personal computer was introduced with the KENBAK-1, although the first personal computer that uses a microprocessor was the Micral, which uses the Intel 8008 processor developed by Vietnamese and French engineers André Truong Trong Thi and Francois Gernelle in 1973.

One of the main parts of a computer is its processor. The processor, also known as Central Processing Unit (CPU), is the brain of a computer. It allows processing of data in binary form and execution of instructions that will later be stored in the memory.

The first commercial processor was invented in 1971. It is the Intel 4004, a single chip CPU containing 4 bits, created by a group led by Federico Faggin in Intel Corporation. Since then, processors have come a long way in terms of its specifications and functions.

Part of the processor is the datapath, which is responsible for the functional unit and connections necessary to implement an instruction set architecture of the processor. With that said, data paths are responsible for holding, operating and transferring of data. The control unit is the component in the processor that tells the datapath what to do and which component will operate based on the instruction passed to it.

Datapaths take on many forms such as single and multicycle, though the multicycle datapath just fixes the shortcomings of the single-cycle processor. The multicycle processor provides a much simpler datapath, though the control unit of the system can be quite complex. It is architecturally secured either by gating the data or clock from reaching the destination flops. There can be many such scenarios inside a system on chip where we can apply multi-cycle paths.

IV. RELATED LITERATURE

The following pieces of information have been taken for the main reason that they are relevant to the project. In addition, this helped the students further understand the entire structure of the multicycle processor and how to accomplish the project as a result.

Multicycle Processor

A single-cycle processor is simple to implement. However, it has multiple disadvantages, such as the cycle time being limited by the longest instruction, which is the load word (lw) instruction. Also, it requires three adders: one in the arithmetic logic unit (ALU) and two for the PC logic. Adders are costly, especially if the speed of the circuit is considered. Lastly, it has separate instruction and data memories, making it impractical since these two memories may as well be joined together.

The multicycle processor solves these problems, providing higher clock speed, different instructions which use different numbers of steps (thus simpler instructions complete faster), and utilization of only one adder, which is reused for different purposes throughout the entire cycle.

Multicycle Datapath

The design for the datapath of a MIPS processor is similar to that of the single-cycle processor, only it involves using a combined memory for both data and instructions, since by doing this it is easier to read in one cycle and is more practical. Reading or writing the data takes place in a separate cycle.

First, the processor fetches the instruction, and then reads the operands from the register file. Then, if there are any, the processor sign-extends the immediate. The memory address is then computed, and the data is read from there. The data will be written back to the register file, specified by the rt field of the instruction memory, after which the program counter (PC) gets incremented by 4. Datapaths specifically for the R-Type, store word (sw) and branch-if-equal (beq) instructions have been enhanced specifically for the multicycle processor.

The multicycle datapath follows the five stage execution process: Fetch, decode, ALU, data access, and register write. Below are the control signals for the fetch and decode steps.

IodD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite = 1
PCWrite = 1

Figure 4.1 Fetch Control Signals

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

Figure 4.2 Decode Control Signals

The fetch control signals are used to calculate PC + 4 for the next instruction. The decode control signals are mainly used for branching. After the fetch and decode portions, the datapath of I-type, J-type, and R-type instructions vary.

R-Type

If the opcode is an R-type instruction, the result must be calculated using the ALU and stored back to the register. To carry out ALU calculation, ALUSrcA is set to 1, ALUSrcB is set to 0, and ALUOp is set to 10. ALUSrcA selects the $rs register to be used as SrcA and ALUSrcB selects $rt register to be used as SrcB of the ALU. ALUOp set to 10 indicates to the controller that ALU operation mode is dependent on the function field of the instruction. For result storage, RegDst and RegWrite are set to 1 and MemtoReg is set to 0. RegDst selects $rd register as the write destination and MemtoReg indicates the data to be written is from ALU. RegWrite serves as a write enable for the Register File.

I-Type

Unlike the R-type instruction, not all I-type instructions are carried out the same. The load word (lw), store word (sw), add immediate (addi), and branch if equal (beq) use different amounts of cycles. After the lw and sw instructions are decoded, the address for memory access must be computed by adding a base address located in the $rs register and a sign extended immediate. Control signals must be set to control the multiplexers that handle inputs at various sections of the datapath. The appropriate control signals for this step are ALUSrcA to 1, ALUSrcB to 10, and ALUOp to 00. Next, the calculated address is used to access memory; IorD is set to 1 to indicate that the incoming address is from the ALU. For sw, MemWrite is set to 1. The data located in the WriteData (WD) portion is stored to memory. Register $rt is always fed to WD, however the data from $rt is not written to memory unless MemWrite is asserted. The sw instruction is done, but lw has to write back to the register. Three control signals are set: RegDst to 0, MemtoReg to 1, and RegWrite to 1. Similar to the memory portion, if RegWrite is not set to 1, the data inside WD3 will not be written to the Register File.

For addi, $rs is still added to a sign extended immediate, but instead of using the result to access memory, the result is stored in the address located in $rt. The control signals for the ALU computation remain the same as the lw and sw instructions. Afterwards, the result is written to the Register File. The control signals for this are RegDst and RegWrite to 1 and MemtoReg to 0.

The beq instruction has less stages than all of the other I type instructions. The branch is evaluated immediately when the ALU result is calculated. To test if a branch is equal, the values stored in $rs and $rt are subtracted from each other. If the differences between the two registers are zero, the values are equal. The result of the subtraction is indicated by the zero signal from the ALU. Once the zero signal is set to one, the result is fed into a two input AND gate with the branch signal. The result of that AND gate will produce a 1 and make PCEn 1. While this takes place, the result of the ALU is fed into the PC. With PCEn set to 1, the value of PC will be overwritten with this result. The control signals needed to carry out the branch are ALUSrcA to 1, ALUSrcB to 00, ALUOp to 01, Branch to 1, and PCSrc to 1.

J-Type

Whenever a J-Type instruction is indicated by the opcode, the 26 least significant bits are taken from the instruction and modified as a pseudo direct address. After the instruction is decoded, the PCSrc control signal is set to 10 and the PCWrite to 1. The PCSrc signal allows the ALUResult to circumvent the register and go directly to the PC. The PCWrite makes the OR gate of PCEn to produced a 1 and enable the PC register to be overwritten.

To further understand how these datapaths are linked together, a state diagram of the Finite State Machine (FSM) is used. The state diagram is also helpful in gauging the performance of the multicycle processor. The FSM is shown below.

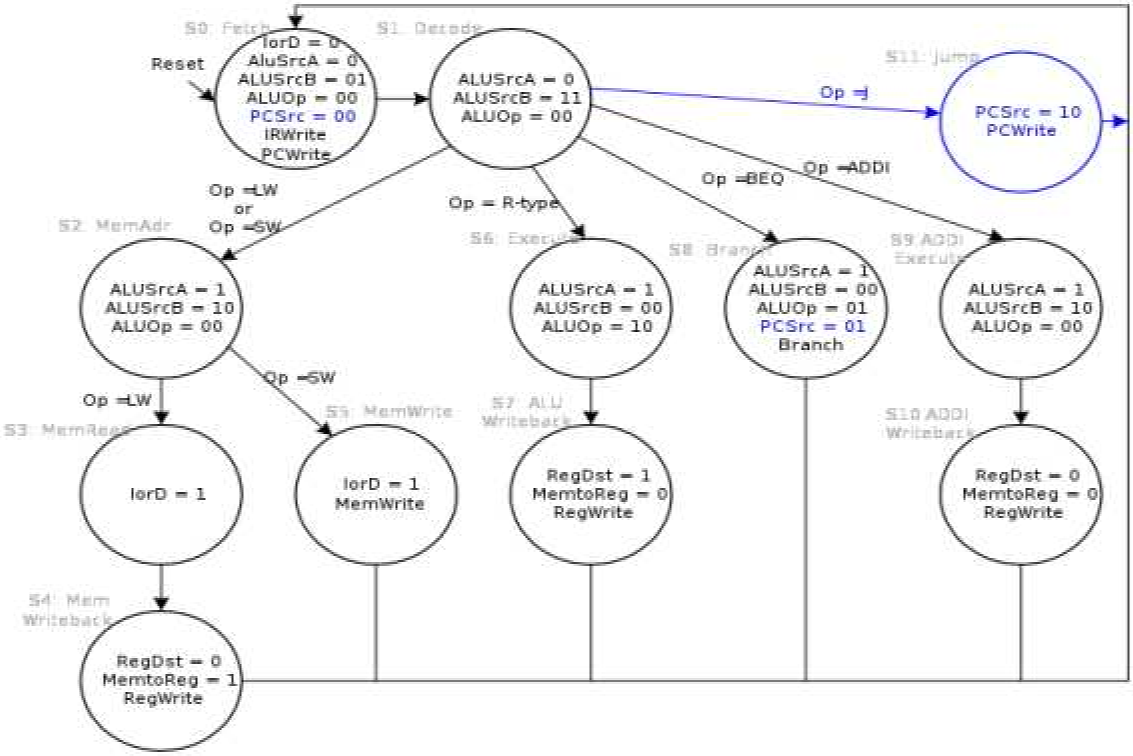


Figure 4.3 FSM of Multicycle

Multicycle Control

Just like in the single-cycle processor, the computation of the control signals based on the opcode and funct fields of the instruction memory are done in the control unit. The control unit is connected to several parts of the datapath, each component having something to do with the control unit.

The sequence of control signals depends on the instruction being executed. There are two kinds of signals that a control unit sends out to each component: a select signal, and an enable signal. A select signal allows a component to select the appropriate data that it will be using for that certain instruction. On the other hand, an enable signal is on whenever the controller tells a component whether they are to act upon the instruction during that cycle. Otherwise, its value is 0.

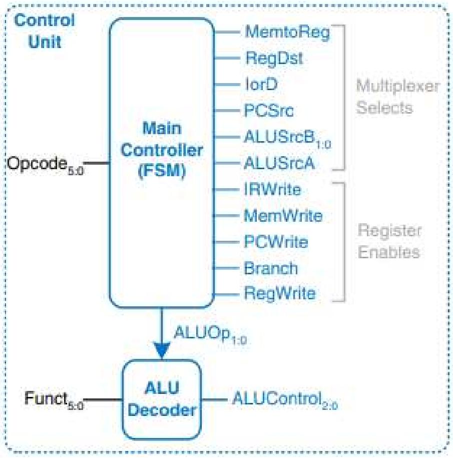


Figure 4.4 Control Unit Internal Structure

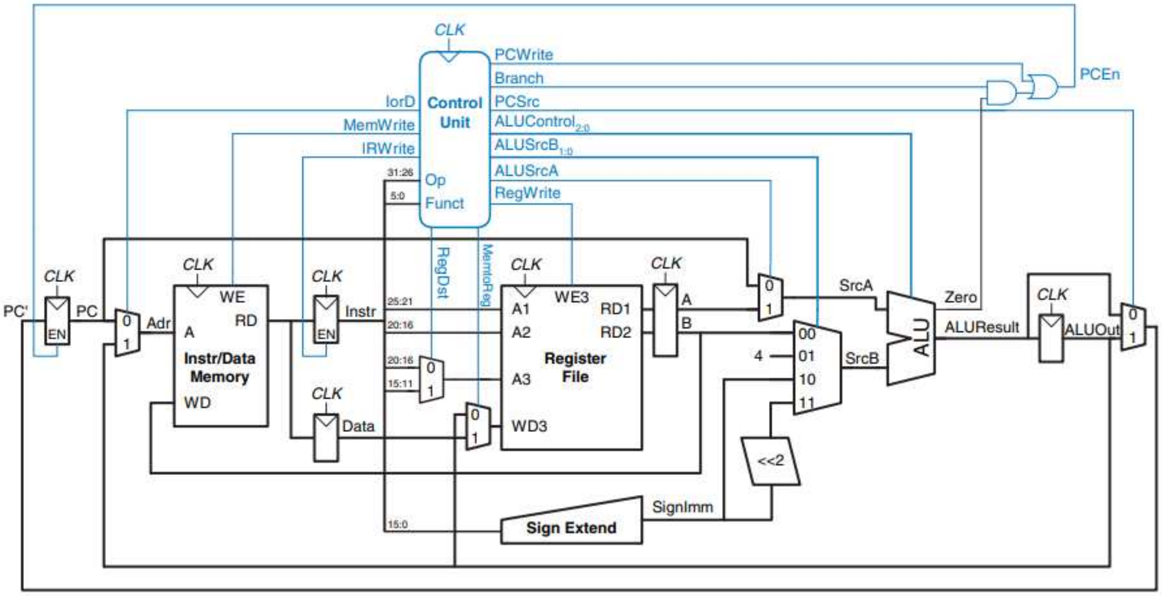


Figure 4.5 Complete multicycle MIPS processor

Multicycle Implementation

The multicycle implementation fixes many of the shortcomings of the single cycle design. The single cycle processor must have a clock period long enough to support the slowest instruction. However the multicycle is not hampered by this limitation. With multicycle, multiple clock cycles with shorter periods are used. Economy of hardware is another weakness of single cycle implementations. Multicycle implementations reuses components of the datapath, making it more cost efficient than the single cycle processor.

Multiple clock cycles allow the processor to break up an instruction into multiple shorter steps. With multicycle, a subset of actions required for one instruction is performed in one cycle. This allows shorter instructions to be executed faster. This process is analogous to a dental office allotting time to patients in multiples of 15 minutes, depending on the amount of work that is anticipated. The following figure illustrates the single cycle and multicycle clock periods and how clock period affects instruction execution.

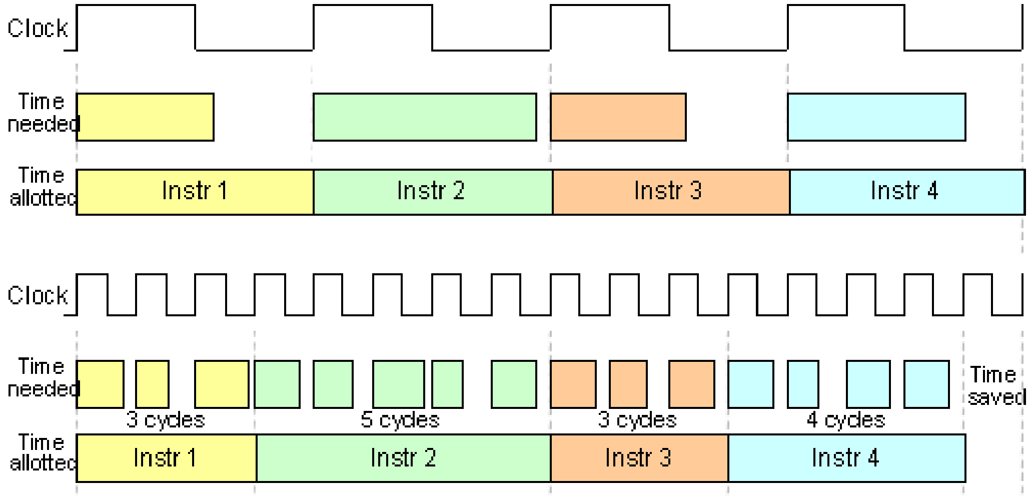


Figure 4.6 Multicycle vs. Single Cycle

The economy of hardware is addressed by reusing components or combining them. The single cycle uses three adders (two for PC logic and one for ALU) and separate memory for data and instructions. The multicycle implementation combines the data and instruction memory and uses one ALU to execute all addition instructions. Adders are expensive circuits because they take up space and require extra transistors.

The design of the multicycle processor is similar to the single cycle. The multicycle processor consists of the datapath and controller block. A controller is added to produce different signals on different steps during execution. These two portions are connected to an external memory. Inside the datapath, combinational logic connects architectural state elements. Non architectural state elements such as registers are used to hold intermediate results between stages.

Multicycle Processor Performance

In a multicycle processor, the number of cycles differ with each instruction, e.g. branch-equal and jump instructions take 3 cycles each; R-Type, store word, and addi instructions take 4 cycles each; and lastly, it takes 5 cycles for the lw instruction. The execution time of each instruction depends on the number of cycles it uses, as well as the cycle time. The multi-cycle processor consumes a lot of cycles, but each cycle takes much less time than in a single-cycle processor.

The CPI (number of cycles per instruction) depends on the usage of each instruction, i.e. the relative probability that it is used. The CPI is basically the “weighted average” number of cycles. According to the SPECINT2000 benchmark, an instruction has 25% load instructions, 10% store instructions, 11% branches, 2% jumps, and 52% R-Type instructions. With this, the average CPI can be computed by adding the products of each instruction’s frequency with its number of cycles.

V. METHODOLOGY

The following section describes the individual elaboration of each processor component, its uses, and the description of its process. It also includes the creation of the programs in Verilog.

ALU

The arithmetic/logic unit (ALU) is a combinational circuit that performs arithmetic and bitwise operations on integer binary numbers. It is mainly composed of two input operands, a function selector, output result and flag bits. In this multicycle processor design, zero flag is used and passed to the control unit.

F[2:0]	**Function**
000	A AND B
001	A OR B
010	A + B
011	Not used
110	A - B
111	SLT

ALU Operations

alu.v

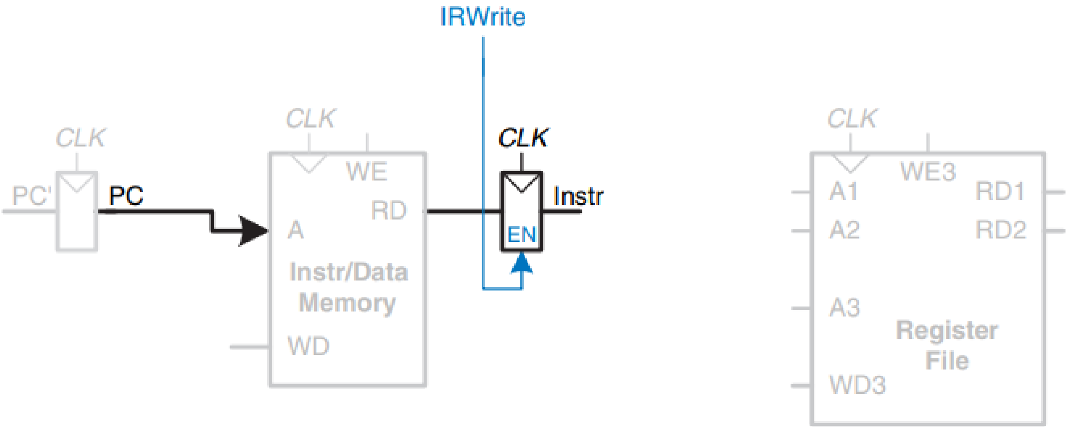
```
module alu(     input [31:0] A, B,
                input [2:0] F,
                output reg [31:0] Y,
                output Zero);

        always @ ( * )
                case (F[2:0])
                        3'b000: Y <= A & B; // AND
                        3'b001: Y <= A | B; // OR
                        3'b010: Y <= A + B; // ADD
                        //3'b011: Y <= 0;  // not used
                        3'b110: Y <= A - B; // SUB
                        3'b111: Y <= A < B ? 1:0; //SLT
                        default: Y <= 0; //default to 0, should not happen
                endcase

        assign Zero = (Y == 32'b0);
endmodule
```

Datapath

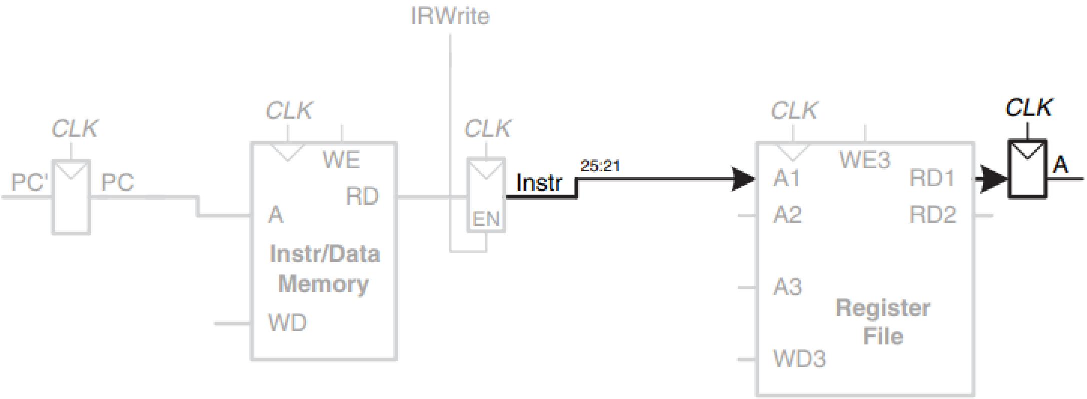
We begin our design with the memory and architectural state of the MIPS processor. In the single-cycle design, we used separate instruction and data memories because we needed to read the instruction memory and read or write the data memory all in one cycle. Now, we choose to use a combined memory for both instructions and data. This is more realistic, and it is feasible because we can read the instruction in one cycle, then read or write the data in a separate cycle. The PC and register file remain unchanged as in the single-cycle processor. We gradually build the datapath by adding components to handle each step of each instruction. The new connections are emphasized in black (or blue, for new control signals), whereas the hardware that has already been studied is shown in gray.

The following are the steps to complete the datapath of the multicycle processor:

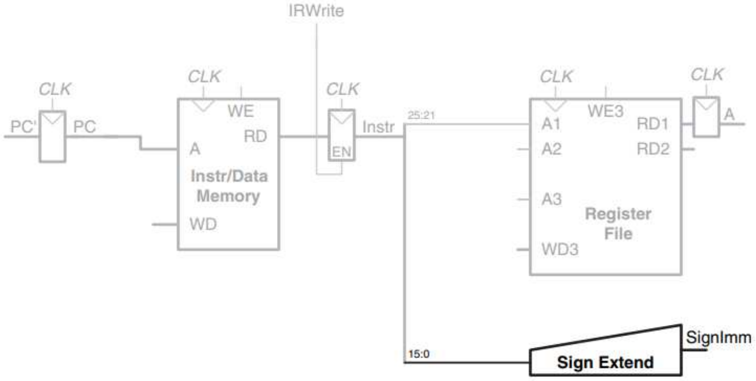
Step 1: Fetch instruction from memory



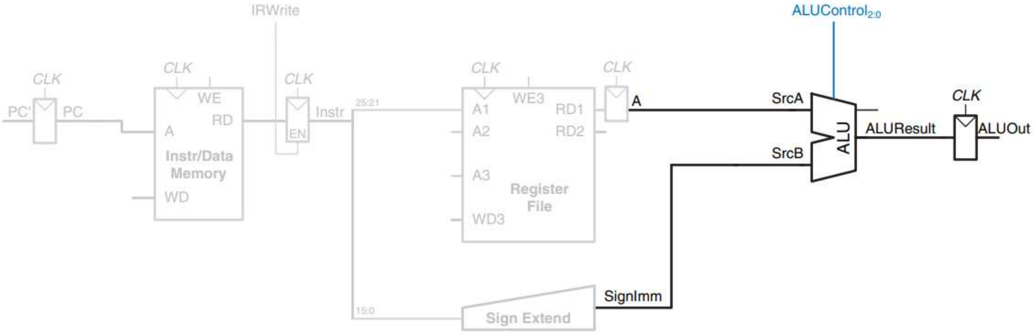
Step 2a: Read source operand from register file



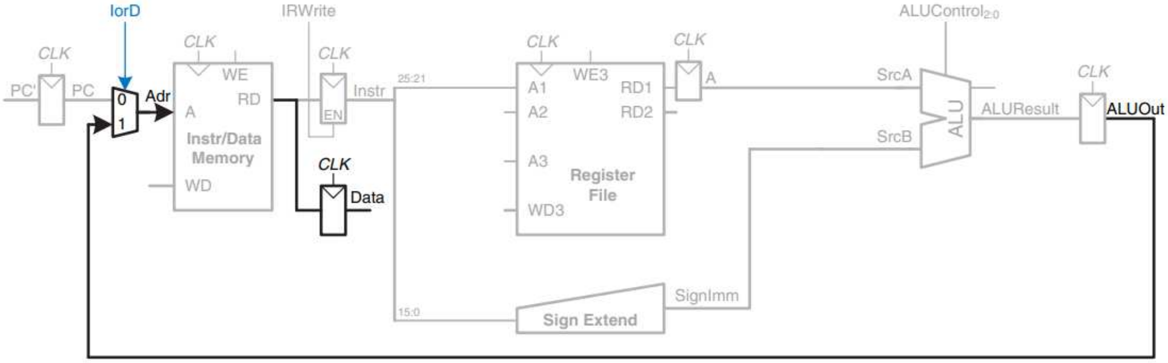
Step2b: Sign-extend the immediate



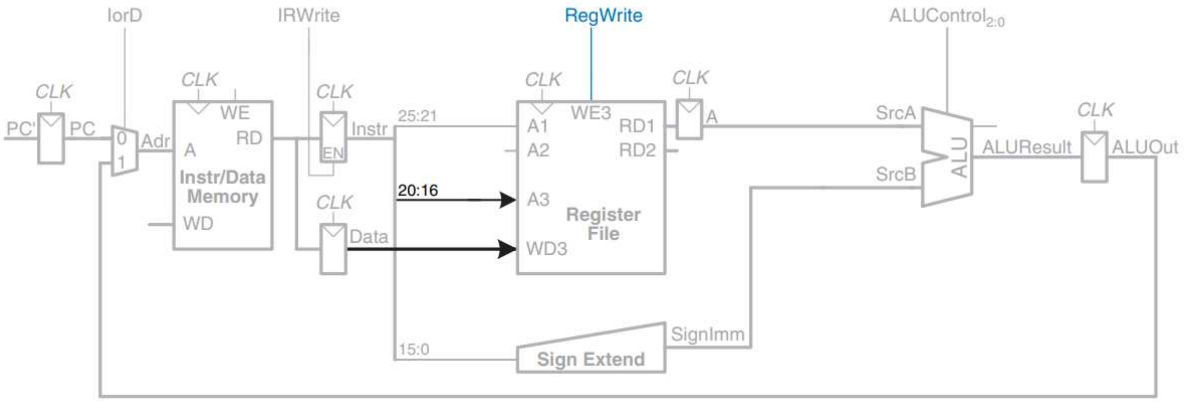
Step 3: Add base address to offset



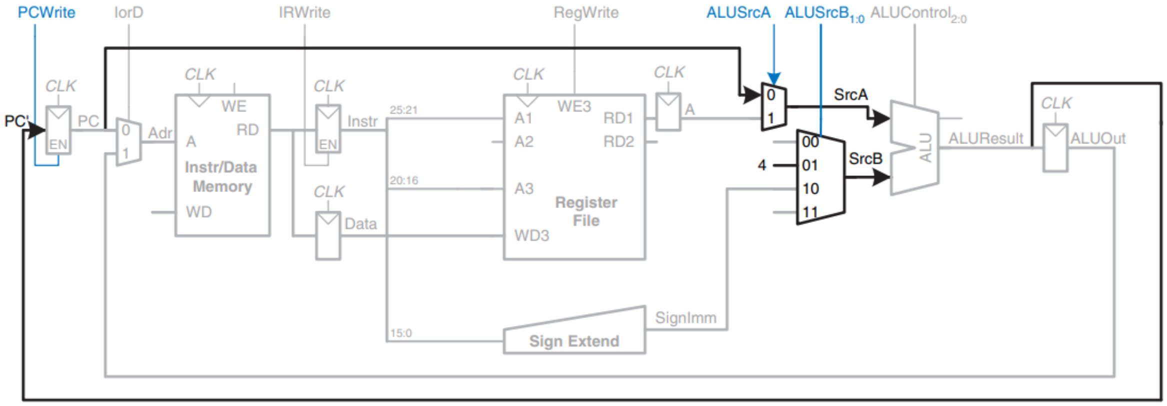
Step 4: Load data from memory



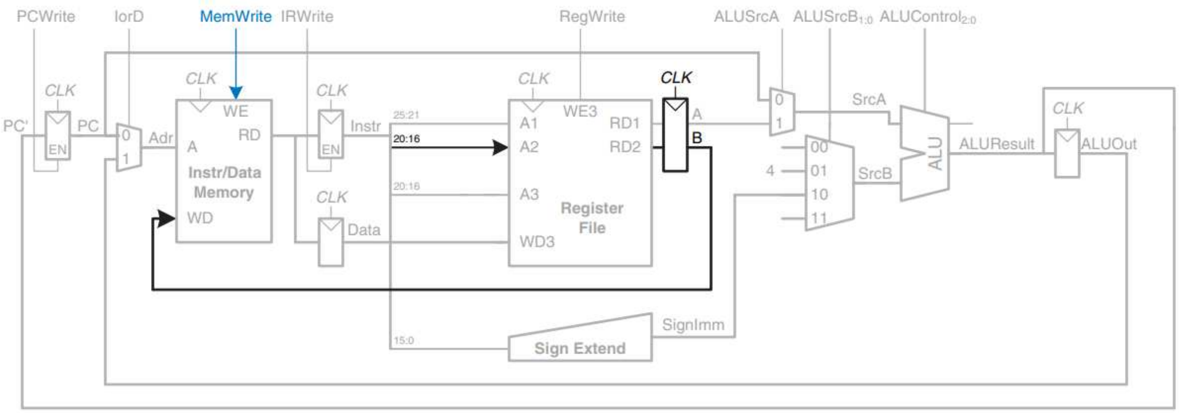
Step 5: Write data back to register file



Step 6: Increment PC by 4

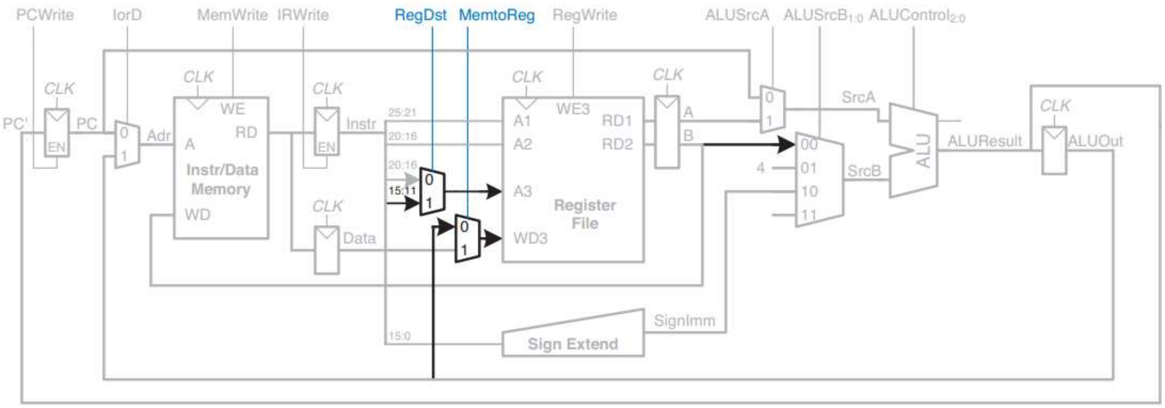


This completes the datapath for the lw instruction. Next, let us extend the datapath to also handle the sw instruction. Like the lw instruction, the sw instruction reads a base address from port 1 of the register file and sign-extends the immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by existing hardware in the datapath.



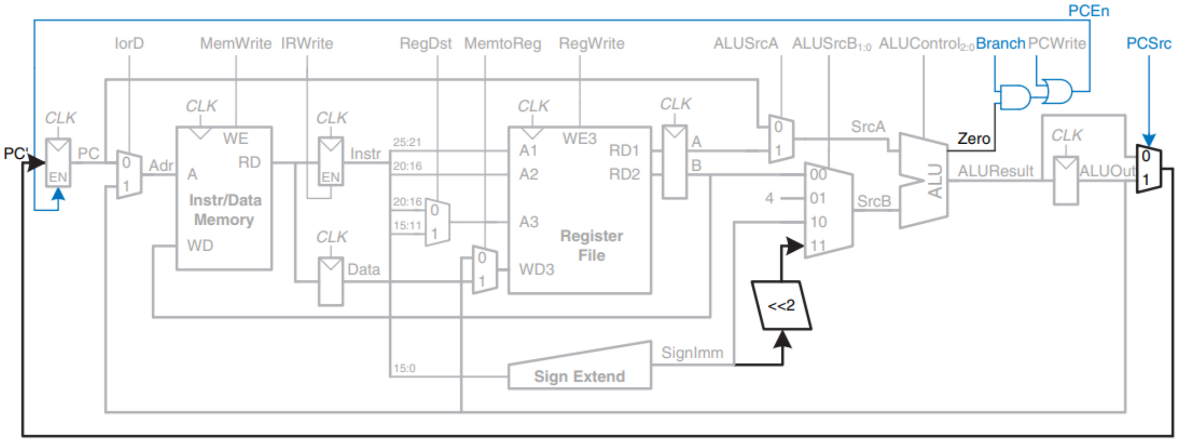
Enhanced datapath for sw instruction

For R-type instructions, the instruction is fetched again, and the two source registers are read from the register file. ALUSrcB1:0, the control input of the SrcB multiplexer, is used to choose register B as the second source register for the ALU.



Enhanced datapath for R-type instructions

For the beq instruction, the instruction is fetched again, and the two source registers are read from the register file. To determine whether the registers are equal, the ALU subtracts the registers and, upon a zero result, sets the Zero flag. Meanwhile, the datapath must compute the next value of the PC if the branch is taken: PC′ = PC + 4 + SignImm × 4.



Enhanced datapath for beq instruction

datapath.v

```
module datapath(input clk, reset,
                input pcen, irwrite, regwrite,
                input alusrca, iord, memtoreg, regdst,
                input [1:0]  alusrcb,
                input [1:0]  pcsrc,
                input [2:0]  alucontrol,
                output [5:0]  op, funct,
                output zero,
                output [31:0] adr, writedata,
                input [31:0] readdata);

  // Internal signals of the datapath module

  wire [4:0]  writereg;
  wire [31:0] pcnext, pc;
  wire [31:0] instr, data, srca, srcb;
  wire [31:0] a;
  wire [31:0] aluresult, aluout;
  wire [31:0] signimm;   // the sign-extended immediate
  wire [31:0] signimmsh;          // the sign-extended immediate shifted left by 2
  wire [31:0] wd3, rd1, rd2;
  // op and funct fields to controller
  assign op = instr[31:26];
  assign funct = instr[5:0];

  // datapath
  flopenr #(32) pcreg(clk, reset, pcen, pcnext, pc);
  mux2 #(32) adrmux(pc, aluout, iord, adr);
  flopenr #(32) instrreg(clk, reset, irwrite, readdata, instr);
  flopr #(32) datareg(clk, reset, readdata, data);

  mux2    #(5)  regdstmux(instr[20:16], instr[15:11], regdst, writereg);
  mux2    #(32) wdmux(aluout, data, memtoreg, wd3);
  regfile       rf(clk, regwrite, instr[25:21], instr[20:16],
                     writereg, wd3, rd1, rd2);
  signext       se(instr[15:0], signimm);
  sl2 immsh(signimm, signimmsh);
  flopr #(32) areg(clk, reset, rd1, a);
  flopr #(32) breg(clk, reset, rd2, writedata);
  mux2 #(32) srcamux(pc, a, alusrca, srca);
  mux4 #(32) srcbmux(writedata, 32'b100, signimm, signimmsh,
                             alusrcb, srcb);
  alu alu(srca, srcb, alucontrol,
                     aluresult, zero);
  flopr #(32) alureg(clk, reset, aluresult, aluout);
  mux3 #(32) pcmux(aluresult, aluout,
                             {pc[31:28], instr[25:0], 2'b00}, pcsrc, pcnext);

endmodule
```

Generic Building Blocks

This section contains generic building blocks that may be useful in any MIPS microarchitecture, including a left shift unit, sign-extension unit, resettable flip-flop, and multiplexer.

Left Shift Unit (Multiply by 4)

Left shifters move bits and multiply by powers of 2. As the name implies, a left shifter shifts a binary number left by a specified number of positions.

Ex: 11001 LSR 2 = 00110; 11001 LSL 2 = 00100

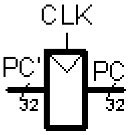
Shifts the number to the left (LSL) and fills empty spots with 0's.

s12.v

```
module sl2(input  [31:0] a, output [31:0] y);
	assign y = {a[29:0], 2'b00};  // shift left by 2
endmodule
```

Resettable Flip Flop

A resettable flip-flop adds another input called RESET. When RESET is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When RESET is TRUE, the resettable flip-flop ignores D and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e., 0) into all the flip-flops in a system when we first turn it on. Such flip-flops may be synchronously or asynchronously resettable. Synchronously resettable flip-flops reset themselves only on the rising edge of CLK. Asynchronously resettable flip-flops reset themselves as soon as RESET becomes TRUE, independent of CLK.

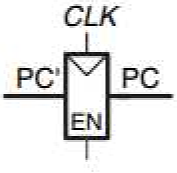


.

Symbol for the resettable flip-flop

Resettable Flip Flop with Enable

This type of flop register is similar to the simple flop register but requires enable input to be high to work.



Symbol for resettable flip-flop with enable

Verilog code for flip-flop:

flopenr.v

```
module flopenr #(parameter WIDTH = 8)
             (input clk, reset,
              input en,
              input [WIDTH-1:0] d,
              output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)
         if (reset)          q <= 0;
         else if (en)        q <= d;
endmodule
```

flopr.v

```
module flopr #(parameter WIDTH = 8)
        (input clk, reset,
        input [WIDTH-1:0] d,
        output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)
        if (reset)      q <= 0;
        else            q <= d;
endmodule
```

Sign Extension

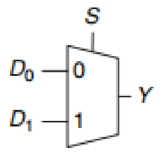
Sign extension is the operation, in computer arithmetic, of increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value. This is done by appending digits to the most significant side of the number, following a procedure dependent on the particular signed number representation used. For example, if six bits are used to represent the number "00 1010" (decimal positive 10) and the sign extend operation increases the word length to 16 bits, then the new representation is simply "0000 0000 0000 1010". Thus, both the value and the fact that the value was positive are maintained. If ten bits are used to represent the value "11 1111 0001" (decimal negative 15) using two's complement, and this is sign extended to 16 bits, the new representation is "1111 1111 1111 0001". Thus, by padding the left side with ones, the negative sign and the value of the original number are maintained.

signext.v

```
module signext(input  [15:0] a, output [31:0] y);
        assign y = {{16{a[15]}}, a};
endmodule
```

Multiplexers

Multiplexers are among the most commonly used combinational circuits. They choose an output from among several possible inputs based on the value of a selected signal. A multiplexer is sometimes affectionately called a mux.

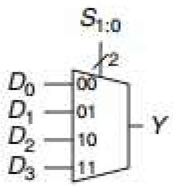


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

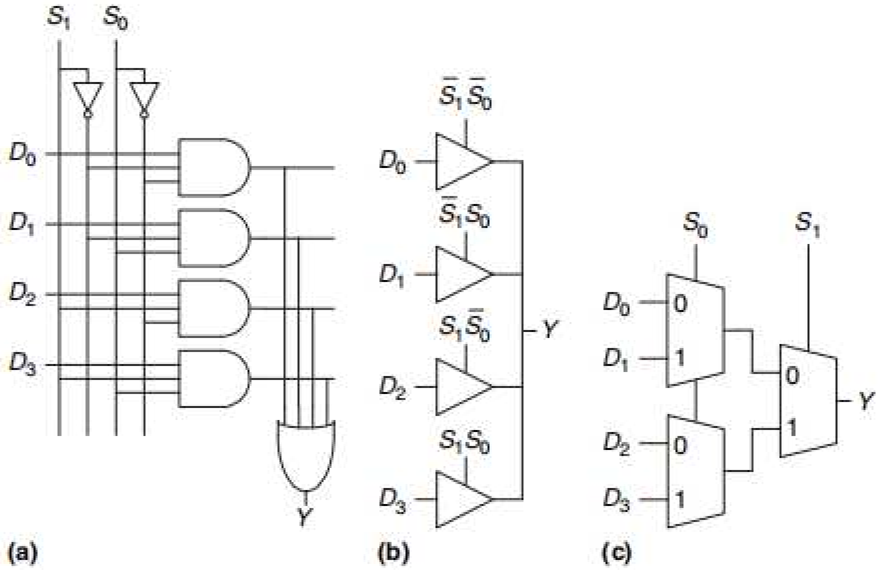
2:1 multiplexer
symbol and truth table

Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output. Two select signals are needed to choose among the four data inputs. The 4:1 multiplexer can be built using sum-of-products logic, tristates, or multiple 2:1 multiplexers.



4:1 Multiplexer



4:1 multiplexer implementations: (a) two-level logic, (b) tristates, (c) hierarchical

mux2.v

```
module mux2 #(parameter WIDTH = 8)
          (input [WIDTH-1:0] d0, d1,
          input s,
          output [WIDTH-1:0] y);
          assign y = s ? d1 : d0;
endmodule
```

mux3.v

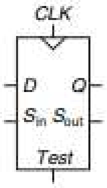
```
module mux3 #(parameter WIDTH = 8)
          (input [WIDTH-1:0] d0, d1, d2,
          input [1:0] s,
          output [WIDTH-1:0] y);
          assign #1 y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

mux4.v

```
module mux4 #(parameter WIDTH = 8)
          (input [WIDTH-1:0] d0, d1, d2, d3,
          input [1:0] s,
          output reg [WIDTH-1:0] y);

always @( * )
          case(s)
                    2'b00: y <= d0;
                    2'b01: y <= d1;
                    2'b10: y <= d2;
                    2'b11: y <= d3;
          endcase
endmodule
```

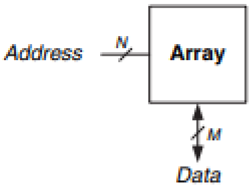
Memory

Memory refers to a device that is used to store information for immediate use in a computer or computer-related hardware device. A single cell memory element is just a single scannable flip flop.

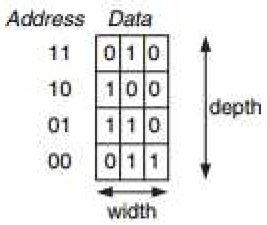


Scannable flip flop

A memory unit can be made by creating an array of flip flops that can be read and written. The following figure shows the basic symbol for the memory.



Generic Memory Symbol



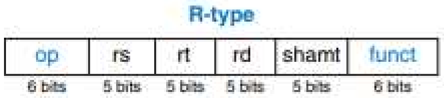
Memory Array Function

The memory size can be determined by multiplying its width and depth. In this case, the memory unit above can store 12 bits of data.

Memory Organization

The memory can read and write data. Each block of memory has its own address that is used to locate the data. The memory is composed of inputs: clock, for timing, write enable latch, address bits for locating data, write data, the data needed to be stored. The output is only read data that outputs data based on the current address bit input.

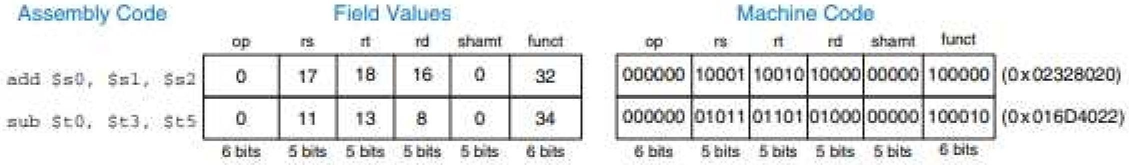
In a multicycle processor, the memory is shared. Both instructions and data share the same memory block. A single memory block is a machine code that is derived from the assembly code of the program. To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all formats start with a 6-bit opcode field. Thus, the best place to begin is to look at the opcode. If it is 0, the instruction is R-type; otherwise it is I-type or J-type.



R-type

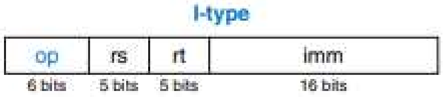
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-type machine instruction format



Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
add $s0, $s1, $s2	0	17	18	16	0	32	000000	10001	10010	10000	00000	100000	(0x02328020)
sub $t0, $t3, $t5	0	11	13	8	0	34	000000	01011	01101	01000	00000	100010	(0x016D4022)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

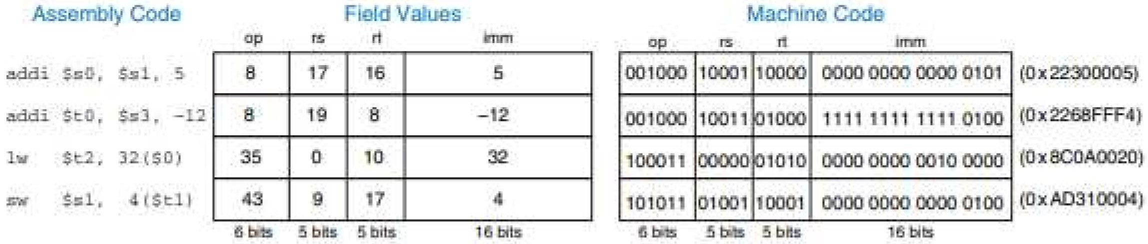
Machine code for R-type instruction



I-type

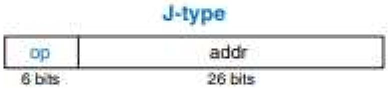
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

I-type instruction format



Assembly Code	Field Values				Machine Code				
	op	rs	rt	imm	op	rs	rt	imm	
addi $s0, $s1, 5	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
addi $t0, $s3, -12	8	19	8	−12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
lw $t2, 32($0)	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
sw $s1, 4($t1)	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	

Machine Code for I-type instruction



J-type

op	addr
6 bits	26 bits

.J-Type instruction format

The following verilog code of the memory already contains stored program from testing.

mem.v

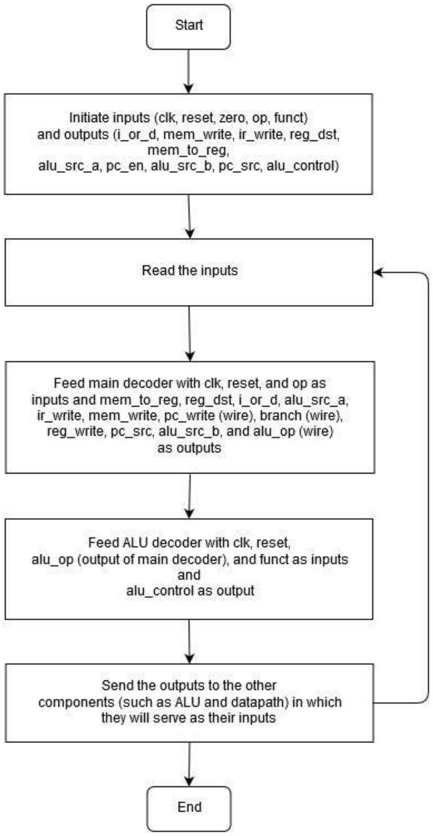
```
module mem(input clk, we,
        input  [31:0] a, wd,
        output [31:0] rd);

reg  [31:0] RAM[63:0];

initial
    begin
        RAM[0] <= 32'h20020005;
        RAM[1] <= 32'h2003000c;
        RAM[2] <= 32'h2067fff7;
        RAM[3] <= 32'h00e22025;
        RAM[4] <= 32'h00642824;
        RAM[5] <= 32'h00a42820;
        RAM[6] <= 32'h10a7000a;
        RAM[7] <= 32'h0064202a;
        RAM[8] <= 32'h10800001;
        RAM[9] <= 32'h20050000;
        RAM[10] <= 32'h00e2202a;
        RAM[11] <= 32'h00853820;
        RAM[12] <=  32'h00e23822;
        RAM[13] <=  32'hac670044;
        RAM[14] <=  32'h8c020050;
        RAM[15] <=  32'h08000011;
        RAM[16] <=  32'h20020001;
        RAM[17] <=  32'hac020054;
    end

assign rd = RAM[a[31:2]]; // word aligned

always @(posedge clk)
    if (we)
    RAM[a[31:2]] <= wd;
endmodule
```

Control Unit

The control unit is the part of the computer's central processing unit (CPU), which directs the operation of the processor. It was included as part of the von Neumann architecture by John von Neumann.

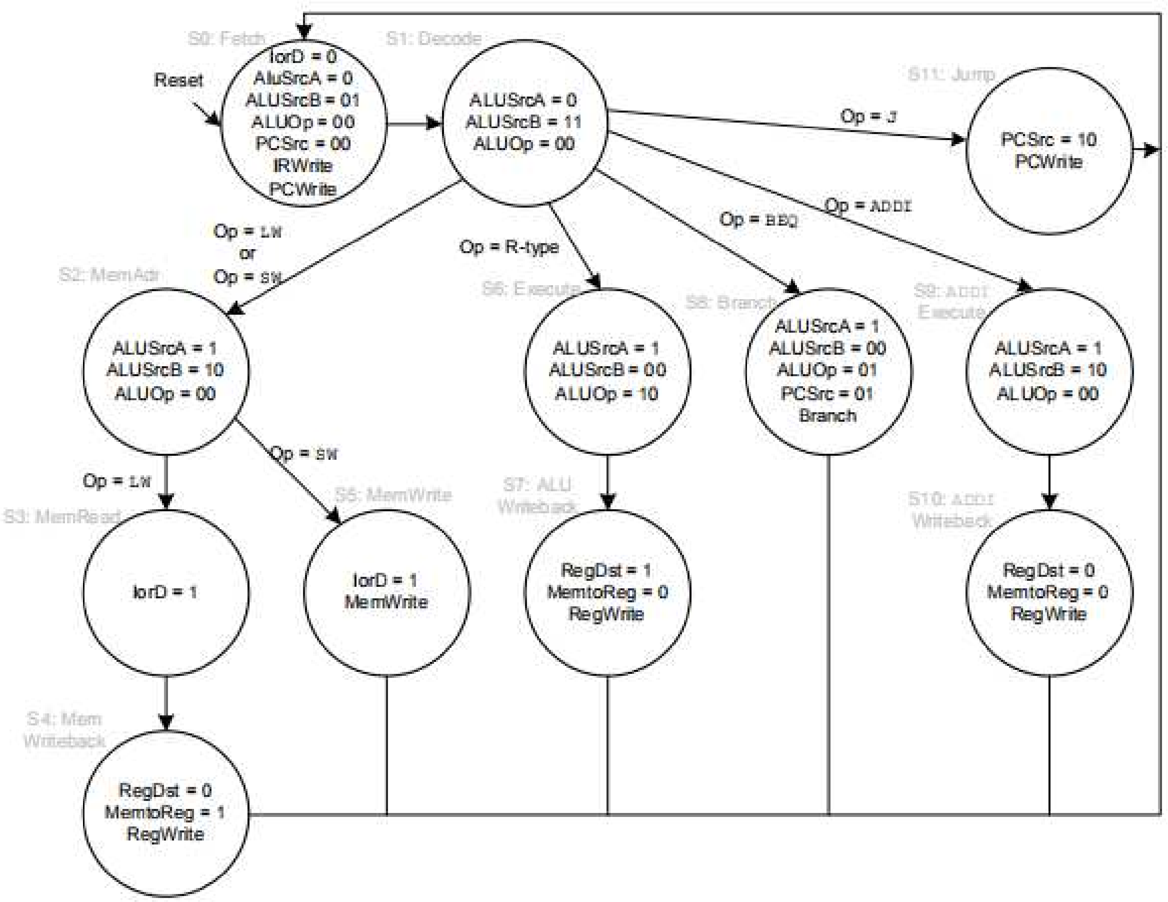
It is the responsibility of the control unit to tell the computer's memory, arithmetic/logic unit (ALU), and input/output (I/O) devices how to respond to the instructions that have been sent to the processor. It fetches internal instructions of the programs from the main memory to the processor instruction register, and based on this register contents, the control unit generates a control signal that supervises the execution of these instructions.

The following figure depicts the flow of the control unit:


Start
Initiate inputs (clk, reset, zero, op, funct) and outputs (i_or_d, mem_write, ir_write, reg_dst, mem_to_reg, alu_src_a, pc_en, alu_src_b, pc_src, alu_control)
Read the inputs
Feed main decoder with clk, reset, and op as inputs and mem_to_reg, reg_dst, i_or_d, alu_src_a, ir_write, mem_write, pc_write (wire), branch (wire), reg_write, pc_src, alu_src_b, and alu_op (wire) as outputs
Feed ALU decoder with clk, reset, alu_op (output of main decoder), and funct as inputs and alu_control as output
Send the outputs to the other components (such as ALU and datapath) in which they will serve as their inputs
End

The controller will have two major modules, the main decoder and the ALU decoder. The controller will read inputs from the top level in which it will provide control signals as outputs which are processed by the main decoder and the ALU decoder. The main decoder uses opcodes as input while the ALU decoder uses functions as input. The alu_op output port of the main decoder will be used as an input port by the ALU decoder. A more detailed operation on how the controller works will be discussed in the state diagram.

The following is a state diagram of the control unit:



Switching reset to 1 will allow the current state to go back to the fetch state. Switching reset to 0 will allow the state transition from state to state depending on the current state. For example, when the state is on fetch, the next state will be the decode state. The opcodes will also determine the next states the current state will take. The decode state may take one of the five states depending on the opcode (considering reset stays at 0): memory address state (for load word and store word), execute state (for R-types), branch state (for branch equal), addition execute state (for addition), and jump state (for jump). Each state will eventually go back to the fetch state.

The following is a state table for the control unit. Each bit of a state will represent each output for the controller:

State	Control Word
S0: FETCH	1010_00000_0100_00
S1: DECODE	0000_00000_1100_00
S2: MEMADR	0000_10000_1000_00
S3: MEMRD	0000_00100_0000_00
S4: MEMWB	0001_00010_0000_00
S5: MEMWR	0100_00100_0000_00
S6: EXECUTE	0000_10000_0000_10
S7: ALUWRITEBACK	0001_00001_0000_00
S8: BRANCH	0000_11000_0001_01
S9: ADDIEXECUTE	0000_10000_1000_00
S10: ADDIWRITEBACK	0001_00000_0000_00
S11: JUMP	1000_00000_0010_00

The control unit is separated into three components: the main controller unit, the main decoder, and the ALU decoder. The code for each segment is provided below:

Controller

controller.v

```
// The main controller produces multiplexer select and register enable
// signals for the datapath. The select signals are MemtoReg, RegDst,
// IorD, PCSrc, ALUSrcB, and ALUSrcA. The enable signals are IRWrite,
// MemWrite, PCWrite, Branch, and RegWrite.
module controller(input clk, reset,
                  input [5:0] op, funct,
                  input zero,
                  output pcen, memwrite, irwrite, regwrite,
                  output alusrca, iord, memtoreg, regdst,
                  output [1:0] alusrcb,
                  output [1:0] pcsrc,
                  output [2:0] alucontrol);

wire [1:0] aluop;
wire branch, pcwrite;

// Main Decoder and ALU Decoder subunits.
maindec md(clk, reset, op,
                  pcwrite, memwrite, irwrite, regwrite,
                  alusrca, branch, iord, memtoreg, regdst,
                  alusrcb, pcsrc, aluop);
aludec ad(funct, aluop, alucontrol);

assign pcen = pcwrite | (branch & zero);

endmodule
```

Main Decoder

maindec.v

```
// The controller receives the current instruction from the datapath
// and tell the datapath how to execute that instruction.
module maindec(input clk, reset,
               input [5:0] op,
               output      pcwrite, memwrite, irwrite, regwrite,
               output      alusrca, branch, iord, memtoreg, regdst,
               output [1:0] alusrcb,
               output [1:0] pcsrc,
               output [1:0] aluop);

// FSM States
parameter   FETCH                      = 5'b00000;    // State 0
parameter   DECODE                     = 5'b00001;    // State 1
parameter   MEMADR                     = 5'b00010;    // State 2
parameter   MEMRD                  = 5'b00011;    // State 3
parameter   MEMWB                      = 5'b00100;    // State 4
parameter   MEMWR                      = 5'b00101;    // State 5
parameter   EXECUTE                    = 5'b00110;    // State 6
parameter   ALUWRITEBACK               = 5'b00111;    // State 7
parameter   BRANCH                     = 5'b01000;    // State 8
parameter   ADDIEXECUTE                = 5'b01001;    // State 9
parameter   ADDIWRITEBACK              = 5'b01010;    // state a
parameter   JUMP                   = 5'b01011;    // State b

// MIPS Instruction Opcodes
parameter   LW       = 6'b100011;   // load word lw
parameter   SW       = 6'b101011;   // store word sw
parameter   RTYPE    = 6'b000000;   // R-type
parameter   BEQ      = 6'b000100;   // branch if equal beq
parameter   ADDI     = 6'b001000;   // add immediate addi
parameter   J        = 6'b000010;   // jump j

reg [4:0]  state, nextstate;
reg [16:0] controls;
```

```
// state register
always @(posedge clk or posedge reset)
        if(reset) state <= FETCH;
        else      state <= nextstate;



// next state logic
always @( * )
        case(state)
                FETCH:   nextstate <= DECODE;
                DECODE:  case(op)
                        LW:             nextstate <= MEMADR;
                        SW:             nextstate <= MEMADR;
                        RTYPE:          nextstate <= EXECUTE;
                        BEQ:            nextstate <= BRANCH;
                        ADDI:  nextstate <= ADDIEXECUTE;
                        J:              nextstate <= JUMP;
                        default: nextstate <= FETCH;  // should never happen
                endcase
                MEMADR:  case(op)
                                LW:      nextstate <= MEMRD;
                                SW:      nextstate <= MEMWR;
                                default: nextstate <= FETCH; // should never happen
                        Endcase
                MEMRD:   nextstate <= MEMWB;
                MEMWB:   nextstate <= FETCH;
                MEMWR:   nextstate <= FETCH;
                EXECUTE: nextstate <= ALUWRITEBACK;
                ALUWRITEBACK: nextstate <= FETCH;
                BRANCH:   nextstate <= FETCH;
                ADDIEXECUTE:  nextstate <= ADDIWRITEBACK;
                ADDIWRITEBACK:  nextstate <= FETCH;
                JUMP:    nextstate <= FETCH;
                default: nextstate <= FETCH;  // should never happen
        endcase
// output logic
assign {pcwrite, memwrite, irwrite, regwrite,
        alusrca, branch, iord, memtoreg, regdst,
        alusrcb, pcsrc, aluop} = controls;
```

```
always @( * )
    case(state)
        FETCH: controls <= 19'b1010_00000_0100_00;
        DECODE: controls <= 19'b0000_00000_1100_00;
        MEMADR: controls <= 19'b0000_10000_1000_00;
        MEMRD: controls <= 19'b0000_00100_0000_00;
        MEMWB: controls <= 19'b0001_00010_0000_00;
        MEMWR: controls <= 19'b0100_00100_0000_00;
        EXECUTE: controls <= 19'b0000_10000_0000_10;
        ALUWRITEBACK: controls <= 19'b0001_00001_0000_00;
        BRANCH: controls <= 19'b0000_11000_0001_01;
        ADDIEXECUTE: controls <= 19'b0000_10000_1000_00;
        ADDIWRITEBACK: controls <= 19'b0001_00000_0000_00;
        JUMP: controls <= 19'b1000_00000_0010_00;
        default: controls <= 19'b0000_xxxxx_xxxx_xx; // should never happen
    endcase
endmodule
```

ALU Decoder

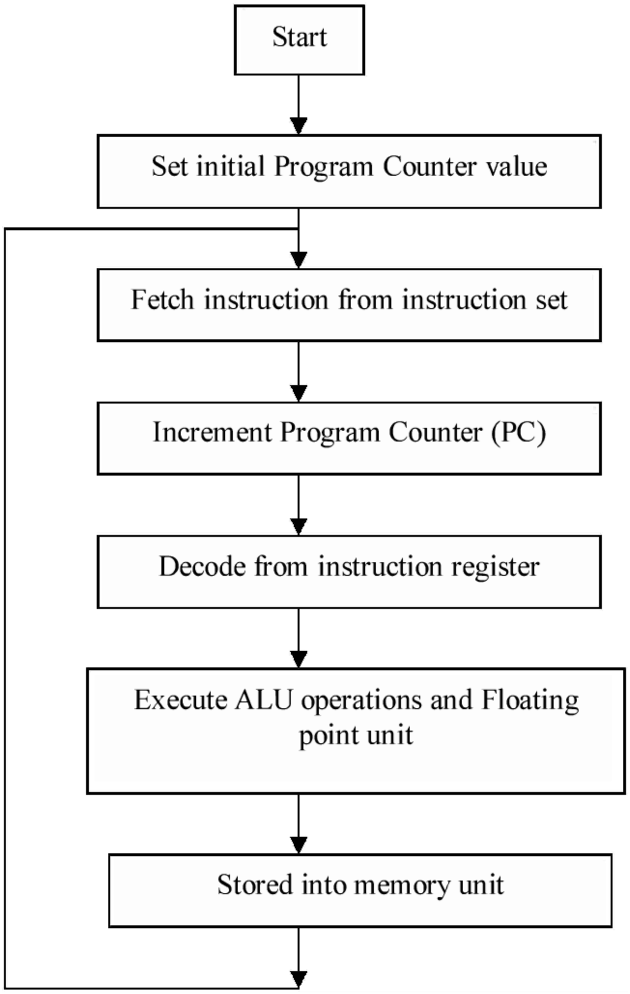
```
module aludec(input [5:0] funct,
        input [1:0] aluop,
        output reg [2:0] alucontrol);
always @( * )
    case(aluop)
        3'b000: alucontrol <= 3'b010;  // add
        3'b001: alucontrol <= 3'b010;  // sub
        // RTYPE instruction use the 6-bit funct field of instruction to specify ALU
        operation
        3'b010: case(funct)
            6'b100000: alucontrol <= 3'b010; // ADD
            6'b100010: alucontrol <= 3'b110; // SUB
            6'b100100: alucontrol <= 3'b000; // AND
            6'b100101: alucontrol <= 3'b001; // OR
            6'b101010: alucontrol <= 3'b111; // SLT
            default:   alucontrol <= 3'bxxx; // ???
        endcase
        default: alucontrol <= 3'bxxx; // ???
    endcase
endmodule
```

Registers

A register is, in general terms, a quick-access storage location for a processor. It usually comprises of a small amount of storage, and each of these storage parts may be either read-only or write-only.

Registers are able to store light data for an amount of time, and can serve as an interface which allows the CPU to communicate with the I/O and memory components.

To demonstrate the flow of processes of a register, a flowchart is illustrated below:


Start
Set initial Program Counter value
Fetch instruction from instruction set
Increment Program Counter (PC)
Decode from instruction register
Execute ALU operations and Floating point unit
Stored into memory unit

The Register Set

The MIPS architecture defines 32 registers. Each register has a name and a number ranging from 0 to 31. The following table lists the name, number, and use for each register. $0 always contains the value 0 because this constant is so frequently used in computer programs.

Name	**Number**	**Use**
$0	0	the constant value 0
$at	1	assembler temporary
$v0–$v1	2–3	function return value
$a0–$a3	4–7	function arguments
$t0–$t7	8–15	temporary variables
$s0–$s7	16–23	saved variables
$t8–$t9	24–25	temporary variables
$k0–$k1	26–27	operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

regfile.v

```
module regfile(input clk,
                    input we3,
                    input  [4:0]  ra1, ra2, wa3,
                    input  [31:0] wd3,
                    output [31:0] rd1, rd2);

reg [31:0] rf[31:0];

// three ported register file
// read two ports combinationally
// write third port on rising edge of clock
// register 0 hardwired to 0

always @(posedge clk)
          if (we3) rf[wa3] <= wd3;

          assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
          assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

Interfacing the Components (Top-Level)

The following code connects the datapath, alu and control unit to create the processor.

mips.v

```
// Multicycle MIPS processor
module mips(input clk, reset,
                    output [31:0] adr, writedata,
                    output memwrite,
                    input  [31:0] readdata);

wire zero, pcen, irwrite, regwrite,
          alusrca, iord, memtoreg, regdst;
wire [1:0]  alusrcb;
wire [1:0]  pcsrc;
wire [2:0]  alucontrol;
wire [5:0]  op, funct;
```

```
// The control unit receives the current instruction from the datapath and tells the
// datapath how to execute that instruction.

controller c(clk, reset, op, funct, zero,
                    pcen, memwrite, irwrite, regwrite,
                    alusrca, iord, memtoreg, regdst,
                    alusrcb, pcsrc, alucontrol);

// The datapath operates on words of data. It
// contains structures such as memories, registers, ALUs, and multiplexers.
// MIPS is a 32-bit architecture, so we will use a 32-bit datapath.
datapath dp(clk, reset,
                    pcen, irwrite, regwrite,
                    alusrca, iord, memtoreg, regdst,
                    alusrcb, pcsrc, alucontrol,
                    op, funct, zero,
                    adr, writedata, readdata);
endmodule
```

The next code connects the processor to the memory to work.

topmulti.v

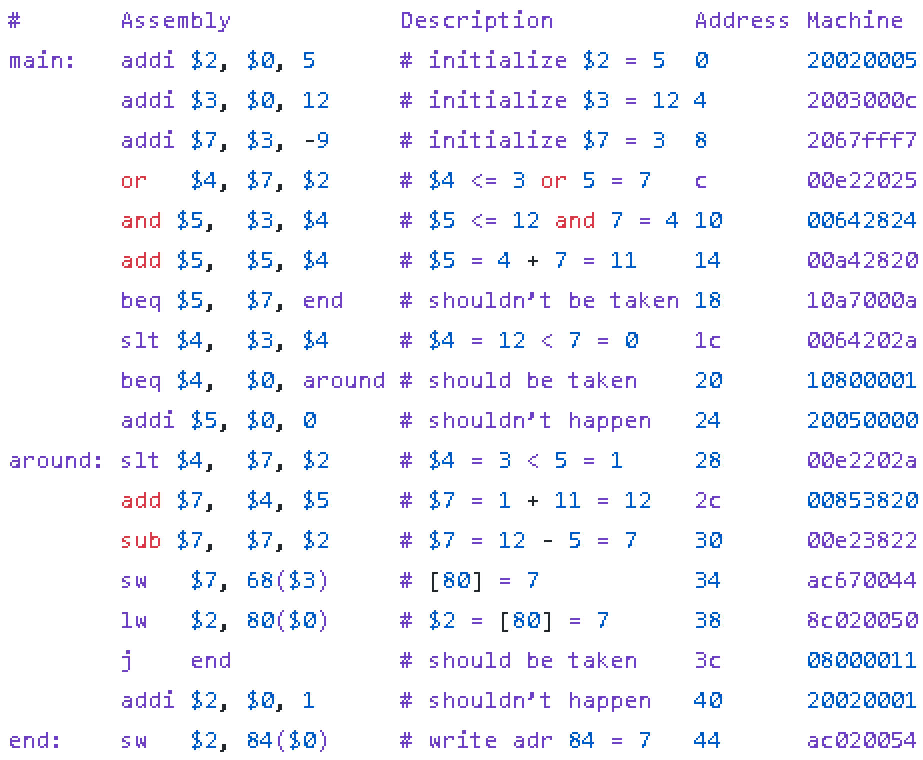
```
module topmulti(input clk, reset,
                    output [31:0] writedata, adr,
                    output memwrite);

wire [31:0] readdata;

// instantiate processor and memory
mips mips(clk, reset, adr, writedata, memwrite, readdata);
mem mem(clk, memwrite, adr, writedata, readdata);

endmodule
```

VI. TESTING & EVALUATION

The following tests were made first on each component to see if each one works, and then they were all tested as a top-level multi-cycle processor in order to verify that the entire program works correctly as expected.

Top Level

To test the correctness of the entire multicycle processor, the following assembly code is used.



```
#       Assembly             Description          Address Machine
main:   addi $2, $0, 5       # initialize $2 = 5  0       20020005
        addi $3, $0, 12      # initialize $3 = 12 4       2003000c
        addi $7, $3, -9      # initialize $7 = 3  8       2067fff7
        or   $4, $7, $2      # $4 <= 3 or 5 = 7   c       00e22025
        and $5,  $3, $4      # $5 <= 12 and 7 = 4 10      00642824
        add $5,  $5, $4      # $5 = 4 + 7 = 11    14      00a42820
        beq $5,  $7, end     # shouldn't be taken 18      10a7000a
        slt $4,  $3, $4      # $4 = 12 < 7 = 0    1c      0064202a
        beq $4,  $0, around  # should be taken    20      10800001
        addi $5, $0, 0       # shouldn't happen   24      20050000
around: slt $4,  $7, $2      # $4 = 3 < 5 = 1     28      00e2202a
        add $7,  $4, $5      # $7 = 1 + 11 = 12   2c      00853820
        sub $7,  $7, $2      # $7 = 12 - 5 = 7    30      00e23822
        sw   $7, 68($3)      # [80] = 7           34      ac670044
        lw   $2, 80($0)      # $2 = [80] = 7      38      8c020050
        j    end             # should be taken    3c      08000011
        addi $2, $0, 1       # shouldn't happen   40      20020001
end:    sw   $2, 84($0)      # write adr 84 = 7   44      ac020054
```

All instructions add, sub, and, or, slt, addi, lw, sw, beq, j are used in the program. If successful, it should write the value 7 to address 84_D (44_H).

Recall that memory is initialized with the equivalent machine code to run the program. The machine code holds both instruction and data and are decoded by the control unit and the register file.

The following code is the testbench of the program:

testbench.v

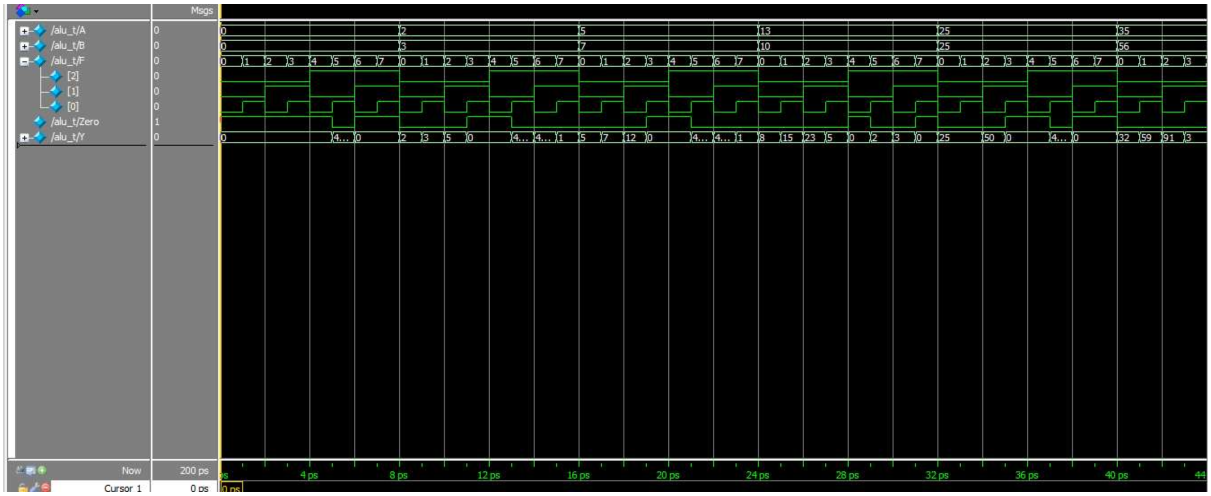
```
module testbench();
reg clk;
reg reset;

wire [31:0] writedata, dataadr;
wire memwrite;
// keep track of execution status
reg  [31:0] cycle;
// instantiate device to be tested
topmulti dut(clk, reset, writedata, dataadr, memwrite);
// initialize test
initial
    begin
        reset <= 1; # 12; reset <= 0;
        cycle <= 1;
    end
// generate clock to sequence tests
always
    begin
        clk <= 1; # 5; clk <= 0; # 5;
        cycle <= cycle + 1;
    end
// check results
// If successful, it should write the value 7 to address 84
always@(negedge clk)
    begin
    if (memwrite) begin
        if (dataadr === 84 & writedata == 7) begin
            $display("Simulation succeeded");
            $stop;
        end else if (dataadr !== 80) begin
            $display("Simulation failed");
            $stop;
        end
    end
end
endmodule
```

ALU

The testbench code for the ALU is as follows:

```
module alu_tb;
reg [31:0] A, B;
reg [2:0] F;
wire Zero;
wire [31:0] Y;

alu uut (A, B, F, Y, Zero);

initial
begin
A = 0;
B = 0;
F = 0;

#8 A = 2;       B = 3;
#8 A = 5;       B = 7;
#8 A = 13;      B = 10;
#8 A = 25;      B = 25;
#8 A = 35;      B = 56;
#8 A = 75;      B = 100;
#8 A = 200;   B = 155;
#8 A = 360;   B = 400;
#8 A = 1023;  B = 780;
#8 A = 1570;  B = 2047;
end

always
begin
#1 F = F + 1;
end

endmodule
```

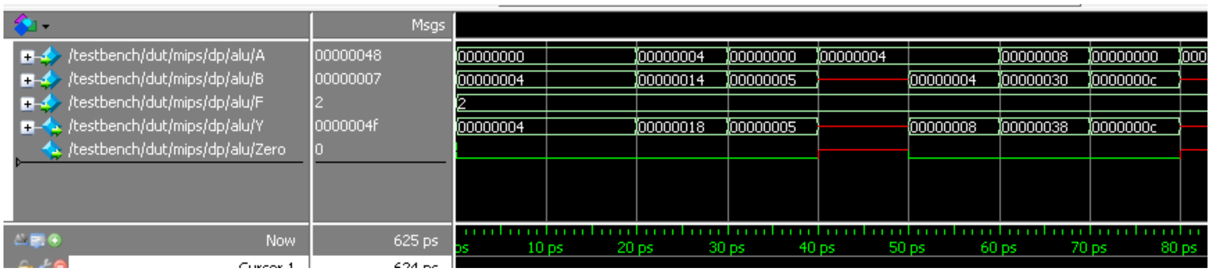
Output Waveform:



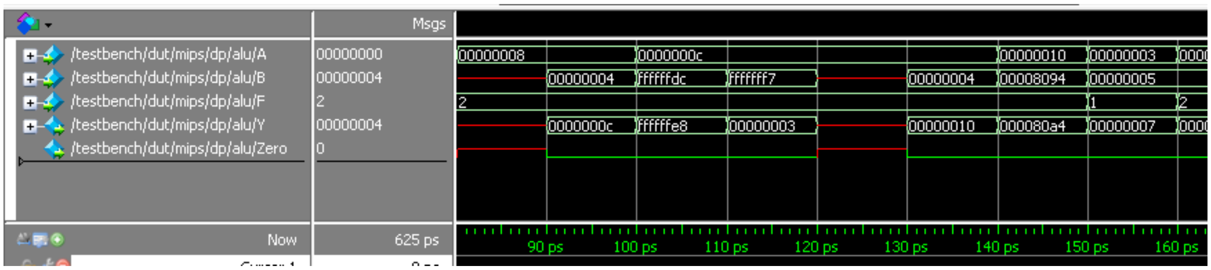
The waveform shows combinational logic given the primary inputs A and B and the selector F. The code shows how the output Y will behave based on the values given in A and B given the operation F (e.g. addition, and, or). The results of the waveform matches the expected manually calculated outputs.

Actual Waveform of ALU using MIPS Assembly Code:

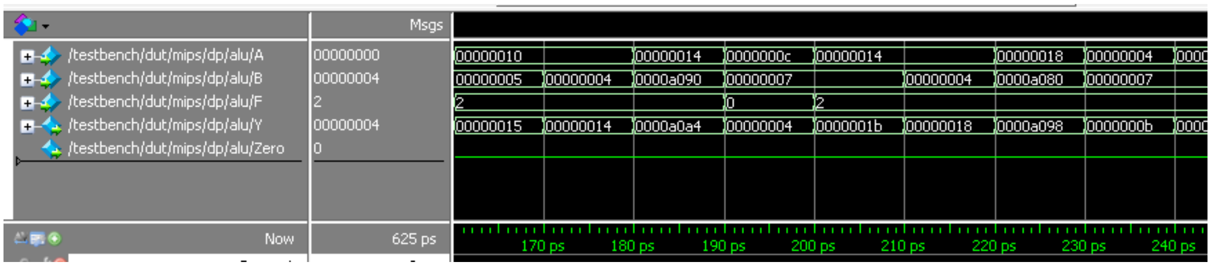
The program starts at cycle 2 after reset. The ALU arithmetic operation usually runs at ADDIEXEC and EXEC state. The cycle location of the ALU will be shown in the instruction trace. The ALU also used to increment the program counter by 4 at the FETCH state and then go to DECODE state which is iterative before and after instruction the result of the ALU is ignored at this state.


Msgs
/testbench/dut/mips/dp/alu/A 00000048
/testbench/dut/mips/dp/alu/B 00000007
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 0000004f
/testbench/dut/mips/dp/alu/Zero 0
00000000
00000004
00000000
00000004
00000008
00000000
00000004
00000014
00000005
00000004
00000030
0000000c
2
00000004
00000018
00000005
00000008
00000038
0000000c
Now 625 ps
ps 10 ps 20 ps 30 ps 40 ps 50 ps 60 ps 70 ps 80 ps

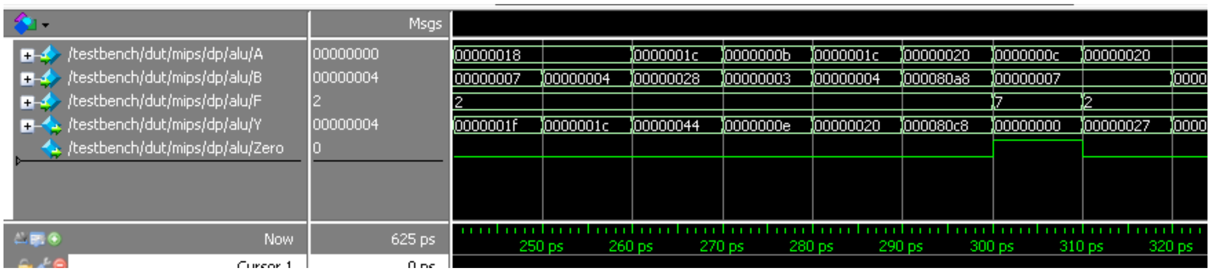
initialized \$2 = 5 at 30ps, initialized \$3 = 12 at 70ps.


Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
00000008
0000000c
00000010
00000003
00000004
ffffffdc
fffffff7
00000004
00008094
00000005
2
1
2
0000000c
ffffffe8
00000003
00000010
000080a4
00000007
Now 625 ps
90 ps 100 ps 110 ps 120 ps 130 ps 140 ps 150 ps 160 ps

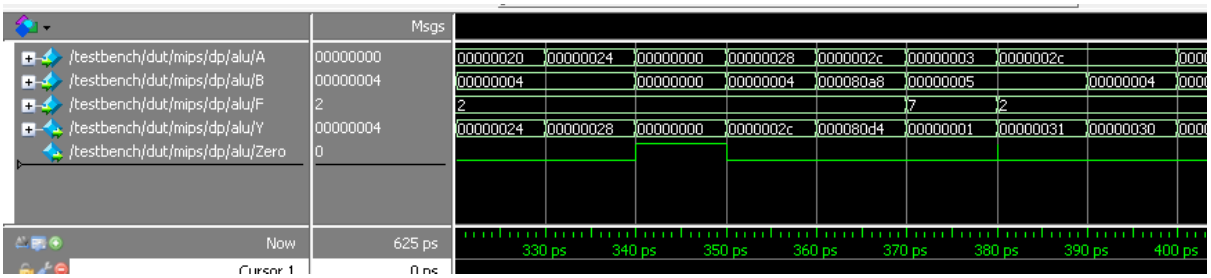
initialized \$7 = \$3 - 9 = 12 - 9 = 3 at 110ps. \$4 <= 3 or 5 = 7 at 150ps.


Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
00000010
00000014
0000000c
00000014
00000018
00000004
00000005
00000004
0000a090
00000007
00000004
0000a080
00000007
2
0
2
00000015
00000014
0000a0a4
00000004
0000001b
00000018
0000a098
0000000b
Now 625 ps
170 ps 180 ps 190 ps 200 ps 210 ps 220 ps 230 ps 240 ps

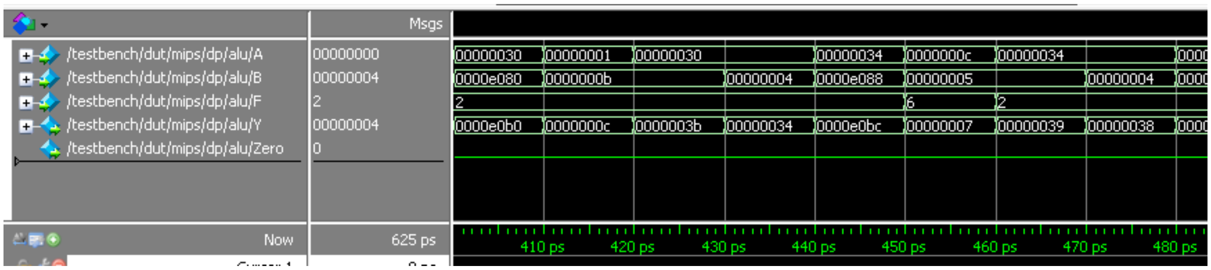
\$5 <= 12 and 7 = 4 at 190ps, \$5 = 4 + 7 = 11 at 230ps


Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
00000018
0000001c
0000000b
0000001c
00000020
0000000c
00000020
00000007
00000004
00000028
00000003
00000004
000080a8
00000007
2
7
2
0000001f
0000001c
00000044
0000000e
00000020
000080c8
00000000
00000027
Now 625 ps
Cursor 1 0 ps
250 ps 260 ps 270 ps 280 ps 290 ps 300 ps 310 ps 320 ps

\$4 = 12 < 7 = 0, zero = 1 at 300ps

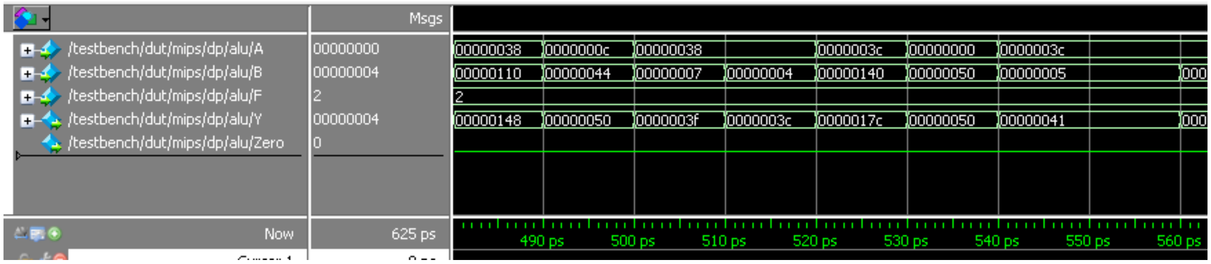

Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
00000020 00000024 00000000 00000028 0000002c 00000003 0000002c
00000004 00000000 00000004 000080a8 00000005 00000004
2 7 2
00000024 00000028 00000000 0000002c 000080d4 00000001 00000031 00000030
Now 625 ps
Cursor 1 0 ps
330 ps 340 ps 350 ps 360 ps 370 ps 380 ps 390 ps 400 ps

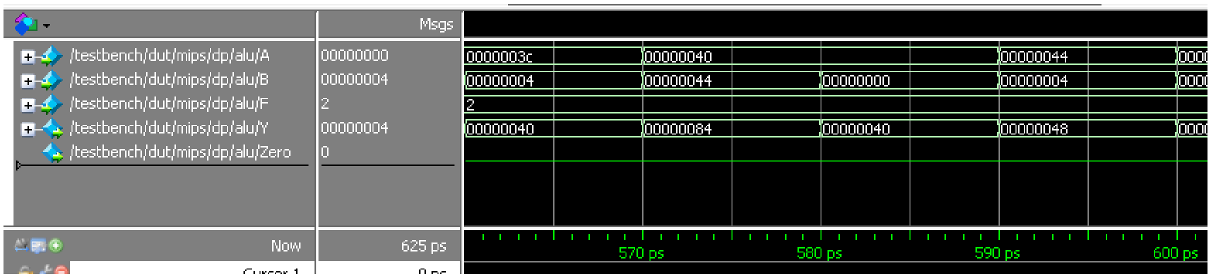
$4 = 3 < 5 = 1 at 370ps

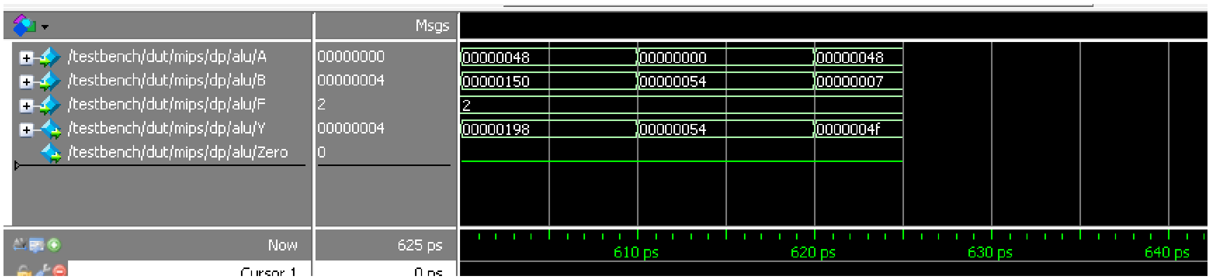

Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
00000030 00000001 00000030 00000034 0000000c 00000034
0000e080 0000000b 00000004 0000e088 00000005 00000004
2 6 2
0000e0b0 0000000c 0000003b 00000034 0000e0bc 00000007 00000039 00000038
Now 625 ps
410 ps 420 ps 430 ps 440 ps 450 ps 460 ps 470 ps 480 ps

$7 = 1 + 11 = 12 at 410ps, $7 = 12 - 5 = 7 at 450ps

The ALU is not used for after 450ps so we don't need the output from here.


Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
00000038 0000000c 00000038 0000003c 00000000 0000003c
00000110 00000044 00000007 00000004 00000140 00000050 00000005
2
00000148 00000050 0000003f 0000003c 0000017c 00000050 00000041
Now 625 ps
490 ps 500 ps 510 ps 520 ps 530 ps 540 ps 550 ps 560 ps


Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
0000003c 00000040 00000044
00000004 00000044 00000000 00000004
2
00000040 00000084 00000040 00000048
Now 625 ps
Cursor 1 0 ps
570 ps 580 ps 590 ps 600 ps


Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
00000048 00000000 00000048
00000150 00000054 00000007
2
00000198 00000054 0000004f
Now 625 ps
Cursor 1 0 ps
610 ps 620 ps 630 ps 640 ps

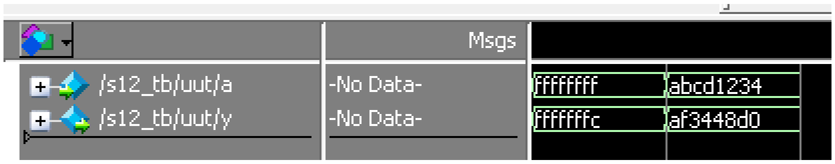
Datapath

The datapath is just composed of wires and cannot be tested without other components. The generic building blocks are tested instead. The testbench code and waveform for the datapath generic building blocks is as follows:

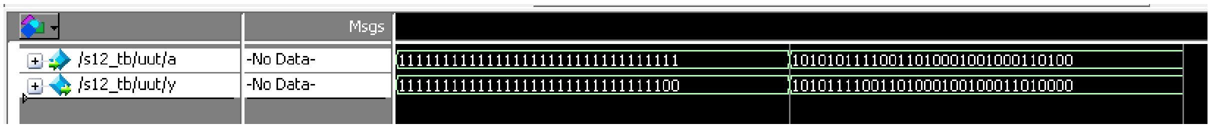
s12_tb.v

```
`timescale 1ps/1ps
module s12_tb;
        reg[31:0] a;
        wire[31:0] y;

        initial begin
                a = 32'hffffffff;
                #1 a = 32'habcd1234;
                #1 $stop;
        end
        sl2 uut(a, y);
endmodule
```

Output Waveform:



```
                     Msgs
/s12_tb/uut/a   -No Data-   ffffffff   abcd1234
/s12_tb/uut/y   -No Data-   fffffffc   af3448d0
```

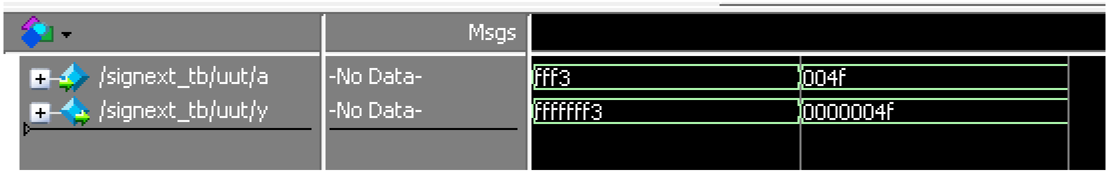


```
                     Msgs
/s12_tb/uut/a   -No Data-   11111111111111111111111111111111   10101011110011010001001000110100
/s12_tb/uut/y   -No Data-   11111111111111111111111111111100   10101111001101000100100011010000
```

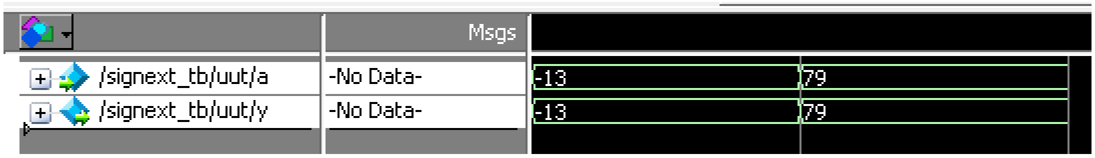
signext_tb.v

```
`timescale 1ps/1ps
module signext_tb;
        reg[15:0] a;
        wire[31:0] y;

        initial begin
                a = 32'hfff3;
                #1 a = 32'h004f;
                #1 $stop;
        end
        signext uut(a, y);
endmodule
```

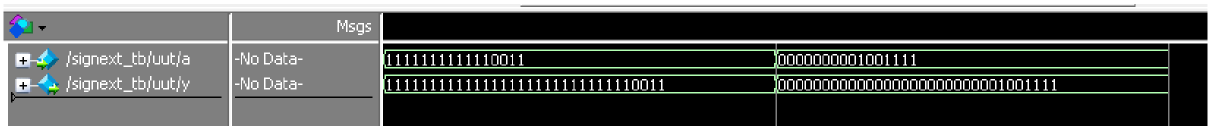
Output Waveform:



```
                    Msgs
/signext_tb/uut/a   -No Data-   fff3       004f
/signext_tb/uut/y   -No Data-   fffffff3   0000004f
```



```
                    Msgs
/signext_tb/uut/a   -No Data-   -13   79
/signext_tb/uut/y   -No Data-   -13   79
```

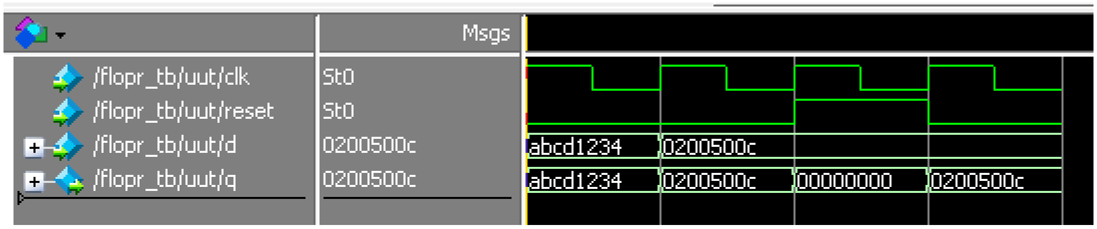


```
                    Msgs
/signext_tb/uut/a   -No Data-   1111111111110011                   0000000001001111
/signext_tb/uut/y   -No Data-   11111111111111111111111111110011   00000000000000000000000001001111
```

flopr_tb.v

```
`timescale 1ps/1ps
module flopr_tb;
        parameter WIDTH = 32;
        reg     clk, reset;
  reg   [WIDTH-1:0] d;
  wire  [WIDTH-1:0] q;

        always begin
                clk <= 1; #1; clk <= 0; #1;
        end
        initial begin
                d = 32'hABCD1234;
                reset = 0;
                #2;
                d = 32'h200500C;
                #2;
                reset = 1;
                #2;
                reset = 0;
                #2 $stop;
        end

        flopr #(WIDTH) uut(clk, reset, d, q);

endmodule
```

Output Waveform:

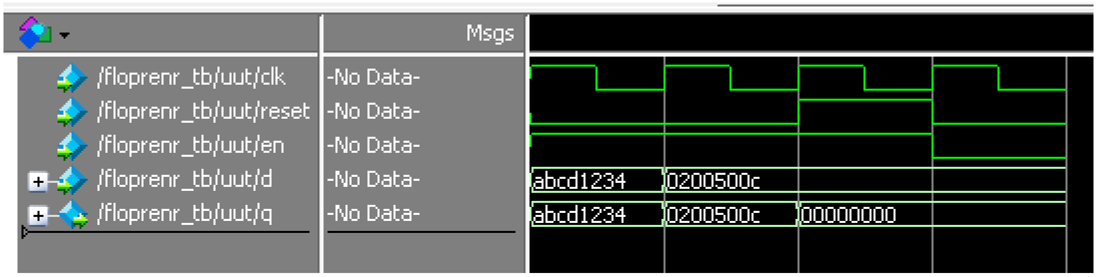


```
                     Msgs
/flopr_tb/uut/clk    St0
/flopr_tb/uut/reset  St0
/flopr_tb/uut/d      0200500c
/flopr_tb/uut/q      0200500c
```

flopenr_tb.v

```
`timescale 1ps/1ps
module floprenr_tb;
        parameter WIDTH = 32;
        reg     clk, reset, en;
  reg   [WIDTH-1:0] d;
  wire  [WIDTH-1:0] q;

        always begin
                clk <= 1; #1; clk <= 0; #1;
        end

        initial begin
                d = 32'hABCD1234;
                reset = 0;
                en = 1;
                #2;
                d = 32'h200500C;
                #2;
                reset = 1;
                #2;
                reset = 0;
                en = 0;
                #2 $stop;
        end
        flopenr #(WIDTH) uut(clk, reset, en, d, q);
Endmodule
```

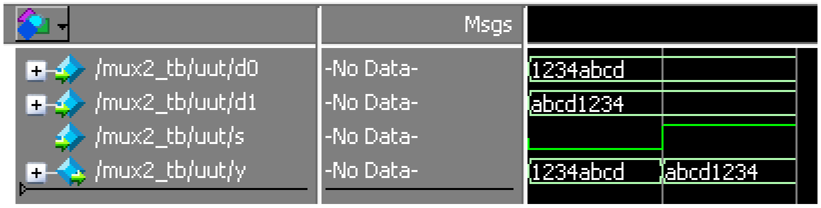
Output Waveform:



mux2_tb.v

```
`timescale 1ps/1ps

module mux2_tb;
        parameter WIDTH = 32;
        reg [WIDTH-1:0] d0, d1;
  reg s;
  wire [WIDTH-1:0] y;

        initial begin
                d0 = 32'h1234abcd;
                d1 = 32'habcd1234;
                s = 0;
                #2 $stop;
        end
        always #1 s = s + 1;
        mux2 #(WIDTH) uut(d0, d1, s, y);
endmodule
```

Output Waveform:

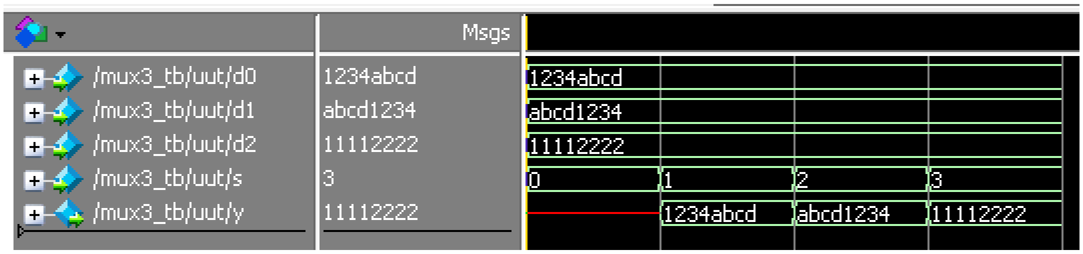


```
                     Msgs
/mux2_tb/uut/d0   -No Data-   1234abcd
/mux2_tb/uut/d1   -No Data-   abcd1234
/mux2_tb/uut/s    -No Data-
/mux2_tb/uut/y    -No Data-   1234abcd    abcd1234
```

mux3_tb.v

```
`timescale 1ps/1ps
module mux3_tb;
        parameter WIDTH = 32;
        reg [WIDTH-1:0] d0, d1, d2;
  reg [1:0]s;
  wire [WIDTH-1:0] y;

        initial begin
                d0 = 32'h1234abcd;
                d1 = 32'habcd1234;
                d2 = 32'h11112222;
                s = 0;
                #4 $stop;
        end

        always #1 s = s + 1;

        mux3 #(WIDTH) uut(d0, d1, d2, s, y);
endmodule
```

Output Waveform:

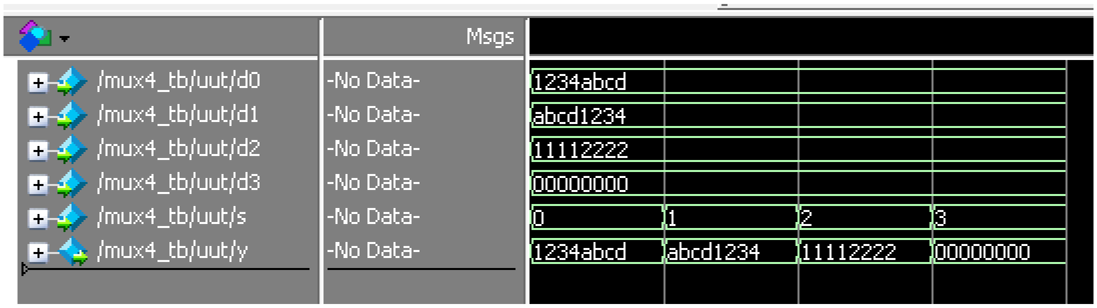


```
                      Msgs
/mux3_tb/uut/d0       1234abcd
/mux3_tb/uut/d1       abcd1234
/mux3_tb/uut/d2       11112222
/mux3_tb/uut/s        3
/mux3_tb/uut/y        11112222
```

mux4_tb.v

```
`timescale 1ps/1ps
module mux4_tb;
        parameter WIDTH = 32;
        reg [WIDTH-1:0] d0, d1, d2, d3;
  reg [2:0] s;
  wire [WIDTH-1:0] y;

        initial begin
                d0 = 32'h1234abcd;
                d1 = 32'habcd1234;
                d2 = 32'h11112222;
                d3 = 32'h0;
                s = 0;
                #4 $stop;
        end

        always #1 s = s + 1;

        mux4 #(WIDTH) uut(d0, d1, d2, d3, s, y);
endmodule
```

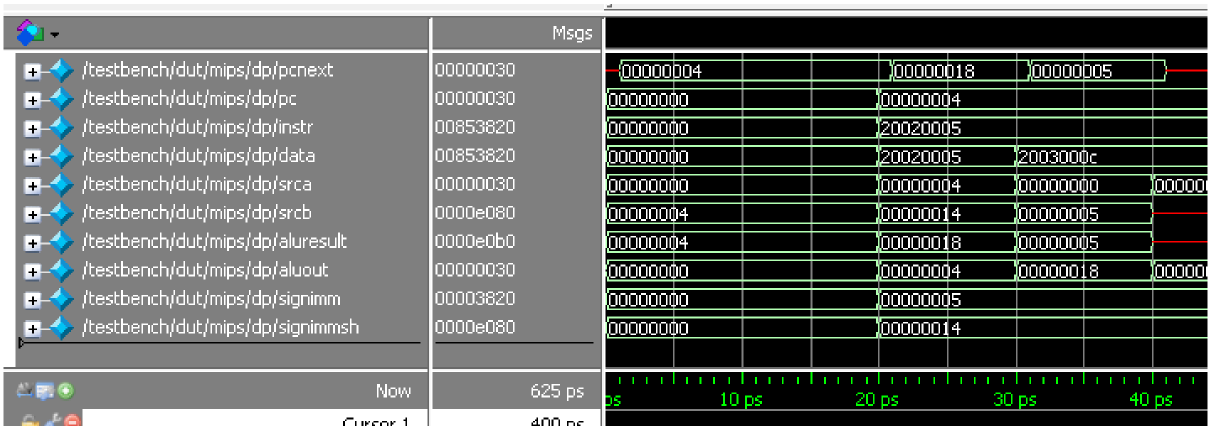
Output Waveform:



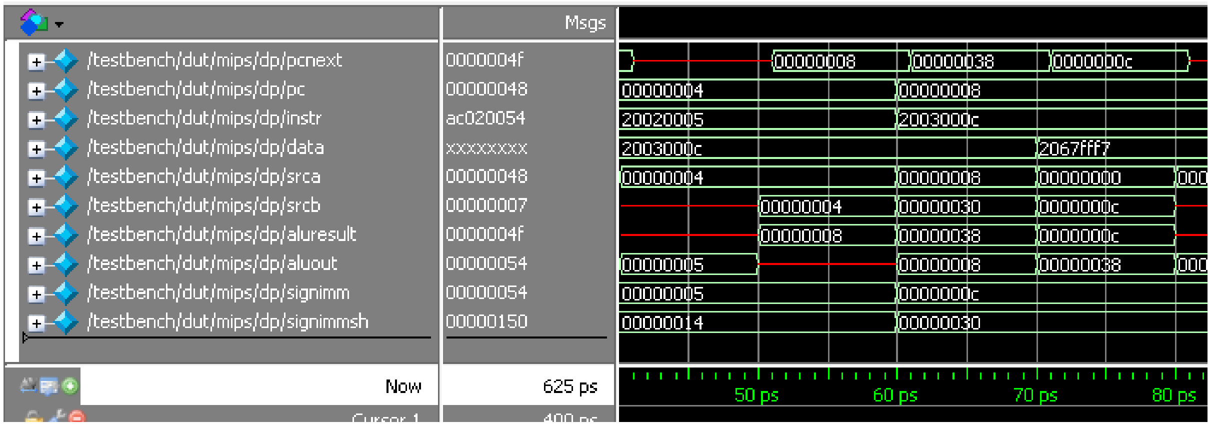
```
                       Msgs
/mux4_tb/uut/d0     -No Data-   1234abcd
/mux4_tb/uut/d1     -No Data-   abcd1234
/mux4_tb/uut/d2     -No Data-   11112222
/mux4_tb/uut/d3     -No Data-   00000000
/mux4_tb/uut/s      -No Data-   0          1          2          3
/mux4_tb/uut/y      -No Data-   1234abcd   abcd1234   11112222   00000000
```

Actual Waveform of Datapath using MIPS Assembly Code:

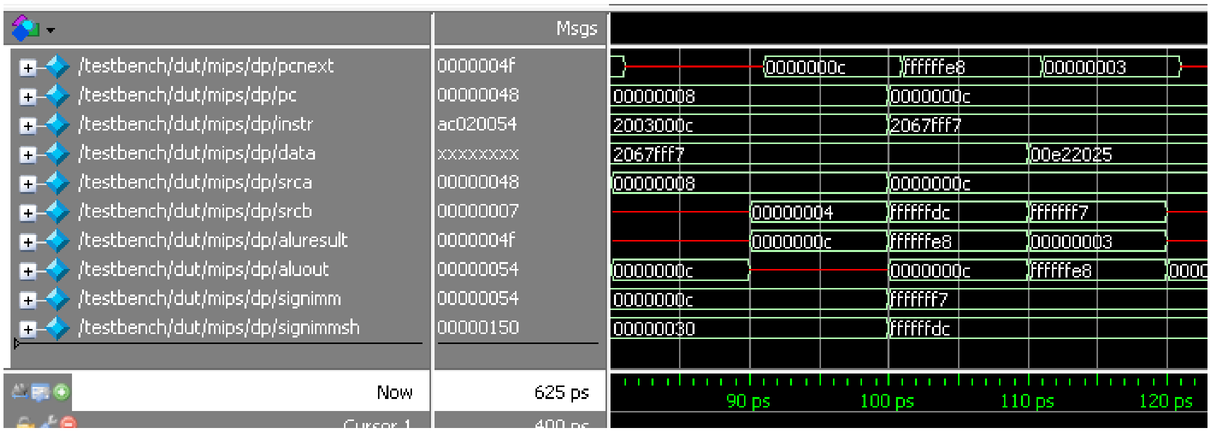
The datapath elements is just the same as the ALU, memory and control unit except the program counter that handles the program flow and the signimm and signimmsh which sign extend the data.



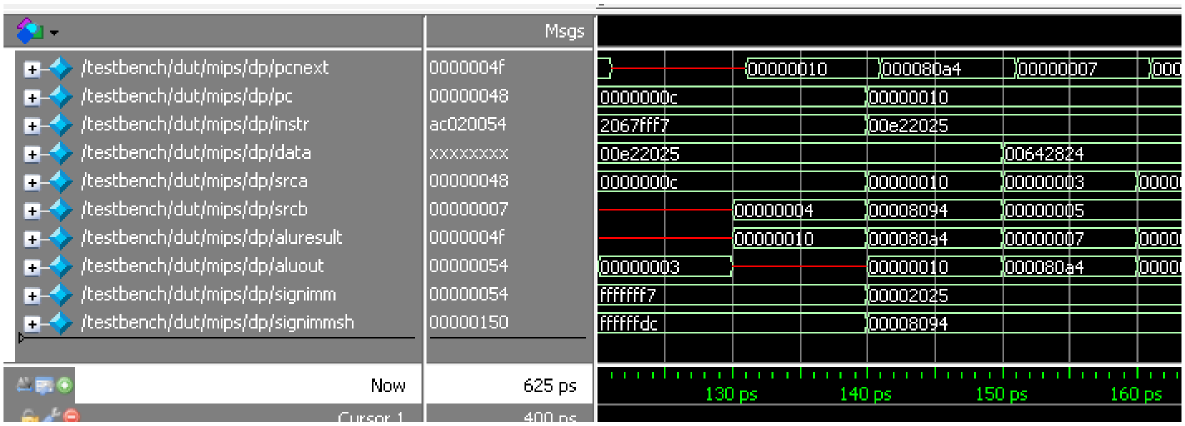
	Msgs
/testbench/dut/mips/dp/pcnext	00000030
/testbench/dut/mips/dp/pc	00000030
/testbench/dut/mips/dp/instr	00853820
/testbench/dut/mips/dp/data	00853820
/testbench/dut/mips/dp/srca	00000030
/testbench/dut/mips/dp/srcb	0000e080
/testbench/dut/mips/dp/aluresult	0000e0b0
/testbench/dut/mips/dp/aluout	00000030
/testbench/dut/mips/dp/signimm	00003820
/testbench/dut/mips/dp/signimmsh	0000e080
Now	625 ps
Cursor 1	400 ps



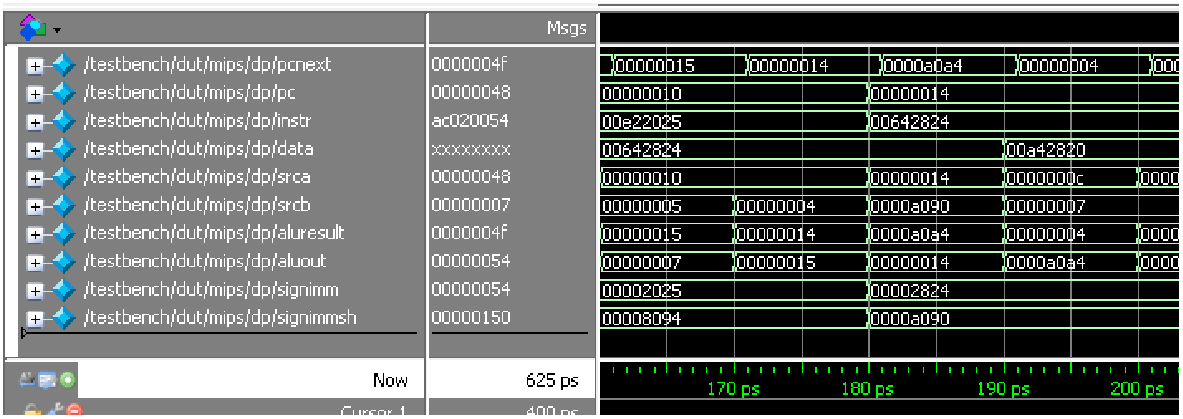
	Msgs
/testbench/dut/mips/dp/pcnext	0000004f
/testbench/dut/mips/dp/pc	00000048
/testbench/dut/mips/dp/instr	ac020054
/testbench/dut/mips/dp/data	xxxxxxxx
/testbench/dut/mips/dp/srca	00000048
/testbench/dut/mips/dp/srcb	00000007
/testbench/dut/mips/dp/aluresult	0000004f
/testbench/dut/mips/dp/aluout	00000054
/testbench/dut/mips/dp/signimm	00000054
/testbench/dut/mips/dp/signimmsh	00000150
Now	625 ps
Cursor 1	400 ps



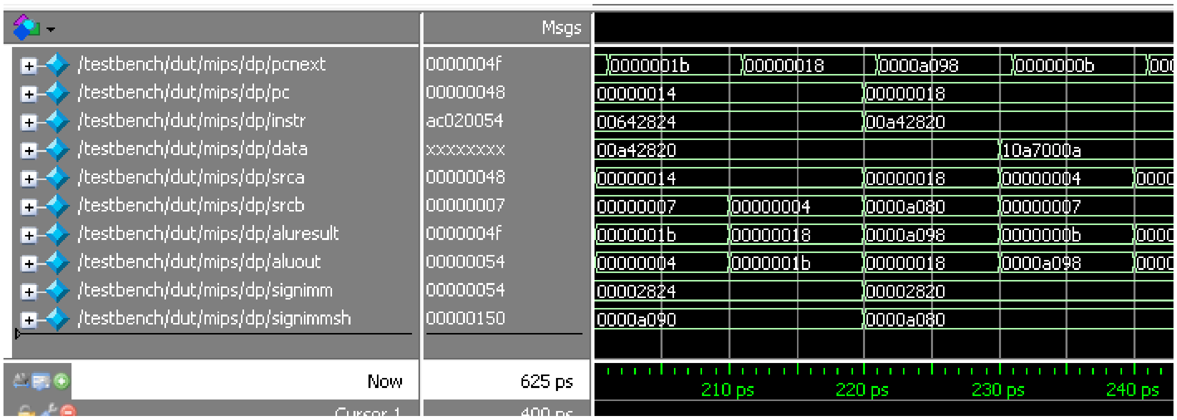
	Msgs
/testbench/dut/mips/dp/pcnext	0000004f
/testbench/dut/mips/dp/pc	00000048
/testbench/dut/mips/dp/instr	ac020054
/testbench/dut/mips/dp/data	xxxxxxxx
/testbench/dut/mips/dp/srca	00000048
/testbench/dut/mips/dp/srcb	00000007
/testbench/dut/mips/dp/aluresult	0000004f
/testbench/dut/mips/dp/aluout	00000054
/testbench/dut/mips/dp/signimm	00000054
/testbench/dut/mips/dp/signimmsh	00000150
Now	625 ps
Cursor 1	400 ps



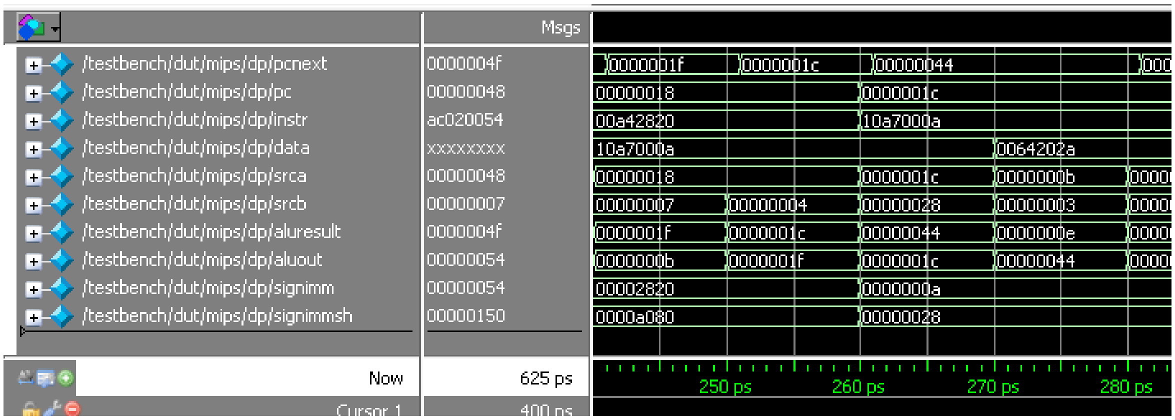
```
                                          Msgs
/testbench/dut/mips/dp/pcnext        0000004f   00000010  000080a4  00000007  000
/testbench/dut/mips/dp/pc            00000048   0000000c  00000010
/testbench/dut/mips/dp/instr         ac020054   2067fff7  00e22025
/testbench/dut/mips/dp/data          xxxxxxxx   00e22025  00642824
/testbench/dut/mips/dp/srca          00000048   0000000c  00000010  00000003  0000
/testbench/dut/mips/dp/srcb          00000007   00000004  00008094  00000005
/testbench/dut/mips/dp/aluresult     0000004f   00000010  000080a4  00000007  0000
/testbench/dut/mips/dp/aluout        00000054   00000003  00000010  000080a4  0000
/testbench/dut/mips/dp/signimm       00000054   fffffff7  00002025
/testbench/dut/mips/dp/signimmsh     00000150   ffffffdc  00008094

                                Now  625 ps     130 ps  140 ps  150 ps  160 ps
                           Cursor 1  400 ps
```



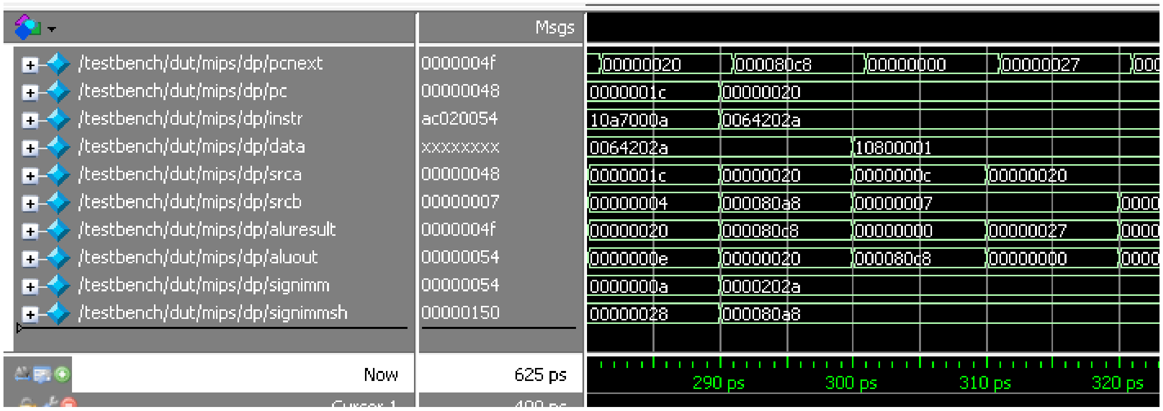
```
                                          Msgs
/testbench/dut/mips/dp/pcnext        0000004f   00000015  00000014  0000a0a4  00000004  000
/testbench/dut/mips/dp/pc            00000048   00000010  00000014
/testbench/dut/mips/dp/instr         ac020054   00e22025  00642824
/testbench/dut/mips/dp/data          xxxxxxxx   00642824  00a42820
/testbench/dut/mips/dp/srca          00000048   00000010  00000014  0000000c  0000
/testbench/dut/mips/dp/srcb          00000007   00000005  00000004  0000a090  00000007
/testbench/dut/mips/dp/aluresult     0000004f   00000015  00000014  0000a0a4  00000004  0000
/testbench/dut/mips/dp/aluout        00000054   00000007  00000015  00000014  0000a0a4  0000
/testbench/dut/mips/dp/signimm       00000054   00002025  00002824
/testbench/dut/mips/dp/signimmsh     00000150   00008094  0000a090

                                Now  625 ps     170 ps  180 ps  190 ps  200 ps
                           Cursor 1  400 ps
```



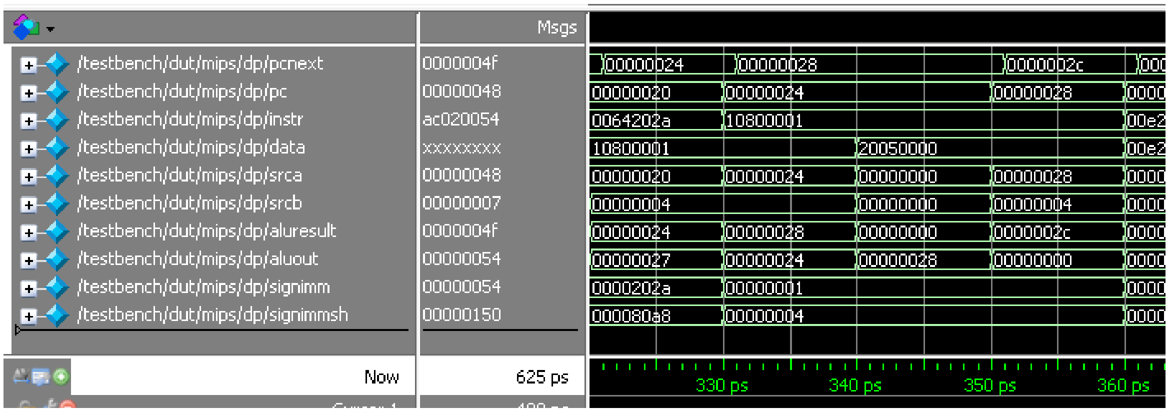
```
                                          Msgs
/testbench/dut/mips/dp/pcnext        0000004f   0000001b  00000018  0000a098  0000000b  000
/testbench/dut/mips/dp/pc            00000048   00000014  00000018
/testbench/dut/mips/dp/instr         ac020054   00642824  00a42820
/testbench/dut/mips/dp/data          xxxxxxxx   00a42820  10a7000a
/testbench/dut/mips/dp/srca          00000048   00000014  00000018  00000004  0000
/testbench/dut/mips/dp/srcb          00000007   00000007  00000004  0000a080  00000007
/testbench/dut/mips/dp/aluresult     0000004f   0000001b  00000018  0000a098  0000000b  0000
/testbench/dut/mips/dp/aluout        00000054   00000004  0000001b  00000018  0000a098  0000
/testbench/dut/mips/dp/signimm       00000054   00002824  00002820
/testbench/dut/mips/dp/signimmsh     00000150   0000a090  0000a080

                                Now  625 ps     210 ps  220 ps  230 ps  240 ps
                           Cursor 1  400 ps
```



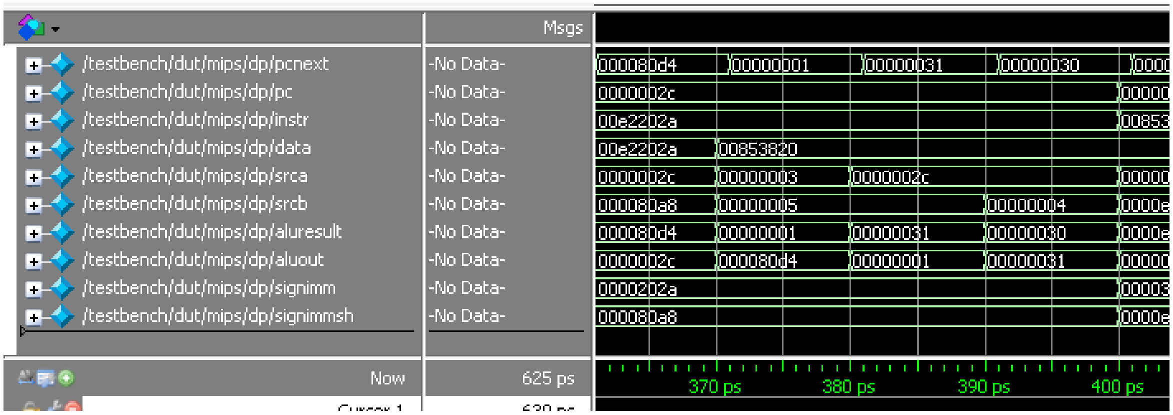
```
                                          Msgs
/testbench/dut/mips/dp/pcnext        0000004f   0000001f  0000001c  00000044  000
/testbench/dut/mips/dp/pc            00000048   00000018  0000001c
/testbench/dut/mips/dp/instr         ac020054   00a42820  10a7000a
/testbench/dut/mips/dp/data          xxxxxxxx   10a7000a  0064202a
/testbench/dut/mips/dp/srca          00000048   00000018  0000001c  0000000b  0000
/testbench/dut/mips/dp/srcb          00000007   00000007  00000004  00000028  00000003  0000
/testbench/dut/mips/dp/aluresult     0000004f   0000001f  0000001c  00000044  0000000e  0000
/testbench/dut/mips/dp/aluout        00000054   0000000b  0000001f  0000001c  00000044  0000
/testbench/dut/mips/dp/signimm       00000054   00002820  0000000a
/testbench/dut/mips/dp/signimmsh     00000150   0000a080  00000028

                                Now  625 ps     250 ps  260 ps  270 ps  280 ps
                           Cursor 1  400 ps
```



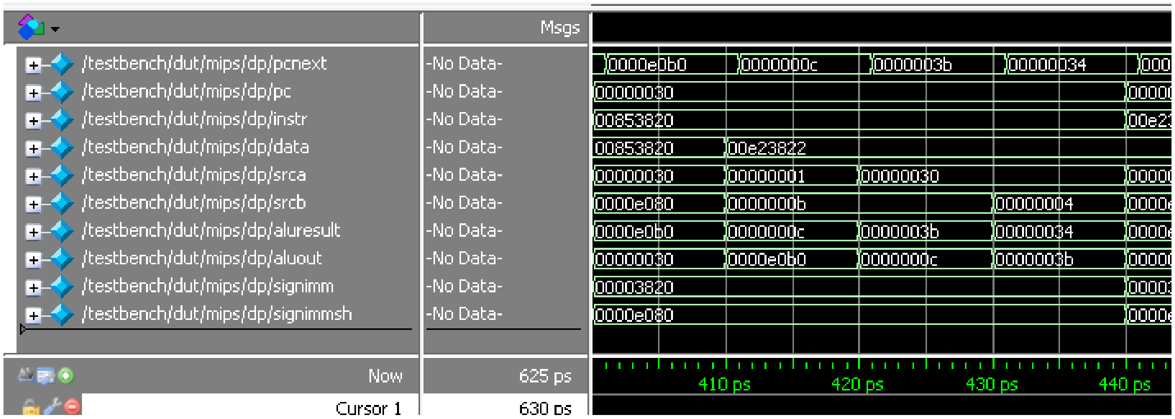
```
                                      Msgs
/testbench/dut/mips/dp/pcnext         0000004f   00000020   000080c8   00000000   00000027   000
/testbench/dut/mips/dp/pc             00000048   0000001c   00000020
/testbench/dut/mips/dp/instr          ac020054   10a7000a   0064202a
/testbench/dut/mips/dp/data           xxxxxxxx   0064202a              10800001
/testbench/dut/mips/dp/srca           00000048   0000001c   00000020   0000000c   00000020
/testbench/dut/mips/dp/srcb           00000007   00000004   000080a8   00000007              0000
/testbench/dut/mips/dp/aluresult      0000004f   00000020   000080c8   00000000   00000027   0000
/testbench/dut/mips/dp/aluout         00000054   0000000e   00000020   000080c8   00000000   0000
/testbench/dut/mips/dp/signimm        00000054   0000000a   0000202a
/testbench/dut/mips/dp/signimmsh      00000150   00000028   000080a8
Now                                   625 ps          290 ps     300 ps     310 ps     320 ps
Cursor 1                              400 ps
```



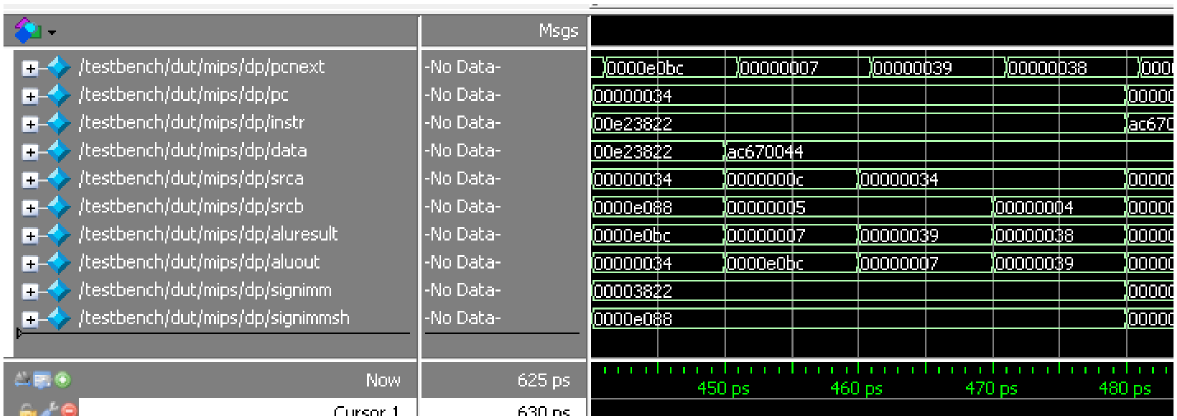
```
                                      Msgs
/testbench/dut/mips/dp/pcnext         0000004f   00000024   00000028              0000002c   000
/testbench/dut/mips/dp/pc             00000048   00000020   00000024              00000028   0000
/testbench/dut/mips/dp/instr          ac020054   0064202a   10800001                         00e2
/testbench/dut/mips/dp/data           xxxxxxxx   10800001              20050000              00e2
/testbench/dut/mips/dp/srca           00000048   00000020   00000024   00000000   00000028   0000
/testbench/dut/mips/dp/srcb           00000007   00000004              00000000   00000004   0000
/testbench/dut/mips/dp/aluresult      0000004f   00000024   00000028   00000000   0000002c   0000
/testbench/dut/mips/dp/aluout         00000054   00000027   00000024   00000028   00000000   0000
/testbench/dut/mips/dp/signimm        00000054   0000202a   00000001                         0000
/testbench/dut/mips/dp/signimmsh      00000150   000080a8   00000004                         0000
Now                                   625 ps          330 ps     340 ps     350 ps     360 ps
Cursor 1                              400 ps
```



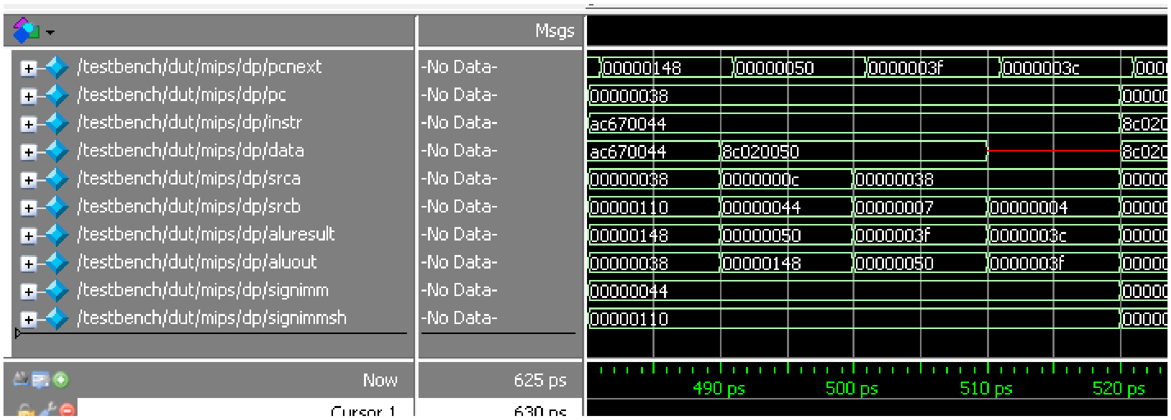
```
                                      Msgs
/testbench/dut/mips/dp/pcnext         -No Data-  000080d4   00000001   00000031   00000030   0000
/testbench/dut/mips/dp/pc             -No Data-  0000002c                                    00000
/testbench/dut/mips/dp/instr          -No Data-  00e2202a                                    00853
/testbench/dut/mips/dp/data           -No Data-  00e2202a   00853820
/testbench/dut/mips/dp/srca           -No Data-  0000002c   00000003   0000002c              00000
/testbench/dut/mips/dp/srcb           -No Data-  000080a8   00000005              00000004   0000e
/testbench/dut/mips/dp/aluresult      -No Data-  000080d4   00000001   00000031   00000030   0000e
/testbench/dut/mips/dp/aluout         -No Data-  0000002c   000080d4   00000001   00000031   00000
/testbench/dut/mips/dp/signimm        -No Data-  0000202a                                    00003
/testbench/dut/mips/dp/signimmsh      -No Data-  000080a8                                    0000e
Now                                   625 ps          370 ps     380 ps     390 ps     400 ps
Cursor 1                              630 ps
```



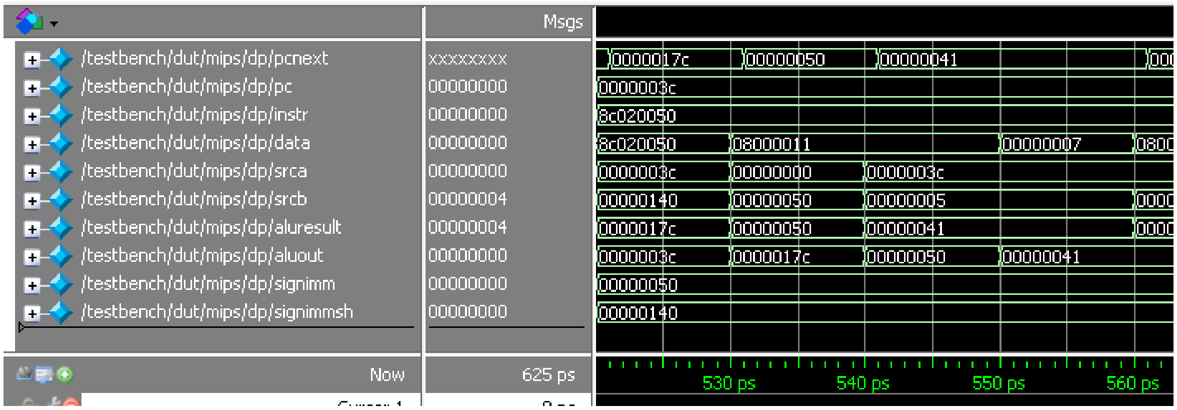
```
                                      Msgs
/testbench/dut/mips/dp/pcnext         -No Data-  0000e0b0   0000000c   0000003b   00000034   000
/testbench/dut/mips/dp/pc             -No Data-  00000030                                    00000
/testbench/dut/mips/dp/instr          -No Data-  00853820                                    00e2
/testbench/dut/mips/dp/data           -No Data-  00853820   00e23822
/testbench/dut/mips/dp/srca           -No Data-  00000030   00000001   00000030              00000
/testbench/dut/mips/dp/srcb           -No Data-  0000e080   0000000b              00000004   0000e
/testbench/dut/mips/dp/aluresult      -No Data-  0000e0b0   0000000c   0000003b   00000034   0000e
/testbench/dut/mips/dp/aluout         -No Data-  00000030   0000e0b0   0000000c   0000003b   00000
/testbench/dut/mips/dp/signimm        -No Data-  00003820                                    00003
/testbench/dut/mips/dp/signimmsh      -No Data-  0000e080                                    0000e
Now                                   625 ps          410 ps     420 ps     430 ps     440 ps
Cursor 1                              630 ps
```



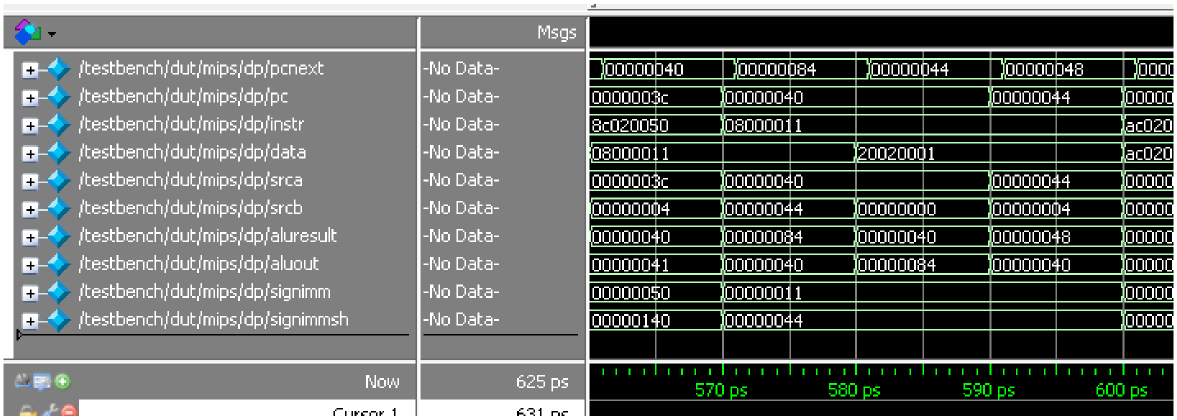
```
                                   Msgs
/testbench/dut/mips/dp/pcnext      -No Data-   0000e0bc  00000007  00000039  00000038  000
/testbench/dut/mips/dp/pc          -No Data-   00000034                                00000
/testbench/dut/mips/dp/instr       -No Data-   00e23822                                ac670
/testbench/dut/mips/dp/data        -No Data-   00e23822  ac670044
/testbench/dut/mips/dp/srca        -No Data-   00000034  0000000c  00000034            00000
/testbench/dut/mips/dp/srcb        -No Data-   0000e088  00000005            00000004  00000
/testbench/dut/mips/dp/aluresult   -No Data-   0000e0bc  00000007  00000039  00000038  00000
/testbench/dut/mips/dp/aluout      -No Data-   00000034  0000e0bc  00000007  00000039  00000
/testbench/dut/mips/dp/signimm     -No Data-   00003822                                00000
/testbench/dut/mips/dp/signimmsh   -No Data-   0000e088                                00000
Now                                625 ps         450 ps    460 ps    470 ps    480 ps
Cursor 1                           630 ps
```



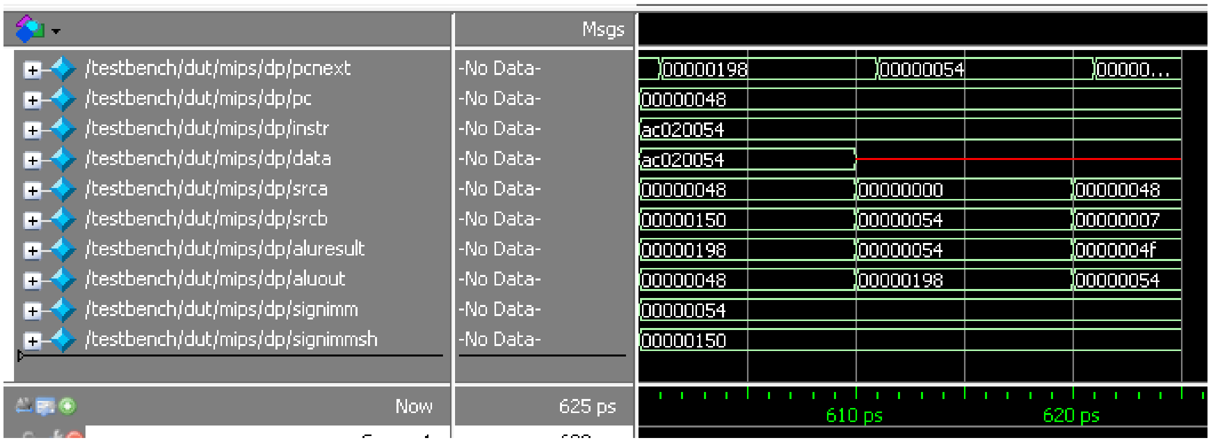
```
                                   Msgs
/testbench/dut/mips/dp/pcnext      -No Data-   00000148  00000050  0000003f  0000003c  000
/testbench/dut/mips/dp/pc          -No Data-   00000038                                00000
/testbench/dut/mips/dp/instr       -No Data-   ac670044                                8c020
/testbench/dut/mips/dp/data        -No Data-   ac670044  8c020050                      8c020
/testbench/dut/mips/dp/srca        -No Data-   00000038  0000000c  00000038            00000
/testbench/dut/mips/dp/srcb        -No Data-   00000110  00000044  00000007  00000004  00000
/testbench/dut/mips/dp/aluresult   -No Data-   00000148  00000050  0000003f  0000003c  00000
/testbench/dut/mips/dp/aluout      -No Data-   00000038  00000148  00000050  0000003f  00000
/testbench/dut/mips/dp/signimm     -No Data-   00000044                                00000
/testbench/dut/mips/dp/signimmsh   -No Data-   00000110                                00000
Now                                625 ps         490 ps    500 ps    510 ps    520 ps
Cursor 1                           630 ps
```



```
                                   Msgs
/testbench/dut/mips/dp/pcnext      xxxxxxxx    0000017c  00000050  00000041            000
/testbench/dut/mips/dp/pc          00000000    0000003c
/testbench/dut/mips/dp/instr       00000000    8c020050
/testbench/dut/mips/dp/data        00000000    8c020050  08000011            00000007  0800
/testbench/dut/mips/dp/srca        00000000    0000003c  00000000  0000003c
/testbench/dut/mips/dp/srcb        00000004    00000140  00000050  00000005            0000
/testbench/dut/mips/dp/aluresult   00000004    0000017c  00000050  00000041            0000
/testbench/dut/mips/dp/aluout      00000000    0000003c  0000017c  00000050  00000041
/testbench/dut/mips/dp/signimm     00000000    00000050
/testbench/dut/mips/dp/signimmsh   00000000    00000140
Now                                625 ps         530 ps    540 ps    550 ps    560 ps
Cursor 1                           0 ps
```



```
                                   Msgs
/testbench/dut/mips/dp/pcnext      -No Data-   00000040  00000084  00000044  00000048  000
/testbench/dut/mips/dp/pc          -No Data-   0000003c  00000040            00000044  00000
/testbench/dut/mips/dp/instr       -No Data-   8c020050  08000011                      ac020
/testbench/dut/mips/dp/data        -No Data-   08000011            20020001            ac020
/testbench/dut/mips/dp/srca        -No Data-   0000003c  00000040            00000044  00000
/testbench/dut/mips/dp/srcb        -No Data-   00000004  00000044  00000000  00000004  00000
/testbench/dut/mips/dp/aluresult   -No Data-   00000040  00000084  00000040  00000048  00000
/testbench/dut/mips/dp/aluout      -No Data-   00000041  00000040  00000084  00000040  00000
/testbench/dut/mips/dp/signimm     -No Data-   00000050  00000011                      00000
/testbench/dut/mips/dp/signimmsh   -No Data-   00000140  00000044                      00000
Now                                625 ps         570 ps    580 ps    590 ps    600 ps
Cursor 1                           631 ps
```


Msgs
/testbench/dut/mips/dp/pcnext -No Data- 00000198 00000054 00000...
/testbench/dut/mips/dp/pc -No Data- 00000048
/testbench/dut/mips/dp/instr -No Data- ac020054
/testbench/dut/mips/dp/data -No Data- ac020054
/testbench/dut/mips/dp/srca -No Data- 00000048 00000000 00000048
/testbench/dut/mips/dp/srcb -No Data- 00000150 00000054 00000007
/testbench/dut/mips/dp/aluresult -No Data- 00000198 00000054 0000004f
/testbench/dut/mips/dp/aluout -No Data- 00000048 00000198 00000054
/testbench/dut/mips/dp/signimm -No Data- 00000054
/testbench/dut/mips/dp/signimmsh -No Data- 00000150
Now 625 ps
610 ps
620 ps

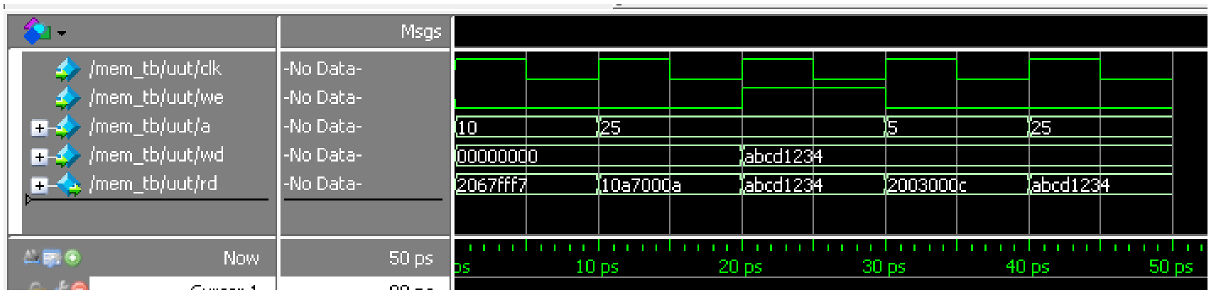
At this point, the data input is red indicating that the next memory address contains no instruction or data which is the end of the program.

Memory

The testbench code for the memory is given below:

```
module mem_tb;
        reg clk, we;
        reg [31:0] a, wd;
        wire [31:0] rd;
        mem uut(clk, we, a, wd, rd);
        always begin clk <= 1; #5; clk <= 0; #5; end

        initial begin
                we <= 0;
                wd <= 0;
        end
        initial begin
                a <= 10; //read memory from address 10
                #10 a <= 25; //read memory from address 25
                #10 a <= 25;
                we <= 1;
                wd <= 32'habcd1234; //write abcd1234 to address 25
                #10 we <= 0;
                a <= 5; //read memory from address 5
                #10 a <= 25; //read memory from address 25
                #10 $stop;
        end
endmodule
```

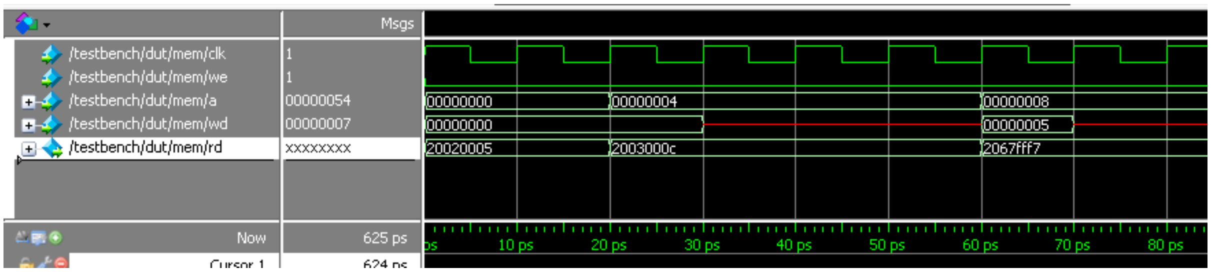
Output Waveform:

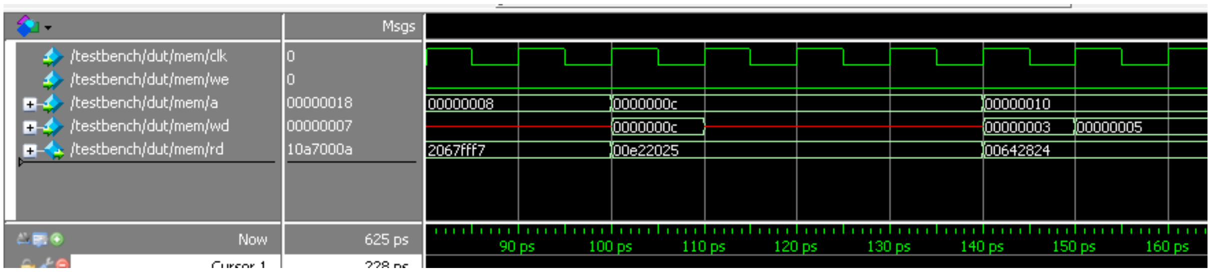


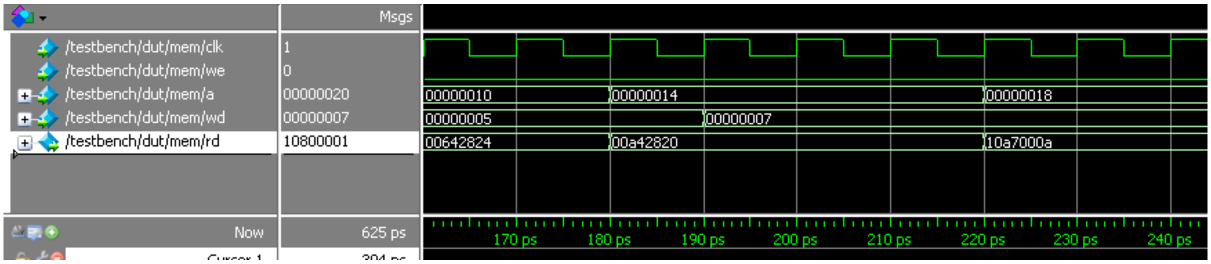
The data shows that rd outputs the data stored at given address and at address 25 the data is overwritten and read again at the last part.

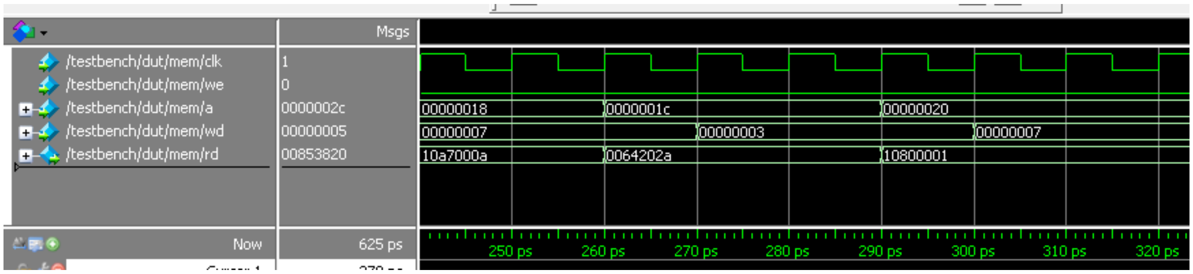
Actual Waveform of Memory using MIPS Assembly Code:

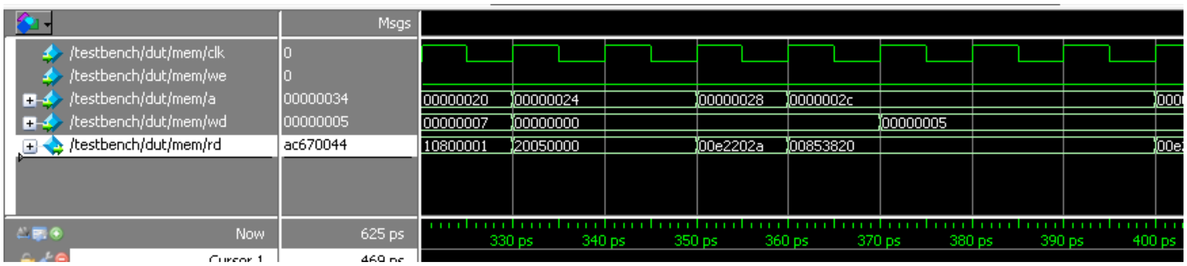
This part is just reading data from the memory at the given address until at 480ps where I-type instruction (sw, lw) is used.

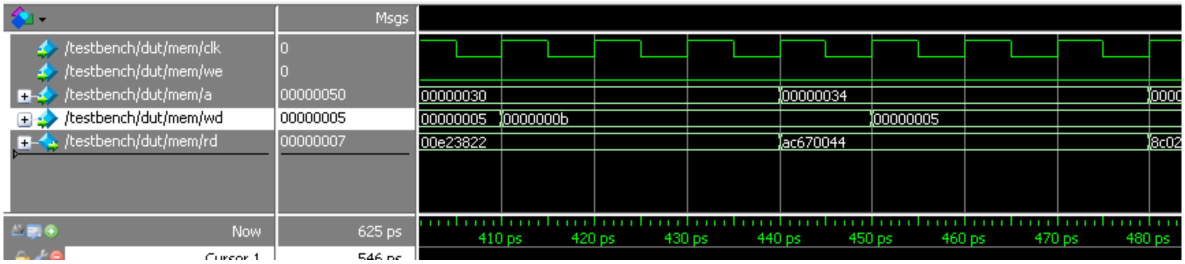


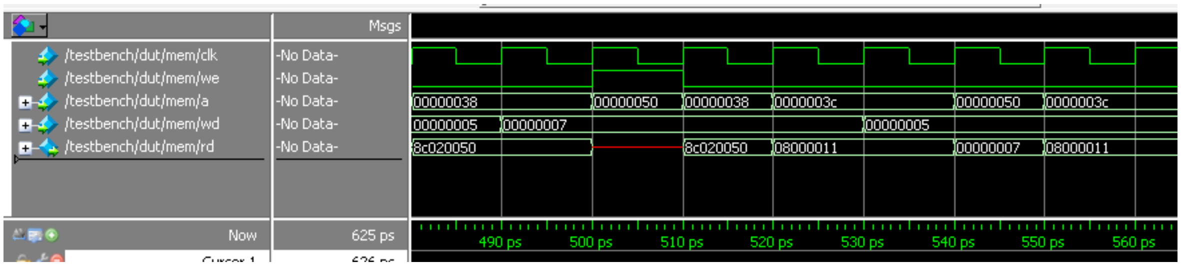




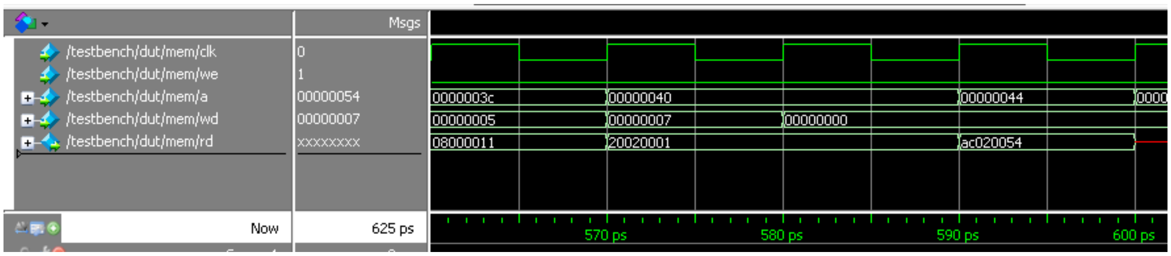

Msgs
/testbench/dut/mem/clk 1
/testbench/dut/mem/we 0
/testbench/dut/mem/a 0000002c
/testbench/dut/mem/wd 00000005
/testbench/dut/mem/rd 00853820
00000018 0000001c 00000020
00000007 00000003 00000007
10a7000a 0064202a 10800001
Now 625 ps
250 ps 260 ps 270 ps 280 ps 290 ps 300 ps 310 ps 320 ps

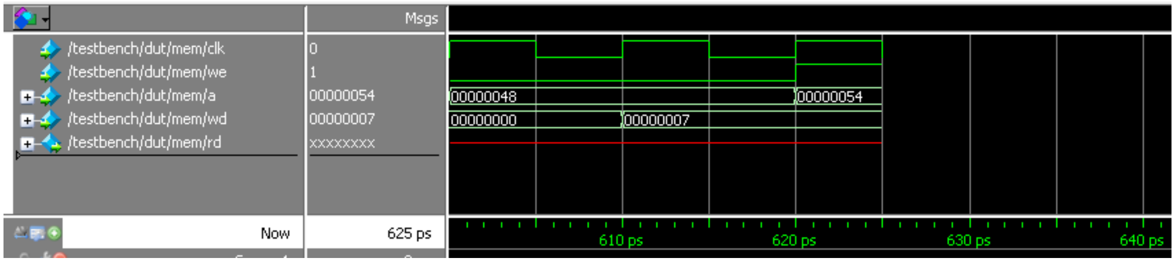

Msgs
/testbench/dut/mem/clk 0
/testbench/dut/mem/we 0
/testbench/dut/mem/a 00000034
/testbench/dut/mem/wd 00000005
/testbench/dut/mem/rd ac670044
00000020 00000024 00000028 0000002c 000
00000007 00000000 00000005
10800001 20050000 00e2202a 00853820 00e
Now 625 ps
Cursor 1 469 ps
330 ps 340 ps 350 ps 360 ps 370 ps 380 ps 390 ps 400 ps


Msgs
/testbench/dut/mem/clk 0
/testbench/dut/mem/we 0
/testbench/dut/mem/a 00000050
/testbench/dut/mem/wd 00000005
/testbench/dut/mem/rd 00000007
00000030 00000034 0000
00000005 0000000b 00000005
00e23822 ac670044 8c02
Now 625 ps
Cursor 1 546 ps
410 ps 420 ps 430 ps 440 ps 450 ps 460 ps 470 ps 480 ps


Msgs
/testbench/dut/mem/clk -No Data-
/testbench/dut/mem/we -No Data-
/testbench/dut/mem/a -No Data-
/testbench/dut/mem/wd -No Data-
/testbench/dut/mem/rd -No Data-
00000038 00000050 00000038 0000003c 00000050 0000003c
00000005 00000007 00000005
8c020050 8c020050 08000011 00000007 08000011
Now 625 ps
Cursor 1 626 ps
490 ps 500 ps 510 ps 520 ps 530 ps 540 ps 550 ps 560 ps

At 500ps the actual writing of data happens at MEMWRITE state where 7 is stored at address [50h] or [80d] at 540ps the data at [80d] is read and the 7 value is returned.


Msgs
/testbench/dut/mem/clk 0
/testbench/dut/mem/we 1
/testbench/dut/mem/a 00000054
/testbench/dut/mem/wd 00000007
/testbench/dut/mem/rd xxxxxxxx
0000003c 00000040 00000044 0000
00000005 00000007 00000000
08000011 20020001 ac020054
Now 625 ps
570 ps 580 ps 590 ps 600 ps



```
/testbench/dut/mem/clk   0
/testbench/dut/mem/we    1
/testbench/dut/mem/a     00000054
/testbench/dut/mem/wd    00000007
/testbench/dut/mem/rd    xxxxxxxx
Now                      625 ps
```

At last, the value 7 is stored at address [54h] or [84d] at 620ps.

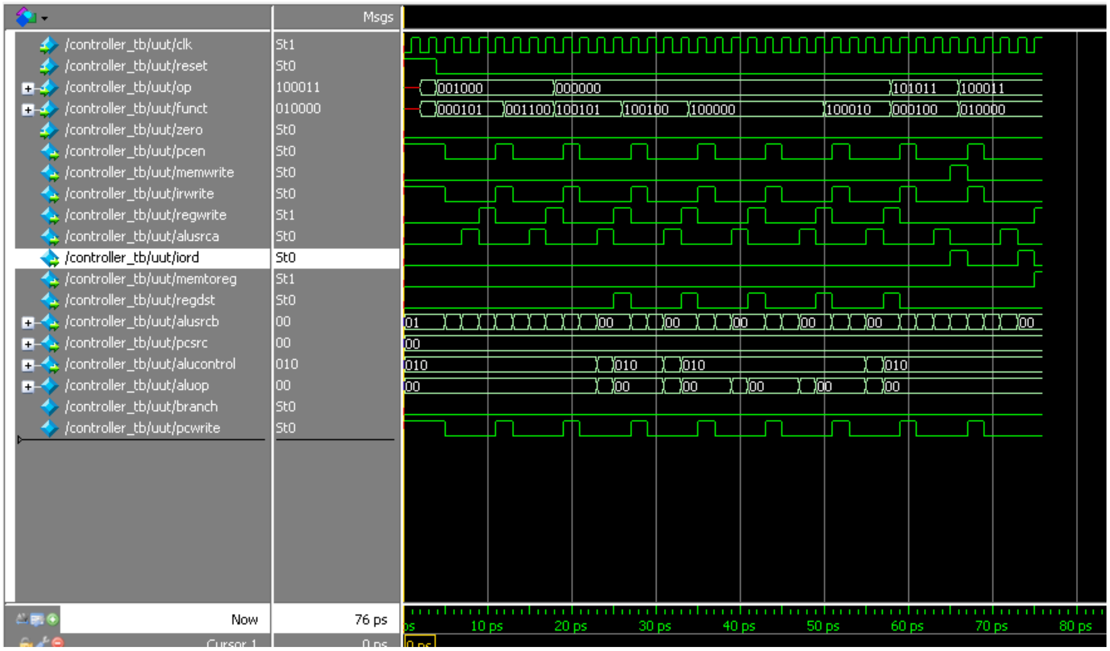
Control Unit

The code for the testbench is given below:

```
module controller_tb;
reg clk, reset, zero;
reg [5:0] op, funct;

wire     iord, memwrite, irwrite, regdst, memtoreg, regwrite, alusrca, pcen;
wire [1:0] alusrcb, pcsrc;
wire [2:0] alucontrol;

reg [31:0] instr;

always @ ( * )
begin
op <= instr [31:26];
funct <= instr [5:0];
end

controller uut(clk, reset, op, funct, zero, pcen, memwrite, irwrite, regwrite, alusrca, iord,
                    memtoreg, regdst, alusrcb, pcsrc, alucontrol);

always #1 clk = ~clk;

initial begin
        clk = 0;
        reset = 1;
        zero = 0;

        #2  reset = 1; instr = 0;
        #2  reset = 0; instr = 32'h20020005; //addi
        #2  reset = 0; instr = 32'h20020005; //addi
```

```
	#2  reset = 0; instr = 32'h20020005; //addi
	#2 reset = 0; instr = 32'h20020005; //addi
	#2 reset = 0; instr = 32'h2003000c; //addi
	#2 reset = 0; instr = 32'h2003000c; //addi
	#2 reset = 0; instr = 32'h2003000c; //addi
	#2 reset = 0; instr = 32'h00e22025; //or
	#2 reset = 0; instr = 32'h00e22025; //or
	#2 reset = 0; instr = 32'h00e22025; //or
	#2 reset = 0; instr = 32'h00e22025; //or
	#2 reset = 0; instr = 32'h00642824; //and
	#2 reset = 0; instr = 32'h00642824; //and
	#2 reset = 0; instr = 32'h00642824; //and
	#2 reset = 0; instr = 32'h00642824; //and
	#2 reset = 0; instr = 32'h00a42820; //add
	#2 reset = 0; instr = 32'h00a42820; //add
	#2 reset = 0; instr = 32'h00a42820; //add
	#2 reset = 0; instr = 32'h00a42820; //add
	#2 reset = 0; instr = 32'h00853820; //add
	#2 reset = 0; instr = 32'h00853820; //add
	#2 reset = 0; instr = 32'h00853820; //add
	#2 reset = 0; instr = 32'h00853820; //add
	#2 reset = 0; instr = 32'h00e23822; //sub
	#2 reset = 0; instr = 32'h00e23822; //sub
	#2 reset = 0; instr = 32'h00e23822; //sub
	#2 reset = 0; instr = 32'h00e23822; //sub
	#2 reset = 0; instr = 32'hac670044; //sw
	#2 reset = 0; instr = 32'hac670044; //sw
	#2 reset = 0; instr = 32'hac670044; //sw
	#2 reset = 0; instr = 32'hac670044; //sw
	#2 reset = 0; instr = 32'h8c020050; //lw
	#2 reset = 0; instr = 32'h8c020050; //lw
	#2 reset = 0; instr = 32'h8c020050; //lw
	#2 reset = 0; instr = 32'h8c020050; //lw
	#2 reset = 0; instr = 32'h8c020050; //lw
	#2 $stop;

end
endmodule
```

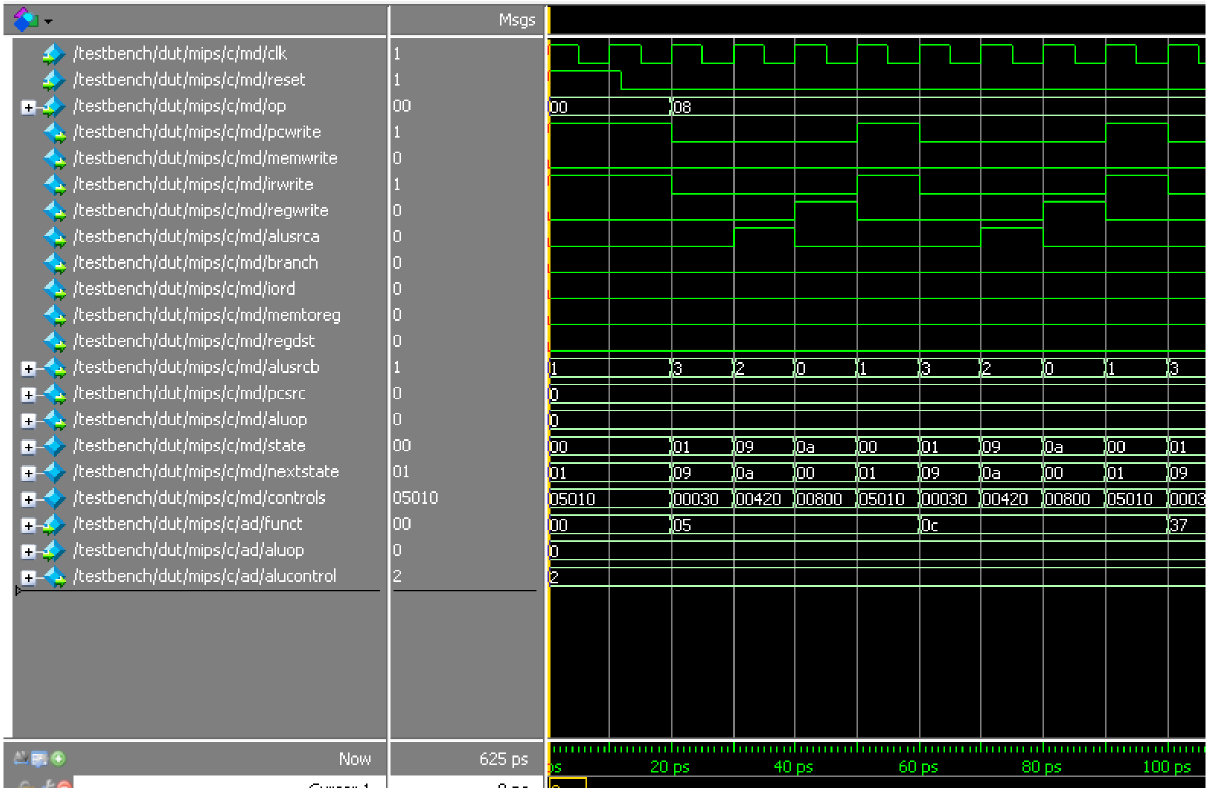
Output waveform:



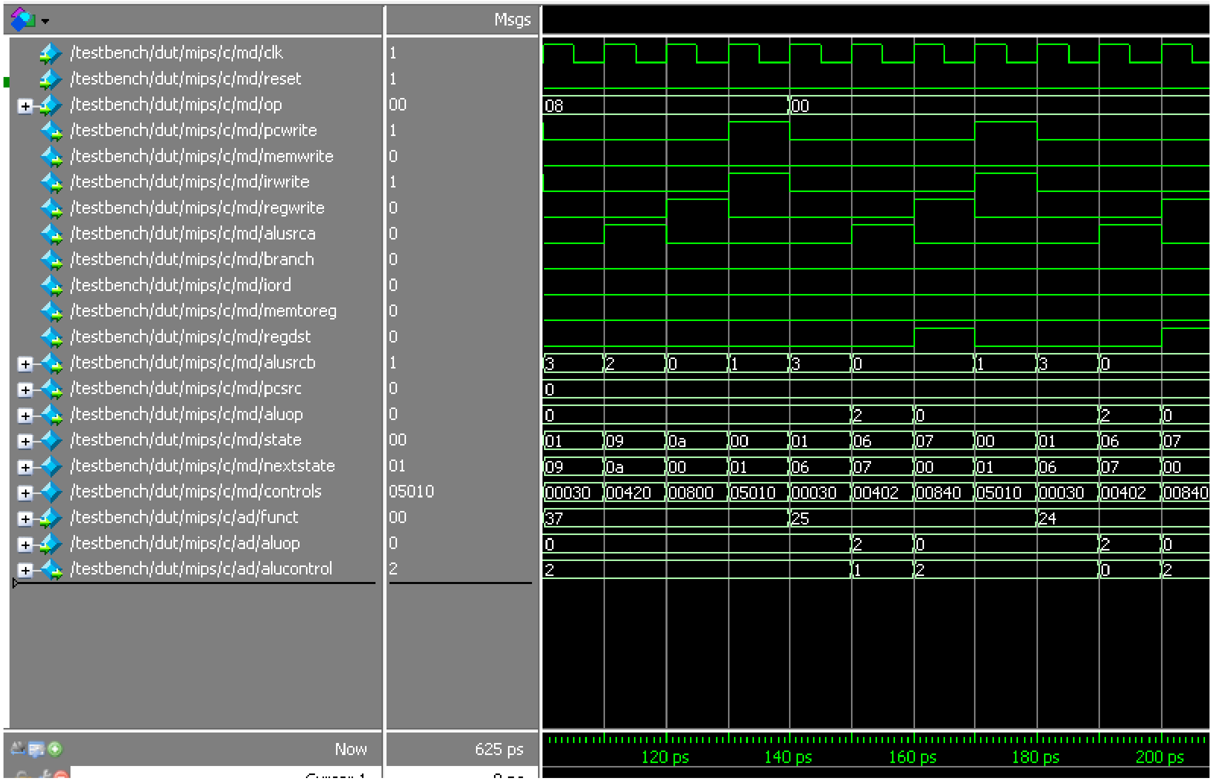
	Msgs
/controller_tb/uut/clk	St1
/controller_tb/uut/reset	St0
/controller_tb/uut/op	100011
/controller_tb/uut/funct	010000
/controller_tb/uut/zero	St0
/controller_tb/uut/pcen	St0
/controller_tb/uut/memwrite	St0
/controller_tb/uut/irwrite	St0
/controller_tb/uut/regwrite	St1
/controller_tb/uut/alusrca	St0
/controller_tb/uut/iord	St0
/controller_tb/uut/memtoreg	St1
/controller_tb/uut/regdst	St0
/controller_tb/uut/alusrcb	00
/controller_tb/uut/pcsrc	00
/controller_tb/uut/alucontrol	010
/controller_tb/uut/aluop	00
/controller_tb/uut/branch	St0
/controller_tb/uut/pcwrite	St0
Now	76 ps
Cursor 1	0 ps

Instructions are given as a 32-bit value with the bits [31:26] as the opcode and bits [5:0] as the functions. The outputs combined (concatenated altogether) matches the state table.

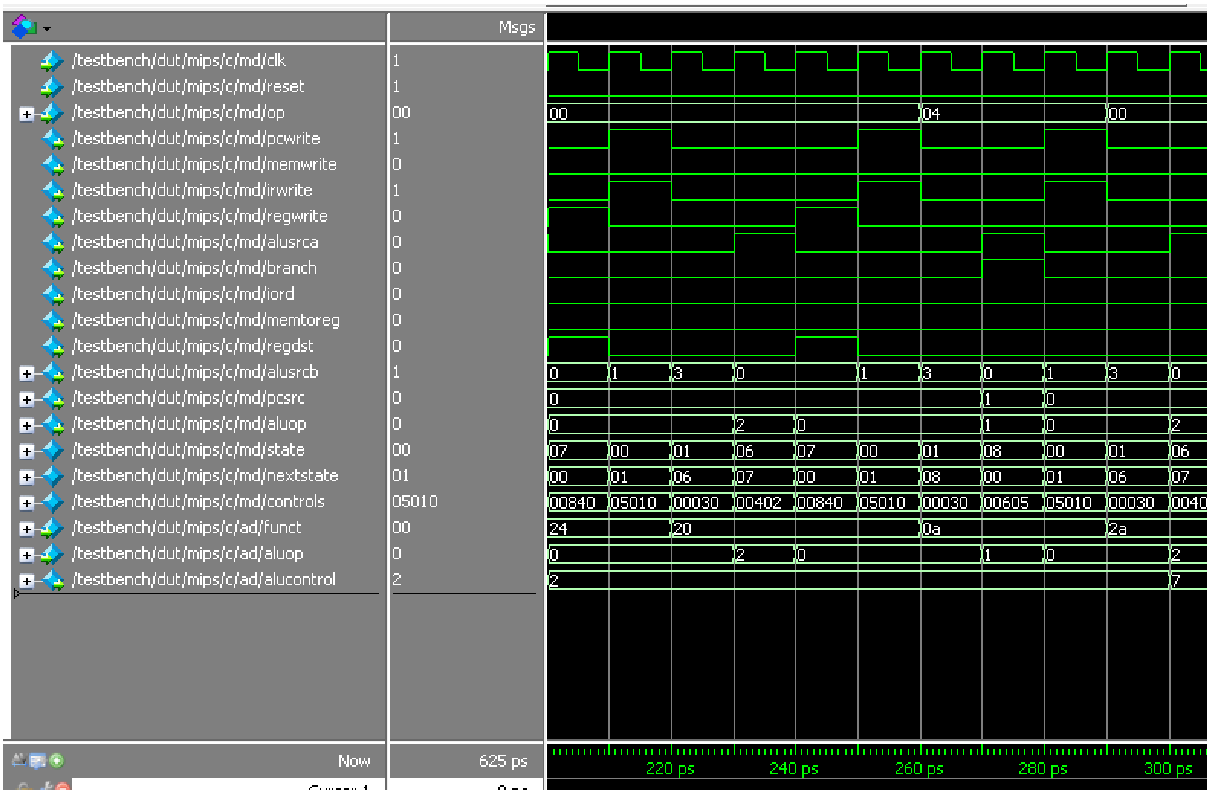
Actual Waveform of Control Unit using MIPS Assembly Code:



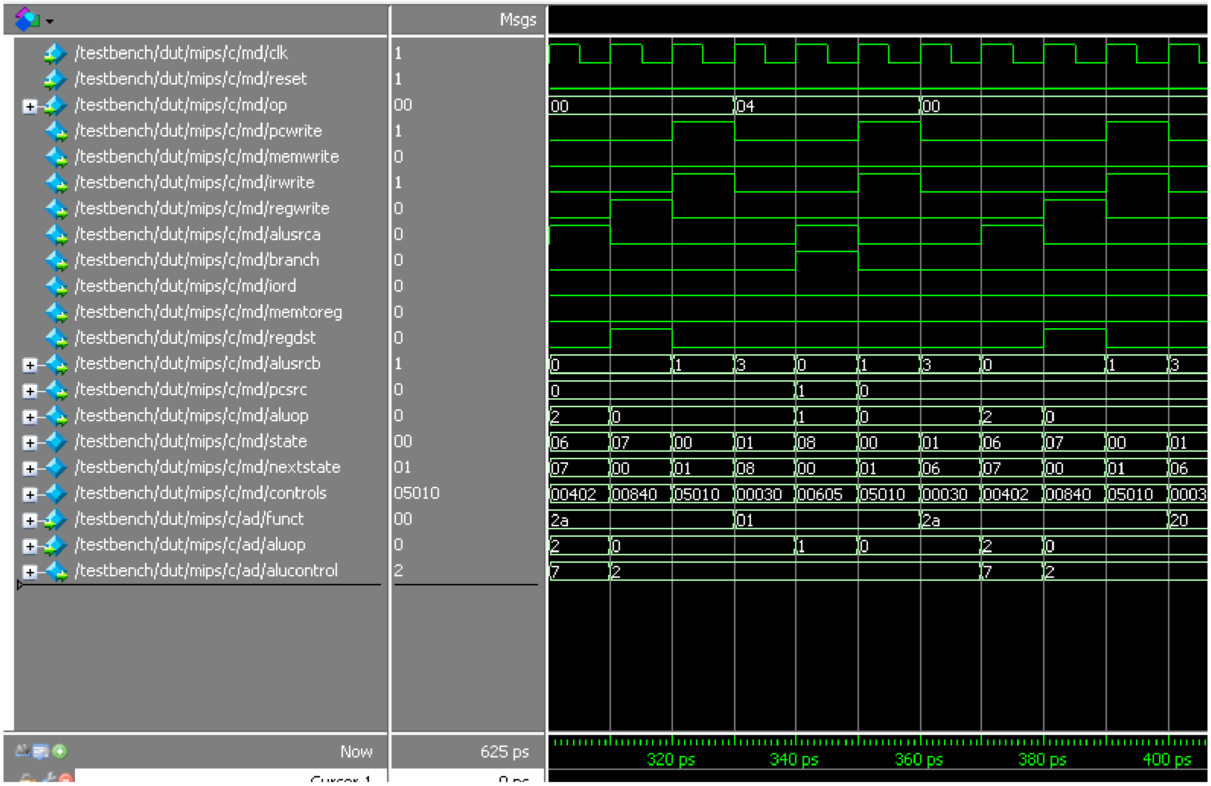
```
Msgs
/testbench/dut/mips/c/md/clk          1
/testbench/dut/mips/c/md/reset        1
/testbench/dut/mips/c/md/op           00
/testbench/dut/mips/c/md/pcwrite      1
/testbench/dut/mips/c/md/memwrite     0
/testbench/dut/mips/c/md/irwrite      1
/testbench/dut/mips/c/md/regwrite     0
/testbench/dut/mips/c/md/alusrca      0
/testbench/dut/mips/c/md/branch       0
/testbench/dut/mips/c/md/iord         0
/testbench/dut/mips/c/md/memtoreg     0
/testbench/dut/mips/c/md/regdst       0
/testbench/dut/mips/c/md/alusrcb      1
/testbench/dut/mips/c/md/pcsrc        0
/testbench/dut/mips/c/md/aluop        0
/testbench/dut/mips/c/md/state        00
/testbench/dut/mips/c/md/nextstate    01
/testbench/dut/mips/c/md/controls     05010
/testbench/dut/mips/c/ad/funct        00
/testbench/dut/mips/c/ad/aluop        0
/testbench/dut/mips/c/ad/alucontrol   2
Now                                   625 ps
```



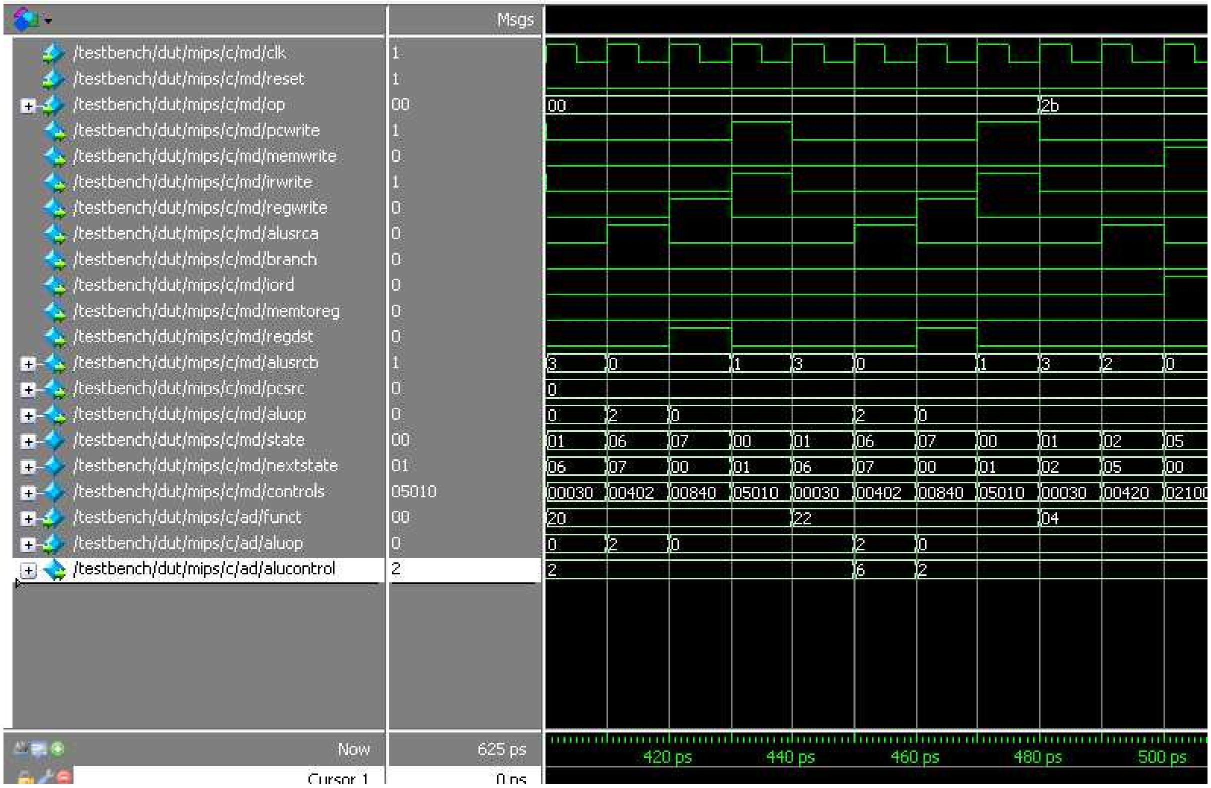
```
Msgs
/testbench/dut/mips/c/md/clk          1
/testbench/dut/mips/c/md/reset        1
/testbench/dut/mips/c/md/op           00
/testbench/dut/mips/c/md/pcwrite      1
/testbench/dut/mips/c/md/memwrite     0
/testbench/dut/mips/c/md/irwrite      1
/testbench/dut/mips/c/md/regwrite     0
/testbench/dut/mips/c/md/alusrca      0
/testbench/dut/mips/c/md/branch       0
/testbench/dut/mips/c/md/iord         0
/testbench/dut/mips/c/md/memtoreg     0
/testbench/dut/mips/c/md/regdst       0
/testbench/dut/mips/c/md/alusrcb      1
/testbench/dut/mips/c/md/pcsrc        0
/testbench/dut/mips/c/md/aluop        0
/testbench/dut/mips/c/md/state        00
/testbench/dut/mips/c/md/nextstate    01
/testbench/dut/mips/c/md/controls     05010
/testbench/dut/mips/c/ad/funct        00
/testbench/dut/mips/c/ad/aluop        0
/testbench/dut/mips/c/ad/alucontrol   2
Now                                   625 ps
```



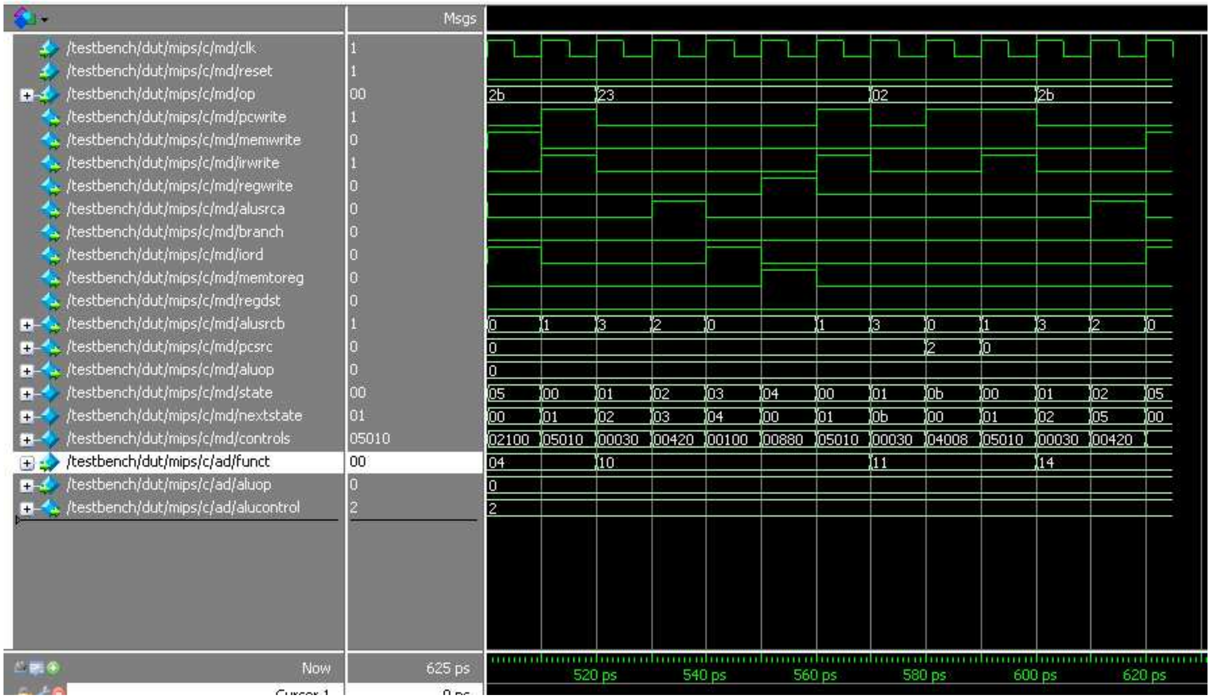
```
/testbench/dut/mips/c/md/clk          1
/testbench/dut/mips/c/md/reset        1
/testbench/dut/mips/c/md/op           00
/testbench/dut/mips/c/md/pcwrite      1
/testbench/dut/mips/c/md/memwrite     0
/testbench/dut/mips/c/md/irwrite      1
/testbench/dut/mips/c/md/regwrite     0
/testbench/dut/mips/c/md/alusrca      0
/testbench/dut/mips/c/md/branch       0
/testbench/dut/mips/c/md/iord         0
/testbench/dut/mips/c/md/memtoreg     0
/testbench/dut/mips/c/md/regdst       0
/testbench/dut/mips/c/md/alusrcb      1
/testbench/dut/mips/c/md/pcsrc        0
/testbench/dut/mips/c/md/aluop        0
/testbench/dut/mips/c/md/state        00
/testbench/dut/mips/c/md/nextstate    01
/testbench/dut/mips/c/md/controls     05010
/testbench/dut/mips/c/ad/funct        00
/testbench/dut/mips/c/ad/aluop        0
/testbench/dut/mips/c/ad/alucontrol   2
Now                                   625 ps
```



```
/testbench/dut/mips/c/md/clk          1
/testbench/dut/mips/c/md/reset        1
/testbench/dut/mips/c/md/op           00
/testbench/dut/mips/c/md/pcwrite      1
/testbench/dut/mips/c/md/memwrite     0
/testbench/dut/mips/c/md/irwrite      1
/testbench/dut/mips/c/md/regwrite     0
/testbench/dut/mips/c/md/alusrca      0
/testbench/dut/mips/c/md/branch       0
/testbench/dut/mips/c/md/iord         0
/testbench/dut/mips/c/md/memtoreg     0
/testbench/dut/mips/c/md/regdst       0
/testbench/dut/mips/c/md/alusrcb      1
/testbench/dut/mips/c/md/pcsrc        0
/testbench/dut/mips/c/md/aluop        0
/testbench/dut/mips/c/md/state        00
/testbench/dut/mips/c/md/nextstate    01
/testbench/dut/mips/c/md/controls     05010
/testbench/dut/mips/c/ad/funct        00
/testbench/dut/mips/c/ad/aluop        0
/testbench/dut/mips/c/ad/alucontrol   2
Now                                   625 ps
```



	Msgs
/testbench/dut/mips/c/md/clk	1
/testbench/dut/mips/c/md/reset	1
/testbench/dut/mips/c/md/op	00
/testbench/dut/mips/c/md/pcwrite	1
/testbench/dut/mips/c/md/memwrite	0
/testbench/dut/mips/c/md/irwrite	1
/testbench/dut/mips/c/md/regwrite	0
/testbench/dut/mips/c/md/alusrca	0
/testbench/dut/mips/c/md/branch	0
/testbench/dut/mips/c/md/iord	0
/testbench/dut/mips/c/md/memtoreg	0
/testbench/dut/mips/c/md/regdst	0
/testbench/dut/mips/c/md/alusrcb	1
/testbench/dut/mips/c/md/pcsrc	0
/testbench/dut/mips/c/md/aluop	0
/testbench/dut/mips/c/md/state	00
/testbench/dut/mips/c/md/nextstate	01
/testbench/dut/mips/c/md/controls	05010
/testbench/dut/mips/c/ad/funct	00
/testbench/dut/mips/c/ad/aluop	0
/testbench/dut/mips/c/ad/alucontrol	2
Now	625 ps
Cursor 1	0 ps



	Msgs
/testbench/dut/mips/c/md/clk	1
/testbench/dut/mips/c/md/reset	1
/testbench/dut/mips/c/md/op	00
/testbench/dut/mips/c/md/pcwrite	1
/testbench/dut/mips/c/md/memwrite	0
/testbench/dut/mips/c/md/irwrite	1
/testbench/dut/mips/c/md/regwrite	0
/testbench/dut/mips/c/md/alusrca	0
/testbench/dut/mips/c/md/branch	0
/testbench/dut/mips/c/md/iord	0
/testbench/dut/mips/c/md/memtoreg	0
/testbench/dut/mips/c/md/regdst	0
/testbench/dut/mips/c/md/alusrcb	1
/testbench/dut/mips/c/md/pcsrc	0
/testbench/dut/mips/c/md/aluop	0
/testbench/dut/mips/c/md/state	00
/testbench/dut/mips/c/md/nextstate	01
/testbench/dut/mips/c/md/controls	05010
/testbench/dut/mips/c/ad/funct	00
/testbench/dut/mips/c/ad/aluop	0
/testbench/dut/mips/c/ad/alucontrol	2
Now	625 ps
Cursor 1	0 ps

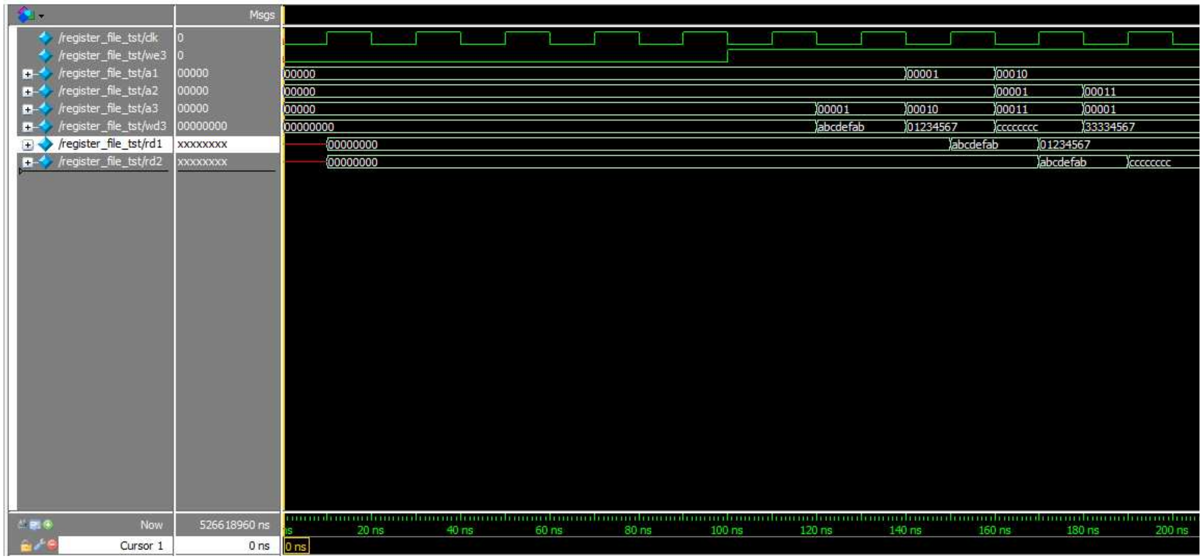
Registers

The testbench code for the register is as follows:

```
module register_file_tb();

reg clk, we3;
reg [4:0] a1, a2,a3;
reg [31:0] wd3;
wire [31:0] rd1, rd2;

register_file UUT (clk, we3,a1,a2, a3,wd3, rd1, rd2);

initial begin
wd3 =32'b0;
a1 = 5'b0;
a2 = 5'b0;
a3 = 5'b0;
we3 = 1'b0;
clk = 1'b0;

#100
we3=1'b1;

#20
wd3=32'habcd_efab;
a1=5'h0;
a2=5'h0;
a3=5'h1;

#20
wd3=32'h0123_4567;
a1=5'h1;
a2=5'h0;
a3=5'h2;

#20
wd3=32'hcccc_cccc;
a1=5'h2;
a2=5'h1;
a3=5'h3;

#20
wd3=32'h3333_4567;
a1=5'h2;
```

```
a2=5'h3;
a3=5'h1;
end

always begin
#10;
clk =~clk;
end
endmodule
```

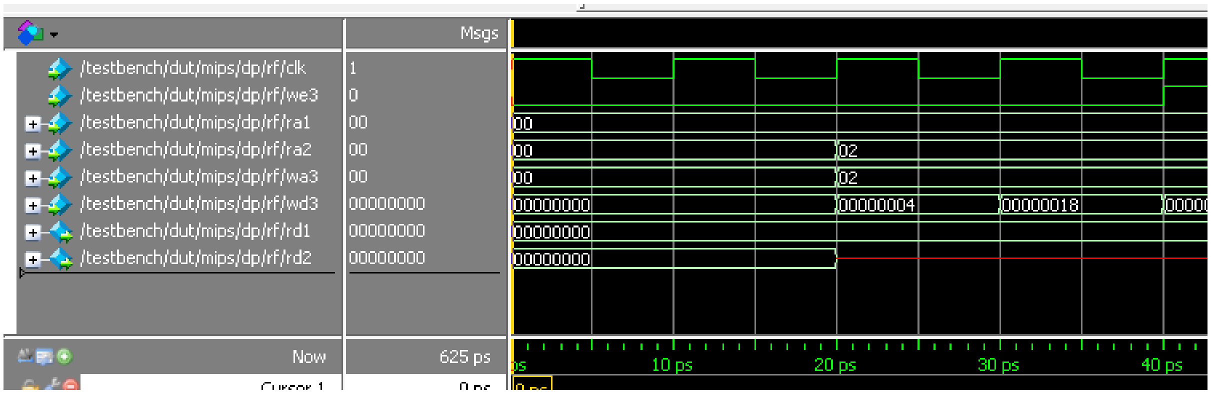
Output waveform:

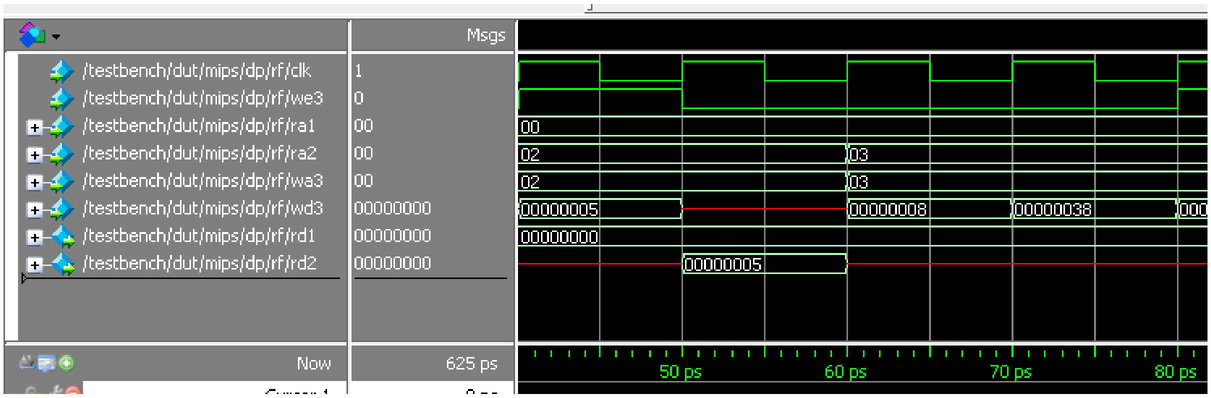

Msgs
/register_file_tst/clk 0
/register_file_tst/we3 0
/register_file_tst/a1 00000
/register_file_tst/a2 00000
/register_file_tst/a3 00000
/register_file_tst/wd3 00000000
/register_file_tst/rd1 xxxxxxxx
/register_file_tst/rd2 xxxxxxxx
00000 00001 00010
00000 00001 00011
00000 00001 00010 00011 00001
00000000 abcdefab 01234567 cccccccc 33334567
00000000 abcdefab 01234567
00000000 abcdefab cccccccc
Now 526618960 ns
Cursor 1 0 ns
ns 20 ns 40 ns 60 ns 80 ns 100 ns 120 ns 140 ns 160 ns 180 ns 200 ns
0 ns

First, the storages are empty. On the first input, the data will be saved first at the Register File or the Rf, the Rd1 and the Rd2 will still be empty. As you input another piece of data, the first data inputted will move to the Rd1 and the newly added data will be stored in the Rf and the Rd2 will still be empty. Then, when you add another data to the storage, the data in the Rd1 will move to the Rd2, and the data in the Rf will move to the Rd1 and the Rf will hold the new data added. This will fill all the storages with the data that has been added. As you added a new data to the storage, the data in the Rd2 will be removed and replaced by the data in the Rd1 and the Rd1 will hold the data from Rf, then the Rf will hold the added data in the storages.

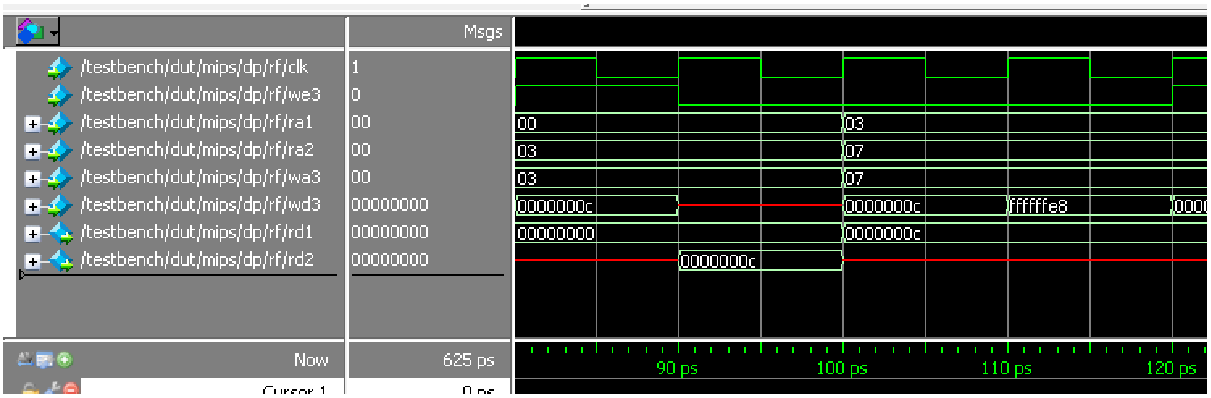
Actual Waveform of Register File using MIPS Assembly Code:

The data is stored at the WRITEBACK state of the processor.

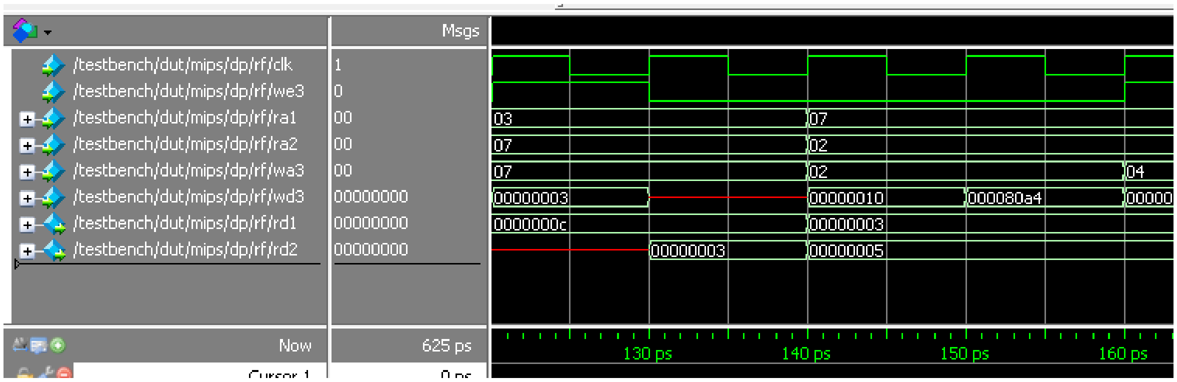

Msgs
/testbench/dut/mips/dp/rf/clk 1
/testbench/dut/mips/dp/rf/we3 0
/testbench/dut/mips/dp/rf/ra1 00
/testbench/dut/mips/dp/rf/ra2 00
/testbench/dut/mips/dp/rf/wa3 00
/testbench/dut/mips/dp/rf/wd3 00000000
/testbench/dut/mips/dp/rf/rd1 00000000
/testbench/dut/mips/dp/rf/rd2 00000000
00
00 02
00 02
00000000 00000004 00000018 0000
00000000
00000000
Now 625 ps
Cursor 1 0 ps
ps 10 ps 20 ps 30 ps 40 ps


Msgs
/testbench/dut/mips/dp/rf/clk 1
/testbench/dut/mips/dp/rf/we3 0
/testbench/dut/mips/dp/rf/ra1 00
/testbench/dut/mips/dp/rf/ra2 00
/testbench/dut/mips/dp/rf/wa3 00
/testbench/dut/mips/dp/rf/wd3 00000000
/testbench/dut/mips/dp/rf/rd1 00000000
/testbench/dut/mips/dp/rf/rd2 00000000
00
02 03
02 03
00000005 00000008 00000038 000
00000000
00000005
Now 625 ps
Cursor 1 0 ps
50 ps 60 ps 70 ps 80 ps

$2 = 5 at 40ps

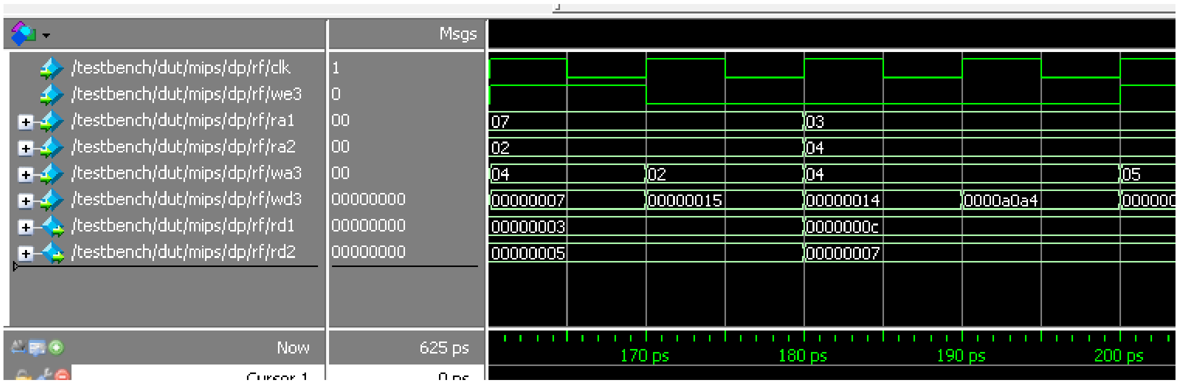

Msgs
/testbench/dut/mips/dp/rf/clk 1
/testbench/dut/mips/dp/rf/we3 0
/testbench/dut/mips/dp/rf/ra1 00
/testbench/dut/mips/dp/rf/ra2 00
/testbench/dut/mips/dp/rf/wa3 00
/testbench/dut/mips/dp/rf/wd3 00000000
/testbench/dut/mips/dp/rf/rd1 00000000
/testbench/dut/mips/dp/rf/rd2 00000000
00 03
03 07
03 07
0000000c 0000000c ffffffe8 000
00000000 0000000c
0000000c
Now 625 ps
Cursor 1 0 ps
90 ps 100 ps 110 ps 120 ps

$3 = 12 at 80ps



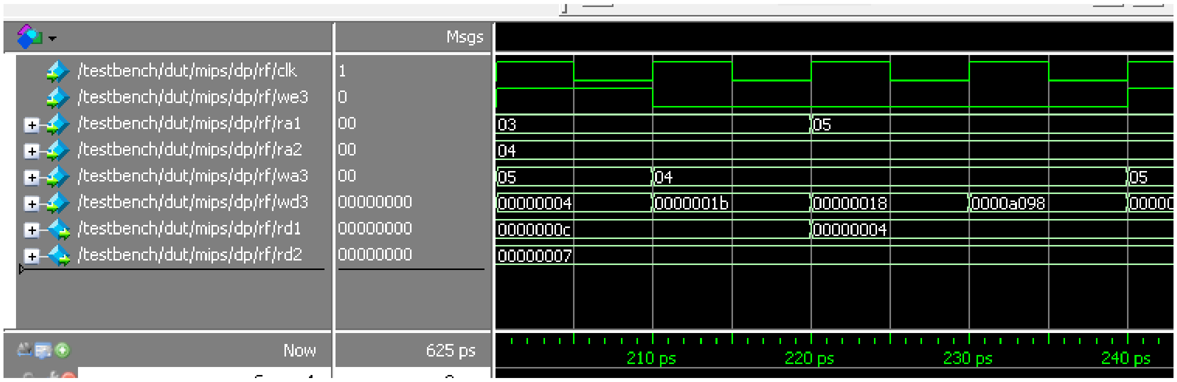
```
                                   Msgs
/testbench/dut/mips/dp/rf/clk      1
/testbench/dut/mips/dp/rf/we3      0
/testbench/dut/mips/dp/rf/ra1      00
/testbench/dut/mips/dp/rf/ra2      00
/testbench/dut/mips/dp/rf/wa3      00
/testbench/dut/mips/dp/rf/wd3      00000000
/testbench/dut/mips/dp/rf/rd1      00000000
/testbench/dut/mips/dp/rf/rd2      00000000
Now                                625 ps
Cursor 1                           0 ps
```

$7 = 3 at 120ps



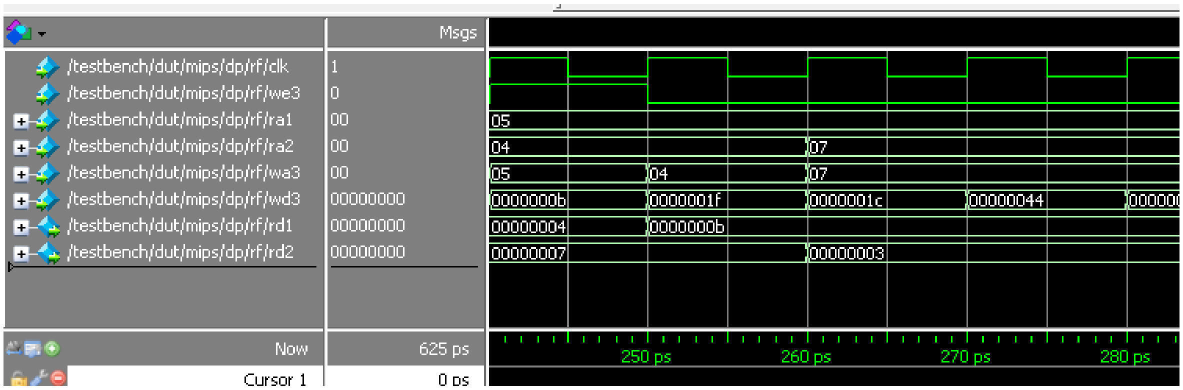
```
                                   Msgs
/testbench/dut/mips/dp/rf/clk      1
/testbench/dut/mips/dp/rf/we3      0
/testbench/dut/mips/dp/rf/ra1      00
/testbench/dut/mips/dp/rf/ra2      00
/testbench/dut/mips/dp/rf/wa3      00
/testbench/dut/mips/dp/rf/wd3      00000000
/testbench/dut/mips/dp/rf/rd1      00000000
/testbench/dut/mips/dp/rf/rd2      00000000
Now                                625 ps
Cursor 1                           0 ps
```

$4 = 7 at 160ps



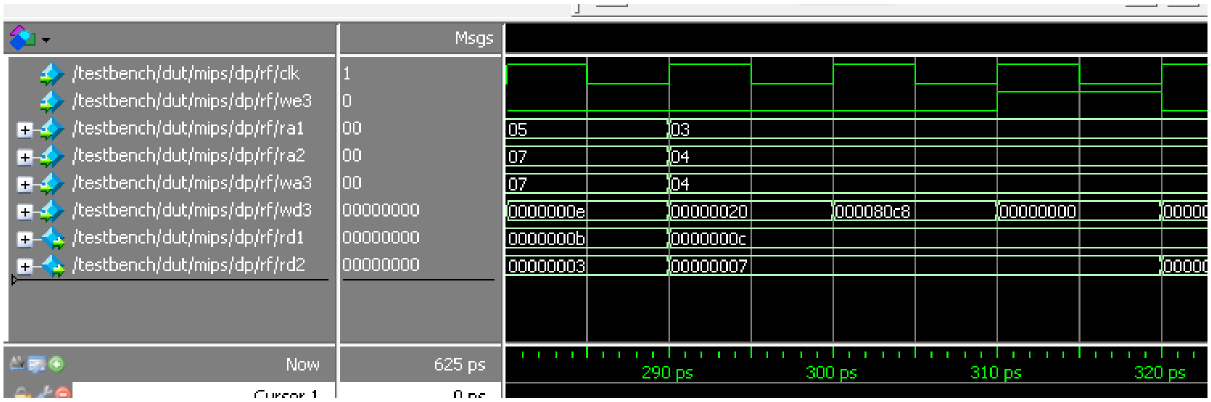
```
                                   Msgs
/testbench/dut/mips/dp/rf/clk      1
/testbench/dut/mips/dp/rf/we3      0
/testbench/dut/mips/dp/rf/ra1      00
/testbench/dut/mips/dp/rf/ra2      00
/testbench/dut/mips/dp/rf/wa3      00
/testbench/dut/mips/dp/rf/wd3      00000000
/testbench/dut/mips/dp/rf/rd1      00000000
/testbench/dut/mips/dp/rf/rd2      00000000
Now                                625 ps
```

$5 = 4 at 200ps



```
                                   Msgs
/testbench/dut/mips/dp/rf/clk      1
/testbench/dut/mips/dp/rf/we3      0
/testbench/dut/mips/dp/rf/ra1      00
/testbench/dut/mips/dp/rf/ra2      00
/testbench/dut/mips/dp/rf/wa3      00
/testbench/dut/mips/dp/rf/wd3      00000000
/testbench/dut/mips/dp/rf/rd1      00000000
/testbench/dut/mips/dp/rf/rd2      00000000
Now                                625 ps
Cursor 1                           0 ps
```

$5 = 11 at 240ps



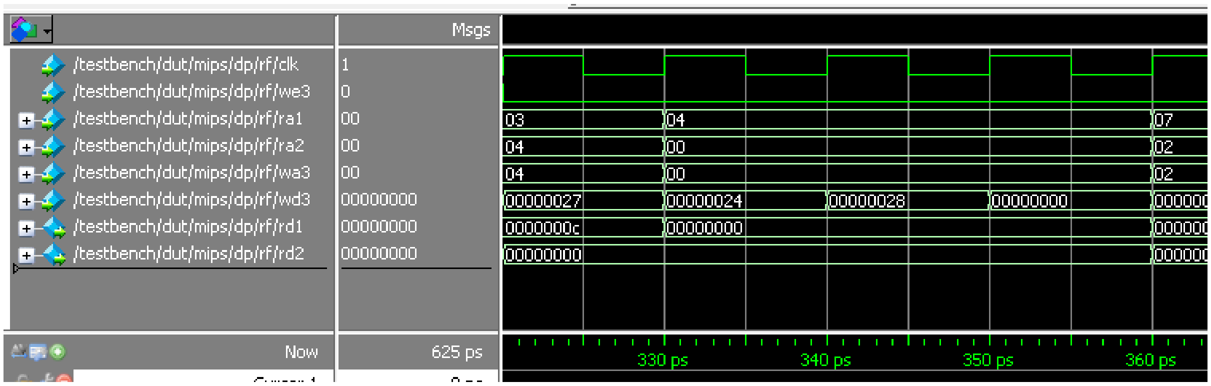
Msgs

/testbench/dut/mips/dp/rf/clk 1
/testbench/dut/mips/dp/rf/we3 0
/testbench/dut/mips/dp/rf/ra1 00
/testbench/dut/mips/dp/rf/ra2 00
/testbench/dut/mips/dp/rf/wa3 00
/testbench/dut/mips/dp/rf/wd3 00000000
/testbench/dut/mips/dp/rf/rd1 00000000
/testbench/dut/mips/dp/rf/rd2 00000000

Now 625 ps

Cursor 1 0 ps

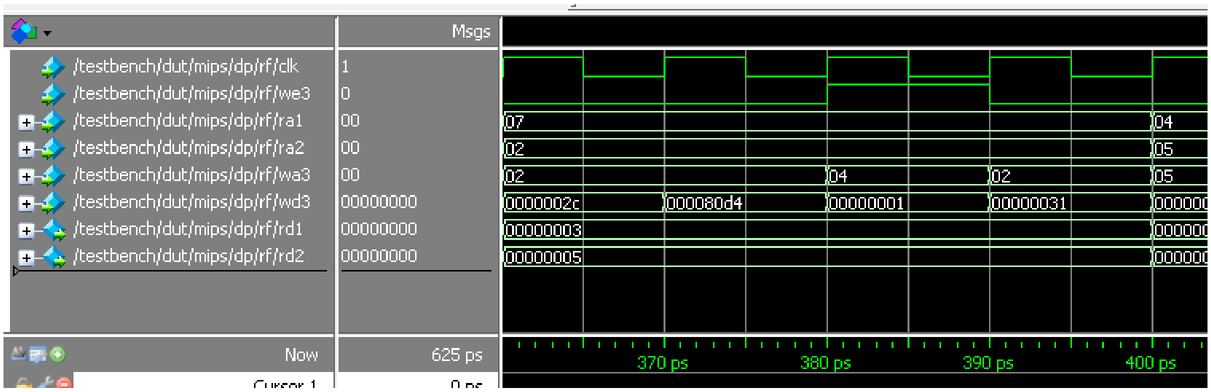
$4 = 0 at 310ps



Msgs

/testbench/dut/mips/dp/rf/clk 1
/testbench/dut/mips/dp/rf/we3 0
/testbench/dut/mips/dp/rf/ra1 00
/testbench/dut/mips/dp/rf/ra2 00
/testbench/dut/mips/dp/rf/wa3 00
/testbench/dut/mips/dp/rf/wd3 00000000
/testbench/dut/mips/dp/rf/rd1 00000000
/testbench/dut/mips/dp/rf/rd2 00000000

Now 625 ps



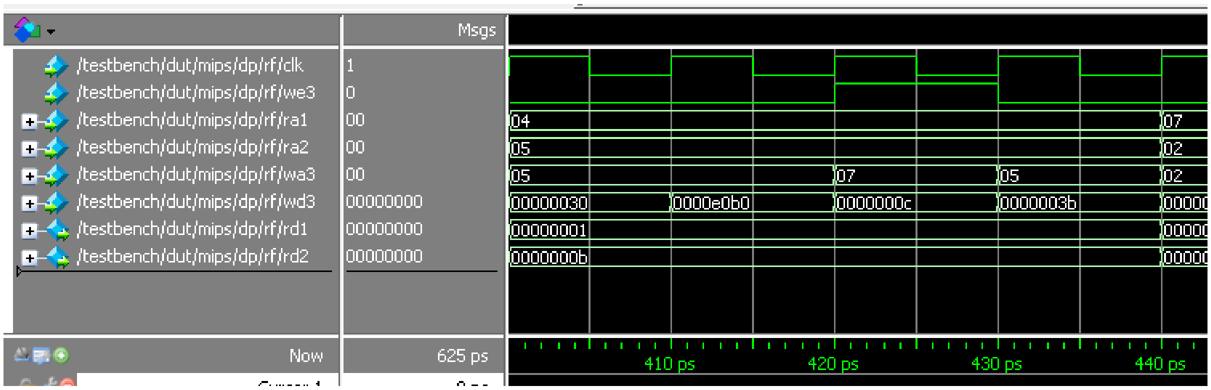
Msgs

/testbench/dut/mips/dp/rf/clk 1
/testbench/dut/mips/dp/rf/we3 0
/testbench/dut/mips/dp/rf/ra1 00
/testbench/dut/mips/dp/rf/ra2 00
/testbench/dut/mips/dp/rf/wa3 00
/testbench/dut/mips/dp/rf/wd3 00000000
/testbench/dut/mips/dp/rf/rd1 00000000
/testbench/dut/mips/dp/rf/rd2 00000000

Now 625 ps

Cursor 1 0 ps

$4 = 1 at 380ps



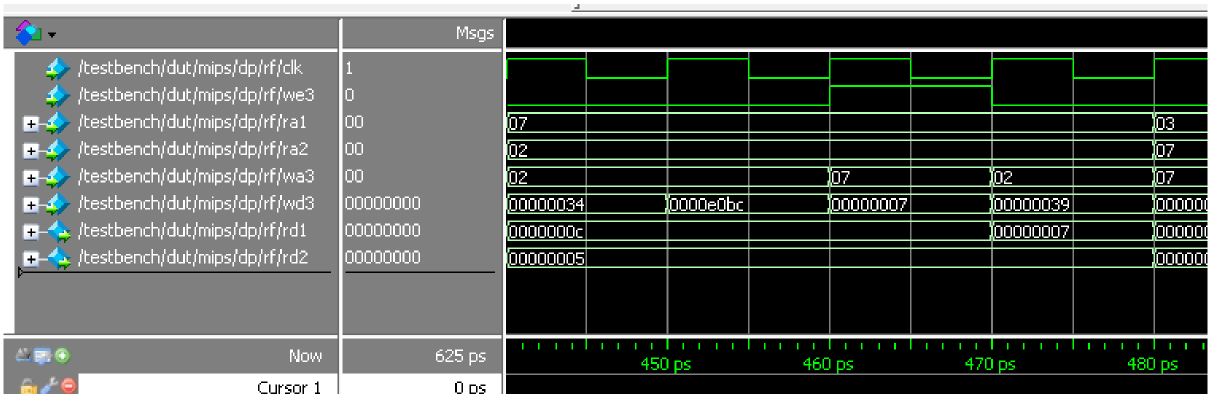
Msgs

/testbench/dut/mips/dp/rf/clk 1
/testbench/dut/mips/dp/rf/we3 0
/testbench/dut/mips/dp/rf/ra1 00
/testbench/dut/mips/dp/rf/ra2 00
/testbench/dut/mips/dp/rf/wa3 00
/testbench/dut/mips/dp/rf/wd3 00000000
/testbench/dut/mips/dp/rf/rd1 00000000
/testbench/dut/mips/dp/rf/rd2 00000000

Now 625 ps

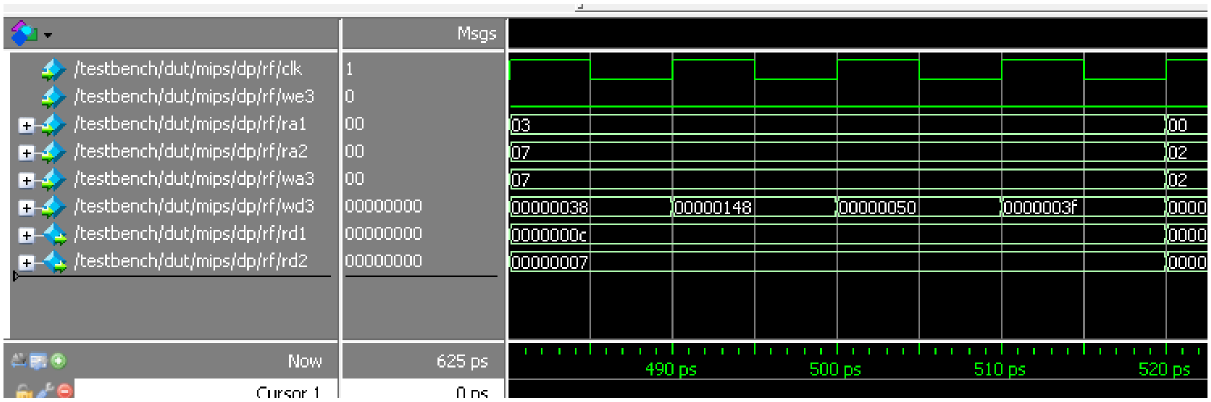
Cursor 1 0 ps

$7 = 12 at 420ps

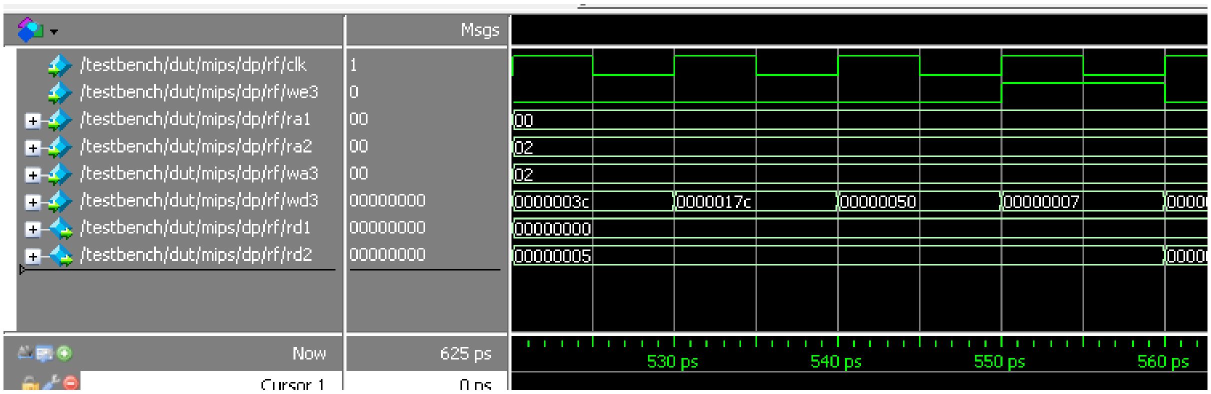


```
                                  Msgs
/testbench/dut/mips/dp/rf/clk     1
/testbench/dut/mips/dp/rf/we3     0
/testbench/dut/mips/dp/rf/ra1     00
/testbench/dut/mips/dp/rf/ra2     00
/testbench/dut/mips/dp/rf/wa3     00
/testbench/dut/mips/dp/rf/wd3     00000000
/testbench/dut/mips/dp/rf/rd1     00000000
/testbench/dut/mips/dp/rf/rd2     00000000
Now                               625 ps
Cursor 1                          0 ps
```

$7 = 7 at 460ps

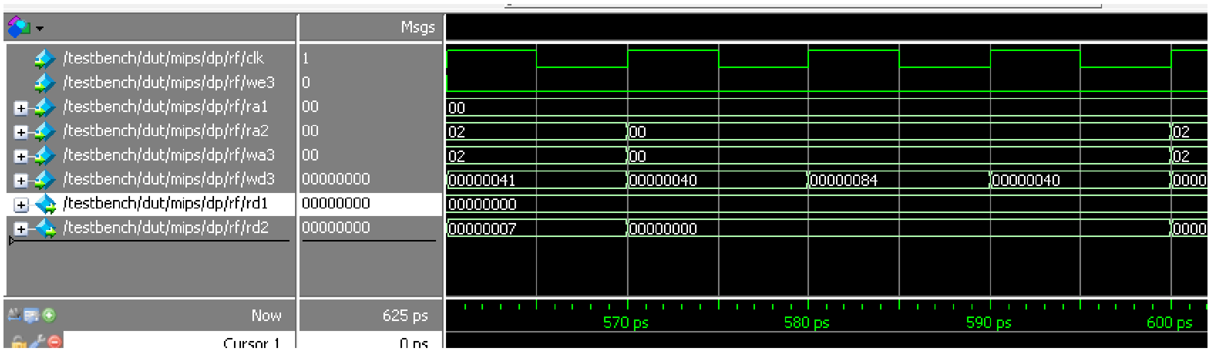


```
                                  Msgs
/testbench/dut/mips/dp/rf/clk     1
/testbench/dut/mips/dp/rf/we3     0
/testbench/dut/mips/dp/rf/ra1     00
/testbench/dut/mips/dp/rf/ra2     00
/testbench/dut/mips/dp/rf/wa3     00
/testbench/dut/mips/dp/rf/wd3     00000000
/testbench/dut/mips/dp/rf/rd1     00000000
/testbench/dut/mips/dp/rf/rd2     00000000
Now                               625 ps
Cursor 1                          0 ps
```

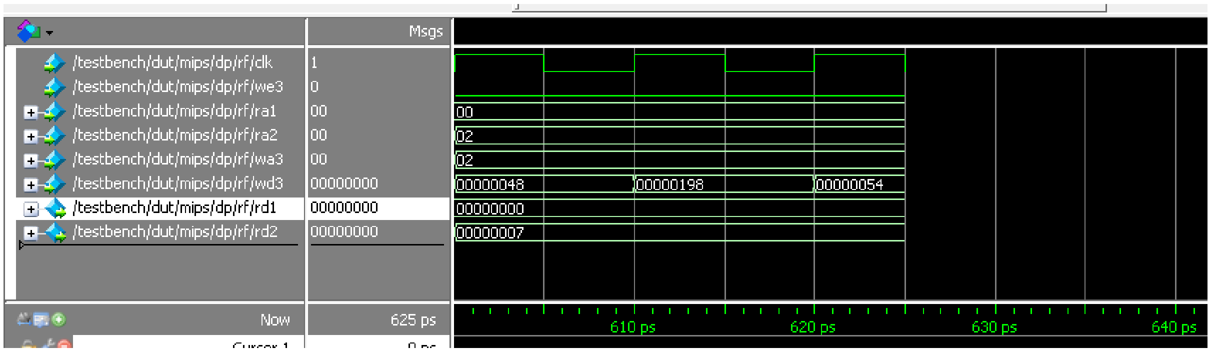


```
                                  Msgs
/testbench/dut/mips/dp/rf/clk     1
/testbench/dut/mips/dp/rf/we3     0
/testbench/dut/mips/dp/rf/ra1     00
/testbench/dut/mips/dp/rf/ra2     00
/testbench/dut/mips/dp/rf/wa3     00
/testbench/dut/mips/dp/rf/wd3     00000000
/testbench/dut/mips/dp/rf/rd1     00000000
/testbench/dut/mips/dp/rf/rd2     00000000
Now                               625 ps
Cursor 1                          0 ps
```

$2 = 7 at 550ps



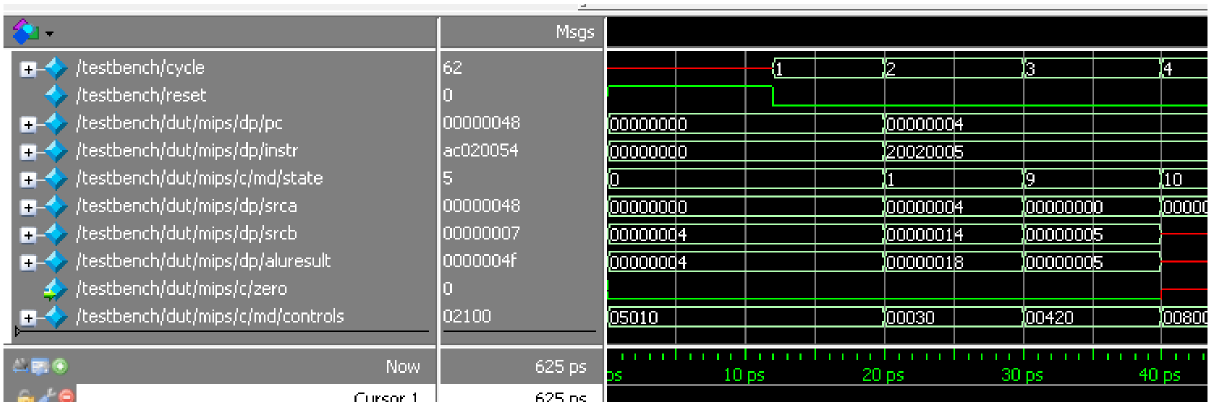
```
                                  Msgs
/testbench/dut/mips/dp/rf/clk     1
/testbench/dut/mips/dp/rf/we3     0
/testbench/dut/mips/dp/rf/ra1     00
/testbench/dut/mips/dp/rf/ra2     00
/testbench/dut/mips/dp/rf/wa3     00
/testbench/dut/mips/dp/rf/wd3     00000000
/testbench/dut/mips/dp/rf/rd1     00000000
/testbench/dut/mips/dp/rf/rd2     00000000
Now                               625 ps
Cursor 1                          0 ps
```



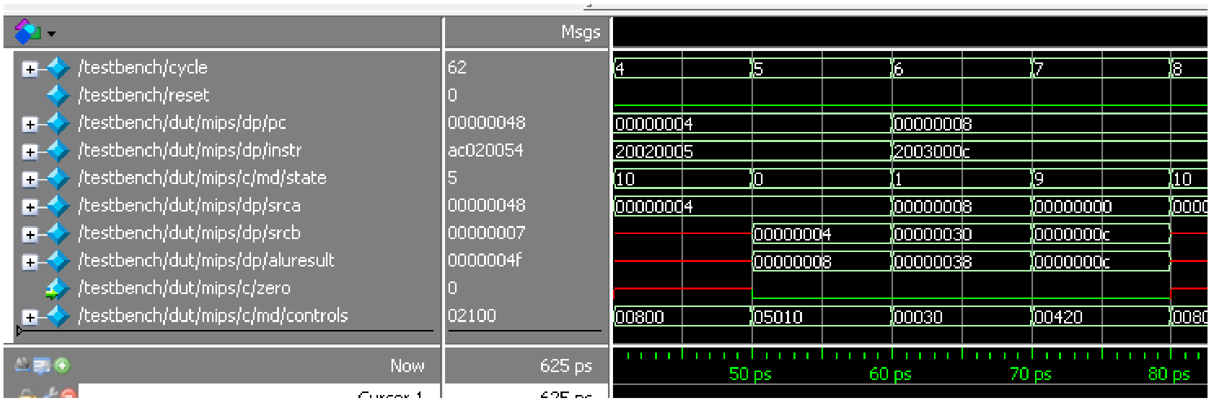
```
/testbench/dut/mips/dp/rf/clk   1
/testbench/dut/mips/dp/rf/we3   0
/testbench/dut/mips/dp/rf/ra1   00
/testbench/dut/mips/dp/rf/ra2   00
/testbench/dut/mips/dp/rf/wa3   00
/testbench/dut/mips/dp/rf/wd3   00000000
/testbench/dut/mips/dp/rf/rd1   00000000
/testbench/dut/mips/dp/rf/rd2   00000000
Now                             625 ps
```

Actual Instruction Trace Waveform

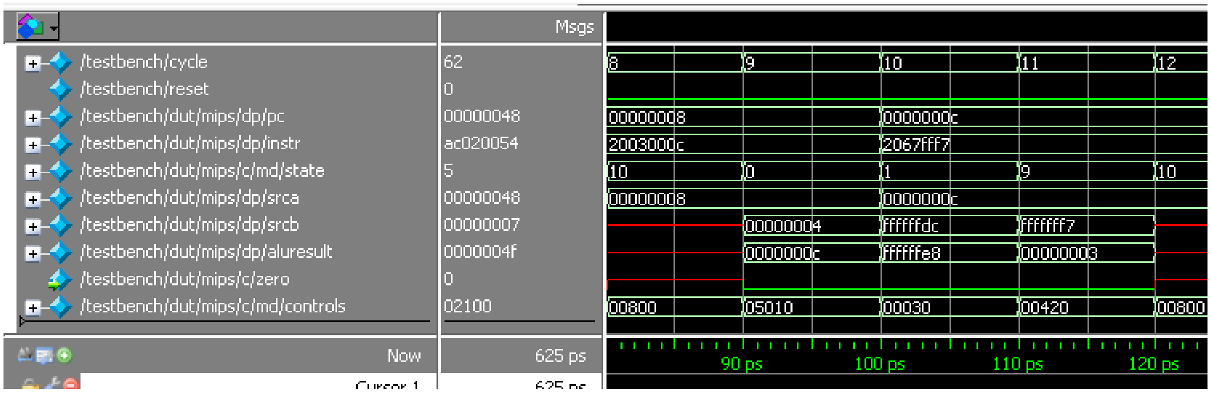
The following waveform contains the instruction trace of the multicycle processor, the cycle, reset, program counter, instruction, state, the ALU inputs and result, zero flag and the FSM control word.



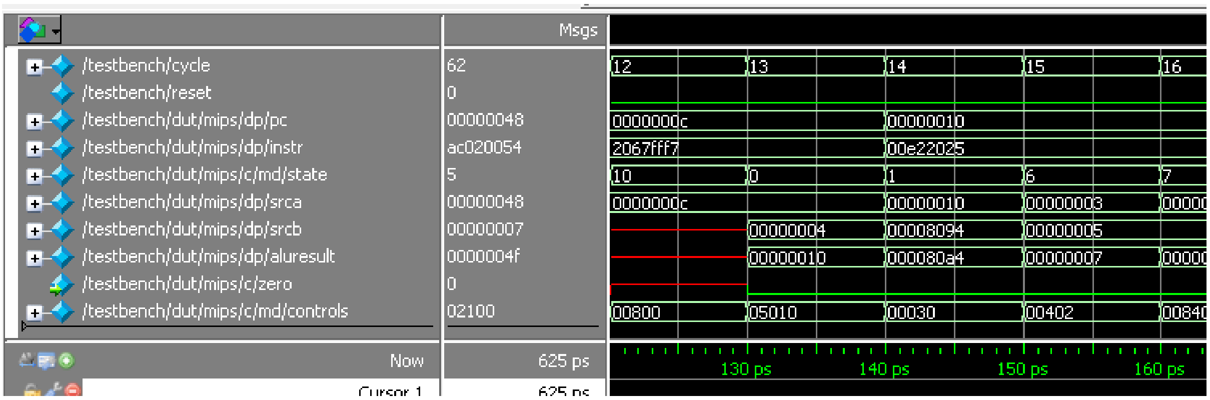
```
/testbench/cycle                   62
/testbench/reset                   0
/testbench/dut/mips/dp/pc          00000048
/testbench/dut/mips/dp/instr       ac020054
/testbench/dut/mips/c/md/state     5
/testbench/dut/mips/dp/srca        00000048
/testbench/dut/mips/dp/srcb        00000007
/testbench/dut/mips/dp/aluresult   0000004f
/testbench/dut/mips/c/zero         0
/testbench/dut/mips/c/md/controls  02100
Now                                625 ps
```



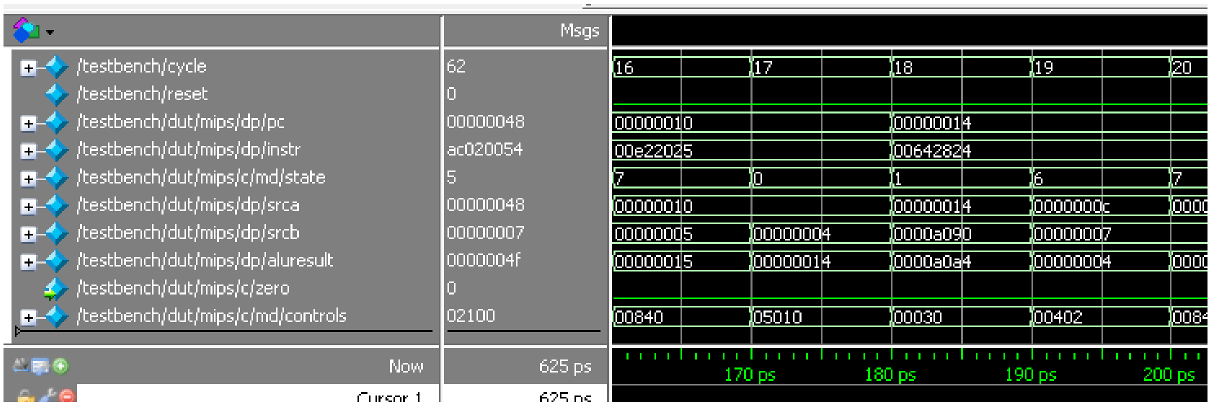
```
/testbench/cycle                   62
/testbench/reset                   0
/testbench/dut/mips/dp/pc          00000048
/testbench/dut/mips/dp/instr       ac020054
/testbench/dut/mips/c/md/state     5
/testbench/dut/mips/dp/srca        00000048
/testbench/dut/mips/dp/srcb        00000007
/testbench/dut/mips/dp/aluresult   0000004f
/testbench/dut/mips/c/zero         0
/testbench/dut/mips/c/md/controls  02100
Now                                625 ps
```



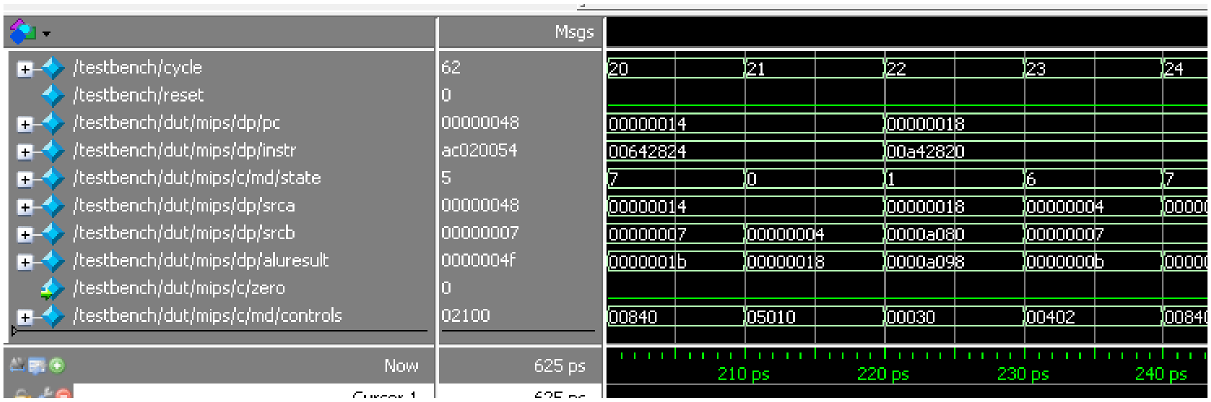
```
                                    Msgs
/testbench/cycle                    62          8 | 9 | 10 | 11 | 12
/testbench/reset                    0
/testbench/dut/mips/dp/pc           00000048    00000008 | 0000000c
/testbench/dut/mips/dp/instr        ac020054    2003000c | 2067fff7
/testbench/dut/mips/c/md/state      5           10 | 0 | 1 | 9 | 10
/testbench/dut/mips/dp/srca         00000048    00000008 | 0000000c
/testbench/dut/mips/dp/srcb         00000007    00000004 | ffffffdc | fffffff7
/testbench/dut/mips/dp/aluresult    0000004f    0000000c | ffffffe8 | 00000003
/testbench/dut/mips/c/zero          0
/testbench/dut/mips/c/md/controls   02100       00800 | 05010 | 00030 | 00420 | 00800
                               Now  625 ps      90 ps   100 ps   110 ps   120 ps
                          Cursor 1  625 ps
```



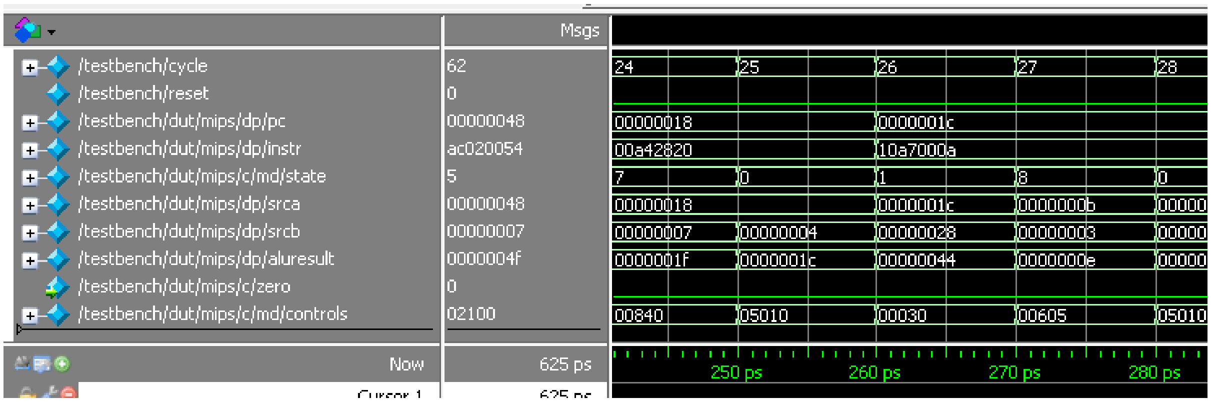
```
                                    Msgs
/testbench/cycle                    62          12 | 13 | 14 | 15 | 16
/testbench/reset                    0
/testbench/dut/mips/dp/pc           00000048    0000000c | 00000010
/testbench/dut/mips/dp/instr        ac020054    2067fff7 | 00e22025
/testbench/dut/mips/c/md/state      5           10 | 0 | 1 | 6 | 7
/testbench/dut/mips/dp/srca         00000048    0000000c | 00000010 | 00000003 | 0000
/testbench/dut/mips/dp/srcb         00000007    00000004 | 00008094 | 00000005
/testbench/dut/mips/dp/aluresult    0000004f    00000010 | 000080a4 | 00000007 | 0000
/testbench/dut/mips/c/zero          0
/testbench/dut/mips/c/md/controls   02100       00800 | 05010 | 00030 | 00402 | 0084
                               Now  625 ps      130 ps   140 ps   150 ps   160 ps
                          Cursor 1  625 ps
```



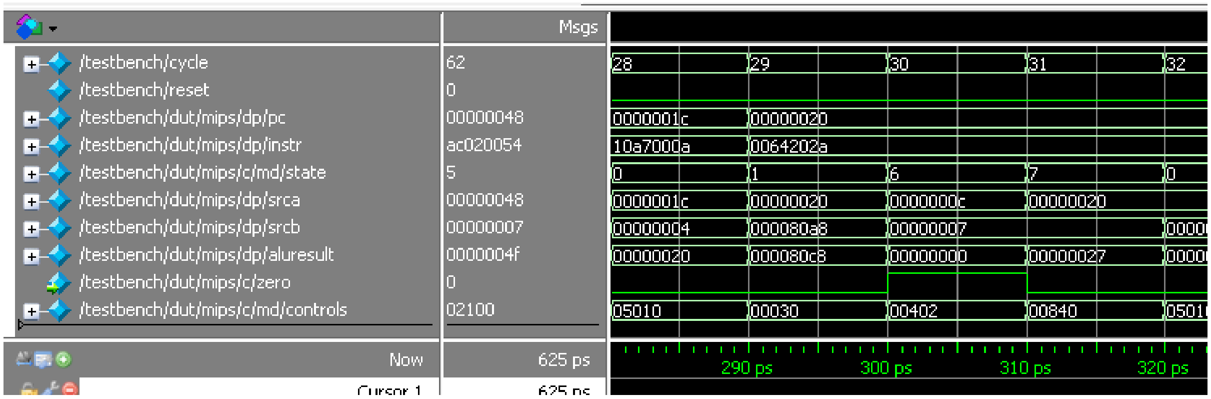
```
                                    Msgs
/testbench/cycle                    62          16 | 17 | 18 | 19 | 20
/testbench/reset                    0
/testbench/dut/mips/dp/pc           00000048    00000010 | 00000014
/testbench/dut/mips/dp/instr        ac020054    00e22025 | 00642824
/testbench/dut/mips/c/md/state      5           7 | 0 | 1 | 6 | 7
/testbench/dut/mips/dp/srca         00000048    00000010 | 00000014 | 0000000c | 0000
/testbench/dut/mips/dp/srcb         00000007    00000005 | 00000004 | 0000a090 | 00000007
/testbench/dut/mips/dp/aluresult    0000004f    00000015 | 00000014 | 0000a0a4 | 00000004 | 0000
/testbench/dut/mips/c/zero          0
/testbench/dut/mips/c/md/controls   02100       00840 | 05010 | 00030 | 00402 | 0084
                               Now  625 ps      170 ps   180 ps   190 ps   200 ps
                          Cursor 1  625 ps
```



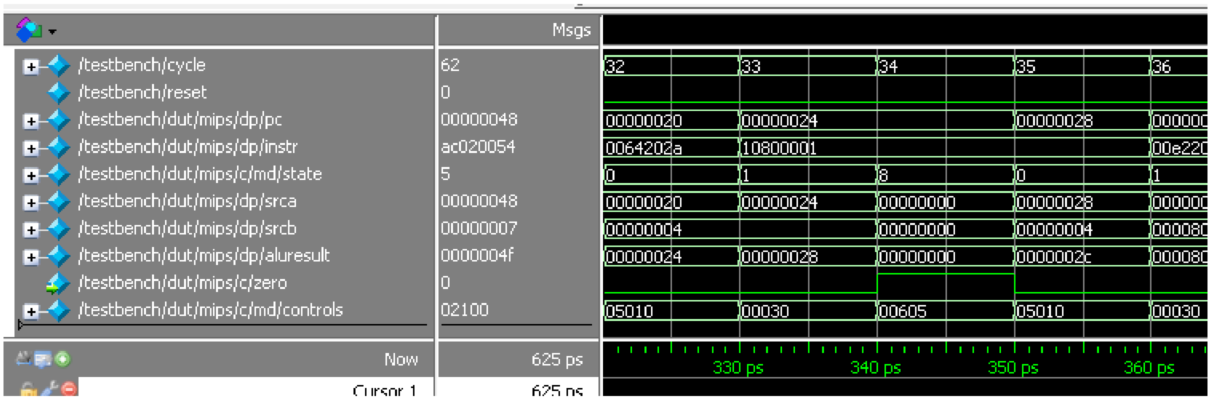
```
                                    Msgs
/testbench/cycle                    62          20 | 21 | 22 | 23 | 24
/testbench/reset                    0
/testbench/dut/mips/dp/pc           00000048    00000014 | 00000018
/testbench/dut/mips/dp/instr        ac020054    00642824 | 00a42820
/testbench/dut/mips/c/md/state      5           7 | 0 | 1 | 6 | 7
/testbench/dut/mips/dp/srca         00000048    00000014 | 00000018 | 00000004 | 0000
/testbench/dut/mips/dp/srcb         00000007    00000007 | 00000004 | 0000a080 | 00000007
/testbench/dut/mips/dp/aluresult    0000004f    0000001b | 00000018 | 0000a098 | 0000000b | 0000
/testbench/dut/mips/c/zero          0
/testbench/dut/mips/c/md/controls   02100       00840 | 05010 | 00030 | 00402 | 0084
                               Now  625 ps      210 ps   220 ps   230 ps   240 ps
                          Cursor 1  625 ps
```



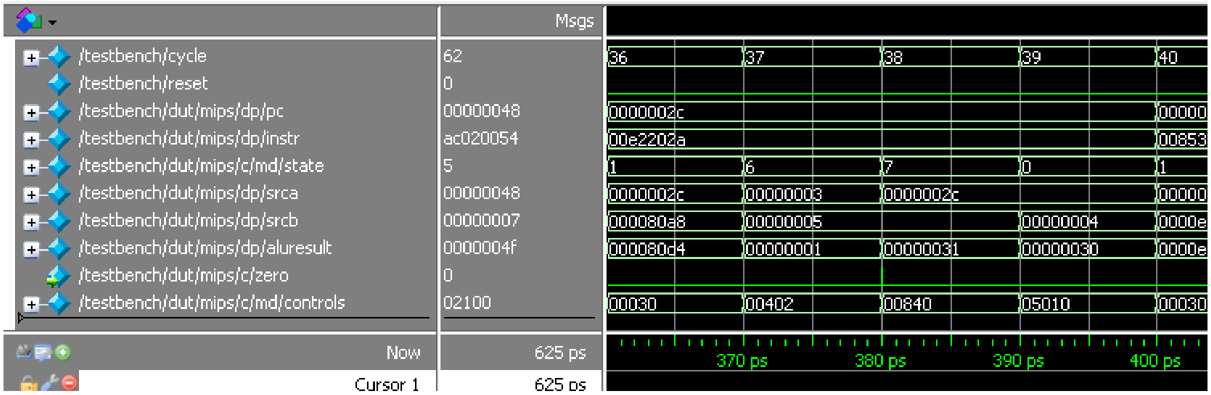
```
                                     Msgs
/testbench/cycle                     62          24        25        26        27        28
/testbench/reset                     0
/testbench/dut/mips/dp/pc            00000048    00000018            0000001c
/testbench/dut/mips/dp/instr         ac020054    00a42820            10a7000a
/testbench/dut/mips/c/md/state       5           7         0         1         8         0
/testbench/dut/mips/dp/srca          00000048    00000018            0000001c  0000000b  00000
/testbench/dut/mips/dp/srcb          00000007    00000007  00000004  00000028  00000003  00000
/testbench/dut/mips/dp/aluresult     0000004f    0000001f  0000001c  00000044  0000000e  00000
/testbench/dut/mips/c/zero           0
/testbench/dut/mips/c/md/controls    02100       00840     05010     00030     00605     05010
Now                                  625 ps           250 ps    260 ps    270 ps    280 ps
Cursor 1                             625 ps
```



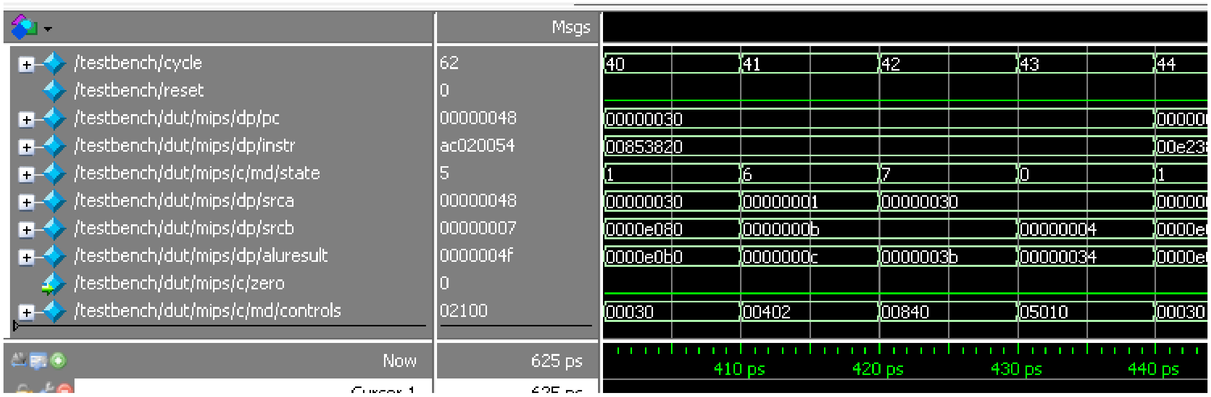
```
                                     Msgs
/testbench/cycle                     62          28        29        30        31        32
/testbench/reset                     0
/testbench/dut/mips/dp/pc            00000048    0000001c  00000020
/testbench/dut/mips/dp/instr         ac020054    10a7000a  0064202a
/testbench/dut/mips/c/md/state       5           0         1         6         7         0
/testbench/dut/mips/dp/srca          00000048    0000001c  00000020  0000000c  00000020
/testbench/dut/mips/dp/srcb          00000007    00000004  000080a8  00000007            0000(
/testbench/dut/mips/dp/aluresult     0000004f    00000020  000080c8  00000000  00000027  0000(
/testbench/dut/mips/c/zero           0
/testbench/dut/mips/c/md/controls    02100       05010     00030     00402     00840     0501(
Now                                  625 ps           290 ps    300 ps    310 ps    320 ps
Cursor 1                             625 ps
```



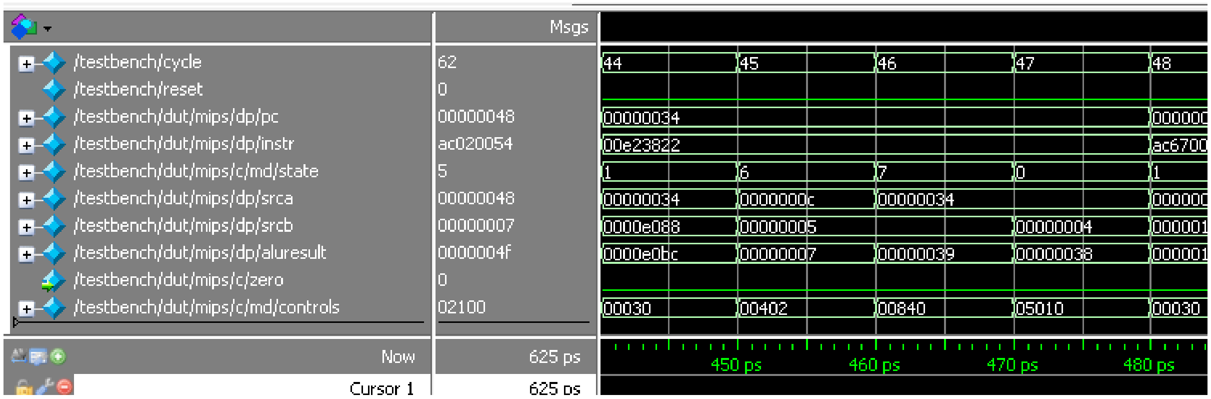
```
                                     Msgs
/testbench/cycle                     62          32        33        34        35        36
/testbench/reset                     0
/testbench/dut/mips/dp/pc            00000048    00000020  00000024            00000028  000000
/testbench/dut/mips/dp/instr         ac020054    0064202a  10800001                      00e220
/testbench/dut/mips/c/md/state       5           0         1         8         0         1
/testbench/dut/mips/dp/srca          00000048    00000020  00000024  00000000  00000028  000000
/testbench/dut/mips/dp/srcb          00000007    00000004            00000000  00000004  000080
/testbench/dut/mips/dp/aluresult     0000004f    00000024  00000028  00000000  0000002c  000080
/testbench/dut/mips/c/zero           0
/testbench/dut/mips/c/md/controls    02100       05010     00030     00605     05010     00030
Now                                  625 ps           330 ps    340 ps    350 ps    360 ps
Cursor 1                             625 ps
```



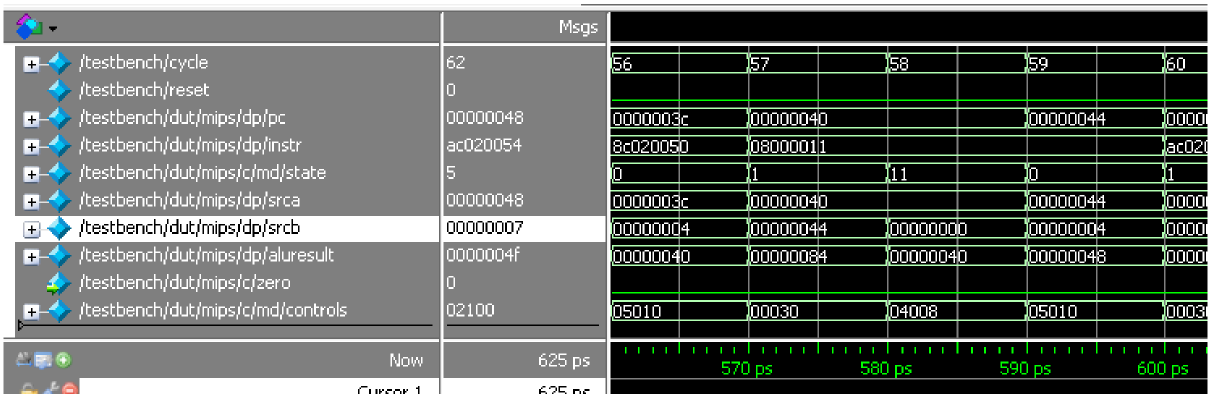
```
                                     Msgs
/testbench/cycle                     62          36        37        38        39        40
/testbench/reset                     0
/testbench/dut/mips/dp/pc            00000048    0000002c                                00000
/testbench/dut/mips/dp/instr         ac020054    00e2202a                                00853
/testbench/dut/mips/c/md/state       5           1         6         7         0         1
/testbench/dut/mips/dp/srca          00000048    0000002c  00000003  0000002c            00000
/testbench/dut/mips/dp/srcb          00000007    000080a8  00000005            00000004  0000e
/testbench/dut/mips/dp/aluresult     0000004f    000080d4  00000001  00000031  00000030  0000e
/testbench/dut/mips/c/zero           0
/testbench/dut/mips/c/md/controls    02100       00030     00402     00840     05010     00030
Now                                  625 ps           370 ps    380 ps    390 ps    400 ps
Cursor 1                             625 ps
```



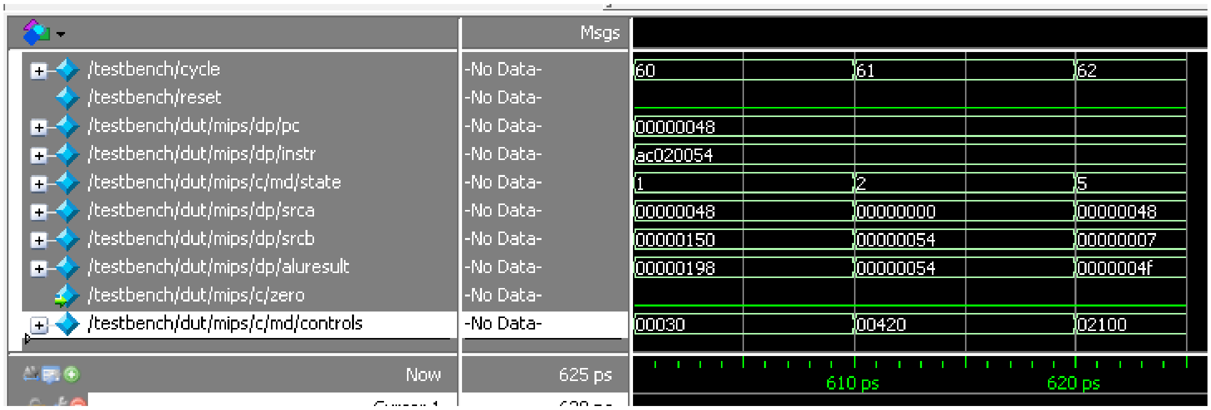
```
                                            Msgs
/testbench/cycle                            62        40         41         42         43         44
/testbench/reset                            0
/testbench/dut/mips/dp/pc                   00000048  00000030                                    00000C
/testbench/dut/mips/dp/instr                ac020054  00853820                                    00e23
/testbench/dut/mips/c/md/state              5         1          6          7          0          1
/testbench/dut/mips/dp/srca                 00000048  00000030   00000001   00000030              00000C
/testbench/dut/mips/dp/srcb                 00000007  0000e080   0000000b              00000004   0000e
/testbench/dut/mips/dp/aluresult            0000004f  0000e0b0   0000000c   0000003b   00000034   0000e
/testbench/dut/mips/c/zero                  0
/testbench/dut/mips/c/md/controls           02100     00030      00402      00840      05010      00030
                                       Now  625 ps          410 ps     420 ps     430 ps     440 ps
                                  Cursor 1  625 ps
```



```
                                            Msgs
/testbench/cycle                            62        44         45         46         47         48
/testbench/reset                            0
/testbench/dut/mips/dp/pc                   00000048  00000034                                    00000C
/testbench/dut/mips/dp/instr                ac020054  00e23822                                    ac6700
/testbench/dut/mips/c/md/state              5         1          6          7          0          1
/testbench/dut/mips/dp/srca                 00000048  00000034   0000000c   00000034              00000C
/testbench/dut/mips/dp/srcb                 00000007  0000e088   00000005              00000004   000001
/testbench/dut/mips/dp/aluresult            0000004f  0000e0bc   00000007   00000039   00000038   000001
/testbench/dut/mips/c/zero                  0
/testbench/dut/mips/c/md/controls           02100     00030      00402      00840      05010      00030
                                       Now  625 ps          450 ps     460 ps     470 ps     480 ps
                                  Cursor 1  625 ps
```



```
                                            Msgs
/testbench/cycle                            62        56         57         58         59         60
/testbench/reset                            0
/testbench/dut/mips/dp/pc                   00000048  0000003c   00000040              00000044   0000C
/testbench/dut/mips/dp/instr                ac020054  8c020050   08000011                         ac02C
/testbench/dut/mips/c/md/state              5         0          1          11         0          1
/testbench/dut/mips/dp/srca                 00000048  0000003c   00000040              00000044   0000C
/testbench/dut/mips/dp/srcb                 00000007  00000004   00000044   00000000   00000004   0000C
/testbench/dut/mips/dp/aluresult            0000004f  00000040   00000084   00000040   00000048   0000C
/testbench/dut/mips/c/zero                  0
/testbench/dut/mips/c/md/controls           02100     05010      00030      04008      05010      0003
                                       Now  625 ps          570 ps     580 ps     590 ps     600 ps
                                  Cursor 1  625 ps
```



```
                                            Msgs
/testbench/cycle                            -No Data-  60          61          62
/testbench/reset                            -No Data-
/testbench/dut/mips/dp/pc                   -No Data-  00000048
/testbench/dut/mips/dp/instr                -No Data-  ac020054
/testbench/dut/mips/c/md/state              -No Data-  1           2           5
/testbench/dut/mips/dp/srca                 -No Data-  00000048    00000000    00000048
/testbench/dut/mips/dp/srcb                 -No Data-  00000150    00000054    00000007
/testbench/dut/mips/dp/aluresult            -No Data-  00000198    00000054    0000004f
/testbench/dut/mips/c/zero                  -No Data-
/testbench/dut/mips/c/md/controls           -No Data-  00030       00420       02100
                                       Now  625 ps            610 ps      620 ps
                                  Cursor 1
```

Completed Instruction Trace

Cycle	Reset	PC	Instr	(FSM) state	SrcA	SrcB	ALUResult	Zero	Control Word
1	1	00	0	0	00	04	04	0	5010
2	0	04	ADDI 2002005	1	04	X	X	0	0030
3	0	04	ADDI 2002005	9	00	05	05	0	0420
4	0	04	ADDI 2002005	10	04	X	X	0	0800
5	0	04	ADDI 2002005	0	04	04	08	0	5010
6	0	08	ADDI 200300c	1	08	X	X	0	0030
7	0	08	ADDI 200300c	9	00	0C	0C	0	0420
8	0	08	ADDI 200300c	10	X	X	X	0	0800
9	0	08	ADDI 200300c	0	08	04	0C	0	5010
10	0	0C	ADDI 2067FFF7	1	0C	X	X	0	0030
11	0	0C	ADDI 2067FFF7	9	0C	F7	03	0	0420
12	0	0C	ADDI 2067FFF7	10	X	X	X	0	0800
13	0	0C	ADDI 2067FFF7	0	0C	04	10	0	5010
14	0	10	OR 00E22025	1	10	X	X	0	0030
15	0	10	OR 00E22025	6	03	05	07	0	0402
16	0	10	OR 00E22025	7	X	X	X	0	0840
17	0	10	OR 00E22025	0	10	04	14	0	5010
18	0	14	AND 00642824	1	14	X	X	0	0030
19	0	14	AND 00642824	6	0C	07	04	0	0402
20	0	14	AND 00642824	7	X	X	X	0	0840
21	0	14	AND 00642824	0	14	04	18	0	5010
22	0	18	ADD 00A42820	1	18	X	X	0	0030
23	0	18	ADD 00A42820	6	04	07	0B	0	0402
24	0	18	ADD 00A42820	7	X	X	X	0	0840

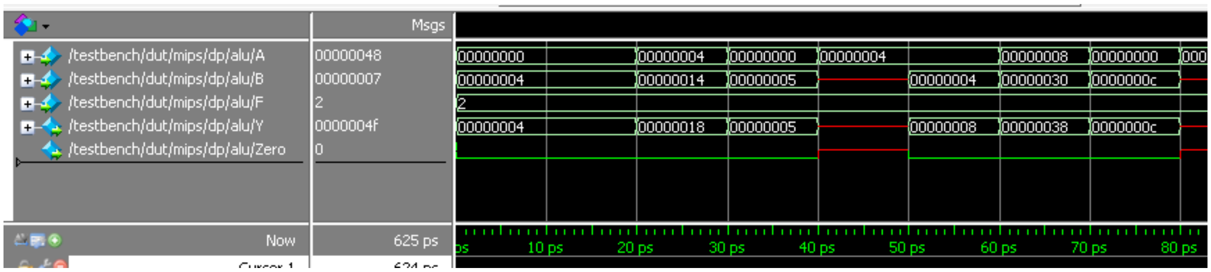
25	0	18	ADD 00A42820	0	18	04	1C	0	5010
26	0	1C	BEQ 10A7000A	1	1C	X	X	0	0030
27	0	1C	BEQ 10A7000A	8	0B	03	0E	0	0605
28	0	1C	BEQ 10A7000A	0	1C	04	20	0	5010
29	0	20	SLT 0064202A	1	20	X	X	0	0030
30	0	20	SLT 0064202A	6	0C	07	00	1	0402
31	0	20	SLT 0064202A	7	X	X	X	0	0840
32	0	20	SLT 0064202A	0	20	04	24	0	5010
33	0	24	BEQ 10800001	1	24	X	X	0	0030
34	0	24	BEQ 10800001	8	00	00	0	1	0605
35	0	28	BEQ 10800001	0	28	04	2C	0	5010
36	0	2C	SLT 00E2202A	1	2C	X	X	0	0030
37	0	2C	SLT 00E2202A	6	03	05	01	0	0402
38	0	2C	SLT 00E2202A	7	X	X	X	0	0840
39	0	2C	SLT 00E2202A	0	2C	04	30	0	5010
40	0	30	ADD 00853820	1	30	X	X	0	0030
41	0	30	ADD 00853820	6	01	0B	0C	0	0402
42	0	30	ADD 00853820	7	X	X	X	0	0840
43	0	30	ADD 00853820	0	30	04	34	0	5010
44	0	34	SUB 00E23822	1	34	X	X	0	0030
45	0	34	SUB 00E23822	6	0C	05	07	0	0402
46	0	34	SUB 00E23822	7	X	X	X	0	0840
47	0	34	SUB 00E23822	0	34	04	38	0	5010
48	0	38	SW AC670044	1	38	X	X	0	0030
49	0	38	SW AC670044	2	0C	44	50	0	0420
50	0	38	SW AC670044	5	X	X	X	0	2100
51	0	38	SW AC670044	0	38	04	3C	0	5010

52	0	3C	LW 8C020050	1	3C	X	X	0	0030
53	0	3C	LW 8C020050	2	00	50	50	0	0420
54	0	3C	LW 8C020050	3	X	X	X	0	0100
55	0	3C	LW 8C020050	4	X	X	X	0	0880
56	0	3C	LW 8C020050	0	3C	04	40	0	5010
57	0	40	J 08000011	1	40	X	X	0	0030
58	0	40	J 08000011	11	X	X	X	0	4008
59	0	44	J 08000011	0	44	04	48	0	5010
60	0	48	SW AC020054	1	48	X	X	0	0030
61	0	48	SW AC020054	2	00	54	54	0	0420
62	0	48	SW AC020054	5	X	X	X	0	2100

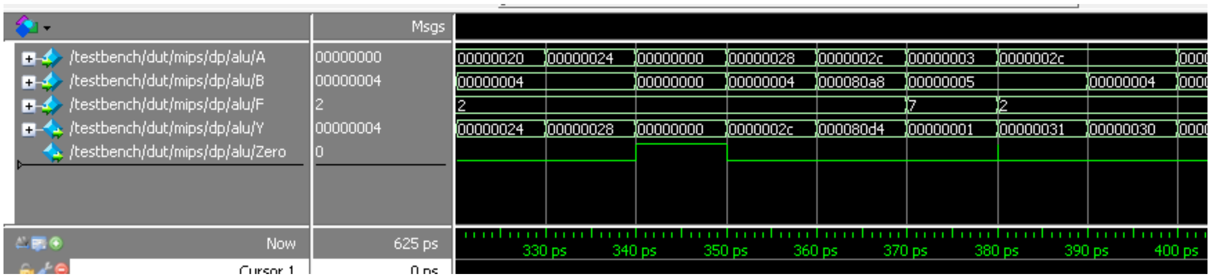
The instruction trace contains the total cycle took for the program, the cycle count of each instruction. The ALU, FSM State and the control word. ALUResult values that are used for decoding is marked as X because it is not needed.

VII. CONCLUSION

The test program took 62 cycles / 620ps to finish and the desired output is achieved. However, some anomalies are found in the ALU.


Msgs
/testbench/dut/mips/dp/alu/A 00000048
/testbench/dut/mips/dp/alu/B 00000007
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 0000004f
/testbench/dut/mips/dp/alu/Zero 0
00000000 00000004 00000000 00000004 00000008 00000000
00000004 00000014 00000005 00000004 00000030 0000000c
2
00000004 00000018 00000005 00000008 00000038 0000000c
Now 625 ps
ps 10 ps 20 ps 30 ps 40 ps 50 ps 60 ps 70 ps 80 ps
Cursor 1

The zero flag outputs x value instead of zero. This is because the scrB contains no data at the start, where the expected instruction trace zero = 0 at this point. This event does not cause any major faults.


Msgs
/testbench/dut/mips/dp/alu/A 00000000
/testbench/dut/mips/dp/alu/B 00000004
/testbench/dut/mips/dp/alu/F 2
/testbench/dut/mips/dp/alu/Y 00000004
/testbench/dut/mips/dp/alu/Zero 0
00000020 00000024 00000000 00000028 0000002c 00000003 0000002c
00000004 00000000 00000004 000080a8 00000005 00000004
2 7 2
00000024 00000028 00000000 0000002c 000080d4 00000001 00000031 00000030
Now 625 ps
330 ps 340 ps 350 ps 360 ps 370 ps 380 ps 390 ps 400 ps
Cursor 1 0 ps

At 380ps, data glitches are found. The possible cause is that the A and F change value at the same time where practically it should not when A = 3, B = 5 and F = SLT (7), and A < B results in Y = 1 and then at a split picosecond the A changes value to 2C where A < B became false and causes Y = 1 and zero = 1 at that moment, and then changes the F selector to ADD (2) resulting to zero = 0. The main reason of this is the propagation delay difference between A and F.

The other components perform correctly as expected and no major issues are found in the processor.

VIII. RECOMMENDATION

For the future students of Computer Architecture, it is highly recommended that the code is optimized to be as short as possible, i.e. the code must be shortened for readability. It is also recommended that the variables used in the code must be uniform and that all members assigned to work on the processor are informed clearly of the variables in order to avoid mistakes in declaring or assigning values to nonexistent variables.

IX. ASSESSMENT

The making of the entire multicycle processor in Verilog took approximately a total of 30 hours, including testing, debugging, and evaluation. Some problems encountered during the making of the program in Verilog include: At times, the ‘begin’ and ‘end’ statements may be misused, which can lead to errors similar to in languages like C, when the opening and closing braces (‘{‘ and ‘}’) are missing or misplaced. Naming conventions are also a problem; when another user creates and adds some lines of code to the program, s/he may confuse or mistake the names of the variables, creating bugs in the code. Regs and wires are also often confused, leading to testbench errors. Despite overlooking some of the errors in the program, the project was accomplished with a significant amount of knowledge acquired from the subject.

Overall, the project successfully increased students' knowledge of processors, processor design, and Verilog implementation of the design. The students also further learned the importance of planning, teamwork, and effective documentation of results.

References

Stallings, W. (2006). Computer Organization And Architecture Designing For Performance (8th ed.).

Harris, D. M. & Harris, S. L. (2013). Digital Design and Computer Architecture (2nd ed.).

Parhami, B. (2005). Computer Architecture: From Microprocessors to Supercomputers, Oxford Univ. Press, New York, 2005.

Hussein, Z. (2017). Pipelined Processor, Lafayette College, Easton, Pennsylvania. Retrieved from: https://github.com/abzain/New_ECE212/tree/master/Lab10

Rangit, L. (2014). Verilog Multicycle MIPS processor, California State University, Fullerton EE557. Retrieved from: https://github.com/lucasrangit/Multicycle_MIPS

Verilog Code for 16-bit RISC Processor. Retrieved from: https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html

Verilog code for 16-bit single cycle MIPS processor. Retrieved from: https://www.fpga4student.com/2017/01/verilog-code-for-single-cycle-MIPS-processor.html

VHDL code for MIPS Processor. Retrieved from:

https://www.fpga4student.com/2017/09/vhdl-code-for-mips-processor.html

Verilog code for Arithmetic Logic Unit (ALU). Retrieved from:

https://www.fpga4student.com/2017/06/Verilog-code-for-ALU.html

Parameterized Modules. Retrieved from: http://www.asic-world.com/verilog/para_modules.html

Art of Writing TestBenches.Retrieved from:

http://www.asic-world.com/verilog/art_testbench_writing.html

Ellard, D. J. (September 1994). MIPS Assembly Language Programming CS50 Discussion and Project Book. Retrieved From: https://sites.cs.ucsb.edu/~franklin/64/lectures/mipsassemblytutorial.pdf