ECE 485 Project #2

Jay Mundrawala

November 23, 2009

Abstract

This project describes the design and implementation of a multicycle MIPS datapath. Along with it are testbenches that issue the appropriate signals for a few signals.

1 Introduction

The goal of this project was to implement a MIPS datapath capable of executing a small set of instructions. For this project, we were to implement the following instructions:

- LW
- SW
- ADD
- SUB
- AND
- OR
- SLT
- BEQ
- J
- BNE
- ADDI
- SLL
- LUI
- JAL

The source code for this datapath is available in the appendix and in a git repository on github. This is available at `http://github.com/whois/ECE485_P2`.

2 Design

The design for a multicycle MIPS datapath is quite simple. It is shown in Figure 1. The data path has the following signals coming from the control unit, which describe how the datapath should operate: PCWrite, IorD, MemRead, MemWrite, IRWrite, RegDst, MemToReg, RegWrite, ALUSrcA, ALUSrcB, ALUOp, and PCSource. Tables 1 through 6 enumerate the possible values for these signals and Listings 5 contains their definitions.

Signal name	Effect when deasserted	Effect when asserted
PCWrite	None	PC is written
PCWriteCondEq	None	PC is written of zero is asserted
PCWriteCondNEq	None	PC is written of zero is deasserted
IorD	Instruction is Fetched	Data is fetched
MemRead	None	Memory at given address is read
MemWrite	None	Data is written to given address
IRWrite	None	Instruction register is written
RegWrite	None	Data is written to register
ALUSrcA	Port A of ALU gets PC	Port A of ALU gets register A's output

Table 1: Actions of 1-bit control signals

ALUSrcB	
Signal value	Effect
ASB_REGB	Port B of ALU gets register B's output
ASB_FOUR	Port B of ALU gets 4
ASB_SEXT	Port B of ALU gets instruction[15-0] sign extended to 32 bits
ASB_SEXTS	Same as previous except value is shifted left by 2

Table 2: Actions of ALUSrcB signal

ALUOp	
Signal value	Effect
AOP_AND	ALU will 'and' the two operands
AOP_OR	ALU will 'or' the two operands
AOP_ADD	ALU will add the two operands
AOP_SUB	ALU will subtract A - B
AOP_SLT	ALU will output '1' if $A < B$, '0' otherwise
AOP_NOR	ALU will NOR the two operands
AOP_SLL	ALU will shift B left by instruction[10-6]

Table 3: Actions of ALUOp signal

With these signals defined, we can begin by defining each functional block in Figure 1. First, we have the PC. This is the program counter and stores the address of the next instruction to be executed. The memory block is our RAM. For this project, the test benches only output one value when the memory is read, and that is the instruction we are testing. This allows us to test the datapath without creating a memory module. Nothing is ever written to the memory; when testing LW we analyze the value on the

PCSource	
Signal value	Effect
PS_PCINC	Value to PC is $PC + 4$
PS_ALUOUT	Value to PC is that of the register ALU OUT
PS_JMP	Value to PC is $PC[31 - 28] + Instr[25 - 0] << 2$

Table 4: Actions of PCSource signal

RegDst	
Signal value	Effect
RD_RT	Write register is set to instr[25-21]
RD_RD	Write register is set to instr[20-16]
RD_RA	Write register is set to $R31$

Table 5: Actions of RegDst signal

write data line when it is time to write to the memory. The instruction register holds the values of the instruction being executed. The memory data register(MDR), stores the value read from memory. This will be written every clock cycle since the value only need be stored for one clock cycle. Thus, it does not need its own control signal and its enable can be attached to the clock. The register file, denoted by Registers in Figure 1, stores the 32 registers. Register A and B store the output of the register file for one clock cycle. The ALU does all the arithmetic computations. Its functionality is fully described by Table 3. The ALU OUT register holds the output of the ALU for one clock cycle. PCWrite Generate generates the enable signal to write to the PC. In the case of this design, it is described by the following equation: $G = PCWrite + PCWriteCondEq * zero + PCWriteCondNEq * \overline{zero}$.

2.1 Instruction Fetch, Instruction Decode, and Branch Target Computation

There are two steps that each instruction execution has in common. The first is the IF stage. Each instruction needs to be fetched from memory. To do this, we need to read the instruction from memory and store it into the instruction register. This is also the step where we must update the PC to $PC + 4$. So, with that in mind, we can define the signals needed to complete this step. First, IorD needs to be deasserted. This will select the PC as the address for memory. MemRead needs to be set, so that the instruction is fetched from memory. IRWrite needs to be asserted so that the instruction is stored after it is read. This is enough to actually fetch the instruction, but the PC still needs to be updated. Thus, we require ALUSrcA to be deasserted to select PC for port A of the ALU, and ALUSrcB to be set to ASB_FOUR to select 4 for the second port($PC + 4$). PCSource needs to be set to PS_PCINC inidcating that $PC + 4$ should be written to the PC. Finally, PCWrite needs to be asserted so that the value is written to the PC. One thing that should be noted is that all writes happen only when the clock is high, except for the IRWrite. This is a bug, however it does not affect the system in any way.

The next step is the instruction decode stage. In this stage, we do two things. First, we read the registers from the register file. This is done automatically by the instruction register. Since the ALU is not being used in this stage, its a good time to calculate the

MemToReg	
Signal value	Effect
MTR_ALUOUT	Register write data gets the output of the ALU OUT register
MTR_MDR	Register write data gets the output of the MDR register
MTR_PC	Register write data gets the PC

Table 6: Actions of MemToReg signal

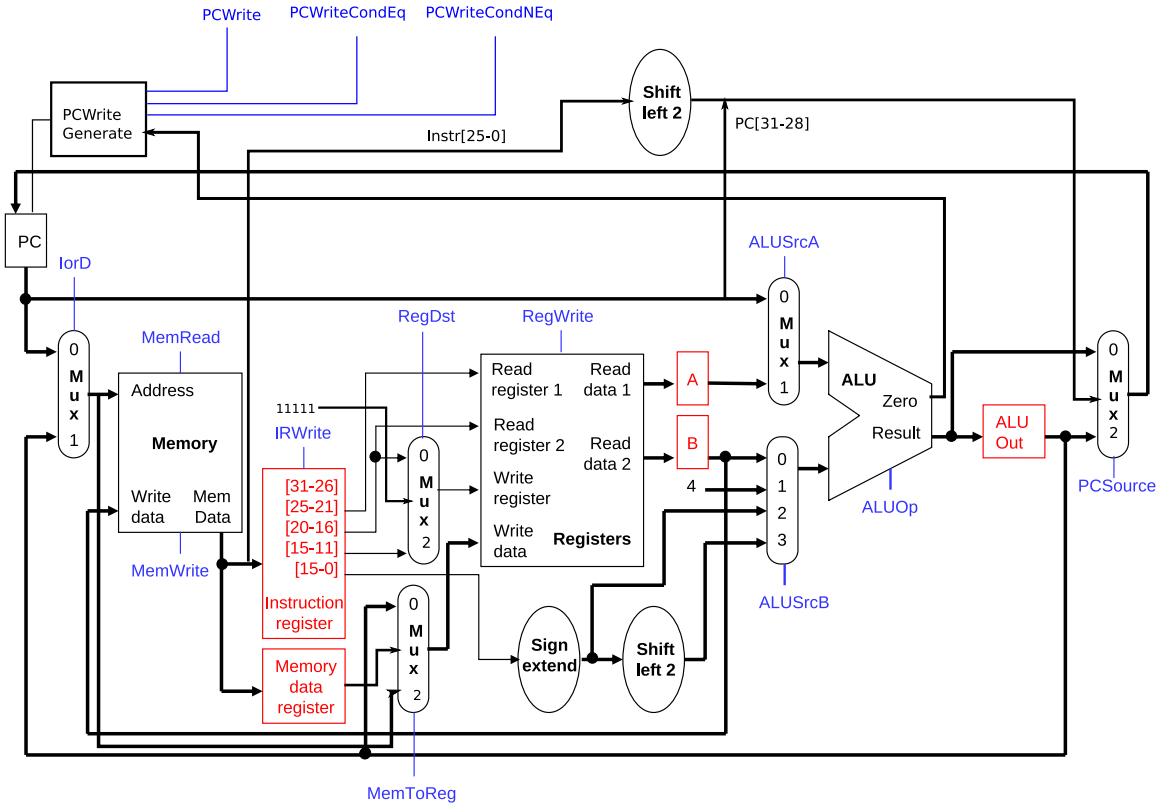

Figure 1: MIPS multicycle datapath

branch target address, regardless of if the instruction is a branch or not. This way, the target is ready incase it is a branch instruction. To do this, ALUSrcA is deasserted, indicating port A of the ALU gets the value of the PC. ALUSrcB is set to indicate the sign extended and shifted value of instruction[15-0]. These two will be added and stored in ALU OUT.

Figure 2 shows these two steps. One thing to note is that the first clock cycle is just a reset; it assigns values to all the signals so they are known. Thus, for this figure, and all future figures, the cycle begins in the second clock cycle. The important thing to note in this figure is that the PC is being assigned $PC + 4$ after executing the IF stage and IR has been assigned an instruction in that stage. In the next stage, register A and register B are assigned values that were read from the register file. The final clock cycle is bogus as the state machine was not updated.

2.2 LW/SW Instructions

LW and SW are the only two instructions allowed to access memory. They both have one step in common, and that is the memory address computation stage. In this step,

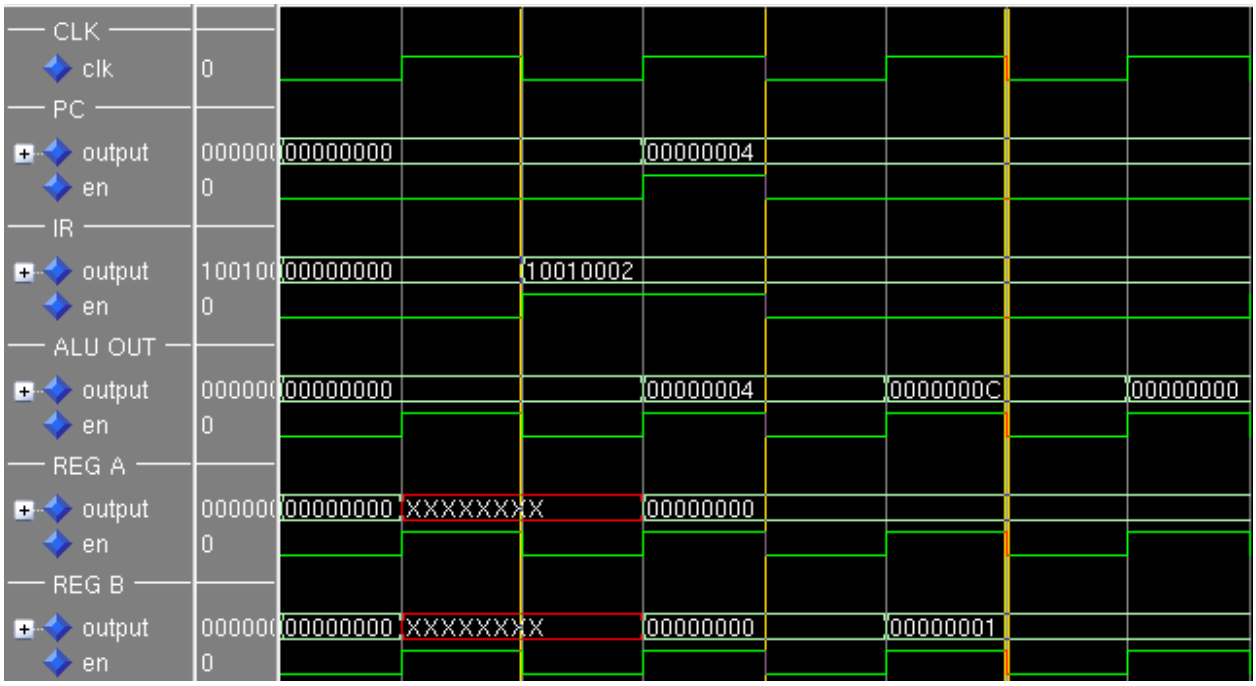

Figure 2: IF and ID stages. Both the first and last clock cycle have no meaning. The middle two are the one of interest.

ALUSrcA will be set to use register A. ALUSrcB will be set to use the bottom 16 bits of the instruction sign extended to 32 bits. These will be added by the ALU and used for the memory access.

Next is the memory access. The previously computed address is used here. Thus, in both the cases of LW and SW IorD is set to use the ALU OUT register. For LW, MemRead is asserted, and for SW, MemWrite is asserted.

At this point, SW is complete, but LW still has to write back. Thus, RegWrite is asserted, MemToReg is set to use the value stored in MDR, and RegDst is set to use RT.

Figure 3 shows the operation of LW. The first clock cycle is a reset. The second shows the IF stage. The third shows the ID stage. The fourth is where the memory address computation is occuring. Here, $0x00FF$ is being added to $R0$ as specified by the instruction. The ALU outputs FF as expected. Next, is the memory access... nothing happens here since the value of memory is kept constant for simplicity. Next, in the final clock cycle, the write back occurs, and $R1$ gets the value of the memory.

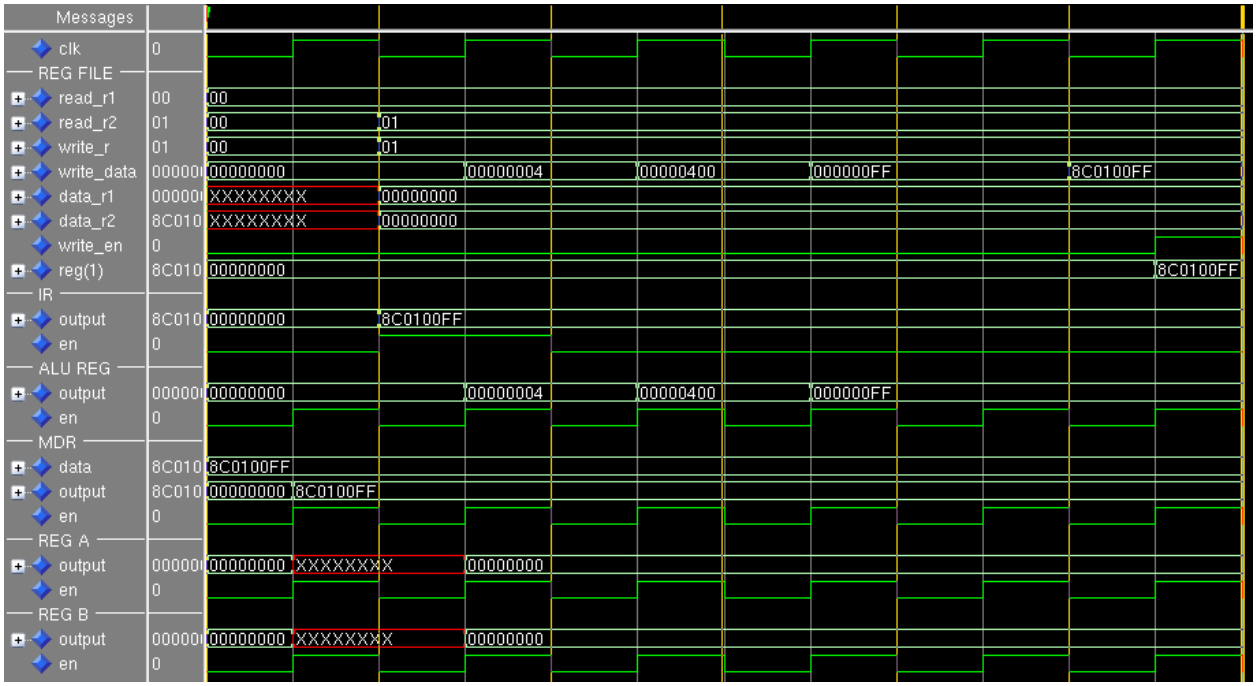

Figure 3: Shows the execution of a LW instruction

Similarly, Figure 4 shows the operation of SW. The first 4 cycles behave the same way. The final cycle is different in that it asserts a MemWrite signal and writes the data to memory.

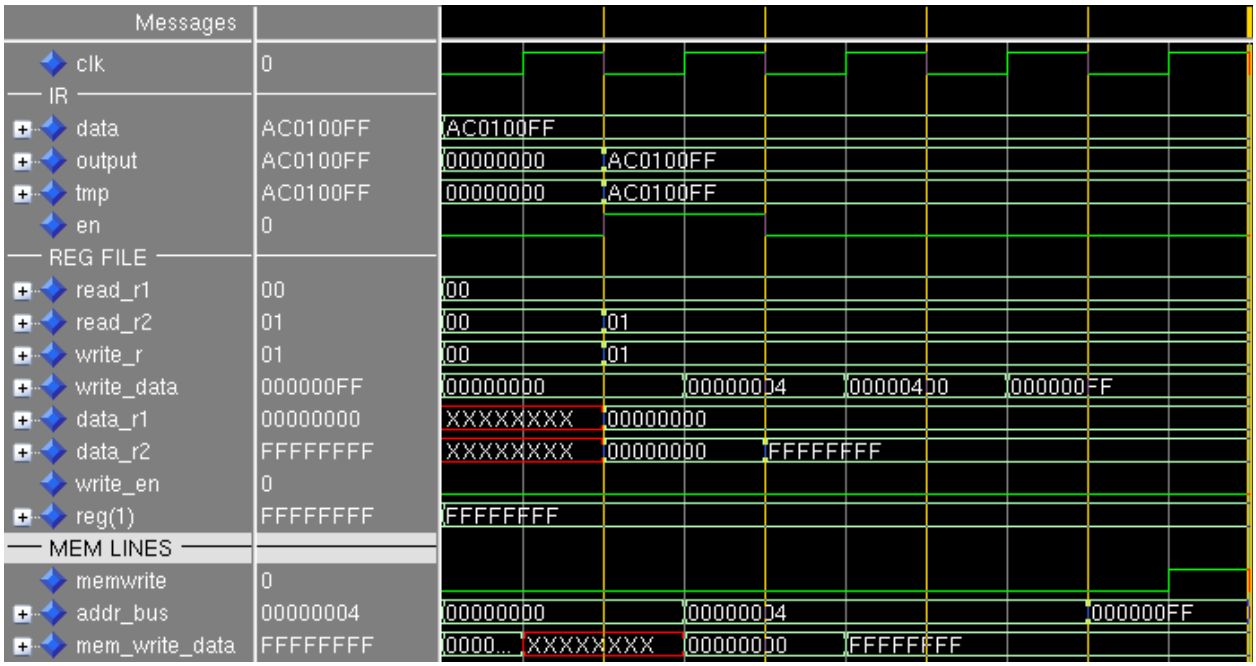

Figure 4: Shows the execution of a SW instruction

2.3 R Type Instructions

This section shows the correct functionality of the R Type instructions. Only add and sll are described in detail. The waveforms for the remaining are at the end of the section.

No further description is needed as only changing the ALU operation for the R Type instructions is done.

In general, R Type instructions consist of first performing the ALU operation specified, and writing that value back to the registers. For example, Figure 5 shows the waveform for an add instruction. First there is the reset cycle, then IF, and ID. Next is the execution stage. Here, the ALU takes the values read from the register file and performs and add on them. And the end of this cycle, write_data has the value of $R0 + R1$. In the next cycle, the write back cycle, $R2 <= R0 + R1$ as expected.

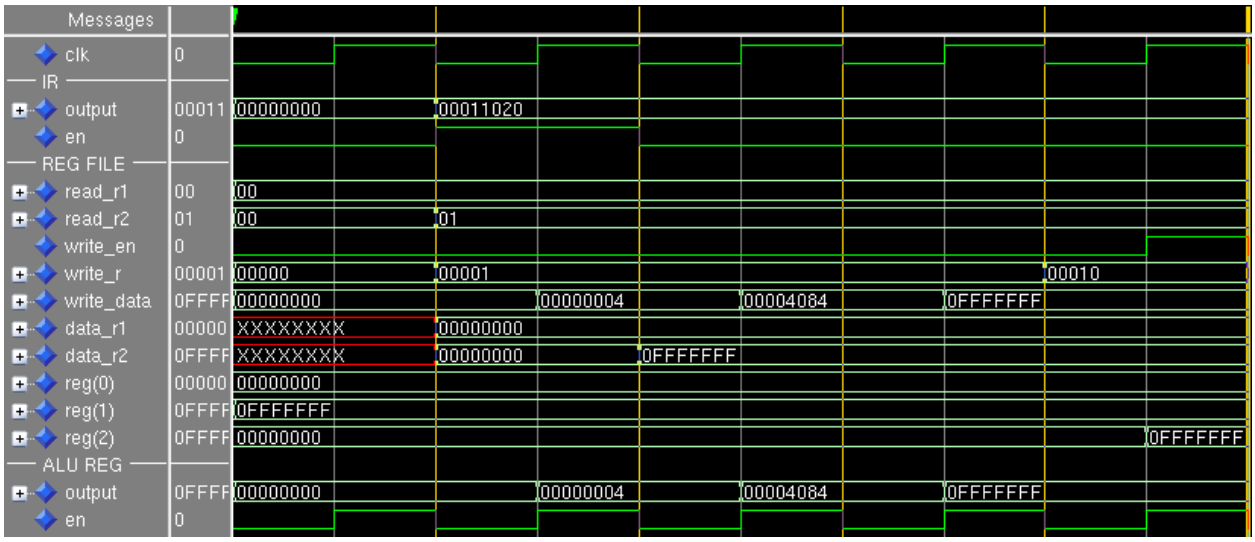

Figure 5: Shows the execution of a add instruction

The SLL is similar, except instead of adding 2 operands, it takes the B operand to the ALU and shifts it by an amount specified in the instruction. This is shown in Figure **??**. The first clock cycle is the reset, followed by the IF, followed by ID, followed by the execution. In the execution, ALUSrcA is irrelevant. Operand B(R1) is '1', which will be left shifted by the amount of 2. Thus, the expected value is 4, and that is the value that is written back in the final clock cycle.

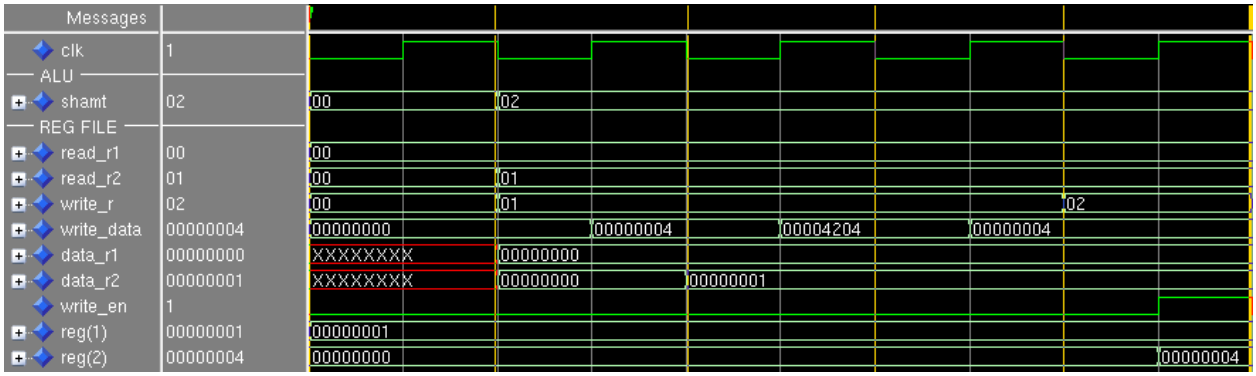

Figure 6: Shows the execution of a SLL instruction

The following are the figures for SUB, AND, OR, and SLT. Whats important to look at in these instructions are the operands and the final write back values. With that, it can be seen that each functions correctly.

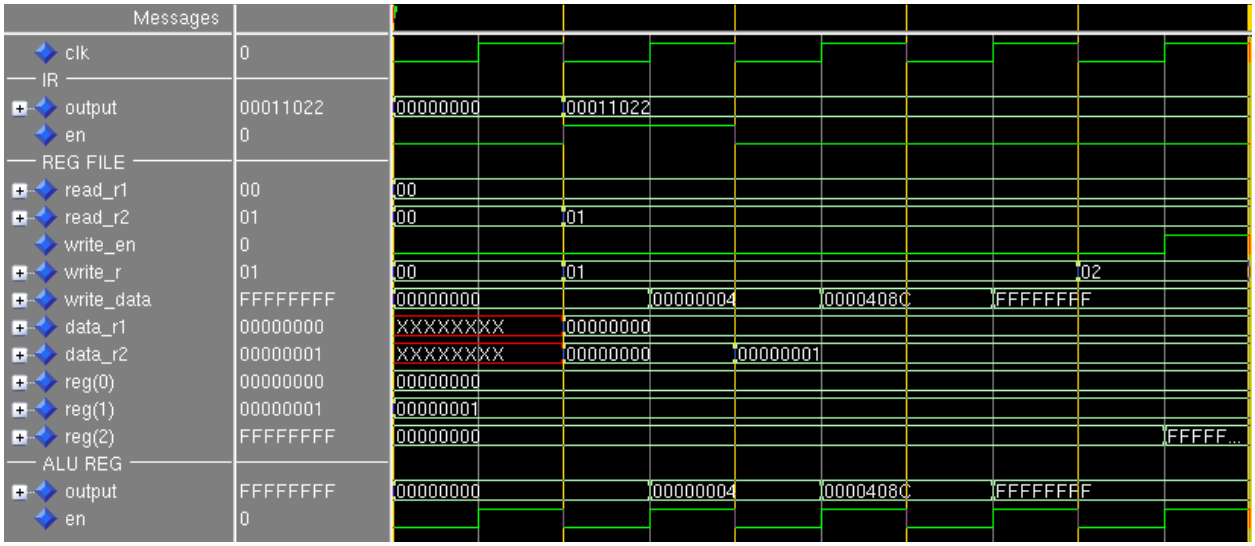

Figure 7: Shows the execution of a SUB instruction

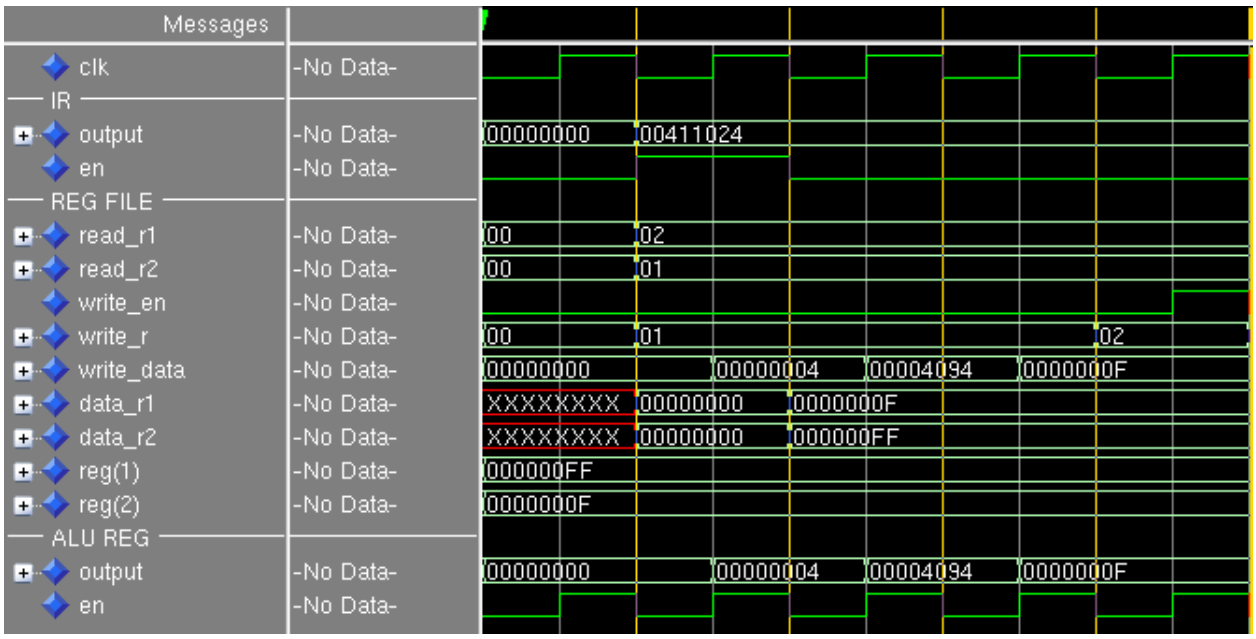

Figure 8: Shows the execution of a AND instruction

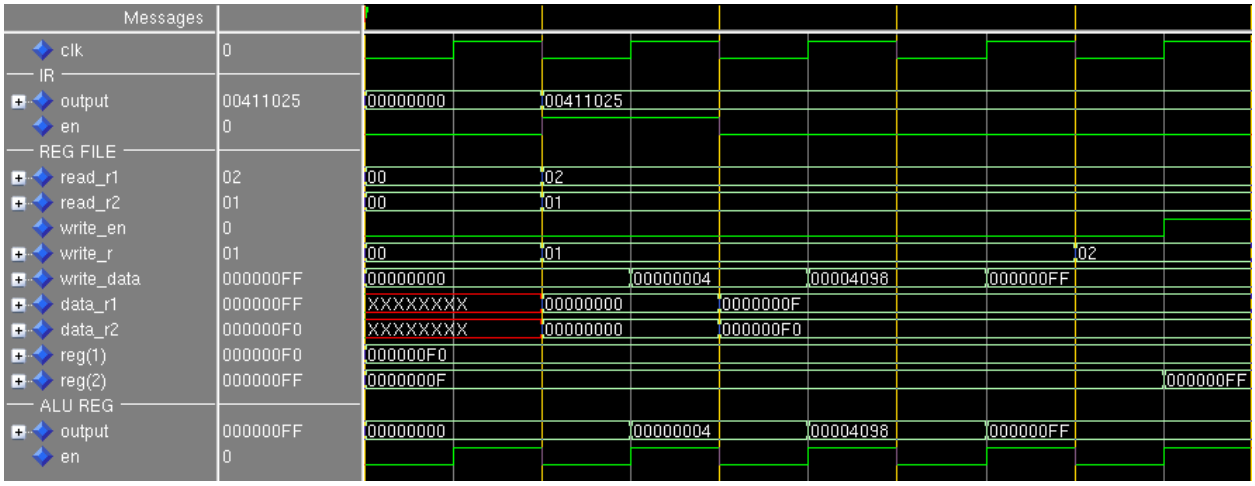

Figure 9: Shows the execution of a OR instruction

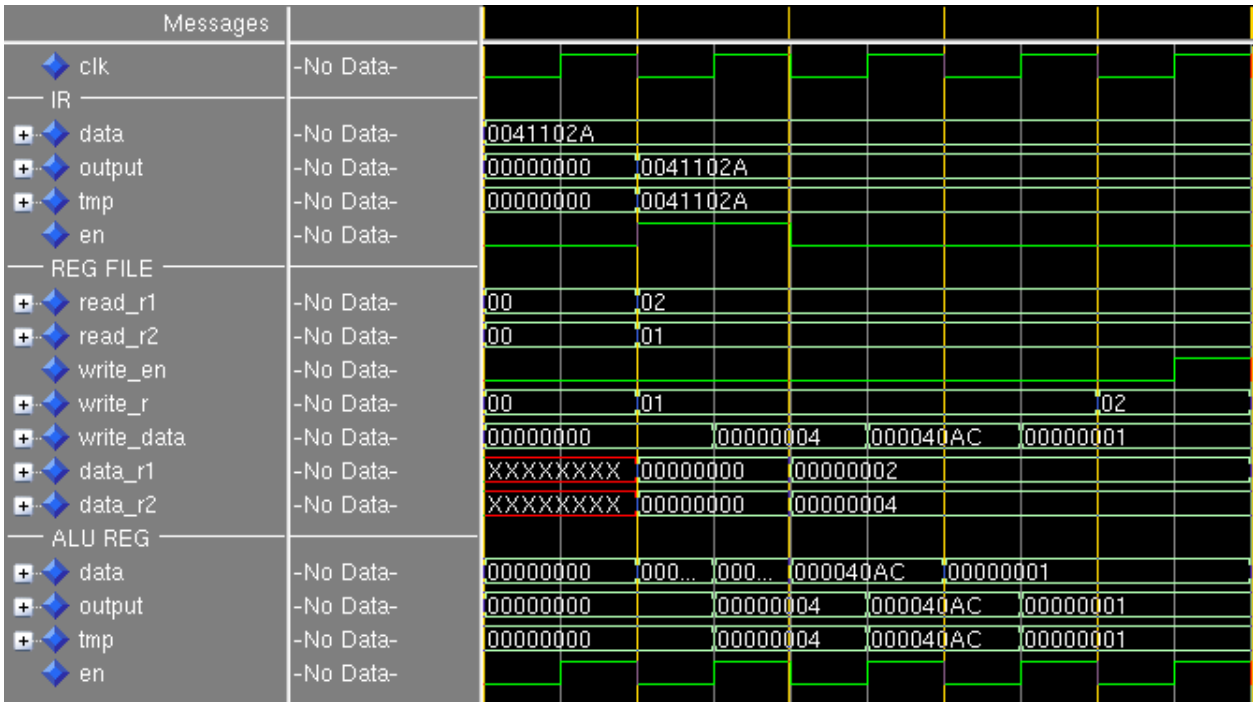

Figure 10: Shows the execution of a SLT instruction

2.4 Immediate Instruction

Only one immediate instruction required testing, and the was ADDI. Its very similar to ADD, except that ALUSrcB is set to use the sign extended lower 16 bits of the instruction. Its waveform is shown in Figure 11. The first cycle is reset, the second is instruction fetch, third is instruction decode, fourth is execution. The lower 16 bits of this instruction are 8, which is being added to $R0$. The final clock cycle shows that 8 is written back as expected.

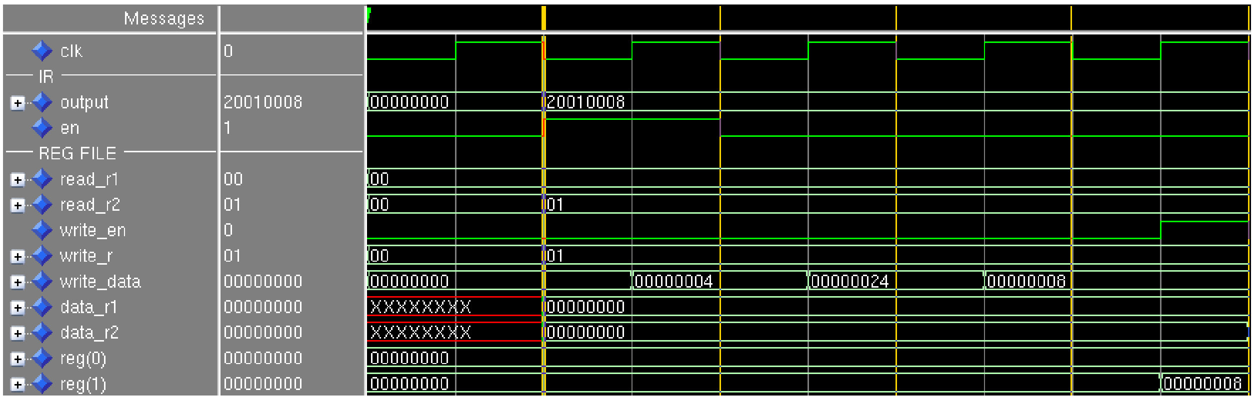

Figure 11: Shows the execution of a ADDI instruction

2.5 Branch and Jump

This section shows the correct functionality of BNE and JAL. BEQ and JMP are shown at the end of this section for completeness, however they are not annotated as the instructions are very similar to BNE and JAL.

The correct functionality for BNE is shown in Figure 12. For branch instructions, the target address is available after the ID stage. Thus, as soon as the appropriate signals have progagated from the ALU, we can branch. For BNE, the control unit will set ALUSrcA to register A. ALUSrcB will be set to allow register B. These two will be compared in the ALU during the execution stage. Along with those signals, BNE needs to assert the PCWriteCondNEq signal to indicate a branch when not equal. The branch occurs in the execution stage, as that is when PC is written with its new value. In Figure 12, the lower 16 bits are 2. Sign extended and shifted left by two makes 8. Thus, $PC + 4 + 8 = C$, as the original PC started off at 0. This is the value that is written to the PC as can be seen.

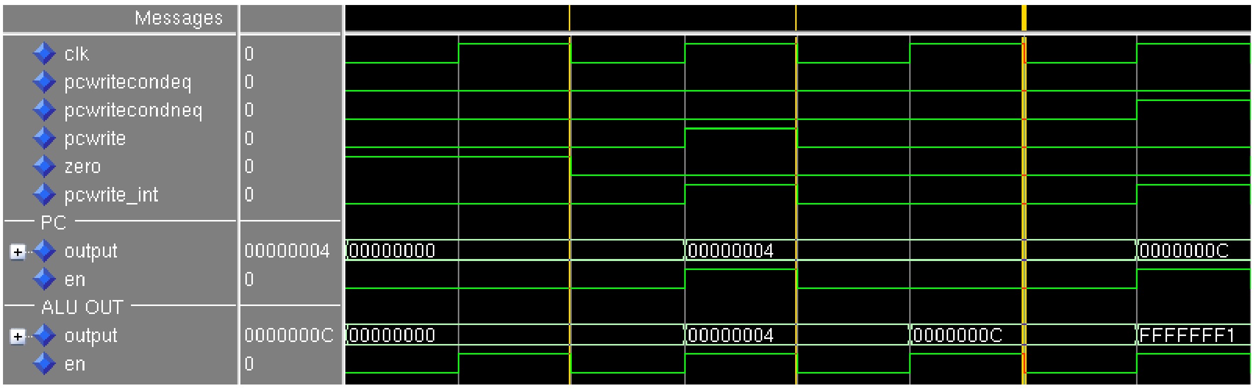

Figure 12: Shows the execution of a BNE instruction

Jumping is similar to a branch, however it uses instruction[26-0] shifted left by 2 for the lower 28 bits. The higher bits are determined by PC[31-28]. JAL, shown in Figure 13, does more than just jump. It also writes the value of the PC to $R31$. The instruction if Figure 13 stores the value of PC as expected into $R31$ in the final clock cycle. The lower 26 bits of the instruction are 8, thus $0x20$ is the expected jump value, the same value that PC gets in the final clock cycle.

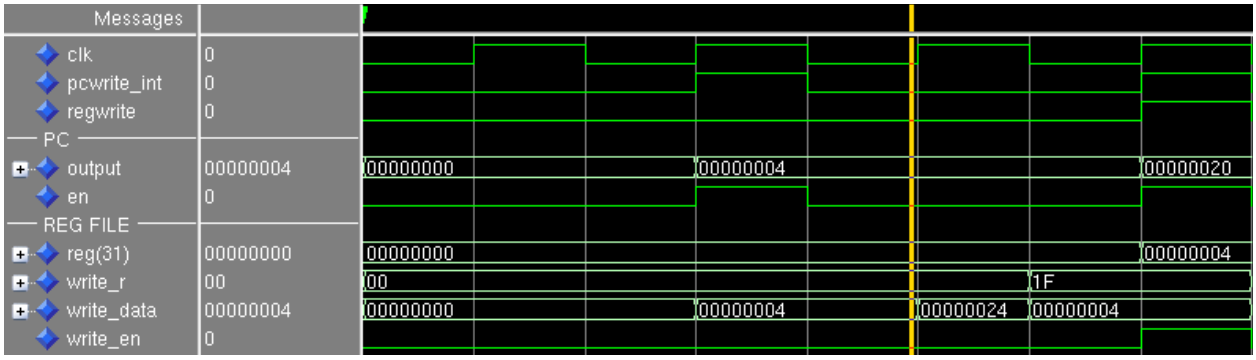

Figure 13: Shows the execution of a JAL instruction

The following figures(13 and 15 show similar instructions J and BEQ.

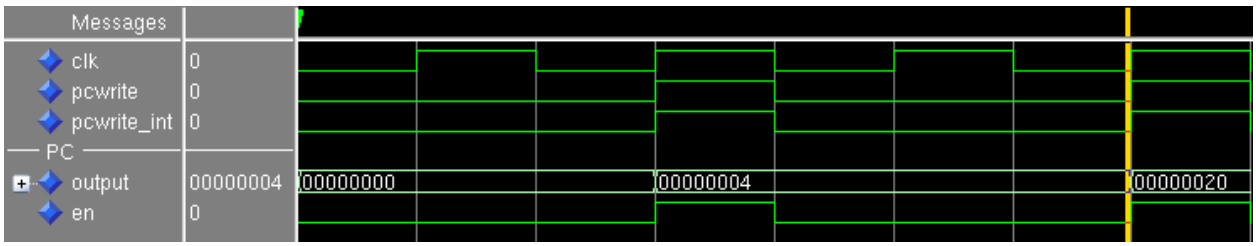

Figure 14: Shows the execution of a J instruction

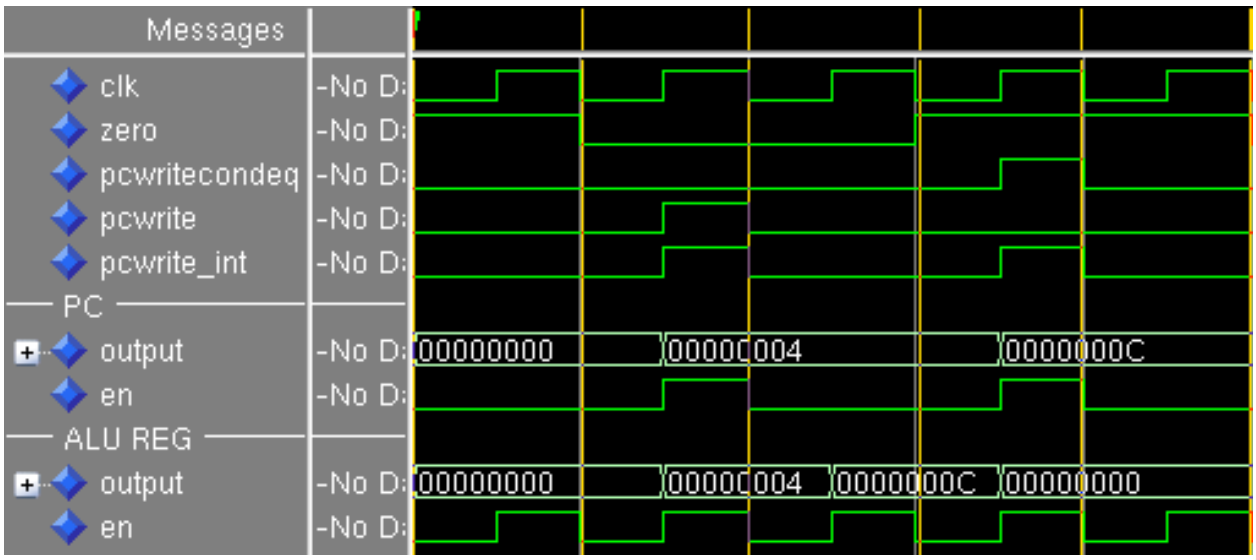

Figure 15: Shows the execution of a BEQ instruction

2.6 LUI

Since the ALU already has the capability to do shift operations on ALU port B operands, nothing needs to be added to the datapath. This instruction can be executed by 3 instructions already described. The instructions needed are SLL, which is done by 16 bits. Next, we can use an AND the lower upper 16 bits of the register we want to LUI into with 1's. This clears out the top bits. This leaves one final OR with the result of SLL and the register we want to LUI into.

2.7 Exception Handling

Exception handling requires two additional registers and addition control signals. The registers needed are the cause register and the EPC. This datapath must allow for two different types of exceptions. First, the arithmetic overflow exception. This can only occur during the execution stage, and may not be caused by the control unit. The second is undefined instruction. Here, there are two possibilities. First, there can be an unkwown opcode. Second, there can be an unknown function for an R-Type. The design here is to let the ALU assert arithmetic overflow exceptions, and the control unit will assert the unknown operation exception. When these are asserted, there will be a jump to some hard-wired memory address, in this case $0x4$ for simplicity.

3 Unfinished

1. ALU does not generate overflow
2. There is no testbench for exception handling

A Code

```
1  ----------------------------------------------------------------------------
2  -- Author(s)    : Jay Mundrawala <mundra@ir.iit.edu>
3  --
4  -- File           : mips_lib.vhdl
5  -- Creation Date : 06/11/2009
6  -- Description:
7  --
8  ----------------------------------------------------------------------------
9
10 library IEEE;
11 use IEEE.STD_LOGIC_1164.ALL;
12 use IEEE.numeric_std.all;
13
14 package mips_lib is
15     constant DATA_WIDTH : integer                := 32;
16     constant ADDR_WIDTH : integer                := 32;
17
18     constant rOpcode    : bit_vector(5 downto 0) :="000000";
19     constant jOpcode    : bit_vector(5 downto 0) :="000010";
20     constant jalOpcode  : bit_vector(5 downto 0) :="000011";
21     constant addiOpcode : bit_vector(5 downto 0) :="001000";
22     constant andiOpcode : bit_vector(5 downto 0) :="001100";
23     constant beqOpcode  : bit_vector(5 downto 0) :="000100";
24     constant bneOpcode  : bit_vector(5 downto 0) :="000101";
25     constant lwOpcode   : bit_vector(5 downto 0) :="100011";
26     constant swOpcode   : bit_vector(5 downto 0) :="101011";
27
28     constant addFunc    : bit_vector(5 downto 0) :="100000";
29     constant subFunc    : bit_vector(5 downto 0) :="100010";
30     constant andFunc    : bit_vector(5 downto 0) :="100100";
31     constant orFunc     : bit_vector(5 downto 0) :="100101";
32     constant sltFunc    : bit_vector(5 downto 0) :="101010";
33     constant sllFunc    : bit_vector(5 downto 0) :="000000";
```

```
34
35
36
37      type t_aluSrcA     is (ASA_PC, ASA_REG_A);
38      type t_aluSrcB     is (ASB_REGB, ASB_FOUR, ASB_SEXT, ASB_SEXTS);
39      type t_aluOp       is (AOP_AND, AOP_OR, AOP_ADD, AOP_SUB, AOP_SLT,
            AOP_NOR, AOP_SLL);
40      type t_pcSrc       is (PS_PCINC, PS_ALUOUT, PS_JMP, PS_FOUR);
41      type t_regDst      is (RD_RT, RD_RD, RD_RA);
42      type t_iord        is (IOD_PC, IOD_ALUOUT);
43      type t_memToReg    is (MTR_ALUOUT, MTR_MDR, MTR_PC);
44      type t_comp        is (eq, ne, gt, lt, lte, gte);
45      type t_microinstr is (
46           s_if,     --Instruction Fetch
47           s_id      --Instruction Decode
48      );
49
50
51  end package;
```

Listing 1: mips_lib.vhdl

```
1   ----------------------------------------------------------------------------
2   -- Author(s)    : Jay Mundrawala <mundra@ir.iit.edu>
3   --
4   -- File           : reg.vhdl
5   -- Creation Date : 06/11/2009
6   -- Description:
7   --
8   ----------------------------------------------------------------------------
9
10  library IEEE;
11  use IEEE.STD_LOGIC_1164.ALL;
12  use IEEE.numeric_std.all;
13
14  ----------------------------------------------------------------------------
15  Entity reg is
16  ----------------------------------------------------------------------------
17  generic
18  (
19      SIZE  : natural := 32;
20      DELAY : time := 0 ns
21  );
22
23
24  port
25  (
26      en     : in std_logic;
27      data   : in std_logic_vector((SIZE-1) downto 0);
28      output : out std_logic_vector((SIZE-1) downto 0)
29  );
30  end entity;
31
32
33  ----------------------------------------------------------------------------
34  Architecture reg_1 of reg is
35  ----------------------------------------------------------------------------
36      signal tmp : std_logic_vector((SIZE-1) downto 0) := (others => '0');
```

```
37 begin
38     process(en)
39     begin
40         if(en='1') then
41             tmp <= data;
42         end if;
43     end process;
44     output <= tmp;
45 end architecture reg_1;
```

Listing 2: Register

```
1  ----------------------------------------------------------------------------
2  -- Author(s)    : Jay Mundrawala <mundra@ir.iit.edu>
3  --
4  -- File         : reg_file.vhdl
5  -- Creation Date : 06/11/2009
6  -- Description:
7  --
8  ----------------------------------------------------------------------------
9
10 library IEEE;
11 use IEEE.STD_LOGIC_1164.ALL;
12 use IEEE.numeric_std.all;
13 use ieee.std_logic_arith.all;
14 use ieee.std_logic_unsigned.all;
15 ----------------------------------------------------------------------------
16 Entity reg_file is
17 ----------------------------------------------------------------------------
18 generic
19 (
20     SIZE : natural :=32;
21     DELAY : time := 0 ns
22 );
23 port
24 (
25     clk        : in std_logic;
26     write_en   : in std_logic;
27     read_r1    : in std_logic_vector(4 downto 0);
28     read_r2    : in std_logic_vector(4 downto 0);
29     write_r    : in std_logic_vector(4 downto 0);
30     write_data : in std_logic_vector((SIZE-1) downto 0);
31     data_r1    : out std_logic_vector((SIZE-1) downto 0);
32     data_r2    : out std_logic_vector((SIZE-1) downto 0)
33 );
34 end entity;
35
36
37 ----------------------------------------------------------------------------
38 Architecture reg_file_1 of reg_file is
39 ----------------------------------------------------------------------------
40     type t_reg is array (0 to 31) of std_logic_vector(31 downto 0);
41     signal reg : t_reg := ((others => (others=>'0')));
42
43 begin
44     process(clk)
45     begin
46         if(clk'event and clk='0') then
```

```
47              data_r1 <= reg(CONV_INTEGER(read_r1)) after DELAY/2;
48              data_r2 <= reg(CONV_INTEGER(read_r2)) after DELAY/2;
49          elsif(clk'event and clk='1') then
50              if(write_en ='1') then
51                  reg(CONV_INTEGER(write_r)) <= write_data after DELAY;
52              end if;
53          end if;
54          reg(0) <= (others => '0');
55      end process;
56  end architecture reg_file_1;
```

Listing 3: Register File

```
1   ----------------------------------------------------------------------------
2   -- Author(s)    : Jay Mundrawala <mundra@ir.iit.edu>
3   --
4   -- File         : alu.vhdl
5   -- Creation Date : 21/11/2009
6   -- Description:
7   --
8   ----------------------------------------------------------------------------
9
10  library IEEE;
11  use work.mips_lib.all;
12  use IEEE.STD_LOGIC_1164.ALL;
13  use IEEE.numeric_std.all;
14  ----------------------------------------------------------------------------
15  Entity alu is
16  ----------------------------------------------------------------------------
17  port
18  (
19      op_A      : in std_logic_vector(31 downto 0);
20      op_B      : in std_logic_vector(31 downto 0);
21      shamt     : in std_logic_vector(4 downto 0);
22      alu_ctrl  : in t_aluOp;
23      f         : out std_logic_vector(31 downto 0);
24      zero      : out std_logic
25      overflow  : out std_logic
26  );
27  end entity;
28
29
30
31  ----------------------------------------------------------------------------
32  Architecture alu_1 of alu is
33  ----------------------------------------------------------------------------
34  -- from http://www.csee.umbc.edu/~squire/download/bshift.vhdl
35    function to_integer(sig : std_logic_vector) return integer is
36      variable num : integer := 0;  -- descending sig as integer
37    begin
38      for i in sig'range loop
39        if sig(i)='1' then
40          num := num*2+1;
41        else
42          num := num*2;
43        end if;
44      end loop;  -- i
45      return num;
```

```
46    end function to_integer;
47      CONSTANT DELAY : time := 0 ns;
48      signal value    : std_logic_vector(31 downto 0);
49  begin
50      process(alu_ctrl, op_A, op_B)
51      begin
52          case alu_ctrl is
53              when AOP_AND =>
54                  value <= op_A and op_B after DELAY;
55              when AOP_OR =>
56                  value <= op_A or op_B after DELAY;
57              when AOP_ADD =>
58                  value <= std_logic_vector(signed(op_A) + signed(op_B))
                        after DELAY;
59              when AOP_SUB =>
60                  value <= std_logic_vector(signed(op_A) - signed(op_B))
                        after DELAY;
61              when AOP_SLT =>
62                  if(signed(op_A) < signed(op_B)) then
63                      value <= (others => '0');
64                      value(0) <=  '1';
65                  else
66                      value <= (others => '0');
67                  end if;
68              when AOP_NOR =>
69                  value <= NOT (op_A or op_B) after DELAY;
70              when AOP_SLL =>
71                  value <= to_stdlogicvector(to_bitvector(op_B) sll
                        to_integer(shamt));
72              when others =>
73                  value <= (others => '1');
74          end case;
75         end process;
76         f <= value;
77         zero <= '1' when value = x"00000000" else
78                 '0';
79  end architecture alu_1;
```

Listing 4: ALU

```
1  ----------------------------------------------------------------------------
2  -- Author(s)    : Jay Mundrawala <mundra@ir.iit.edu>
3  --
4  -- File           : datapath.vhdl
5  -- Creation Date : 06/11/2009
6  -- Description:
7  --
8  ----------------------------------------------------------------------------
9
10 library IEEE;
11 use IEEE.STD_LOGIC_1164.ALL;
12 use IEEE.numeric_std.all;
13 use work.mips_lib.all;
14
15 ----------------------------------------------------------------------------
16 Entity datapath is
17 ----------------------------------------------------------------------------
18     port(
```

```
19              clk             : in std_logic;
20      -- Control Unit
21              PCWriteCondEq   : in std_logic;
22              PCWriteCondNEq  : in std_logic;
23              PCWrite         : in std_logic;
24              IorD            : in t_iord;
25              MemRead         : in std_logic;
26              MemWrite        : in std_logic;
27              MemToReg        : in t_memToReg;
28              IRWrite         : in std_logic;
29              RegWrite        : in std_logic;
30              RegDst          : in t_regDst;
31              ALUSrcA         : in t_aluSrcA;
32              ALUSrcB         : in t_aluSrcB;
33              PCSource        : in t_pcSrc;
34              ALUOp           : in t_aluOp;
35              UndefInstrEx    : in std_logic;
36              OverflowEx      : out std_logic;
37
38      --Memory
39              mem_data_out    : in  std_logic_vector((DATA_WIDTH-1) downto 0);
40              mem_read        : out std_logic;
41              addr_bus        : out std_logic_vector((DATA_WIDTH-1) downto 0);
42              mem_write_data  : out std_logic_vector((DATA_WIDTH-1) downto 0)
43          );
44
45  end entity;
46
47
48
49  ------------------------------------------------------------------------
50  Architecture datapath_1 of datapath is
51  ------------------------------------------------------------------------
52      component reg
53          generic (
54                      SIZE  : natural := 32;
55                      DELAY : time := 0 ns
56                  );
57          port (
58                   en     : in std_logic;
59                   data   : in std_logic_vector((SIZE-1) downto 0);
60                   output : out std_logic_vector((SIZE-1) downto 0)
61                );
62      end component reg;
63
64      component reg_file
65          generic (
66                      SIZE  : natural :=32;
67                      DELAY : time := 0 ns
68                  );
69          port (
70                   clk        : in std_logic;
71                   write_en   : in std_logic;
72                   read_r1    : in std_logic_vector(4 downto 0);
73                   read_r2    : in std_logic_vector(4 downto 0);
74                   write_r    : in std_logic_vector(4 downto 0);
75                   write_data : in std_logic_vector((SIZE-1) downto 0);
76                   data_r1    : out std_logic_vector((SIZE-1) downto 0);
```

```
77                  data_r2     : out std_logic_vector((SIZE-1) downto 0)
78              );
79      end component reg_file;
80
81      component alu
82          port(
83                  op_A      : in std_logic_vector(31 downto 0);
84                  op_B      : in std_logic_vector(31 downto 0);
85                  shamt     : in std_logic_vector(4 downto 0);
86                  alu_ctrl  : in t_aluOp;
87                  f         : out std_logic_vector(31 downto 0);
88                  zero      : out std_logic
89                  overflow  : out std_logic
90              );
91      end component alu;
92
93      signal PCDATA_int      : std_logic_vector((DATA_WIDTH - 1) downto 0);
94      signal PCOUT_int       : std_logic_vector((DATA_WIDTH - 1) downto 0);
95      signal instruction     : std_logic_vector((DATA_WIDTH - 1) downto 0);
96      signal mdreg           : std_logic_vector((DATA_WIDTH - 1) downto 0);
97      signal rf_write_data   : std_logic_vector((DATA_WIDTH - 1) downto 0);
98      signal rf_write_reg    : std_logic_vector(4 downto 0);
99      signal alu_a           : std_logic_vector((DATA_WIDTH - 1) downto 0);
100     signal alu_b           : std_logic_vector((DATA_WIDTH - 1) downto 0);
101     signal alu_reg_in      : std_logic_vector((DATA_WIDTH - 1) downto 0);
102     signal alu_reg_out     : std_logic_vector((DATA_WIDTH - 1) downto 0);
103     signal rega_in         : std_logic_vector((DATA_WIDTH - 1) downto 0);
104     signal regb_in         : std_logic_vector((DATA_WIDTH - 1) downto 0);
105     signal rega_out        : std_logic_vector((DATA_WIDTH - 1) downto 0);
106     signal regb_out        : std_logic_vector((DATA_WIDTH - 1) downto 0);
107     signal instr_sext      : std_logic_vector((DATA_WIDTH - 1) downto 0);
108     signal instr_sexts     : std_logic_vector((DATA_WIDTH - 1) downto 0);
109     signal zero            : std_logic;
110     signal overflow        : std_logic;
111     signal PCWrite_int     : std_logic;
112 begin
113     RF: reg_file
114     port map(
115                 clk => clk,
116                 write_en => RegWrite,
117                 read_r1 => instruction(25 downto 21),
118                 read_r2 => instruction(20 downto 16),
119                 write_r => rf_write_reg,
120                 write_data => rf_write_data,
121                 data_r1 => rega_in,
122                 data_r2 => regb_in
123             );
124
125     PC: reg
126     port map(
127                 en      => PCWRITE_int,
128                 data    => PCDATA_int,
129                 output => PCOUT_int
130             );
131
132     IR: reg
133     port map(
134                 en      => IRWrite,
```

```
135                 data   => mem_data_out,
136                 output => instruction
137             );
138     AOR: reg
139     port map(
140                  en => clk,
141                  data => alu_reg_in,
142                  output => alu_reg_out
143              );
144
145     MDR: reg
146     port map(
147                 en     => clk,
148                 data   => mem_data_out,
149                 output => mdreg
150             );
151
152     RRA: reg
153     port map(
154                  en     => clk,
155                  data   => rega_in,
156                  output => rega_out
157              );
158
159     RRB: reg
160     port map(
161                  en     => clk,
162                  data   => regb_in,
163                  output => regb_out
164              );
165
166     ALUU: alu
167     port map(
168                  op_A     => alu_a,
169                  op_B     => alu_b,
170                  alu_ctrl => ALUOp,
171                  f        => alu_reg_in,
172                  zero     => zero,
173                  shamt    => instruction(10 downto 6),
174                  overflow => overflow
175              );
176
177
178     mem_write_data <= regb_out;
179
180     ---- PC Write/Branch MUX ----
181     PCFinal : process(PCWriteCondEq, PCWriteCondNEq, PCWrite)
182     begin
183         if((PCWriteCondEq='1' and zero='1') or (PCWriteCondNEq='1' and zero
                ='0') or PCWrite='1') then
184             PCWRITE_int <= '1';
185         else
186             PCWRITE_int <= '0';
187         end if;
188     end process;
189
190     -- PCSource Mux --
191     PCSMux: process(PCSource, alu_reg_out, alu_reg_in)
```

```
192     begin
193         if(PCSource = PS_PCINC) then
194             PCDATA_int <= alu_reg_in;
195         elsif(PCSource = PS_ALUOUT) then
196             PCDATA_int <= alu_reg_out;
197         elsif(PCSource = PS_JMP) then
198             PCDATA_int <= PCOUT_int(31 downto 28) & instruction(25 downto
                    0) & "00";
199         end if;
200     end process;
201
202     -- IorD Mux
203     IorDMux : process (PCOUT_int, alu_reg_out, IorD)
204     begin
205         if(IorD = IOD_PC) then
206             addr_bus <= PCOUT_int;
207         else
208             addr_bus <= alu_reg_out;
209         end if;
210     end process;
211
212     -- MemToReg Mux --
213     MTRMux: process (MemToReg, mdreg, alu_reg_out)
214     begin
215         if(MemToReg = MTR_ALUOUT) then
216             rf_write_data <= alu_reg_out;
217         elsif(MemToReg = MTR_MDR) then
218             rf_write_data <= mdreg;
219         else
220             rf_write_data <= PCOUT_int;
221         end if;
222     end process;
223
224     -- RegDst Mux --
225     RDMux : process (instruction, RegDst)
226     begin
227         if(RegDst = RD_RT) then
228             rf_write_reg <= instruction(20 downto 16);
229         elsif(RegDst = RD_RD) then
230             rf_write_reg <= instruction(15 downto 11);
231         else
232             rf_write_reg <= "11111";
233         end if;
234     end process;
235
236     -- ALUSrcA Mux --
237     ALUSA: process (ALUSrcA, PCOUT_int, rega_out)
238     begin
239         if(ALUSrcA = ASA_PC) then
240             alu_a <= PCOUT_int;
241         else
242             alu_a <= rega_out;
243         end if;
244     end process;
245
246     -- ALUSrcB Mux --
247     ALUSB: process (ALUSrcB, regb_out, instr_sext, instr_sexts)
248     begin
```

```
249             if(ALUSrcB = ASB_REGB) then
250                 alu_b <= regb_out;
251             elsif(ALUSrcB = ASB_FOUR) then
252                 alu_b <= "00000000000000000000000000000100";
253             elsif(ALUSrcB = ASB_SEXT) then
254                 alu_b <= instr_sext;
255             else
256                 alu_b <= instr_sexts;
257             end if;
258         end process;
259 
260         -- Sign Extend and Shift --
261         SEXTS: process(instruction)
262         begin
263             instr_sext <= (31 downto 16 => instruction(15)) & instruction(15
                downto 0);
264             instr_sexts <= (31 downto 18 => instruction(15)) & instruction(15
                downto 0) & "00";
265         end process;
266 
267 end architecture datapath_1;
```

Listing 5: Datapath