LAB: SINGLE CYCLE DATAPATH

Brandon Chin

CSC343 - Instructor: Prof. Izidor Gertner

3/27/2015

Table of Contents

Objective... **3**

Functionality and Specifications.. **3**

Register File.. 3

Component: Modified SRAM Cell.. 4

Component: Modified SRAM Register... 5

Component: Modified SRAM Chip.. 5

Instruction Register... 6

CPU 8

Component: 4-to-16 Decoder 10

Component: Sign Extension 10

Component: 16-Bit Multiplexor 11

Simulation **11**

Vector Waveform Simulations 11

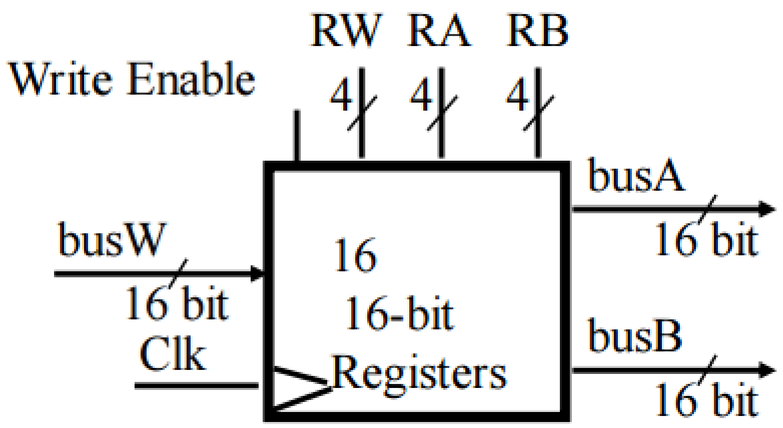
Conclusion **14**

Objective

We must design a single cycle CPU datapath which can store memory and implement 16-bit addition and subtraction. The CPU is composed of an instruction register, a register file, and an ALU (arithmetic logic unit). The instruction register is used to store a 16-bit instruction string that can either store a value into the register file or add/subtract two values from the register file. The register file is a 16x16 SRAM chip which can store 16 different values, each 16 bits long. The ALU takes two parameters and is capable of performing two operations on these values -- addition or subtraction. We must then demonstrate this design on the DE2 Circuit Board.

Functionality and Specifications

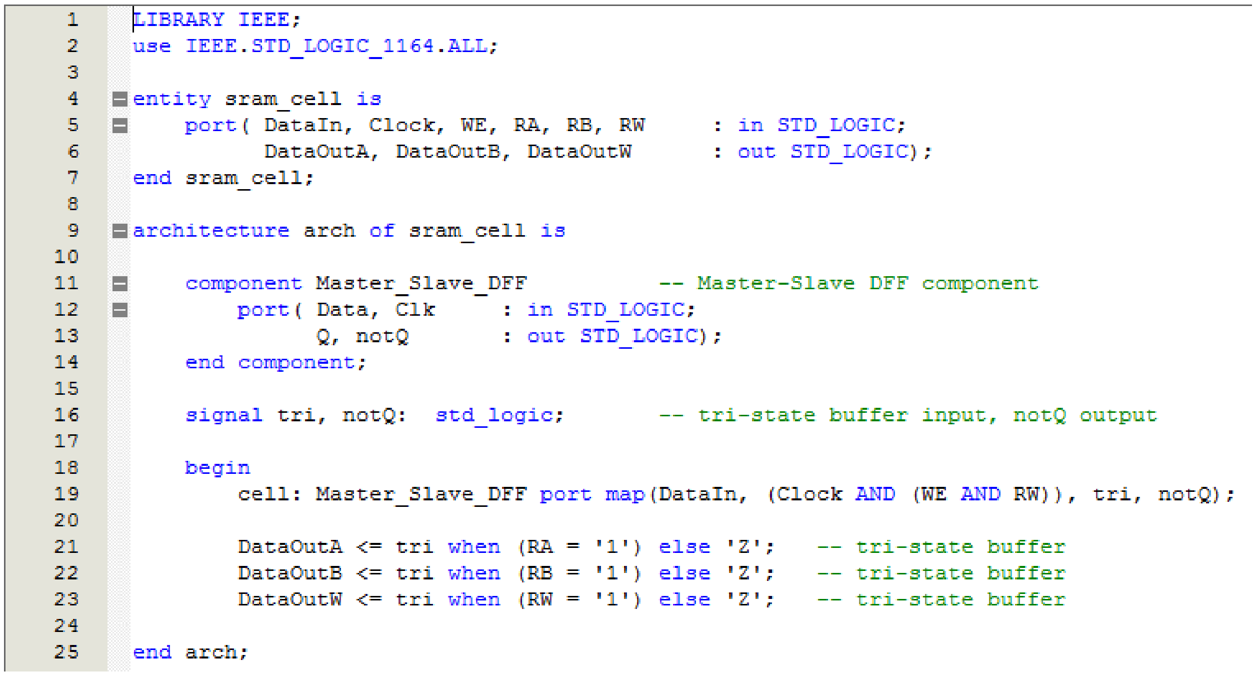
Register File



Register File Diagram

The register file is an extended SRAM chip from the previous lab. It takes in several inputs -- the Write Enable, RA, RB, RW, and busW, as well as a clock to regulate data flow (operations occur on the falling edge of the clock). There are also outputs busA and busB. The write enable must be on in order to allow the register file to store data. RA and RB are the register locations of the data that we want to pass into busA and busB respectively. RW represents the register location that we want to store our incoming data. BusW represents the data that we want to store into the register specified by RW. In the design, I have renamed busW to DataIn, and I have instead made busW an output which will serve as a way to view the register we are currently looking at.

Component: Modified SRAM Cell

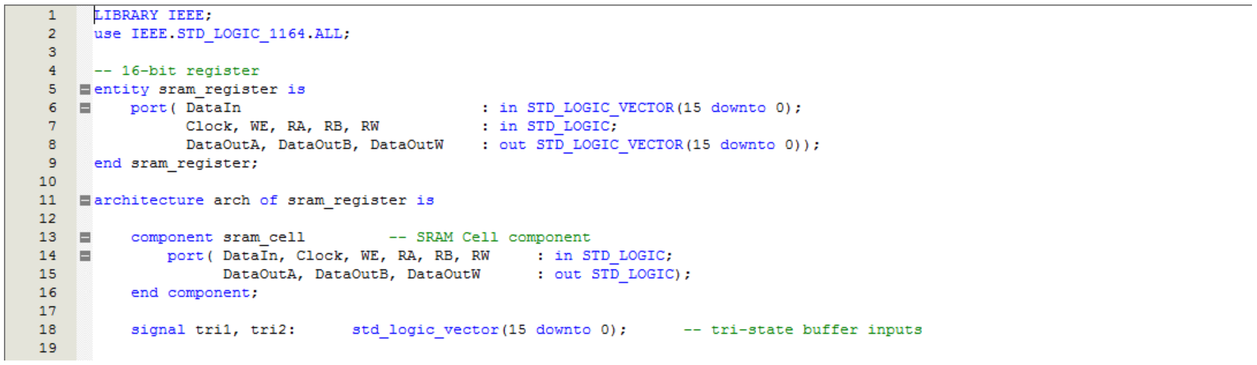


```vhdl
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sram_cell is
    port( DataIn, Clock, WE, RA, RB, RW      : in STD_LOGIC;
          DataOutA, DataOutB, DataOutW       : out STD_LOGIC);
end sram_cell;

architecture arch of sram_cell is

    component Master_Slave_DFF          -- Master-Slave DFF component
        port( Data, Clk     : in STD_LOGIC;
              Q, notQ       : out STD_LOGIC);
    end component;

    signal tri, notQ:  std_logic;       -- tri-state buffer input, notQ output

    begin
        cell: Master_Slave_DFF port map(DataIn, (Clock AND (WE AND RW)), tri, notQ);

        DataOutA <= tri when (RA = '1') else 'Z';   -- tri-state buffer
        DataOutB <= tri when (RB = '1') else 'Z';   -- tri-state buffer
        DataOutW <= tri when (RW = '1') else 'Z';   -- tri-state buffer

end arch;
```

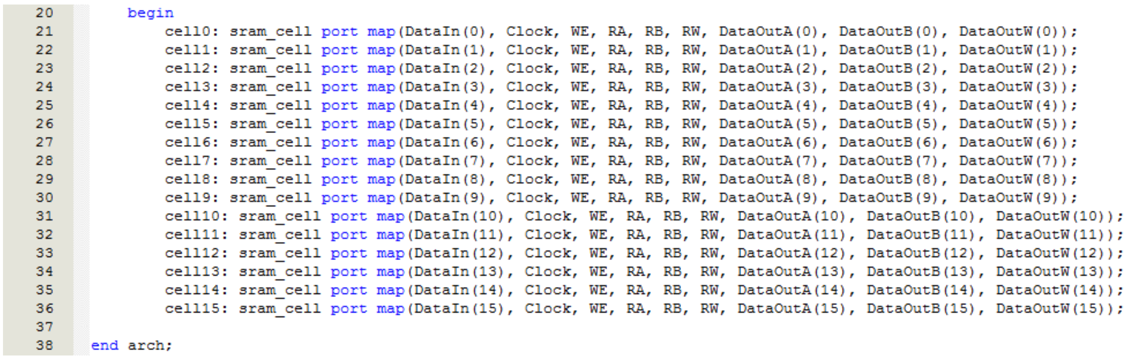
sram_cell.vhd

This is a modified cell from the SRAM in the previous lab. The flip-flop has a clock input that does not turn to high unless the clock signal, write enable, and the RW inputs are all on. This then allows the flip-flop to store new information (DataIn). When storing information, the DataOutW output displays the information we have just stored. Once information is stored, we can access this information by specifying which output we want our information to go to. By turning RA on, we forward the stored information to DataOutA, and by turning RB on, we forward the stored information to DataOutB. This will become useful later on when we want to specify whether we want the information stored in the particular flip-flop to go into the ALU as busA or busB.

Component: Modified SRAM Register



```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- 16-bit register
entity sram_register is
    port( DataIn                        : in STD_LOGIC_VECTOR(15 downto 0);
          Clock, WE, RA, RB, RW         : in STD_LOGIC;
          DataOutA, DataOutB, DataOutW  : out STD_LOGIC_VECTOR(15 downto 0));
end sram_register;

architecture arch of sram_register is

    component sram_cell         -- SRAM Cell component
        port( DataIn, Clock, WE, RA, RB, RW     : in STD_LOGIC;
              DataOutA, DataOutB, DataOutW      : out STD_LOGIC);
    end component;

    signal tri1, tri2:      std_logic_vector(15 downto 0);      -- tri-state buffer inputs
```

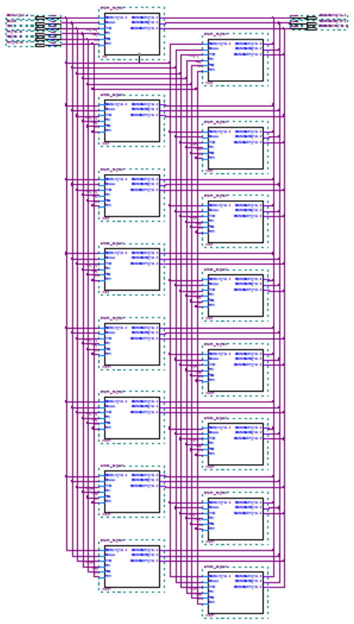


```
    begin
        cell0: sram_cell port map(DataIn(0), Clock, WE, RA, RB, RW, DataOutA(0), DataOutB(0), DataOutW(0));
        cell1: sram_cell port map(DataIn(1), Clock, WE, RA, RB, RW, DataOutA(1), DataOutB(1), DataOutW(1));
        cell2: sram_cell port map(DataIn(2), Clock, WE, RA, RB, RW, DataOutA(2), DataOutB(2), DataOutW(2));
        cell3: sram_cell port map(DataIn(3), Clock, WE, RA, RB, RW, DataOutA(3), DataOutB(3), DataOutW(3));
        cell4: sram_cell port map(DataIn(4), Clock, WE, RA, RB, RW, DataOutA(4), DataOutB(4), DataOutW(4));
        cell5: sram_cell port map(DataIn(5), Clock, WE, RA, RB, RW, DataOutA(5), DataOutB(5), DataOutW(5));
        cell6: sram_cell port map(DataIn(6), Clock, WE, RA, RB, RW, DataOutA(6), DataOutB(6), DataOutW(6));
        cell7: sram_cell port map(DataIn(7), Clock, WE, RA, RB, RW, DataOutA(7), DataOutB(7), DataOutW(7));
        cell8: sram_cell port map(DataIn(8), Clock, WE, RA, RB, RW, DataOutA(8), DataOutB(8), DataOutW(8));
        cell9: sram_cell port map(DataIn(9), Clock, WE, RA, RB, RW, DataOutA(9), DataOutB(9), DataOutW(9));
        cell10: sram_cell port map(DataIn(10), Clock, WE, RA, RB, RW, DataOutA(10), DataOutB(10), DataOutW(10));
        cell11: sram_cell port map(DataIn(11), Clock, WE, RA, RB, RW, DataOutA(11), DataOutB(11), DataOutW(11));
        cell12: sram_cell port map(DataIn(12), Clock, WE, RA, RB, RW, DataOutA(12), DataOutB(12), DataOutW(12));
        cell13: sram_cell port map(DataIn(13), Clock, WE, RA, RB, RW, DataOutA(13), DataOutB(13), DataOutW(13));
        cell14: sram_cell port map(DataIn(14), Clock, WE, RA, RB, RW, DataOutA(14), DataOutB(14), DataOutW(14));
        cell15: sram_cell port map(DataIn(15), Clock, WE, RA, RB, RW, DataOutA(15), DataOutB(15), DataOutW(15));

end arch;
```

sram_register.vhd

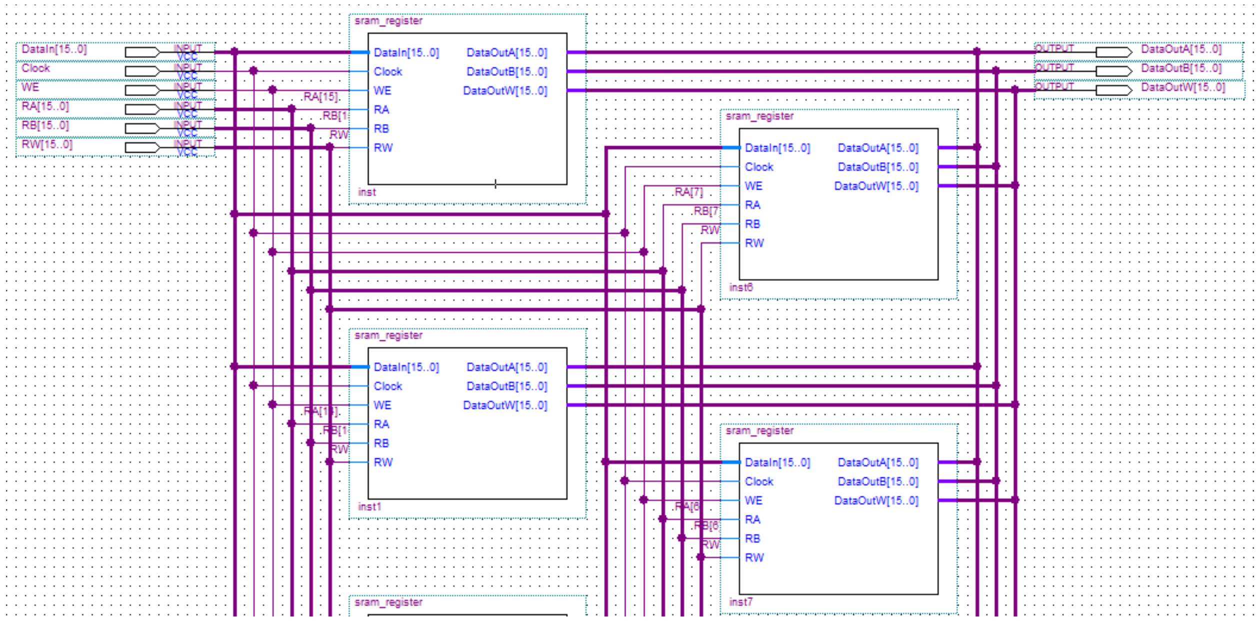
This register is composed of 16 SRAM cells, allowing the register to store words up to 16 bits.

Component: Modified SRAM Chip



sram_chip.bdf

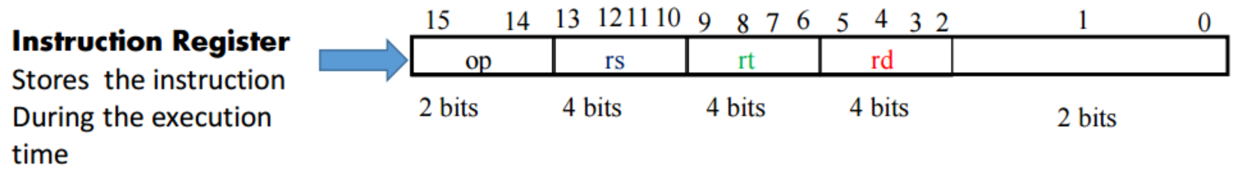
The SRAM chip contains 16 SRAM registers total, that way we can store up to 16 different words, each being 16 bits long. This memory chip will make up the register file that will be used in conjunction with the rest of the CPU. Let's take a closer look at a section of the layout of this register file:



sram_chip.bdf -- close up

Now we can see that if we want to store a word, we must specify the register in which to store this word using the RW input. The register that receives an RW of '1' will store the word coming in from the DataIn input. RA and RB will select which register to forward its stored data to DataOutA or DataOutB respectively.

Instruction Register



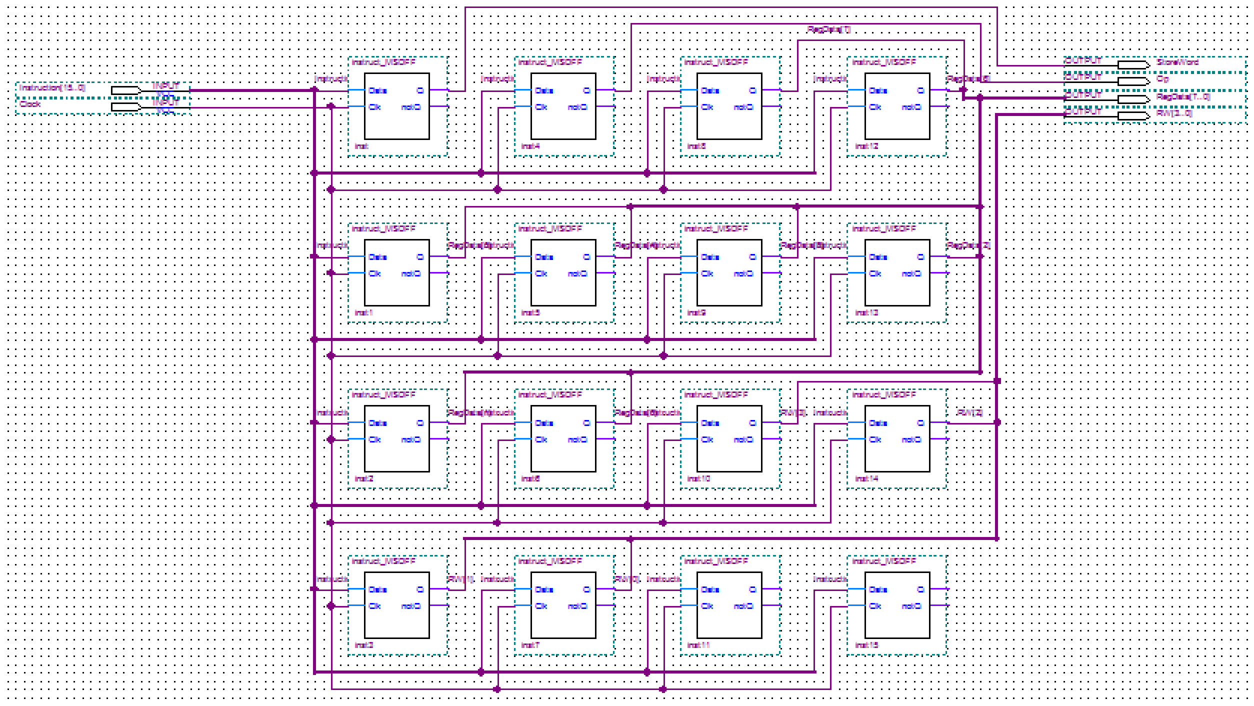
Instruction Register

In this lab, this instruction format can perform three operations -- add, subtract, or store. The two most significant bit represent the op code. When bit 15 is 1, it means we are storing a word into the register file, and bit 14 becomes irrelevant, or a "don't care". However, when bit 15 is 0, then we look at bit 14, which represents the ALU operation (0 is add, 1 is subtract).

Bit[15]	Bit[14]	Operation
0	0	Add
0	1	Subtract
1	X	Store Word

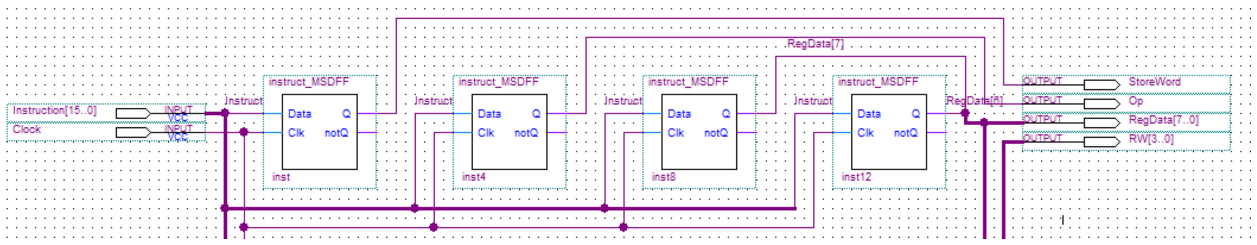
Op Code Logic Table

The following bits RS, RT, and RD are each 4 bits long and represent the registers we are reading from when we are either adding or subtracting. RS and RT are the two registers we are sending through the register file as RA and RB respectively, and to the ALU to be added/ subtracted. RD is the register in which we are going to store the result. When we are storing however, RS and RT are combined together to represent the 8-bit word that we are going to store into register RD. (The word we are storing will be sent through a sign extension component to be converted into a 16-bit word so that it can be stored in the register file or used in the ALU). The remaining two least significant bits are not used in our instruction register.



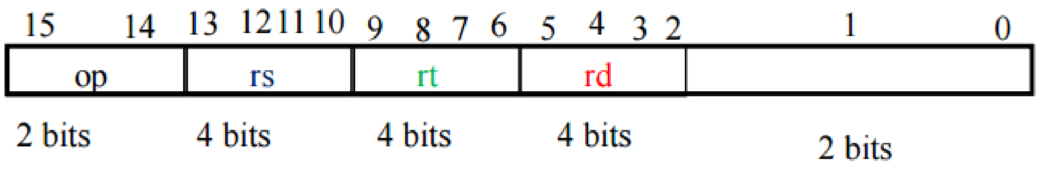
instruction_register.bdf

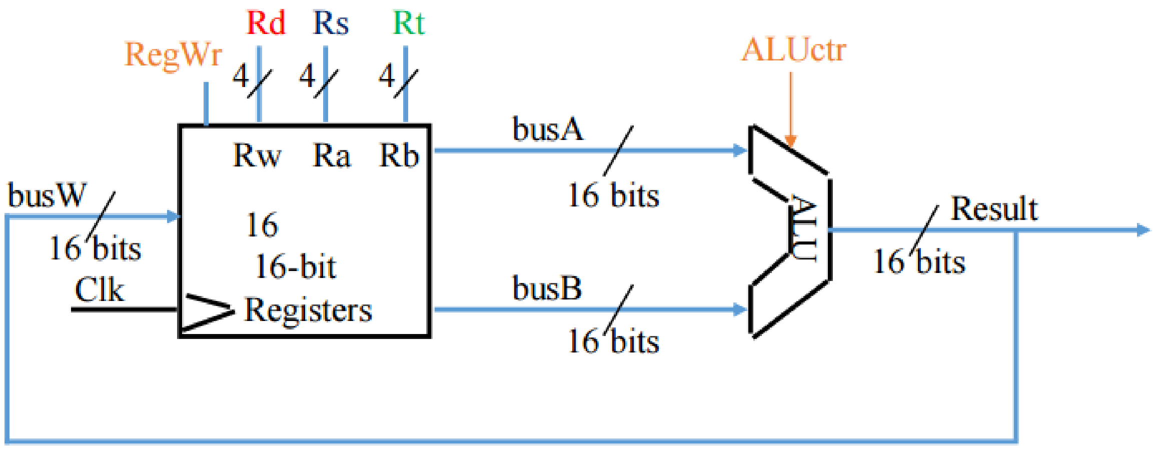
The instruction register simply takes in a 16-bit string input and a clock input, and stores the bits in 16 flip-flops. The bits are then forwarded to their corresponding outputs, for example, one flip-flop goes to StoreWord, another to the operation output (Op), 8 of them to the register data output (RegData), and 4 of them to the RW output. Let's tke a closer look at the instruction register:



instrcution_register.bdf -- close up

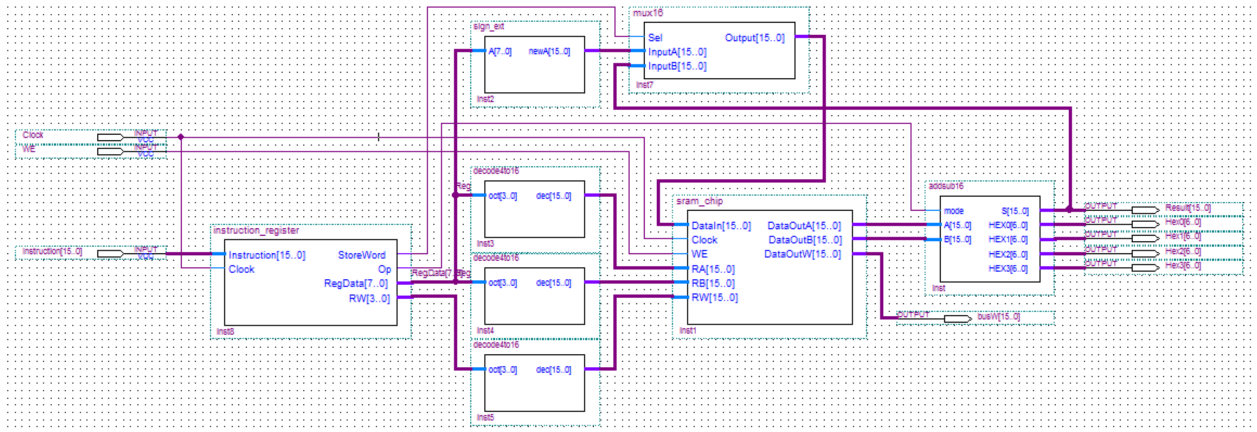
CPU



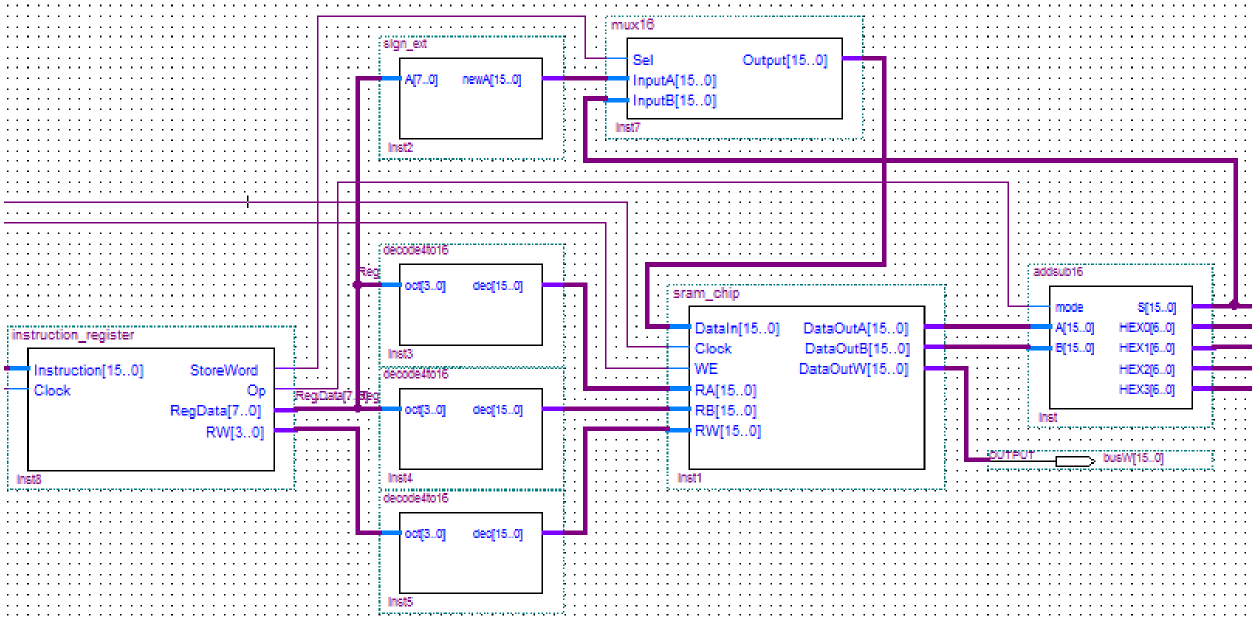


Simplified Single Cycle Datapath

Here we can see the major components put together. The instruction register is processed and Rs, Rt, Rd, and bit[15] (RegWr) are sent to the register file, while bit[14] (ALUCtr) is sent to the ALU. The register file reads these inputs and sends the data stored in Rs and Rt to busA and busB as inputs for the ALU. The result is then written back to the register file as busW to be stored in Rd.



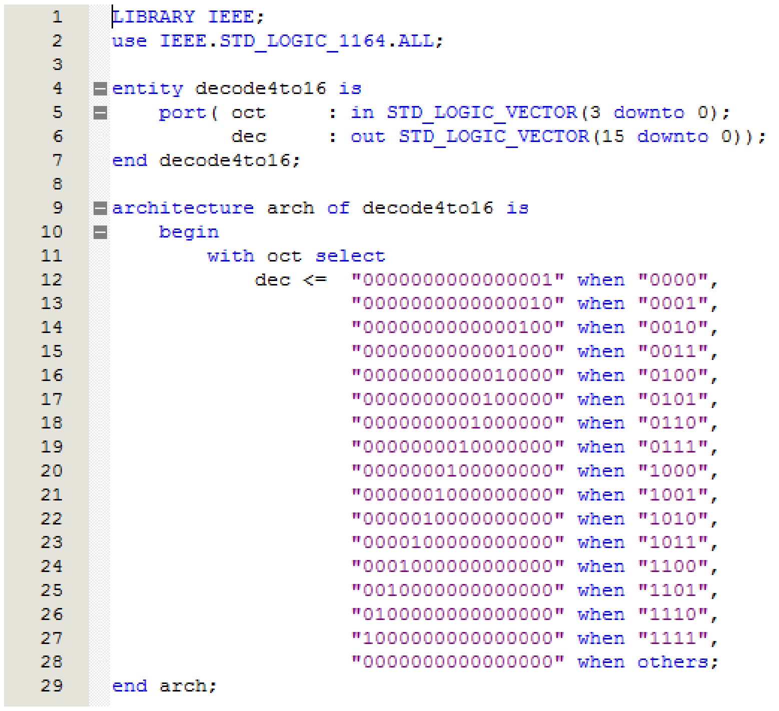
CPU.bdf



CPU.bdf -- close up

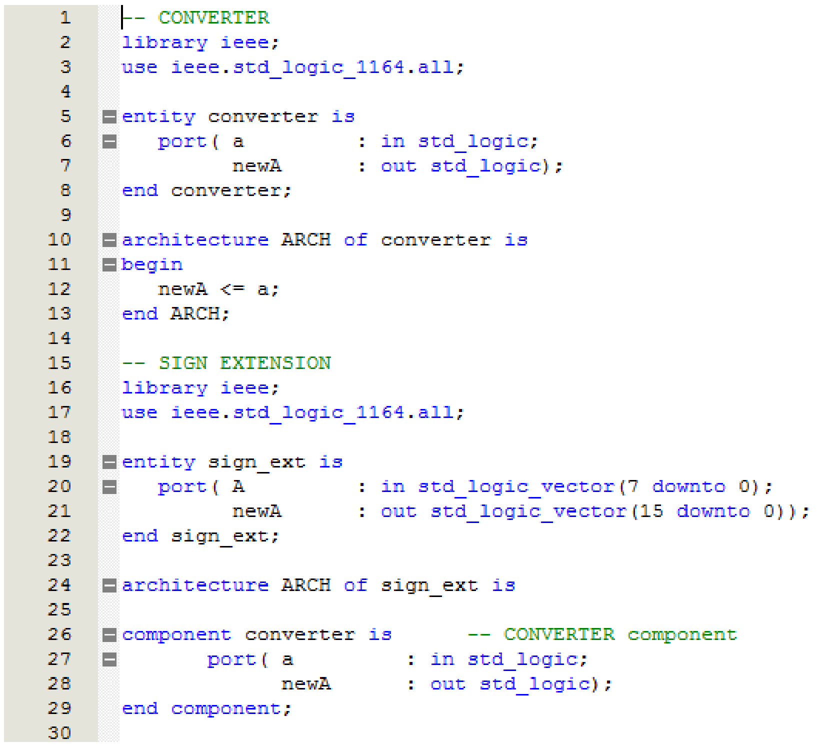
In addition to the instruction register, the register file (sram chip) and the ALU (addsub16 from a previous lab), we can see a few other components necessary to complete this single cycle CPU datapath. The three register outputs of the instruction must be passed through 4-to-16 decoders so that they can be used in the register file. The data coming out of the instruction must be sign extended from 8 bits to16 bits so that it can be stored into the register file. Finally, the data coming into the register file must be passed through a multiplexor in order to distinguish between storing an instructed word or storing the result of an arithmetic operation.

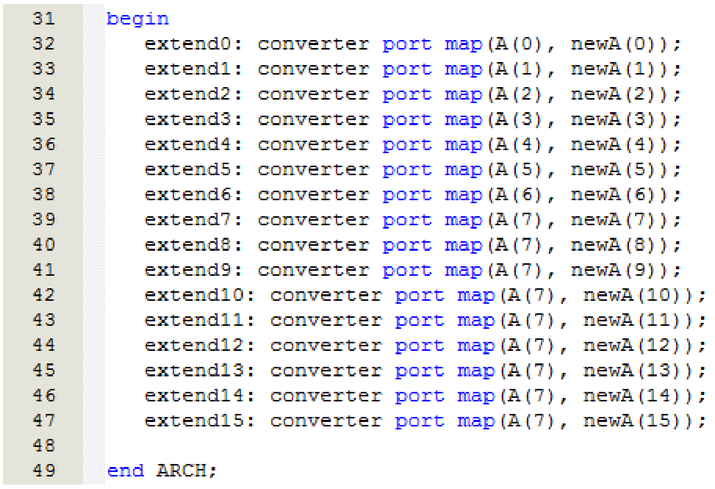
Component: 4-to-16 Decoder


```vhdl
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decode4to16 is
    port( oct      : in STD_LOGIC_VECTOR(3 downto 0);
          dec      : out STD_LOGIC_VECTOR(15 downto 0));
end decode4to16;

architecture arch of decode4to16 is
    begin
        with oct select
            dec <=  "0000000000000001" when "0000",
                    "0000000000000010" when "0001",
                    "0000000000000100" when "0010",
                    "0000000000001000" when "0011",
                    "0000000000010000" when "0100",
                    "0000000000100000" when "0101",
                    "0000000001000000" when "0110",
                    "0000000010000000" when "0111",
                    "0000000100000000" when "1000",
                    "0000001000000000" when "1001",
                    "0000010000000000" when "1010",
                    "0000100000000000" when "1011",
                    "0001000000000000" when "1100",
                    "0010000000000000" when "1101",
                    "0100000000000000" when "1110",
                    "1000000000000000" when "1111",
                    "0000000000000000" when others;
end arch;
```

decode4to16.vhd

Component: Sign Extension

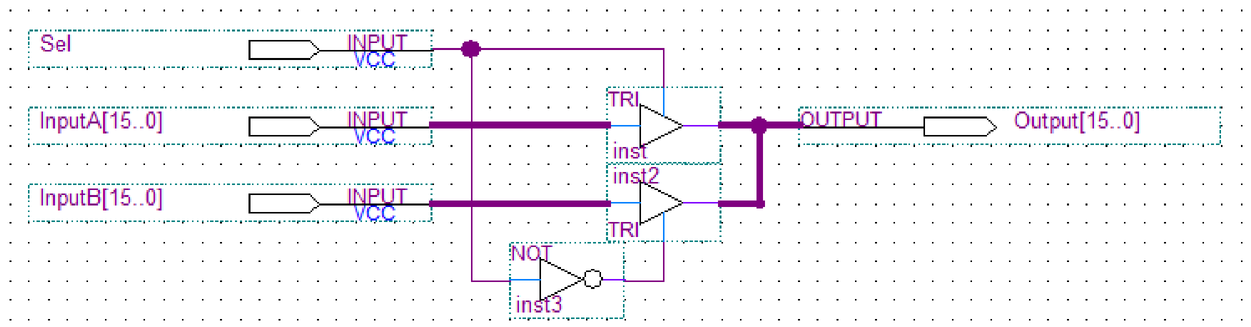

```vhdl
-- CONVERTER
library ieee;
use ieee.std_logic_1164.all;

entity converter is
   port( a          : in std_logic;
         newA       : out std_logic);
end converter;

architecture ARCH of converter is
begin
   newA <= a;
end ARCH;

-- SIGN EXTENSION
library ieee;
use ieee.std_logic_1164.all;

entity sign_ext is
   port( A          : in std_logic_vector(7 downto 0);
         newA       : out std_logic_vector(15 downto 0));
end sign_ext;

architecture ARCH of sign_ext is

component converter is       -- CONVERTER component
       port( a          : in std_logic;
             newA       : out std_logic);
end component;

```



```
begin
   extend0: converter port map(A(0), newA(0));
   extend1: converter port map(A(1), newA(1));
   extend2: converter port map(A(2), newA(2));
   extend3: converter port map(A(3), newA(3));
   extend4: converter port map(A(4), newA(4));
   extend5: converter port map(A(5), newA(5));
   extend6: converter port map(A(6), newA(6));
   extend7: converter port map(A(7), newA(7));
   extend8: converter port map(A(7), newA(8));
   extend9: converter port map(A(7), newA(9));
   extend10: converter port map(A(7), newA(10));
   extend11: converter port map(A(7), newA(11));
   extend12: converter port map(A(7), newA(12));
   extend13: converter port map(A(7), newA(13));
   extend14: converter port map(A(7), newA(14));
   extend15: converter port map(A(7), newA(15));

end ARCH;
```

sign_ext.vhd

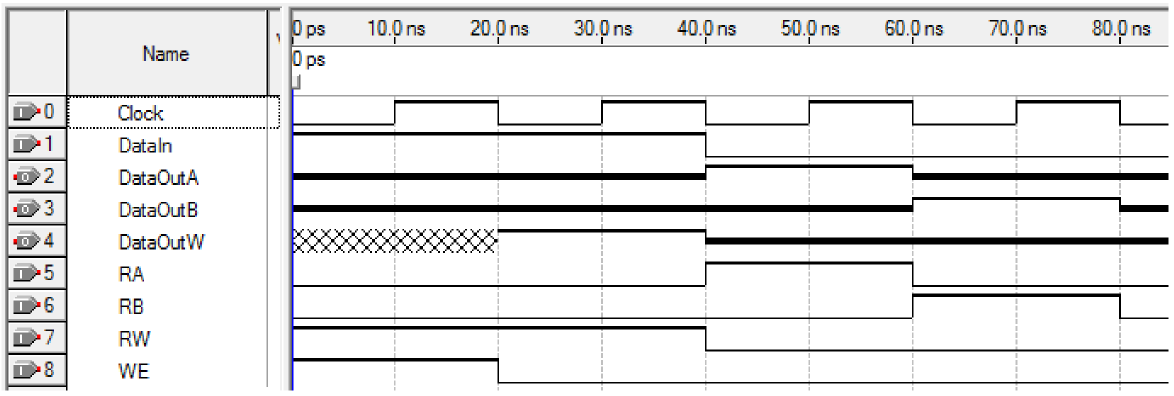
Component: 16-Bit Multiplexor



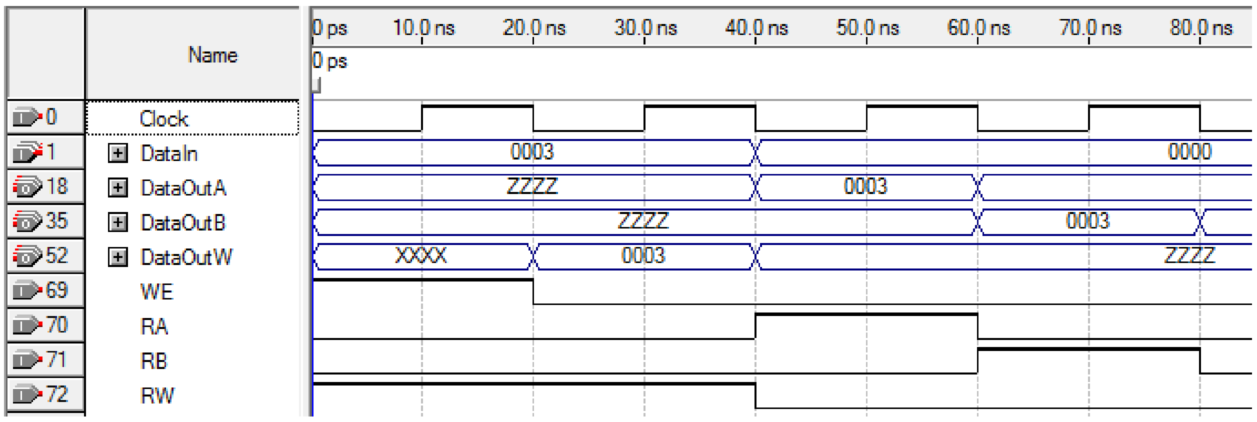
mux16.bdf

Simulation

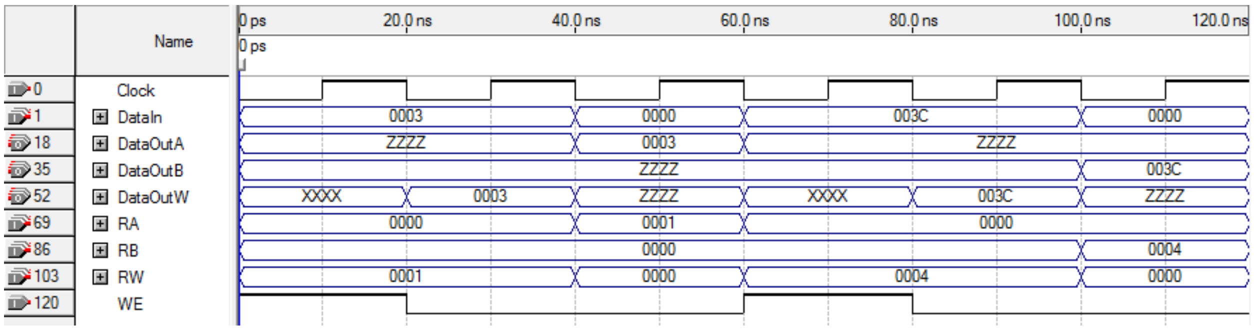
Vector Waveform Simulations



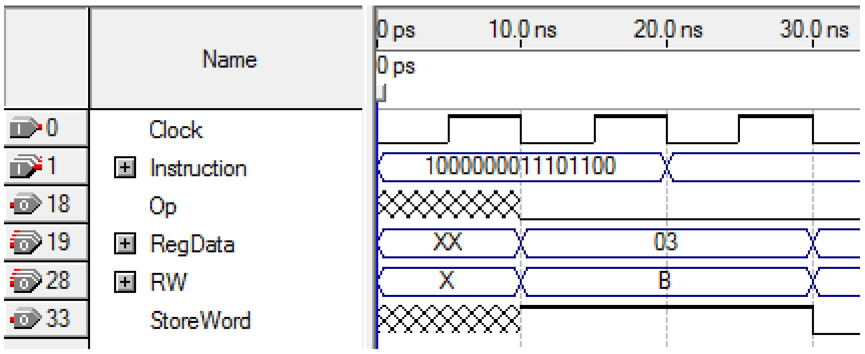
test_sram_cell.vwf



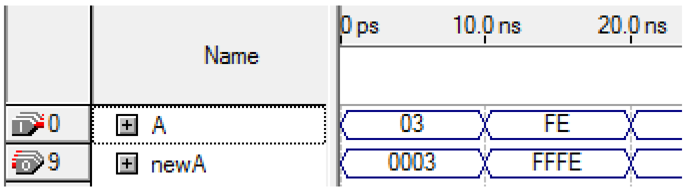
test_sram_register.vwf



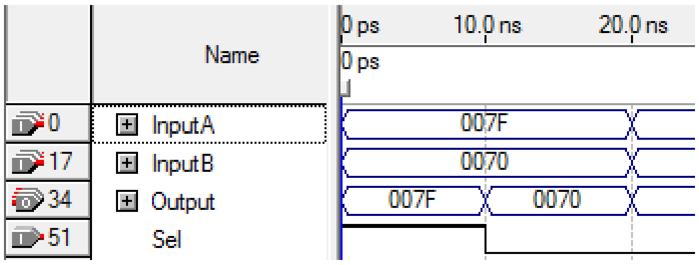
test_sram_chip.vwf



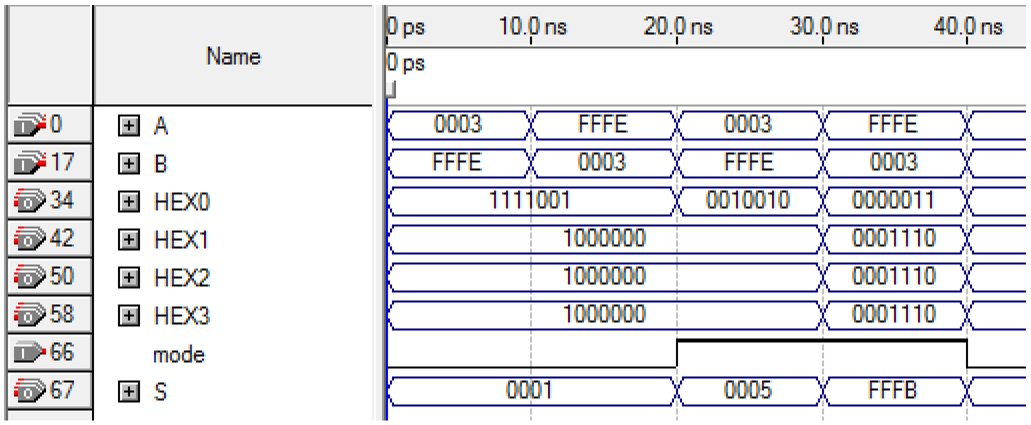
test_instruction_register.vwf



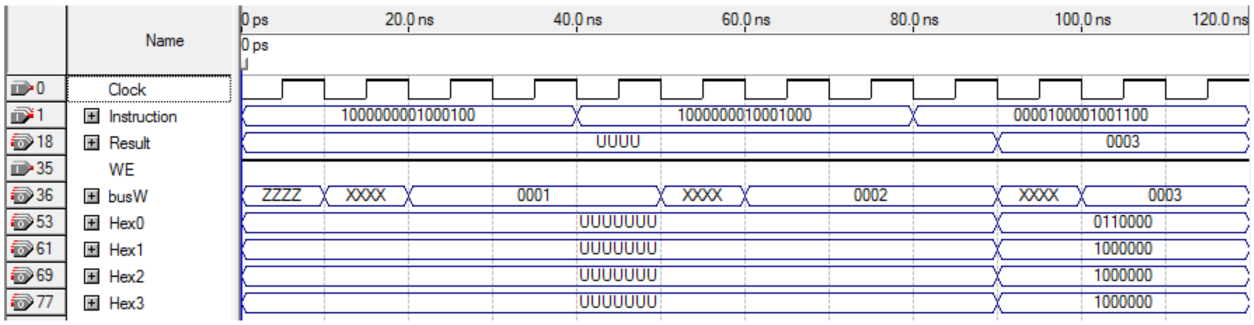
test_sign_ext.vwf



test_mux16.vwf



test_addsub16.vwf



test_CPU.vwf

Conclusion

We have designed a single cycle CPU datapath that can process an instruction and implement operations to either store 16-bit information into memory or perform addition/ subtraction on two 16-bit words. The CPU is composed of an instruction register, a register file, and an ALU (arithmetic logic unit), as well as a few intermediate components such as some decoders, sign extension, and a multiplexor. We then demonstrated and verified our design with vector waveform simulation files.