Lab 5: Pipelining the Small16 Processor

By Duy Nguyen

Department of Engineering, CSU Fullerton

EGCP 381: Computer Design and Organization

Dr. Mike Turi

May 13, 2022

Introduction

The purpose of this lab is to be able to understand the concept of pipelining and be able to pipeline the program from the single-cycle small16 processor. We will create new registers and components in each stage of the pipelined datapath based on the instruction chart, and also remove unnecessary components from lab 3. Then reconnecting the signal that connected each component to create a complete datapath that performs the instruction.

Procedure

At the very beginning, I examined the differences between the datapath chart of single-cycle and the pipelined datapath chart. For the pipeline, it divided the datapath into 3 stages and data will go into newly created components to flow to the next stage. Instruction_Word, Opcode, Immediate, Data, Reg Value, and Reg are components that separate each stage. There is also a register ABCD that replaces the accumulator. In stage 1 we will perform the instruction fetch from the instruction memory. Then we will register the fetch, decode the instruction, execute the instruction and read the data memory in stage 2. Lastly, we will write the data back into the ABCD register file. By dividing the datapath into stages, we can increase the throughput by performing the new instruction in stage 1 when the previous instruction is moving into stage 2.

Starting off with VHDL, I decided to separate my control file into two control files, one for control signals in stage 1 and stage 2, the other one for control signals in stage 3. The reason why I combined stage 1 and stage 2 is that stage 1 only has a PC enable control signal, which it should always enable. Then I created enable signals for new registers that are introduced in the datapath. So for stage 1 and stage 2, my control signals are

```
entity control_s2 is
    port( CLK   : in std_logic;
          RESET : in std_logic;
          START : in std_logic;

          WE     : out std_logic;
          --ALU_OP : out std_logic_vector(1 downto 0);
          --B_INV  : out std_logic;
          --CIN    : out std_logic;
          --A_SEL  : out std_logic;
          --B_SEL  : out std_logic;
          --PC_SEL : out std_logic;
          --EN_A   : out std_logic;
          EN_PC  : out std_logic;

          --EN_ABCD : out std_logic;
          --Z_FLAG   : in std_logic;
          --N_FLAG   : in std_logic;
          --INSTR_OP : in sml6_opcode
          EN_InstrWord : out std_logic;
          EN_OPCODE : out std_logic;
          EN_Immediate: out std_logic;
          EN_DATA : out std_logic;
          EN_RegVal : out std_logic;
          EN_REG : out std_logic;

          Instr_OP_S2 : in sml6_opcode

          );
end control_s2;
```

Since there are opcodes for both stage 2 and stage 3, I also created two opcode signals, one for each control file. Based on these signals, I need to create architecture for the opcode, read/write control, and load/hold control.

```
-- control Architecture Description
architecture behavorial of control_s2 is

    -- op codes
    constant op_add   : sml6_opcode := "0000";
    constant op_sub   : sml6_opcode := "0001";
    constant op_load  : sml6_opcode := "0010";
    constant op_store : sml6_opcode := "0011";
    constant op_addi  : sml6_opcode := "0100";
    constant op_seti  : sml6_opcode := "0101";
    constant op_jump  : sml6_opcode := "0110";
    constant op_jz    : sml6_opcode := "0111";

        -- register load control
        constant hold : std_logic := '0';
        constant load : std_logic := '1';

        -- data memory write enable control
        constant rd : std_logic := '0';
        constant wr : std_logic := '1';

        --pc select
        --constant from_plus1 : std_logic := '0';
        --constant from_alu   : std_logic := '1';
 --MOVED TO S3
    -- control signal values
```

Since we need to enable these signals to perform store instructions, I need to load all of them.

```
  elsif Instr_OP_S2 = op_store then

      EN_PC <= load;    EN_InstrWord <= load;
      EN_OPCODE <= load; EN_Immediate <= load;
      EN_DATA <= load;  EN_RegVal <= load;
      EN_REG <= load;  pre_we <= wr;

      --EN_A  <= load;
          --PC_SEL <= from_plus1;
      --B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_nop;
      --A_SEL <= a_a;     B_SEL  <= b_mem;

      next_state <= running;
```

The control file for stage 3 will have signals for ALU, Amux Sel, Bmux Sel and Opcode for stage 3, and also the enable signal for the ABCD register.

```
entity control_s3 is
    port( CLK   : in std_logic;
          RESET : in std_logic;
          START : in std_logic;

          --WE     : out std_logic;
          ALU_OP : out std_logic_vector(1 downto 0);
          B_INV  : out std_logic;
          CIN    : out std_logic;
          A_SEL  : out std_logic;
          B_SEL  : out std_logic;


          EN_ABCD : out std_logic;

          Instr_OP_S3 : in sml6_opcode
          );
end control_s3;


architecture behavorial of control_s3 is

  -- register load control
       constant hold : std_logic := '0';
       constant load : std_logic := '1';

   -- control signal values
       -- alu operations
       constant alu_nop : std_logic_vector(1 downto 0) := "00";
       constant alu_and : std_logic_vector(1 downto 0) := "00";
       constant alu_or  : std_logic_vector(1 downto 0) := "01";
       constant alu_add : std_logic_vector(1 downto 0) := "10";
       constant alu_mult : std_logic_vector(1 downto 0) := "11";

  -- a select control
       constant a_0 : std_logic := '0';
       constant a_a : std_logic := '1';

  -- b select control
       constant b_mem : std_logic := '0';
       constant b_imm : std_logic := '1';

  -- b invert control
       constant pos : std_logic := '0';
       constant inv : std_logic := '1';

 -- op codes
    constant op_add   : sml6_opcode := "0000";
    constant op_sub   : sml6_opcode := "0001";
    constant op_load  : sml6_opcode := "0010";
    constant op_store : sml6_opcode := "0011";
    constant op_addi  : sml6_opcode := "0100";
    constant op_seti  : sml6_opcode := "0101";
    constant op_jump  : sml6_opcode := "0110";
    constant op_jz    : sml6_opcode := "0111";
```

```
elsif Instr_OP_S3 = op_store then

    B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_or;
    A_SEL <= a_a;     B_SEL  <= b_mem;
    EN_ABCD  <= load;

    next_state <= running;
```

Then the behavior of these signals will be the same as what we had done in lab 3.

Next, I modified the processor file by removing the component of the old control and adding components of my control_s2 and control_s3 into the processor to be able to create connecting signals between each component.

```
component control_s2 is
    port( CLK   : in std_logic;
      RESET : in std_logic;
      START : in std_logic;

      WE     : out std_logic;

      EN_PC  : out std_logic;


      EN_InstrWord : out std_logic;
      EN_OPCODE : out std_logic;
      EN_Immediate: out std_logic;
      EN_DATA : out std_logic;
      EN_RegVal : out std_logic;
      EN_REG : out std_logic;

      Instr_OP_S2 : in sml6_opcode

      );
end component;

component control_s3 is

    port( CLK   : in std_logic;
      RESET : in std_logic;
      START : in std_logic;

      --WE     : out std_logic;
      ALU_OP : out std_logic_vector(1 downto 0);
      B_INV  : out std_logic;
      CIN    : out std_logic;
      A_SEL  : out std_logic;
      B_SEL  : out std_logic;
```

Since we replaced Accumulator with the ABCD register and removed PC select, I commented out those signals. I also created connection signals for these new registers.

```
signal DataAddress_Connect, PCAddress_Connect : sml6_address;
signal Data_IntoMem_Connect : sml6_data;
signal DataOut_OutofMem_Connect, Instruction_Connect : sml6_data;
signal ReadWrite_Connect : std_logic;


signal EN_ABCD_Connect : std_logic;

signal InstrOpCode_Connect : sml6_opcode;
signal ALUOp_Connect : std_logic_vector(1 downto 0);
signal Binv_Connect  : std_logic;
signal Cin_Connect   : std_logic;

signal ASel_Connect  : std_logic;
signal BSel_Connect  : std_logic;
--signal PCSel_Connect : std_logic;

--signal EnA_Connect  : std_logic;
signal EnPC_Connect : std_logic;

signal EN_InstrWord_Connect: std_logic;
signal EN_OPCODE_Connect: std_logic;
signal EN_Immediate_Connect: std_logic;
signal EN_DATA_Connect: std_logic;
signal EN_RegVal_Connect: std_logic;
signal EN_REG_Connect: std_logic;
signal Instr_OP_S2_Connect: sml6_opcode;
signal Instr_OP_S3_Connect: sml6_opcode;
```

Then I assigned pipelined connections to the corresponding pipelined registers.

```
the_control_s2: control_s2 port map (
    CLK   => CLK,
    RESET => RESET,
    START => START,
    WE     => ReadWrite_Connect,
    EN_PC => EnPC_Connect,

    EN_InstrWord => EN_InstrWord_Connect,
    EN_OPCODE => EN_OPCODE_Connect,
    EN_Immediate => EN_Immediate_Connect,
    EN_DATA => EN_DATA_Connect,
    EN_RegVal => EN_RegVal_Connect,
    EN_REG => EN_REG_Connect,
    Instr_OP_S2 => Instr_OP_S2_Connect
    );

the_control_s3: control_s3 port map (
    CLK   => CLK,
    RESET => RESET,
    START => START,
    --WE     => ReadWrite_Connect,
    ALU_OP => ALUOp_Connect,
    B_INV  => Binv_Connect,
    CIN    => Cin_Connect,
    A_SEL  => ASel_Connect,
    B_SEL  => BSel_Connect,
    EN_ABCD => EN_ABCD_Connect,
    Instr_OP_S3 => Instr_OP_S3_Connect
    );
```

Lastly, I rearranged my datapath file based on components, and signals that I had created for this pipelined datapath. Since the ABCD register is replaced accumulator, I need to add the component of ABCD into my datapath.

```
component ABCDRegFile is
    port( CLK: in std_logic;
          RESET: in std_logic;

    -- ABCD*
          RD_REG : in std_logic_vector(1 downto 0);  -- Which register to read and output
          REG_OUT : out sml6_data;

          ABCD_WE : in std_logic; -- Write enable signal
          WR_REG : in std_logic_vector(1 downto 0);  -- Which register to write to
          REG_IN : in sml6_data);
end component;


signal zero_16 : sml6_data := "0000000000000000";
signal alu_a, alu_b, alu_out : sml6_data;
signal pc_out, pc_in : sml6_address;
signal a_out, immediate_zero_extend_out : sml6_data;
signal adder_out : sml6_data;
signal instr_word_out, immediate_out, the_data_out, reg_value_out, reg_value_in : sml6_data;
signal reg_out: std_logic_vector(1 downto 0);
```

To be able to transfer signals into and out of pipelined components, I need to create the mapping for each of them. Then assigned the correct signal for each input and output just like how we did in lab 3.

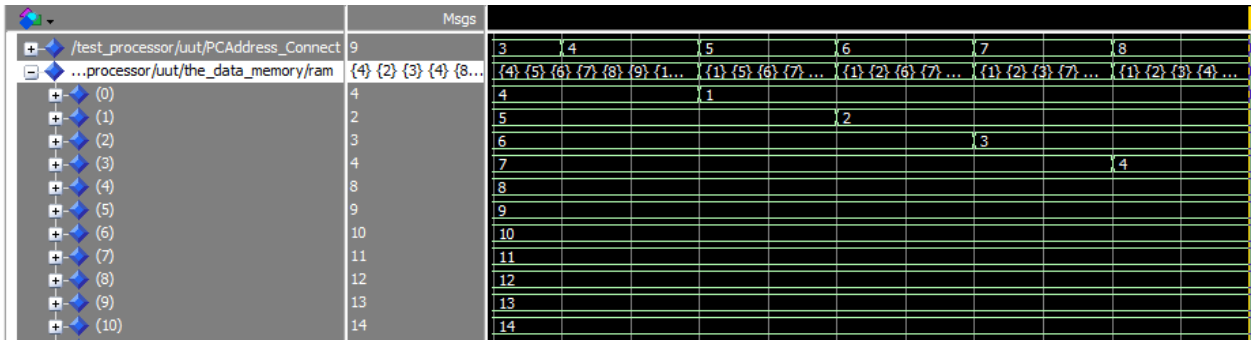
```
 InstrWord : reg generic map ( DWIDTH => 16)
     port map (
     CLK => CLK,
     RST => RESET,
     CE => EN_InstrWord,
     D => INSTR_OUT,
     Q => instr_word_out
     );

 TheOpcode : reg generic map ( DWIDTH => 4)
     port map (
     CLK => CLK,
     RST => RESET,
     CE => EN_OPCODE,
     D => instr_word_out(15 downto 12),
     Q => Instr_OP_S3
     );

TheImmediate : reg generic map ( DWIDTH => 16)
     port map (
     CLK => CLK,
     RST => RESET,
     CE => EN_Immediate,
     D => immediate_zero_extend_out,
     Q => immediate_out
     );

TheData : reg generic map ( DWIDTH => 16)
     port map (
     CLK => CLK,
     RST => RESET,
     CE => EN_DATA,
     D => DATA_OUT,
     Q => the_data_out
     );


 RegVal : reg generic map ( DWIDTH => 16)
      port map (
      CLK => CLK,
      RST => RESET,
      CE => EN_RegVal,
      D => reg_value_in,
      Q => reg_value_out
      );

  TheReg : reg generic map ( DWIDTH => 2)
      port map (
      CLK => CLK,
      RST => RESET,
      CE => EN_REG,
      D => instr_word_out(11 downto 10),
      Q => reg_out
      );

  ABCD_RegFile : ABCDRegFile port map (
      CLK => CLK,
      RESET => RESET,
      RD_REG => instr_word_out(11 downto 10),
      REG_OUT => reg_value_in,

      ABCD_WE => EN_ABCD,
      WR_REG => reg_out,
      REG_IN => alu_out
      );
```

After finishing designing my pipeline processor, I moved on to examine the instruction and the output to see if I had correctly implemented my code. From the data memory address, we can see that addresses 0-3 store value 4-7 respectively.

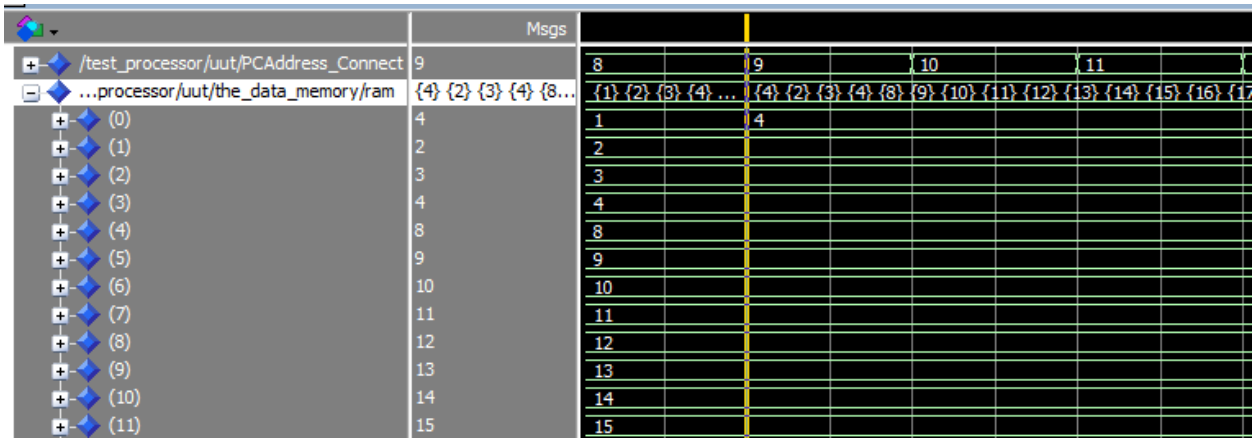
```
-------------------------------------------------
signal ram : rammemory := ("0000000000000100",   --  0: array[0]=4
                           "0000000000000101",   --  1: array[1]=5
                           "0000000000000110",   --  2: array[2]=6
                           "0000000000000111",   --  3: array[3]=7
                           "0000000000001000",   --  4: array[4]=8
                           "0000000000001001",   --  5: array[5]=9
                           "0000000000001010",   --  6: array[6]=10
                           "0000000000001011",   --  7: array[7]=11
```

In the instruction memory, the first 4 instructions were seti, set immediately, which set a value into a register that is immediately available without going to the memory.

```
signal ram : rammemory := ("0101000000000001",   -- 0:  seti A 1
                           "0101010000000010",   -- 1:  seti B 2
                           "0101100000000011",   -- 2:  seti C 3
                           "0101110000000100",   -- 3:  seti D 4
                           "0011000000000000",   -- 4:  store A 0
                           "0011010000000001",   -- 5:  store B 1
                           "0011100000000010",   -- 6:  store C 2
                           "0011110000000011",   -- 7:  store D 3
                           others => "0000000000000000");
```

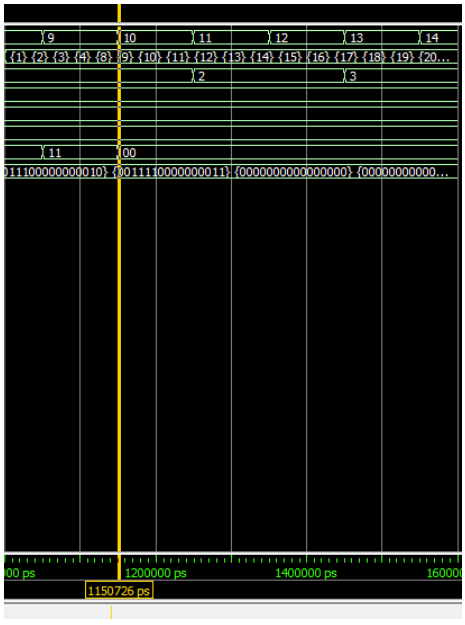


That's why from the above waveform we can see that the data ram changed from 4,5,6,7,8,9… into 1,2,3,4,8,9. After the first 4 seti instructions are executed, then the next 4 instructions, stored in addresses 0,1,2,3,4, are called.



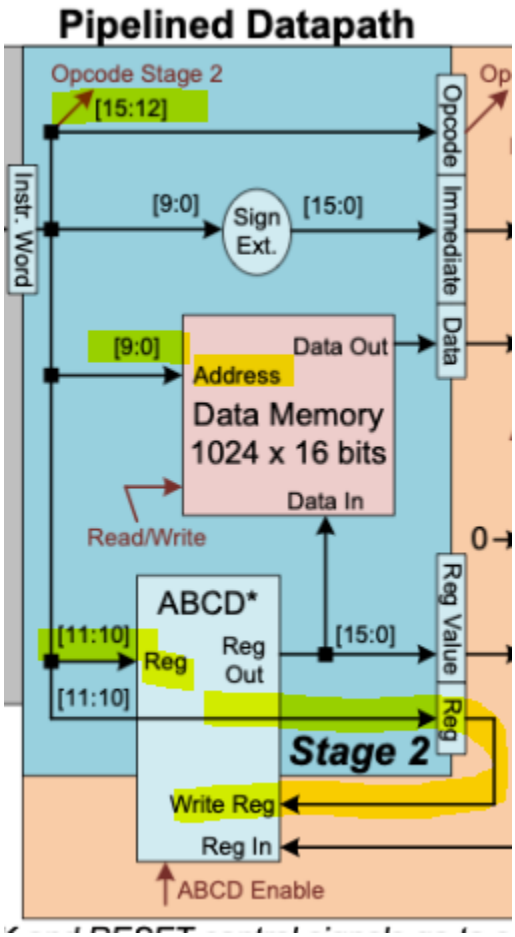
From this waveform, we can see that the data memory updated from 1,2,3,4,8,9 to 4,2,3,4,8,9.

When I kept executing the program, I noticed that after all instructions were executed, the a_out value kept incrementing every counter. I examined the code again to seek an explanation and came up with my personal thought but I'm not confidently saying that this is the correct explanation.



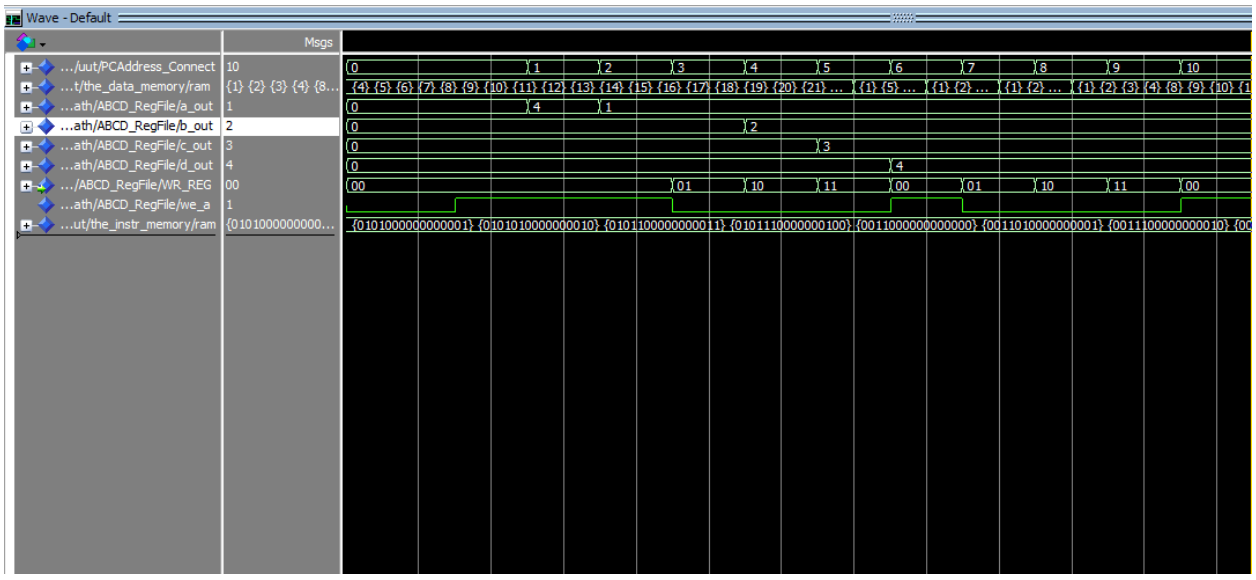
```
"0101110000000100",   -- 3:  seti D 4
"0011000000000000",   -- 4:  store A 0
"0011010000000001",   -- 5:  store B 1
"0011100000000010",   -- 6:  store C 2
"0011110000000011",   -- 7:  store D 3
others => "0000000000000000");
```

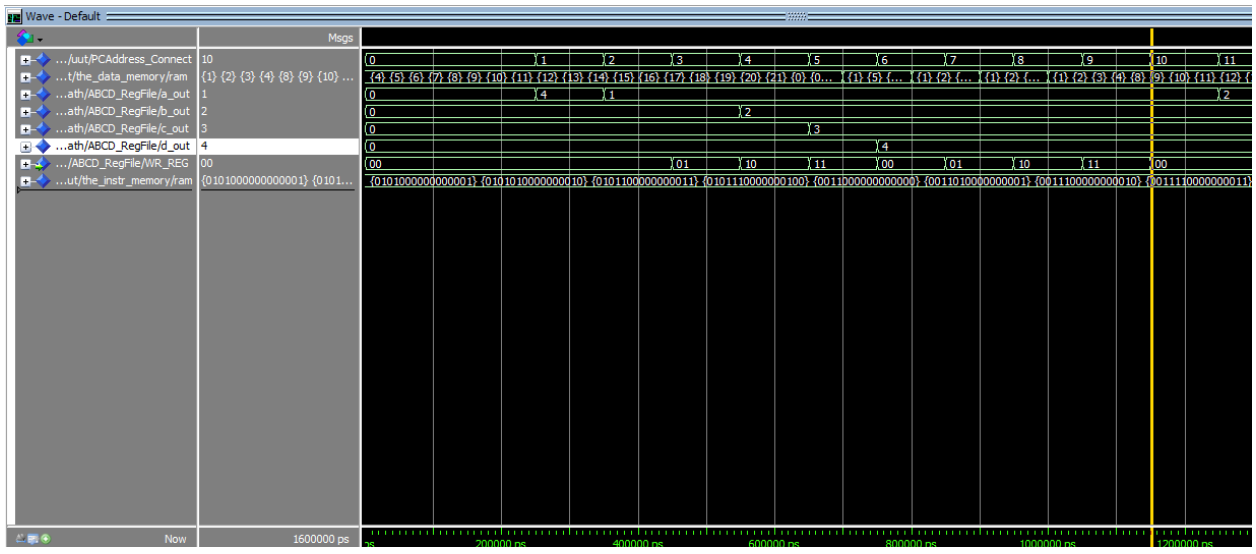
Since we have the statement “others => “0000000000000000”) after all instructions executed, then the program might run others case over again.



The First 4 bits [15:12] represent the opcode for adding, “0000”, and the next 2 bits [11:10] represent input from instr_word into ABCDRegFile, the input will go through Reg and return back to WR_Reg. WR_Reg will decide which register to write to and “00” represents we_a, which write to register A. So I viewed this instruction “0000000000000000” as adding or incrementing register A That’s why the value for a_out after we finished all assigned instructions

incremented, and it will keep incrementing by 1 every counter. This is my understanding of the process after examining the output of the program.





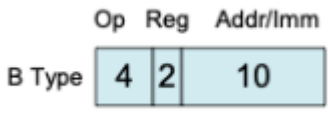
Discussion

The main differences in the datapaths between this new pipelined processor and the single-cycle processor from lab 3 are for this pipelined datapath, we divided it into 3 different stages and added more registers to act as transitions between each stage. By adding more registers into the datapath, we need to create the enable signal for each of them so that data can flow through each stage.

Before adding my stages and control signals, I had deleted the PC mux, Accumulator, and Zero checker registers for this pipelined chart because we were not using them for the datapath. The program counter input will directly come from the adder now instead of going into the 2x1 mux between the output of the adder and the output of Alu. I also removed the negative flag and zero flag signals that connected to the Zero checker and Accumulator.

Dividing the datapath into stages will help us to pipeline the instruction, we can start fetching the next instruction right after the previous instruction is passed through the next stage.

The chart

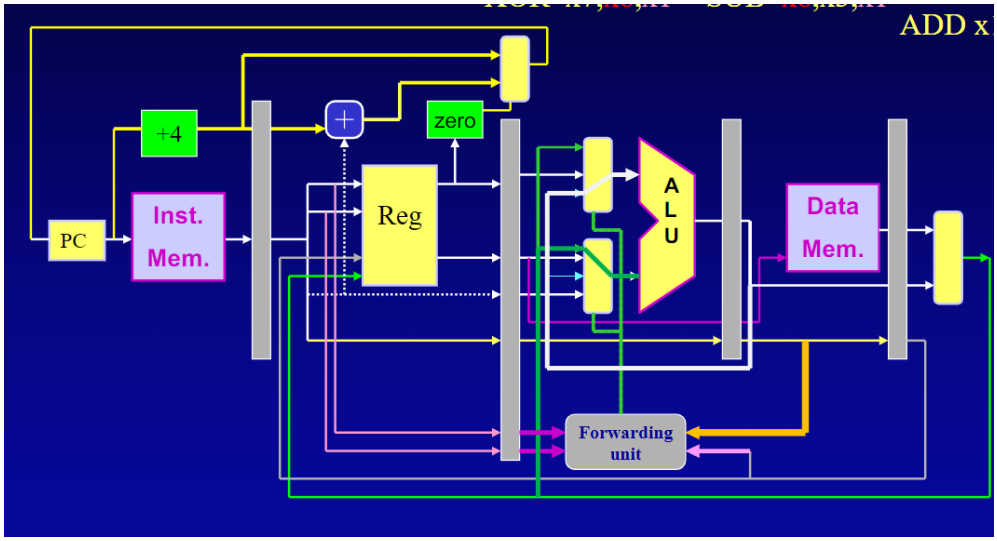


shows the function of the new Instruction Word register that we added between stages 1 and 2. This register will take input from the output of the instruction memory and divided it into the opcode, represented by the first 4 bits [15:12], the input of ABCD Register, [11:10], and the last 10 bits will flow into the data memory address, it also passed through the Immediate component after going into sign extension to extend from 10 bits to 16 bits address.

ABCD Register file replaces the Accumulator and acts as a decoder where it used the mux4 generic map-based. The output bits [11:10] of Instruction Word will go into the Reg

component and return back to the WR_Reg port to determine which register, A or B or C or D, will be used.

This processor, as it is currently designed, does not correctly handle data hazards yet. For example, if we have two or more consecutive instructions that used the same data address at the same time, the next instruction has to stall and wait until the previous instruction is finished write back the data into the register file. This hazard will cause a delay in our pipelining. Based on what we learned in lecture 10-Pipelining, I understand that to be able to handle pipeline hazards, we should create 2 units, either of them will help to resolve data hazards. One is the hazard detection unit, and the other one is a forwarding unit. The hazard detection unit can detect the hazard when it comes from PC and Instruction Word, it can stall the next instruction until the hazard is cleared. The forwarding unit can help pass the value from the register to ALU without waiting for it to get written back to the register file.



This is an example of a hazard detection and forwarding unit that we learned in class.

Conclusion

Since this is our last lab, it included almost everything that we had learned since the beginning of this class. From creating the datapath to creating new components, and registers and being able to connect them together as one big processor. Furthermore, the real-world processor is much bigger than the one we implemented in our course, so being able to know how to speed up, using fewer cycles, multicycles, and pipelines will help us improve the performance and save a lot of memory. If I was to redo this lab, I would try to add a hazard detection unit or forwarding unit to control hazards. Since I understand the concept of data forwarding but I couldn't yet figure out where should I create the unit whether in stage 2 or stage 3 due to time limitations.

References

Dr. Mike Turi. “09-Datapath-Control-The Processor Datapath and Control” Canvas. Spring 2022 EGCP 381

Dr. Mike Turi. “Lab 3: Single-Cycle Small16 Processor Datapath and Control” Canvas. Spring 2022 EGCP 381

Dr. Mike Turi. “week8-single-cyc-datapath-381.mp4” Canvas. Spring 2022 EGCP 381

Dr. Mike Turi. “10-Pipelining” Canvas. Spring 2022 EGCP 381

Dr. A.P Shanthi. “Handling Data Hazards” CS UMD. Retrieved from https://www.cs.umd.edu/~meesh/411/CA-online/chapter/handling-data-hazards/index.htm

l

Appendix

Control_s2.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.sm16_types.all;

-- control Entity Description
-- Adapted from Dr. Michael Crocker's Spring 2013 CSCE 385 Lab 4 at Pacific Lutheran
University
entity control_s2 is
    port( CLK   : in std_logic;
          RESET : in std_logic;
          START : in std_logic;

          WE     : out std_logic;
          --ALU_OP : out std_logic_vector(1 downto 0);
          --B_INV  : out std_logic;
          --CIN    : out std_logic;
          --A_SEL  : out std_logic;
          --B_SEL  : out std_logic;
          --PC_SEL : out std_logic;
          --EN_A   : out std_logic;
          EN_PC  : out std_logic;
```

```
            --EN_ABCD : out std_logic;
        --Z_FLAG   : in std_logic;
        --N_FLAG   : in std_logic;
        --INSTR_OP : in sm16_opcode
            EN_InstrWord : out std_logic;
            EN_OPCODE : out std_logic;
        EN_Immediate: out std_logic;
            EN_DATA : out std_logic;
            EN_RegVal : out std_logic;
            EN_REG : out std_logic;

            Instr_OP_S2 : in sm16_opcode

            );
end control_s2;

-- control Architecture Description
architecture behavorial of control_s2 is

    -- op codes
    constant op_add   : sm16_opcode := "0000";
    constant op_sub   : sm16_opcode := "0001";
```

```
  constant op_load  : sm16_opcode := "0010";
  constant op_store : sm16_opcode := "0011";
  constant op_addi  : sm16_opcode := "0100";
  constant op_seti  : sm16_opcode := "0101";
  constant op_jump  : sm16_opcode := "0110";
  constant op_jz    : sm16_opcode := "0111";

    -- register load control
    constant hold : std_logic := '0';
    constant load : std_logic := '1';

    -- data memory write enable control
    constant rd : std_logic := '0';
    constant wr : std_logic := '1';

       --pc select
    --constant from_plus1 : std_logic := '0';
    --constant from_alu   : std_logic := '1';
--MOVED TO S3
  -- control signal values
    -- alu operations
    --constant alu_nop : std_logic_vector(1 downto 0) := "00";
    --constant alu_and : std_logic_vector(1 downto 0) := "00";
```

```
--constant alu_or  : std_logic_vector(1 downto 0) := "01";
--constant alu_add : std_logic_vector(1 downto 0) := "10";
    --constant alu_mult : std_logic_vector(1 downto 0) := "11";

-- a select control
--constant a_0 : std_logic := '0';
--constant a_a : std_logic := '1';

-- b select control
--constant b_mem : std_logic := '0';
--constant b_imm : std_logic := '1';

    --NOT USED
-- pc select
--constant from_plus1 : std_logic := '0';
--constant from_alu   : std_logic := '1';



-- b invert control
--constant pos : std_logic := '0';
--constant inv : std_logic := '1';
```

```
    -- definitions of the states the control can be in
    type states is (stopped, running);  -- single cycle now, so only one running state
    signal state, next_state : states := stopped;

    -- internal write enable, ungated by the clock
    signal pre_we : std_logic;

begin

    -- write enable is gated when the clock is low
    WE <= pre_we and (not CLK);

    -- process to state register
    state_reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            state <= stopped;
        elsif( rising_edge(CLK) ) then
            state <= next_state;
        end if;
    end process state_reg;
```

```
-- ############################################## --

-- process to define next state transitions and output signals
next_state_and_output: process( state, START, Instr_OP_S2)--Z_FLAG, N_FLAG )
begin
   case state is
      -- Stopped is the stopped state; wait for start
      when stopped =>

         if( START /= '1' ) then

            -- issue nop

                EN_PC <= hold;    EN_InstrWord <= hold;
                EN_OPCODE <= hold; EN_Immediate <= hold;
                EN_DATA <= hold;  EN_RegVal <= hold;
              EN_REG <= hold;

         --    EN_A  <= hold;    EN_PC  <= hold;
               --0-- rd
         pre_we <= rd;     --PC_SEL <= from_plus1;
          --   B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_nop;
         --    A_SEL <= a_0;     B_SEL  <= b_mem;
```

```
      next_state <= stopped;
   else

            EN_PC <= hold;    EN_InstrWord <= hold;
            EN_OPCODE <= hold; EN_Immediate <= hold;
            EN_DATA <= hold;  EN_RegVal <= hold;
         EN_REG <= hold;

   --    EN_A  <= hold;    EN_PC  <= hold;
      pre_we <= rd;     --PC_SEL <= from_plus1;
    --   B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_and;
   --     A_SEL <= a_0;     B_SEL  <= b_mem;

      next_state <= running; -- go to fetch state
   end if;

-- In running state, each instruciton has its own control signals
when running =>

   if Instr_OP_S2 = op_add then
      -- A <- A + Mem
            EN_PC <= load;    EN_InstrWord <= load;
```

```
            EN_OPCODE <= load; EN_Immediate <= load;
            EN_DATA <= load;  EN_RegVal <= load;
         EN_REG <= load;



    --EN_A  <= load;
    pre_we <= rd;    --PC_SEL <= from_plus1;
    --B_INV <= pos;     --CIN    <= '0';     --ALU_OP <= alu_add;
    --A_SEL <= a_a;      B_SEL  <= b_mem;


    next_state <= running;


elsif Instr_OP_S2 = op_sub then
    -- A <- A - Mem


            EN_PC <= load;    EN_InstrWord <= load;
            EN_OPCODE <= load; EN_Immediate <= load;
            EN_DATA <= load;  EN_RegVal <= load;
         EN_REG <= load;


            --EN_A  <= load;
    pre_we <= rd;     --PC_SEL <= from_plus1;
    --B_INV <= inv;     CIN    <= '1';     ALU_OP <= alu_add;
```

```
--A_SEL <= a_a;     B_SEL  <= b_mem;


next_state <= running;


   --elsif instr_op = op_subi then
-- A <- A - Mem
--EN_A  <= load;    EN_PC  <= load;
--pre_we <= rd;     PC_SEL <= from_plus1;
--B_INV <= inv;     CIN    <= '1';     ALU_OP <= alu_add;
--A_SEL <= a_a;     B_SEL  <= b_imm;


next_state <= running;
   elsif Instr_OP_S2 = op_load then


EN_PC <= load;    EN_InstrWord <= load;
       EN_OPCODE <= load; EN_Immediate <= load;
       EN_DATA <= load;  EN_RegVal <= load;
   EN_REG <= load;


       --EN_A  <= load;
pre_we <= rd;     --PC_SEL <= from_plus1;
--B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_add;
--A_SEL <= a_0;     B_SEL  <= b_mem;
```

```
        next_state <= running;

    elsif Instr_OP_S2 = op_store then

        EN_PC <= load;    EN_InstrWord <= load;
        EN_OPCODE <= load; EN_Immediate <= load;
        EN_DATA <= load;  EN_RegVal <= load;
    EN_REG <= load;  pre_we <= wr;

--EN_A  <= load;
     --PC_SEL <= from_plus1;
--B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_nop;
--A_SEL <= a_a;     B_SEL  <= b_mem;

next_state <= running;

    elsif Instr_OP_S2 = op_addi then

        EN_PC <= load;    EN_InstrWord <= load;
        EN_OPCODE <= load; EN_Immediate <= load;
        EN_DATA <= load;  EN_RegVal <= load;
    EN_REG <= load;
```

```
        --EN_A  <= load;
    pre_we <= rd;     --PC_SEL <= from_plus1;
    --B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_add;
    --A_SEL <= a_a;     B_SEL  <= b_imm;

    next_state <= running;

elsif Instr_OP_S2 = op_seti then
    -- A <- 0 + Immediate
    EN_PC <= load;    EN_InstrWord <= load;
          EN_OPCODE <= load; EN_Immediate <= load;
          EN_DATA <= load;  EN_RegVal <= load;
       EN_REG <= load;

    --EN_A  <= load;
    pre_we <= rd;     --PC_SEL <= from_plus1;
    --B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_add;
    --A_SEL <= a_0;     B_SEL  <= b_imm;

    next_state <= running;
```

```
    else -- unknown opcode
        -- should never get here, but if it does, set PC<=0 and stop
            EN_PC <= load;    EN_InstrWord <= load;
            EN_OPCODE <= load; EN_Immediate <= load;
            EN_DATA <= load;  EN_RegVal <= load;
          EN_REG <= load;

        --EN_A  <= hold;
        pre_we <= rd;     --PC_SEL <= from_plus1;
        --B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_and;
        --A_SEL <= a_0;     B_SEL  <= b_mem;

        next_state <= stopped;

    end if;

when others => -- unknown state
        -- should never get here, but if it does, set PC<=0 and stop
            EN_PC <= load;    EN_InstrWord <= load;
            EN_OPCODE <= load; EN_Immediate <= load;
            EN_DATA <= load;  EN_RegVal <= load;
          EN_REG <= load;
```

```
            --EN_A  <= hold;
            pre_we <= rd;     --PC_SEL <= from_plus1;
            --B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_and;
            --A_SEL <= a_0;     B_SEL  <= b_mem;

            next_state <= stopped;
        end case;
    end process next_state_and_output;

end behavorial;
```

Control_s3.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.sm16_types.all;

entity control_s3 is
    port( CLK   : in std_logic;
          RESET : in std_logic;
          START : in std_logic;
```

```
        --WE     : out std_logic;
        ALU_OP : out std_logic_vector(1 downto 0);
        B_INV  : out std_logic;
        CIN    : out std_logic;
        A_SEL  : out std_logic;
        B_SEL  : out std_logic;


           EN_ABCD : out std_logic;


        Instr_OP_S3 : in sm16_opcode
           );
end control_s3;

architecture behavorial of control_s3 is

  -- register load control
       constant hold : std_logic := '0';
       constant load : std_logic := '1';

   -- control signal values
       -- alu operations
```

```
    constant alu_nop : std_logic_vector(1 downto 0) := "00";
    constant alu_and : std_logic_vector(1 downto 0) := "00";
    constant alu_or  : std_logic_vector(1 downto 0) := "01";
    constant alu_add : std_logic_vector(1 downto 0) := "10";
      constant alu_mult : std_logic_vector(1 downto 0) := "11";

-- a select control
    constant a_0 : std_logic := '0';
    constant a_a : std_logic := '1';

-- b select control
    constant b_mem : std_logic := '0';
    constant b_imm : std_logic := '1';

-- b invert control
    constant pos : std_logic := '0';
    constant inv : std_logic := '1';

-- op codes
  constant op_add   : sm16_opcode := "0000";
  constant op_sub   : sm16_opcode := "0001";
  constant op_load  : sm16_opcode := "0010";
  constant op_store : sm16_opcode := "0011";
```

```
    constant op_addi  : sm16_opcode := "0100";
    constant op_seti  : sm16_opcode := "0101";
    constant op_jump  : sm16_opcode := "0110";
    constant op_jz    : sm16_opcode := "0111";


    -- definitions of the states the control can be in
    type states is (stopped, running);  -- single cycle now, so only one running state
    signal state, next_state : states := stopped;



begin


    -- process to state register
    state_reg: process( CLK, RESET )
    begin
        if( RESET = '1' ) then
            state <= stopped;
        elsif( rising_edge(CLK) ) then
            state <= next_state;
        end if;
```

```
end process state_reg;

-- ################################################ --

-- process to define next state transitions and output signals
next_state_and_output: process( state, START, Instr_OP_S3)--Z_FLAG, N_FLAG )
begin
    case state is
        -- Stopped is the stopped state; wait for start
        when stopped =>

            if( START /= '1' ) then

                -- issue nop


                        B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_nop;
                A_SEL <= a_0;     B_SEL  <= b_mem;
                EN_ABCD  <= hold;
                next_state <= stopped;
            else
```

```
            B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_and;
        A_SEL <= a_0;     B_SEL  <= b_mem;
        EN_ABCD  <= hold;
        next_state <= running; -- go to fetch state
    end if;

-- In running state, each instruciton has its own control signals
when running =>

    if Instr_OP_S3 = op_add then
        -- A <- A + Mem



            B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_add;
        A_SEL <= a_a;      B_SEL  <= b_mem;
        EN_ABCD  <= load;


        next_state <= running;


    elsif Instr_OP_S3 = op_sub then
        -- A <- A - Mem
```

```
        B_INV <= inv;     CIN    <= '1';     ALU_OP <= alu_add;
A_SEL <= a_a;     B_SEL  <= b_mem;
EN_ABCD  <= load;

next_state <= running;

    --elsif instr_op = op_subi then
-- A <- A - Mem
--EN_A  <= load;    EN_PC  <= load;
--pre_we <= rd;     PC_SEL <= from_plus1;
--B_INV <= inv;     CIN    <= '1';     ALU_OP <= alu_add;
--A_SEL <= a_a;     B_SEL  <= b_imm;

next_state <= running;
    elsif Instr_OP_S3 = op_load then


B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_or;
A_SEL <= a_0;     B_SEL  <= b_mem;
EN_ABCD  <= load;

        next_state <= running;
```

```
    elsif Instr_OP_S3 = op_store then

   B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_or;
   A_SEL <= a_a;     B_SEL  <= b_mem;
   EN_ABCD  <= load;

   next_state <= running;

    elsif Instr_OP_S3 = op_addi then


   B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_add;
   A_SEL <= a_a;     B_SEL  <= b_imm;
   EN_ABCD  <= load;

   next_state <= running;

elsif Instr_OP_S3 = op_seti then
   -- A <- 0 + Immediate

   B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_add;
   A_SEL <= a_0;     B_SEL  <= b_imm;
```

```
        EN_ABCD  <= load;

        next_state <= running;



    else -- unknown opcode
        -- should never get here, but if it does, set PC<=0 and stop


        B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_add;
        A_SEL <= a_a;     B_SEL  <= b_imm;
        EN_ABCD  <= load;

        next_state <= stopped;

    end if;

when others => -- unknown state
        -- should never get here, but if it does, set PC<=0 and stop

        EN_ABCD  <= hold;
```

```
                B_INV <= pos;     CIN    <= '0';     ALU_OP <= alu_and;
                A_SEL <= a_0;     B_SEL  <= b_mem;


                next_state <= stopped;
        end case;
    end process next_state_and_output;


end behavorial;
```

Processor.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.sm16_types.all;


-- processor Entity Description
-- Adapted from Dr. Michael Crocker's Spring 2013 CSCE 385 Lab 4 at Pacific Lutheran
University
entity processor is
    port( CLK : in std_logic;
          RESET : in std_logic;
          START : in std_logic  -- signals to run the processor
          );
```

```
end processor;

-- processor Architecture Description
architecture structural of processor is

   -- declare all components and their ports
   component datapath is
      port( CLK   : in std_logic;
            RESET : in std_logic;

            -- I/O with Data Memory
            DATA_IN   : out sm16_data; --out from ABCD_Regout
            DATA_OUT  : in  sm16_data;
            DATA_ADDR : out sm16_address; --out from instr. word s1-s2

            -- I/O with Instruction Memory
            INSTR_OUT  : in  sm16_data; --input of instr. word s1-s2
            INSTR_ADDR : out sm16_address; --out from pc s1

            -- OpCode sent to the Control
            INSTR_OP   : out sm16_opcode; --(4bits) out from instr. word to opcode s2-s3 (15
downto 12)
```

```
-- Control Signals to the ALU
ALU_OP : in std_logic_vector(1 downto 0); --s3
B_INV  : in std_logic; --s3
CIN    : in std_logic; --s3

-- Control Signals from the Accumulator
--ZERO_FLAG : out std_logic; --might not be used
--NEG_FLAG  : out std_logic; --might not be used

-- ALU Multiplexer Select Signals
A_SEL  : in std_logic;
B_SEL  : in std_logic; -- input from immediate s2-s3
--PC_SEL : in std_logic;

-- Enable Signals for all registers
--EN_A  : in std_logic;
EN_PC : in std_logic;
   EN_ABCD : in std_logic;

   EN_InstrWord : in std_logic;
   EN_OPCODE : in std_logic;
EN_Immediate: in std_logic;
   EN_DATA : in std_logic;
```

```
        EN_RegVal : in std_logic;
        EN_REG : in std_logic;

        Instr_OP_S2 : out sm16_opcode;
        Instr_OP_S3 : out sm16_opcode);
end component;

component instr_memory is
    port( DIN : in sm16_data;
        ADDR: in sm16_address;
        DOUT: out sm16_data;
        WE: in std_logic);
end component;

component data_memory is
    port( DIN : in sm16_data;
        ADDR : in sm16_address;
        DOUT : out sm16_data;
        WE : in std_logic);
end component;
```

```
component control_s2 is
    port( CLK   : in std_logic;
      RESET : in std_logic;
      START : in std_logic;

      WE     : out std_logic;

      EN_PC  : out std_logic;


         EN_InstrWord : out std_logic;
         EN_OPCODE : out std_logic;
      EN_Immediate: out std_logic;
         EN_DATA : out std_logic;
         EN_RegVal : out std_logic;
         EN_REG : out std_logic;

         Instr_OP_S2 : in sm16_opcode

         );
end component;

component control_s3 is
```

```
        port( CLK   : in std_logic;
      RESET : in std_logic;
      START : in std_logic;

      --WE     : out std_logic;
      ALU_OP : out std_logic_vector(1 downto 0);
      B_INV  : out std_logic;
      CIN    : out std_logic;
      A_SEL  : out std_logic;
      B_SEL  : out std_logic;


         EN_ABCD : out std_logic;

      Instr_OP_S3 : in sm16_opcode
         );
end component;

 signal DataAddress_Connect, PCAddress_Connect : sm16_address;
 signal Data_IntoMem_Connect : sm16_data;
 signal DataOut_OutofMem_Connect, Instruction_Connect : sm16_data;
 signal ReadWrite_Connect : std_logic;
```

```
signal EN_ABCD_Connect : std_logic;

signal InstrOpCode_Connect : sm16_opcode;
signal ALUOp_Connect : std_logic_vector(1 downto 0);
signal Binv_Connect  : std_logic;
signal Cin_Connect   : std_logic;

signal ASel_Connect  : std_logic;
signal BSel_Connect  : std_logic;
--signal PCSel_Connect : std_logic;

--signal EnA_Connect  : std_logic;
signal EnPC_Connect : std_logic;

signal EN_InstrWord_Connect: std_logic;
signal EN_OPCODE_Connect: std_logic;
signal EN_Immediate_Connect: std_logic;
signal EN_DATA_Connect: std_logic;
signal EN_RegVal_Connect: std_logic;
signal EN_REG_Connect: std_logic;
signal Instr_OP_S2_Connect: sm16_opcode;
```

```
    signal Instr_OP_S3_Connect: sm16_opcode;

begin

    the_instr_memory: instr_memory port map (
        DIN => "0000000000000000",
        ADDR => PCAddress_Connect,
        DOUT => Instruction_Connect,
        WE => '0'  -- always read
        );

    the_data_memory: data_memory port map (
        DIN => Data_IntoMem_Connect,
        ADDR => DataAddress_Connect,
        DOUT => DataOut_OutofMem_Connect,
        WE => ReadWrite_Connect
        );

    the_datapath: datapath port map (
        CLK   => CLK,
        RESET => RESET,
        DATA_IN   => Data_IntoMem_Connect,
        DATA_OUT  => DataOut_OutofMem_Connect,
```

```
DATA_ADDR => DataAddress_Connect,
INSTR_OUT  => Instruction_Connect,
INSTR_ADDR => PCAddress_Connect,
INSTR_OP   => InstrOpCode_Connect,


ALU_OP => ALUOp_Connect,
B_INV  => Binv_Connect,
CIN    => Cin_Connect,

A_SEL  => ASel_Connect,
B_SEL  => BSel_Connect,


EN_PC => EnPC_Connect,
   EN_ABCD => EN_ABCD_Connect,
   EN_InstrWord => EN_InstrWord_Connect,
EN_OPCODE => EN_OPCODE_Connect,
   EN_Immediate => EN_Immediate_Connect,
EN_DATA => EN_DATA_Connect,
EN_RegVal => EN_RegVal_Connect,
EN_REG => EN_REG_Connect,
Instr_OP_S2 => Instr_OP_S2_Connect,
```

```
    Instr_OP_S3 => Instr_OP_S3_Connect
    );




the_control_s2: control_s2 port map (
    CLK   => CLK,
    RESET => RESET,
    START => START,
    WE    => ReadWrite_Connect,
       EN_PC => EnPC_Connect,


       EN_InstrWord => EN_InstrWord_Connect,
    EN_OPCODE => EN_OPCODE_Connect,
       EN_Immediate => EN_Immediate_Connect,
    EN_DATA => EN_DATA_Connect,
    EN_RegVal => EN_RegVal_Connect,
    EN_REG => EN_REG_Connect,
    Instr_OP_S2 => Instr_OP_S2_Connect
    );


the_control_s3: control_s3 port map (
    CLK   => CLK,
```

```
        RESET => RESET,
        START => START,
        --WE     => ReadWrite_Connect,
        ALU_OP => ALUOp_Connect,
        B_INV  => Binv_Connect,
        CIN    => Cin_Connect,
        A_SEL  => ASel_Connect,
        B_SEL  => BSel_Connect,
           EN_ABCD => EN_ABCD_Connect,
           Instr_OP_S3 => Instr_OP_S3_Connect
        );
end structural;
```

Datapath.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.sm16_types.all;

-- datapath Entity Description
-- Adapted from Dr. Michael Crocker's Spring 2013 CSCE 385 Lab 4 at Pacific Lutheran
University
entity datapath is
    port( CLK   : in std_logic;
```

```
RESET : in std_logic;

-- I/O with Data Memory
DATA_IN   : out sm16_data;
DATA_OUT  : in  sm16_data;
DATA_ADDR : out sm16_address;

-- I/O with Instruction Memory
INSTR_OUT  : in  sm16_data;
INSTR_ADDR : out sm16_address;

-- OpCode sent to the Control
INSTR_OP   : out sm16_opcode;

-- Control Signals to the ALU
ALU_OP : in std_logic_vector(1 downto 0);
B_INV  : in std_logic;
CIN    : in std_logic;

-- Control Signals from the Accumulator
--ZERO_FLAG : out std_logic;
--NEG_FLAG  : out std_logic;
```

```
        -- ALU Multiplexer Select Signals
        A_SEL  : in std_logic;
        B_SEL  : in std_logic;
        --PC_SEL : in std_logic;

        -- Enable Signals for all registers
        --EN_A  : in std_logic;
        EN_PC : in std_logic;
           EN_ABCD : in std_logic;
           EN_InstrWord : in std_logic;
           EN_OPCODE : in std_logic;
        EN_Immediate: in std_logic;
           EN_DATA : in std_logic;
           EN_RegVal : in std_logic;
           EN_REG : in std_logic;

           Instr_OP_S2 : out sm16_opcode;
           Instr_OP_S3 : out sm16_opcode
           );
end datapath;

-- datapath Architecture Description
```

```
architecture structural of datapath is

    -- declare all components and their ports
    component reg is
        generic( DWIDTH : integer := 8 );
        port( CLK : in std_logic;
              RST : in std_logic;
              CE : in std_logic;

              D : in std_logic_vector( DWIDTH-1 downto 0 );
              Q : out std_logic_vector( DWIDTH-1 downto 0 ) );
    end component;

    component alu is
        port( A : in sm16_data;
              B : in sm16_data;
              OP : in std_logic_vector(1 downto 0);
              D : out sm16_data;
              CIN : in std_logic;
              B_INV : in std_logic);
    end component;

    component adder is
```

```
    port( A : in sm16_address;
          B : in sm16_address;
          D : out sm16_address);
end component;

component mux2 is
    generic( DWIDTH : integer := 16 );
    port( IN0 : in std_logic_vector( DWIDTH-1 downto 0 );
          IN1 : in std_logic_vector( DWIDTH-1 downto 0 );
          SEL : in std_logic;
          DOUT : out std_logic_vector( DWIDTH-1 downto 0 ) );
end component;

component ABCDRegFile is
        port( CLK: in std_logic;
              RESET: in std_logic;

        -- ABCD*
              RD_REG : in std_logic_vector(1 downto 0);  -- Which register to read and output
         REG_OUT : out sm16_data;

         ABCD_WE : in std_logic; -- Write enable signal
         WR_REG : in std_logic_vector(1 downto 0);  -- Which register to write to
```

```
            REG_IN : in sm16_data);
    end component;

    component zero_extend is
        port( A : in sm16_address;
              Z : out sm16_data);
    end component;

    --component zero_checker is
        --port( A : in sm16_data;
              --Z : out std_logic);
    --end component;

    signal zero_16 : sm16_data := "0000000000000000";
    signal alu_a, alu_b, alu_out : sm16_data;
    signal pc_out, pc_in : sm16_address;
    signal a_out, immediate_zero_extend_out : sm16_data;
    signal adder_out : sm16_data;
    signal instr_word_out, immediate_out, the_data_out, reg_value_out, reg_value_in :
sm16_data;
    signal reg_out: std_logic_vector(1 downto 0);
begin
```

```
InstrWord : reg generic map ( DWIDTH => 16)
        port map (
        CLK => CLK,
        RST => RESET,
        CE => EN_InstrWord,
        D => INSTR_OUT,
        Q => instr_word_out
        );

TheOpcode : reg generic map ( DWIDTH => 4)
        port map (
        CLK => CLK,
        RST => RESET,
        CE => EN_OPCODE,
        D => instr_word_out(15 downto 12),
        Q => Instr_OP_S3
        );

TheImmediate : reg generic map ( DWIDTH => 16)
        port map (
        CLK => CLK,
        RST => RESET,
        CE => EN_Immediate,
```

```
        D => immediate_zero_extend_out,
        Q => immediate_out
        );

TheData : reg generic map ( DWIDTH => 16)
        port map (
        CLK => CLK,
        RST => RESET,
        CE => EN_DATA,
        D => DATA_OUT,
        Q => the_data_out
        );

RegVal : reg generic map ( DWIDTH => 16)
        port map (
        CLK => CLK,
        RST => RESET,
        CE => EN_RegVal,
        D => reg_value_in,
        Q => reg_value_out
        );

 TheReg : reg generic map ( DWIDTH => 2)
```

```
    port map (
    CLK => CLK,
       RST => RESET,
       CE => EN_REG,
       D => instr_word_out(11 downto 10),
       Q => reg_out
       );

ABCD_RegFile : ABCDRegFile port map (
    CLK => CLK,
       RESET => RESET,
       RD_REG => instr_word_out(11 downto 10),
    REG_OUT => reg_value_in,

    ABCD_WE => EN_ABCD,
    WR_REG => reg_out,
    REG_IN => alu_out
       );

TheAlu: alu port map (
    A     => alu_a,
    B     => alu_b,
    OP    => ALU_OP,
```

```
    D     => alu_out,
    CIN   => CIN,
    B_INV => B_INV
    );

PCadder: adder port map (
        A => pc_out,
        B => "0000000001",
        D => adder_out(9 downto 0)
        );

 Amux: mux2 generic map ( DWIDTH => 16 )
    port map (
    IN0  => zero_16,  -- 00
    IN1  => reg_value_out,  -- 01
    SEL  => A_SEL,
    DOUT => alu_a
    );

 Bmux: mux2 generic map ( DWIDTH => 16 )
    port map (
    IN0  => the_data_out,  -- 00
    IN1  => immediate_out,  -- 01
```

```
    SEL  => B_SEL,
    DOUT => alu_b
    );

--PCmux: mux2 generic map ( DWIDTH => 10 )
    --port map (
    --IN0  => adder_out(9 downto 0),  -- 00
    --IN1  => alu_out(9 downto 0),  -- 01
    --SEL  => PC_SEL,
    --DOUT => pc_in
    --);

ProgramCounter: reg generic map ( DWIDTH => 10 )
    port map (
    CLK => CLK,
    RST => RESET,
    CE  => EN_PC,
    D   => adder_out(9 downto 0),
    Q   => pc_out
    );

--Accumulator: reg generic map ( DWIDTH => 16 )
    --port map (
```

```
   --CLK => CLK,
   -- RST => RESET,
   -- CE  => EN_A,
   -- D   => alu_out,
   -- Q   => a_out
  --  );

ImmediateZeroExt: zero_extend port map (
    A => instr_word_out(9 downto 0),
    Z => immediate_zero_extend_out
    );

--ZeroCheck: zero_checker port map (
    --A => a_out,
    --Z => ZERO_FLAG
    --);

--NEG_FLAG <= alu_out(15);

DATA_IN   <= reg_value_in;
DATA_ADDR <= instr_word_out(9 downto 0);
```

```
    Instr_OP_S2 <= instr_word_out(15 downto 12);



    INSTR_ADDR <= pc_out(9 downto 0);

end structural;
```