Amirkabir University of Technology

Norooz Project

SAYEH Basic Computer

Farzan Dehbashi
Parham Alvani

supervised by
Dr. Saeid Shiri Gheydari

March 19, 2017

Contents

1 Purpose

Design and implementation of a small modular processor, called SAYEH (Simple Architecture, Yet Enough Hardware) which contains the following major components:

- **Controller**
- **Datapath**

Functionality of the processor: This CPU exploits a 16-bit data-bus and also a 16-bit address-bus. Instructions used in this processor has 8 or 16-bit width. Short instructions (8-bit ones) contain shadow instructions, which effectively pack 2 such instructions (8-bit) into a single 16-bit word. Figure 1 shows SAYEH's interface (it's the overview of the whole module):

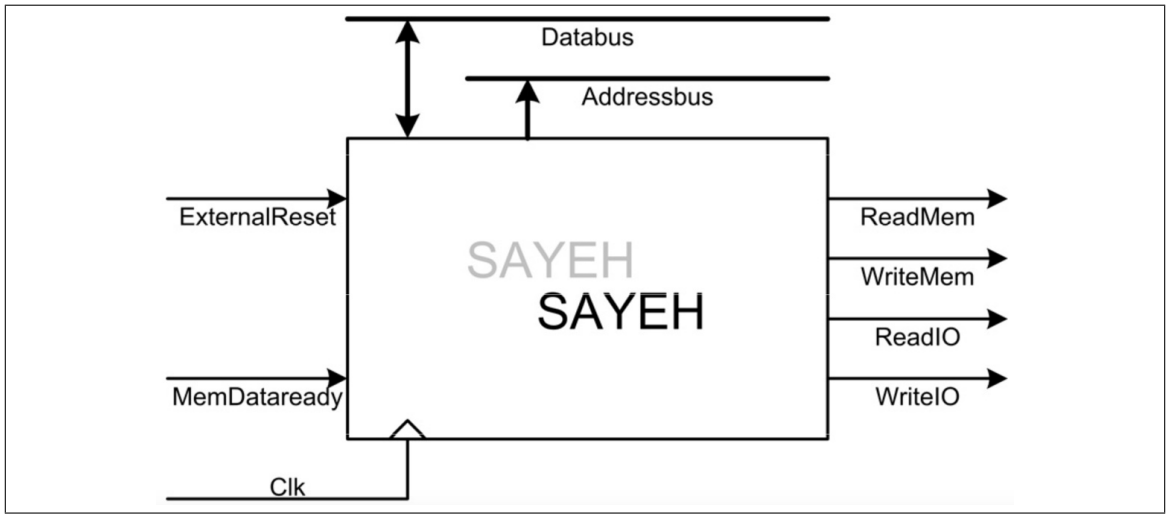

Figure 1: Sayeh Interface

2 CPU Components

2.1 Registerfile

Registerfile is a set of general purpose registers used in many cases (more information about this component and it's process of implementation could be found here) ,SAYEH uses it's *Registerfile* in arithmetic and logical operations, also addressing modes of the processor take advantage of this structure, by means of using register file's output in addressing calculations. therefore addressing component of SAYEH has been simplified.

- **Registerfile:** includes 64 registers each of them has **16-bit width**, 4 of which are called R0, R1, R2, R3 (note that these four register are not constantly located! Think about addressing them by index of 10,11,12,13 now and changing them to 2, 3, 4, 5 in a moment, although

the location is not constant but they are placed all next to each other, means 2,4,6,7 is not allowed).

- **R0, R1, R2, R3:** as described in register file section, **(16-bit)**.

2.2 Other Registers

Alongside *Registerfile* and R0, R1, R2, R3 as a part of that, these registers are also used:

- **Window Pointer(WP):** In order to point to R0 as the base of R0, R1, R2 and R3, Window Pointer is used, as we can specify a stuf out of 64 ones by 6 bits this register has 6-bit width.
- **Program Counter(PC):**program counter, (16-bit).
- **Instruction Register(IR):**Instruction Register, which has 16-bit width and would be loaded by a single 16-bit instruction or by two 8-bit instructions, (16-bit).
- **Zero Flag(Z):**becomes one when the ALU's output is zero, (1-bit).
- **Carry Flag(C):** becomes one when the ALU's output has got carry digit, (1-bit).

2.3 ALU (Arithmetic Logic Unit) Components

The ALU itself contains these components, each of them is capable of doing the named operation on 16-bit input(s), selecting the desired operation is done by the OPCODE of the instruction.

- **AND Component:** This component will perform AND operation on Rs (Source Register) and Rd (Destination Register), obviously the result would be another 16-bit vector and should be stored in Rd(Destination Register).
- **OR Component:** This component will perform OR operation on Rs (Source Register) and Rd (Destination Register), obviously the result would be another 16-bit vector and should be stored in destination register.
- **Shift-Right Component(Optional):** This component will perform Shift to Right operation on Rs (Source Register) obviously the result would be another 16-bit vector and should be stored in Rd (destination register).
- **Shift-Left Component:** This component will perform Shift to Left operation on Rs (Source Register) obviously the result would be another 16-bit vector and should be stored in Rd (destination register).
- **Comparison Component:** This component will compare Rs and Rd (if equal then zero flag must become one and if Rd is less than Rs then Carry flag (C) would become one)
- **Addition Component:** This component will perform Addition between Rs and Rd and Carry flag (C) and will store the result in Rd(Destination Register).
- **Subtraction Component:** This component will perform Subtraction by means of Rd =Rd-Rs-C.

- **Multiplication Component(Optional):** This component will perform Multiplication by means of Rd =Rd*Rs. note that as the multiplication's result of two 8-bit operands would be a 16-bit one so in this processor we will multiply right 8-bits of Rd by right 8-bits of Rs and 16 bit result would be stored in Rd.
- **Division Component(Optional):** This component will perform Division by means of Rd =Rs/Rd note that just right 8-bits (Least Significant Bits) of Rd would be used in this action.
- **Square Root Component(Optional):** This component will perform Rd= square root(Rs).
- **Random Generator Component(Optional):** Generates a random number between 0 to 64000.
- **two's Complement Component(Optional):** Performs two's complement operation on Rs and stores it in Rd(Destination Register).
- **XOR Component(Optional):** Just like AND :)
- **Trigonometry Component (sin, cos, tan, cot) by CORDIC IP core(Optional):** this component's extra mark is much more than the other ALU components, CORDIC is an IP core used for VHDL (like usual libraries in java,C,...) it's free and all documents could be found on the Internet.

2.4 Other Components

There is a list of other components needed for the processor to work well and these are not embedded into ALU component, such as:

- **No Operation:** When this instruction is executed CPU would do nothing for one clock cycle.
- **Halt:** By executing this instruction fetching stops for one clock period and the previous instruction which has been fetched remains as the last fetched item.
- **Set Zero Flag:** This instruction will set zero flag to 1.
- **Clear Zero Flag:** When this instruction has been executed zero flag would become 0.
- **Set Carry Flag:** This instruction will set carry flag to 1.
- **Clear Carry Flag:** When this instruction has been executed carry flag would become 0.
- **Clear Window Pointer:** When this instruction has been executed window pointer would become 000000.
- **Move Register:** This operation will move the value stored in Rs to Rd.
- **Load Addressed:** By this instruction you can load the value stored Rs'th row of memory to Rd.
- **Store Addressed:** By this instruction you can store Rs to Rd'th row of memory.

- **Port Manager (Optional):** This component is about to manage input/output ports of SAYEH, SAYEH has 64 ports(named as *P0... to P63*) these should be implemented by you. Imagine 64 ports that could be written by the processor and also read by it. these operations would be done by executing instructions like:Input Port and Output to Port (as mentioned in the Table1).Test case of this section would be reading form a desired port and storing it to Rd or reading from Rs and writing it in a desired port.
- **Move Immediate Low:** By this instruction 8 bits of *Immediate* operand would be copied to Rd's left 8 bits (8 least significant bits).
- **Move Immediate High:** Exactly like Move *Immediate* Low, but *Immediate* would be stored in most significant bits of Rd.
- **Save PC:** This stores PC to Rd.
- **Jump Addressed:** As shown in Table1.
- **Jump Relative:** As shown in Table1.
- **Branch if Zero:** As shown in Table1.
- **Branch if Carry:** As shown in Table1.
- **Add Win pointer:** As shown in Table1.

3 SAYEH Instructions

The general format of 8-bit and 16-bit instructions of SAYEH is shown in figure 2. 16-bit instructions contains *Immediate* field opposed to 8-bit instructions. The OPCODE field is a 4-bit code that specifies the type of the instruction. The *Left* and *Right* is used to specify the destination of the operation and *Right* for the source of it (source and destination are one of R0 to R3, so within 2 bits we can clarify which is the one we desire). The *Immediate* field is used for immediate data if instruction type is 16-bit one, and used for the second 8-bit instruction elsewhere.

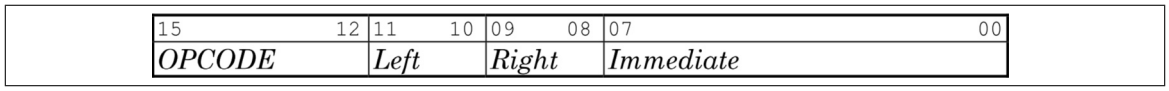

Figure 2: SAYEH Instructions Format

Our processor (SAYEH) has total of 35 instructions some of which are optional. Instructions within 16-bit length are the ones contains *Immediate* field and others don't. Instructions that use destination and source fields (shown as D and S in the table of instructions set) have an OPCODE limited to 4 bits. Instructions that don't require specification of source and destination registers use these fields as OPCODE extensions. Finally the overview of SAYEH's instruction set would be as shown in table1 (note that some of optional parts hasn't been mentioned in this table and Mnemonic and bit vector representation of them should be designed by you).

In the instruction set, addressed locations in the memory are indicated by enclosing the address in a set of parenthesis (instructions like load addressed or store addressed). For these instructions, the processor issues ReadMem or WriteMem signals to the memory. When input and output

instructions (input, output) are executed, SAYEH issues ReadIO or WriteIO signals to its IO devices(refer to Port Manager component).

Instruction Mnemonic and Definition		**Bits 15:0**	**RTL notation: *comments or condition***
nop	No operation	0000-00-00	No operation
hlt	Halt	0000-00-01	Halt, fetching stops
szf	Set zero flag	0000-00-10	Z <= '1'
czf	Clr zero flag	0000-00-11	Z <= '0'
scf	Set carry flag	0000-01-00	C <= '1'
ccf	Clr carry flag	0000-01-01	C <= '0'
cwp	Clr Window pointer	0000-01-10	WP <= "000"
mvr	Move Register	0001-D-S	R_D <= R_S
lda	Load Addressed	0010-D-S	R_D <= (R_S)
sta	Store Addressed	0011-D-S	(R_D) <= R_S
inp	Input from port	0100-D-S	In from R_S write to R_D
oup	Output to port	0101-D-S	Out to port R_D from R_S
and	AND Registers	0110-D-S	R_D <= R_D & R_S
orr	OR Registers	0111-D-S	R_D <= R_D | R_S
not	NOT Register	1000-D-S	R_D <= ~R_S
shl	Shift Left	1001-D-S	R_D <= sla R_S
shr	Shift Right	1010-D-S	R_D <= sra R_S
add	Add Registers	1011-D-S	R_D <= R_D + R_S + C
sub	Subtract Registers	1100-D-S	R_D <= R_D - R_S – C
mul	Multiply Registers	1101-D-S	R_D <= R_D * R_S *:8-bit multiplication*
cmp	Compare	1110-D-S	R_D, R_S *(if equal:Z=1; if R_D<R_S: C=1)*
mil	Move Immd Low	1111-D-00-I	R_{DL} <= {8'bZ, I}
mih	Move Immd High	1111-D-01-I	R_{DH} <= {I, 8'bZ }
spc	Save PC	1111-D-10-I	R_D <= PC + I
jpa	Jump Addressed	1111-D-11-I	PC <= R_D + I
jpr	Jump Relative	0000-01-11-I	PC <= PC + I
brz	Branch if Zero	0000-10-00-I	PC <= PC + I *:if Z is 1*
brc	Branch if Carry	0000-10-01-I	PC <= PC + I *:if C is 1*
awp	Add Win pntr	0000-10-10-I	WP <= WP + I

Table 1: Instruction Set of SAYEH

4 Datapath

The datapath of SAYEH is shown in Figure 3. The main components of this machine are the followings:

- **PC (Program Counter)**
- **Address Logic**
- **IR (Instruction Register)**
- **WP (Window Pointer)**
- **Register File**
- **ALU(Arithmetic Logic Unit)**
- **Flags**

As shown in figure 3 these components are either hardwired or connected through three state buses. Component inputs with multiple sources, such as the right hand side input of ALU, use three-state buses. Three-state buses in this structure are Databus and OpndBus . In this figure, signals that are in italic are control signals issued by the controller. These signals control register clocking, logic unit operations and placement of data in buses.

4.1 Datapath Components

Figure 4 shows the hierarchical structure of SAYEH components. The processor has a Datapath and a Controller. Datapath components are Addressing Unit, IR Module, WP, Register File, Arithmetic Logic Unit (ALU), and the Flags register. The Addressing Unit is further partitioned into the PC and Address Logic.

The Address Logic is a combinational circuit that is capable of adding its inputs to generate a 16-bit output which represents the address of the row we are about to fetch from memory. Register File is a two-port memory and a file of 64 16-bit registers. The Window Pointer is a 6-bit register that is used as the base of the Register File. Specific registers for read and write (R0, R1, R2 or R3) in the Register File are selected by its 4-bit input bus coming from the Instruction Register. Two bits select the source and another two selects the destination (means all of our operands, except *Immediate* ones are one of R0, R1, R2 or R3 selected by the mentioned signal).

When the Window Pointer add is enabled, it adds its 6-bit input to its current data. The Flags Register is a 2-bit register that saves the flag outputs of the Arithmetic Unit . The Flags Register is a 2bit register that saves the flag outputs of the Arithmetic Unit. The Arithmetic Unit is a 16-bit arithmetic and logic unit that has logical, shift, add and compare operations and ... (as discussed in it's own section). A 9-bit input selects one of the nine functions of the ALU. This code is provided by the SAYEH's controller component.

Controller Component of SAYEH has eleven states for various reset, fetch, decode, execute and halt operations. Signals generated by the controller control logic unit operations and register clicking in the data-path.

SAYEH sequential data components and its controller are triggered on the rising-edge of the main system clock. Control signals remain active after one rising edge through the next.This duration allows for propagation of signals through the buses and logic units in the data-path.

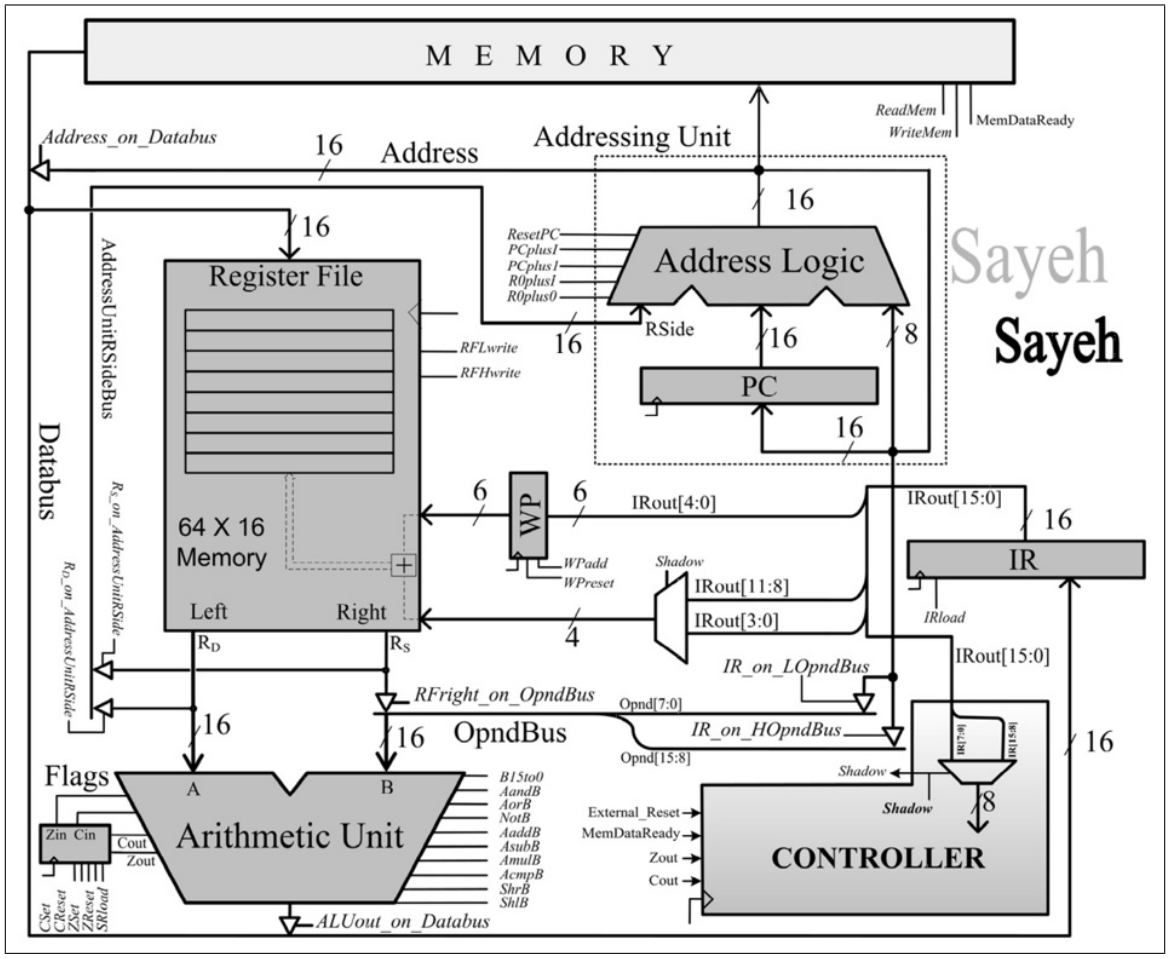

Figure 3: SAYEH Datapart

5 SAYEH VHDL Description

SAYEH should be described according to the hierarchical structure of Figure4. Data components should be described separately, and then wired to each other from the datapath component. Controller is described in a single VHDL module. In the complete SAYEH description, the datapath and controller are wired together.

5.1 Data Components

Combinational and sequential SAYEH data components are partially described here. The combinational ones are like the ALU that performs arithmetic and logical operations. The function of such units is controlled by the controller. The sequential components are clocked with the negative edge of the main CPU clock. These components have functionalities like loading and resetting and are controlled by the controller.

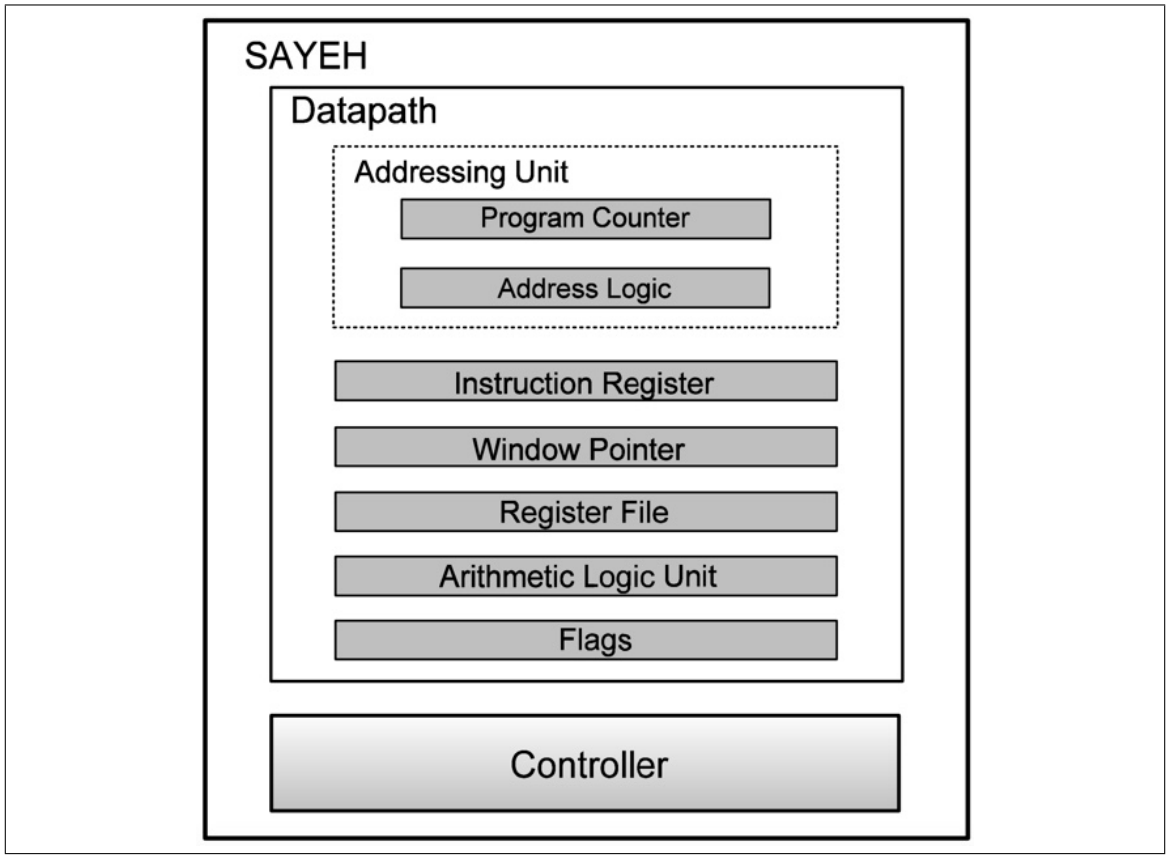

Figure 4: SAYEH Hierarchical Structure

Addressing Unit

The Addressing Unit of figure 5 consists of the Program Counter and Address Logic. The Program Counter is a simple register with enabling and resetting mechanisms, while the Address Logic is a small arithmetic unit that performs adding and incrementing for calculating PC or memory addresses.

This unit has a 16-bit input coming from the Register File, an 8-bit input from the Instruction Register, and a 16-bit address output. Control signals of the Addressing Unit are *ResetPC, PCplusI, PCplus1, RplusI, Rplus0,* and *PCenable*. These control signals select what goes on the output of this unit. Shown in Figure 6 is the VHDL code of the Program Counter. The Address Logic of Figure7 uses control signal inputs of the Addressing Unit to generate input data to the Program Counter via the PCout of Figure 5. Constants defined and used here correspond to one-hot control signals from the controller.

```vhdl
ENTITY AddressUnit IS
    PORT (
        Rside : IN std_logic_vector (15 DOWNTO 0);
        Iside : IN std_logic_vector (7 DOWNTO 0);
        Address : OUT std_logic_vector (15 DOWNTO 0);
        clk, ResetPC, PCplusI, PCplus1 : IN std_logic;
        RplusI, Rplus0, EnablePC : IN std_logic
    );
END AddressUnit;

ARCHITECTURE dataflow OF AddressUnit IS
    COMPONENT pc . . . END COMPONENT;
    COMPONENT al . . . END COMPONENT;

    SIGNAL pcout : std_logic_vector (15 DOWNTO 0);
    SIGNAL AddressSignal : std_logic_vector (15 DOWNTO 0);
BEGIN
    Address <= AddressSignal;
    l1 : pc PORT MAP (EnablePC, AddressSignal, clk, pcout);
    l2 al PORT MAP
        (pcout, Rside, Iside, AddressSignal,
        ResetPC, PCplusI, PCplus1, RplusI, Rplus0);
END dataflow;
```

Figure 5: *Addressing Unit* VHDL code

```vhdl
ENTITY ProgramCounter IS
    PORT (
        EnablePC : IN std_logic;
        input: IN std_logic_vector (15 DOWNTO 0);
        clk : IN std_logic;
        output: OUT std_logic_vector (15 DOWNTO 0);
    );
END ProgramCounter;

ARCHITECTURE dataflow OF ProgramCounter IS BEGIN
    PROCESS (clk) BEGIN
        IF (clk = '1') THEN
            IF (EnablePC = '1') THEN
                output <= input;
            END IF;
        END IF;
    END PROCESS;
END dataflow;
```

Figure 6: *Program Counter* VHDL code

```vhdl
ENTITY AddressLogic IS
    PORT (
        PCside, Rside : IN std_logic_vector (15 DOWNTO 0);
        Iside : IN std_logic_vector (7 DOWNTO 0);
        ALout : OUT std_logic_vector (15 DOWNTO 0);
        ResetPC, PCplusI, PCplus1, RplusI, Rplus0 : IN std_logic
    );
END AddressLogic;

ARCHITECTURE dataflow of AddressLogic IS
    CONSTANT one   : std_logic_vector (4 DOWNTO 0)
                   := "10000"
    CONSTANT two   : std_logic_vector (4 DOWNTO 0)
                   := "01000"
    CONSTANT three : std_logic_vector (4 DOWNTO 0)
                   := "00100"
    CONSTANT four  : std_logic_vector (4 DOWNTO 0)
                   := "00010"
    CONSTANT five  : std_logic_vector (4 DOWNTO 0)
                   := "00001"
BEGIN
    PROCESS (PCside, Rside, Iside, ResetPC,
            PCplusI, PCplus1, RplusI, Rplus0)
        VARIABLE temp : std_logic_vector (4 DOWNTO 0);
    BEGIN
        temp := (ResetPC & PCplusI & PCplus1 & RplusI & Rplus0);
        CASE temp IS
            WHEN one => ALout <= (OTHERS => '0');
            WHEN two => ALout <= PCside + Iside;
            WHEN three => ALout <= PCside + 1;
            WHEN four => ALout <= Rside + Iside;
            WHEN five => ALout <= Rside;
            WHEN OTHERS => ALout <= PCside;
        END CASE;
    END PROCESS;
END dataflow;
```

Figure 7: *Address Logic* VHDL code

Memory

You need to use memory module as a storage. we design this unit for you and you can instantiate this module in your datapath. for using this module you just put it's entity as component and pass arrays with correct size to it, it works :)

```vhdl
entity memory is
        port (address : in std_logic_vector;
                data_in : in std_logic_vector;
                data_out : out std_logic_vector;
                clk, rwbar : in std_logic);
end entity memory;
```

Figure 8: *Memory* VHDL entity code

Source codes are available here.

6 Important Notes

1. All your project should be implemented by VHDL not Verilog or...
2. A part of your score would be devoted to the report you deliver within your projects code. In this report you would explain the whole job and anything special you have done. (note that it shouldn't be shorter than this document and should be typed!)
3. Controller should be designed by an Finite State Machine (FSM) and this FSM should be mentioned in your project report.
4. Name of your modules, OPCODEs,... should be exactly the same as the ones mentioned in this document.
5. In addition to bonus sections mentioned beforehand, implementation of this project on *Altera DE2* FPGA boards will result in extra mark as well.
6. You can simulate your code with softwares like: Modelsim, Altera Quartus, GHDL, Xilinx ISE, Active HDL, Xilinx Vivado Design Suite,... (note that Modelsim and Quartus has got free versions and GHDL is totally free so it's better to obey *Copyright*) but the TA's laptop is only equipped with Modelsim and in case of delivery you should be able of running your code in that environment.
7. All questions would be answered by the course's email: computerarchitecture95@gmail.com.
8. This project would be followed by your Final Project, so try to do it in the best manner possible, otherwise your Final Project will suffer from a bad SAYEH and this will result in wasting a lot of time debugging this project or redesigning it's components.
9. Place all your modules and report into a .zip file named as "FirstName LastName StudentID" before upload. if any other module is used in your implementation but hasn't been mentioned in this document place it in it's proper place next to modules within the same hierarchy.
10. This project could be done both individually and in a group:
    - **Individual:**
      (a) your score which is out of 100 would be multiplied by 1.2.
    - **As a member of a group:**
      (a) your score which is out of 100 would be multiplied by 0.9.
      (b) We assume each of the two members had been involved in every single line of project so the project would be delivered individually.
      (c) **One** of group members should email the names of group mates by **96/1/4 23:55** to the course's email. at 96/1/4 23:56 you are considered as a student eager to complete the project **alone**.
11. In case of delivery, your code will be downloaded by the responsible TA from Moodle, so the only way to convey your code is Moodle and in if you need to reform your code please upload it when possible to be used in the due date.
12. Doing all bonus sections will result you 125 out of 100 :) .

13. **Cheating Alert!**

    (a) All your codes would be processed, both manually and automatically and in case of any similarity by any means, both of individuals involved, would be fined by getting -35 percent of the project's score (note that if pushing your code to Github or any other VCS, exposing your code to a friend or ... results in unexpected similarities of others, you ALL, are responsible!).

    (b) Any cooperation beyond members of the group is forbidden!

    (c) The only source you are allowed to copy form, is AUT/CEIT/Arch101 repo which has been devoted to this course, copying any other source from the Internet,... would be consider just like from another classmate.

14. Remember that, any HDL(Hardware Design Language) is a piece of art :) and could be really enjoyable, if you try your best to understand what's going on instead of just doing it to make it end.

Good Luck