Amirkabir
University of Technology
Tehran Polytechnic

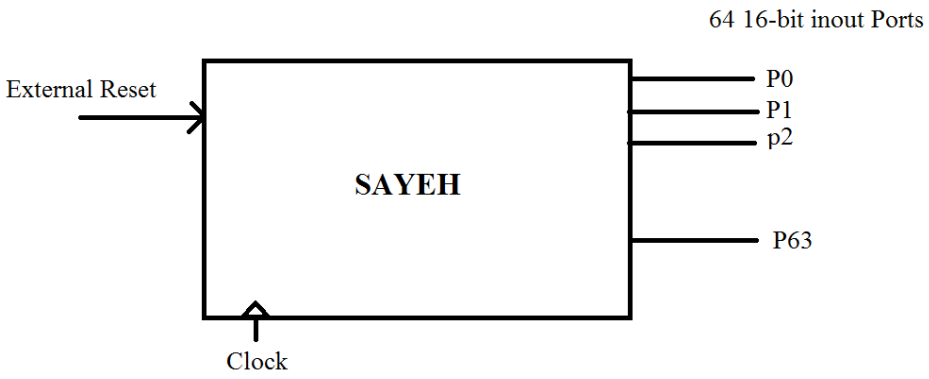
SAYEH simple computer
Project report

Negin Firouzian

9431018

April,2017

What is SAYEH?



SAYEH Stands for Simple Architecture Yet Enough Hardware. It contains 3 major components: Datapath(containing CPU and Memory), Port Manager and Controller.

Datapath: data path connects memory and CPU components using a 16 bit data bus and 16 bit address bus.

1 Datapath

CPU Components:

1.1 Registerfile

Registerfile is a set of general purpose registers used in many cases, SAYEH uses its *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

1.2 Other Registers

• **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 register 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).

1.3 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), the result would be another 16-bit vector and is stored in Rd(Destination Register).

• **OR Component:** This component will perform OR operation on Rs (Source Register) and Rd (Destination Register), the result would be another 16-bit vector and is stored in destination register.

• **Shift-Right and Shift-Left Component:** This component will perform Shift operation on Rs (Source Register) the result would be another 16-bit vector and is stored in Rd (destination register).

• **Comparison Component:** This component will compare Rs and Rd (if equal then zero flag will 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).

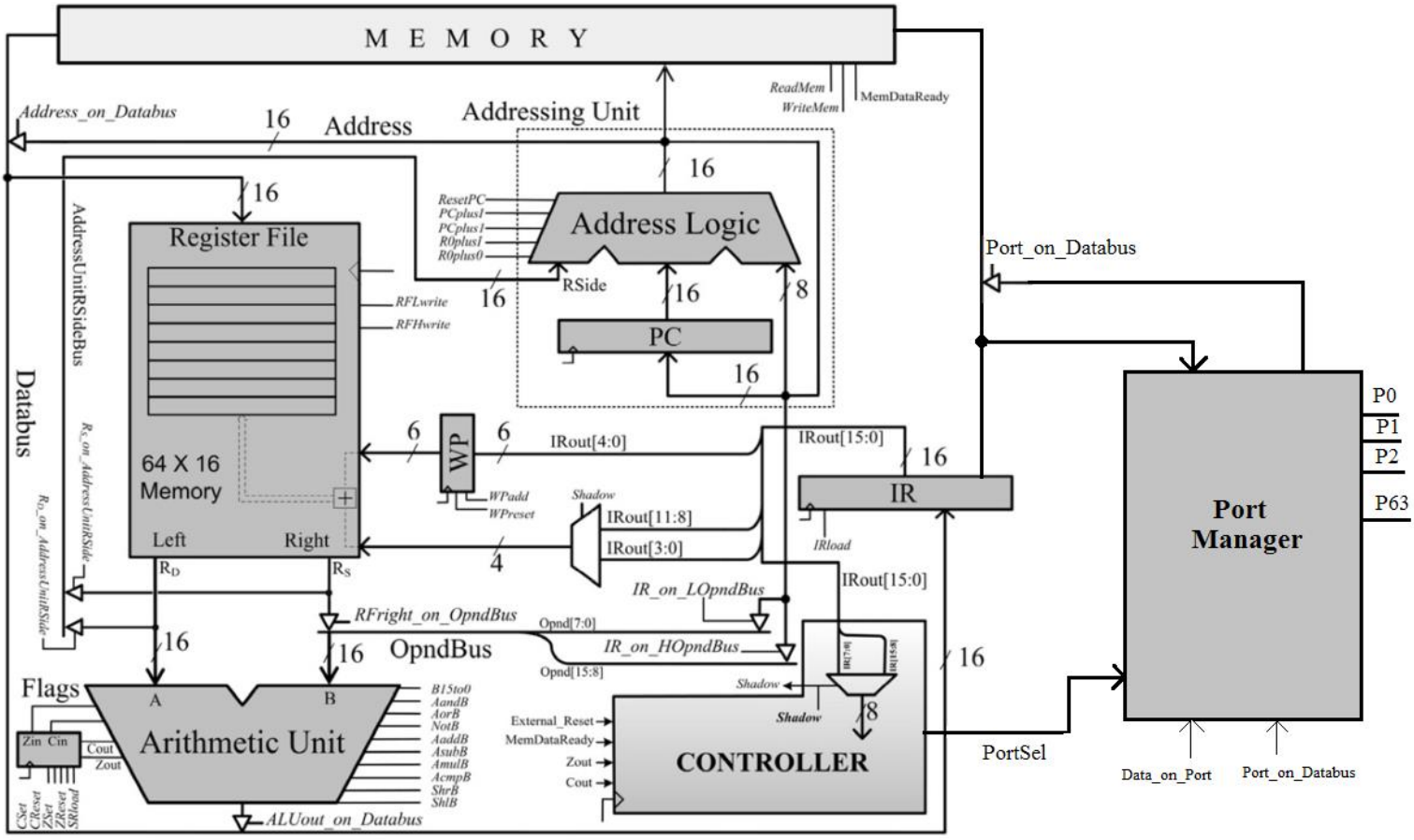
• **Subtraction Component:** This component will perform Subtraction by means of Rd =Rd-Rs-C.

• **Multiplication Component:** This component will perform Multiplication by means of Rd =Rd*Rs. This component is a combinational multiplier which implemented with 8 for loops as shown below:

```
--------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
--------------------
--8 Multiplication Component:
--------------------

entity mul is
  port(Rs: in std_logic_vector(7 downto 0);
    Rd: in std_logic_vector(7 downto 0);
    res: out std_logic_vector(15 downto 0));
end entity;

architecture behavioral of mul is
  type twoD is array (7 downto 0) of std_logic_vector(7 downto 0);
  signal tmp: twoD := ((others=> (others=>'0')));
  signal t0,t1,t2,t3,t4,t5,t6,t7: std_logic_vector(7 downto 0);
begin
```

```
        U0: for i in 0  to 7 generate
    begin
         t0(i)<=Rs(i) and Rd(0);
    end generate;
tmp(0)<=t0;

        U1: for i in 0 to 7 generate
    begin
         t1(i)<=Rs(i) and Rd(1);
    end generate;
tmp(1)<=t1;

        U2: for i in 0 to 7 generate
    begin
         t2(i)<=Rs(i) and Rd(2);
    end generate;
tmp(2)<=t2;

        U3: for i in 0 to 7 generate
    begin
         t3(i)<=Rs(i) and Rd(3);
    end generate;
tmp(3)<=t3;

        U4: for i in 0 to 7 generate
    begin
         t4(i)<=Rs(i) and Rd(4);
    end generate;
tmp(4)<=t4;

        U5: for i in 0 to 7 generate
    begin
         t5(i)<=Rs(i) and Rd(5);
    end generate;
tmp(5)<=t5;

        U6: for i in 0 to 7 generate
    begin
         t6(i)<=Rs(i) and Rd(6);
    end generate;
tmp(6)<=t6;

        U7: for i in 0 to 7 generate
    begin
         t7(i)<=Rs(i) and Rd(7);
    end generate;
tmp(7)<=t7;

res<=tmp(0)+(tmp(1)&'0')+(tmp(2)&"00")+(tmp(3)&"000")+(tmp(4)&"0000")+(tmp(5
)&"00000")+(tmp(6)&"000000")+('0'&tmp(7)&"0000000");
end architecture;
```

Image below illustrates the overall schematic of SAYEH Datapath:



2 Port Manager

This component is about to manage input/output ports of SAYEH, SAYEH has 64 ports named as P0 . . . to P63. 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.

The Portmanager module gets PortSel signal from controller which is a 6-bit vector determining which port should be chosen based on instruction.

Regarding to signals: Port_on_Databus and Data_on_Port Portmanager decide whether read from or write on the Databus.

These features implemented by using Case – When statements as shown below:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity multiplexer64 is
Port (P0,P1,P2,P3,…,P63: inout std_logic_vector(15
downto 0);
      EN: in std_logic;
      Port_on_Databus: in std_logic;
    Data_on_Port: in std_logic;
PortSel:in STD_LOGIC_VECTOR (5 downto 0);

y : out STD_LOGIC_vector(15 downto 0);
yi : in STD_LOGIC_vector(15 downto 0));
end multiplexer64;
architecture Behavioral of multiplexer64 is
begin
process
(EN,Data_on_Port,Port_on_Databus,P0,P1,P2,P3,…,P63,Po
rtSel)
begin

 if Port_on_Databus='1'   then


case PortSel is

when "000000"=>y<=P0;
when "000001"=>y<=P1;
when "000010"=>y<=P2;
.
.
.
when "111111"=>y<=P63;
when others=> null;
end case;
```

```
else
if Data_on_Port='1' and EN='1'  then
  case PortSel is
when "000000"=> P0<= yi;
when "000001"=> P1<= yi;
when "000010"=> P2<= yi;
.
.
.
when "111111"=> P63<= yi;
  when others=> null;
end case;

else
  case PortSel is
when "000000"=> P0<= (others => 'Z');
when "000001"=> P1<= (others => 'Z');
when "000010"=> P2<= (others => 'Z');
.
.
.
when "111111"=> P63<= (others => 'Z');
  when others=> null;
end case;
  end if;
end if;
end process;
End Behavioral;
```

3 Controller

SAYEH Instructions

The general format of 8-bit and 16-bit instructions of SAYEH is shown in image below. 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.

How *Shadow* works?

Shadow signal is used to determine whether we can use another 8 bit of instruction or not. At first shadow is considered ‘1’ in controller this means we expect two 8-bit instructions. By OPCODE we find the length of our instruction, for 8 bit ones we check the shadow signal and if shadow equals ‘1’ after first execute we proceed to the second execute of our current IR data. In that state shadow will become ‘0’ it means second execute have been done so we’ve reached the end of the cycle and we can fetch again.

Instruction Set of SAYEH:

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

Controller sate machine:

State machine used for SAYEH is Moore machine which mean outputs are determined by machine's current state. SAYEH has dozens of control signal which should be handle in each state of controller so to make it simpler in the beginning of process all control signal are assigned to '0' and in each state we only change the signals that are needed to be '1'.

The main states of controller are: Reset, Fetch, Decode, Execute, Execute2, Halt, IDA, OP0, OP1, IFP, OTP, and PCplusOne.

Reset state actives all reset signal, so SAYEH starts from the beginning. *Fetch* and *Decode* states read the memory and transfer it to IR and IR data is transferred to signal IRout. In *Execute*, based on the OPCODE of instructions we decide about the output of signals. For the instructions starting with "0000" and "1111" two separate states, *OP0* and *OP1* are considered so that in these states we can look at next four and two bits of instruction to apply the output signals accordingly. At the end of instructions with the length of 8 bit, shadow signal is check if is '1' we proceed to *Execute2* in this state shadow will become '0' which mean we have processed the first 8 bit of instruction and IRout will be shifted 8 bit to the left so that we don't need repeat Case-When statement in *Execute2* and we will proceed to *Execute* and behave same as newly fetched instructions except shadow is '0'. *IDA* is for "Load Addressed" instruction because this operation cannot be done in one clock cycle. IFP and OTP states are for "Input from Port" and "Output to Port" instruction which can't be executed in one clock cycle. The last state is *PCplusOne*, since we cannot increment PC and execute simultaneously, in some operations, we've considered a separate state for incrementing PC and then we proceed to *Fetch* to execute the next instruction.