10.1 Design of SAYEH Processor Core

This section shows design and description of a small processor in VHDL. The CPU is SAYEH (Simple Architecture, Yet Enough Hardware) that has been designed for educational and benchmarking purposes. The design is simple, and follows the design strategy used for the Adding CPU of Chapter 8.

This section heavily relies on the reader knowing the implementation details that we presented in Chapter 8. Unlike the Adding CPU, SAYEH is a processor with a set of instructions and an architecture that can be used for real embedded processor applications. However, the design, implementation, VHDL coding style, and test strategy of SAYEH are very similar to those of the Adding CPU. Because of the size of the VHDL code of SAYEH, we will not show complete codes of all components of SAYEH in this chapter. Instead, In many cases we show code outlines similar to those of the Adding CPU.

10.1.1 Details of Processor Functionality

The simple CPU example discussed here has a register file that is used for data processing instructions. The CPU has a 16-bit data bus and a 16-bit address bus. The processor has 8 and 16-bit instructions. Short instructions contain shadow instructions, which effectively pack two such instructions into a 16-bit word. Figure 10.1 shows SAYEH interface signals.

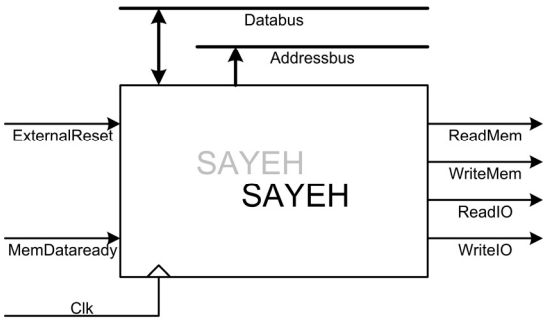

Figure 10.1 SAYEH Interface

10.1.1.1 CPU Components. SAYEH uses its register file for most of its data instructions. Addressing modes of this processor also take advantage of this structure. Because of this, the addressing hardware of SAYEH is simple and the register file output is used in address calculations.

SAYEH components that are used by its instructions include the standard registers such as the Program Counter, Instruction Register, the Arithmetic Logic Unit, and Status Register. In addition, this processor has a register file forming registers *R0*, *R1*, *R2* and *R3* as well as a Window Pointer that defines *R0*, *R1*, *R2* and *R3* within the register file. CPU components and a brief description of each are shown below.

- **PC:** Program Counter, 16 bits
- **R0, R1, R2, and R3:** General purpose registers part of the register file, 16 bits
- **Reg File:** The general purpose registers form a window of 4 in a register file of 64 registers
- **WP:** Window Pointer points to the register file to define *R0*, *R1*, *R2* and *R3*, 6 bits
- **IR:** Instruction Register that is loaded with a 16-bit, an 8-bit, or two 8-bit instructions, 16 bits
- **ALU:** The ALU that can AND, OR, NOT, Shift, Compare, Add, Subtract and Multiply its inputs, 16 bit operands
- **Z flag:** Becomes **1** when the ALU output is **0**
- **C flag:** Becomes **1** when the ALU has a carry output

10.1.1.2 SAYEH Instructions. The general format of 8-bit and 16-bit SAYEH instructions is shown in Figure 10.2. The 16-bit instructions have the *Immediate* field and the 8-bit instructions do not. The *OP-CODE* filed is a 4-bit code that specifies the type of instruction. The *Left* and *Right* fields are two bit codes selecting *R0* through *R3* for source and/or destination of an instruction. Usually, *Left* is used for destination and *Right* for source. The *Immediate* filed is used for immediate data, or if two 8-bit instructions are packed, it is used for the second instruction.

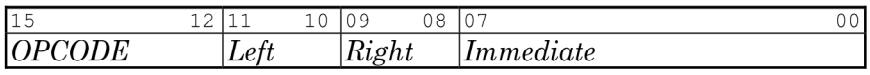

Figure 10.2 SAYEH Instruction Format

Our processor has a total of 29 instructions as shown in Table 10.1. Instructions with *I* immediate field are 16-bit instructions and the rest are 8-bit instructions. Instructions that use the *Destination* and *Source* fields (designated by *D* and *S* in the table of instruction set) have an opcode that is limited to 4 bits. Instructions that do not

require specification of source and destination registers use these fields as opcode extensions. In addition to *nop*, hex code 0F is used as filler for the right most 8-bits of a 16-bit word that only contains an 8-bit instruction in its 8 left-most bits.

Table 10.1 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

In the instruction set, addressed locations in the memory are indicated by enclosing the address in a set of parenthesis. For these instructions, the processor issues *ReadMem* or *WriteMem* signals to the memory. When input and output instructions (*inp, oup*) are executed, SAYEH issues *ReadIO* or *WriteIO* signals to its IO devices.

10.1.2 SAYEH Datapath

The datapath of SAYEH is shown in Figure 10.3. The main components of this machine are: *Addressing Unit* that consists of *PC (Program Counter)* and *Address Logic*, *IR (Instruction Register)*, *WP (Window Pointer)*, *Register File* that consists of *Left Decoder1* and *Right Decoder2*, *ALU (Arithmetic Logic Unit)*, and *Flags*. As shown in Figure 10.3, these components are either hardwired or connected through three-state busses. Component inputs with multiple sources, such as the right hand side input of *ALU*, use three-state busses. Three-state busses in this structure are *Dastabus* 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 busses.

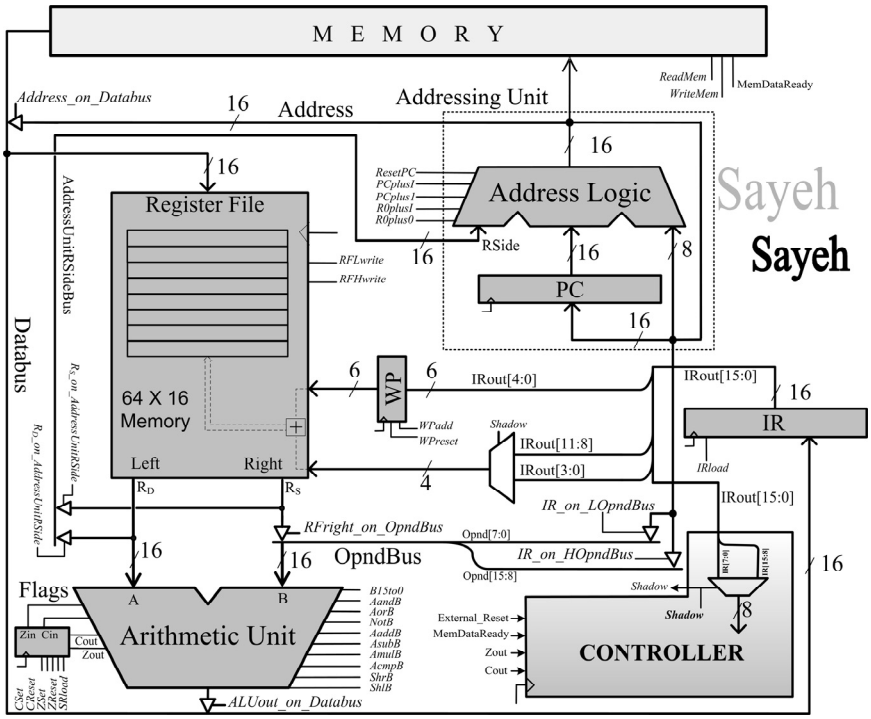

Figure 10.3 SAYEH Datapath

10.1.2.1 Datapath Components. Figure 10.4 shows the hierarchical structure of SAYEH components. The processor has a *datapath* and a *controller*. *Datapath* components are *Addressing Unit*, *IR*, *WP*, *Register File*, *Arithmetic Unit*, and the *Flags* register. The *Addressing Unit* is further partitioned into the *PC* and *Address Logic*.

The *Addressing Logic* is a combinational circuit that is capable of adding its inputs to generate a 16-bit output that forms the address for the processor memory. *Program Counter* and *Instruction Register* are 16-bit registers. *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 are used to select a source register and other two bits select the destination register.

When the Window Pointer 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 *Arithmetic Unit* is a 16-bit arithmetic and logic unit that has logical, shift, add and compare operations. A 9-bit input selects one of the nine functions of the ALU. This code is provided by the processor controller.

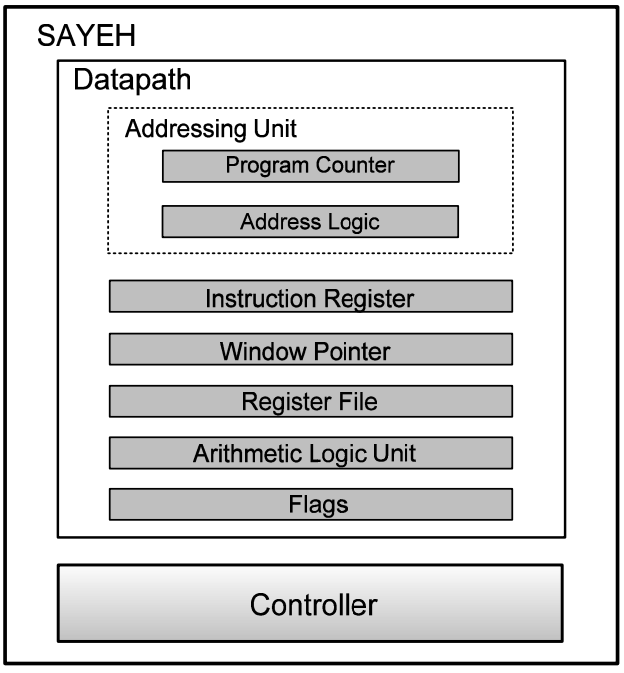

Figure 10.4 SAYEH Hierarchical Structure

Controller 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 clocking in the datapath.

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 busses and logic units in the datapath.

10.2 SAYEH VHDL Description

SAYEH is described according to the hierarchical structure of Figure 10.4. Data components are described separately, and then wired to form the datapath. Controller is described in a single VHDL module. In the complete SAYEH description, the datapath and controller are wired together.

The coding style used for the description of this processor is similar to that of the description of *AddingCPU* of Chapter 8. The CD accompanying this book has the complete code of this processor. As previously mentioned, only an outline and key parts of the VHDL code of SAYEH will be included in this chapter.

10.2.1 Data Components

Combinational and sequential SAYEH data components are 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.

10.2.1.1 Addressing Unit. The Addressing Unit of Figure 10.5 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 10.6 is the VHDL code of the Program Counter. The Address Logic of Figure 10.7

uses control signal inputs of the Addressing Unit to generate input data to the Program Counter via the *PCout* of Figure 10.5. Constants defined and used here correspond to one-hot control signals from the controller.

```
ENTITY AddressingUnit 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 AddressingUnit;

ARCHITECTURE dataflow OF AddressingUnit 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 10.5 ***AddressingUnit*** **VHDL Code**

```
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 10.6 ***ProgramCounter*** **VHDL Code**

```
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 10.7 ***AddressLogic*** **VHDL Code**

10.2.1.2 Arithmetic Unit. The ALU of SAYEH has nine functions shown in Table 10.2. A Mnemonic and the control code of each function are also shown in this table. The complete code of the ALU is shown in Figure 10.8. For readability, constants corresponding to the mnemonics of Table 10.2 are defined and used in this code.

ALU control codes are one-hot. For example, the select input that causes the ALU to perform the add operation is 0000001000, and it is defined as *AaddBH*. Control inputs of this unit are *B15to0*, *AandB*, *AorB*, *notB*, *shlB*, *shrB*, *AaddB*, *AsubB*, *AmulB* and *AcmpB* that select its various operations.

```
ENTITY ArithmeticUnit IS
   PORT (
      A, B : IN std_logic_vector (15 DOWNTO 0);
      B15to0, AandB, AorB, notB : IN std_logic;
      shlB, shrB, AaddB, AsubB : IN std_logic;
      AmulB, AcmpB : IN std_logic;
      aluout : OUT std_logic_vector (15 DOWNTO 0);
      cin : IN std_logic;
      zout, cout : OUT std_logic
   );
END ArithmeticUnit;

ARCHITECTURE dataflow OF ArithmeticUnit IS

   COMPONENT mult
      PORT ( x, y : IN std_logic_vector (7 DOWNTO 0);
             z : OUT std_logic_vector (15 DOWNTO 0) );
   END COMPONENT;
   FOR ALL : mult USE ENTITY work.mult_8by8 (bitwise);

   CONSTANT B15to0H : std_logic_vector (9 DOWNTO 0)
                    := "1000000000";
   CONSTANT AandBH  : std_logic_vector (9 DOWNTO 0)
                    := "0100000000";
   CONSTANT AorBH   : std_logic_vector (9 DOWNTO 0)
                    := "0010000000";
   CONSTANT notBH   : std_logic_vector (9 DOWNTO 0)
                    := "0001000000";
   CONSTANT shlBH   : std_logic_vector (9 DOWNTO 0)
                    := "0000100000";
   CONSTANT shrBH   : std_logic_vector (9 DOWNTO 0)
                    := "0000010000";
   CONSTANT AaddBH  : std_logic_vector (9 DOWNTO 0)
                    := "0000001000";
   CONSTANT AsubBH  : std_logic_vector (9 DOWNTO 0)
                    := "0000000100";
   CONSTANT AmulBH  : std_logic_vector (9 DOWNTO 0)
                    := "0000000010";
   CONSTANT AcmpBH  : std_logic_vector (9 DOWNTO 0)
                    := "0000000001";

   SIGNAL product : std_logic_vector (15 DOWNTO 0);
   SIGNAL aluoutSignal : std_logic_vector (15 DOWNTO 0);

BEGIN
   PROCESS (A, B, B15to0, AandB, AorB, notB, shlB, shrB,
            AaddB, AsubB, AmulB, cin, aluoutSignal,
            product, AcmpB)
      VARIABLE temp :  std_logic_vector (9 DOWNTO 0);
      VARIABLE sum :  std_logic_vector (16 DOWNTO 0);
      VARIABLE sub :  std_logic_vector (16 DOWNTO 0);
   BEGIN --                                          Continued
```

```
      zout <= '0'; cout <= '0';
      aluoutSignal <= (OTHERS=>'0');
      temp := (B15to0, AandB, AorB, notB,
               shlB, shrB, AaddB, AsubB, AmulB, AcmpB);
      sum := A + B + (16 DOWNTO 1=> '0', 0=> cin) ;
      sub := A - B - (16 DOWNTO 1=> '0', 0=> cin);

      CASE temp IS
         WHEN B15to0H =>
            aluoutSignal <= B;
         WHEN AandBH =>
            aluoutSignal <= A and B;
         WHEN AorBH =>
            aluoutSignal <= A or B;
         WHEN notBH =>
            aluoutSignal <= not (B);
         WHEN shlBH =>
            aluoutSignal <= B (14 DOWNTO 0) & B (0);
         WHEN shrBH =>
            aluoutSignal <= B (15) & B (15 DOWNTO 1);
         WHEN AaddBH =>
            aluoutSignal <= sum (15 DOWNTO 0);
            cout <= sum (16);
         WHEN AsubBH  =>
            aluoutSignal <= sub (15 DOWNTO 0);
            cout <= sub (16);
         WHEN AmulBH  =>
            aluoutSignal <= product;
         WHEN AcmpBH  =>
            aluoutSignal <= (OTHERS=>'1');
            IF (A>B) THEN
               cout <= '1';
            ELSE
               cout <= '0'; END IF;

            IF (A=B) THEN
               zout <= '1';
            ELSE
               zout <= '0'; END IF;
         WHEN OTHERS => aluoutSignal <= (OTHERS=>'0');
      END CASE;
      IF (aluoutSignal = "0000000000000000") THEN
         zout <= '1';
      END IF;
   END PROCESS;

   MULT8x8: mult PORT MAP (A (7 DOWNTO 0),
                           B (7 DOWNTO 0), product);
   aluout <= aluoutSignal;

END dataflow;
```

Figure 10.8 ***ArithmeticUnit*** **VHDL Code**

In order to insure that no unwanted latches are implied, all ALU outputs are set to their inactive values at the beginning of the process statement of its VHDL code. In a case statement in this code, *aluout* and its flags outputs are set according to the selected control input of the ALU.

The multiplication function (*AmulBH*) of the ALU is handled by instantiating *bitwise* architecture of *mult_8by8*. This is the array multiplier that we presented in Chapter 8. Instantiation of the multiplier appears near the end of the code in Figure 10.8. The output of the multiplier is put on the *product* local signal and is assigned to *aluoutSignal* in the body of the process statement of the ALU.

Instead of instantiating this predefined multiplier, we could use the multiplication operation to leave the implementation of multiplication up to the synthesis tool. Another alternative would be to use a sequential multiplier such as add-and-shift or the Booth multiplier of Chapter 8. In such cases, a separate clock and proper handshaking would be required.

Table 10.2 ALU Operations

Mnemonic	Description	Code
B15to0H	Place B on the output	1000000000
AandBH	Place A and B on the output	0100000000
AorBH	Place A or B on the output	0010000000
notBH	Place not B on the output	0001000000
shlBH	Shift B one bit to the left	0000100000
shrBH	Shift B one bit to the right	0000010000
AaddBH	Place A + B on the output	0000001000
AsubBH	Place A - B on the output	0000000100
AmulBH	Place A * B on the output	0000000010
AcmpBH	Z = 1 if A = B; C = 1 if A < B	0000000001

10.2.1.3 Instruction Register. SAYEH Instruction Register is shown in Figure 10.9. This unit is a 16-bit register with an active high load-enable input. As shown, the only control input of the *InstructionRegister* module is *IRload*.

10.2.1.4 Register File. Figure 10.10 shows the VHDL code of SAYEH Register File. This is a two-port memory with a moving window pointer. Reading the register file is unclocked, while writing is controlled by the rising edge of the clock. At all times, contents of two locations appear on the right and left output ports (*Rout*, *Lout*) of the register file.

```
ENTITY InstrunctionRegister IS
   PORT ( input : IN std_logic_vector (15 DOWNTO 0);
          IRload, clk : IN std_logic;
          output : OUT std_logic_vector (15 DOWNTO 0) );
END InstrunctionRegister;

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

Figure 10.9 ***InstructionRegister*** **VHDL Code**

```
ENTITY RegisterFile IS
   PORT (
      input : IN std_logic_vector (15 DOWNTO 0);
      clk : IN std_logic;
      base : IN std_logic_vector (5 DOWNTO 0);
      Laddr, Raddr : IN std_logic_vector (1 DOWNTO 0);
      RFLwrite, RFHwrite : IN std_logic;
      Lout, Rout : OUT std_logic_vector (15 DOWNTO 0) );
END RegisterFile;

ARCHITECTURE dataflow OF RegisterFile IS --MS: please check
   SIGNAL Raddress : std_logic_vector (5 DOWNTO 0);
   SIGNAL Laddress : std_logic_vector (5 DOWNTO 0);
BEGIN
   Laddress <= Base + Laddr;
   Raddress <= Base + Raddr;
   Lout <= MemoryFile (conv_integer(Laddress));
   Rout <= MemoryFile (conv_integer(Raddress));

   PROCESS (clk) BEGIN
      IF (clk = '1') THEN
         IF (RFLwrite = '1') THEN
            MemoryFile (conv_integer(Laddress))(7 DOWNTO 0)
                                    <= input(7 DOWNTO 0);
         END IF;
         IF (RFHwrite = '1') THEN
            MemoryFile(conv_integer(Laddress))(15 DOWNTO 8)
                                    <= input(15 DOWNTO 8);
         END IF;
      END IF;
   END PROCESS;
END dataflow;
```

Figure 10.10 ***RegisterFile*** **VHDL Code**

For calculation of memory addresses, the base of the window pointer (*Base*) is added to the left and right addresses (*Laddr* and *Raddr*) and absolute addresses are calculated (*Laddress* and *Raddress*). Memory words are read on appropriate left and right outputs (*Lout* and *Rout*) from *Laddress* and *Raddress* absolute locations.

Writing into the memory is done in the location pointed by the left absolute address, *Laddress*. The *RFLwrite* and *RFHwrite* control signals decide whether a write is done to the low order or the high order bits of the Register File. If both these signals are active, writing is done in a 16-bit word addressed by *Laddress*.

10.2.1.5 Window Pointer. The VHDL code of the Window Pointer is shown in Figure 10.11. This unit has two control lines; one is for resetting it and the other is for adding its 6-bit input to its register contents. As with other register structures of SAYEH, this register is positive edge triggered.

```
ENTITY WindowPointer IS
   PORT (
      input : IN std_logic_vector (5 DOWNTO 0);
      clk : IN std_logic;
      WPreset, WPadd : IN std_logic;
      output : OUT std_logic_vector (5 DOWNTO 0)
   );
END WindowPointer;

ARCHITECTURE dataflow OF WindowPointer IS
   SIGNAL outputSignal : std_logic_vector (5 DOWNTO 0);
BEGIN

   PROCESS (clk)
   BEGIN
      IF (clk = '1') THEN
         IF (WPreset = '1') THEN
            outputSignal <= "000000";
         ELSIF (WPadd = '1') THEN
            outputSignal <= outputSignal + input;
         END IF;
      END IF;
   END PROCESS;

   output <= outputSignal;

END dataflow;
```

Figure 10.11 ***WindowPointer*** **VHDL Code**

10.2.1.6 Status Register. SAYEH Status Register is a collection of two flags that are set or reset according to their control signals. The VHDL code of this register is shown in Figure 10.12. This unit has five control signals for setting and resetting the two flags and for its synchronous load control. The latter (*SRload*) is used when an ALU operation is to affect the status flags. As with other register structures of SAYEH, this register is positive edge triggered.

```
ENTITY StatusRegister IS
   PORT (
      Cin, Zin, SRload, clk : IN std_logic;
      Cset, Creset, Zset, Zreset : IN std_logic;
      Cout, Zout : OUT std_logic
     );
END StatusRegister;

ARCHITECTURE dataflow OF StatusRegister IS BEGIN
   PROCESS (clk) BEGIN
      IF (clk = '1') THEN
         IF (SRload = '1') THEN
            Cout <= Cin;
            Zout <= Zin;
         ELSIF (Cset='1') THEN
            Cout <= '1';
         ELSIF (Creset='1') THEN
            Cout <= '0';
         ELSIF (Zset='1') THEN
            Zout <= '1';
         ELSIF (Zreset='1') THEN
            Zout <= '0';
         END IF;
      END IF;
   END PROCESS;
END dataflow;
```

Figure 10.12 ***StatusRegister*** **VHDL Code**

10.2.2 SAYEH Datapath

Figure 10.13 shows the datapath entity and architecture of SAYEH. This unit specifies component instantiations and bussing structure of the CPU according to the diagram of Figure 10.3. Inputs of this module are the processor's data and address busses, as well as control signals that are provided by the controller of the CPU. Control signals shown in the Data Path are routed to the instantiated data components or to the internal buses that are specified in this module.