14 Design of SAYEH Processor

This chapter shows design of a small computer in Verilog and implementation of it on UP2 using Quartus II. 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 multiplier of Chapter 1. We rely on the material of the chapter on computer architectures for providing the necessary background for understanding details of the hardware of SAYEH in this chapter.

14.1 CPU Description

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 14.1 shows SAYEH interface signals.

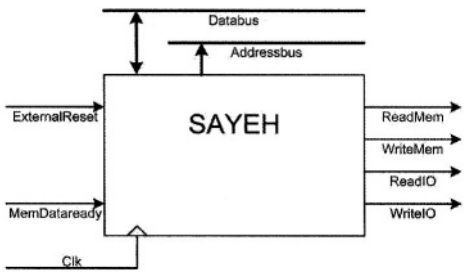

Figure 14.1 SAYEH Interface

14.1.1 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 a simple one 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 8 registers
- **WP:** Window Pointer points to the register file to define *R0, R1, R2* and *R3,* 3 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

14.1.2 SAYEH Instructions

The general format of 8-bit and 16-bit SAYEH instructions is shown in Figure 14.2. The 16-bit instructions have the *Immediate* field and the 8-bit instructions do not. The *OPCODE* 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.

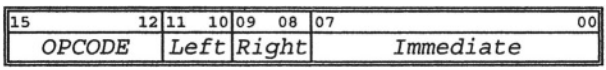

Figure 14.2 SAYEH Instruction Format

Our processor has a total of 29 instructions as shown in Table 14.1. 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. Because of this, our

processor has room for extending its instruction set beyond what is shown in Table 14.1. 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 14.1 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 port R_S and 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 Immediate Low	1111-D-00-I	R_{DL} <= {8'bZ, I}
mih	Move Immediate 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 window pointer	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. When these instructions are executed, 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.

14.1.3 SAYEH Datapath

The datapath of SAYEH is shown in Figure 14.3. Main components and their lower level structures are listed below.

1. Addressing Unit

   a. PC (Program Counter)
   b. Address Logic
2. IR (Instruction Register)
3. WP (Window Pointer)
4. Register File
   a. Decoder 1 (Left)
   b. Decoder2 (Right)
5. ALU (Arithmetic Unit)
6. Flags

As shown in Figure 14.3, 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 buses. Three-state busses in this structure are *Databus* and *OpndBus*. Names shown on component interconnections are used in the Verilog description of the processor.

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.

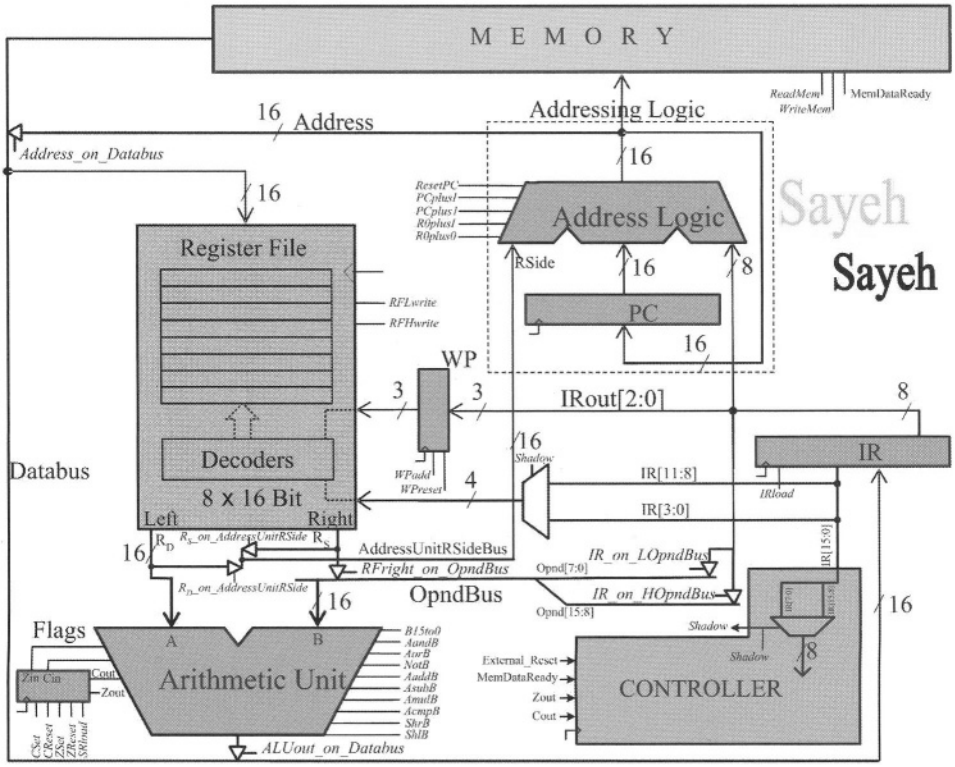

Figure 14.3 SAYEH Datapath

14.1.4 Datapath Components

Figure 14.4 shows the hierarchical structure of SAYEH components. The processor has a *datapath* and a *controller*. *Datapath* components are *Addressing Unit, Instruction Register, Window Pointer, Register File, Arithmetic Unit,* and the *Flags* register. The *Addressing Unit* is further partitioned into the *Program Counter* and *Address Logic*.

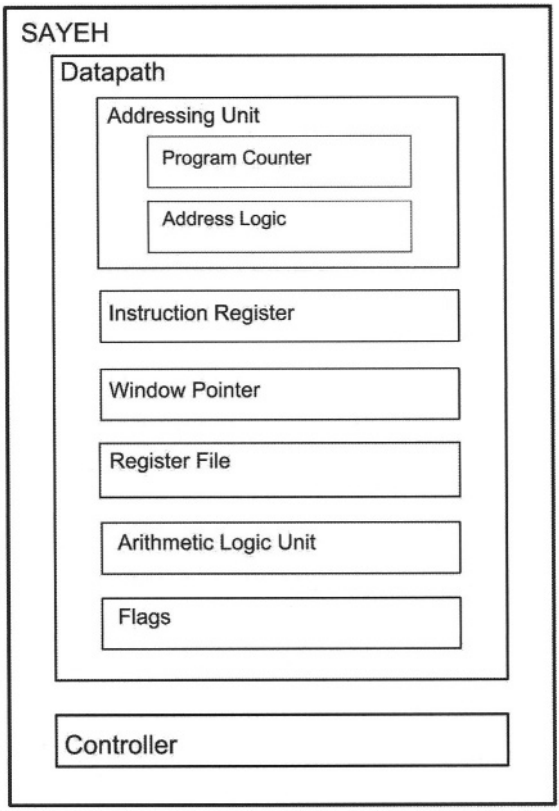

Figure 14.4 Sayeh Hierarchical Structure

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. The *Program Counter* and *Instruction Register* are 16-bit registers. The *Register File* is a two-port memory and a file of 8 16-bit registers. The *Window Pointer* is a 3-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 3-bit input to its current input. 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 the functions shown in Table 14.2. A 9-bit input selects the function of the ALU shown in this table. This code is provided by the processor controller.

Table 14.2 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	One bit shift to left	0000100000
shrBH	One bit shift to 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

Controller of SAYEH has eleven states for 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 falling edge of the main system clock. Control signals remain active after one falling edge through the next. This duration allows for propagation of signals through the busses and logic units in the datapath.

14.2 SAYEH Verilog Description

SAYEH is described according to the hierarchical structure of Figure 14.4. Data components are described separately, and then wired to form the datapath. Controller is described in a single Verilog module. In the complete SAYEH description, the datapath and controller are wired together.

14.2.1 Data Components

Combinational and sequential SAYEH data components are described here. The combinational ones are like the ALU that perform 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 that are controlled by the controller.

```
module AddressingUnit (
   Rside, Iside, Address, clk, ResetPC, PCplusI, PCplus1, RplusI, Rplus0, PCenable);
input [15:0] Rside;
input [7:0] Iside;
input ResetPC, PCplusI, PCplus1, RplusI, Rplus0, PCenable;
input clk;
output [15:0] Address;
wire [15:0] PCout;
   ProgramCounter PC (Address, PCenable, clk, PCout);
   AddressLogic AL (PCout, Rside, Iside, Address, ResetPC, PCplusI, PCplus1, RplusI, Rplus0);
Endmodule
```

Figure 14.5 ***AddressingUnit*** **Verilog Code**

```
module ProgramCounter (in, enable, clk, out);
input [15:0] in;
input enable, clk;
output [15:0] out;
reg [15:0] out;

   always @ (negedge clk) if (enable) out = in;

endmodule
```

Figure 14.6 ***ProgramCounter*** **Verilog Code**

```
module AddressLogic (
   PCside, Rside, Iside, ALout, ResetPC, PCplusI, PCplus1, RplusI, Rplus0);
input [15:0] PCside, Rside;
input [7:0] Iside;
input ResetPC, PCplusI, PCplus1, RplusI, Rplus0;
output [15:0] ALout;
reg [15:0] ALout;

   always @ (PCside or Rside or Iside or ResetPC or PCplusI or PCplus1 or RplusI or Rplus0)
      case ({ResetPC, PCplusI, PCplus1, RplusI, Rplus0})
         5'b10000: ALout = 0;
         5'b01000: ALout = PCside + Iside;
         5'b00100: ALout = PCside + 1;
         5'b00010: ALout = Rside + Iside;
         5'b00001: ALout = Rside;
         default: ALout = PCside;
   endcase

endmodule
```

Figure 14.7 ***AddressLogic*** **Verilog Code**

```
`define B15to0H  10'b1000000000
`define AandBH   10'b0100000000
`define AorBH    10'b0010000000
`define notBH    10'b0001000000
`define shlBH    10'b0000100000
`define shrBH    10'b0000010000
`define AaddBH   10'b0000001000
`define AsubBH   10'b0000000100
`define AmulBH   10'b0000000010
`define AcmpBH   10'b0000000001

module ArithmeticUnit ( A, B,
        B15to0, AandB, AorB, notB, shlB, shrB, AaddB, AsubB, AmulB, AcmpB, aluout,
         cin, zout, cout);
input [15:0] A, B;
input B15to0, AandB, AorB, notB, shlB, shrB, AaddB, AsubB, AmulB, AcmpB;
input cin;
output [15:0] aluout;
output zout, cout;
reg [15:0] aluout;
reg zout, cout;

   always @( A or B or B15to0 or AandB or AorB or notB or
              shlB or shrB or AaddB or AsubB  or AmulB  or AcmpB or cin)
   begin
      zout = 0; cout = 0; aluout = 0;

      case ({B15to0, AandB, AorB, notB, shlB, shrB, AaddB, AsubB, AmulB, AcmpB})
          `B15to0H:aluout = B;
          `AandBH: aluout = A & B;
          `AorBH:  aluout = A | B;
          `notBH:  aluout = ~B;
          `shlBH:  aluout = {B[15:0], B[0]};
          `shrBH:  aluout = {B[15], B[15:1]};
          `AaddBH: {cout, aluout} = A + B + cin;
          `AsubBH:  {cout, aluout} = A - B - cin;
          `AmulBH:  aluout = A[7:0] * B[7:0];
          `AcmpBH: begin
             aluout = A;
             if (A> B) cout = 1 ; else cout = 0;
             if (A==B) zout = 1 ; else zout = 0;
          end
          default: aluout = 0;
      endcase

      if (aluout == 0) zout = 1'b1;
   end

endmodule
```

Figure 14.8 ***ArithmeticUnit*** **Verilog Code**

Addressing Unit. The Addressing Unit, shown in Figure 14.5, 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 14.6 is the Verilog code of the Program Counter. The Address Logic of Figure 14.7 uses control signal inputs of the Addressing Unit to generate input data to the Program Counter via the *PCout* of Figure 14.5.

Arithmetic Unit. The ALU of SAYEH is shown in Figure 14.8. For readability, control input codes of this unit are defined according to their function. 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. In order to insure that no unwanted latches are made, all ALU outputs are set to their inactive values at the beginning of the **always** statement of its Verilog code. In a **case**-statement in this code, *aluout* and its flags outputs are set according to the selected control input of the ALU.

Instruction Register. SAYEH Instruction Register is shown in Figure 14.9. 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*.

Register File. Figure 14.10 shows the Verilog code of SAYEH Register File. This is a two-port memory with a moving window pointer. For reading from the memory, the base of the window pointer (*Base*) is added to the left and right addresses *(Laddress* and *Raddress)* and memory words are read on appropriate left and right outputs *(Lout* and *Rout).* Writing into the memory is done in the location pointed by its left address (left is used for instruction destinations). 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* plus *Base*.

```
module InstrunctionRegister (in, IRload, clk, out);
input [15:0] in;
input IRload, clk;
output [15:0] out;
reg [15:0] out;

   always @(negedge clk) if (IRload == 1) out <= in;
endmodule
```

Figure 14.9 *InstructionRegister* Verilog Code

```
module RegisterFile (in, clk, Laddr, Raddr, Base, RFLwrite, RFHwrite, Lout, Rout);

input [15:0] in;
input clk, RFLwrite, RFHwrite;
input [1:0] Laddr, Raddr;
input [2:0] Base;
output [15:0] Lout, Rout;

reg [15:0] MemoryFile [0:7];

wire [2:0] Laddress = Base + Laddr;
wire [2:0] Raddress = Base + Raddr;

assign Lout = MemoryFile [Laddress];
assign Rout = MemoryFile [Raddress];

reg [15:0] TempReg;

   always @(negedge clk) begin
      TempReg = MemoryFile [Laddress];
      if (RFLwrite) TempReg [7:0] = in [7:0];
      if (RFHwrite) TempReg [15:8] = in [15:8];
      MemoryFile [Laddress] = TempReg;
   end

endmodule
```

Figure 14.10 ***RegisterFile*** **Verilog Code**

14.2.2 SAYEH Datapath

Figure 14.11 shows the datapath of SAYEH module. This module specifies component instantiations and bussing structure of the CPU according to the diagram of Figure 14.3. 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 module are routed to the data components that are instantiated here or the internal buses that are specified in this module.

Following the declarations, the Data Path module instantiates Addressing Unit, Arithmetic Unit, Register File, Instruction Register, Status Register, and the Window Pointer. Control signals that are inputs of *DataPath* are passed from this module to the data components via their port connections. For example, *ResetPC* that is an input of *DataPath* and a control signal of *AddressingUnit* appears on the port list of *AddressingUnit* in its instantiation statement.

The part that follows module instantiations makes bus assignments to the internal buses of this module. For example, assignment of the output of *ArithmeticUnit* to *Databus* that is controlled by *ALU_on_Databus* is done by an **assign** statement with a right hand side that is a conditional expression. Note the assignment of 16'bZZZZZZZZZZZZZZZZ to *Databus* when none of control signals of this bus are active.

```
module DataPath (
   clk, Databus, Addressbus,
   ResetPC, PCplusI, PCplus1, RplusI, Rplus0,
   Rs_on_AddressUnitRSide, Rd_on_AddressUnitRSide, EnablePC,
   B15to0, AandB, AorB, notB, shlB, shrB, AaddB, AsubB, AmulB, AcmpB, RFLwrite, RFHwrite,
   WPreset, WPadd, IRload, SRload, Address_on_Databus, ALU_on_Databus,
   IR_on_LOpndBus, IR_on_HOpndBus, RFright_on_OpndBus,
   Cset, Creset, Zset, Zreset, Shadow, Instruction, Cout, Zout );
input clk;
inout [15:0] Databus;
output [15:0] Addressbus, Instruction;
output Cout, Zout;
input
   ResetPC, PCplusI, PCplus1, RplusI, Rplus0,
   Rs_on_AddressUnitRSide, Rd_on_AddressUnitRSide, EnablePC,
   B15to0, AandB, AorB, notB, shlB, shrB, AaddB, AsubB, AmulB, AcmpB, RFLwrite, RFHwrite,
   WPreset, WPadd, IRload, SRload, Address_on_Databus, ALU_on_Databus,
   IR_on_LOpndBus, IR_on_HOpndBus, RFright_on_OpndBus,
   Cset, Creset, Zset, Zreset, Shadow;
wire [15:0] Right, Left, OpndBus, ALUout, IRout, Address, AddressUnitRSideBus;
wire SRCin, SRZin, SRZout, SRCout;
wire [2:0] WPout;
wire [1:0] Laddr, Raddr;
   AddressingUnit AU (AddressUnitRSideBus, IRout[7:0], Address, clk,
                      ResetPC, PCplusI, PCplus1, RplusI, Rplus0, EnablePC);
   ArithmeticUnit AL (Left, OpndBus, B15toO, AandB, AorB, notB, shlB, shrB,
                      AaddB, AsubB, AmulB, AcmpB, ALUout, SRCout, SRZin, SRCin);
   RegisterFile RF (Databus, clk, Laddr, Raddr, WPout, RFLwrite, RFHwrite, Left, Right);
   InstrunctionRegister IR (Databus, IRload, clk, IRout);
   StatusRegister SR (SRCin, SRZin, SRload, clk, Cset, Creset,
                      Zset, Zreset, SRCout, SRZout);
   WindowPointer WP (IRout[2:0], clk, WPreset, WPadd, WPout);

   assign AddressUnitRSideBus = (Rs_on_AddressUnitRSide) ? right :
                      (Rd_on_AddressUnitRSide) ? Left : 16'bZZZZZZZZZZZZZZZZ;
   assign Addressbus = Address;
   assign Databus = (Address_on_Databus) ? Address :
            (ALU_on_Databus) ? ALUout : 16'bZZZZZZZZZZZZZZZZ;
   assign OpndBus[07:0] = IR_on_LOpndBus == 1 ? IRout[7:0] : 8'bZZZZZZZZ;
   assign OpndBus[15:8] = IR_on_HOpndBus == 1 ? IRout[7:0] : 8'bZZZZZZZZ;
   assign OpndBus = RFright_on_OpndBus == 1 ? Right : 16'bZZZZZZZZZZZZZZZZ;

   assign Zout = SRZout;
   assign Cout = SRCout;
   assign Instruction = IRout[15:0];

   assign Laddr = (~Shadow) ? IRout[11:10] : IRout[3:2];
   assign Raddr = (~Shadow) ? IRout[09:08] : IRout[1:0];
endmodule
```

Figure 14.11 SAYEH *DataPath* Module

In the last part of the *DataPath* module, bits of *IR* that indicate source and destination registers to the Register File are placed on *Laddr* and *Raddr* inputs

of this register. The *Shadow* signal that becomes **1** if a shadow instruction is being executed is used to select appropriate bits of the *IR* as source and destination addresses.

14.2.3 SAYEH Controller

The controller of SAYEH is a state machine with nine states that issues appropriate control signals to the Data Path. The controller uses the Huffman style of coding, in which the state machine has a large combinational part that is responsible for state transitions and issuing controller outputs. State transitions are done by setting next state values to the *Nstate.* Figure 14.12 shows a general outline of this controller. Various sections of this outline are discussed below.

Controller Ports. The instruction register output, ALU flags, and external control signals constitute the inputs of the controller. The outputs of the controller are 38 control signals going to the Data Path and a *Shadow* output that indicates that the controller is handling a shadow instruction. As shown in Figure 14.12, controller outputs are declared as **reg** and are assigned values in the combinational **always** block of the controller module.

Control States. A **parameter** declaration declares the eight states of the controller. States *reset* and *halt* are for the initial state of the machine and its halt state. In state *fetch* the machine begins fetching a 16-bit instruction that can include an 8-bit instruction and a shadow. State *memread* is entered while our controller is waiting for *memDataReady* signal from the memory indicating that its data is ready. Execution of instructions is performed in the *exec1* state. This state is entered from the *memread* state. The *lda* instruction that is not completed by the *exec1* state requires the additional state of *exec1lda* to complete its memory read. States *exec2* and *exec2lda* are like *exec1* and *exec1lda* except that they handle the shadow part of an instruction. The execute state of most instructions *(exec1* or *exec2)* increment the program counter while the instruction is being executed. However, certain instructions that use the address bus for their execution cannot increment *PC* while they are being executed. For these instructions, the *incpc* state increments the program counter.

Opcodes. Referring to Figure 14.12, instruction opcodes are declared as 4-bit parameters in the controller of SAYEH. These parameters are according to the processor's instruction set of Table 14.1.

State Declarations. As mentioned, the coding style the controller is according to the Huffman style of Figure 3.56 discussed in Section 3.3.4. The next state and present states, required by this style of coding, are declared in the controller of SAYEH as 4-bit registers, *Nstate* and *Pstate.*

```
module controller (
  ExternalReset, clk, ResetPC, PCplusI, PCplus1, RplusI, Rplus0, . . . );
input
```

```
  ExternalReset, clk, . . .
output
  ResetPC, PCplusI, PCplus1, RplusI, Rplus0, . . .
reg
  ResetPC, PCplusI, PCplus1, RplusI, Rplus0, . . .
parameter [3:0]
  reset = 0,     halt = 1 ,       fetch = 2,       memread = 3,
  exec1 = 4,    exec2 = 5,     exec1lda = 6,  exec2lda = 7,        incpc    =8;
parameter nop = 4'b0000;
parameter hlt = 4'b0001;
parameter szf = 4'b0010;
. . .
reg [3:0] Pstate, Nstate;

wire ShadowEn = ~(Instruction[7:0] == 8'b000011111)

   always @ (Instruction or Pstate or ExternalReset or Cflag or Zflag or memDataReady) begin
      ResetPC            = 1'b0;
      PCplusI            = 1'b0;
      PCplus1            = 1'b0;
      RplusI             = 1'b0;
      Rplus0             = 1'b0;
      . . .
      case (Pstate)
         reset :
         . . .
         halt:
         . . .
         fetch :
         . . .
         memread :
         . . .
         exec1 :
         . . .
         exec1lda :
         . . .
         exec2 :
         . . .
         exec2lda :
         . . .
         incpc :
         . . .
         default:  Nstate = reset;
      endcase
   end

   always @ (negedge clk) Pstate = Nstate;

endmodule
```

Figure 14.12 SAYEH Controller General Outline

Shadow Instructions: The *ShadowEn* signal that is internal to the controller is set when the hex code 0F (this code indicates that the right-most bits are not used) is not found in the right-most eight bits of a 16-bit instruction. If this

wire is **1** and execution of an 8-bit instruction is complete, the controller branches to *exec2* to execute the second half of the instruction before the next fetching begins.

Combinational Block. The combinational block of SAYEH controller has an **always** block that has a main **case** statement with case choices for every state of the machine. Transitions from one state to another and issuing control signals are performed in the case statement. At the beginning of the **always** statement, all control signals are set to their inactive values in order to avoid latches on these outputs.

```
always @ (Instruction, Pstate, ExternalReset, Cflag, Zflag) begin
 . . .
   case (Pstate)
   . . .
      exec1 :
         if (ExternalReset == 1'b1) Nstate = reset;
         else begin
            case (Instruction[15:12])
            . . .
               mvr : begin
                  RFright_on_OpndBus = 1'b1;
                  B15to0 = 1'b1;
                  ALU_on_Databus = 1'b1;
                  RFLwrite = 1'b1;
                  RFHwrite = 1'b1;
                  SRload = 1'b1;
                  if(ShadowEn==1'b1)
                     Nstate = exec2;
                  else begin
                     PCplus1 = 1'b1;
                     EnablePC=1'b1;
                     Nstate = fetch;
                  end
               end

               lda : begin
                  Rplus0 = 1'b1;
                  Rs_on_AddressUnitRSide = 1'b1;
                  ReadMem = 1'b1;
                  Nstate = exec1lda;
               end
               . . .
            endcase
         end
   endcase
end
. . .
```

Figure 14.13 Instruction Execution

Sequential Block. The last part of the code outline of Figure 14.12 is the sequential **always** block for clocking *Pstate* into *Nstate*. The control state register of SAYEH and all its data registers are falling edge trigger. Control

signals issues by the controller remain active through the next falling edge of the system clock.

Instruction Execution. Figure 14.13 zooms on the combinational **always** statement of the *controller* **module** and shows the details of execution of *mvr* in the *exec1* state of the controller. Signals issued for the execution of this instruction are shown in this figure. This instruction reads a word from the right address of the Register File and writes it into its left address. The right and left (source and destination) addresses are provided in the Data Path by connections made from *IR* to the Register File.

```
always @ (Instruction, Pstate, ExternalReset, Cflag, Zflag) begin
 . . .
  case (Pstate)
    . . .
    exec1Ida:
      if (ExternalReset == 1'b1)
        Nstate = reset;
      else begin
        if (memDataReady == 1'b0) begin
          Rplus0 = 1'b1;
          Rs_on_AddressUnitRSide = 1'b1;
          ReadMem = 1'b1;
          Nstate = exec1Ida;
        end
        else begin
          RFLwrite = 1'b1;
          RFHwrite = 1'b1;
          if(ShadowEn==1'b1)
            Nstate = exec2;
          else begin
            PCplus1 =1'b1;
            EnablePC=1'b1;
            Nstate = fetch;
          end
        end
      end
      . . .
  endcase
end
. . .
```

Figure 14.14 Memory Handshaking for *exec1Ida*

The *RFright_on_OpndBus* control signal is issued to read the source register from *RegisterFile* onto *OpndBus*. Since this bus is the input of the ALU, the data on the ALU's right input *(B)* must pass through it to reach its output. For this purpose, the *B15to0* control input of ALU is issued. Once the data reaches the ALU output, it becomes available at the input of the Register File. Issuing *RFLwrite* and *RFHwrite* cause data to be written into the destination into *RegisterFile*.

The partial code of Figure 14.13 shows assignment of *exec2* to *Nstate* if the instruction we are executing has a shadow. Otherwise, signals for incrementing the Program Counter are issued and the next state is set to *fetch.*

The execution discussed here applies to most SAYEH instructions. However, instructions that require memory access, e.g., *lda,* require an extra clock for reading the memory. The first part of the execution of *lda* is shown in Figure 14.13. As shown, for the execution of this instruction, the address is read from Register File and put on the address bus. At the same time, *ReadMem* is issued to initiate the memory read process.

The next state for execution of *lda* after *exec1* is *exec1lda* shown in the partial code of Figure 14.14. In this state, *ReadMem* continues to be issued and state remains in *exec1lda* until *memDataReady* becomes **1**. In this case, memory data that is available on *Databus* will be clocked into *RegisterFile* by issuing *RFLwrite* and *RFHwrite.*

Executions of other SAYEH instructions are similar to the examples we discussed. The complete Verilog code of SAYEH controller is over 800 lines and is included on the CD that accompanies this book.

14.2.4 Complete SAYEH Processor

The top-level Verilog code of SAYEH that is shown in Figure 14.15 consists of instantiation of *DataPath* and *controller* modules. In *Sayeh* module, control signal outputs of *controller* are wired to the similarly named signals of *DataPath.* The ports of the processor are according to the block diagram of Figure 14.1.

```
module Sayeh ( clk, ReadMem, WriteMem, ReadIO, WriteIO,
   Databus, Addressbus, ExternalReset, MemDataready);
input clk;
output ReadMem, WriteMem, ReadIO, WriteIO;
inout[15:0] Databus;
output [15: 0] Addressbus;
input ExternalReset, MemDataready;

wire [15:0] Instruction;
wire esetPC, PCplusI, PCplus1, RplusI, Rplus0,
 . . .
DataPath dp ( clk, Databus, Addressbus,
   ResetPC, PCplusI, PCplus1, RplusI, Rplus0, . . . );

controller Ctrl ( ExternalReset, clk,
   ResetPC, PCplusI, PCplus1, RplusI, Rplus0, . . . );

endmodule
```

Figure 14.15 SAYEH Top Level Description

14.3 SAYEH Testing

Because of the complexity of this design, it is best to test it with an HDL simulator and a high level testbench. Tools for generation and application of test data and monitoring and generation of output data are provided in HDL simulators. These tools together with ability to describe high level testbenches provide an efficient test and debugging environment for HDL based designs.

The testbench for SAYEH is shown in Figure 14.16. The use of external files for reading and writing test data are demonstrated by this example. As shown in this figure, *SayehRAM* that is a memory of 1024 16-bits words is declared in this testbench. The testbench reads test data that is the memory image of our processor in this file and when the test is completed contents of this memory are written into another external file. The input file is *SayehRAM.hex* and the output file is *OutputRAM.hex*. Contents of both files are in hexadecimal. 16-bit hexadecimal codes in these files represent memory data starting from location 0.

The first **initial** block is labeled *IOfiles*. This block opens the *OutputRAM.hex* output file for later writing and reads the contents of *SayehRAM.hex* into the declared *SayehRAM* memory. Reading the input file (memory image) is done by the **$readmemh** system task. This task expects data in the file to match the word length of the memory it is writing into.

An **always** block shown in *SayehTest* testbench generates a periodic signal on the circuit clock input.

The next procedural block shown in this testbench is an **initial** block that is labeled *RunCPU*. This block applies the resetting signal, runs the CPU for 370,000 nanoseconds, and when this time expires, it writes all 1024 words of *SayehRAM* into *OutputRAM.hex* external file. Note here that the **$fopen** statement in the *IOfiles* block made *memout* a file handler for the output file. The **$stop** statement in *RunCPU* block stops the simulation after the memory image has been written.

The **always** procedural block that is labeled *MemoryRead* handles reading data from *SayehRAM* when requested by the CPU. When *ReadMem* is issued by the CPU, the testbench issues *MemDataready* and places data from *SayehRAM* at the *Addressbus* location on *MemoryData*. At all other times, *MemoryData* bus is at the high-impedance state. This is done because *MemoryData* connects to *Databus* hat is a bi-directional bus.

The **always** block that appears next in Figure 14.16 handles writing data that appears on *Databus* into *SayehRAM*. This block has delays to allow signals from the CPU to stabilize.

This testbench allows for any SAYEH program to be loaded into the CPU memory and executed. Out testing of this processor consisted of an instruction based testing as well as several programs. For the instruction testing we applied independent instructions and monitored internal registers of SAYEH. For example, F205, that is the hex code for *"mil r2 05",* loads 05 into R2 of the Register File. Similarly, 0204 is the packing of two 8-bit instructions that set the zero and carry flags. An initial testing of a CPU requires verification of individual CPU instructions.

A more elaborate test program is discussed in the next section.

```
`timescale 1 ns /1 ns

module SayehTest ();

reg clk, ExternalReset, MemDataready;
reg [15:0] MemoryData;
wire [15:0] Databus, Addressbus;
wire ReadMem, WriteMem, ReadIO, WriteIO;

reg [15:0] SayehRAM [0:1023];

integer memout;
initial begin : IOfiles
   memout = $fopen ("OutputRAM.hex");
   $readmemh ("SayehRAM.hex", SayehRAM);
   clk = 0; ExternalReset = 0; MemDataready = 0;
   MemoryData = 16'bZ;
end

always #20 clk = ~clk;

integer i;
initial begin : RunCPU
   #05 ExternalReset = 1; #81 ExternalReset = 0;
   #370000;
   for (i=0; i<= 1023; i=i+1)
      $fdisplay (memout, "%h: %h", i, SayehRAM[i]);
   $stop;
end

always @(negedge clk) begin : MemoryRead
   if (ReadMem) begin
      #1 MemDataready = 1 ;
      MemoryData = SayehRAM [Addressbus];
   end else begin
      #1 MemDataready = 0;
      MemoryData = 16'hZZZZ;
   end
end

always @(negedge clk) begin : MemoryWrite
  #1 if (WriteMem) #1 SayehRAM [Addressbus] = Databus;
end

assign Databus = MemoryData;

Sayeh U1 (clk, ReadMem, WriteMem, ReadIO, WriteIO,
              Databus, Addressbus, ExternalReset, MemDataready);

endmodule
```

Figure 14.16 A Testbench for SAYEH

14.4 Sorting Test Program

Figure 14.17 shows a sorting program for SAYEH. This program reads data starting from the CPU memory and sorts them in descending order. The number of data item to sort is in location 768 and data begins in the next memory location. This sorting program uses two loops for the sorting to be done. When completed, the CPU is put into the halt state.

```
0000 mil  r0 00     r0 = 768    starting address  in memory
0001 mih r0 03
0002 lda r1 r0      r1 =        total number of elements
0003 awp 5
0004 mil  r0 01     r5=1        for adding with index each time
0005 mih r0 00
0006 cwp
0006 add r1 r0      r1=         limit for final r4
0007 mvr r2 r1
0008 awp 2
0009 sub r0 r3      r2=         limit for index r3
0009 cwp
000A mvr r3 r0      r3=         outer index
000A nop
000B cwp
000B cmp r3 r2                  check if outer index is equal to its limit
000C brz 19                     branch to 0025 if zero
000D awp 3
000E add r0 r2      r3=r3+1     increment outer index
000E mvr r1 r0      r4=r3       set inner index to outer index as initial
000F cwp
0010 awp 1
0011 cmp r3 r0                  check if inner index is equal to its limit
0012 brz 10                     branch to 0022 if zero
0013 awp 2
0014 lda r3 r0      r6=(r3)
0015 awp 1
0016 add r0 r1      r4=r4+r5    increment inner index
0016 lda r3 r0      r7=(r4)
0017 cmp r2 r3                  check if r6 is greater than r7
0018 brc 07                     branch to 001F if carry
0019 lda r1 r0      r5=(r4)     r5 as an temperary register
0019 sta r0 r2      (r4)=r6
001A cwp
001B awp 3
001C sta r0 r2      (r3)=r5
001D mil  r2 01
001E mih r2 00      r5=1        for adding with index each time
001F cwp
0020 awp 5
0021 jpa r0 0E                  jump to 000F
0022 cwp
0023 awp 5
0024 jpa r0 0A                  jump to 000B
0025 hlt
```

Figure 14.17 Sorting Program for SAYEH

The program shown in Figure 14.17 is translated into its hexadecimal equivalent and is put in *SayehRAM.hex* file. As discussed in the previous section, SAYEH testbench reads instructions from this file and applies to the CPU.

14.5 FPGA Programming

The CPU described in this chapter has been programmed into the FLEX device of an Altera UP2. We used a RAM from Altera's megafunctions and configured it as a memory of 1024 16-bit words. The number of logic elements used by this CPU was 1,125, which is 30% of the available LEs. Memory bits used was 16,384, which is 44% of the available memory bits. This usage indicates that we can form a complete system with a keyboard and VGA output on a FLEX 10K of UP2.

14.6 Summary

This chapter showed design, testing and implementation of a complete CPU. This design put all that we have covered in this book into one package. The design is complete and typical of any large system with a complex controller and data path. Use of the synthesizable subset of Verilog for development of a design for FPGA programming was shown. On the other hand, utilization of behavioral constructs of Verilog was demonstrated in developing a testbench for our processor.