7.6.1 Single-Cycle Processor

The main modules of the single-cycle processor module are given in the following HDL examples.

HDL Example 7.1 SINGLE-CYCLE PROCESSOR

SystemVerilog

```
module arm(input  logic         clk, reset,
           output logic  [31:0] PC,
           input  logic  [31:0] Instr,
           output logic         MemWrite,
           output logic  [31:0] ALUResult, WriteData,
           input  logic  [31:0] ReadData);

  logic [3:0]  ALUFlags;
  logic        RegWrite,
               ALUSrc, MemtoReg, PCSrc;
  logic [1:0]  RegSrc, ImmSrc, ALUControl;

  controller c(clk, reset, Instr[31:12], ALUFlags,
               RegSrc, RegWrite, ImmSrc,
               ALUSrc, ALUControl,
               MemWrite, MemtoReg, PCSrc);
  datapath dp(clk, reset,
              RegSrc, RegWrite, ImmSrc,
              ALUSrc, ALUControl,
              MemtoReg, PCSrc,
              ALUFlags, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- single cycle processor
  port(clk, reset:        in  STD_LOGIC;
       PC:                out STD_LOGIC_VECTOR(31 downto 0);
       Instr:             in  STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:          out STD_LOGIC;
       ALUResult, WriteData: out STD_LOGIC_VECTOR(31 downto 0);
       ReadData:          in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
  component controller
  port(clk, reset:     in  STD_LOGIC;
       Instr:          in  STD_LOGIC_VECTOR(31 downto 12);
       ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
       RegSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:       out STD_LOGIC;
       ImmSrc:         out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:         out STD_LOGIC;
       ALUControl:     out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:       out STD_LOGIC;
       MemtoReg:       out STD_LOGIC;
       PCSrc:          out STD_LOGIC);
  end component;
  component datapath
    port(clk, reset:      in     STD_LOGIC;
      RegSrc:             in     STD_LOGIC_VECTOR(1 downto 0);
      RegWrite:           in     STD_LOGIC;
      ImmSrc:             in     STD_LOGIC_VECTOR(1 downto 0);
      ALUSrc:             in     STD_LOGIC;
      ALUControl:         in     STD_LOGIC_VECTOR(1 downto 0);
      MemtoReg:           in     STD_LOGIC;
      PCSrc:              in     STD_LOGIC;
      ALUFlags:           out    STD_LOGIC_VECTOR(3 downto 0);
      PC:                 buffer STD_LOGIC_VECTOR(31 downto 0);
      Instr:              in     STD_LOGIC_VECTOR(31 downto 0);
      ALUResult, WriteData:buffer STD_LOGIC_VECTOR(31 downto 0);
      ReadData:           in     STD_LOGIC_VECTOR(31 downto 0));
    end component;
    signal RegWrite, ALUSrc, MemtoReg, PCSrc: STD_LOGIC;
    signal RegSrc, ImmSrc, ALUControl: STD_LOGIC_VECTOR
                                        (1 downto 0);
    signal ALUFlags: STD_LOGIC_VECTOR(3 downto 0);
begin
  cont: controller port map(clk, reset, Instr(31 downto 12),
                            ALUFlags, RegSrc, RegWrite,
                            ImmSrc, ALUSrc, ALUControl,
                            MemWrite, MemtoReg, PCSrc);
  dp: datapath port map(clk, reset, RegSrc, RegWrite, ImmSrc,
                        ALUSrc, ALUControl, MemtoReg, PCSrc,
                        ALUFlags, PC, Instr, ALUResult,
                        WriteData, ReadData);
end;
```

HDL Example 7.2 CONTROLLER

SystemVerilog

```
module controller(input  logic         clk, reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic [1:0]   RegSrc,
                  output logic         RegWrite,
                  output logic [1:0]   ImmSrc,
                  output logic         ALUSrc,
                  output logic [1:0]   ALUControl,
                  output logic         MemWrite, MemtoReg,
                  output logic         PCSrc);
  logic [1:0] FlagW;
  logic       PCS, RegW, MemW;

  decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
              FlagW, PCS, RegW, MemW,
              MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, RegW, MemW,
               PCSrc, RegWrite, MemWrite);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- single cycle control
  port(clk, reset:    in  STD_LOGIC;
       Instr:         in  STD_LOGIC_VECTOR(31 downto 12);
       ALUFlags:      in  STD_LOGIC_VECTOR(3 downto 0);
       RegSrc:        out STD_LOGIC_VECTOR(1 downto 0);
       RegWrite:      out STD_LOGIC;
       ImmSrc:        out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrc:        out STD_LOGIC;
       ALUControl:    out STD_LOGIC_VECTOR(1 downto 0);
       MemWrite:      out STD_LOGIC;
       MemtoReg:      out STD_LOGIC;
       PCSrc:         out STD_LOGIC);
end;

architecture struct of controller is
  component decoder
    port(Op:                in  STD_LOGIC_VECTOR(1 downto 0);
         Funct:             in  STD_LOGIC_VECTOR(5 downto 0);
         Rd:                in  STD_LOGIC_VECTOR(3 downto 0);
         FlagW:             out STD_LOGIC_VECTOR(1 downto 0);
         PCS, RegW, MemW:   out STD_LOGIC;
         MemtoReg, ALUSrc:  out STD_LOGIC;
         ImmSrc, RegSrc:    out STD_LOGIC_VECTOR(1 downto 0);
         ALUControl:        out STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component condlogic
    port(clk, reset:       in  STD_LOGIC;
         Cond:             in  STD_LOGIC_VECTOR(3 downto 0);
         ALUFlags:         in  STD_LOGIC_VECTOR(3 downto 0);
         FlagW:            in  STD_LOGIC_VECTOR(1 downto 0);
         PCS, RegW, MemW:  in  STD_LOGIC;
         PCSrc, RegWrite:  out STD_LOGIC;
         MemWrite:         out STD_LOGIC);
  end component;
  signal FlagW: STD_LOGIC_VECTOR(1 downto 0);
  signal PCS, RegW, MemW: STD_LOGIC;
begin
  dec: decoder port map(Instr(27 downto 26), Instr(25 downto 20),
                        Instr(15 downto 12), FlagW, PCS,
                        RegW, MemW, MemtoReg, ALUSrc, ImmSrc,
                        RegSrc, ALUControl);
  cl: condlogic port map(clk, reset, Instr(31 downto 28),
                         ALUFlags, FlagW, PCS, RegW, MemW,
                         PCSrc, RegWrite, MemWrite);
end;
```

HDL Example 7.3 DECODER

SystemVerilog

```
module decoder(input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic       PCS, RegW, MemW,
               output logic       MemtoReg, ALUSrc,
               output logic [1:0] ImmSrc, RegSrc, ALUControl);

  logic [9:0] controls;
  logic       Branch, ALUOp;

  // Main Decoder
  always_comb
      casex(Op)
                                 // Data-processing immediate
        2'b00: if (Funct[5])     controls = 10'b0000101001;
                                 // Data-processing register
               else              controls = 10'b0000001001;
                                 // LDR
        2'b01: if (Funct[0])     controls = 10'b0001111000;
                                 // STR
               else              controls = 10'b1001110100;
                                 // B
        2'b10:                   controls = 10'b0110100010;
                                 // Unimplemented
        default:                 controls = 10'bx;
      endcase

  assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
    RegW, MemW, Branch, ALUOp} = controls;

  // ALU Decoder
  always_comb
  if (ALUOp) begin              // which DP Instr?
    case(Funct[4:1])
        4'b0100: ALUControl = 2'b00; // ADD
        4'b0010: ALUControl = 2'b01; // SUB
        4'b0000: ALUControl = 2'b10; // AND
        4'b1100: ALUControl = 2'b11; // ORR
        default: ALUControl = 2'bx;  // unimplemented
    endcase

    // update flags if S bit is set (C & V only for arith)
    FlagW[1]      = Funct[0];
    FlagW[0]      = Funct[0] &
      (ALUControl == 2'b00 | ALUControl == 2'b01);
  end else begin
    ALUControl = 2'b00; // add for non-DP instructions
    FlagW      = 2'b00; // don't update Flags
  end

  // PC Logic
  assign PCS  = ((Rd == 4'b1111) & RegW) | Branch;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity decoder is -- main control decoder
  port(Op:               in  STD_LOGIC_VECTOR(1 downto 0);
       Funct:            in  STD_LOGIC_VECTOR(5 downto 0);
       Rd:               in  STD_LOGIC_VECTOR(3 downto 0);
       FlagW:            out STD_LOGIC_VECTOR(1 downto 0);
       PCS, RegW, MemW:  out STD_LOGIC;
       MemtoReg, ALUSrc: out STD_LOGIC;
       ImmSrc, RegSrc:   out STD_LOGIC_VECTOR(1 downto 0);
       ALUControl:       out STD_LOGIC_VECTOR(1 downto 0));
end;
architecture behave of decoder is
  signal controls:     STD_LOGIC_VECTOR(9 downto 0);
  signal ALUOp, Branch: STD_LOGIC;
  signal op2:          STD_LOGIC_VECTOR(3 downto 0);
begin
  op2 <= (Op, Funct(5), Funct(0));
  process(all) begin -- Main Decoder
    case? (op2) is
      when "000-"  => controls <= "0000001001";
      when "001-"  => controls <= "0000101001";
      when "01-0"  => controls <= "1001110100";
      when "01-1"  => controls <= "0001111000";
      when "10--"  => controls <= "0110100010";
      when others  => controls <= "----------";
    end case?;
  end process;

  (RegSrc, ImmSrc, ALUSrc, MemtoReg, RegW, MemW,
  Branch, ALUOp) <= controls;

  process(all) begin -- ALU Decoder
  if (ALUOp) then
    case Funct(4 downto 1) is
      when "0100" => ALUControl <= "00"; -- ADD
      when "0010" => ALUControl <= "01"; -- SUB
      when "0000" => ALUControl <= "10"; -- AND
      when "1100" => ALUControl <= "11"; -- ORR
      when others => ALUControl <= "--"; -- unimplemented
    end case;
    FlagW(1)  <= Funct(0);
    FlagW(0)  <= Funct(0) and (not ALUControl(1));
  else
    ALUControl  <= "00";
    FlagW  <= "00";
  end if;
  end process;

  PCS  <= ((and Rd) and RegW) or Branch;
end;
```

HDL Example 7.4 CONDITIONAL LOGIC

SystemVerilog

```
module condlogic(input   logic         clk, reset,
                 input   logic  [3:0]  Cond,
                 input   logic  [3:0]  ALUFlags,
                 input   logic  [1:0]  FlagW,
                 input   logic         PCS, RegW, MemW,
                 output  logic         PCSrc, RegWrite,
                                       MemWrite);

  logic  [1:0]  FlagWrite;
  logic  [3:0]  Flags;
  logic         CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  // write controls are conditional
  condcheck cc(Cond,  Flags,  CondEx);
  assign FlagWrite =  FlagW  &  {2{CondEx}};
  assign RegWrite  =  RegW   &  CondEx;
  assign MemWrite  =  MemW   &  CondEx;
  assign PCSrc     =  PCS    &  CondEx;
endmodule

module condcheck(input  logic  [3:0]  Cond,
                 input  logic  [3:0]  Flags,
                 output logic         CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;              // EQ
      4'b0001: CondEx = ~zero;             // NE
      4'b0010: CondEx = carry;             // CS
      4'b0011: CondEx = ~carry;            // CC
      4'b0100: CondEx = neg;               // MI
      4'b0101: CondEx = ~neg;              // PL
      4'b0110: CondEx = overflow;          // VS
      4'b0111: CondEx = ~overflow;         // VC
      4'b1000: CondEx = carry & ~zero;     // HI
      4'b1001: CondEx = ~(carry & ~zero);  // LS
      4'b1010: CondEx = ge;                // GE
      4'b1011: CondEx = ~ge;               // LT
      4'b1100: CondEx = ~zero & ge;        // GT
      4'b1101: CondEx = ~(~zero & ge);     // LE
      4'b1110: CondEx = 1'b1;              // Always
      default: CondEx = 1'bx;              // undefined
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condlogic is -- Conditional logic
  port(clk, reset:       in  STD_LOGIC;
       Cond:             in  STD_LOGIC_VECTOR(3 downto 0);
       ALUFlags:         in  STD_LOGIC_VECTOR(3 downto 0);
       FlagW:            in  STD_LOGIC_VECTOR(1 downto 0);
       PCS, RegW, MemW:  in  STD_LOGIC;
       PCSrc, RegWrite:  out STD_LOGIC;
       MemWrite:         out STD_LOGIC);
end;

architecture behave of condlogic is
  component condcheck
  port(Cond:       in  STD_LOGIC_VECTOR(3 downto 0);
       Flags:      in  STD_LOGIC_VECTOR(3 downto 0);
       CondEx:     out STD_LOGIC);
  end component;
  component flopenr generic(width: integer);
  port(clk, reset, en: in  STD_LOGIC;
       d:     in  STD_LOGIC_VECTOR (width-1 downto 0);
       q:     out STD_LOGIC_VECTOR (width-1 downto 0));
  end component;
  signal FlagWrite: STD_LOGIC_VECTOR(1 downto 0);
  signal Flags:     STD_LOGIC_VECTOR(3 downto 0);
  signal CondEx:    STD_LOGIC;
begin
  flagreg1: flopenr generic map(2)
    port map(clk, reset, FlagWrite(1),
             ALUFlags(3 downto 2), Flags(3 downto 2));
  flagreg0: flopenr generic map(2)
    port map(clk, reset, FlagWrite(0),
             ALUFlags(1 downto 0), Flags(1 downto 0));
  cc: condcheck port map(Cond, Flags, CondEx);

  FlagWrite  <=  FlagW and (CondEx, CondEx);
  RegWrite   <=  RegW  and CondEx;
  MemWrite   <=  MemW  and CondEx;
  PCSrc      <=  PCS   and CondEx;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity condcheck is
  port(Cond:     in  STD_LOGIC_VECTOR(3 downto 0);
    Flags:       in  STD_LOGIC_VECTOR(3 downto 0);
    CondEx:      out STD_LOGIC);
end;

architecture behave of condcheck is
  signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
  (neg, zero, carry, overflow) <= Flags;
  ge <= (neg xnor overflow);

  process(all) begin -- Condition checking
    case Cond is
      when "0000"  =>  CondEx  <=  zero;
      when "0001"  =>  CondEx  <=  not zero;
      when "0010"  =>  CondEx  <=  carry;
      when "0011"  =>  CondEx  <=  not carry;
      when "0100"  =>  CondEx  <=  neg;
      when "0101"  =>  CondEx  <=  not neg;
      when "0110"  =>  CondEx  <=  overflow;
      when "0111"  =>  CondEx  <=  not overflow;
      when "1000"  =>  CondEx  <=  carry and (not zero);
      when "1001"  =>  CondEx  <=  not(carry and (not zero));
      when "1010"  =>  CondEx  <=  ge;
      when "1011"  =>  CondEx  <=  not ge;
      when "1100"  =>  CondEx  <=  (not zero) and ge;
      when "1101"  =>  CondEx  <=  not ((not zero) and ge);
      when "1110"  =>  CondEx  <=  '1';
      when others  =>  CondEx  <=  '-';
    end case;
  end process;
end;
```

HDL Example 7.5 DATAPATH

SystemVerilog

```
module datapath(input  logic         clk, reset,
                input  logic  [1:0]  RegSrc,
                input  logic         RegWrite,
                input  logic  [1:0]  ImmSrc,
                input  logic         ALUSrc,
                input  logic  [1:0]  ALUControl,
                input  logic         MemtoReg,
                input  logic         PCSrc,
                output logic  [3:0]  ALUFlags,
                output logic  [31:0] PC,
                input  logic  [31:0] Instr,
                output logic  [31:0] ALUResult, WriteData,
                input  logic  [31:0] ReadData);

  logic  [31:0]  PCNext, PCPlus4, PCPlus8;
  logic  [31:0]  ExtImm, SrcA, SrcB, Result;
  logic  [3:0]   RA1, RA2;

  // next PC logic
  mux2 #(32)   pcmux(PCPlus4, Result, PCSrc, PCNext);
  flopr #(32)  pcreg(clk, reset, PCNext, PC);
  adder #(32)  pcadd1(PC, 32'b100, PCPlus4);
  adder #(32)  pcadd2(PCPlus4, 32'b100, PCPlus8);

  // register file logic
  mux2 #(4)  ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
  mux2 #(4)  ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
  regfile    rf(clk, RegWrite, RA1, RA2,
                Instr[15:12], Result, PCPlus8,
                SrcA, WriteData);
  mux2 #(32) resmux(ALUResult, ReadData, MemtoReg, Result);
  extend     ext(Instr[23:0], ImmSrc, ExtImm);

  // ALU logic
  mux2 #(32)  srcbmux(WriteData, ExtImm, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
  port(clk, reset:         in     STD_LOGIC;
     RegSrc:               in     STD_LOGIC_VECTOR(1 downto 0);
     RegWrite:             in     STD_LOGIC;
     ImmSrc:               in     STD_LOGIC_VECTOR(1 downto 0);
     ALUSrc:               in     STD_LOGIC;
     ALUControl:           in     STD_LOGIC_VECTOR(1 downto 0);
     MemtoReg:             in     STD_LOGIC;
     PCSrc:                in     STD_LOGIC;
     ALUFlags:             out    STD_LOGIC_VECTOR(3 downto 0);
     PC:                   buffer STD_LOGIC_VECTOR(31 downto 0);
     Instr:                in     STD_LOGIC_VECTOR(31 downto 0);
     ALUResult, WriteData: buffer STD_LOGIC_VECTOR(31 downto 0);
     ReadData:             in     STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
  component alu
  port(a, b:         in     STD_LOGIC_VECTOR(31 downto 0);
       ALUControl:   in     STD_LOGIC_VECTOR(1 downto 0);
       Result:       buffer STD_LOGIC_VECTOR(31 downto 0);
       ALUFlags:     out    STD_LOGIC_VECTOR(3 downto 0));
  end component;
  component regfile
  port(clk:            in  STD_LOGIC;
       we3:            in  STD_LOGIC;
       ra1, ra2, wa3:  in  STD_LOGIC_VECTOR(3 downto 0);
       wd3, r15:       in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:       out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
  port(a, b:  in  STD_LOGIC_VECTOR(31 downto 0);
       y:     out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component extend
  port(Instr:   in  STD_LOGIC_VECTOR(23 downto 0);
       ImmSrc:  in  STD_LOGIC_VECTOR(1 downto 0);
       ExtImm:  out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
  port(clk, reset:  in  STD_LOGIC;
       d:           in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:           out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
  port(d0, d1:  in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:       in  STD_LOGIC;
       y:       out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal PCNext, PCPlus4,
         PCPlus8:         STD_LOGIC_VECTOR(31 downto 0);
  signal ExtImm, Result:  STD_LOGIC_VECTOR(31 downto 0);
  signal SrcA, SrcB:      STD_LOGIC_VECTOR(31 downto 0);
  signal RA1, RA2:        STD_LOGIC_VECTOR(3 downto 0);
begin
  -- next PC logic
  pcmux:  mux2 generic map(32)
           port map(PCPlus4, Result, PCSrc, PCNext);
  pcreg: flopr generic map(32) port map(clk, reset, PCNext, PC);
  pcadd1: adder port map(PC, X"00000004", PCPlus4);
  pcadd2: adder port map(PCPlus4, X"00000004", PCPlus8);

  -- register file logic
  ra1mux: mux2 generic map (4)
    port map(Instr(19 downto 16), "1111", RegSrc(0), RA1);
  ra2mux: mux2 generic map (4) port map(Instr(3 downto 0),
                            Instr(15 downto 12), RegSrc(1), RA2);
  rf: regfile port map(clk, RegWrite, RA1, RA2,
                          Instr(15 downto 12), Result,
                          PCPlus8, SrcA, WriteData);
  resmux: mux2 generic map(32)
    port map(ALUResult, ReadData, MemtoReg, Result);
  ext: extend port map(Instr(23 downto 0), ImmSrc, ExtImm);

  -- ALU logic
  srcbmux: mux2 generic map(32)
    port map(WriteData, ExtImm, ALUSrc, SrcB);
  i_alu:  alu port map(SrcA, SrcB, ALUControl, ALUResult,
                          ALUFlags);
end;
```

7.6.2 Generic Building Blocks

This section contains generic building blocks that may be useful in any digital system, including a register file, adder, flip-flops, and a 2:1 multiplexer. The HDL for the ALU is left to Exercises 5.11 and 5.12.

HDL Example 7.6 REGISTER FILE

SystemVerilog

```
module regfile(input  logic          clk,
               input  logic          we3,
               input  logic  [3:0]   ra1, ra2, wa3,
               input  logic  [31:0]  wd3, r15,
               output logic  [31:0]  rd1, rd2);

  logic [31:0] rf[14:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 15 reads PC+8 instead

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
  assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:           in  STD_LOGIC;
       we3:           in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
       wd3, r15:      in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of
    STD_LOGIC_VECTOR(31 downto 0);
  signal mem: ramtype;
begin
  process(clk) begin
  if rising_edge(clk) then
    if we3 = '1' then mem(to_integer(wa3)) <= wd3;
    end if;
  end if;
  end process;
  process(all) begin
    if (to_integer(ra1) = 15) then rd1 <= r15;
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 15) then rd2 <= r15;
    else rd2 <= mem(to_integer(ra2));
    end if;
  end process;
end;
```

HDL Example 7.7 ADDER

SystemVerilog

```
module adder #(parameter WIDTH=8)
              (input  logic [WIDTH-1:0] a, b,
               output logic [WIDTH-1:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

HDL Example 7.8 IMMEDIATE EXTENSION

SystemVerilog

```
module extend(input  logic  [23:0]  Instr,
              input  logic  [1:0]   ImmSrc,
              output logic  [31:0]  ExtImm);

  always_comb
    case(ImmSrc)
                // 8-bit unsigned immediate
      2'b00:    ExtImm = {24'b0, Instr[7:0]};
                // 12-bit unsigned immediate
      2'b01:    ExtImm = {20'b0, Instr[11:0]};
                // 24-bit two's complement shifted branch
      2'b10:    ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
      default:  ExtImm = 32'bx; // undefined
  endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
  port(Instr:   in  STD_LOGIC_VECTOR(23 downto 0);
       ImmSrc:  in  STD_LOGIC_VECTOR(1 downto 0);
       ExtImm:  out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
  process(all) begin
    case ImmSrc is
      when "00"  => ExtImm <= (X"000000", Instr(7 downto 0));
      when "01"  => ExtImm <= (X"00000", Instr(11 downto 0));
      when "10"  => ExtImm <= (Instr(23), Instr(23),
                               Instr(23), Instr(23),
                               Instr(23), Instr(23),
                               Instr(23 downto 0), "00");
      when others  =>  ExtImm  <=  X"--------";
    end case;
  end process;
end;
```

HDL Example 7.9 RESETTABLE FLIP-FLOP

SystemVerilog

```
module flopr #(parameter WIDTH = 8)
             (input  logic              clk, reset,
              input  logic  [WIDTH-1:0] d,
              output logic  [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)  q <= 0;
    else        q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is -- flip-flop with synchronous reset
  generic(width: integer);
  port(clk, reset:  in  STD_LOGIC;
       d:           in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:           out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then  q <= (others => '0');
    elsif rising_edge(clk) then
        q <= d;
    end if;
  end process;
end;
```

HDL Example 7.10 RESETTABLE FLIP-FLOP WITH ENABLE

SystemVerilog

```
module flopenr #(parameter WIDTH = 8)
              (input  logic              clk, reset, en,
               input  logic  [WIDTH-1:0]  d,
               output logic  [WIDTH-1:0]  q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en)  q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopenr is -- flip-flop with enable and synchronous reset
  generic(width: integer);
  port(clk, reset, en: in  STD_LOGIC;
       d:      in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopenr is
begin
  process(clk, reset) begin
    if reset then  q <= (others => '0');
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

HDL Example 7.11 2:1 MULTIPLEXER

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
           (input   logic  [WIDTH-1:0]  d0, d1,
            input   logic               s,
            output  logic  [WIDTH-1:0]  y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
  generic(width: integer);
  port(d0, d1:  in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:       in  STD_LOGIC;
       y:       out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;
```

7.6.3 Testbench

The testbench loads a program into the memories. The program in Figure 7.60 exercises all of the instructions by performing a computation that should produce the correct result only if all of the instructions are functioning correctly. Specifically, the program will write the value 7 to address 100 if it runs correctly, but it is unlikely to do so if the hardware is buggy. This is an example of *ad hoc* testing.

The machine code is stored in a hexadecimal file called memfile.dat, which is loaded by the testbench during simulation. The file consists of the machine code for the instructions, one instruction per line. The testbench, top-level ARM module, and external memory HDL code are given in the following examples. The memories in this example hold 64 words each.

```
ADDR        PROGRAM           ; COMMENTS                       BINARY MACHINE CODE                       HEX CODE
00  MAIN    SUB R0, R15, R15  ; R0 = 0                         1110 000 0010 0 1111 0000 0000 0000 1111  E04F000F
04          ADD R2, R0, #5    ; R2 = 5                         1110 001 0100 0 0000 0010 0000 0000 0101  E2802005
08          ADD R3, R0, #12   ; R3 = 12                        1110 001 0100 0 0000 0011 0000 0000 1100  E280300C
0C          SUB R7, R3, #9    ; R7 = 3                         1110 001 0010 0 0011 0111 0000 0000 1001  E2437009
10          ORR R4, R7, R2    ; R4 = 3 OR 5 = 7                1110 000 1100 0 0111 0100 0000 0000 0010  E1874002
14          AND R5, R3, R4    ; R5 = 12 AND 7 = 4              1110 000 0000 0 0011 0101 0000 0000 0100  E0035004
18          ADD R5, R5, R4    ; R5 = 4 + 7 = 11                1110 000 0100 0 0101 0101 0000 0000 0100  E0855004
1C          SUBS R8, R5, R7   ; R8 = 11 - 3 = 8, set Flags     1110 000 0010 1 0101 1000 0000 0000 0111  E0558007
20          BEQ END           ; shouldn't be taken             0000 1010 0000  0000 0000 0000 0000 1100  0A00000C
24          SUBS R8, R3, R4   ; R8 = 12 - 7  = 5               1110 000 0010 1 0011 1000 0000 0000 0100  E0538004
28          BGE AROUND        ; should be taken                1010 1010 0000  0000 0000 0000 0000 0000  AA000000
2C          ADD R5, R0, #0    ; should be skipped              1110 001 0100 0 0000 0101 0000 0000 0000  E2805000
30 AROUND   SUBS R8, R7, R2   ; R8 = 3 - 5 = -2, set Flags     1110 000 0010 1 0111 1000 0000 0000 0010  E0578002
34          ADDLT R7, R5, #1  ; R7 = 11 + 1 = 12               1011 001 0100 0 0101 0111 0000 0000 0001  B2857001
38          SUB R7, R7, R2    ; R7 = 12 - 5 = 7                1110 000 0010 0 0111 0111 0000 0000 0010  E0477002
3C          STR R7, [R3, #84] ; mem[12+84] = 7                 1110 010 1100 0 0011 0111 0000 0101 0100  E5837054
40          LDR R2, [R0, #96] ; R2 = mem[96] = 7               1110 010 1100 1 0000 0010 0000 0110 0000  E5902060
44          ADD R15, R15, R0  ; PC = PC+8 (skips next)         1110 000 0100 0 1111 1111 0000 0000 0000  E08FF000
48          ADD R2, R0, #14   ; shouldn't happen               1110 001 0100 0 0000 0010 0000 0000 0001  E280200E
4C          B END             ; always taken                   1110 1010 0000  0000 0000 0000 0000 0001  EA000001
50          ADD R2, R0, #13   ; shouldn't happen               1110 001 0100 0 0000 0010 0000 0000 0001  E280200D
54          ADD R2, R0, #10   ; shouldn't happen               1110 001 0100 0 0000 0010 0000 0000 0001  E280200A
58 END      STR R2, [R0, #100] ; mem[100] = 7                  1110 010 1100 0 0000 0010 0000 0101 0100  E5802064
```

Figure 7.60 Assembly and machine code for test program

HDL Example 7.12 TESTBENCH

SystemVerilog

```
module testbench();
  logic        clk;
  logic        reset;
  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  // initialize test
  initial
  begin
    reset <= 1; # 22; reset <= 0;
  end

  // generate clock to sequence tests
  always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
  end
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset:          in  STD_LOGIC;
         WriteData, DataAadr: out STD_LOGIC_VECTOR(31 downto 0);
         MemWrite:            out STD_LOGIC);
  end component;
  signal WriteData, DataAdr:    STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset,  MemWrite: STD_LOGIC;
begin
  -- instantiate device to be tested
  dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

  -- generate clock with 10 ns period
  process begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
  end process;
  // check that 7 gets written to address 0x64
  // at end of program
  always @(negedge clk)
  begin
    if(MemWrite) begin
      if(DataAdr === 100 & WriteData === 7) begin
        $display("Simulation succeeded");
        $stop;
      end else if (DataAdr !== 96) begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

```
  -- generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 7 gets written to address 0x64
  -- at end of program
  process (clk) begin
    if (clk'event and clk = '0' and MemWrite = '1') then
      if (to_integer(DataAdr) = 100 and
         to_integer(WriteData) = 7) then
        report "NO ERRORS: Simulation succeeded" severity
        failure;
      elsif (DataAdr /= 96) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;
```

HDL Example 7.13 TOP-LEVEL MODULE

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;

  // instantiate processor and memories
  arm arm(clk, reset, PC, Instr, MemWrite, DataAdr,
          WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity top is -- top-level design for testing
  port(clk, reset:        in     STD_LOGIC;
       WriteData, DataAdr: buffer STD_LOGIC_VECTOR(31 downto 0);
       MemWrite:          buffer STD_LOGIC);
end;

architecture test of top is
  component arm
    port(clk, reset:         in  STD_LOGIC;
      PC:                    out STD_LOGIC_VECTOR(31 downto 0);
      Instr:                 in  STD_LOGIC_VECTOR(31 downto 0);
      MemWrite:              out STD_LOGIC;
      ALUResult, WriteData:out STD_LOGIC_VECTOR(31 downto 0);
      ReadData:              in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
  port(a:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:  out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
  port(clk, we:  in   STD_LOGIC;
    a, wd:       in   STD_LOGIC_VECTOR(31 downto 0);
    rd:          out  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal PC, Instr,
         ReadData: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  i_arm: arm port map(clk, reset, PC, Instr, MemWrite, DataAdr,
                      WriteData, ReadData);
  i_imem: imem port map(PC, Instr);
  i_dmem: dmem port map(clk, MemWrite, DataAdr,
                        WriteData, ReadData);
end;
```

HDL Example 7.14 DATA MEMORY

SystemVerilog

```
module dmem(input  logic          clk, we,
            input  logic  [31:0]  a, wd,
            output logic  [31:0]  rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity dmem is -- data memory
  port(clk, we:  in  STD_LOGIC;
       a, wd:    in  STD_LOGIC_VECTOR(31 downto 0);
       rd:       out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array (63 downto 0) of
                  STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin -- read or write memory
    loop
      if clk'event and clk = '1' then
        if (we = '1') then
          mem(to_integer(a(7 downto 2))) := wd;
        end if;
      end if;
      rd <= mem(to_integer(a(7 downto 2)));
      wait on clk, a;
    end loop;
  end process;
end;
```

HDL Example 7.15 INSTRUCTION MEMORY

SystemVerilog

```
module imem(input  logic  [31:0]  a,
            output logic  [31:0]  rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("memfile.dat",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity imem is -- instruction memory
  port(a:   in  STD_LOGIC_VECTOR(31 downto 0);
       rd:  out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of imem is -- instruction memory
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable i, index, result: integer;
    type ramtype is array (63 downto 0) of
                        STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      mem(i) := (others => '0');
    end loop;
```

```
      index := 0;
      FILE_OPEN(mem_file, "memfile.dat", READ_MODE);
      while not endfile(mem_file) loop
        readline(mem_file, L);
        result := 0;
        for i in 1 to 8 loop
          read(L, ch);
          if '0' <= ch and ch <= '9' then
            result := character'pos(ch) - character'pos('0');
          elsif 'a' <= ch and ch <= 'f' then
            result := character'pos(ch) - character'pos('a') + 10;
          elsif 'A' <= ch and ch <= 'F' then
            result := character'pos(ch) - character'pos('A') + 10;
          else report "Format error on line " & integer'image(index)
            severity error;
          end if;
          mem(index)(35-i*4 downto 32-i*4) :=
            to_std_logic_vector(result,4);
        end loop;
        index := index + 1;
      end loop;

      -- read memory
      loop
        rd <= mem(to_integer(a(7 downto 2)));
        wait on a;
      end loop;
    end process;
  end;
```

7.7 ADVANCED MICROARCHITECTURE*

High-performance microprocessors use a wide variety of techniques to run programs faster. Recall that the time required to run a program is proportional to the period of the clock and to the number of clock cycles per instruction (CPI). Thus, to increase performance, we would like to speed-up the clock and/or reduce the CPI. This section surveys some existing speed-up techniques. The implementation details become quite complex, so we focus on the concepts. Hennessy & Patterson's *Computer Architecture* text is a definitive reference if you want to fully understand the details.

Advances in integrated circuit manufacturing have steadily reduced transistor sizes. Smaller transistors are faster and generally consume less power. Thus, even if the microarchitecture does not change, the clock frequency can increase because all the gates are faster. Moreover, smaller transistors enable placing more transistors on a chip. Microarchitects use the additional transistors to build more complicated processors or to put more processors on a chip. Unfortunately, power consumption increases with the number of transistors and the speed at which they operate (see Section 1.8). Power consumption is now an essential concern. Microprocessor designers have a challenging task juggling the trade-offs among speed, power, and cost for chips with billions of transistors in some of the most complex systems that humans have ever built.