9 General-Purpose Microprocessors

Unlike a dedicated or custom microprocessor that is capable of performing only one function, a general-purpose microprocessor is capable of performing many different functions under the direction of instructions. Given a different instruction set or program, the general-purpose microprocessor will perform a different function. On the other hand, a general-purpose microprocessor can also be viewed as a dedicated microprocessor because it is made to perform only one function, and that is to execute software instructions. In this sense, we can design and construct our general-purpose microprocessor in the same way that we construct our dedicated microprocessors as shown in the previous chapter.

9.1 Overview of the CPU Design

A general-purpose microprocessor is often referred to as the central processing unit (CPU). The CPU is simply a dedicated microprocessor that only executes software instructions. So in designing a CPU, the first thing that we need to do is to define its instruction set and how they are encoded. The instructions are executed by a module call the **datapath** inside the CPU. The datapath is responsible for executing all data manipulations required by the processor. The datapath contains functional units such as the ALU and shifter; storage elements such as registers; and connection buses including multiplexers for transferring data between all the different units. The operations of the datapath, i.e., the operations of all the units within the datapath, are control by another module inside the CPU call the **controller or control unit**. The controller is a finite-state machine that cycles through three main steps: 1) fetch an instruction; 2) decode the instruction; and 3) execute the instruction. These steps are performed by the controller sending the appropriate control signals to the datapath. The relationship between the controller and datapath is shown in Figure 1. Figure 2 is a picture of the actual circuit of the Intel 80386 CPU.

![Figure 1. Modules inside a CPU.](image-url)
9.2 Instruction Set

The instructions that our general-purpose microprocessor can execute and the corresponding encoding are defined in Table 1. The Instruction column shows the syntax and mnemonic to use for the instruction when writing in assembly language. The OpCode column shows the encoding for the instruction. The instructions fall into four general categories: 1) data movement instructions for transferring data between the accumulator, the general registers and the memory; 2) jump instructions for changing the instruction execution sequence; 3) arithmetic and logical instructions for performing arithmetic and logics; and 4) input/output and miscellaneous instructions. There are five data movement instructions, six jump instructions, nine arithmetic and logic instructions, two input/output instructions, and two miscellaneous instructions.

The number of instructions implemented will determine the number of bits required to encode all the instructions. All our instructions are encoded using one byte except for instructions requiring a memory address, which uses a second byte for the address. In our encoding scheme the first four bits is the operation code. If the instruction requires two operands, it uses the accumulator for one operand and the last four bits in the encoding specifies the second operand. An example of this is the load accumulator from register (LDA) instruction. If the instruction requires only one operand, it always uses the accumulator. In this case, the last three bits in the encoding is used to further decode the instructions. An example of this is the increment accumulator (INC) instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OpCode</th>
<th>Operation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>0000 0000</td>
<td>no operation</td>
<td>No operation</td>
</tr>
<tr>
<td>Data movement instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDA A,rrr</td>
<td>0001 xrrr</td>
<td>A ← R[rrr]</td>
<td>Load accumulator from register</td>
</tr>
<tr>
<td>STA rrr,A</td>
<td>0010 xrrr</td>
<td>R[rrr] ← A</td>
<td>Load register from accumulator</td>
</tr>
<tr>
<td>LDM A,aaaaaa</td>
<td>0011 xxxx xaaaaaa</td>
<td>A ← M[aaaaaa]</td>
<td>Load accumulator from memory</td>
</tr>
<tr>
<td>STM aaaaaa,A</td>
<td>0100 xxxx xaaaaaa</td>
<td>M[aaaaaa] ← A</td>
<td>Load memory from accumulator</td>
</tr>
<tr>
<td>LDI A,iii</td>
<td>0101 iiii</td>
<td>A ← iii</td>
<td>Load accumulator with immediate value (signed number)</td>
</tr>
<tr>
<td>Jump instructions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMPR relative</td>
<td>0110 ijjj</td>
<td>if ijjj != 0 then PC = PC + ijjj</td>
<td>Unconditional jump relative</td>
</tr>
<tr>
<td>JMP address</td>
<td>0110 0000 xaaaaaa</td>
<td>if(rrrr == 0) then PC = aaaaaa</td>
<td>Unconditional jump absolute</td>
</tr>
<tr>
<td>JZR relative</td>
<td>0111 ijjj</td>
<td>if(A == 0 and ijjj != 0) then PC = PC + ijjj</td>
<td>Jump if zero relative</td>
</tr>
<tr>
<td>JZ address</td>
<td>0111 0000 xaaaaaa</td>
<td>if(A == 0 and rrrr == 0) then PC = aaaaaa</td>
<td>Jump if zero absolute</td>
</tr>
<tr>
<td>JPR relative</td>
<td>1000 ijjj</td>
<td>if(A == positive and ijjj != 0) then PC = PC</td>
<td>Jump if positive relative</td>
</tr>
</tbody>
</table>
### Arithmetic and Logical Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operation Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND A, rrr</td>
<td>A ← A AND R[rrr] Accumulator AND register</td>
</tr>
<tr>
<td>OR A, rrr</td>
<td>A ← A OR R[rrr] Accumulator OR register</td>
</tr>
<tr>
<td>ADD A, rrr</td>
<td>A ← A + R[rrr] Accumulator + register</td>
</tr>
<tr>
<td>SUB A, rrr</td>
<td>A ← A − R[rrr] Accumulator − register</td>
</tr>
<tr>
<td>NOT A</td>
<td>A ← NOT A Invert accumulator</td>
</tr>
<tr>
<td>INC A</td>
<td>A ← A + 1 Increment accumulator</td>
</tr>
<tr>
<td>DEC A</td>
<td>A ← A − 1 Decrement accumulator</td>
</tr>
<tr>
<td>SHF Left</td>
<td>A ← A &lt;&lt; 1 Shift accumulator left</td>
</tr>
<tr>
<td>SHF Right</td>
<td>A ← A &gt;&gt; 1 Shift accumulator right</td>
</tr>
</tbody>
</table>

### Input / Output and Miscellaneous

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operation Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InA</td>
<td>A ← input Input to accumulator</td>
</tr>
<tr>
<td>OutA</td>
<td>output ← A Output from accumulator</td>
</tr>
<tr>
<td>HALT</td>
<td>Halt Halt execution</td>
</tr>
</tbody>
</table>

**Figure 3.** Instruction set for the general-purpose microprocessor.

### 9.3 Datapath

The datapath is shown in Figure 4. The width of the datapath is four bits. The VHDL code for the datapath is shown in Listing 1.
library IEEE;
use IEEE.std_logic_1164.all;

entity dp is
port(  clk_dp  : in bit;
       muxsel_dp : in std_logic_vector(1 downto 0);
       imm_dp   : in std_logic_vector(3 downto 0);
       memio_dp : in std_logic_vector(3 downto 0);
       out_dp   : out std_logic_vector(3 downto 0);
       zero_dp  : out std_logic;
       greater_dp : out std_logic;
       rfaddr_dp : in std_logic_vector(1 downto 0);
       rfwr_dp  : in std_logic;
       alusel_dp : in std_logic_vector(2 downto 0);
       shiftsel_dp : in std_logic
          );
end dp;

architecture struct of dp is

component mux4
port( sel_mux : in std_logic_vector(1 downto 0);
     in3_mux,in2_mux,in1_mux,in0_mux : in std_logic_vector(3 downto 0);
     out_mux : out std_logic_vector(3 downto 0)
       );
end component;

component acc
port (  clk_acc  : in bit;
        wr_acc  : in STD_LOGIC;
        input_acc : in STD_LOGIC_VECTOR (3 downto 0);
        output_acc : out STD_LOGIC_VECTOR (3 downto 0);
        zero_acc : out std_logic;
        greater_acc : out std_logic
           );
end component;

component reg_file
port(  clk_rf  : in bit;
       wr_rf  : in std_logic;
       addr_rf  : in std_logic_vector(1 downto 0);
       input_rf : in std_logic_vector(3 downto 0);
       output_rf: out std_logic_vector(3 downto 0)
          );
end component;

component alu
port( sel_alu : in std_logic_vector(2 downto 0);
     inA_alu : in std_logic_vector(3 downto 0);
     inB_alu : in std_logic_vector(3 downto 0);
     out_alu : out STD_LOGIC_VECTOR (3 downto 0)
       );
end component;
component shifter
deriv( sel_shift : in std_logic;
    input_shift : in std_logic_vector(3 downto 0);
    output_shift: out std_logic_vector(3 downto 0)
);end component;

signal C_aluout, C_accout, C_rfout, C_muxout, C_shiftout : std_logic_vector(3 downto 0);

begin
    U0: mux4 port map(muxsel_dp, imm_dp, memio_dp, C_rfout, C_shiftout, C_muxout);
    U1: acc port map(clk_dp, accwr_dp, C_muxout, C_accout, zero_dp, greater_dp);
    U2: reg_file port map(clk_dp, rfwr_dp, rfaddr_dp, C_accout, C_rfout);
    U3: alu port map(alusel_dp, C_accout, C_rfout, C_aluout);
    U4: shifter port map(shiftsel_dp, C_aluout, C_shiftout);

    out_dp <= C_accout;
end struct;

Listing 1. Datapath.

9.4 Control Unit

Asdf

9.5 VHDL Code

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity computer is
port(clock : in bit;
     reset : in std_logic;
     memio_comp : in std_logic_vector(3 downto 0);
     segs_comp : out std_logic_vector(1 to 8);
     segs_temp : out std_logic_vector(1 to 8)
);end computer;

architecture struc of computer is

component cpu
port( clk_cpu : in bit;
    rst_cpu : in std_logic;
    memio_cpu : in std_logic_vector(3 downto 0);
    out_cpu : out std_logic_vector(3 downto 0)
);end component;

component bcd
port ( in_bcd : in std_logic_vector(3 DOWNTO 0);
    Segs : out std_logic_vector(1 TO 7)
); end component;

signal C_outcpu: std_logic_vector(3 downto 0);
signal C_segs: std_logic_vector(1 to 7);

begin
  U0: cpu port map(clock,reset,memio_comp,C_outcpu);
  U1: bcd port map(C_outcpu,C_segs);
  segs_comp <= not C_segs & '1';
  segs_temp <= "11111111";
end struc;

Listing 2. Top level computer.

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ctrl is
  port(  clk_ctrl : in bit;
         rst_ctrl : in std_logic;
         muxsel_ctrl : out std_logic_vector(1 downto 0);
         imm_ctrl : out std_logic_vector(3 downto 0);
         accwr_ctrl : out std_logic;
         zero_ctrl : in std_logic;
         greater_ctrl: in std_logic;
         rfaddr_ctrl : out std_logic_vector(1 downto 0);
         rfwr_ctrl : out std_logic;
         alusel_ctrl : out std_logic_vector(2 downto 0);
         shiftsel_ctrl: out std_logic
           );
end ctrl;

architecture fsm of ctrl is
  type state_type is
    (S1,S2,S9,S10,S11,S12,S13,S14,S210,S220,
     S230,S30,S31,S32,S33,S41,S42,S43,S44,S45,
     S46,S47,S99);
  signal state : state_type;

  -- Instructions
  constant NOP : std_logic_vector(3 downto 0) := "0000";
    -- S1
  constant LDA : std_logic_vector(3 downto 0) := "0001";
    -- S10
  constant STA : std_logic_vector(3 downto 0) := "0010";
    -- S11
  constant LDM : std_logic_vector(3 downto 0) := "0011";
    -- S12
  constant STM : std_logic_vector(3 downto 0) := "0100";
    -- S13
  constant LDI : std_logic_vector(3 downto 0) := "0101";
    -- S14

  -- jump instructions
constant JMP : std_logic_vector(3 downto 0) := "0110";
   -- S210
--constant JMPR : std_logic_vector(3 downto 0) := "0110"; --the relative
   jumps are determined by the 4 LSBs
constant JZ  : std_logic_vector(3 downto 0) := "0111";
   -- S220
--constant JZR : std_logic_vector(3 downto 0) := "0111";
constant JP  : std_logic_vector(3 downto 0) := "1000";
   -- S230
--constant JPR : std_logic_vector(3 downto 0) := "1000";
-- arithmetic and logical instructions
constant ANDA : std_logic_vector(3 downto 0) := "1001";
   -- S30
constant ORA : std_logic_vector(3 downto 0) := "1010";
   -- S31
constant ADD : std_logic_vector(3 downto 0) := "1011";
   -- S32
constant SUB : std_logic_vector(3 downto 0) := "1100";
   -- S33
-- single operand instructions
constant SOI : std_logic_vector(3 downto 0) := "1101";
   -- S40
constant NOTA : std_logic_vector(2 downto 0) := "000";
   -- S41
constant INC : std_logic_vector(2 downto 0) := "001";
   -- S42
constant DEC : std_logic_vector(2 downto 0) := "010";
   -- S43
constant SHFL : std_logic_vector(2 downto 0) := "011";
   -- S44
constant SHFR : std_logic_vector(2 downto 0) := "100";
   -- S45
constant INA : std_logic_vector(2 downto 0) := "101";
   -- S46
constant OUTA : std_logic_vector(2 downto 0) := "110";
   -- S47
constant HALT : std_logic_vector(3 downto 0) := "1111";
   -- S99

type PM_BLOCK is array (0 to 31) of std_logic_vector(7 downto 0);
-- constant PM : PM_BLOCK := (}
-- "0010XXXX",
-- "00101000",
-- "00110000",
-- "00111100",
-- "00110001",
-- "00011100",
-- "00000100",
-- "01010000",
-- "00010001",
-- "10011000",
-- "00010100",
-- "00010100"
begin
process (rst_ctrl,clk_ctrl)
    variable IR : std_logic_vector(7 downto 0);
    variable OPCODE : std_logic_vector( 3 downto 0);
    variable PC : integer range 0 to 31;
    variable PM : PM_BLOCK;
begin
    if (rst_ctrl='1') then
        PC := 0;
        muxsel_ctrl <= "11";
        imm_ctrl <= "0000";
        accwr_ctrl <= '0';
        rfwr_ctrl <= '0';
        state <= S1;
        -- load program memory with statements
        PM(0) := "01010101"; -- LDI A <- 0101
        PM(1) := "00100001"; -- STA R[001] <- A
        PM(2) := "01010111"; -- LDI A <- 0111
        PM(3) := "0010011"; -- STA R[011] <- A
        PM(4) := "0110111"; -- JMP
        -- PM(4) := "00010011"; -- LDA A <- R[001]
        PM(5) := "10010001"; -- AND A <- A and R[001]
        PM(6) := "00010011"; -- LDA A <- R[011]
        PM(7) := "1010001"; -- OR A <- A or R[001]
        PM(8) := "00010011"; -- LDA A <- R[011]
        PM(9) := "10110001"; -- ADD A <- A + R[001]
        PM(10) := "00010011"; -- LDA A <- R[011]
        PM(11) := "11000000"; -- SUB A <- A - R[001]
        PM(12) := "00010011"; -- LDA A <- R[011]
        PM(13) := "11010000"; -- NOT A <- NOT A
        PM(14) := "00010011"; -- LDA A <- R[011]
        PM(15) := "1101001"; -- INC A <- A++
        PM(16) := "00010011"; -- LDA A <- R[011]
        PM(17) := "11010100"; -- DEC A <- A--
        PM(18) := "00010011"; -- LDA A <- R[011]
        PM(19) := "11010011"; -- SHR A <- A >> 1
        PM(20) := "00010011"; -- LDA A <- R[011]
        PM(21) := "11010100"; -- SHR A <- A << 1
        PM(22) := "11111111"; -- HALT
    elsif (clk_ctrl'event and clk_ctrl = '1') then
        case state is
            when S1 => -- fetch instruction
                IR := PM(PC);
                OPCODE := IR(7 downto 4);
                PC := PC + 1;
                --imm_ctrl <= "0000";
                accwr_ctrl <= '0';
    end process;
end begin;
rfwr_ctrl <='1';
state <= S9;

when S13 => -- STM
state <= S9;

when S14 => -- LDI -- OK
muxsel_ctrl <="11";
imm_ctrl <= IR(3 downto 0);
 accwr_ctrl <='1';
rfaddr_ctrl <= "00";
rfwr_ctrl <= '0';
alusel_ctrl <= "000";
shiftsel_ctrl<='0';
state <= S1;

when S210 => -- JMP
if (IR(3 downto 0) = "0000") then
-- absolute
IR := PM(PC);
case IR(4 downto 0) is
when "00000" => PC:=0;
when "00001" => PC:=1;
when "00010" => PC:=2;
when "00011" => PC:=3;
when "00100" => PC:=4;
when "00101" => PC:=5;
when "00110" => PC:=6;
when "00111" => PC:=7;
when "01000" => PC:=8;
when "01001" => PC:=9;
when "01010" => PC:=10;
when "01011" => PC:=11;
when "01100" => PC:=12;
when "01101" => PC:=13;
when "01110" => PC:=14;
when "01111" => PC:=15;
when "10000" => PC:=16;
when "10001" => PC:=17;
when "10010" => PC:=18;
when "10011" => PC:=19;
when "10100" => PC:=20;
when others =>
end case;
else -- relative
--     test:=CONV_STD_LOGIC_VECTOR(PC,5)+IR(3 downto 0);
end if;
state <= S1;

when S220 => -- JZ
if (zero_ctrl='1') then
IR := PM(PC);
case IR(4 downto 0) is
when "00000" => PC:=0;
when "00001" => PC:=1;
when "00010" => PC:=2;
when "00011" => PC:=3;
when "00100" => PC:=4;
when "00101" => PC:=5;
when "00110" => PC:=6;
when "00111" => PC:=7;
when "01000" => PC:=8;
when "01001" => PC:=9;
when "01010" => PC:=10;
when "01011" => PC:=11;
when "01100" => PC:=12;
when "01101" => PC:=13;
when "01110" => PC:=14;
when "01111" => PC:=15;
when "10000" => PC:=16;
when "10001" => PC:=17;
when "10010" => PC:=18;
when "10011" => PC:=19;
when "10100" => PC:=20;
when others =>
  end case;
state <= S9;
else
  PC := PC + 1;
  state <= S9;
end if;

when S230 => -- JP
  state <= S9;

when S30 => -- ANDA
  muxsel_ctrl <="00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= IR(2 downto 0);
  rfwr_ctrl <= '0';
  alusel_ctrl <="001";
  shiftsel_ctrl <='0';
  accwr_ctrl <='1'; -- write occurs in the next cycle
  state <= S9; -- need one extra cycle to write back result

when S31 => -- ORA
  muxsel_ctrl <="00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= IR(2 downto 0);
  rfwr_ctrl <= '0';
  alusel_ctrl <="010";
  shiftsel_ctrl <='0';
  accwr_ctrl <='1'; -- write occurs in the next cycle
  state <= S9; -- need one extra cycle to write back result

when S32 => -- ADD -- OK
  muxsel_ctrl <="00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= IR(2 downto 0);
  rfwr_ctrl <= '0';
  alusel_ctrl <="100";
  shiftsel_ctrl <='0';
  accwr_ctrl <='1'; -- write occurs in the next cycle
  state <= S9; -- need one extra cycle to write back result
when S33 => -- SUB
  muxsel_ctrl <= "00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= IR(2 downto 0);
  rfwr_ctrl <= '0';
  alusel_ctrl <= "101";
  shiftsel_ctrl <= '0';
  accwr_ctrl <= '1'; -- write occurs in the next cycle
  state <= S9; -- need one extra cycle to write back result
when S41 => -- NOTA
  muxsel_ctrl <= "00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= "00";
  rfwr_ctrl <= '0';
  alusel_ctrl <= "011";
  shiftsel_ctrl <= '0';
  accwr_ctrl <= '1'; -- write occurs in the next cycle
  state <= S9; -- need one extra cycle to write back result
when S42 => -- INC
  muxsel_ctrl <= "00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= "00";
  rfwr_ctrl <= '0';
  alusel_ctrl <= "110";
  shiftsel_ctrl <= '0';
  accwr_ctrl <= '1'; -- write occurs in the next cycle
  state <= S9; -- need one extra cycle to write back result
when S43 => -- DEC
  muxsel_ctrl <= "00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= "00";
  rfwr_ctrl <= '0';
  alusel_ctrl <= "111";
  shiftsel_ctrl <= '0';
  accwr_ctrl <= '1'; -- write occurs in the next cycle
  state <= S9; -- need one extra cycle to write back result
when S44 => -- SHFL
  muxsel_ctrl <= "00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= "00";
  rfwr_ctrl <= '0';
  alusel_ctrl <= "000"; -- pass
  shiftsel_ctrl <= '1';
  accwr_ctrl <= '1'; -- write occurs in the next cycle
  state <= S9; -- need one extra cycle to write back result
when S45 => -- SHFR
  muxsel_ctrl <= "00";
  imm_ctrl <= "0000";
  rfaddr_ctrl <= "00";
  rfwr_ctrl <= '0';
  alusel_ctrl <= "000"; -- pass
shiftsel_ctrl <='1';
accwr_ctrl <='1';  -- write occurs in the next cycle 
state <= S9;  -- need one extra cycle to write back result

when S46 => -- INA
state <= S9;
when S47 => -- OUTA
state <= S9;
when S99 => -- HALT
state <= S99;
when others =>
  end case;
end if;
end process;
end fsm;

Listing 3. Control unit.