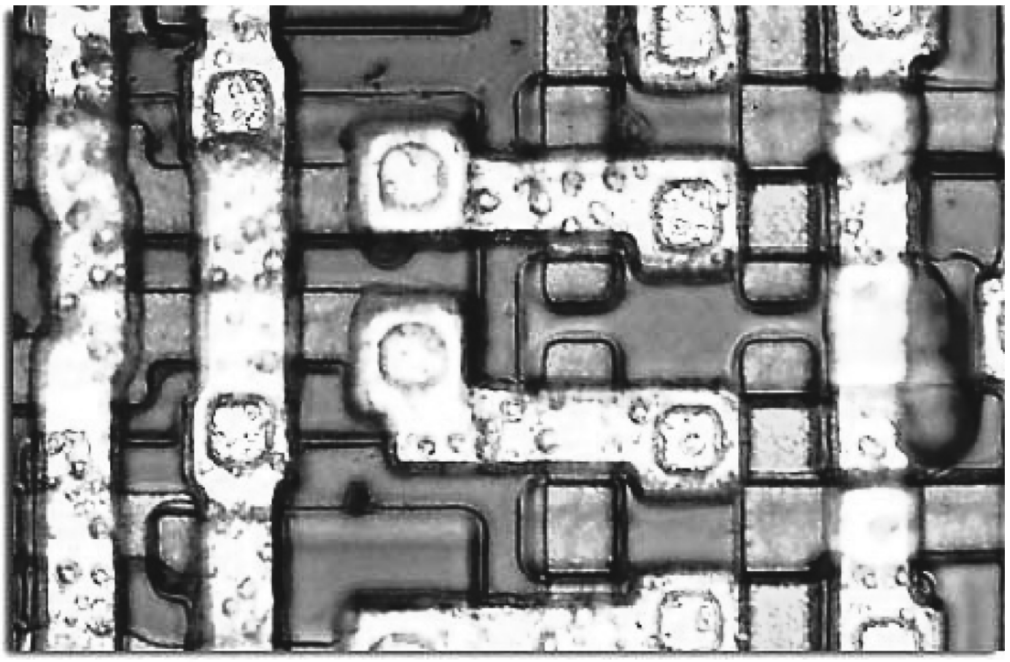
CHAPTER 9

# A Simple Computer Design: The $\mu$P 3



A partial die photograph of individual transistors about 10 microns tall on the Intel i4004 microprocessor is seen above. The 1971 Intel 4004 was the world's first single chip microprocessor. Prior to the 4004, Intel made memory chips. The 4004 was a 4-bit CPU with a clock rate of 108 kHz that contains 2,300 transistors. Photograph ©1995-2004 courtesy of Michael Davidson, http://micro.magnet.fsu.edu/chipshots.

# 9 A Simple Computer Design: The µP 3

A traditional digital computer consists of three main units, the processor or central processing unit (CPU), the memory that stores program instructions and data, and the input/output hardware that communicates to other devices. As seen in Figure 9.1, these units are connected by a collection of parallel digital signals called a bus. Typically, signals on the bus include the memory address, memory data, and bus status. Bus status signals indicate the current bus operation, memory read, memory write, or input/output operation.

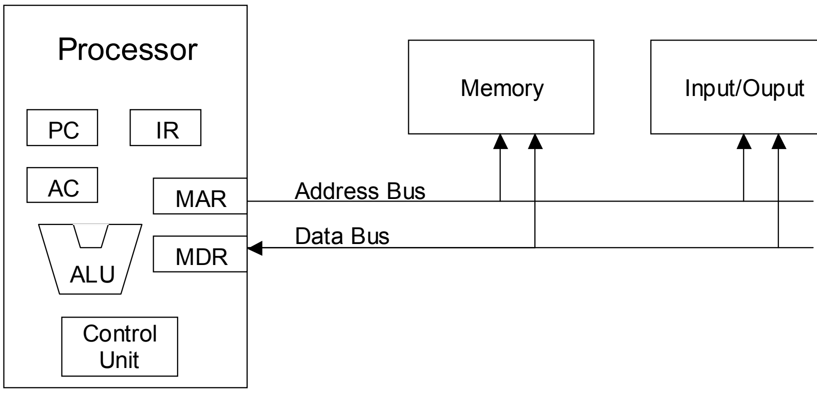


**Figure 9.1** Architecture of a Simple Computer System.

Internally, the CPU contains a small number of registers that are used to store data inside the processor. Registers such as PC, IR, AC, MAR and MDR are built using D flip-flops for data storage. One or more arithmetic logic units (ALUs) are also contained inside the CPU. The ALU is used to perform arithmetic and logical operations on data values. Common ALU operations include add, subtract, and logical and/or operations. Register-to-bus connections are hard wired for simple point-to-point connections. When one of several registers can drive the bus, the connections are constructed using multiplexers, open collector outputs, or tri-state outputs. The control unit is a complex state machine that controls the internal operation of the processor.

The primary operation performed by the processor is the execution of sequences of instructions stored in main memory. The CPU or processor reads or fetches an instruction from memory, decodes the instruction to determine what operations are required, and then executes the instruction. The control unit controls this sequence of operations in the processor.

## 9.1 Computer Programs and Instructions

A computer program is a sequence of instructions that perform a desired operation. Instructions are stored in memory. For the following simple µP 3 computer design, an instruction consists of 16 bits. As seen in Figure 9.2 the high eight bits of the instruction contain the opcode. The instruction operation code or "opcode" specifies the operation, such as add or subtract, that will be performed by the instruction. Typically, an instruction sends one set of data values through the ALU to perform this operation. The low eight bits of each instruction contain a memory address field. Depending on the opcode, this address may point to a data location or the location of another instruction. Some example instructions are shown in Figure 9.3.

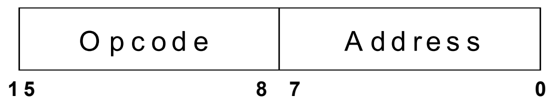


**Figure 9.2** Simple µP 3 Computer Instruction Format.

| Instruction Mnemonic | | Operation Preformed | Opcode Value |
|---|---|---|---|
| **ADD** | ***address*** | **AC <= AC + contents of memory address** | 00 |
| **STORE** | ***address*** | **contents of memory address <= AC** | 01 |
| **LOAD** | ***address*** | **AC <= contents of memory address** | 02 |
| **JUMP** | ***address*** | **PC <= address** | 03 |
| **JNEG** | ***address*** | **If AC < 0 Then PC <= address** | 04 |

**Figure 9.3** Basic µP 3 Computer Instructions.

An example program to compute A = B + C is shown in Figure 9.4. This program is a sequence of three instructions. Program variables such as A, B, and C are typically stored in dedicated memory locations. The symbolic representation of the instructions, called assembly language, is shown in the first column. The second column contains the same program in machine language (the binary pattern that is actually loaded into the computer's memory).

The machine language can be derived using the instruction format in Figure 9.2. First, find the opcode for each instruction in the first column of Figure 9.3. This provides the first two hexadecimal digits in machine language. Second, assign the data values of A, B, and C to be stored in hexadecimal addresses 10,11, and 12 in memory. The address provides the last two hexadecimal digits of each machine instruction.

| Assembly Language | Machine Language |
|---|---|
| **LOAD B** | **0211** |
| **ADD C** | **0012** |
| **STORE A** | **0110** |

**Figure 9.4** Example Computer Program for A = B + C.

The assignment of the data addresses must not conflict with instruction addresses. Normally, the data is stored in memory after all of the instructions in the program. In this case, if we assume the program starts at address 0, the three instructions will use memory addresses 0,1, and 2.

The instructions in this example program all perform data operations and execute in strictly sequential order. Instructions such as JUMP and JNEG are used to transfer control to a different address. Jump and Branch instructions do not execute in sequential order. Jump and Branch instructions must be used to implement control structures such as an IF…THEN statement or program loops. Details are provided in an exercise at the end of this section.

Assemblers are computer programs that automatically convert the symbolic assembly language program into the binary machine language. Compilers are programs that automatically translate higher-level languages, such as C or Pascal, into a sequence of machine instructions. Many compilers also have an option to output assembly language to aid in debugging.

The programmer's view of the computer only includes the registers (such as the program counter) and details that are required to understand the function of assembly or machine language instructions. Other registers and control hardware, such as the instruction register (IR), memory address register (MAR), and memory data register (MDR), are internal to the CPU and are not described in the assembly language level model of the computer. Computer engineers designing the processor must understand the function and operation of these internal registers and additional control hardware.

## 9.2 The Processor Fetch, Decode and Execute Cycle

The processor reads or fetches an instruction from memory, decodes the instruction to determine what operations are required, and then executes the instruction as seen in Figure 9.5. A simple state machine called the control unit controls this sequence of operations in the processor. The fetch, decode, and execute cycle is found in machines ranging from microprocessor-based PCs to supercomputers. Implementation of the fetch, decode, and execute cycle requires several register transfer operations and clock cycles in this example design.

The program counter contains the address of the current instruction. Normally, to fetch the next instruction from memory the processor must increment the program counter (PC). The processor must then send the address value in the PC to memory over the bus by loading the memory address register (MAR) and start a memory read operation on the bus. After a small delay, the instruction data will appear on the memory data bus lines, and it will be latched into the memory data register (MDR).

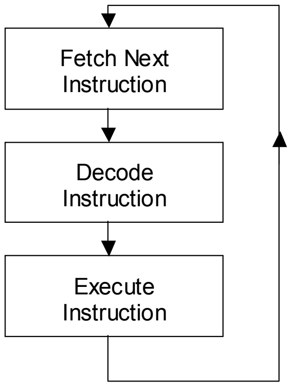


**Figure 9.5** Processor Fetch, Decode and Execute Cycle.

Execution of the instruction may require an additional memory cycle so the instruction is normally saved in the CPU's instruction register (IR). Using the value in the IR, the instruction can now be decoded. Execution of the instruction will require additional operations in the CPU and perhaps additional memory operations.

The Accumulator (AC) is the primary register used to perform data calculations and to hold temporary program data in the processor. After completing execution of the instruction the processor begins the cycle again by fetching the next instruction.

The detailed operation of a computer is often modeled by describing the register transfers occurring in the computer system. A variety of register transfer level (RTL) languages such as VHDL or Verilog are designed for this application. Unlike more traditional programming languages, RTL languages can model parallel operations and map easily into hardware designs. Logic synthesis tools can also be used to implement a hardware design automatically using an RTL description.

To explain the function and operation of the CPU in detail, consider the example computer design in Figure 9.1. The CPU contains a general-purpose data register called the accumulator (AC) and the program counter (PC). The arithmetic logic unit (ALU) is used for arithmetic and logical operations.

The fetch, decode, and execute cycle can be implemented in this computer using the sequence of register transfer operations shown in Figure 9.6. The next instruction is fetched from memory with the following register transfer operations:

**MAR = PC**

**Read Memory, MDR = Instruction value from memory**

**IR = MDR**

**PC = PC + 1**

After this sequence of operations, the current instruction is in the instruction register (IR). This instruction is one of several possible machine instructions such as ADD, LOAD, or STORE. The opcode field is tested to decode the specific machine instruction. The address field of the instruction register contains the address of possible data operands. Using the address field, a memory read is started in the decode state.

The decode state transfers control to one of several possible next states based on the opcode value. Each instruction requires a short sequence of register transfer operations to implement or execute that instruction. These register transfer operations are then performed to execute the instruction. Only a few of the instruction execute states are shown in Figure 9.6. When execution of the current instruction is completed, the cycle repeats by starting a memory read operation and returning to the fetch state. A small state machine called a control unit is used to control these internal processor states and control signals.

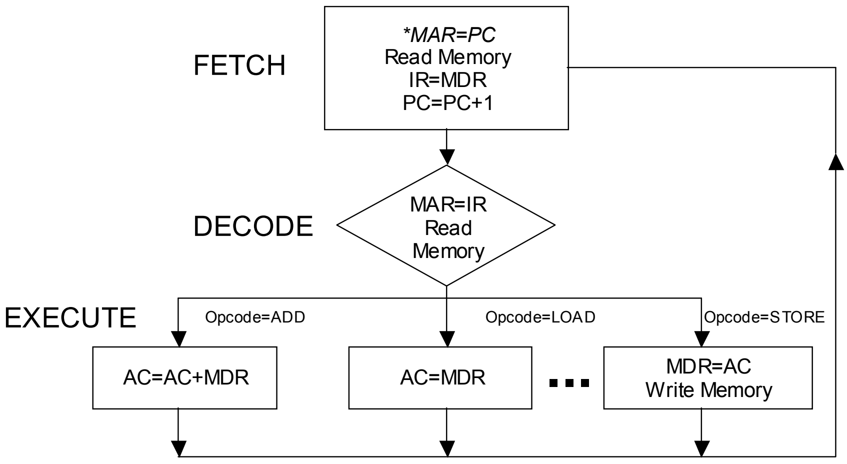


**Figure 9.6** Detailed View of Fetch, Decode, and Execute for the μP 3 Computer Design.

Figure 9.7 is the datapath used for the implementation of the μP 3 Computer. A computer's datapath consists of the registers, memory interface, ALUs, and the bus structures used to connect them. The vertical lines are the three major busses used to connect the registers. On the bus lines in the datapath, a "/" with a number indicates the number of bits on the bus. Data values present on the active busses are shown in hexadecimal. MW is the memory write control line.

A reset must be used to force the processor into a known state after power is applied. The initial contents of registers and memory produced by a reset can also be seen in Figure 9.7**.** Since the PC and MAR are reset to 00, program execution will start at 00.

Note that memory contains the machine code for the example program presented earlier. Recall that the program consists of a LOAD, ADD, and

STORE instruction starting at address 00. Data values for this example program are stored in memory locations, 10, 11, and 12.

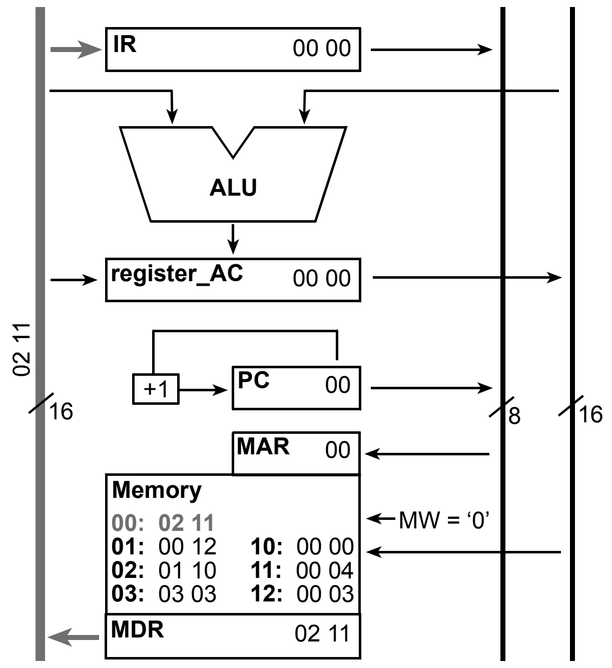


**Figure 9.7** Datapath used for the μP 3 Computer Design after applying reset.

Consider the execution of the ADD machine instruction (0012) stored at program location 01 in detail. The instruction, ADD *address*, adds the contents of the memory location at address 12 to the contents of AC and stores the result in AC. The following sequence of register transfer operations will be required to fetch and execute this instruction.

**FETCH**: *REGISTER TRANSFER CYCLE 1*:

***MAR = PC prior to fetch*, read memory, IR = MDR, PC = PC + 1**

First, the memory address register is loaded with the PC. In the example program, the ADD instruction (0012) is at location 01 in memory, so the PC and MAR will both contain 01. In this implementation of the computer, the MAR=PC operation will be moved to the end of the fetch, decode, and execute loop to the execute state in order to save a clock cycle. To fetch the instruction, a memory read operation is started. After a small delay for the memory access time, the ADD instruction is available at the input of the instruction register. To set up for the next instruction fetch, one is added to the program counter. The last two operations occur in parallel during one clock cycle using two different data busses.

At the rising edge of the clock signal, the decode state is entered. A block diagram of the register transfer operations for the fetch state is seen in Figure 9.8. Inactive busses are not shown.

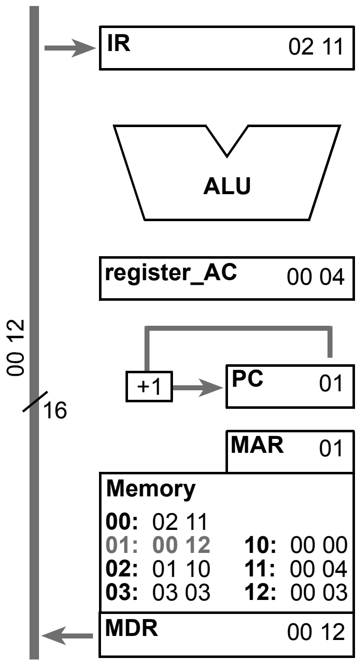


**Figure 9.8** Register transfers in the ADD instruction's Fetch State.

**DECODE**: *REGISTER TRANSFER CYCLE 2***:**

**Decode Opcode to find Next State, MAR = IR, and start memory read**

Using the new value in the IR, the CPU control hardware decodes the instruction's opcode of 00 and determines that this is an ADD instruction. Therefore, the next state in the following clock cycle will be the execute state for the ADD instruction.

Instructions typically are decoded in hardware using combinational circuits such as decoders, programmable logic arrays (PLAs), or perhaps even a small ROM. A memory read cycle is always started in decode, since the instruction may require a memory data operand in the execute state.

The ADD instruction requires a data operand from memory address 12. In Figure 9.9, the low 8–bit address field portion of the instruction in the IR is transferred to the MAR. At the next clock, after a small delay for the memory access time, the ADD instruction's data operand value from memory (0003) will be available in the MDR.

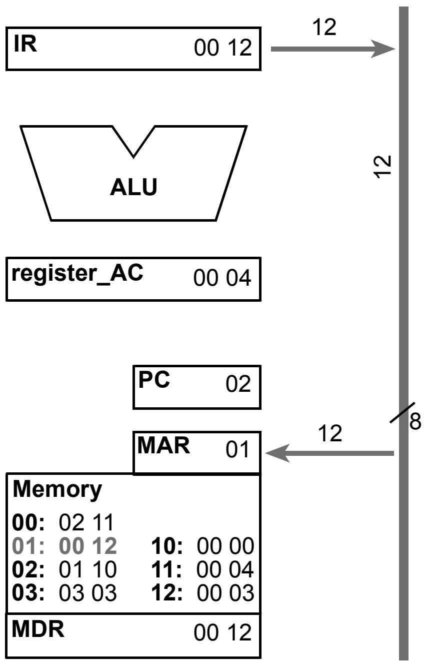


**Figure 9.9** Register transfers in the ADD instruction's Decode State.

## EXECUTE ADD: *REGISTER TRANSFER CYCLE 3*: AC = AC + MDR, MAR = PC*, and GOTO FETCH

The two values can now be added. The ALU operation input is set for addition by the control unit. As shown in Figure 9.10, the MDR's value of 0003 is fed into one input of the ALU. The contents of register AC (0004) are fed into the other ALU input. After a small delay for the addition circuitry, the sum of 0007 is produced by the ALU and will be loaded into the AC at the next clock. To provide the address for the next instruction fetch, the MAR is loaded with the current value of the PC (02). Note that by moving the operation, MAR=PC, to every instruction's final execute state, the fetch state can execute in one clock cycle. The ADD instruction is now complete and the processor starts to fetch the next instruction at the next clock cycle. Since three states were required, an ADD instruction will require three clock cycles to complete the operation.

After considering this example, it should be obvious that a thorough understanding of each instruction, the hardware organization, busses, control signals, and timing is required to design a processor. Some operations can be performed in parallel, while others must be performed sequentially. A bus can only transfer one value per clock cycle and an ALU can only compute one value per clock cycle, so ALUs, bus structures, and data transfers will limit those operations that can be done in parallel during a single clock cycle. In the states examined, a maximum of three buses were used for register transfers. Timing in critical paths, such as ALU delays and memory access times, will determine the clock speed at which these operations can be performed.

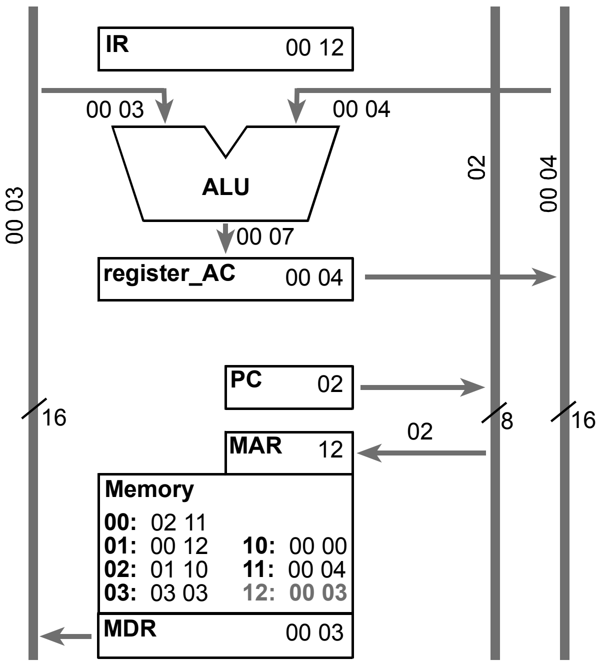


**Figure 9.10** Register transfers in the ADD instruction's Execute State.

The μP 3's multiple clock cycles per instruction implementation approach was used in early generation microprocessors. These computers had limited hardware, since the VLSI technology at that time supported orders of magnitude fewer gates on a chip than is now possible in current devices. Current generation processors, such as those used in personal computers, have a hundred or more instructions, and use additional means to speedup program execution. Instruction formats are more complex with up to 32 data registers and with additional instruction bits that are used for longer address fields and more powerful addressing modes.

Pipelining converts fetch, decode, and execute into a parallel operation mode instead of sequential. As an example, with three stage pipelining, the fetch unit fetches instruction n + 2, while the decode unit decodes instruction n + 1, and the execute unit executes instruction n. With this faster pipelined approach, an instruction finishes execution every clock cycle rather than three as in the simple computer design presented here.

Superscalar machines are pipelined computers that contain multiple fetch, decode and execute units. Superscalar computers can execute several instructions in one clock cycle. Most current generation processors including those in personal computers are both pipelined and superscalar. An example of a pipelined, reduced instruction set computer (RISC) design can be found in Chapter 14.

## 9.3 VHDL Model of the μP 3

To demonstrate the operation of a computer, a VHDL model of the μP 3 computer is shown in Figure 9.11. The simple μP 3 computer design fits easily into a FPGA device using less than 1-10% of its logic. The computer's RAM memory is implemented using the Altsyncram function which uses the FPGA's internal memory blocks.

The remainder of the computer model is basically a VHDL-based state machine that implements the fetch, decode, and execute cycle. The first few lines declare internal registers for the processor along with the states needed for the fetch, decode and execute cycle. A long CASE statement is used to implement the control unit state machine. A reset state is needed to initialize the processor. In the reset state, several of the registers are reset to zero and a memory read of the first instruction is started. This forces the processor to start executing instructions at location 00 in a predictable state after a reset.

The fetch state adds one to the PC and loads the instruction into the instruction register (IR). After the rising edge of the clock signal, the decode state starts. In decode, the low eight bits of the instruction register are used to start a memory read operation in case the instruction needs a data operand from memory. The decode state contains another CASE statement to decode the instruction using the opcode value in the high eight bits of the instruction. This means that the computer can have up to 256 different instructions, although only four are implemented in the basic model. Other instructions can be added as exercises. After the rising edge of the clock signal, control transfers to an execute state that is specific for each instruction.

Some instructions can execute in one clock cycle and some instructions may take more than one clock cycle. Instructions that write to memory will require more than one state for execute because of memory timing constraints. As seen in the STORE instruction, the memory address and data needs to be stable before and after the memory write signal is High, hence, additional states are used to avoid violating memory setup and hold times. When each instruction finishes the execute state, MAR is loaded with the PC to start the fetch of the next instruction. After the final execute state for each instruction, control returns to the fetch state.

Since the FPGA's synchronous memory block requires and contains an internal memory address and memory write register, it is necessary to make all assignments to the memory address register and memory write outside of the process to avoid having two cascaded registers. Recall that any assignment made in a clocked process synthesizes registers. Two cascaded MAR registers would require a delay of two clocks to load a new address for a memory operation.

The machine language program shown in Figure 9.12 is loaded into memory using a memory initialization file (*.mif). This produces 256 words of 16-bit memory for instructions and data. The memory initialization file, program.mif can be edited to change the loaded program. A write is performed only when the memory_write signal is High. On a Cyclone FPGA device, the access time for memory operations is in the range of 5-10ns.

```
                    -- Simple Computer Model Scomp.vhd
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY altera_mf;
USE  altera_mf.altera_mf_components.ALL;

ENTITY SCOMP IS
PORT(  clock, reset                    : IN STD_LOGIC;
       program_counter_out             : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
       register_AC_out                 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0 );
       memory_data_register_out        : OUT STD_LOGIC_VECTOR(15 DOWNTO 0 ));
       memory_address_register_out     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0 );
       memory_write_out                : OUT STD_LOGIC);
END SCOMP;

ARCHITECTURE a OF scomp IS
TYPE STATE_TYPE IS ( reset_pc, fetch, decode, execute_add, execute_load, execute_store,
                     execute_store2, execute_jump );
SIGNAL state: STATE_TYPE;
SIGNAL instruction_register, memory_data_register     : STD_LOGIC_VECTOR(15 DOWNTO 0 );
SIGNAL register_AC                                    : STD_LOGIC_VECTOR(15 DOWNTO 0 );
SIGNAL program_counter                                :  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL memory_address_register                        : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL memory_write                                   : STD_LOGIC;
BEGIN
                    -- Use Altsyncram function for computer's memory (256 16-bit words)
   memory: altsyncram
        GENERIC MAP (
                    operation_mode => "SINGLE_PORT",
                    width_a => 16,
                    widthad_a => 8,
                    lpm_type => "altsyncram",
                    outdata_reg_a => "UNREGISTERED",
                          -- Reads in mif file for initial program and data values
                    init_file => "program.mif",
                    intended_device_family => "Cyclone")

        PORT MAP (wren_a => memory_write, clock0 => clock,
                     address_a =>memory_address_register,  data_a => Register_AC,
                     q_a => memory_data_register );
                          -- Output major signals for simulation
  program_counter_out             <= program_counter;
  register_AC_out                 <= register_AC;
  memory_data_register_out        <= memory_data_register;
  memory_address_register_out <= memory_address_register;
```

```
PROCESS ( CLOCK, RESET )
    BEGIN
    IF reset = '1' THEN
        state <= reset_pc;
    ELSIF clock'EVENT AND clock = '1' THEN

        CASE state IS
                            -- reset the computer, need to clear some registers
        WHEN reset_pc =>
            program_counter         <= "00000000";
            register_AC             <= "0000000000000000";
            state                   <= fetch;
                            -- Fetch instruction from memory and add 1 to PC
        WHEN fetch =>
            instruction_register    <= memory_data_register;
            program_counter         <= program_counter + 1;
            state                   <= decode;
                            -- Decode instruction and send out address of any data operands
        WHEN decode =>
            CASE instruction_register( 15 DOWNTO 8 ) IS
                    WHEN "00000000" =>
                      state <= execute_add;
                    WHEN "00000001" =>
                      state <= execute_store;
                    WHEN "00000010" =>
                      state <= execute_load;
                    WHEN "00000011" =>
                      state <= execute_jump;
                    WHEN OTHERS =>
                      state <= fetch;
            END CASE;
                            -- Execute the ADD instruction
        WHEN execute_add =>
            register_ac             <= register_ac + memory_data_register;
            state                   <= fetch;
                            -- Execute the STORE instruction
                            -- (needs two clock cycles for memory write and fetch mem setup)
        WHEN execute_store =>
                            -- write register_A to memory, enable memory write
                            -- load memory address and data registers for memory write
            state                   <= execute_store2;
                            --finish memory write operation and load memory registers
                            --for next fetch memory read operation
        WHEN execute_store2 =>
            state                   <= fetch;
                            -- Execute the LOAD instruction
        WHEN execute_load =>
            register_ac             <=  memory_data_register;
            state                   <=  fetch;
                            -- Execute the JUMP instruction
        WHEN execute_jump =>
            program_counter         <= instruction_register( 7 DOWNTO 0 );
            state                   <= fetch;
        WHEN OTHERS =>
            state <= fetch;
        END CASE;
    END IF;
    END PROCESS;

                -- memory address register is already inside synchronous memory unit
                -- need to load its value based on current state
                -- (no second register is used - not inside a process here)
  WITH state SELECT
                memory_address_register <= "00000000"  WHEN reset_pc,
                  program_counter                      WHEN fetch,
                  instruction_register(7 DOWNTO 0)     WHEN decode,
                  program_counter                      WHEN execute_add,
                  instruction_register(7 DOWNTO 0)     WHEN execute_store,
                  program_counter                      WHEN execute_store2,
                  program_counter                      WHEN execute_load,
                  instruction_register(7 DOWNTO 0)     WHEN execute_jump;
   WITH state SELECT
                memory_write <=    '1'                 WHEN execute_store,
                                   '0'                 WHEN Others;
END a;
```

**Figure 9.11** VHDL Model of μP 3 Computer.

## 9.4 Verilog Model of the μP 3

To demonstrate the operation of the computer using Verilog, a Verilog model of the μP 3 computer is shown in Figure 9.12. The computer's RAM memory is implemented using the Altsyncram function which uses the FPGA's internal memory blocks. The remainder of the computer model is basically a Verilog-based state machine that implements the fetch, decode, and execute cycle. The first few lines declare internal registers for the processor along with the states needed for the fetch, decode and execute cycle. A long CASE statement is used to implement the control unit state machine. A reset state is needed to initialize the processor. In the reset state, several of the registers are reset to zero and a memory read of the first instruction is started. This forces the processor to start executing instructions at location 00 in a predictable state after a reset. A second case statement at the end of the code makes assignments to the memory address register based on the current state.

```
                  //uP3 Computer Design in Verilog
module scomp (clock,reset,program_counter,register_A,
                  memory_data_register_out, instruction_register);

  input clock,reset;
  output [7:0] program_counter;
  output [15:0] register_A, memory_data_register_out, instruction_register;

  reg  [15:0] register_A,  instruction_register;
  reg  [7:0] program_counter;
  reg  [3:0] state;
```

```
// State Encodings for Control Unit
parameter       reset_pc        = 0,
                fetch           = 1,
                decode          = 2,
                execute_add     = 3,
                execute_store   = 4,
                execute_store2 = 5,
                execute_store3 = 6,
                execute_load    = 7,
                execute_jump    = 8;

reg [7:0] memory_address_register;
reg memory_write;

wire [15:0] memory_data_register;
wire [15:0] memory_data_register_out = memory_data_register;
wire [15:0] memory_address_register_out = memory_address_register;
wire memory_write_out = memory_write;

// Use Altsynram function for computer's memory (256 16-bit words)
altsyncram      altsyncram_component (
                                .wren_a (memory_write_out),
                                .clock0 (clock),
                                .address_a (memory_address_register_out),
                                .data_a (register_A),
                                .q_a (memory_data_register));
        defparam
                altsyncram_component.operation_mode = "SINGLE_PORT",
                altsyncram_component.width_a = 16,
                altsyncram_component.widthad_a = 8,
                altsyncram_component.outdata_reg_a = "UNREGISTERED",
                altsyncram_component.lpm_type = "altsyncram",
// Reads in mif file for initial program and data values
                altsyncram_component.init_file = "program.mif",
                altsyncram_component.intended_device_family = "Cyclone";


  always @(posedge clock or posedge reset)
   begin
     if (reset)
        state = reset_pc;
     else
         case (state)
// reset the computer, need to clear some registers
                reset_pc :
                        begin
                                program_counter = 8'b00000000;
                                register_A = 16'b0000000000000000;
                                state = fetch;
                        end
// Fetch instruction from memory and add 1 to program counter
```

```
          fetch :
               begin
                         instruction_register = memory_data_register;
                         program_counter = program_counter + 1;
                         state = decode;
               end
// Decode instruction and send out address of any required data operands
          decode :
               begin
                    case (instruction_register[15:8])
                         8'b00000000:
                              state = execute_add;
                         8'b00000001:
                              state = execute_store;
                         8'b00000010:
                              state = execute_load;
                         8'b00000011:
                              state = execute_jump;
                         default:
                              state = fetch;
                    endcase
               end
// Execute the ADD instruction
          execute_add :
               begin
                    register_A = register_A + memory_data_register;
                    state = fetch;
               end
// Execute the STORE instruction (needs three clock cycles for memory write)
          execute_store :
               begin
// write register_A to memory
                    state = execute_store2;
               end
// This state ensures that the memory address is valid until after memory_write goes low
          execute_store2 :
               begin
                    state = execute_store3;
               end
// Execute the LOAD instruction
          execute_load :
               begin
                    register_A = memory_data_register;
                    state = fetch;
// Execute the JUMP instruction
               end
          execute_jump :
               begin
                    program_counter = instruction_register[7:0];
                    state = fetch;
               end
          default :
```

```
                        begin
                                state = fetch;
                        end
                endcase
    end
        // Make these assignments immediately during current state (i.e., unregistered)
         always @(state or program_counter or instruction_register)
          begin
                case (state)
                        reset_pc:          memory_address_register = 8'h 00;
                        fetch:             memory_address_register = program_counter;
                        decode:            memory_address_register = instruction_register[7:0];
                        execute_add:       memory_address_register = program_counter;
                        execute_store:     memory_address_register = instruction_register[7:0];
                        execute_store2:    memory_address_register = program_counter;
                        execute_load:      memory_address_register = program_counter;
                        execute_jump:      memory_address_register = instruction_register[7:0];
                        default:           memory_address_register = program_counter;
                endcase
                case (state)
                        execute_store:  memory_write = 1'b 1;
                        default:        memory_write = 1'b 0;
                endcase
          end
endmodule
```

**Figure 9.12** Verilog Model of µP 3 Computer.

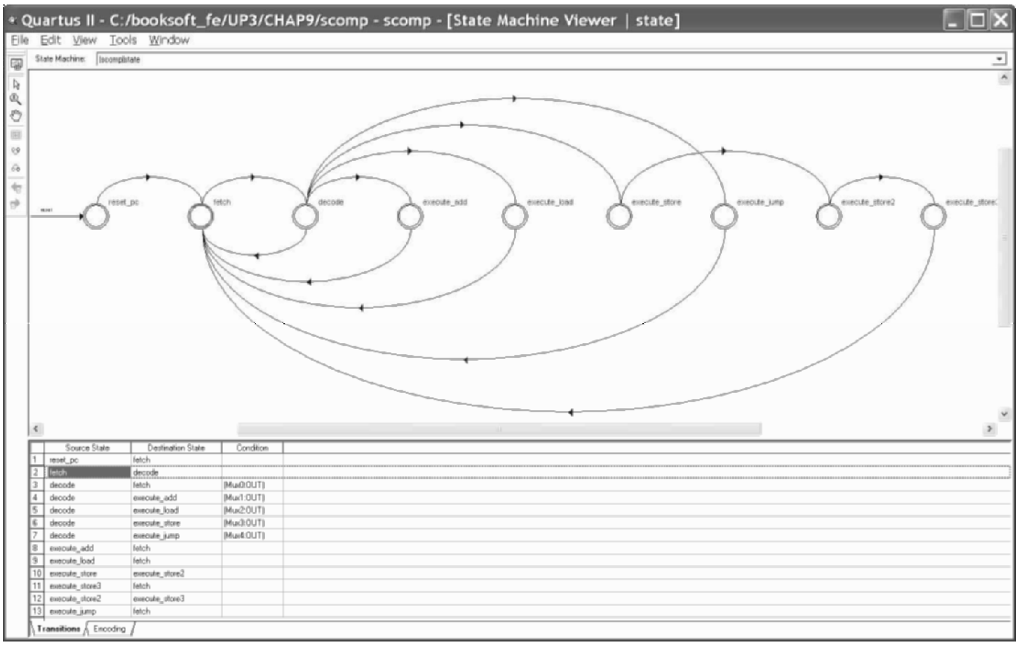
```
DEPTH = 256;                    % Memory depth and width are required       %
WIDTH = 16;                     % Enter a decimal number          %

ADDRESS_RADIX = HEX;            % Address and value radixes are optional    %
DATA_RADIX = HEX;               % Enter BIN, DEC, HEX, or OCT; unless       %
                                % otherwise specified, radixes = HEX        %

                        -- Specify values for addresses, which can be single address or range
CONTENT
  BEGIN
    [00..FF]    :       0000;   % Range--Every address from 00 to FF = 0000 (Default) %
      00   :            0210;   % LOAD AC with MEM(10) %
      01   :            0011;   % ADD MEM(11) to AC %
      02   :            0112;   % STORE  AC in MEM(12) %
      03   :            0212;   % LOAD AC with MEM(12) check for new value of FFFF %
      04   :            0304;   % JUMP to 04 (loop forever) %
      10   :            AAAA;   % Data Value of B %
      11   :            5555;   % Data Value of C%
      12   :            0000;   % Data Value of A - should be FFFF after running program %
  END ;
```

**Figure 9.13** Progam.mif file containg µP 3 Computer Program and DATA.

## 9.5 Automatically Generating a State Diagram of the μP3

Using **Tools ⇨Netlist Viewers ⇨State Diagram Viewer** to automatically produce a state diagram of the μP3 model's state machine, the state diagram seen in Figure 9.14 is generated.

Note that the Fetch, Decode and Execute cycle is clearly displayed in the state diagram. The initial reset state is seen on the far left of the state diagram. The Fetch state (highlighted) jumps to Decode. Decode then jumps to one of several Execute states depending on the instruction opcode. After execution of the instruction is complete, all of the various execute states jump back to Fetch. The state table displayed below the state diagram. Click on the encoding tab at the very bottom to see how the different states are encoded in hardware.


Quartus II - C:/booksoft_fe/UP3/CHAP9/scomp - scomp - [State Machine Viewer | state]
File Edit View Tools Window
State Machine: |scomp|state
reset
reset_pc
fetch
decode
execute_add
execute_load
execute_store
execute_jump
execute_store2
execute_store3
Transitions
Encoding

| | Source State | Destination State | Condition |
|---|---|---|---|
| 1 | reset_pc | fetch | |
| 2 | fetch | decode | |
| 3 | decode | fetch | (Mux0:OUT) |
| 4 | decode | execute_add | (Mux1:OUT) |
| 5 | decode | execute_load | (Mux2:OUT) |
| 6 | decode | execute_store | (Mux3:OUT) |
| 7 | decode | execute_jump | (Mux4:OUT) |
| 8 | execute_add | fetch | |
| 9 | execute_load | fetch | |
| 10 | execute_store | execute_store2 | |
| 11 | execute_store3 | fetch | |
| 12 | execute_store2 | execute_store3 | |
| 13 | execute_jump | fetch | |

**Figure 9.14** Automatically generated state diagram of the μP3 model.

## 9.6 Simulation of the µP3 Computer

A simulation output from the VHDL model is seen in Figure 9.15. After a reset, the test program seen in Figure 9.13, loads, adds, and stores a data value to compute A = B + C. The final value is then loaded again to demonstrate that the memory contains the correct value for A. The program then ends with a jump instruction that jumps back to its own address producing an infinite loop. After running the program, FF is stored in location 12. Memory can be examined in the Simulator after running a program by clicking on the Logical Memories section in the left column of the Simulation Report. An example is shown in Figure 9.16. Note that the clock period is set to 20ns for simulation.

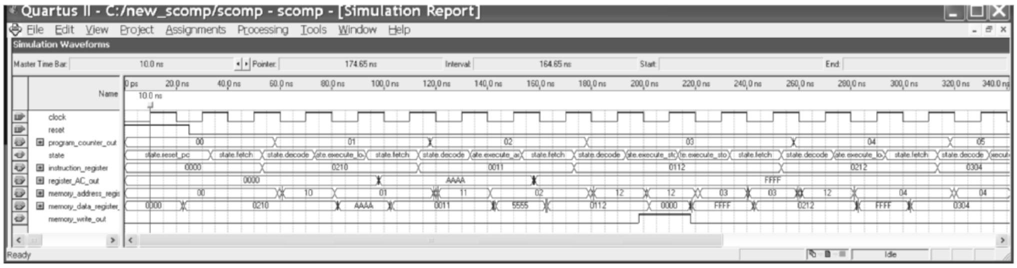
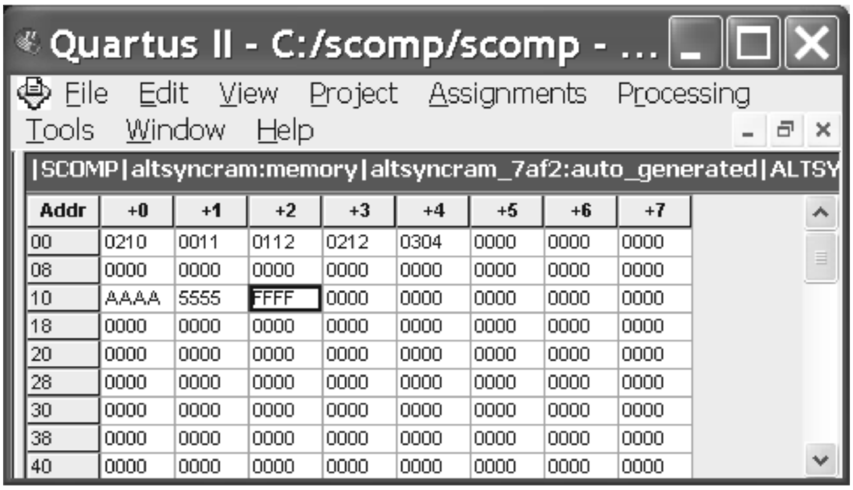


**Figure 9.15** Simulation of the Simple µP 3 Computer Program.


Quartus II - C:/scomp/scomp - ...
File Edit View Project Assignments Processing Tools Window Help

|SCOMP|altsyncram:memory|altsyncram_7af2:auto_generated|ALTSY

| Addr | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 |
|---|---|---|---|---|---|---|---|---|
| 00 | 0210 | 0011 | 0112 | 0212 | 0304 | 0000 | 0000 | 0000 |
| 08 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 10 | AAAA | 5555 | FFFF | 0000 | 0000 | 0000 | 0000 | 0000 |
| 18 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 20 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 28 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 30 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 38 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| 40 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |

**Figure 9.16** Simulation display of µP 3 Computer Memory showing result stored in memory

## 9.7 Laboratory Exercises

1. Compile and simulate the μP 3 computer VHDL or Verilog model. Rewrite the machine language program in the program.mif file to compute A = (B + C) + D. Store D in location 13 in memory. End the program with a Jump instruction that jumps to itself. Be sure to select the UP3's Cyclone device as the target. Find the maximum clock rate of the μP 3 computer. Examine the project's compiler report and find the logic cell (LC) percentage utilized.

2. Add the JNEG execute state to the CASE statement in the model. JNEG is Jump if AC < 0. If A >= 0 the next sequential instruction is executed. In most cases, a new instruction will just require a new execute state in the decode CASE statement. Use the opcode value of 04 for JNEG. Test the new instruction with the following test program that implements the operation, IF A>= 0 THEN B = C

| | Assembly Language | | Machine Language | Memory Address |
|---|---|---|---|---|
| | LOAD | A | 0210 | 00 |
| | JNEG | End_of_If | 0404 | 01 |
| | LOAD | C | 0212 | 02 |
| | STORE | B | 0111 | 03 |
| End_of_If: | JMP | End_of_If | 0304 | 04 |

End_of_If is an example of a label; it is a symbolic representation for a location in the program. Labels are used in assembly language to mark locations in a program. The last line that starts out with End_of_If: is the address used for the End_of_If symbol in the Jump instruction address field. Assuming the program starts at address 00, the value of the End_of_If label will be 04. Test the JNEG instruction for both cases A < 0 and A >= 0. Place nonzero values in the *.mif file for B and C so that you can verify the program executes correctly.

3. Add the instructions in the table below to the VHDL model, construct a test program for each instruction, compile and simulate to verify correct operation. In JPOS and JZERO instructions, both cases must be tested.

| Instruction | Function | Opcode |
|---|---|---|
| SUBT *address* | AC = AC - MDR | 05 |
| XOR *address* | AC = AC XOR MDR | 06 |
| OR *address* | AC = AC OR MDR | 07 |
| AND *address* | AC = AC AND MDR | 08 |
| JPOS *address* | IF AC > 0 THEN PC = address | 09 |
| JZERO *address* | IF AC = 0 THEN PC = address | 0A |
| ADDI *address* | AC = AC + *address* | 0B |

In the logical XOR instruction each bit is exclusive OR'ed with the corresponding bit in each operation for a total of sixteen independent exclusive OR operations. This is called a bitwise logical operation. OR and AND are also bitwise logical operations. The add-immediate instruction, ADDI, sign extends the 8-bit address field value to 16 bits. To sign extend, copy the sign bit to all eight high bits. This allows the use of both positive and negative two's complement numbers for the 8-bit immediate value stored in the instruction.

4. Add the following two shift instructions to the simple computer model and verify with a test program and simulation.

| Instruction | Function | Opcode |
|---|---|---|
| SHL address | AC = AC shifted left *address* bits | 0C |
| SHR address | AC = AC shifted right *address* bits | 0D |

The function LPM_CLSHIFT is useful to implement multiple bit shifts. SHL and SHR can also be used if 1993 VHDL features are enabled in the compiler. Only the low four bits of the address field contain the shift amount. The other four bits are always zero.

5. Run the µP 3 computer model using one of the FPGA boards. Use a debounced pushbutton for the clock and the other pushbutton for reset. Output the PC in hex to the LCD display or seven segment LEDs. Run a test program on the board and verify the correct value of the PC appears in the LCD display by stepping through the program using the pushbutton.

6. Add these two input/output (I/O) instructions to the µP 3 computer model running on the UP3 board.

| Instruction | Function | Opcode |
|---|---|---|
| IN i/o address | AC = switch bits (low 4 bits) | 0E |
| OUT i/o address | LCD or 7-Seg LED displays hex value of AC | 0F |

These instructions modify or use only the low eight bits of AC. Remove the PC display feature from the previous problem, if it was added or for more of a challenge place the AC value on the second line of the hex display by modifying the LCD display code. Test the new I/O instructions by writing a program that reads in the switches, adds one to the switch value, and outputs this value to the LED display. Repeat the input, add, and output operation in an infinite loop by jumping back to the start of the program. Add a new register, register_output, to the input of the seven-segment decoder that drives the LED display or use the LCD display. The register is loaded with the value of AC only when an OUT instruction is executed. Compile, download, and execute the program on the FPGA board. When several I/O devices are present, they should respond only to their own unique I/O address, just like memory.

7. Use the timing analyzer to determine the maximum clock rate for the µP 3 computer. Using this value, compute the execution time for the example program in Figure 9.4.

8. Modify the video output display described in Chapter 9 for the MIPS computer example to display the µP 3's internal registers. While running on the FPGA board, use the pushbuttons for clock and reset as suggested in problem 5.

9. Add video character output and keyboard input to the computer, after studying the material presented in Chapters 9 and 10.

10. Add the WAIT instruction to the simple computer model and verify with a test program and simulation. WAIT *value*, loads and starts an 8-bit ten-millisecond ($10^{-2}$ second) timer and then waits *value*10* ms before returning to fetch for the next instruction. Use an opcode of 10 for the WAIT instruction.

11. Expand the memory address space of the µP 3 computer from eight bits to nine bits. Some registers will also need an additional bit. Use 512 locations of 16-bit memory. Expand the address field by 1-bit by reducing the size of the opcode field by 1-bit. This will limit the number of different instructions to 128 but the maximum program size can now increase from 256 locations to 512 locations.

12. Modify the µP 3 computer so that it uses two different memories. Use one memory for instructions and a new memory for data values. The new data memory should be 256 or 512 (see previous problem ) locations of 16-bit data.

13. Add a subroutine CALL and RETURN instruction to the µP 3 computer design. Use a dedicated register to store the return address or use a stack with a stack pointer register. The stack should start at high addresses and as it grows move to lower addresses.

14. Implement a stack as suggested in the previous problem and add instructions to PUSH or POP register AC from the stack. At reset, set the stack pointer to the highest address of data memory.

15. Add all of the instructions and features suggested in the exercises to the µP 3 computer and use it as a microcontroller core for one of the robot projects suggested in Chapter 12. Additional instructions of your own design along with an interval timer that can be read using the IN instruction may also be useful.

16. Using the two low-bits from the opcode field, add a register address field that selects one of four different data registers A, B, C, or D for each instruction.

17. Use the implementation approach in the µP 3 computer model as a starting point to implement the basic instruction set of a different computer from your digital logic textbook or other reference manual.