VHDL Examples

Figures in this lecture are from:

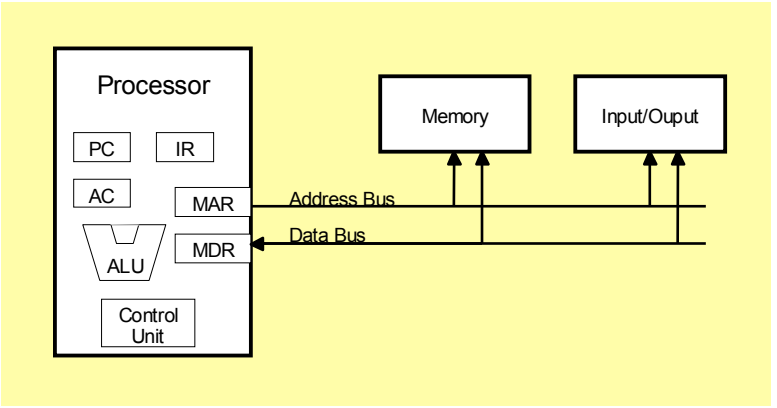
Rapid Prototyping of Digital Systems, Second Edition
James O. Hamblen & Michael D. Furman,
Kluwer Academic Publishers, 2001,
ISBN 0-7923-7439-8

Dr. Robert A. Walker

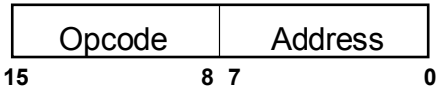
Computer Science Department
Kent State University
Kent, OH 44242 USA

http://www.cs.kent.edu/~walker

Architecture of Typical Computer System



µP1 Instruction Format & Instructions

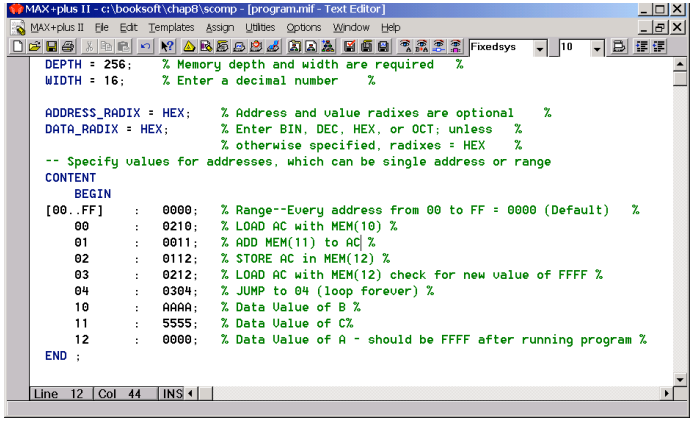


Instruction Mnemonic	Operation Preformed	Opcode Value
ADD *address*	AC <= AC + contents of memory address	00
STORE *address*	contents of memory address <= AC	01
LOAD *address*	AC <= contents of memory address	02
JUMP *address*	PC <= address	03
JNEG *address*	If AC < 0 Then PC <= address	04

µP1 Program for "A = B + C"

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

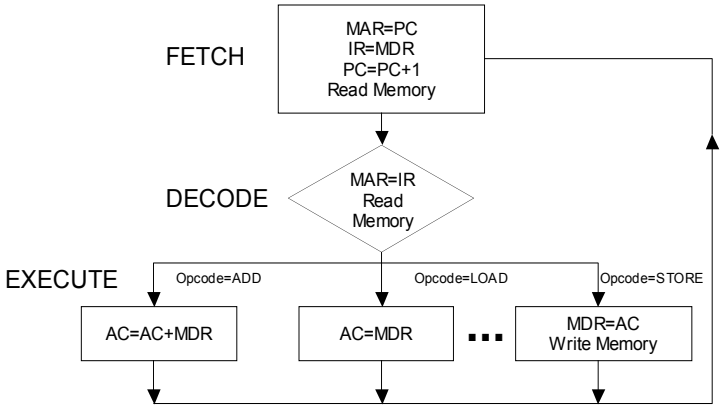
MIF (Memory) File of µP1 Program



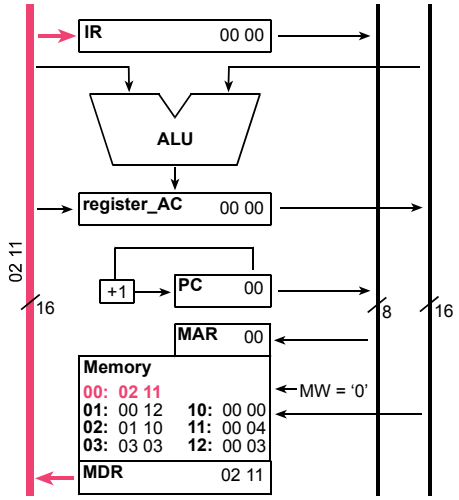
```
DEPTH = 256;    % Memory depth and width are required   %
WIDTH = 16;     % Enter a decimal number    %

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 ;
```

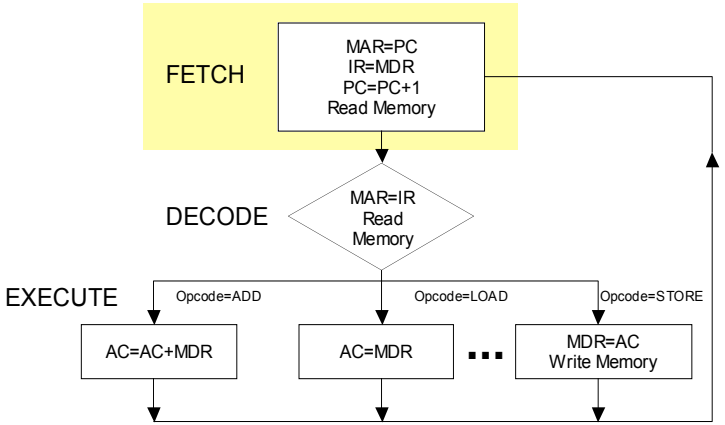
µP1 Fetch, Decode, Execute Cycle



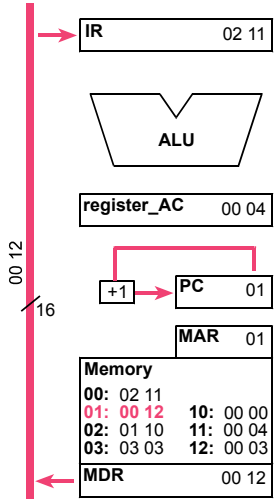
µP1 Datapath (Values After Reset)



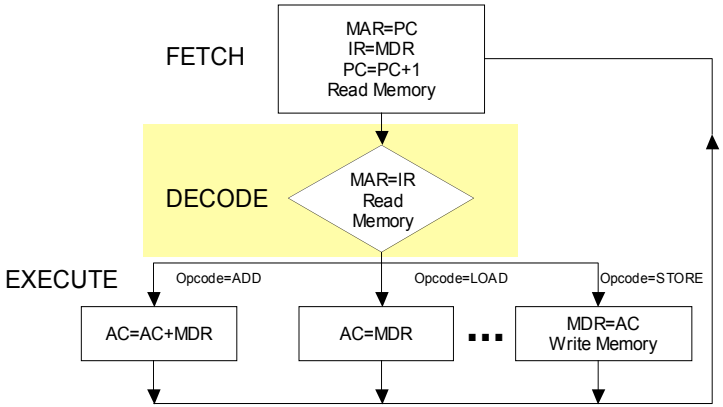
µP1 Fetch, Decode, Execute Cycle



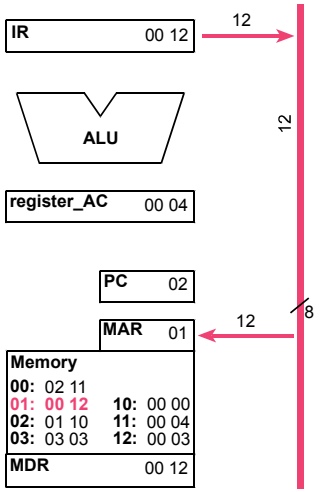
Register Transfers in ADD's Fetch State



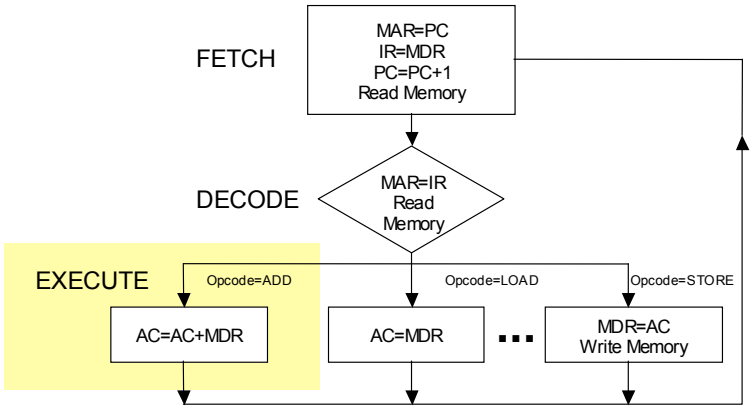
µP1 Fetch, Decode, Execute Cycle



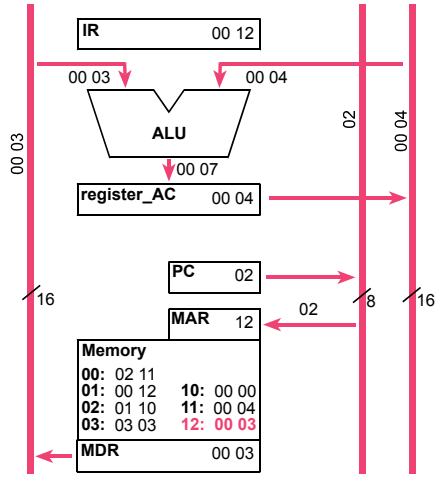
Register Transfers in ADD's Decode State



µP1 Fetch, Decode, Execute Cycle



Register Transfers in ADD's Execute State



```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY lpm;
USE lpm.lpm_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 ));
END SCOMP;
ARCHITECTURE a OF scomp IS
TYPE STATE_TYPE IS ( reset_pc, fetch, decode, execute_add, execute_load, execute_store,
                     execute_store3, 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 LPM function for computer's memory (256 16-bit words)
 memory: lpm_ram_dq
    GENERIC MAP (
          lpm_widthad          => 8,
          lpm_outdata          => "UNREGISTERED",
          lpm_indata           => "REGISTERED",
          lpm_address_control => "UNREGISTERED",
                               -- Reads in mif file for initial program and data values
          lpm_file             => "program.mif",
          lpm_width            => 16 )
    PORT MAP (  data => Register_AC, address => memory_address_register,
                we => memory_write, inclock => clock, q => memory_data_register );

          program_counter_out       <= program_counter;
          register_AC_out           <= register_AC;
          memory_data_register_out  <= memory_data_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";
   memory_address_register   <= "00000000";
   register_AC               <= "0000000000000000";
   memory_write              <= '0';
   state             <= fetch;
        -- Fetch instruction from memory and add 1 to PC
WHEN fetch =>
   instruction_register      <= memory_data_register;
   program_counter   <= program_counter + 1;
   memory_write              <= '0';
   state             <= decode;
        -- Decode instruction and send out address of any data operands
WHEN decode =>
   memory_address_register <= instruction_register( 7 DOWNTO 0);
   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;
            memory_address_register   <= program_counter;
            state                     <= fetch;
                    -- Execute the STORE instruction
                    -- (needs three clock cycles for memory write)
        WHEN execute_store =>
                    -- write register_A to memory
            memory_write              <= '1';
            state                     <= execute_store2;
                    -- This state ensures that the memory address is
                    --  valid until after memory_write goes inactive
        WHEN execute_store2 =>
            memory_write              <= '0';
            state                     <= execute_store3;
        WHEN execute_store3 =>
            memory_address_register   <= program_counter;
            state                     <= fetch;
                    -- Execute the LOAD instruction
        WHEN execute_load =>
            register_ac               <=  memory_data_register;
            memory_address_register   <= program_counter;
            state <= fetch;
                    -- Execute the JUMP instruction
        WHEN execute_jump =>
            memory_address_register   <= instruction_register( 7 DOWNTO 0 );
            program_counter           <= instruction_register( 7 DOWNTO 0 );
            state                     <= fetch;
        WHEN OTHERS =>
            memory_address_register   <= program_counter;
            state <= fetch;
        END CASE;
    END IF;
    END PROCESS;
END a;
```


Simulation of μP1 Program