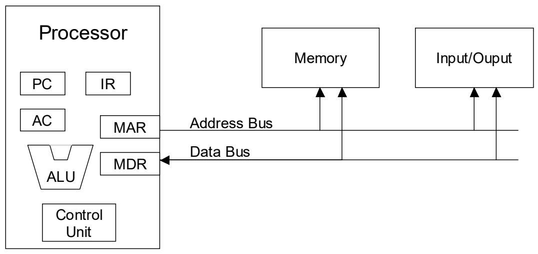
Digital Systems Design

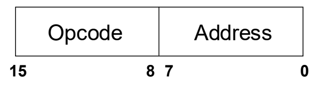
A Simple Computer Design

Architecture of a Simple Computer System

- Three main units
  - Processor (CPU)
  - Memory
    - Instruction and data
  - Input/Output hardware
    - Communicates with other devices
- CPU contains basic registers
  - PC – program counter
  - IR – instruction register
  - AC – accumulator
  - MAR – memory address register
  - MDR –memory data register



Computer Programs and Instructions



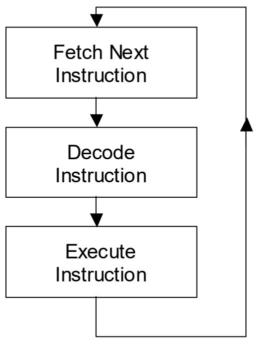
- Instructions stored in memory according to a format determined by the processor designer
  - For the simple computer, the format is 16 bits
    - 8-bit opcode – to determine the instruction
    - 8-bit address – to determine the address of one of the operands for the instruction
    - 256 word x 16-bits/word memory

Basic Computer Instructions

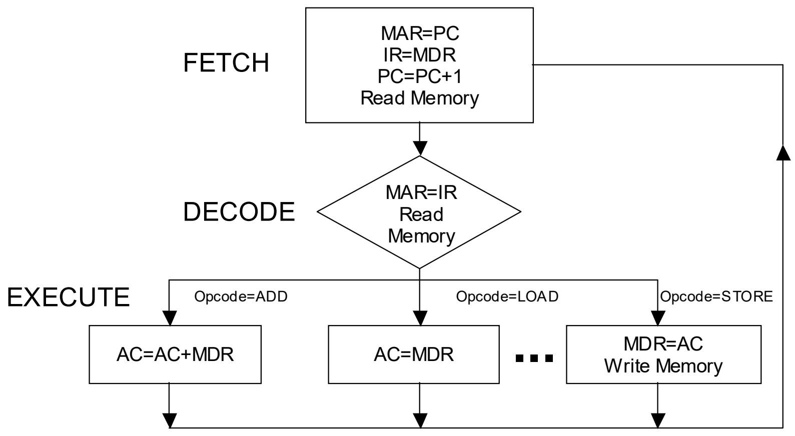
Instruction Mnemonic	Operation Preformed	Opcode Value
ADD address	AC <= AC + contents of memory address	00
STORE address	Contents of memory address <= AC	01
LOAD address	AC <= contents of memory address	02
JUMP address	PC <= address	03
JNEG address	IF AC<0 THEN PC <= address	04

Processor Fetch, Decode and Execute Cycle

- The CPU
  - Reads instructions from memory
  - Decodes the instruction
  - Carries out the operation
- A state machine (the control unit) controls the operation sequencing
- Implementing the fetch/decode/execute cycle requires
  - Several register transfers
  - Several clock cycles

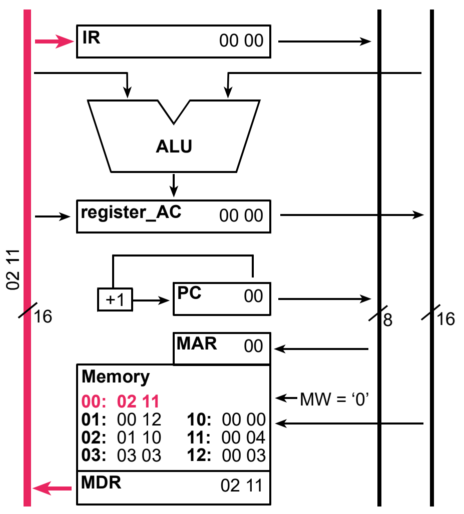


Detailed View of Fetch/Decode/Execute



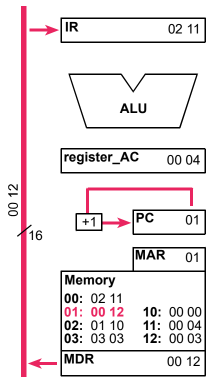
Data Values at Reset

- Reset forces the processor to a known state
- All registers will be cleared
- Since PC and MAR are reset to 00, the first instruction will be loaded from address 00



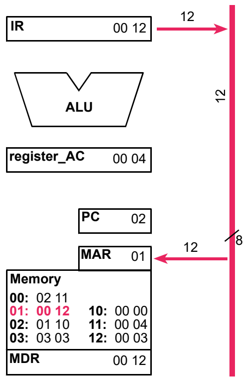
Example Instruction Execution (ADD)

- FETCH
  - MAR=PC prior to fetch
  - Read memory
  - IR=MDR
  - PC=PC+1
- After the memory access delay, the ADD instruction is available at the input to the IR
- At the same time, PC is incremented using a different bus
- In implementation, MAR=PC at the end of the fetch/decode/execute step
  - This saves a clock cycle in the FETCH step



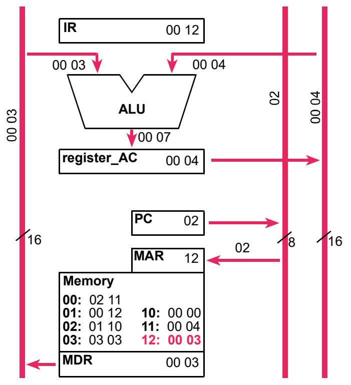
Example Instruction Execution (ADD)

- DECODE
  - Decode opcode to determine next state
  - MAR=IR (low 8-bits)
  - Begin memory read
- Instructions are decoded in hardware using
  - Combinational logic
  - A small PLA
  - ROM
- When the address is clocked into the MAR, the data from memory will be available in the MDR



Example Instruction Execution (ADD)

- EXECUTE
  - AC=AC+MDR
  - MAR=PC*
  - GOTO FETCH
- The control unit has placed the processor in the execute state for the ADD instruction
  - Control signals for the ALU instruct it to perform an ADD instruction
- Depending on the complexity of the instruction, there could be multiple steps (sub-states) for an instruction
- * By setting MAR=PC in every instruction's final execute state, a clock cycle can be saved in the FETCH state



VHDL Source Code

```
library  ieee;
use      ieee.std_logic_1164.all;
use      ieee.numeric_std.all;

entity scomp is
generic( address_width    : integer := 8;
         data_width       : integer := 16);
port(

clock, reset                 : in std_logic:= '1';
program_counter_out          : out std_logic_vector(address_width-1 downto 0);
register_ac_out              : out std_logic_vector(data_width-1 downto 0);
memory_data_register_out     : out std_logic_vector(data_width-1 downto 0);
memory_address_register_out  : out std_logic_vector(address_width-1 downto 0);
memory_write_out             : out std_logic

);
end scomp;
```

VHDL Source Code

```
architecture rtl of scomp is
type ram is array(0 to 2 ** address_width-1) of unsigned(data_width-1 downto 0);
signal     ram_block : ram;
attribute ram_init_file   : string;
attribute ram_init_file of ram_block : signal is "program.mif";

type scomp_fsm is ( reset_pc, fetch, decode, execute_add, execute_load,
                    execute_jneg,execute_jneg2, execute_store,
                     execute_store2, execute_jump );

signal state                      : scomp_fsm;
signal instruction_register       : unsigned(data_width-1 downto 0);
signal memory_data_register       : unsigned(data_width-1 downto 0);
signal register_ac                : signed(data_width-1 downto 0);
signal program_counter            : unsigned(address_width-1 downto 0);
signal memory_address_register    : unsigned(address_width-1 downto 0);
signal memory_write               : std_logic;
```

VHDL Source Code

```
begin
-- Output major signals
program_counter_out        <= std_logic_vector(program_counter);
register_AC_out            <= std_logic_vector(register_AC);
memory_data_register_out   <= std_logic_vector(memory_data_register);
memory_address_register_out <= std_logic_vector(memory_address_register);
memory_write_out           <= memory_write;

process (clock)
begin
  if rising_edge(clock) then
    if (memory_write = '1') then
      ram_block(to_integer(memory_address_register)) <= unsigned(register_ac);
    end if;
    memory_data_register <= ram_block(to_integer(memory_address_register));
  end if;
end process;
```

VHDL Source Code

```
process (clock,reset)
  begin
    if reset = '1' then
      state <= reset_pc;
    elsif rising_edge(clock) then
      case state is
       -- reset the computer, need to clear some registers
       when reset_pc =>
         program_counter   <= (others => '0');
         register_ac       <= (others => '0');
         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;
```

VHDL Source Code

```
-- 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 "00000100" =>
            state        <= execute_jneg;
    when others =>
           state         <= fetch;
  end case;
-- execute the add instruction
when execute_add =>
  register_ac     <= register_ac + signed(memory_data_register);
  state           <= fetch;
```

VHDL Source Code

```
-- execute the store instruction
-- (needs two clock cycles for memory write and fetch mem setup)
when execute_store =>
        -- enable memory write, write register_ac to memory
        -- 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      <= signed(memory_data_register);
        state            <= fetch;

-- execute the jump instruction
when execute_jump =>
        program_counter  <= instruction_register(address_width-1 downto 0);
        state            <= fetch;
```

VHDL Source Code

```
        when execute_jneg =>
          if (register_ac < 0) then
             program_counter <= instruction_register(address_width-1 downto 0);
          end if;
          state  <= execute_jneg2;
        when execute_jneg2 =>
          state <= fetch;
        when others =>
          state <= fetch;
        end case;
     end if;
   end process;
```

VHDL Source Code

```
-- 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 <= (others => '0')          when reset_pc,
         program_counter                                when fetch,
         instruction_register(address_width-1 downto 0) when decode,
         program_counter                                when execute_add,
         instruction_register(address_width-1 downto 0) when execute_store,
         program_counter                                when execute_store2,
         program_counter                                when execute_load,
         instruction_register(address_width-1 downto 0) when execute_jump,
         program_counter                                when execute_jneg,
         program_counter                                when execute_jneg2;

  with state select
         memory_write <= '1'  when execute_store,
                         '0'  when others;
end rtl;
```

Example MIF File

```
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      %
CONTENT
BEGIN
[00..FF] :        0000;    % Range--Every address from 00 to FF = 0000 %
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) %
04       :        04FF;    % JNEG FF should jump to FFH %
05       :        0305;    % JUMP to 05 (loop forever) %
06       :        0306;    % JUMP to 06 (loop forever) %
10       :        AAAA;    % Data Value %
11       :        5555;    % Data Value %
12       :        0000;    % Data Value - should be FFFF after program %
FF       :        0306;    % JUMP to 06 %
END ;
```