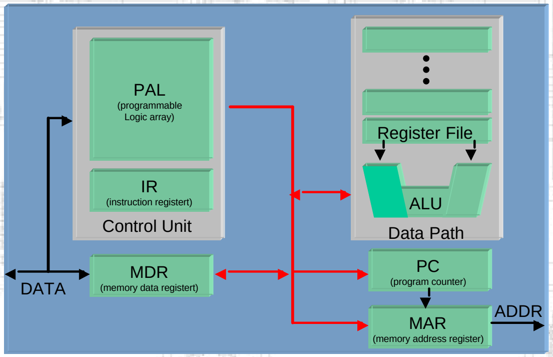
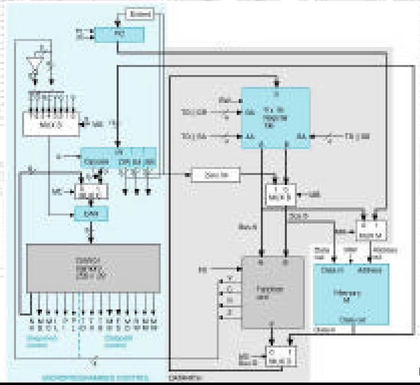
CPU



Multiple-Cycle Microprogrammed Computer



Instruction format

Four different instructions

Opcode	Operand	Operand	Operand

$R_d \leftarrow$ MEMORY[ADDRESS]

LOAD	00	R_d	Memory Address
	7 6	5 4	3 0

MEMORY[ADDRESS] $\leftarrow R_S$

STORE	01	R_S	Memory Address
	7 6	5 4	3 0

$R_d \leftarrow R_{S1} + R_{S2}$

ADDR	10	R_d	R_{s1}	R_{s2}
	7 6	5 4	3 2	1 0

$R_d \leftarrow R_d$ + MEMORY[ADDRESS]

ADDM	11	R_d	Memory Address
	7 6	5 4	3 0

Small Program

* code adds two numbers at
* memory address 13 and 14
* Saves the result at memory address 15

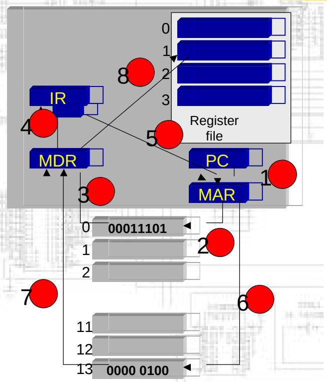
```
LOAD     1,13    * R1 ¬ MEMORY[13]
ADDM     1,14    * R1 ¬ R1 + MEMORY[14]
STORE    1,15    * MEMORY[15] ¬ R1
```

Data and Instructions in Memory

0	00 01 1101	LOAD 1,13	* R1 ← MEMORY[13]
1	11 01 1110	ADDM 1,14	* R1 ← R1 + MEMORY[14]
2	00 01 1101	STORE 1,15	* MEMORY[15] ← R1
⋮			
13	0000 0100	4	
14	0000 0010	2	
15			

Instruction and Data Streams

R1 ← MEMORY[13]



0 PC ← 0

1 MAR ← PC

2 ADDR ← MAR

3 MDR ← M[ADDR], PC ← PC + 1

4 IR ← MDR

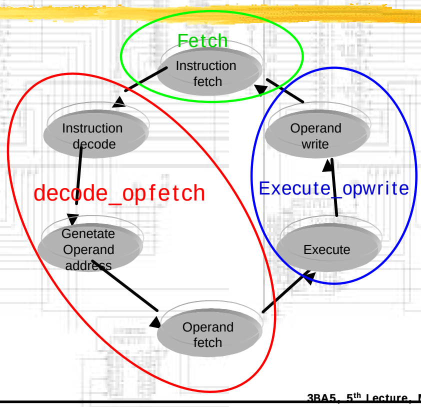
5 MAR ← IR(3 downto 0)

6 ADDR ← MDR

7 MDR ← M[ADDR]

8 R1 ← MDR

Main Phases



Control_State

Controlling the sequence of execution phases

```
-- Mehdi R. Zargham, page 27
control_state: process (inst_fetch, decode_opfetch, execute_opwrite, CLOCK)
begin
  if ((CLOCK='1') and not(CLOCK'stable)) then
    if ((not inst_fetch) and (not decode_opfetch) and (not execute_opwrite)) then
      case (next_state) is
        when "inst_fetch_st" => inst_fetch <= true;
                      next_state := 'decode_opfetch_st';
        when "decode_opfetch_st" => decode_opfetch <= true;
                      next_state := 'execute_opwrite_st';
        when "execute_opwrite_st" => execute_opwrite <= true;
                      next_state := 'inst_fetch_st';
      end case;
    end if;
  end if;
end process control_state;
```

Inst_fetch_state

Instruction fetch phase

```
-- Mehdi R. Zargham, page 28
inst_fetch_state: process
begin
  wait on inst_fetch until inst_fetch;
  MAR <= PC;
  ADDR <= MAR after 15 ns; -- set the address for desired memory location
  MR <= '1' after 15 ns;    -- sets CS1 of each memory module to 1
  RW <= '0' after 20 n;     -- read from memory
  wait for 100 ns;          -- required time to read data from from memory
  MR <= '0';
  IR <= MDR after 15 ns;
  for i in 0 to 3 loop      -- increment PC by one
    if PC(i) = '0' then
      PC(i) := '1';
      exit
    else
      PC(i) := '0';
    end if
  end loop
  inst_fetch <= false;
end process inst_fetch_state;
```

decode_opfetch_state:

-- LOAD

```
-- Mehdi R. Zargham, page 29
decode_opfetch_state: process
begin
        wait on decode_opfetch until decode_opfetch;
  case (IR(7 downto 6)) is

        -- LOAD
        when "00" => MAR <= IR(3 downto 0);
                ADDR <= MAR after 15 ns;
                MR <= '1' after 25 ns;
                RW <= '0' after 20 ns;
                wait for 100 ns; -- 100 ns is required to read mem
                MDR <= tri_vector_to_bit_vector(DATA);
                MR <= '0';
                -- copy MDR to the destination register
                reg_file(intval(IR(5 downto 4))) <= MDR;
```

decode_opfetch_state:

-- STORE

```
-- Mehdi R. Zargham, page 29
-- STORE
when "01" => MDR <= reg_file(intval(IR(5 downto 4)));
  DATA <= MDR after 20 ns;
  MAR <= IR(3 downto 0);
  MR <= '1' after 25 ns;
  ADDR <= MAR after 15 ns;
  RW <= '1' after 20 ns;
  wait for 110 ns;            -- 110 ns is required to write mem
  MR <= '0';
```

decode_opfetch_state:

-- ADDR and ADDR

```
-- Mehdi R. Zargham, page 29
    -- ADDR
    when "10" => ALU_REG1 <= reg_file(intval(IR(3 downto 2)));
      ALU_REG2 <= reg_file(intval(IR(1 downto 0)));
      add_op <= true after 20 ns;
    -- ADDM
    when "11" => ALU_REG1 <= reg_file(intval(IR(5 downto 4)));
      MAR <= IR(3 downto 0);
      ADDR <= MAR after 15 ns;
      MR <= '1' after 25 ns;
      RW <= '0' after 20 ns;
      wait for 100 ns;        -- 100 ns is required to read mem
      MDR <= tri_vector_to_bit_vector(DATA);
      MR <= '1' after 25 ns;
      ALU_REG2 <= MDR;
      add_op <= true after 20 ns;
   end case
  decode_opfetch <= false;
end process decode_opfetch_state;
```

3BA5

execute_opwrite_state :

```
-- Mehdi R. Zargham, page 30
execute_opwrite_state: process
begin
  wait on execute_opwrite until execute_opwrite;
    if add_op then
      reg_file(intval(IR(5 downto 4))) := ADD(ALU_REG1, ALU_REG2);
      add_op <= false;
    end if;
    execute_opwrite <= false;
end process execute_opwrite_state;
```