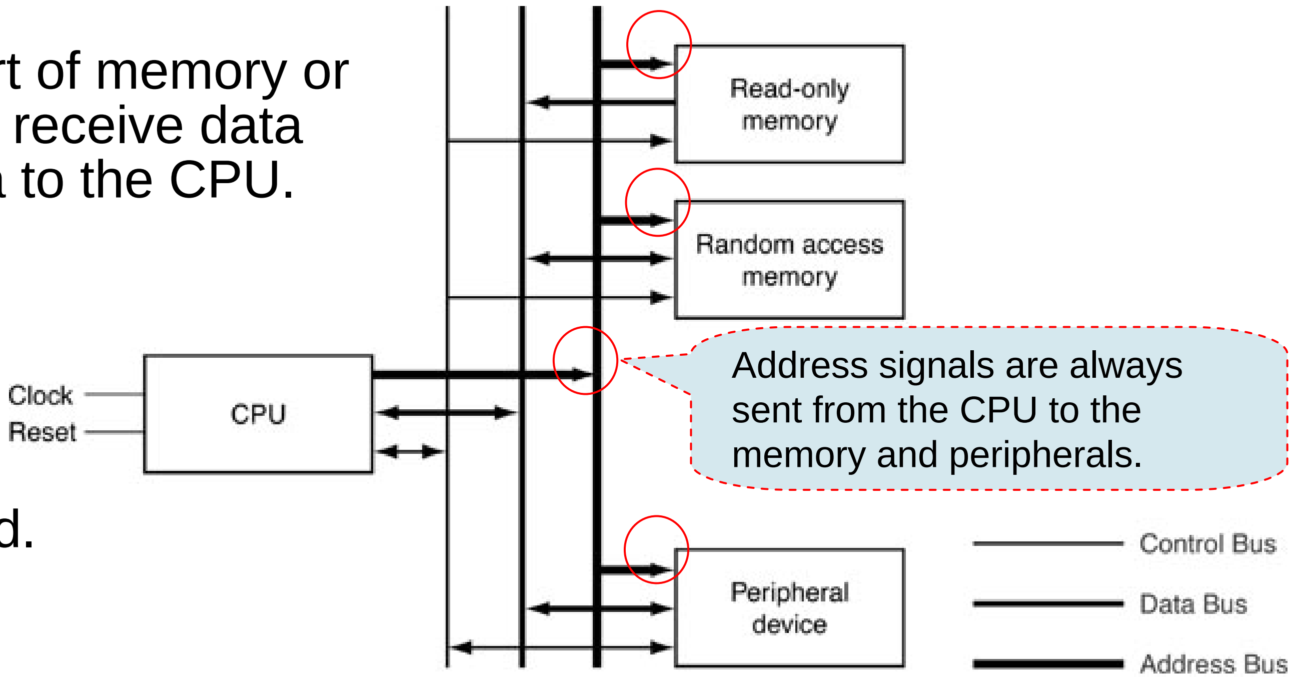
# Class 14
# Microprocessors

# Simplified Microcomputer System

- Control bus
  - A set of control lines going from the CPU to various components.
  - Direct the flow of data among modules of the microcomputer (MCU) by enabling and disabling the various data paths.
- Address bus
  - Specify which part of memory or peripheral should receive data from or send data to the CPU.
- Data bus
  - Send or receive the actual data.
  - Bus contention should be avoided.



Address signals are always sent from the CPU to the memory and peripherals.

# Asynchronous Tristate Data Bus

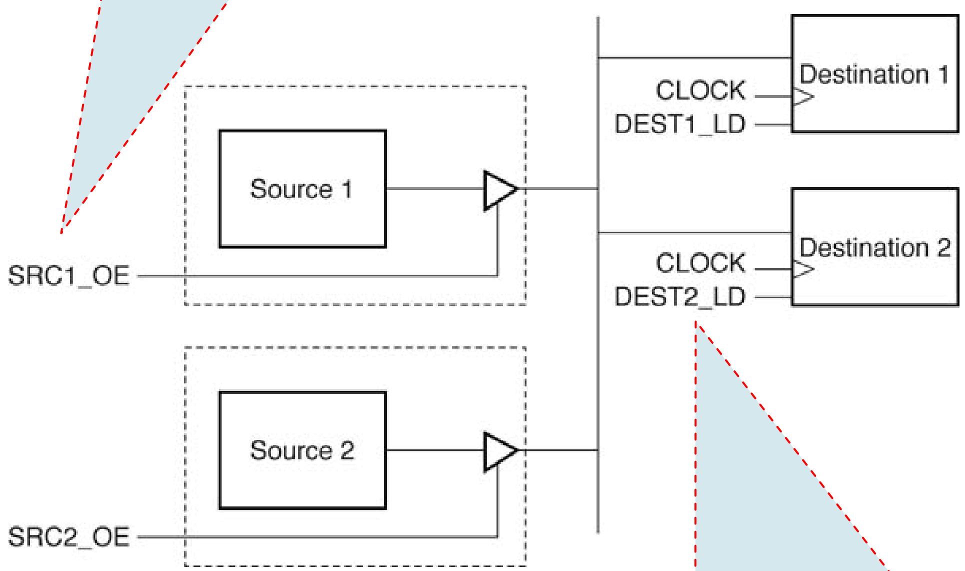
Load their data to the bus when their control lines are HIGH.

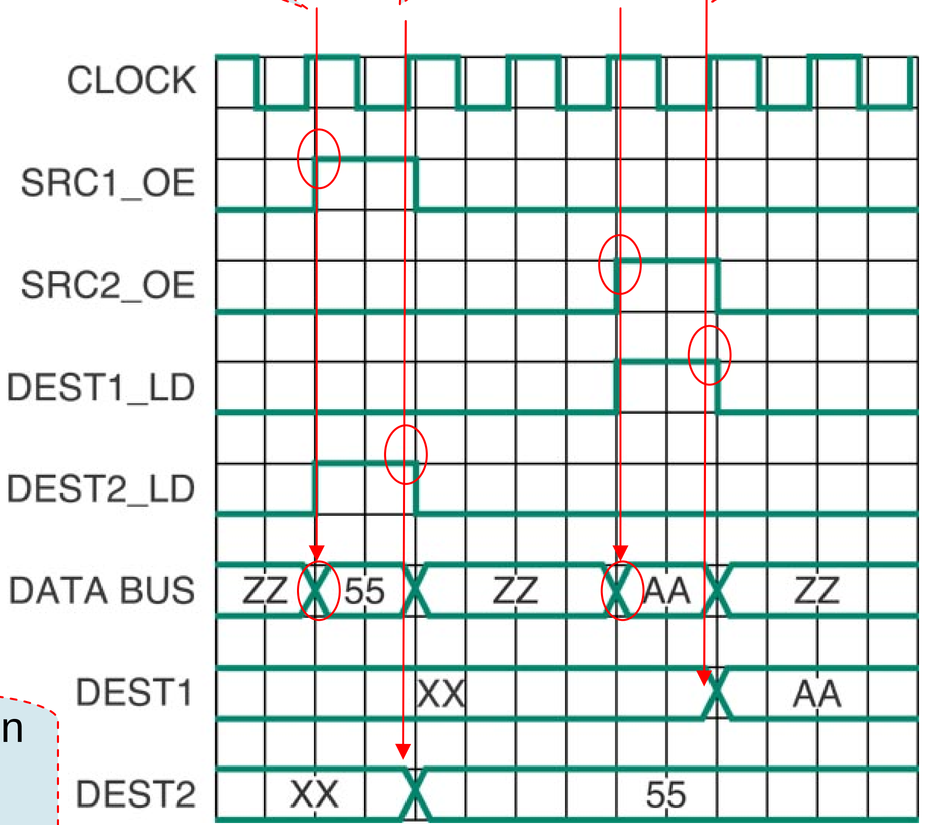
Load data to the bus

Write data to DEST2

Load data to the bus

Write data to DEST1

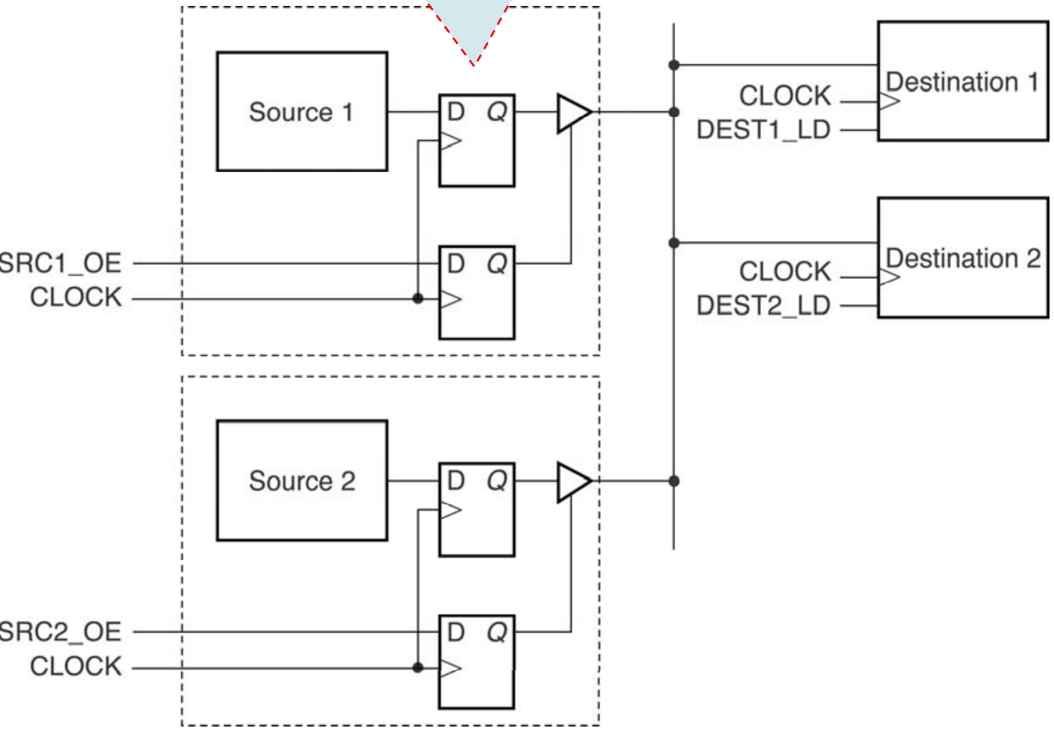




Write data from bus to the destination when their control lines are HIGH at the positive clock edge.

# Synchronous Tristate Data Bus

Flip-flops to make the source output synchronous to the system clock.

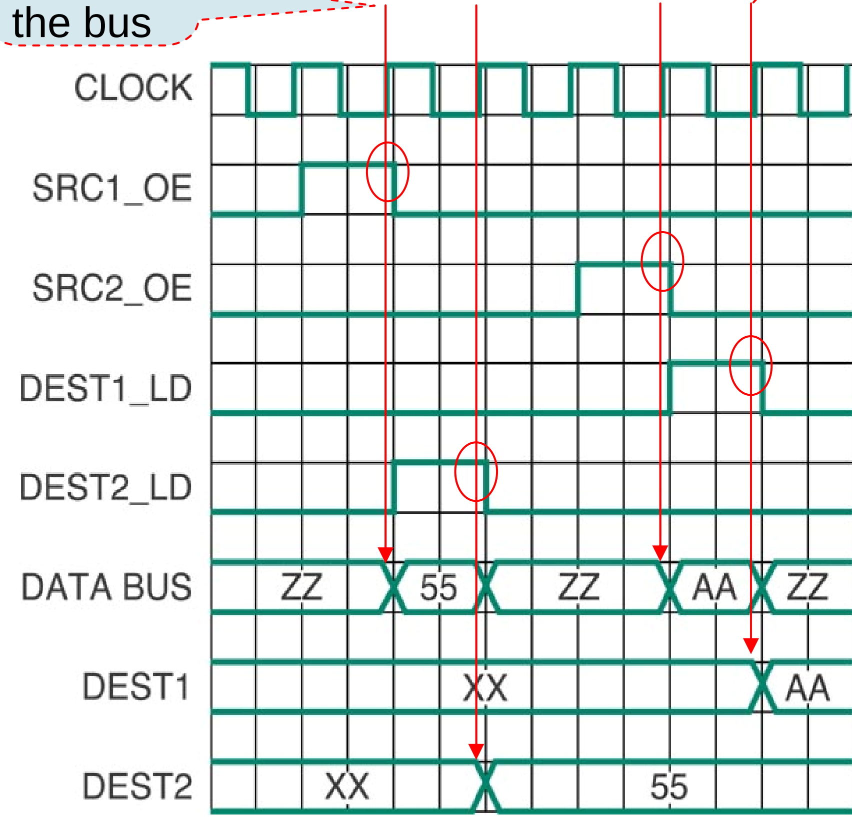


Load data to the bus

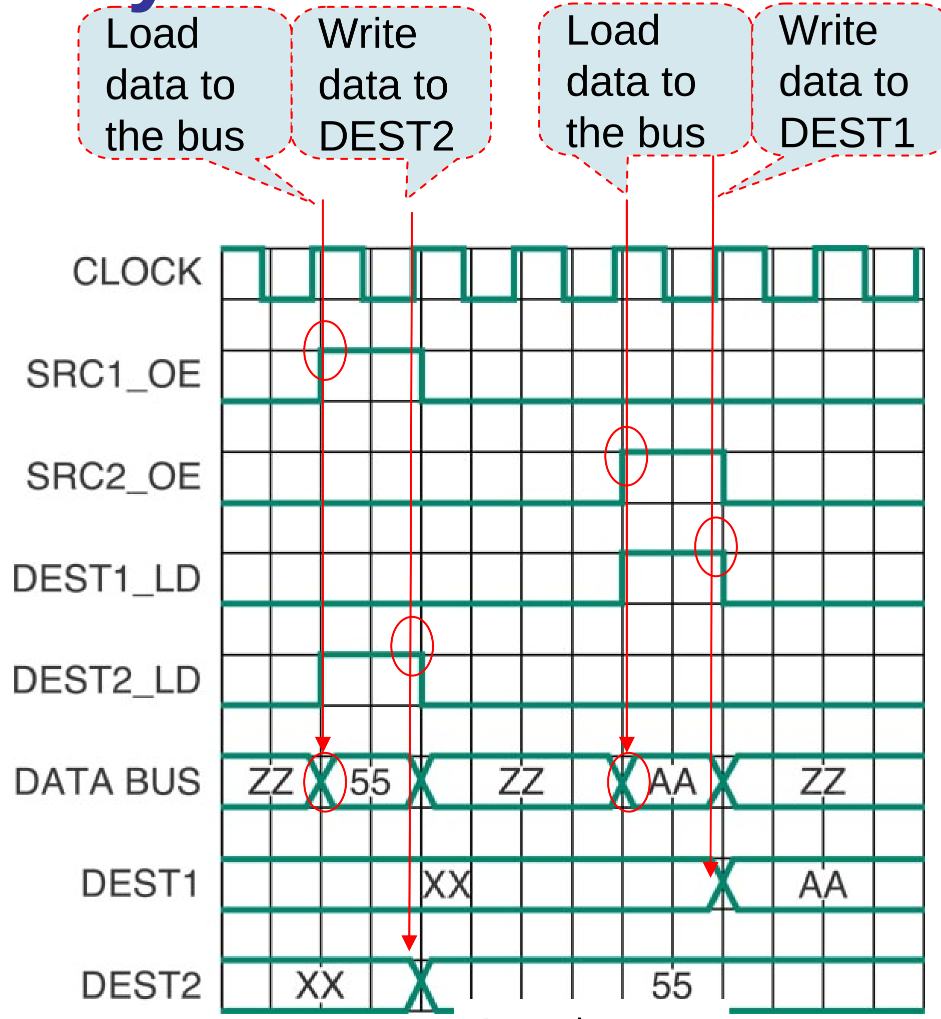
Write data to DEST2

Load data to the bus

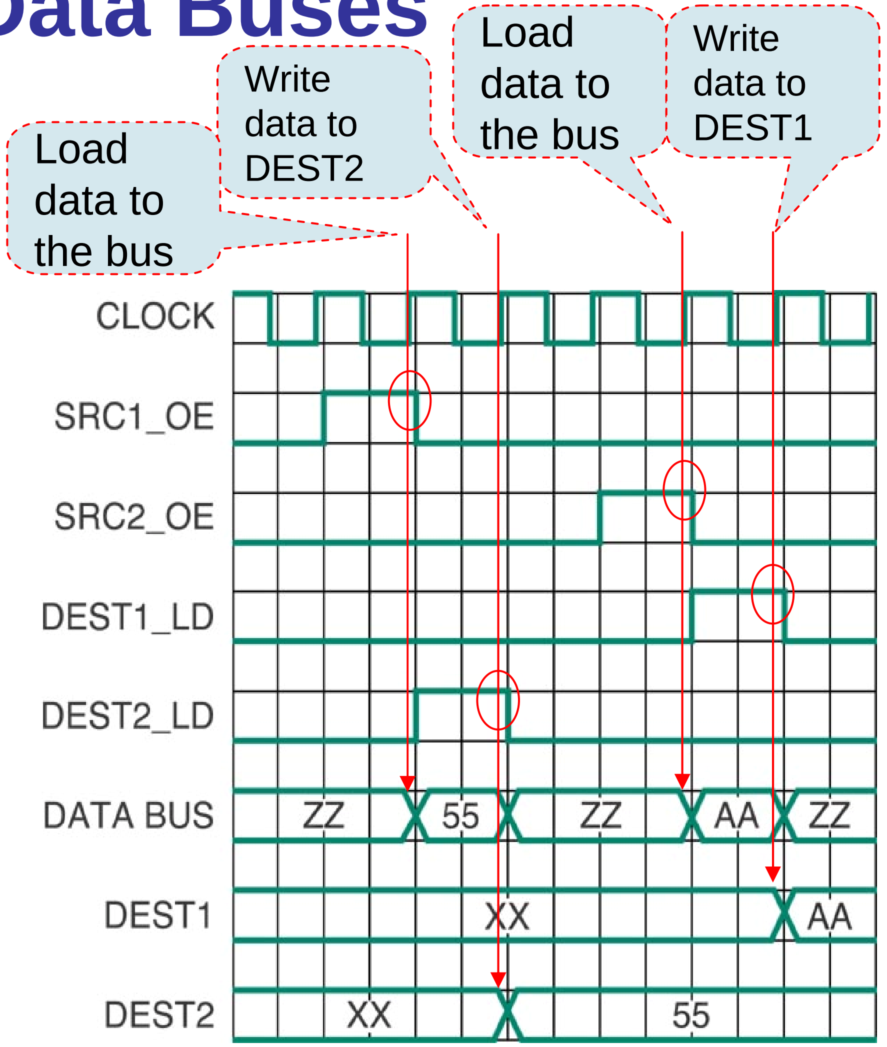
Write data to DEST1



# Comparison Between Asynchronous and Synchronous Tristate Data Buses



Asynchronous



Synchronous

# A Simple Microprocessor

A combinational circuit within the CPU that interprets the binary value of the op code in the instruction register, then direct the controller to execute the instruction.

A state machine within the CPU that generates control signals required to transfer data among the various CPU registers.

A CPU register that holds the accumulated result of arithmetic and logic operations performed in ALU.

A combinational circuit within the CPU that performs arithmetic and logical operations, such as ADD, OR, and XOR.

A CPU register that holds data transferred from memory as a second operand in an arithmetic or logical operation.

A CPU register that holds the current instruction op code being executed by the CPU.

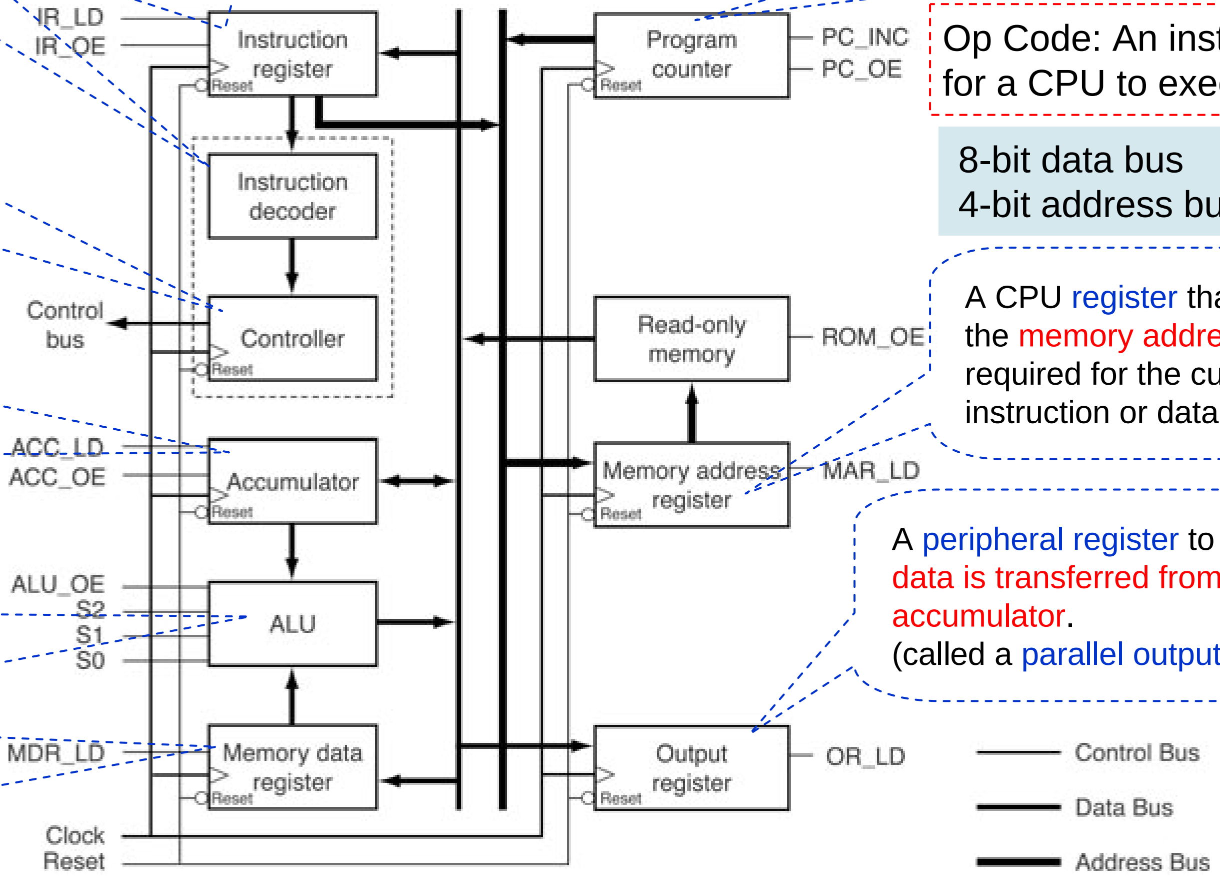
A counter within a CPU that keeps track of the address of the next program instruction to be fetched from memory.

Op Code: An instruction for a CPU to execute.

8-bit data bus
4-bit address bus

A CPU register that holds the memory address required for the current instruction or data.

A peripheral register to which data is transferred from the accumulator.
(called a parallel output port).



## A Simplified Op Codes and Program

- Every instruction is fetched and then executed

| Instructions | Op Codes (Hex Values) |
|---|---|
| Add | 1 |
| Load | 8 |
| Output | 9 |
| Halt | F |

Simplified Op Codes

| Address | Data | Comment |
|---|---|---|
| 0 | 8C | Load contents of C |
| 1 | 1D | Add contents of D |
| 2 | 90 | Send accumulator contents to output register |
| 3 | F0 | Halt |
| 4-B | Blank(00) | |
| C | 55 | Data for Load instruction |
| D | 64 | Data for Add instruction |

A Simplified Program:
Load, add, output, and halt
(Stored in a 16-byte ROM)

# Fetch Cycles

- Fetch 1:
  - Transfer the contents of the **program counter (PC)** to the 4-bit address bus.
  - Active control line: **pc_oe**.
- Fetch 2:
  - Transfer the PC address from the address bus to the **memory address register (MAR)**.
  - Increment the program counter so that it is ready to point to the next instruction.
  - Active control line: **mar_ld**, **pc_inc**.
- Fetch 3:
  - The value in MAR points to a ROM address containing an instruction to be executed. (4-bit op code and 4-bit operand address in this example)
  - The data at the pointed ROM address is transferred to the 8-bit data bus.
  - Active control line: **rom_oe**.
- Fetch 4:
  - The op code/address pair is transferred to the **instruction register (IR)** from the 8-bit data bus.
- Fetch 5:
  - This is a “do nothing” state to wait for the data to stabilize.
  - **Instruction decoder (ID)** decodes the IR contents and direct the CPU to begin executing the selected instruction.

# Fetch Cycles (Cont.)

Fetch 4: IR loads instruction/address pair from data bus (**IR_LD**)

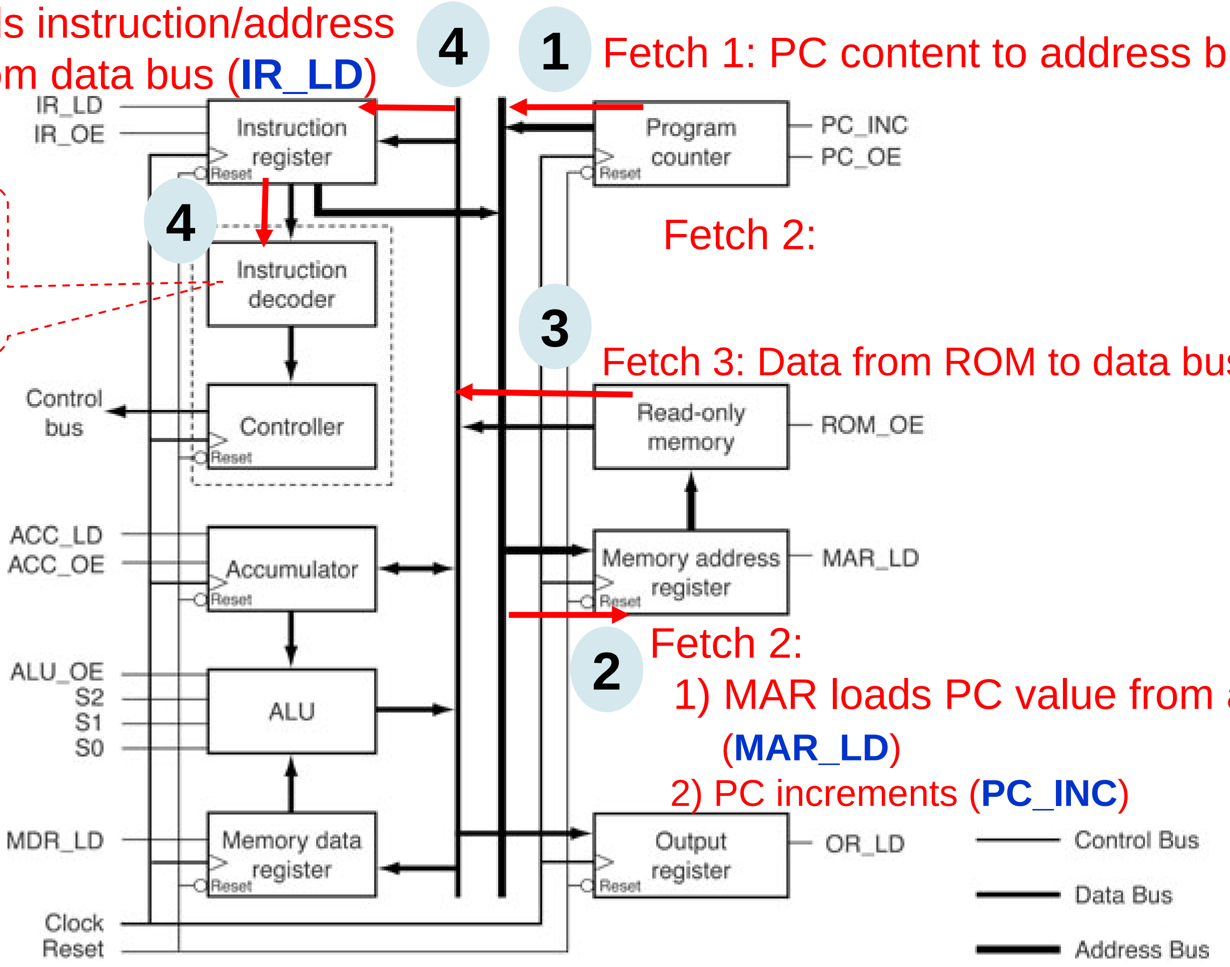
Fetch 1: PC content to address bus (**PC_OE**)

Fetch 5: wait state, and ID decodes the IR content.

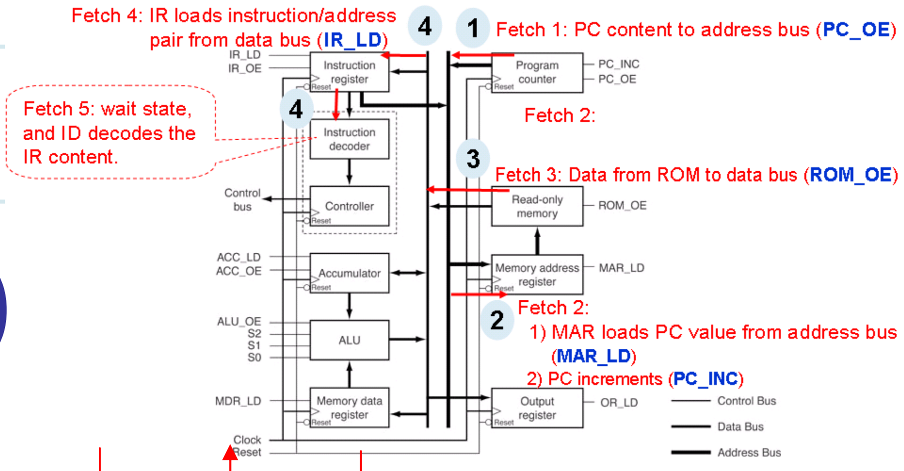
Fetch 2:

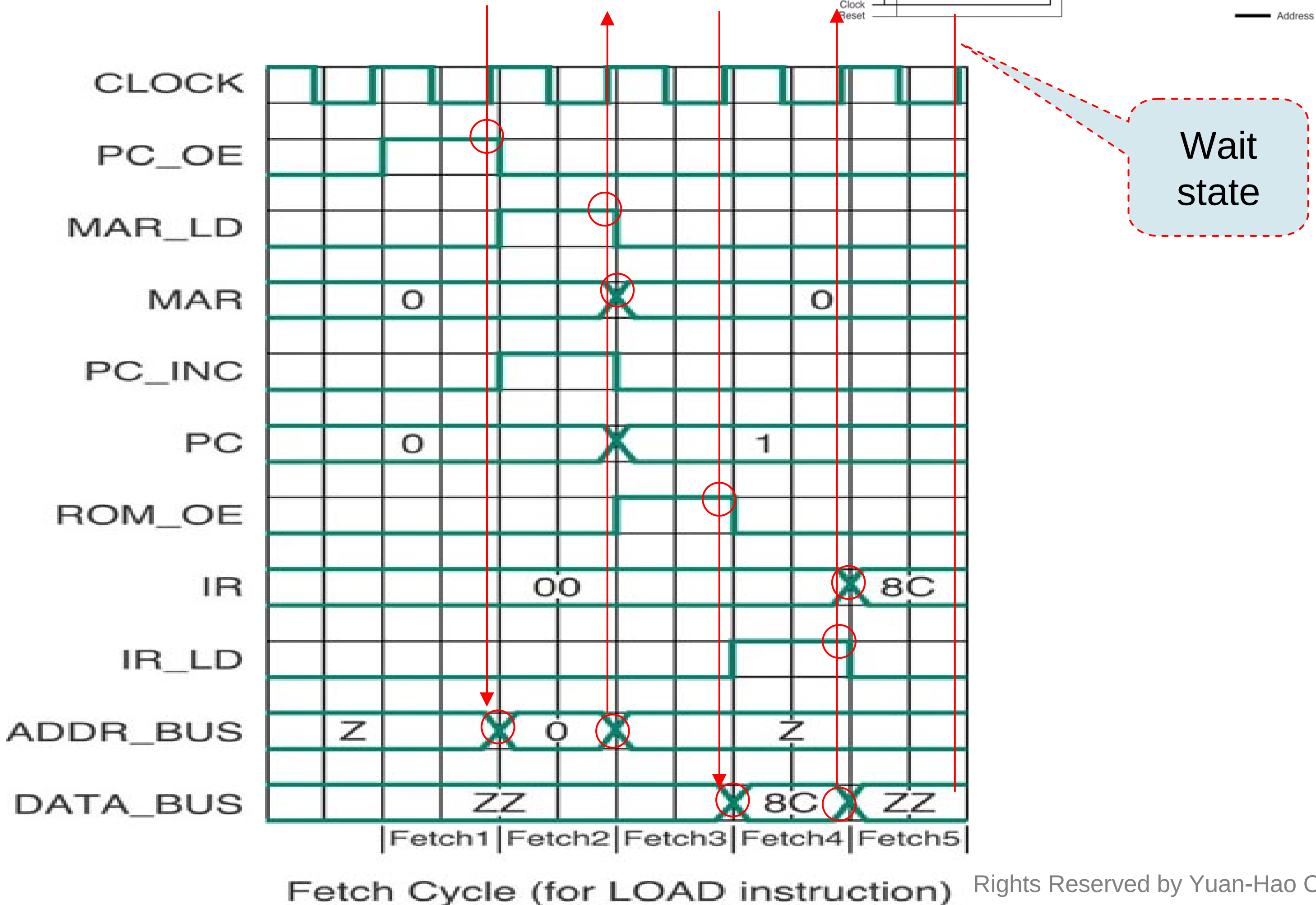
Fetch 3: Data from ROM to data bus (**ROM_OE**)

Fetch 2:
1) MAR loads PC value from address bus (**MAR_LD**)
2) PC increments (**PC_INC**)



# Fetch Cycles (Cont.)





Fetch Cycle (for LOAD instruction)

# Execute Cycle (Load Instruction)

- Load 1:
  - The instruction/operand address pair in **instruction register (IR)** is split into the op code (4MSBs) and operand address (4LSBs).
  - The op code goes by a direct connection to the **instruction decoder (ID)** in the controller state machine that determines the op code value and generate the correct control signals.
  - The operand address is transferred to the address bus.
  - Active control line: **ir_oe**.
- Load 2:
  - The MAR loads the contents of the address bus, thus latching the ROM address of the operand for the Load instruction.
  - Active control line: **rom_oe**.
- Load 3:
  - Data transfers from the ROM to data bus.
  - Active control line: **rom_oe**.
- Load 4:
  - Data transfers from the data bus to the accumulator.
  - Active control line: **acc_oe**.

# Execute Cycle (Load Instruction) (Cont.)

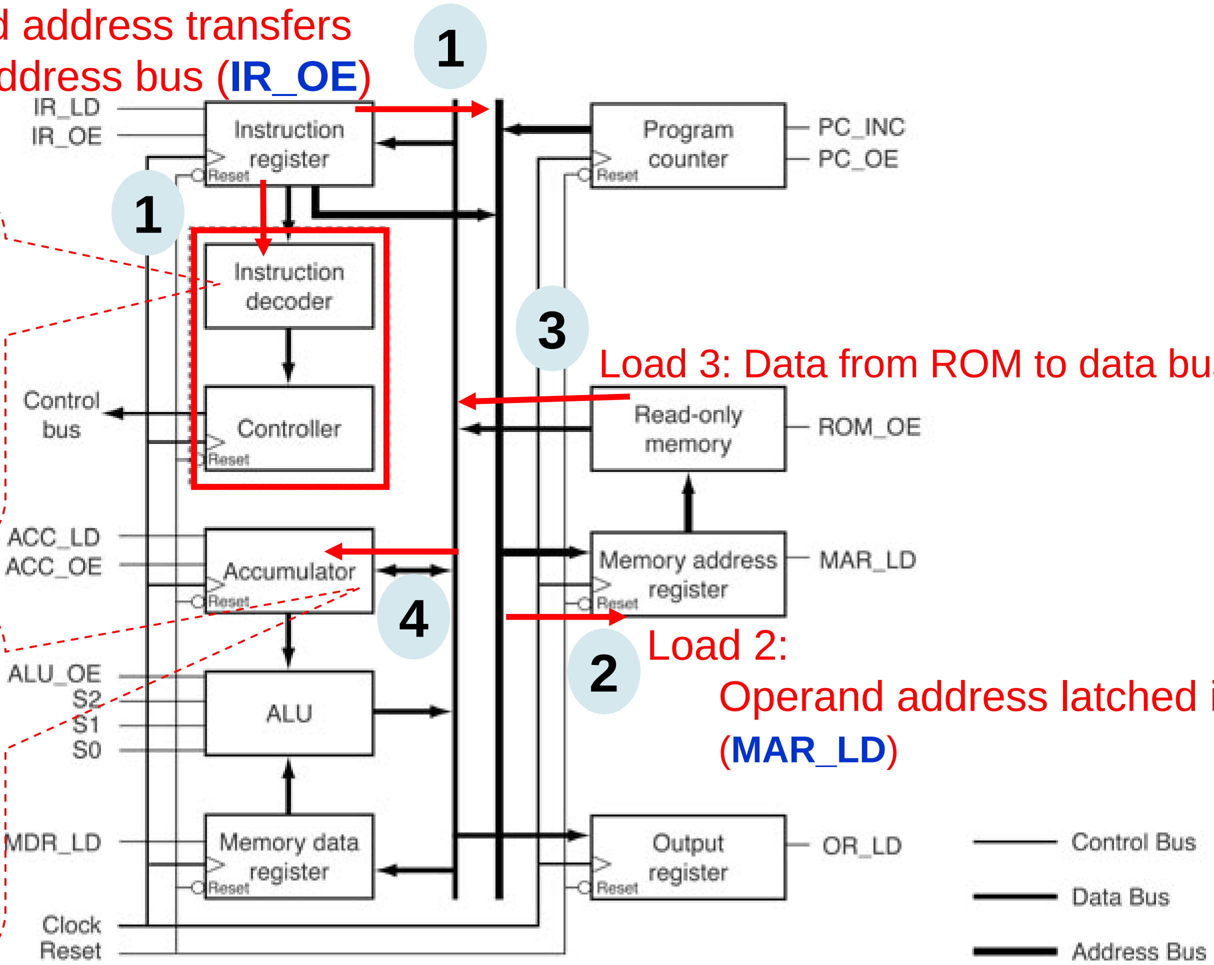
Load 1: operand address transfers from IR to address bus (**IR_OE**)

Load 1:ID decodes the op code from IR to let the controller state machine generate the correct control signals.

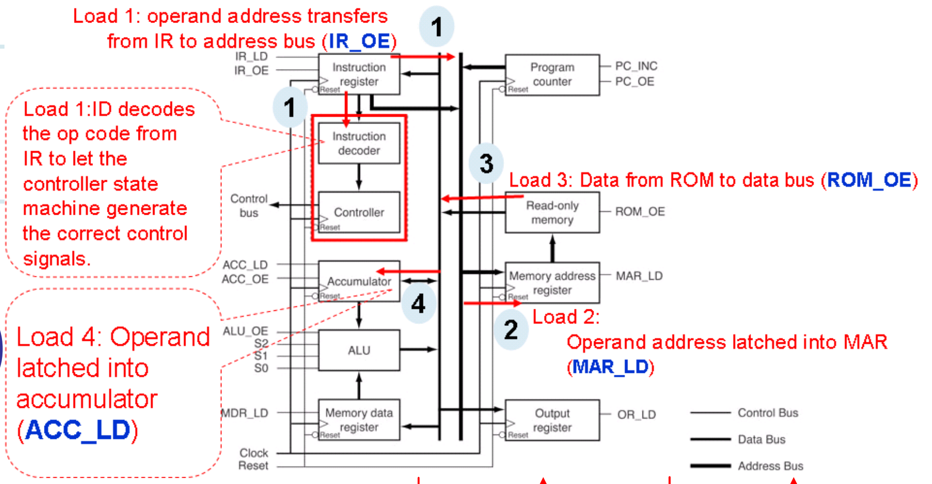
Load 3: Data from ROM to data bus (**ROM_OE**)

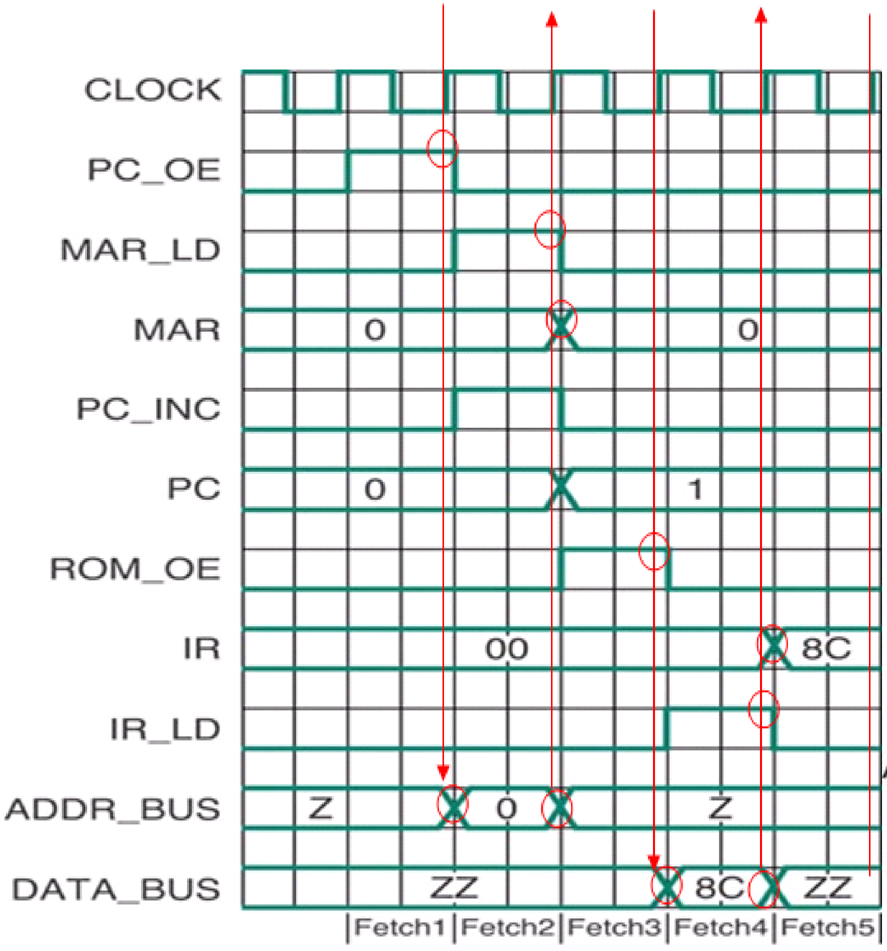
Load 2: Operand address latched into MAR (**MAR_LD**)

Load 4: Operand latched into accumulator (**ACC_LD**)

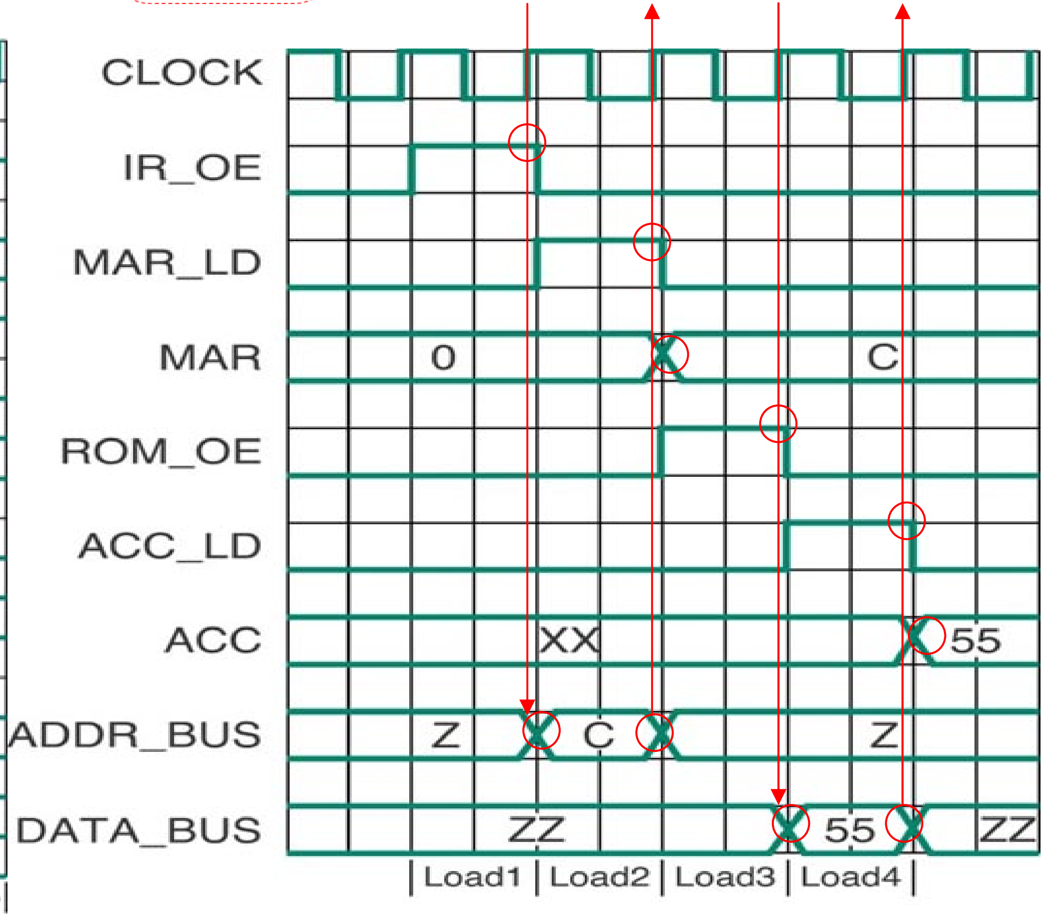


# Execute Cycle (Load Instruction) (Cont.)


Load 1: operand address transfers from IR to address bus (IR_OE)
1
Load 1:ID decodes the op code from IR to let the controller state machine generate the correct control signals.
3
Load 3: Data from ROM to data bus (ROM_OE)
Load 4: Operand latched into accumulator (ACC_LD)
4
2
Load 2: Operand address latched into MAR (MAR_LD)
IR_LD
IR_OE
Instruction register
Instruction decoder
Controller
Control bus
ACC_LD
ACC_OE
Accumulator
ALU_OE
S2
S1
S0
ALU
MDR_LD
Memory data register
Clock
Reset
Program counter
PC_INC
PC_OE
Read-only memory
ROM_OE
Memory address register
MAR_LD
Output register
OR_LD
Control Bus
Data Bus
Address Bus



Fetch Cycle (for LOAD instruction)



Execute Cycle (Load)

# Execute Cycle (Add Instruction)

- Add 1:
  - The instruction/operand address pair in **instruction register (IR)** is split into the op code (4MSBs) and operand address (4LSBs).
  - The op code goes by a direct connection to the **instruction decoder (ID)** in the controller state machine that determines the op code value and generate the correct control signals.
  - The operand address is transferred to the address bus.
  - Active control line: **ir_oe , s[2..0]=001**.
- Add 2:
  - The MAR loads the contents of the address bus, thus latching the ROM address of the operand for the Load instruction.
  - Active control line: **rom_oe , s[2..0]=001**.
- Add 3:
  - Data transfers from the ROM to data bus.
  - Active control line: **rom_oe , s[2..0]=001**.
- Add 4:
  - Transfer data from data bus to the **memory data register (MDR)**
  - Active control line: **acc_oe, s[2..0]=001**.
- Add 5:
  - The ALU adds the accumulator contents to the MDR contents.
  - The results transfers to the data bus.
  - Active control line: **alu_oe, s[2..0]=001**.
- Add 6:
  - The accumulator transfers the final result om the data bus to accumulator..
  - Active control line: **acc_ld, s[2..0]=001**.

| S[2..0] | Function | Operation |
|---|---|---|
| 000 | Increment | Acc + 1 |
| **001** | **Add** | **Acc + MDR** |
| 010 | Subtract | Acc - MDR |
| 011 | Decrement | Acc - 1 |
| 100 | Complement | NOT Acc MDR |
| 101 | AND | Acc AND |
| 110 | OR | Acc OR MDR |
| 111 | XOR | Acc XOR MDR |

**ALU**

# Execute Cycle (Add Instruction) (Cont.)

Add 1: operand address transfers from IR to address bus (**IR_OE**)

Load 1:ID decodes the op code from IR to let the controller state machine generate the correct control signals.

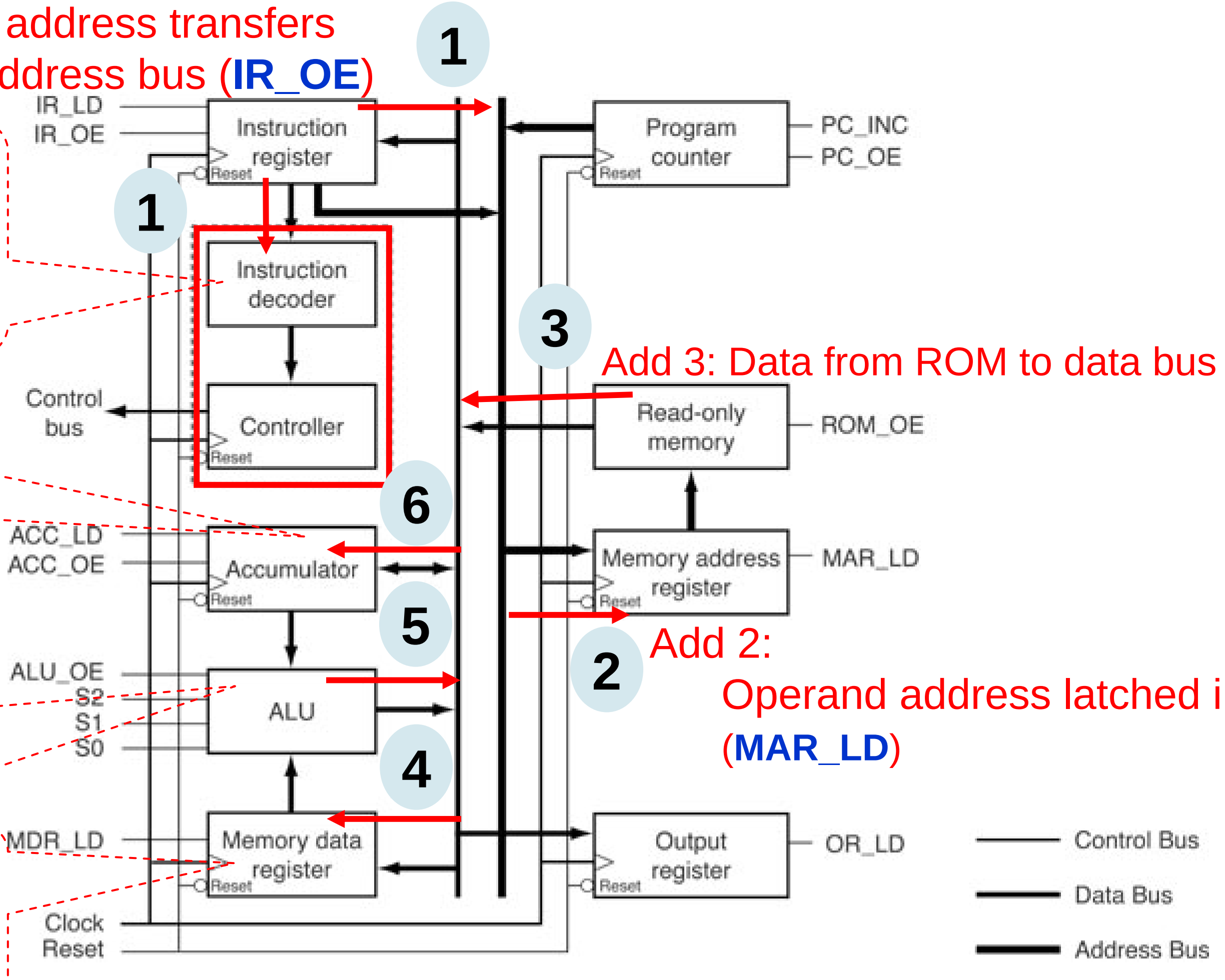
Add 6: Final result transferred from data bus to Acc (**ACC_LD**)

Add 5: Add Acc contents to the MDR contents, and send result to data bus (**ALU_OE, S[2..0]=001**)

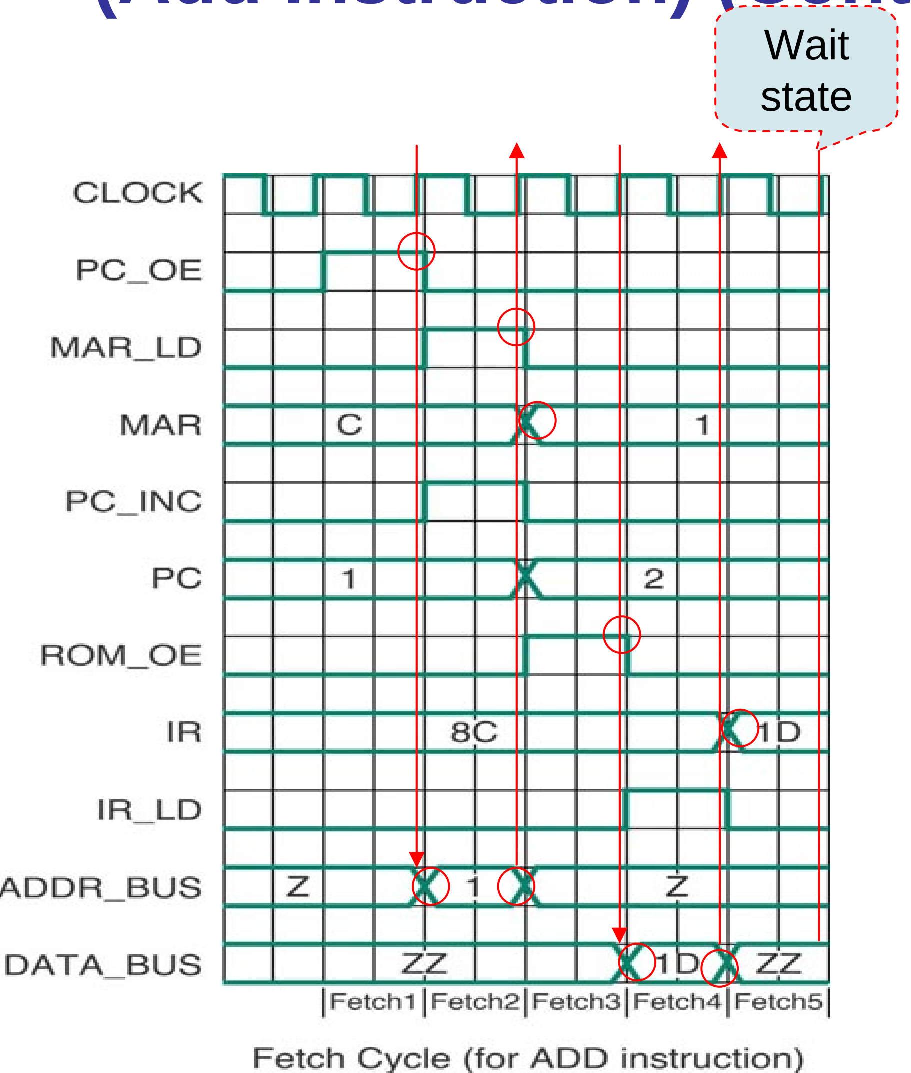
Add 4: Data transferred from data bus to MDR (**MDR_LD**)

Add 3: Data from ROM to data bus (**ROM_OE**)

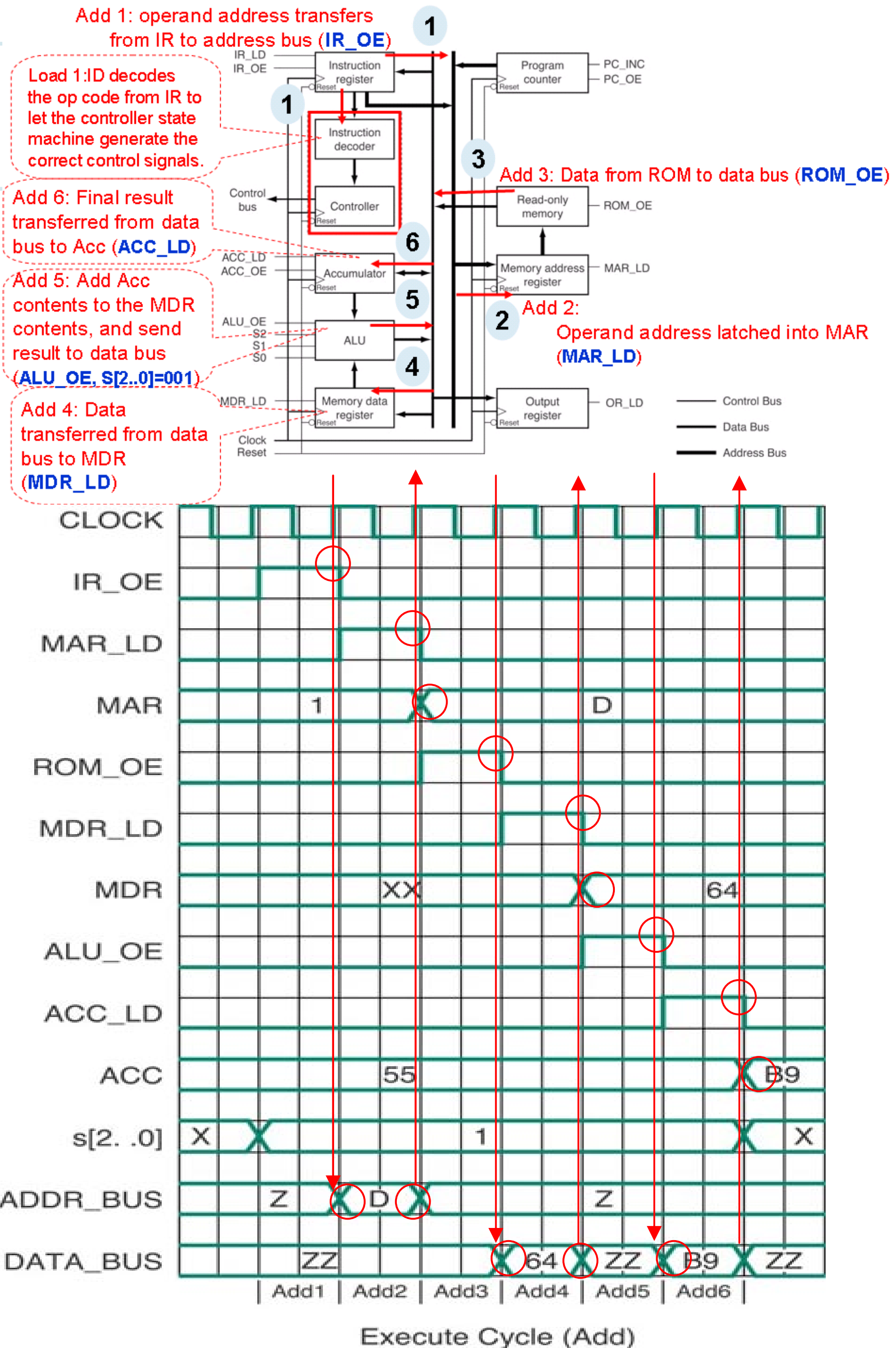
Add 2: Operand address latched into MAR (**MAR_LD**)



# Execute Cycle (Add Instruction) (Cont.)



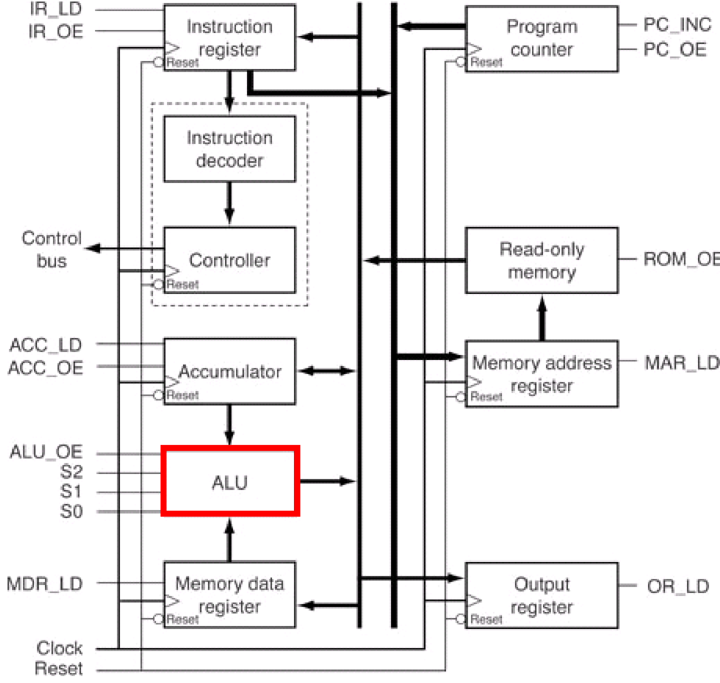
Fetch Cycle (for ADD instruction)


Add 1: operand address transfers from IR to address bus (IR_OE)
Load 1: ID decodes the op code from IR to let the controller state machine generate the correct control signals.
Add 3: Data from ROM to data bus (ROM_OE)
Add 6: Final result transferred from data bus to Acc (ACC_LD)
Add 5: Add Acc contents to the MDR contents, and send result to data bus (ALU_OE, S[2..0]=001)
Add 2: Operand address latched into MAR (MAR_LD)
Add 4: Data transferred from data bus to MDR (MDR_LD)
IR_LD
IR_OE
Instruction register
Instruction decoder
Controller
Control bus
ACC_LD
ACC_OE
Accumulator
ALU_OE
S2
S1
S0
ALU
MDR_LD
Memory data register
Clock
Reset
Program counter
PC_INC
PC_OE
Read-only memory
ROM_OE
Memory address register
MAR_LD
Output register
OR_LD
Control Bus
Data Bus
Address Bus
CLOCK
IR_OE
MAR_LD
MAR
1
D
ROM_OE
MDR_LD
MDR
XX
64
ALU_OE
ACC_LD
ACC
55
B9
s[2..0]
X
1
X
ADDR_BUS
Z
D
Z
DATA_BUS
ZZ
64
ZZ
B9
ZZ
Add1
Add2
Add3
Add4
Add5
Add6

Execute Cycle (Add)

# ALU

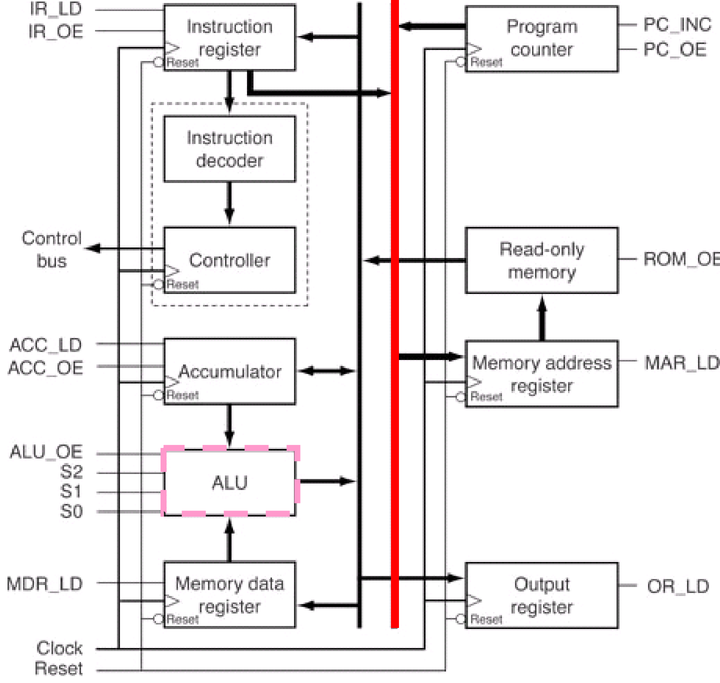
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY alu IS
PORT(
   operand_a     : IN STD_LOGIC_VECTOR(7 downto 0);
   s : IN            STD_LOGIC_VECTOR(2 downto 0);
   memory_data: IN STD_LOGIC_VECTOR(7 downto 0);
   alu_data       : OUT STD_LOGIC_VECTOR(7 downto 0));
END alu;

ARCHITECTURE a OF alu IS
BEGIN
   PROCESS (operand_a, memory_data, s)
   BEGIN
      CASE s IS
         WHEN "000" =>alu_data <= operand_a + 1; -- Increment A
         WHEN "001" =>alu_data <= operand_a + memory_data;  -- Add
         WHEN "010" =>alu_data <= operand_a - memory_data;   -- Subtract
         WHEN "011" =>alu_data <= operand_a - 1; -- Decrement A
         WHEN "100" =>alu_data <= not operand_a; -- Complement A
         WHEN "101" =>alu_data <= operand_a and memory_data;                -- AND
         WHEN "110" =>alu_data <= operand_a or memory_data; -- OR
         WHEN "111" =>alu_data <= operand_a xor memory_data;                 -- XOR
         WHEN others =>alu_data <= (others => '0');
      END CASE;
   END PROCESS;
END a;
```



-- Arithmetic Logic Unit
-- Capable of implementing 4 arithmetic and 4 logic functions
-- Input: from accumulator and memory data register
-- Output: tristate data bus via Data_MUX

# Address Bus

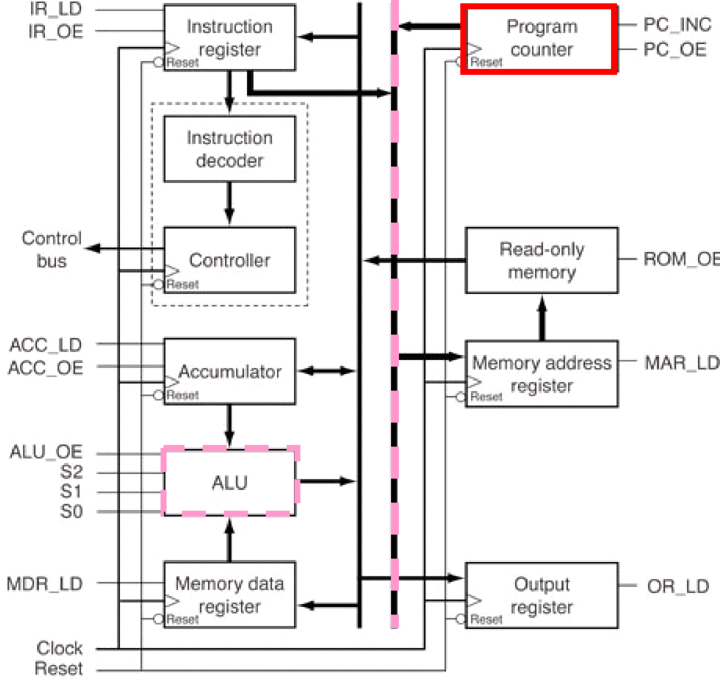
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY address_mux IS
   PORT(
      clock, reset, ir_oe, pc_oe : IN  STD_LOGIC;
      ir_addr, pc_addr : IN  STD_LOGIC_VECTOR(3 downto 0);
      addr_bus  : OUT STD_LOGIC_VECTOR(3 downto 0));
END address_mux;
ARCHITECTURE mux OF address_mux IS
   SIGNAL controls : STD_LOGIC_VECTOR(1 downto 0);
BEGIN
   -- Concatenate output enable lines to use in
   -- selected signal assignment statement
   controls <= pc_oe & ir_oe;
   PROCESS(clock, reset)
   BEGIN
      IF(reset = '0')THEN
         addr_bus <= (others => 'Z');
      ELSIF(clock'EVENT and clock = '1')THEN
         CASE controls IS
            WHEN "01" => addr_bus <= ir_addr;
            WHEN "10" => addr_bus <= pc_addr;
            WHEN others => addr_bus <= (others => 'Z');
         END CASE;
      END IF;
   END PROCESS;
END mux;
```



-- Address multiplexer with synchronous tristate outputs
-- Places instruction register or program counter contents
-- on address bus when selected. Otherwise output is high-impedance.

# Program Counter

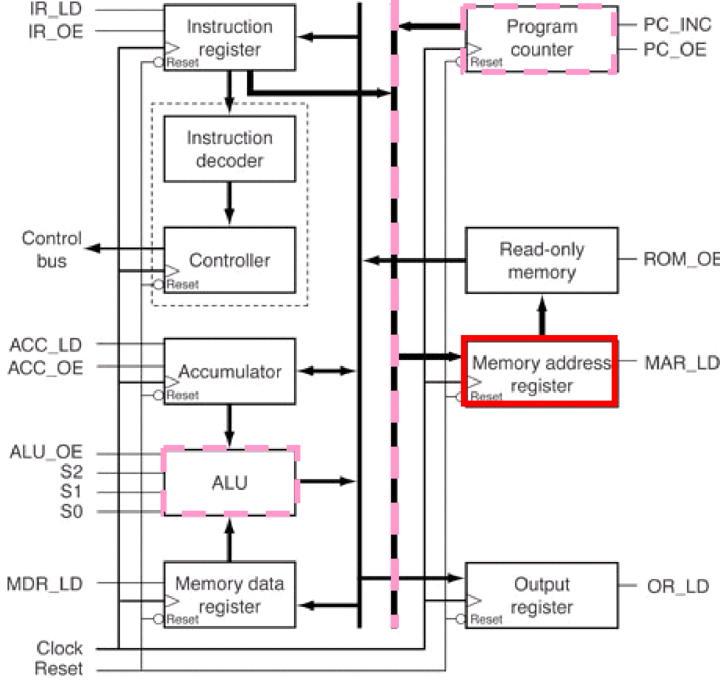
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY program_counter IS
    PORT(
        clock, reset, pc_inc : IN STD_LOGIC;
        pc_addr : BUFFER STD_LOGIC_VECTOR(3 downto 0));
END program_counter;

ARCHITECTURE pc OF program_counter IS
BEGIN
    PROCESS(clock, reset)
    BEGIN
        IF(reset = '0')THEN
            pc_addr <= (others => '0');
        ELSIF(clock'EVENT and clock = '1')THEN
            IF(pc_inc = '1')THEN
                pc_addr <= pc_addr + 1;
            END IF;
        END IF;
    END PROCESS;
END pc;
```



-- Program Counter
-- 4-bit counter with active-LOW asynchronous reset
-- Holds address of next instruction to be fetched
-- Increments when PC_INC is HIGH
-- Output is multiplexed onto tristate address bus by Address_MUX

# Memory Address Register

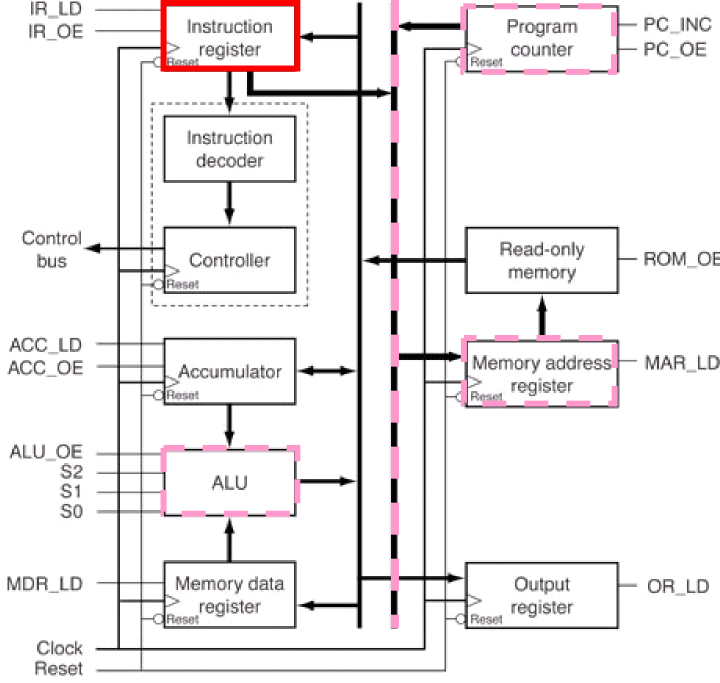
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY memory_address_register IS
    PORT(
        clock, reset, mar_ld : IN  STD_LOGIC;
        addr_bus  : IN  STD_LOGIC_VECTOR(3 downto 0);
        rom_addr  : OUT STD_LOGIC_VECTOR(3 downto 0));
END memory_address_register;

ARCHITECTURE mar OF memory_address_register IS
BEGIN
    PROCESS(reset, clock)
    BEGIN
        IF(reset = '0')THEN
            rom_addr <= (others => '0');
        ELSIF(clock'EVENT and clock = '1')THEN
            IF(mar_ld = '1')THEN
                rom_addr <= addr_bus;
            END IF;
        END IF;
    END PROCESS;
END mar;
```



```
-- Memory address register
-- 4-bit flip-flop
-- Output is always applied to
address inputs of ROM
-- Input; 4-bit address bus
-- Register has active-LOW
asynchronous reset
```

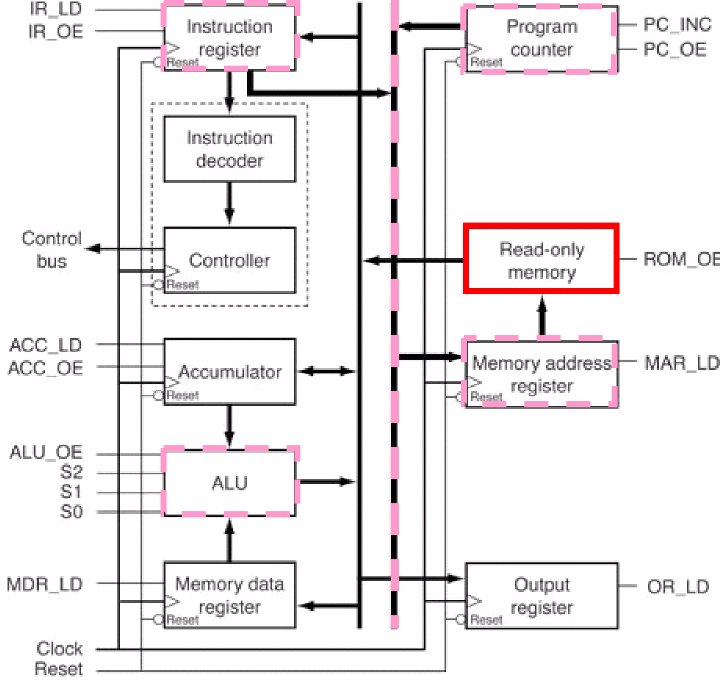
# Instruction Register

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY instruction_register IS
PORT(
   data_bus : IN  STD_LOGIC_VECTOR(7 downto 0);
   clock, reset, ir_ld : IN  STD_LOGIC;
   ir_addr: OUT STD_LOGIC_VECTOR(3 downto 0);
   instruction: OUT STD_LOGIC_VECTOR(3 downto 0));
END instruction_register;
ARCHITECTURE instr OF instruction_register IS
   -- Internal flip-flop values
   SIGNAL q_int : STD_LOGIC_VECTOR(7 downto 0);
BEGIN
   PROCESS(clock, reset)
   BEGIN
      IF (reset = '0') THEN
         q_int <= (others => '0'); -- Clear all register bits
      ELSIF (clock'EVENT and clock = '1') THEN
         -- Load register on positive clock edge, if input is enabled
         IF (ir_ld = '1') THEN
            q_int <= data_bus;
         END IF;
      END IF;
   END PROCESS;
   -- Split data into opcode and operand address
   instruction <= q_int(7 downto 4);
   ir_addr <= q_int(3 downto 0);
END instr ;
```



-- Instruction register
-- 8-bit flip-flop with input and output enables
-- Input:op code (4-bit) and address of operand (4-bit) from system ROM
-- Output: opcode to Instruction Decoder module in controller, address to tristate address bus
-- Register has active-LOW asynchronous reset

# Read Only Memory (ROM)



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY rom16b IS
PORT(
   rom_addr : IN  STD_LOGIC_VECTOR(3 downto 0);
   rom_data : OUT  STD_LOGIC_VECTOR(7 downto 0));
END rom16b;

ARCHITECTURE r OF rom16b IS
BEGIN
   WITH rom_addr SELECT
   rom_data <=
          x"8C" WHEN x"0",
          x"1D" WHEN x"1",
          x"90" WHEN x"2",
          x"F0" WHEN x"3",
          x"55" WHEN x"C",
          x"64" WHEN x"D",
          x"00" WHEN others;
END r ;
```

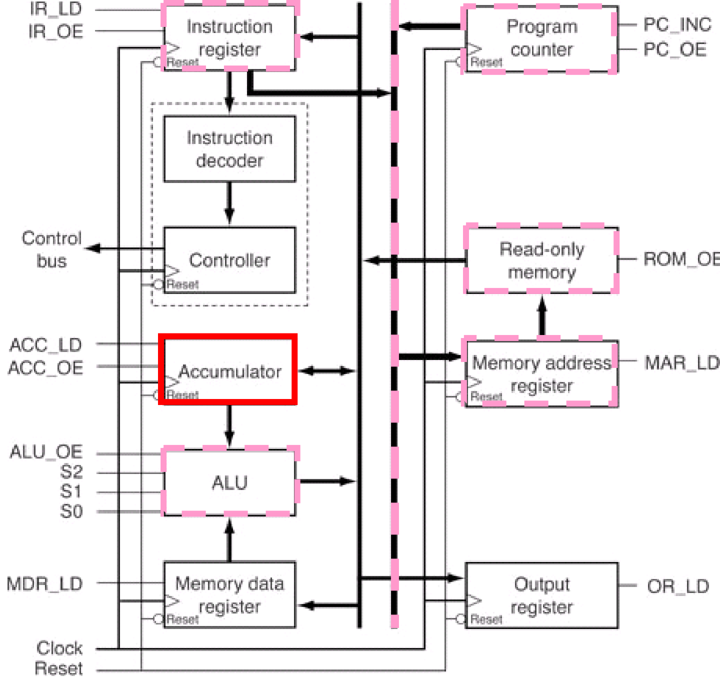
| Address | Data | Comment |
|---|---|---|
| 0 | 8C | Load contents of C |
| 1 | 1D | Add contents of D |
| 2 | 90 | Send accumulator contents to output register |
| 3 | F0 | Halt |
| 4-B | Blank(00) | |
| C | 55 | Data for Load instruction |
| D | 64 | Data for Add instruction |
| E-F | Blank(00) | |

A Simplified Program:
Load, add, output, and halt
(Stored in a 16-byte ROM)

## Accumulator

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY accumulator IS
    PORT(
        clock, reset, acc_ld : IN  STD_LOGIC;
        data_bus: IN  STD_LOGIC_VECTOR(7 downto 0);
        acc_data, operand_a  : OUT STD_LOGIC_VECTOR(7 downto 0));
END accumulator;

ARCHITECTURE acc OF accumulator IS
BEGIN
    PROCESS(clock, reset)
    BEGIN
        IF(reset = '0')THEN
            acc_data  <= (others => '0');
            operand_a <= (others => '0');
        ELSIF(clock'EVENT and clock = '1')THEN
            IF(acc_ld = '1')THEN
                acc_data  <= data_bus;
                operand_a <= data_bus;
            END IF;
        END IF;
    END PROCESS;
END acc;
```

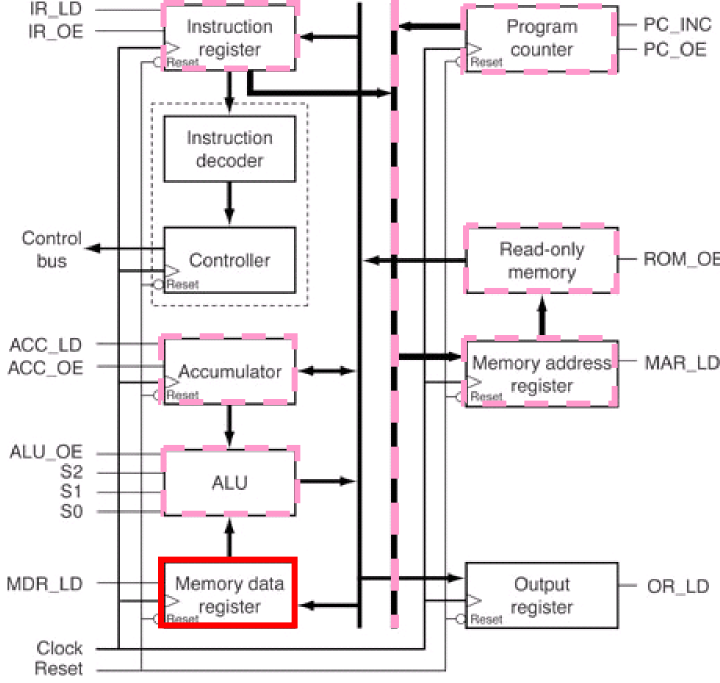
Data on Acc is directly applied to the ALU's operand.



-- Accumulator
-- 8-bit flip-flop with input and output enables
-- Input: 8-bit operand from data bus
-- Output: 8-bit operand to aLU (direct connection) and 8-bit output to data bus
-- Register has active-LOW asynchronous reset

# Memory Data Register

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY memory_data_register IS
PORT(
    data_bus : IN  STD_LOGIC_VECTOR(7 downto 0);
    clock, reset, mdr_ld  : IN  STD_LOGIC;
    memory_data : OUT STD_LOGIC_VECTOR(7 downto 0));
END memory_data_register;

ARCHITECTURE mdr OF memory_data_register IS
BEGIN
    PROCESS(clock, reset)
    BEGIN
        IF(reset = '0')THEN
            -- Clear all register bits
            memory_data <= (others => '0');
        ELSIF (clock'EVENT and clock = '1') THEN
            -- Load register on positive clock edge, if input is enabled
            IF(mdr_ld = '1')THEN
                memory_data <= data_bus;
            END IF;
        END IF;
    END PROCESS;
END mdr ;
```



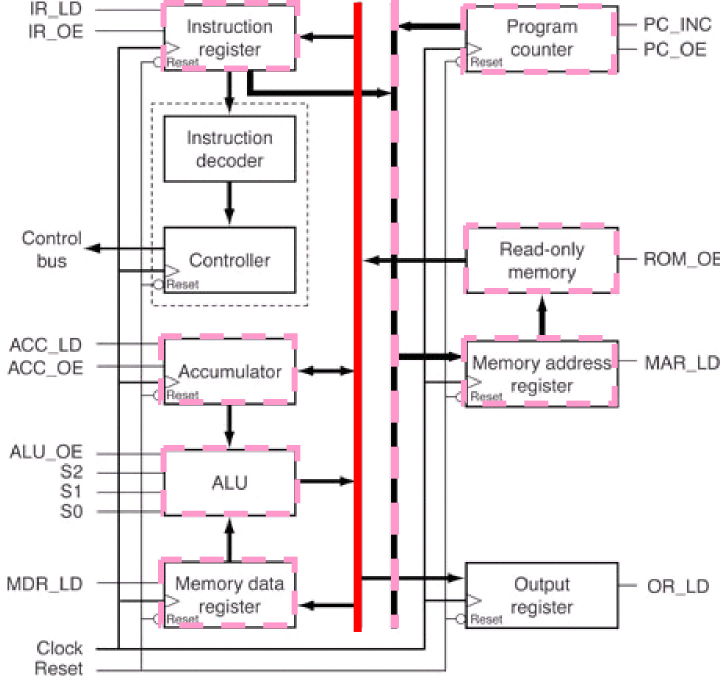
-- 8-bit flip-flop with input
-- Output is always applied to "B" inputs of ALU
-- Input: 8-bit data bus
-- Register has active-LOW asynchronous reset

# Data Bus

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY data_mux IS
   PORT(
      clock, reset : IN STD_LOGIC;
      rom_data : IN STD_LOGIC_VECTOR(7 downto 0); -- ROM instruction/data
      acc_data : IN STD_LOGIC_VECTOR(7 downto 0); -- Accumulator contents
      alu_data : IN STD_LOGIC_VECTOR(7 downto 0); -- ALU contents
      rom_oe, acc_oe, alu_oe : IN STD_LOGIC;
      data_bus : OUT STD_LOGIC_VECTOR(7 downto 0));
END data_mux;
ARCHITECTURE a OF data_mux IS
   SIGNAL controls : STD_LOGIC_VECTOR(2 downto 0);
BEGIN
   controls <= rom_oe & acc_oe & alu_oe;
   PROCESS(clock, reset)
   BEGIN
      IF(reset = '0')THEN
         data_bus <= (others => 'Z');
      ELSIF(clock'EVENT and clock = '1')THEN
         CASE controls IS
            WHEN "100" =>data_bus <= rom_data;
            WHEN "010" =>data_bus <= acc_data;
            WHEN "001" =>data_bus <= alu_data;
            WHEN others =>data_bus <= (others => 'Z');
         END CASE;
      END IF;
   END PROCESS;
END a;
```

High impedance

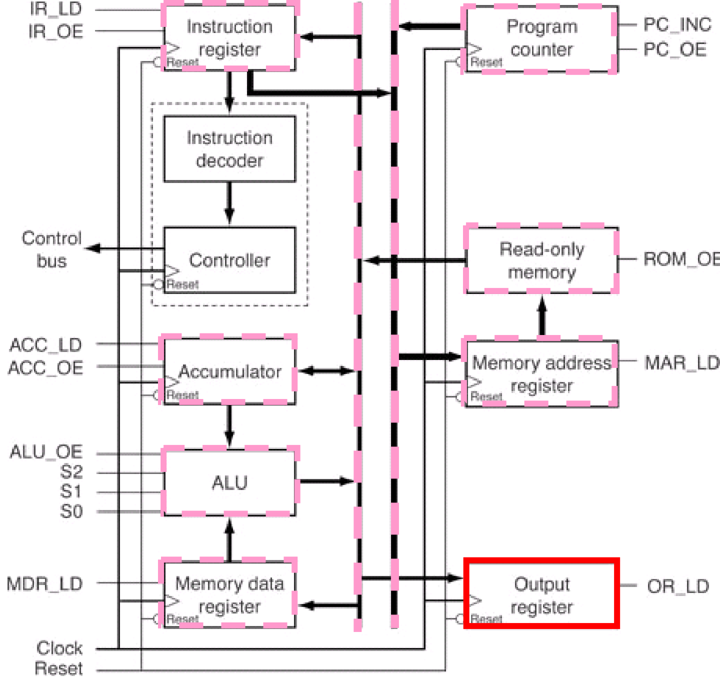
High impedance



-- Multiplexes data from ROM, Accumulator, or ALU onto tristate data bus
-- Each data input is separately enabled; only one enable permitted at a time.

# Output Register

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY output_register IS
PORT(
    data_bus  : IN  STD_LOGIC_VECTOR(7 downto 0);
    clock, reset, or_ld : IN  STD_LOGIC;
    output  : OUT STD_LOGIC_VECTOR(7 downto 0));
END output_register;

ARCHITECTURE output OF output_register IS
BEGIN
    PROCESS(clock, reset)
    BEGIN
        IF(reset = '0')THEN
            -- Clear all register bits
            output <= (others => '0');
        ELSIF (clock'EVENT and clock = '1') THEN
            -- Load register on positive clock edge, if input is enabled
            IF(or_ld = '1')THEN
                output <= data_bus;
            END IF;
        END IF;
    END PROCESS;
END output ;
```

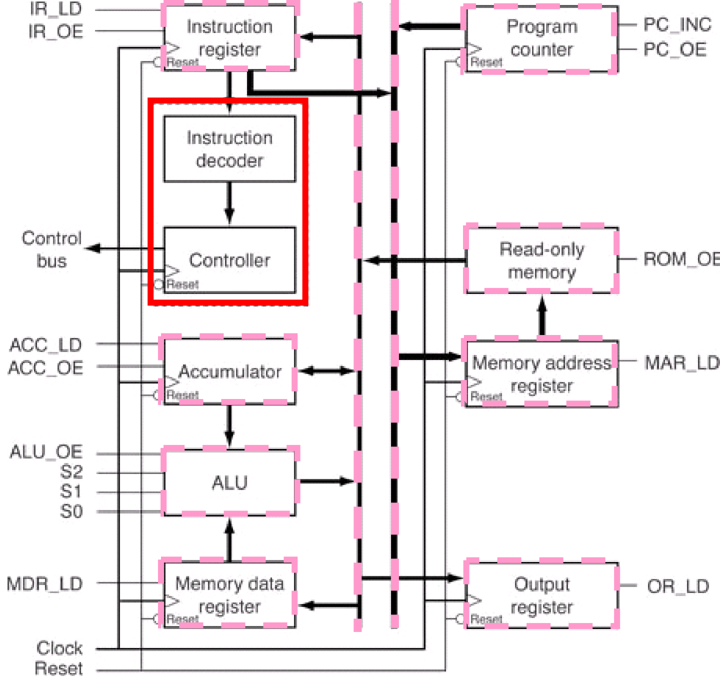


```
-- Output register
-- 8-bit flip-flop with input
-- Input: 8-bit data bus
-- Register has active-LOW
asynchronous reset
```

# Controller

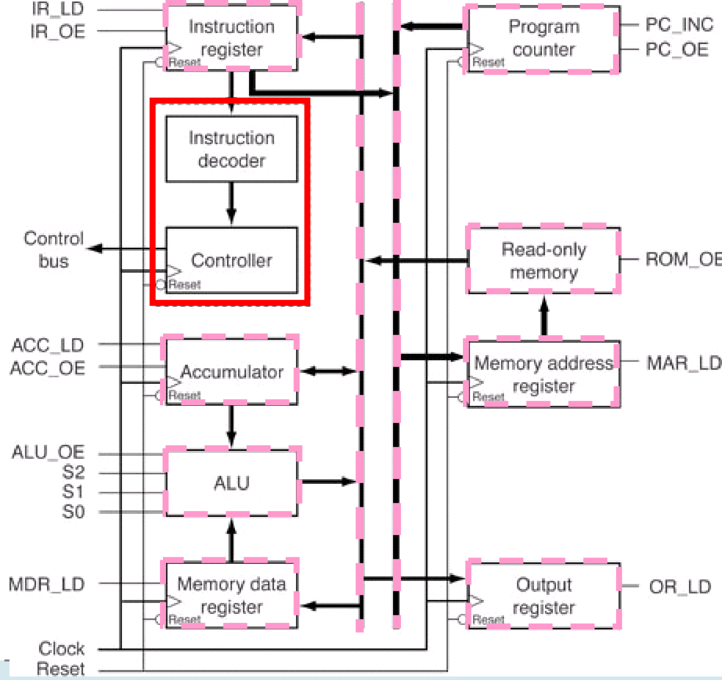
```
ENTITY controller_v1 IS
    PORT(
        clock, reset : IN     BIT;
        instruction : IN      BIT_VECTOR(3 dowoto 0);
        fetch, pc_inc, pc_oe : OUT      BIT;
        ir_ld, ir_oe, mar_ld, rom_oe : OUT            BIT;
        acc_ld, acc_oe, alu_oe, mdr_ld : OUT       BIT;
        or_ld : OUT           BIT;
        s : OUT               BIT_VECTOR(2 downto 0));
END controller_v1;
ARCHITECTURE ctrl OF controller_v1 IS
    TYPE state_type IS (
        start, fetch1, fetch2, fetch3, fetch4, fetch5,
        load1, load2, load3, load4,
        add1, add2, add3, add4, add5, add6,
        output1, output2, halt);
    SIGNAL state : state_type;
    SIGNAL control_word : BIT_VECTOR(15 downto 0);
BEGIN
    PROCESS(clock, reset)
    BEGIN
        IF(reset = '0')THEN
            state <= start;
        ELSIF(clock'EVENT and clock = '1')THEN
```

Initial state



```
-- Controller for 8-bit RISC CPU
-- Generates signals to transfer data between CPU
modules
-- and controls arithmetic/logic functions.
-- Version 1 has four instructions, indicated in
hexadecimal:
--1 ADD --8 LOAD --9 OUTPUT --F HALT
--Opcodes 0 to 7 are reserved for ALU functions,
as follows:
--0 INC (Increments Accumulator (ACC))
--1 ADD (Adds contents of ACC and MDR)
--2 SUB (Subtracts contents of MDR from ACC)
--3 DEC (Decrements ACC)
--4 NOT (Complements ACC)
--5 AND (ACC AND MDR)
--6 OR (ACC OR MDR)
--7 XOR (ACC XOR MDR)
```

# Controller (Cont.)



```
-- Create state machine for instruction sequences.
    CASE state IS
      WHEN start =>state <= fetch1; -- Fetch cycle
      WHEN fetch1 =>state <= fetch2;
      WHEN fetch2 =>state <= fetch3;
      WHEN fetch3 =>state <= fetch4;
      WHEN fetch4 =>state <= fetch5;
      WHEN fetch5 =>
        CASE instruction IS-- Decode instruction
          WHEN x"1" =>state <= add1;
          WHEN x"8" =>state <= load1;
          WHEN x"9" =>state <= output1;
          WHEN x"F" =>state <= halt;
          WHEN others =>state <= halt;
        END CASE;
```

```
          WHEN add1 =>state <= add2; -- ADD
          WHEN add2 =>state <= add3;
          WHEN add3 =>state <= add4;
          WHEN add4 =>state <= add5;
          WHEN add5 =>state <= add6;
          WHEN add6 =>state <= fetch1;
          WHEN load1 =>state <= load2; -- LOAD
          WHEN load2 =>state <= load3;
          WHEN load3 =>state <= load4;
          WHEN load4 =>state <= fetch1;
          WHEN output1 =>state <= output2; -- OUTPUT
          WHEN output2 =>state <= fetch1;
          WHEN halt =>state <= halt; -- HALT
          WHEN others =>state <= halt;
        END CASE;
      END IF;
    END PROCESS;
```

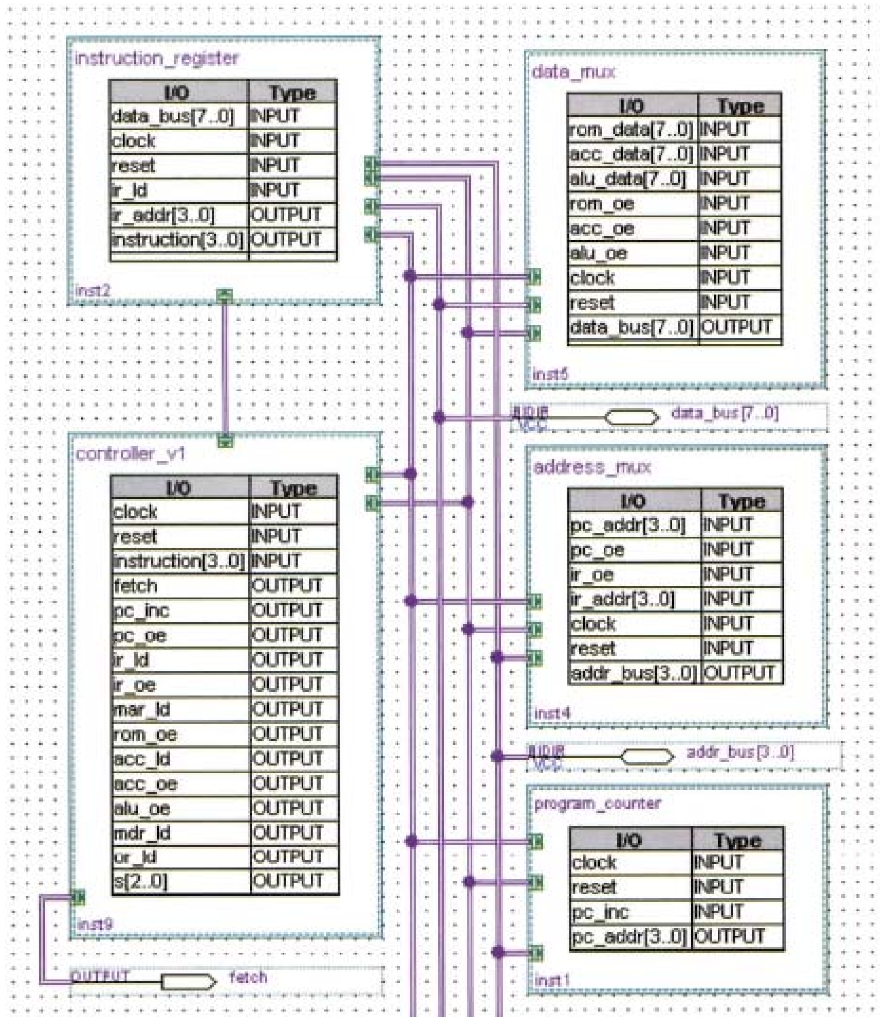
## Controller (Cont.)

```
-- Assign control bus outputs.
  fetch     <=control_word(14);
  pc_inc    <=control_word(13);
  pc_oe     <=control_word(12);

  ir_ld     <=control_word(11);
  ir_oe     <=control_word(10);
  mar_ld    <=control_word(9);
  rom_oe    <=control_word(8);

  acc_ld    <=control_word(7);
  acc_oe    <=control_word(6);
  alu_oe    <=control_word(5);
  mdr_ld    <=control_word(4);

  or_ld     <=control_word(3);
  s         <=control_word(2 downto 0);
```

```
  -- Output decoder
  -- Assign output control lines for each control state
  -- "Fetch" output goes LOW to turn on LED during fetch cycle
  WITH state SELECT
     control_word <=
        x"4000" WHEN start, -- Fetch LED OFF

        x"1000" WHEN fetch1, -- pc_oe
        x"2200" WHEN fetch2, -- pc_inc, mar_ld
        x"0100" WHEN fetch3, -- rom_oe
        x"0800" WHEN fetch4, -- ir_ld
        x"0000" WHEN fetch5, -- wait state

        x"4400" WHEN load1, -- ir_oe
        x"4200" WHEN load2, -- mar_ld
        x"4100" WHEN load3, -- rom_oe
        x"4080" WHEN load4, -- acc_ld

        x"4401" WHEN add1, -- ir_oe,  s=001
        x"4201" WHEN add2, -- mar_ld, s=001
        x"4101" WHEN add3, -- rom_oe, s=001
        x"4011" WHEN add4, -- mdr_ld, s=001
        x"4021" WHEN add5, -- alu_oe, s=001
        x"4081" WHEN add6, -- acc_ld, s=001

        x"4040" WHEN output1, -- acc_oe
        x"4008" WHEN output2, -- or_ld

        x"4000" WHEN others; -- Fetch LED stays OFF
END ctrl;
```

## Integration with Block Diagram File (.BDF)

Conduit: connect outputs and inputs with the same name together.



instruction_register

| I/O | Type |
|---|---|
| data_bus[7..0] | INPUT |
| clock | INPUT |
| reset | INPUT |
| ir_ld | INPUT |
| ir_addr[3..0] | OUTPUT |
| instruction[3..0] | OUTPUT |

inst2

controller_v1

| I/O | Type |
|---|---|
| clock | INPUT |
| reset | INPUT |
| instruction[3..0] | INPUT |
| fetch | OUTPUT |
| pc_inc | OUTPUT |
| pc_oe | OUTPUT |
| ir_ld | OUTPUT |
| ir_oe | OUTPUT |
| mar_ld | OUTPUT |
| rom_oe | OUTPUT |
| acc_ld | OUTPUT |
| acc_oe | OUTPUT |
| alu_oe | OUTPUT |
| mdr_ld | OUTPUT |
| or_ld | OUTPUT |
| s[2..0] | OUTPUT |

inst9

data_mux

| I/O | Type |
|---|---|
| rom_data[7..0] | INPUT |
| acc_data[7..0] | INPUT |
| alu_data[7..0] | INPUT |
| rom_oe | INPUT |
| acc_oe | INPUT |
| alu_oe | INPUT |
| clock | INPUT |
| reset | INPUT |
| data_bus[7..0] | OUTPUT |

inst5

address_mux

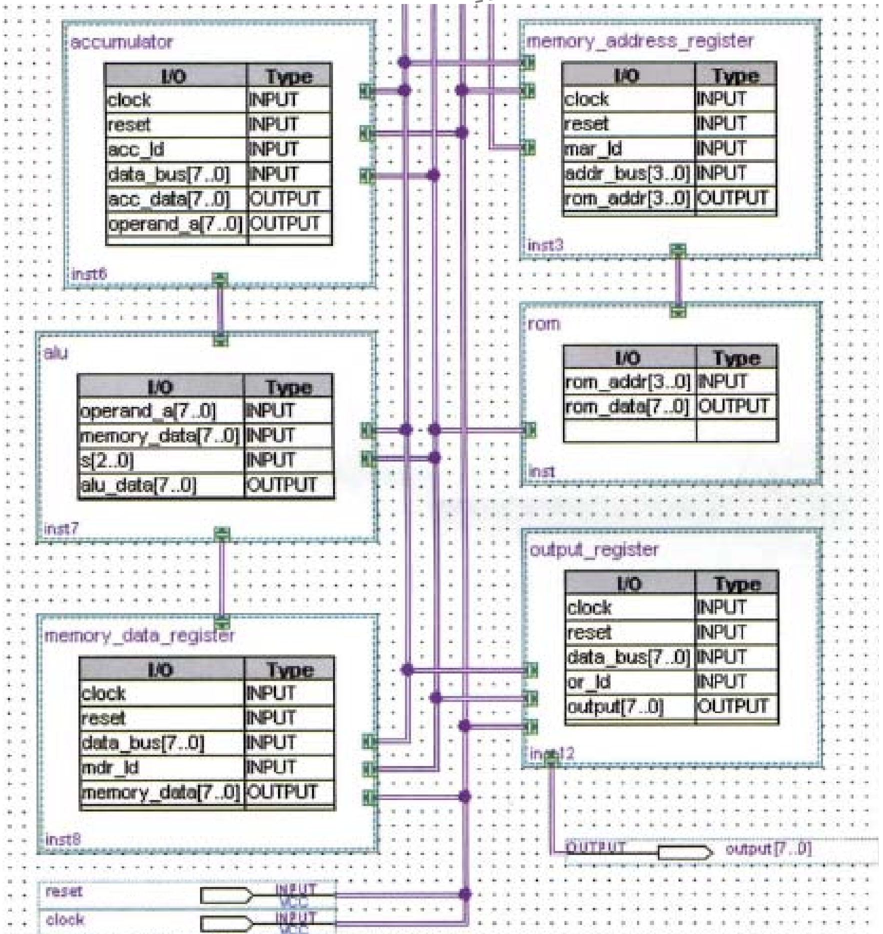
| I/O | Type |
|---|---|
| pc_addr[3..0] | INPUT |
| pc_oe | INPUT |
| ir_oe | INPUT |
| ir_addr[3..0] | INPUT |
| clock | INPUT |
| reset | INPUT |
| addr_bus[3..0] | OUTPUT |

inst4

program_counter

| I/O | Type |
|---|---|
| clock | INPUT |
| reset | INPUT |
| pc_inc | INPUT |
| pc_addr[3..0] | OUTPUT |

inst1



accumulator

| I/O | Type |
|---|---|
| clock | INPUT |
| reset | INPUT |
| acc_ld | INPUT |
| data_bus[7..0] | INPUT |
| acc_data[7..0] | OUTPUT |
| operand_a[7..0] | OUTPUT |

inst6

memory_address_register

| I/O | Type |
|---|---|
| clock | INPUT |
| reset | INPUT |
| mar_ld | INPUT |
| addr_bus[3..0] | INPUT |
| rom_addr[3..0] | OUTPUT |

inst3

rom

| I/O | Type |
|---|---|
| rom_addr[3..0] | INPUT |
| rom_data[7..0] | OUTPUT |
| | |

inst

alu

| I/O | Type |
|---|---|
| operand_a[7..0] | INPUT |
| memory_data[7..0] | INPUT |
| s[2..0] | INPUT |
| alu_data[7..0] | OUTPUT |

inst7

output_register

| I/O | Type |
|---|---|
| clock | INPUT |
| reset | INPUT |
| data_bus[7..0] | INPUT |
| or_ld | INPUT |
| output[7..0] | OUTPUT |

inst12

memory_data_register

| I/O | Type |
|---|---|
| clock | INPUT |
| reset | INPUT |
| data_bus[7..0] | INPUT |
| mdr_ld | INPUT |
| memory_data[7..0] | OUTPUT |

inst8