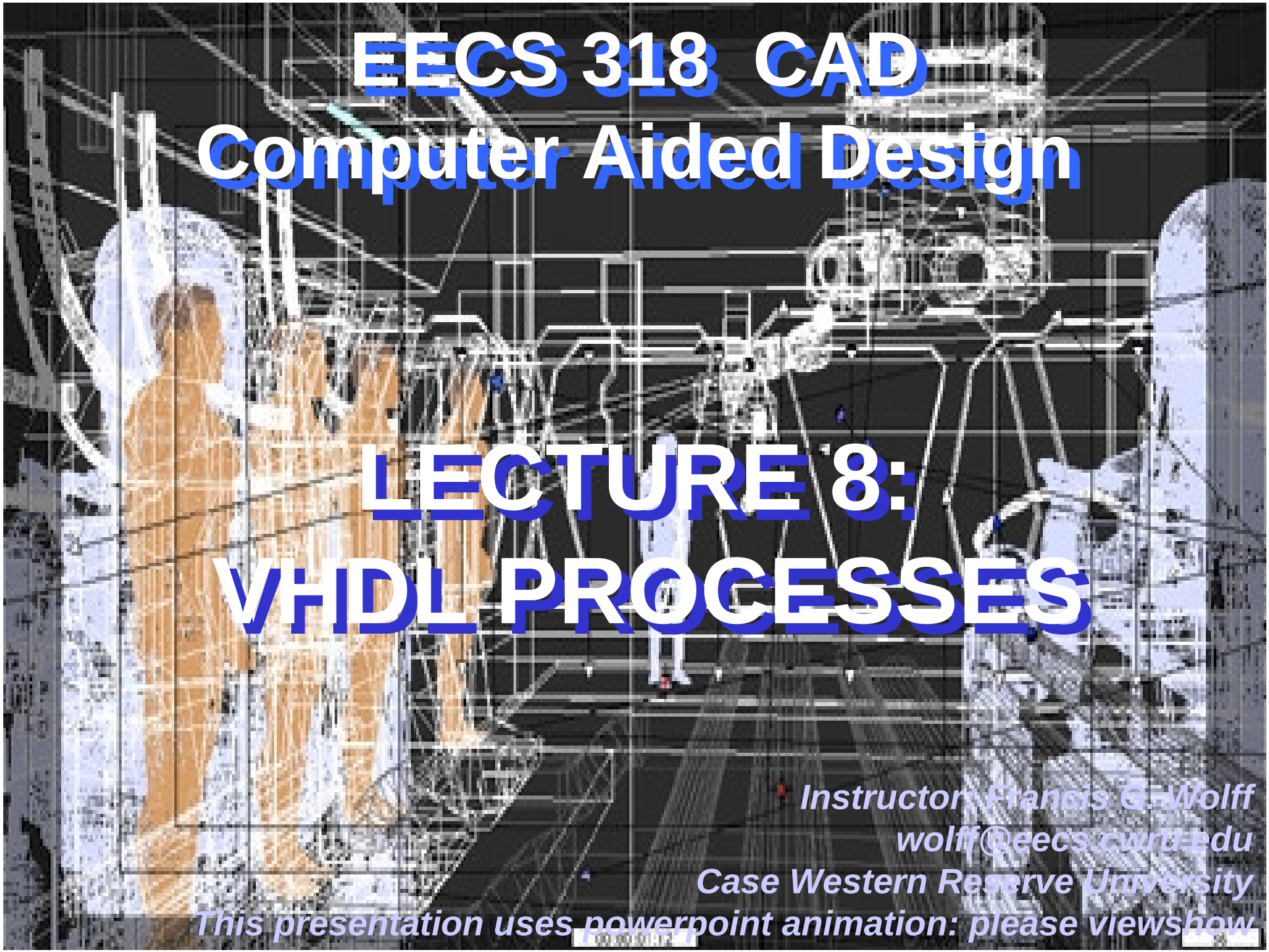


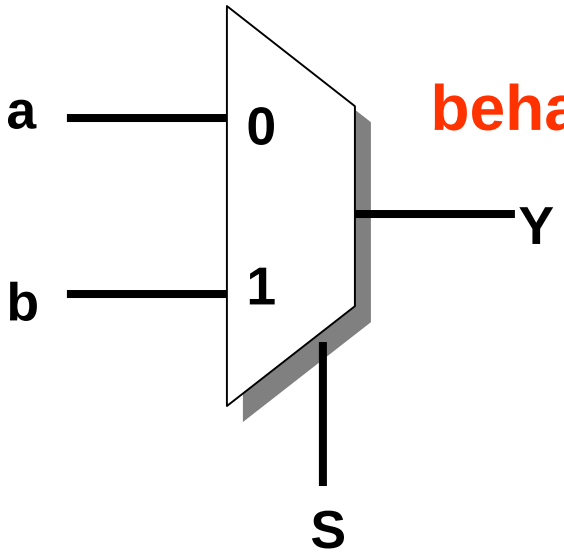
EECS 318 CAD
Computer Aided Design

LECTURE 8:
VHDL PROCESSES

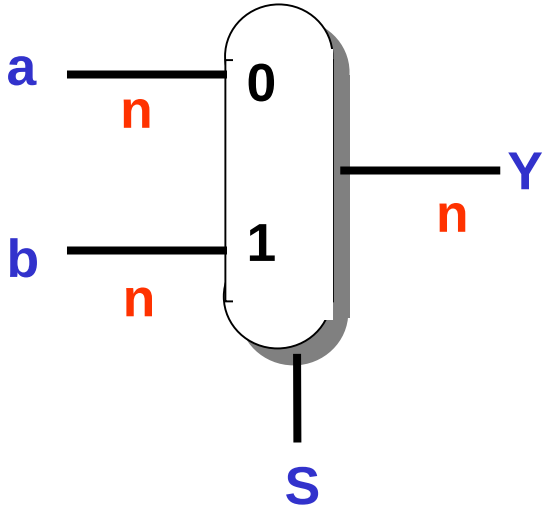
Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

2-to-1 Multiplexor: and Datapath multiplexor



behavioral



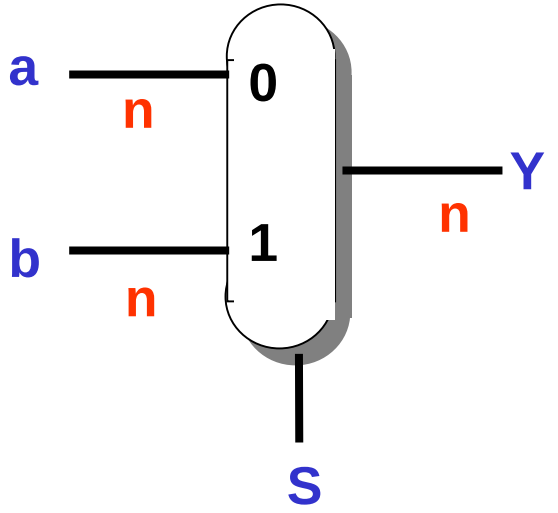
Datapath is n bits wide

```
WITH s SELECT
    Y <= a WHEN '0',
         b WHEN OTHERS;
```

```
WITH s SELECT
    Y <= a WHEN '0',
         b WHEN OTHERS;
```

Where is the difference?

Generic 2-to-1 Datapath Multiplexor Entity



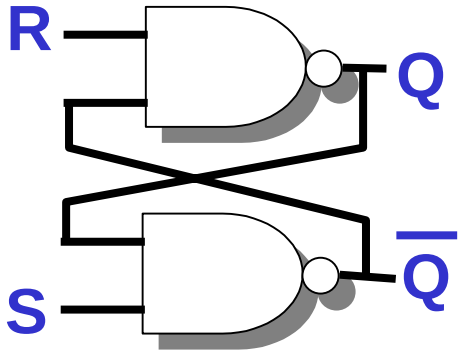

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;

ENTITY Generic_Mux IS
        GENERIC (n:    INTEGER);
        PORT (Y:     OUT   std_logic_vector(n-1 downto 0);
                a:     IN    std_logic_vector(n-1 downto 0);
                b:     IN    std_logic_vector(n-1 downto 0);
                S:     IN    std_logic_vector(0 downto 0)
        );
END ENTITY;
```

Generic 2-to-1 Datapath Multiplexor Architecture

```
ARCHITECTURE Generic_Mux_arch OF Generic_Mux IS
BEGIN
    WITH S SELECT
    Y <= a WHEN "1",
         b WHEN OTHERS;
END ARCHITECTURE;


CONFIGURATION Generic_Mux_cfg OF Generic_Mux IS
  FOR Generic_Mux_arch
  END FOR;
END CONFIGURATION;
```

Configurations are require for simulation

Structural SR Flip-Flop (Latch)

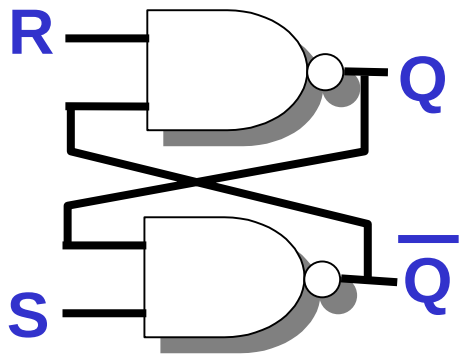


NAND

R	S	Q_{n+1}
0	0	U
0	1	1
1	0	0
1	1	Q_n

```
ENTITY Latch IS
        PORT(R, S: IN std_logic; Q, NQ: OUT std_logic);
END ENTITY;

ARCHITECTURE latch_arch OF Latch IS
BEGIN
        Q       <= R NAND NQ;
        NQ      <= S NAND Q;
END ARCHITECTURE;
```

Inferring Behavioral Latches: Asynchronous



NAND

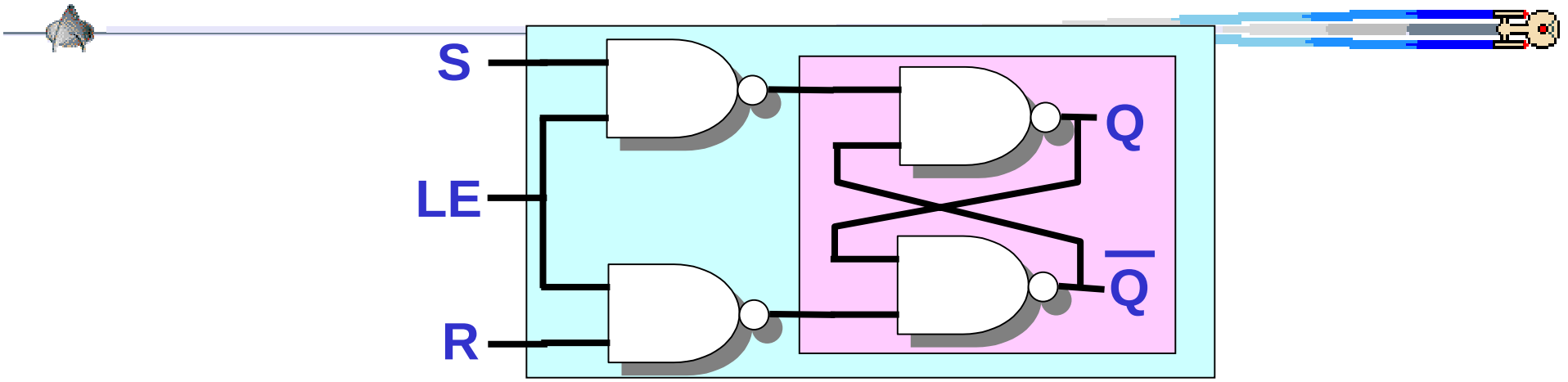
R	S	Q_{n+1}
0	0	U
0	1	1
1	0	0
1	1	Q_n

Sensitivity list of signals: Every time a change of state or event occurs on these signals this process will be called

Sequential Statements

```
ARCHITECTURE Latch2_arch OF Latch IS
BEGIN
        PROCESS (R, S) BEGIN
                IF R= '0' THEN
                        Q <= '1'; NQ<='0';
                ELSIF S='0' THEN
                        Q <= '0'; NQ<='1';
                END IF;
        END PROCESS;
END ARCHITECTURE;
```

Gated-Clock SR Flip-Flop (Latch Enable)




```
ARCHITECTURE Latch_arch OF GC_Latch IS BEGIN
    PROCESS (R, S, LE) BEGIN
        IF LE='1' THEN
            IF R= '0' THEN
                Q <= '1'; NQ<='0';
            ELSIF S='0' THEN
                Q <= '0'; NQ<='1';
            END IF;
        END IF;
    END PROCESS;
END ARCHITECTURE;
```

Inferring D-Flip Flops: Synchronous

```
ARCHITECTURE Dff_arch OF Dff IS
BEGIN

        PROCESS (Clock) BEGIN
                IF Clock'EVENT AND Clock='1' THEN
                        Q <= D;
                END IF;
        END PROCESS;
END ARCHITECTURE;
```

Notice the Process does not contain D:
PROCESS(Clock, D)

Sensitivity lists contain signals used in conditionals (i.e. IF)

Clock'EVENT is what distinguishes a D-FlipFlip from a Latch

Inferring D-Flip Flops: rising_edge

```
ARCHITECTURE Dff_arch OF Dff IS BEGIN
        PROCESS (Clock) BEGIN
                IF  Clock'EVENT AND Clock='1' THEN
                        Q <= D;
                END IF;
        END PROCESS;
END ARCHITECTURE;

ARCHITECTURE dff_arch OF dff IS BEGIN
        PROCESS (Clock) BEGIN
                IF  rising_edge(Clock) THEN
                        Q <= D;
                END IF;
        END PROCESS;
END ARCHITECTURE;
```

Alternate and more readable way is to use the rising_edge function

Inferring D-Flip Flops: Asynchronous Reset

```
ARCHITECTURE dff_reset_arch OF dff_reset IS BEGIN

        PROCESS (Clock, Reset) BEGIN

                IF Reset= '1' THEN -- Asynchronous Reset
                        Q <= '0'

                ELSIF  rising_edge(Clock) THEN --Synchronous
                        Q <= D;
                END IF;
        END PROCESS;


END ARCHITECTURE;
```

Inferring D-Flip Flops: Synchronous Reset

```
PROCESS (Clock, Reset) BEGIN
        IF rising_edge(Clock) THEN
                IF Reset='1' THEN
                        Q <= '0'
                ELSE
                        Q <= D;
                END IF;
        END IF;
END PROCESS;
```

Synchronous Reset

Synchronous FF

```
PROCESS (Clock, Reset) BEGIN
        IF Reset='1' THEN
                Q <= '0'
        ELSIF  rising_edge(Clock) THEN
                Q <= D;
        END IF;
END PROCESS;
```

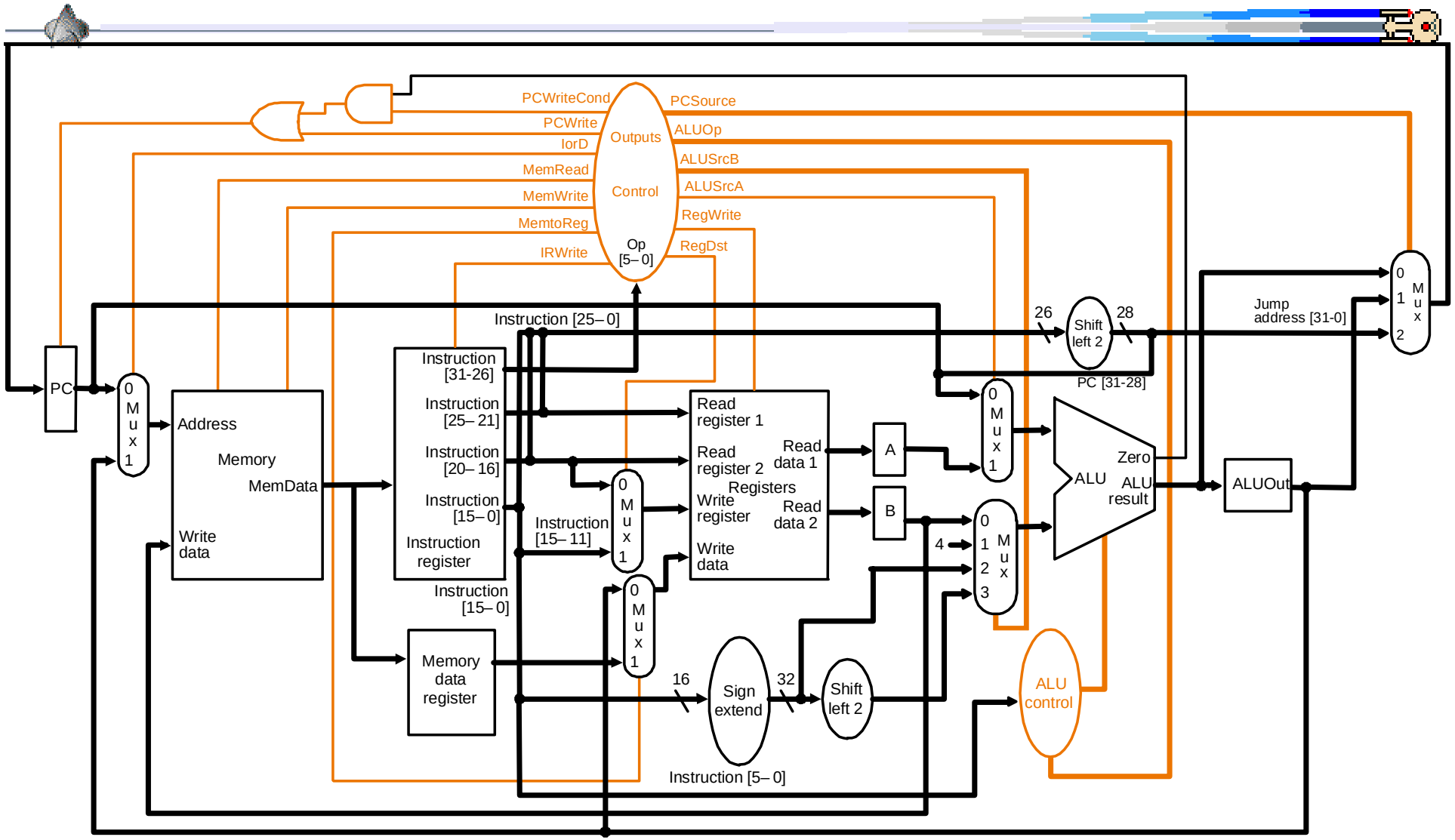
Asynchronous Reset

Synchronous FF

D-Flip Flops: Asynchronous Reset & Preset

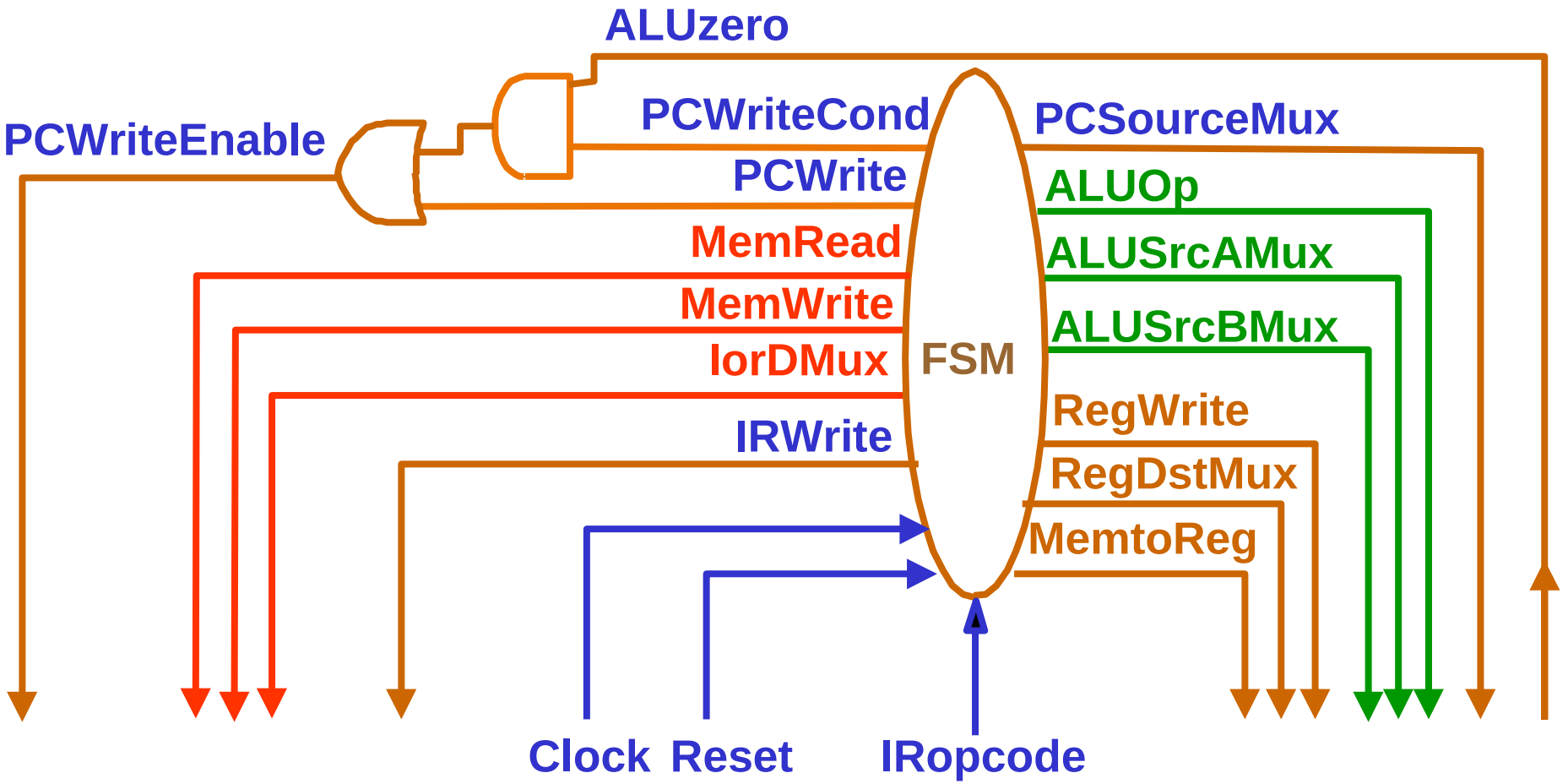
```
PROCESS (Clock, Reset, Preset) BEGIN
        IF Reset='1' THEN --highest priority
                Q <= '0';
        ELSIF Preset='1' THEN
                Q <= '0';
        ELSIF rising_edge(Clock) THEN
                Q <= D;
        END IF;


END PROCESS;
```

RTL Multi-cycle Datapath: with controller



Register Transfer Level (RTL) View

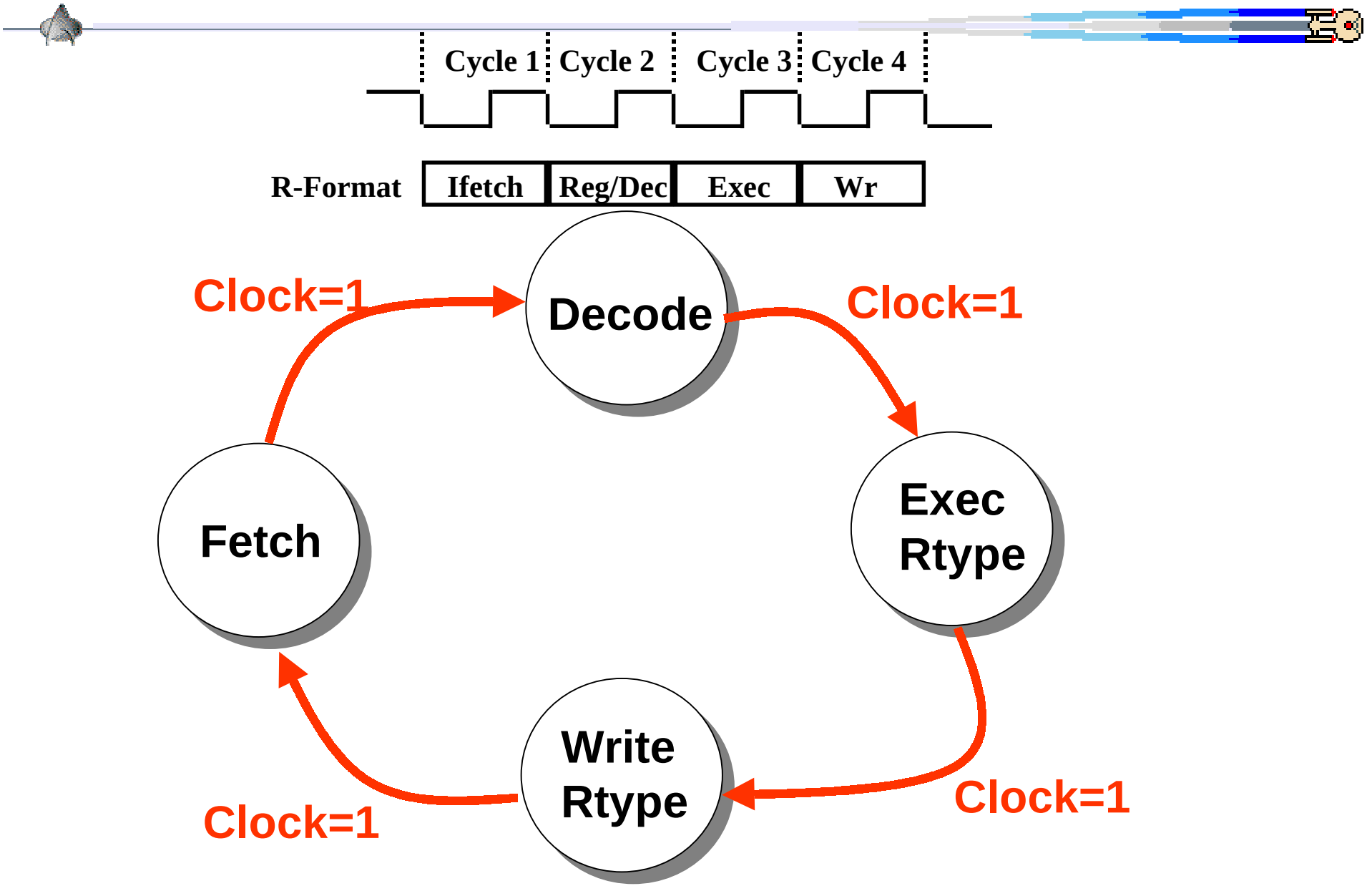
CPU controller: Finite State Machine



CPU Controller: Entity

```
ENTITY cpu_controller is PORT(
        CLK, RST              :IN    std_logic;
        IRopcode              :IN    std_logic_vector(5 downto 0);
        ALUzero               :IN    std_logic;
        PCWriteEnable         :OUT std_logic;
        PCSourceMux           :OUT std_logic_vector(1 downto 0);
        MemRead, MemWrite :OUT std_logic;
        IorDMux               :OUT std_logic;
        IRWrite               :OUT std_logic;
        RegWrite              :OUT std_logic;
        RegDstMux             :OUT std_logic;
        MemtoRegMux           :OUT std_logic;
        ALUOp                 :OUT std_logic_vector(2 downto 0)
        ALUSrcAMux            :OUT std_logic;
        ALUSrcBMux            :OUT std_logic_vector(1 downto 0);
); END ENTITY;
```

CPU controller: R-Format State Machine



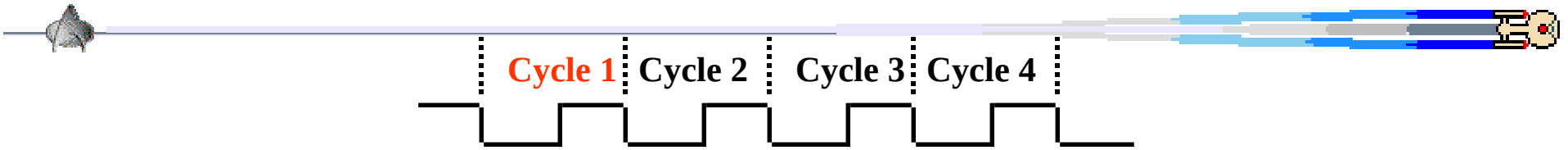
CPU Controller: Current State Process

```
ARCHITECTURE cpu_controller_arch OF cpu_controller IS
  TYPE CPUStates IS (Fetch, Decode, ExecRtype, WriteRtype);
  SIGNAL State, NextState :CPUStates;
BEGIN

  PROCESS (State) BEGIN
    CASE State IS
        WHEN Fetch       => NextState <= Decode;
        WHEN Decode      => NextState <= ExecRtype;
        WHEN ExecRtype   => NextState <= WriteRtype;
        WHEN WriteRtype  => NextState <= Fetch;
        WHEN OTHERS      => NextState <= Fetch;
    END CASE;
  END PROCESS;
  • • •
```

CPU controller: NextState Clock Process

```
PROCESS (CLK, RST) BEGIN
  IF RST='1' THEN -- Asynchronous Reset
    State <= Fetch;

  ELSIF rising_edge(CLK) THEN
    State <= NextState;
  END IF;
 END PROCESS;

END ARCHITECTURE;
```

T_1 Fetch: State machine



R-Format | Ifetch | Reg/Dec | Exec | Wr

Start

MemRead=1, MemWrite=0
IorD=1 (MemAddr←PC)
IRWrite=1 (IR←Mem[PC])
ALUOP=ADD (PC←4+PC)
ALUSrcA=0 (=PC)
ALUSrcB=1 (=4)
PCWrite=1, PCSource=1 (=ALU)
RegWrite=0,
RegDst=X, MemtoReg=X

Instruction Fetch

Decode

Exec

Write Reg

T_1 Fetch: VHDL with Moore Output States

```
PROCESS (State) BEGIN
  CASE State IS
  WHEN Fetch =>
     NextState      <= Decode;
     MemRead        <= '1';
     MemWrite       <= '0';
     IorD           <= '1'
     IRWrite        <= '1';
     ALUOp          <= "010";     --add
     ALUSrcAMux     <= '1';       --PC
     ALUSrcBMux     <= "01";      --4
     PCWriteEnable<= '1';
     PCSourceMux <= "00"; --ALU (not ALUOut)
     RegWrite       <= '0';
     RegDstMux      <= 'D'; MemtoReg <= 'D';
```

MemRead=1, MemWrite=0
IorD=1 (MemAddr←PC)
IRWrite=1 (IR←Mem[PC])
ALUOP=ADD (PC←4+PC)
ALUSrcA=0 (=PC)
ALUSrcB=1 (=4)
PCWrite=1, PCSource=1 (=ALU)
RegWrite=0,
RegDst=X, MemtoReg=X

Instruction Fetch

'D' for Don't Care

VHDL inferred Latches: WARNING

In VHDL case statement

The same signal must be defined for each case

Otherwise that signal will be inferred as a latch

and not as combinatorial logic!

For example,

Even though RegDstMux <= 'D' is not used

and was removed from the Decode state

This will result in a RegDstMux
being inferred as latch not as logic

even though in the WriteRtype state it is set

Assignment #3: CPU Architecture design (1/3)

Cyber Dynamics Corporation (18144 El Camino Real, S'Vale California) needs the following embedded model 101 microprocessor designed by *Thursday October 5, 2000* with the following specifications

- **16 bit instruction memory using ROM**
- **8 bit data memory using RAM**
- **There are eight 8-bit registers**
- **The instruction set is as follows**

- **All Arithmetic and logical instructions set a Zero one-bit flag (Z) based on ALU result**
- **add, adc, sub, sbc set the Carry/Borrow one-bit Flag (C) based on ALU result**

Assignment #3: CPU Architecture design (2/3)

Arithmetic and logical instructions

```
add $rt,$rs          #$rt = $rt +$rs; C=ALUcarry; Z=$rt
adc $rt, $rs         #$rt = $rt +$rs+C; C=ALUcarry; Z;
sub $rt, $rs         #$rt = $rt - $rs; C=ALUborrow; Z;
sub $rt, $rs         #$rt = $rt - $rs - borrow; C; Z;
and $rt, $rs         #$rt = $rt & $rs;     C=0; Z=$rt;
or  $rt, $rs         #$rt = $rt | $rs;     C=0; Z=$rt;
xor $rt, $rs         #$rt = $rt ^ $rs;     C=1; Z=$rt;
```

Other Instructions continued):

```
lbi   $r, immed    #$r = immediate
lbr   $rt,$rs      #$rt = Mem[$rs]
lb    $r, address  #$r = Mem[address]
stb   $r, address  #Mem[address]=$r
bz    address      #if zero then pc=pc+2+addr
bc    address      #if carry then pc=pc+2+addr
j     address      #pc = address
jr    $r           #pc = $r
```

Assignment #3: CPU Architecture design (2/3)

(1a) Design an RTL diagram of the model 101 processor and

(1b) Opcode formats and

(1c) Opcode bits of a Harvard Architecture CPU (determine your own field sizes).

(1d) What is the size of the Program Counter?

(2) Write the assembly code for a 32 bit add Z=X+Y located in memory address for @X = 0x80 @Y = 0x84 and @Z=0x88

(3) Draw the state diagram for your FSM controller

(4) Write the VHDL code for the FSM controller

Note: this will be part of your final project report