Design using VHDL

Review

&

Good Design Practices

For ECE412 Spring 10
Christine Chen
Slides by Alex Papakonstantinou

HDL Characteristics

- VHDL and Verilog are both hardware description languages
- Look similar to conventional programming languages
- However, they have a significant difference with regards to programming languages: they are inherently Parallel
- Allow different levels of abstraction:
  - Behavioral

```
Result := a*b + c mod i
```

  - Structural

```
mul0: MUL
   port map (A_op, B_op, R1)
mod0: MOD
   port map (C_op, I_op, R2)
add1: ADD
   port map (R1, R2, Result)
```

Concurrent Statements

Simple Signal Assignment:

```
sum <= (a XOR b) XOR cin;
carry <= a AND b         ;
```

Conditional Signal Assignment:

```
z <= a WHEN s1='0' AND s0='0' ELSE
     b WHEN s1='0' AND s0='1' ELSE
     c WHEN s1='1' AND s0='0' ELSE
     d                        ;
```

- Priority is inferred based on the order of selection values

Selected Signal Assignment:

```
WITH sel SELECT
     z <= a WHEN "00",
          b WHEN "01",
          c WHEN "10",
          d WHEN "11";
```

- No two choices can overlap
- All possible cases must be covered if "when others" is not present
- All options have equal priority

Combinational Procedures as Concurrent Statements

- Combinational process:

```
PROCESS (a, b, c)
BEGIN
        IF (c = '1') THEN
                out <= a;
        ELSE
                out <= b;
        END IF;
END PROCESS;
```

- Make sure there are no missing clauses (otherwise a latch may be inferred)
- Latches should be avoided in synchronous designs

- **Sensitivity list is usually ignored during synthesis (it must contain all read signals)**
- **Sensitivity list can be replaced by "WAIT ON a,b,c" at the end of process**
- **Use either WAIT ON or Sens. List, NOT both**

Clocked Procedures as Concurrent Statements

- Sequential process:

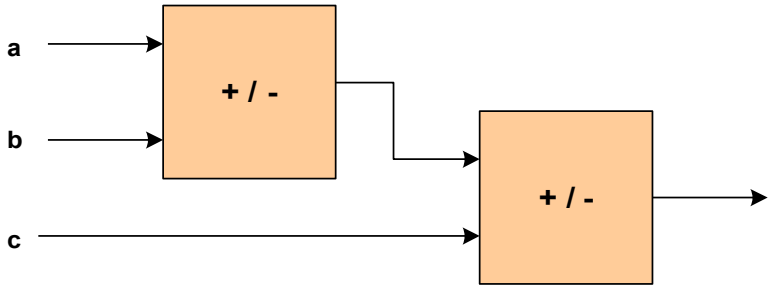
```
PROCESS (clk)
BEGIN
IF (clk'EVENT AND clk = '1') THEN
        IF (c = '1') THEN
                out <= a;
        END IF;
END IF;
END PROCESS;
```

- Missing clauses do not infer any latches in this case

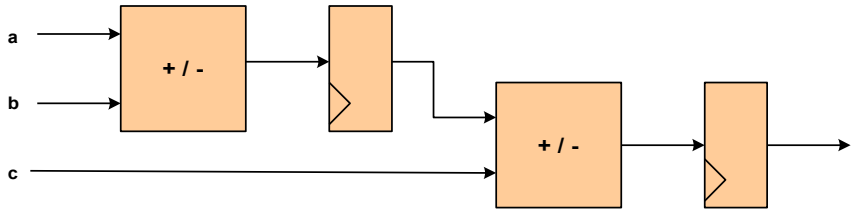
- **Sensitivity list must only contain clock signal**

Combinational vs Sequential Process

```
PROCESS (a, b, c)
  BEGIN
   IF (c = '1') THEN
    tmp <= a + b;
    out <= tmp + c;
   ELSE
     tmp <= a - b;
     out <= tmp – c;
   END IF;
END PROCESS;
```




```
PROCESS (clk)
  BEGIN
   IF rising_edge(clk) THEN
     IF (c='1') THEN
       tmp <= a + b;
       out <= tmp + c;
     ELSE
       tmp <= a - b;
       out <= tmp – c;
     END IF;
   END IF;
END PROCESS;
```



Result is ready 2 cycles later!

Synchronous / Asynchronous Reset in Clocked Procedures

- **Synchronous Reset**:

```
PROCESS (clk)
BEGIN
IF (clk'EVENT AND clk = '1') THEN
        IF (rst = '1') THEN
            out <= '0';
          ELSE
            out <= a;
          END IF;
END IF;
END PROCESS;
```

- **Asynchronous Reset**

```
PROCESS (clk, rst)
    BEGIN
    IF (rst = '1') THEN
                out < '0';
ELSIF (clk'EVENT AND clk = '1') THEN
                out <= a;
    END IF;
END PROCESS;
```

Types of Procedural Assignments

Both Variables and Signals can be assigned within procedures

"<=" signal assignments

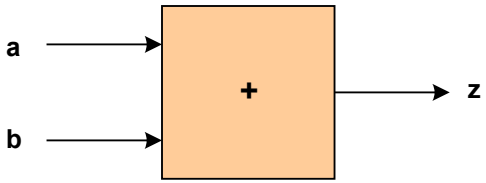
- Evaluated in parallel regardless of their order
- Should be preferred in sequential procedures to implement flip-flops

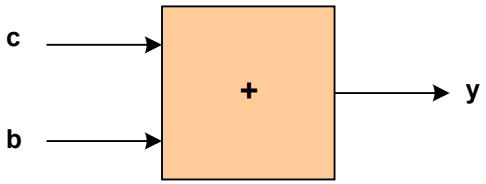
":=" variable assignments

- Evaluated in the order statements are written
- Good for algorithm implementation with combinational logic
- Variables only accessible within procedure
- Registers are generated for variables that might be read before being updated (since variables keep their value between process calls)!

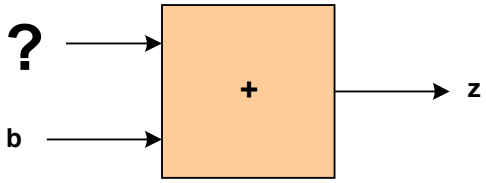
Signal vs Variable Assignment

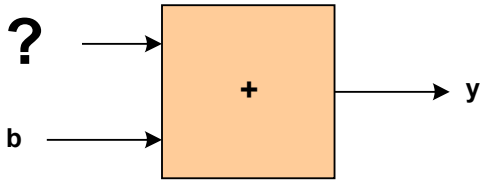
```
PROCESS (a, b, c)
  BEGIN
   VARIABLE M, N : integer;
    M := a;
    N := b;
    z <= M + N;
    M := c;
    y <= M + N;
 END PROCESS;
```





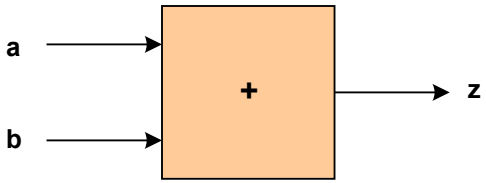

```
PROCESS (a, b, c, M, N)
  BEGIN
    M <= a;
    N <= b;
    z <= M + N;
    M <= c;
    y <= M + N;
 END PROCESS;
```

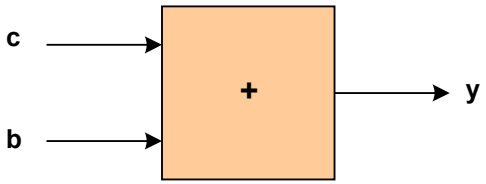




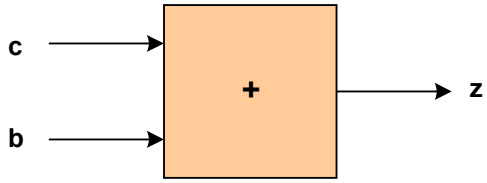
Signal vs Variable Assignment

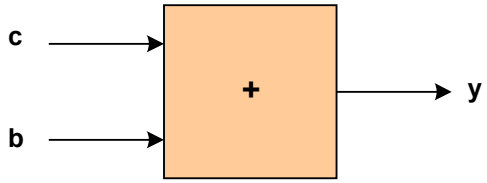
```
PROCESS (a, b, c)
  BEGIN
   VARIABLE M, N : integer;
    M := a;
    N := b;
    z <= M + N;
    M := c;
    y <= M + N;
 END PROCESS;
```






```
PROCESS (a, b, c, M, N)
  BEGIN
    M <= a;
    N <= b;
    z <= M + N;
    M <= c;
    y <= M + N;
 END PROCESS;
```





“if” statements

```
z <= a;
IF (x = “1111”) THEN
  Z <= B;
ELSIF (x > “1000”) THEN
  z <= c;
END IF;
```

```
IF (x = “1111”) THEN
  z <= b;
ELSIF (x > “1000”) THEN
  z <= c;
ELSE
  z <= a;
END IF;
```

- **Used only within procedures**
- **Conditions might overlap**
- **Processed sequentially and thus implies priority (statements within first true condition will be executed)**
- **Instead of an “else” clause a default statement may be used before “if – elsif” statement (as shown in above example)**
- **All cases should be covered in order to avoid latch inference**

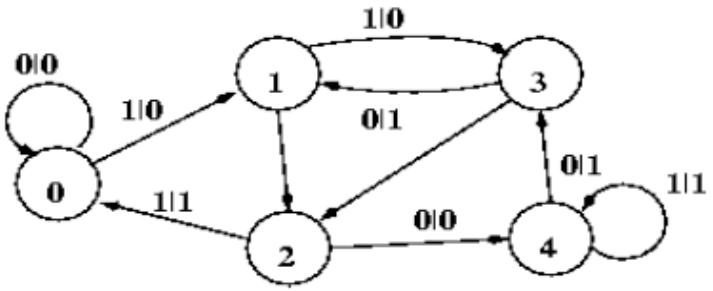
“case” statements

```
CASE x IS
  WHEN “0000”                => z <= a;
  WHEN “0111” | “1001”       => z <= b;
  WHEN OTHERS                => z <= 0;
END CASE;
```

- **Used only within procedures**
- **case options must not overlap**
- **All choice options have to be covered**
- **All branches equal in priority**
- **“when others” covers all remaining choice options**

Example FSM

- This is a simple FSM to illustrate good design practice
- It is a Mealy machine




```
entity divby5 is port (
        x, clk   :  in std_logic;
        y        :  out std_logic;
end divby5
```

FSM (in VHDL)

```
architecture state_machine of divby5 is
        type StateType is (state0, state1, state2, state3, state4);
        signal p_s, n_s : StateType;
  begin
        fsm: process(p_s,x)
        begin
                case p_s is
                  when state0 => y <= '0';
                                if x = '1' then
                                  n_s <= state1;
                                else
                                  n_s <= state0;
                                end if;
                  when state1 => y <= '0';
                                if x = '1' then
                                  n_s <= state3;
                                else
                                  n_s <= state2;
                                end if;
                  when state2 =>
                                if x = '1' then
                                  n_s <= state0;
                                  y <= '1';
                                else
                                  n_s <= state4;
                                  y <= '0';
                                end if;
                  when state3 => y <= '1';
                                if x='1' then
                                  n_s <= state2;
                                else
                                  n_s <= state1;
                                end if;
                  when state4 => y <= '1';
                                if x = '1' then
                                  n_s <= state4;
                                else
                                  n_s <= state3;
                                end if;
                  -- avoid trap states
                  when others => n_s <= state0;
                end case
        end process fsm;

        state_clocked : process(clk)
                        begin
                          if rising_edge(clk) then
                                  p_s <= n_s;
                          end if;
                        end process state_clocked;
end architecture state machine;
```

Real Student Design Pitfalls

- A summary of some design styles that were used in past student projects, which caused unexpected behavior or other side effects

No clock edge detection:

```
backend: process (present_state, clock) is
begin
case present_state is
--STATE S1-- Reset State
        when s1 =>
        scroll_internal <= '0';
        backend_reset <= '1';
        if (reset = '0' and rdempty = '0') then
        next_state <= s10;
        else


        next_state <= s1;
        --          next_state <= s1;
        end if;
        .
        .
        .
```

Process is triggered for both falling and rising edges of “clock”

Sensitivity list issues:

```
seq1: PROCESS (present_state) IS
BEGIN
        case present_state is
                when s0 =>
                        wr_req <= '0';
                        grn_led <= '0';
                        red_led <= '1';
                        frontend_reset <= '1';
                        if start = '1' then
                                next_state <= s1;
                        else
                                next_state <= s0;
                        end if;

                when s1 =>
        .
        .
        .
```

Incomplete sensitivity lists can lead to simulation-synthesis mismatches

Redundant use of Reset in combinational process:

```
control_reg: process(reset, clk_50)
begin
if(reset = '0') then
state <= IDLE;
next_addr <= "00000";
fifo_wr_req <= '0';
else
if(rising_edge(clk_50)) then
if(fifo_full = '0') then
  state <= next_state;
  next_addr <= next_addr - '1';
  fifo_wr_req <= '1';
end if;          end if;          end if;
end process;

get_next_state: process(reset, start, state)
begin
case state is
when IDLE =>
if(start = '0') then
next_state <= ENABLED;
else next_state <= IDLE;
end if;
when ENABLED =>
if(reset = '0') then
next_state <= IDLE;
else next_state <= ENABLED;
end if;
end case;
end process;
```

Usually it is a good practice to use reset only in sequential processes. In this example the use of reset in the combinational process is redundant because the state is reset in the sequential process

fifo_wr_req handling

```
control_reg: process(reset, clk_50)
begin
        if(reset = '0') then
                state <= IDLE;
                next_addr <= "00000";
                fifo_wr_req <= '0';
        else
                if(rising_edge(clk_50)) then
                        if(fifo_full = '0') then
                                state <= next_state;
                                next_addr <= next_addr - '1';
                                fifo_wr_req <= '1';
                        end if;
                end if;
        end if;
end process;
```

This is a functionality issue. fifo_wr_req is only set during normal operation (without considering reset) without any provision for clearing it.

Enumeration of all counter states:

```
get_next_state: process (fifo_full, reset, start, state)
        begin
                case state is
                        when IDLE =>
                                if (start = '0') then next_state <= S31;
                                else next_state <= IDLE;
                                end if;
                        when S31 =>
                                if (reset = '0') then next_state <= IDLE;
                                elsif (fifo_full = '0') then next_state <= S30;
                                else next_state <= S31;
                                end if;
                        when S30 =>
                                if (reset = '0') then next_state <= IDLE;
                                elsif (fifo_full = '0') then next_state <= S29;
                                else next_state <= S30;
                                end if;
                        when S29 =>
                                if (reset = '0') then next_state <= IDLE;
                                elsif (fifo_full = '0') then next_state <= S28;
                                else next_state <= S29;
                                end if;
                        when S28 =>
                                if (reset = '0') then next_state <= IDLE;
                                elsif (fifo_full = '0') then next_state <= S27;
                                else next_state <= S28;
                                end if;
        .
        .
        .
```

A unnecessary complex FSM is implemented where every count value is treated as a separate state

Redundant Sensitivity signals

```
process(C, reset,start,input)
begin
    if reset='1' then
                        tmp<=(others =>'0');
                        elsif C='0' and C'event then
                          if start='1' then
                            tmp <=  tmp(27 downto 0)& input(3 downto 0);
                          end if;
end if;
```

In a sequential process like this (C seems to be a clock signal) it is redundant to list any other signals in the sensitivity list apart from the register clock and the asynchronous reset. The value changes of any other signals will not have an effect, unless there is a clock edge present.

Clock detection in case clause:

```
process(clk50,present_state,counter,wr_full)
begin
case present_state is
when idle =>
temp_addr<=(others =>'0');

when active0 =>
if clk50='0' and clk50'event then
if wr_full='0' then
temp_addr<=temp_addr-1;
end if;
end if;

when others =>
null;

end case;

end process;
```

Not very clean way of describing a state machine. It seems as if the FSM uses the clock for its synchronization only during the “active0” state. Synthesis tools will probably produce strange logic for such descriptions

Other Design Pitfalls

- Making things too complex
  - Make simple FSMs
  - Separate sequential and combinational logic
  - No combinational loops
- Be careful when dealing with multiple clocks
  - Multiple clocks can create bugs that only appear some of the time
  - Worse yet, can create bugs that only appear in hardware vs. being testable in simulation
  - Need to handle the hand-shaking well
    - Can use FIFOs
    - Or multiple stages of registers with proper enable signals

Other Design Pitfalls (cont')

- INOUT Ports
  - If you have a port of type INOUT in a module, you need to explicitly drive it to high impedance "z" when the module is not driving the port.
  - Simply failing to specify an output value for the port isn't good enough, you can get flaky signal
  - Example
    - Case (enable) is
      when "1" => wr <= data;
      when "0" => wr <= "z";
- Gated clocks
  - Avoid multiple clocks if possible
  - Never put any logic between the clock and the clock pin of a module