Lecture 8

VHDL, Part III

Sequential logic and FSMs

Outline

Sequential circuits: principles
  - System reset

Finite state machines
  - Output buffering
Section 1

SEQUENTIAL CIRCUITS: PRINCIPLES

Overview on sequential circuit

Combinational vs sequential circuit

- Sequential circuit: output is a function of current input and state (memory)

Basic memory elements

- D latch
- D FF (Flip-Flop)
- RAM

Synchronous vs asynchronous circuit

- Globally synchronous circuit: all memory elements (D FFs) controlled (synchronized) by a common global clock signal
- Globally asynchronous but locally synchronous circuit (GALS).
- Globally asynchronous circuit (no clock, sometimes no FF)
Storage Elements

D latch: level sensitive
D FF: edge sensitive

Inference of basic memory elements

VHDL code should be clear so that the *pre-designed* cells can be inferred

VHDL code
- D Latch
- Positive edge-triggered D FF
- Negative edge-triggered D FF
- D FF with asynchronous reset
D Latch

No else branch
D latch will be inferred

```
library ieee;
use ieee.std_logic_1164.all;
entity dlatch is
  port(
    c : in std_logic;
    d : in std_logic;
    q : out std_logic
  );
end dlatch;
architecture arch of dlatch is
begin
  process (c, d)
  begin
    if (c='1') then
      q <= d;
  end if;
end process;
end arch;
```

(a) D latch

Positive edge-triggered D FF

No else branch
Note the sensitivity list

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port(
    clk : in std_logic;
    d : in std_logic;
    q : out std_logic
  );
end dff;
architecture arch of dff is
begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      q <= d;
  end if;
end process;
end arch;
```

(b) pos-edge triggered D FF
D FF with async reset

No else branch
Note the sensitivity list

library ieee;
use ieee.std_logic_1164.all;

entity dff is
port(
clk: in std_logic;
reset: in std_logic;
d: in std_logic;
q: out std_logic
);
end dff;

architecture arch of dff is
begin
process (clk,reset)
begin
if (reset='1') then
q <= '0';
elsif (clk'event and clk='1') then
q <= d;
end if;
end process;
end arch;

Register

Multiple D FFs with same clock and reset

library ieee;
use ieee.std_logic_1164.all;

entity reg8 is
port(
clk: in std_logic;
reset: in std_logic;
d: in std_logic_vector(7 downto 0);
q: out std_logic_vector(7 downto 0)
);
end reg8;

architecture arch of reg8 is
begin
process (clk,reset)
begin
if (reset='1') then
q <= (others=>'0');
elsif (clk'event and clk='1') then
q <= d;
end if;
end process;
end arch;
Designing Synchronous Circuits

All signals that are assigned a value should be in branch

\[ \text{ELSIF } \text{clk'EVENT AND clk='1'} \text{ THEN} \]

(or similar) are implemented as registers!

This is because the signals should change state only during a clock edge
- This is how the flip flops work, load the input value on clock edge

The signal type does not matter. Integers become registers (32-bit) as well as std_logic_vectors and own defined types.
- A flip-flop is instantiated for each bit
- Integer ranges should be defined

Designing synchronous circuits

Remember the RTL design
With VHDL synchronous design style, this is actually just what you do
- Define what happens before a register (register assignment), not explicitly the registers

\[ \text{elsif clk'event and clk = '1' then -- rising clock edge} \]
\[ \text{mac_out <= std_logic_vector(unsigned(a_in)*unsigned(b_in)+unsigned(c_in))}; \]

... end if;

... Do this operation and move the result to register

The code shows that mac_out is a register

Inputs may come from registers, but code does not show that
Examples in VHDL

Combinatorial circuit:

\[ A_{\text{in}} \times B_{\text{in}} + C_{\text{in}} \rightarrow \text{Mac}_{\text{out}} \]

Synchronous circuit:

\[ A_{\text{in}} \times B_{\text{in}} + C_{\text{in}} \rightarrow \text{Mac}_{\text{out}} \]

Pipelined Synchronous circuit:

\[ A_{\text{in}} \times B_{\text{in}} + C_{\text{in}} \rightarrow \text{Mac}_{\text{out}} \]

Combinatorial circuit

```
architecture rtl of mac is
begin  -- rtl
  mac_comb: process (a_in, b_in, c_in)
  begin
    mac_out <= std_logic_vector(unsigned(a_in)*unsigned(b_in)+unsigned(c_in));
  end process mac_comb;
end rtl;
```

\[ A_{\text{in}} \times B_{\text{in}} + C_{\text{in}} \rightarrow \text{Mac}_{\text{out}} \]
Synchronous circuit

architecture rtl of mac is
begin -- rtl
  mac_sync : process (clk, rst_n)
  begin
    if rst_n = '0' then        -- asynchronous reset (active low)
      mac_out <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      mac_out <= std_logic_vector(unsigned(a_in)*unsigned(b_in)+unsigned(c_in));
    end if;
  end process mac_sync;
end rtl;

Calculation inside the "clock region"

A register is generated for mac_out since:
- It is a signal (port out)
- It is assigned a value with the clock region

Pipelined synchronous circuit

architecture rtl of mac is
begin -- rtl
  mac_pipe : process (clk, rst_n)
  begin
    if rst_n = '0' then        -- asynchronous reset (active low)
      mac_out <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      c_r <= unsigned(c_in);
      mul_r <= unsigned(a_in)*unsigned(b_in);
      mac_out <= std_logic_vector(mul_r + c_r);
    end if;
  end process mac_pipe;
end rtl;

Calculation inside the "clock region"

Registers implemented for each signal

The mul_r is updated in previous statement. However, signal values do not change until the next clock edge with the clock region (unlike variables). Therefore, mac_out functions correctly as it uses mul_r(t) while the preceding statement produced mul_r(t+1).
Section 2

SYSTEM RESET

System reset

Brings the system into known state
- At start-up
- After crash

Known state = the value of flip-flops in the system
- Concerns sequential logic (seq. processes)

Flip-flop’s value may be set
- Asynchronously via special input pin – Asynchronous reset
- Constant Value - Synchronous reset
- Synchronously via D input - Normal operation
DFF with asynchronous reset

Common way
Reset
– Output Q updated even if no clock signal present
– Must not have any logic gates!

Normal operation
– Value of D appears on Q after each rising edge of clock
– D driven by some combinatorial logic (or input pins)
– Clock driven by input pin, PLL or other special clock generation logic. Not by your own logic!

DFF with asynchronous reset (2)

```vhdl
async_rst : process (clk, rst_n)
-- use exactly this sensitivity list
begin
  if rst_n = '0' then
    -- Assign bootup values for DFFs in this branch.
    -- Use constant values only! Do not read signals or
    -- input ports! Comb. logic in reset signal is
    -- very bad.
    elsif clk'event and clk = '1' then
      -- Assign the values of normal operation in this
      -- branch.
      -- No other conditions to elsif. Otherwise you’ll
get comb. logic into the clock tree which is
      -- disastrous.
      -- Use nested if-statements instead.
      end if;
  end if;
end process async_rst;
```
DFF with synchronous reset

Not so common way but ok

Uses simpler DFFs that have no asynchronous inputs

Now, also the reset needs valid clock signal

Reset
  - Sets/clears D input
  - Minor increase to critical path
  - Routing the reset signal throughout the chip is easier than in async. case

Normal operation as previously

DFF with synchronous reset (2)

```vhdl
sync_rst : process (clk)
begin
  if clk'event and clk = '1' then
    if sync_rst_n = '0' then
      -- Assign bootup values for DFFs in this branch.
      -- No other conditions here.
      -- Assign constant reset values only.
      else
        -- Assign the values of normal operation in this branch.
      end if;
    end if;
  end if;
end process sync_rst;
```

DFF set to '0' at sync. reset

DFF set to '1' at sync. reset
Few further notes about reset

Terms “Async./Sync. Reset” refers how the reset signal is connected to the DFFs

All flip-flops must leave the reset state simultaneously

Section 3

FINITE STATE MACHINES
State Machine

Basic block diagram
- State register (memory elements)
- Next-state logic (combinational circuit)
- Output logic (combinational circuit)

Operation
- At the rising edge of the clock, state_next sampled and stored into the register (and becomes the new value of state_reg)
- The next-state logic determines the new value (new state_next) and the output logic generates the output
- At the rising edge of the clock, the new value of state_next sampled and stored into the register

Overview on FSM

Contain “random” logic in next-state logic
Used mainly as a controller in a large system

Mealy vs Moore output
Moore vs Mealy output

Moore machine:
- output is a function of state

Mealy machine:
- output function of state and input

From theoretical point of view
- Both machines have similar "computation capability"

In general
- Moore is better: the output changes only when the state changes AND lasts at least one clock cycle

VHDL Description of FSM

Follow the basic block diagram code the next-state/output logic according to the state diagram
library ieee;
use ieee.std_logic_1164.all;
entity mem_ctrl is
port (]
: clk, reset in std_logic;
mem, req, burst in std_logic;
ce, we, we_out out std_logic);
ead mem_ctrl ;
architecture multi_reg_unit of mem_ctrl is
  type mem_state_type is
    (idle, read1, read2, read3, write);
  signal state_reg, state_next : mem_state_type;
begin
  -- state register
  process(clk, reset)
  begin
    if (reset='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end process;

Next State Logic

Use of “case” statement

```vhdl
-- next-state logic
process(state_reg, rw, read1) begin
  case state_reg is
    when idle =>
      if rw = '1' then
        if read1 = '1' then
          state_next <= read1;
        else
          state_next <= idle;
        end if;
      else
        state_next <= idle;
      end if;
    when write =>
      state_next <= idle;
    when read1 =>
      if (rw = '1') then
        state_next <= read2;
      else
        state_next <= idle;
      end if;
    when read2 =>
      state_next <= read3;
    when read3 =>
      state_next <= read4;
    when read4 =>
      state_next <= idle;
  end case;
end process;
```

Output Logic

Moore: the output depends only on the state

Mealy: the output depends ALSO on the input signal

```vhdl
-- moore output logic
process(state_reg) begin
  case state_reg is
    when idle =>
      v_reg <= '0';        -- default value
      on <= '0';           -- default value
      when write =>
        v_reg <= '1';
      when read1 =>
        on <= '1';
      when read2 =>
        v_reg <= '1';
      when read3 =>
        on <= '1';
      when read4 =>
        on <= '1';
  end case;
end process;
```

```vhdl
-- mealy output logic
process(state_reg, rw, read1) begin
  v_reg <= '0';        -- default value
  on <= '0';           -- default value
  case state_reg is
    when idle =>
      if (rw = '1') and (read1 = '1') then
        v_reg <= '1';
      else
        v_reg <= '0';
      end if;
    when write =>
      when read1 =>
      when read2 =>
      when read3 =>
      when read4 =>
      end case;
end process;
```

```vhdl
end all_arch;
```
Section 4

OUTPUT BUFFERING

Moore output buffering

FSM as control circuit
  - Sometimes fast, glitch-free signal is needed
  - An extra output buffer can be added, but introduce one-clock delay

Special schemes can be used for Moore output
Moore Output Logic

Potential problems of the Moore output logic:
- Potential hazards introduce glitches
- Increase the Tco delay (Tco = Tcq + Toutput)

Look-ahead output circuit

Output buffer introduces one-clock delay

The “next” value of Moore output can be obtained by using state_next signal

Buffer the next value cancel out the one-clock delay

More systematic and easier to revise and maintain
Look-ahead output circuit (cont’d)

(a) Moore output with a simple output buffer
(b) Moore output with a look-ahead buffer

Summary

Sequential design in VHDL: instantiation of flip-flops and registers
– using a clk sensitive process and a branch like
  \[ ELSIF (clk'EVENT AND clk='1') THEN \]

Reset: asynchronous or synchronous
– Do not put combinational logic in front of reset or clk ports!

Finite state machines: Moore or Mealy
– Use of case statement for next state logic
– Implicit/explicit state encoding

Output buffering