CSE 535 : Lecture 2

Language-based Hardware Description

Washington University
Fall 2003

http://www.arl.wustl.edu/arl/projects/fpx/cse535/

Lockwood@arl.wustl.edu

Describing Hardware

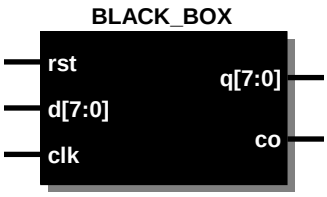
- Schematic-based
  - Can be cumbersome for large projects
  - Requires version control to maintain libraries
- Language-based
  - Must express parallelism
    - C does not have explicit commands for parallism
  - Must allow for *fine grain* parallelism
    - Allow *gate-level* parallelism where possible
  - Must be exact
    - Eliminate ambiguity
  - Synthesizable
    - Generate hardware from description

VHDL

- VHSIC Hardware Description Language
  - VHSIC = Very High Speed Integrated Circuit
  - IEEE 1076-1987
    - First ratified version of VHDL
    - Originally developed to *describe* digital system
      - [not build them!]
  - IEEE 1076-1993
    - Updated version, as used in textbook
  - IEEE 1164
    - Package for Synthesis

Example Entity declaration

```
ENTITY black_box IS PORT (
   clk: IN  std_logic;
   rst: IN  std_logic;
   d:   IN  std_logic_vector(7 DOWNTO 0);
   q:   OUT std_logic_vector(7 DOWNTO 0);
   co:  OUT std_logic);
END black_box;
```



The Architecture

- Architectures describe what is in the black box (i.e., the structure or behavior of entities)
- Descriptions can be either a combination of
  - Structural descriptions
    - Instantiations (placements of logic-much like in a schematic-and their connections) of building blocks referred to as components
  - Behavioral/Dataflow descriptions
    - Algorithmic (or "high-level") descriptions:
    - IF a = b THEN state <= state5;
    - Boolean equations (also referred to as dataflow):
    - x <= (a OR b) AND c;

The Architecture Declaration

```
ARCHITECTURE arch_name OF entity_name IS
    -- optional signal declarations, etc.
BEGIN
    --VHDL statements
END arch_name;
```

- arch_name is an arbitrary name
- optional signal declarations are used for signals local to the architecture body (that is, not the entity's I/O).
- entity_name is the entity name
- statements describe the function or contents of the entity

Architecture Body Styles : Behavioral

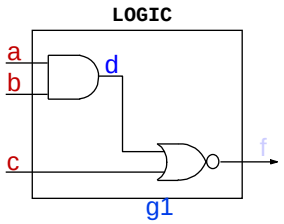
```
ENTITY compare IS PORT (
      a, b  : IN  std_logic_vector(0 TO 3);
      equals: OUT std_logic);
END compare;
ARCHITECTURE behavior OF compare IS
BEGIN
  comp: PROCESS (a,b)
  BEGIN
      IF a = b THEN
          equals <= '1' ;
      ELSE
          equals <= '0' ;
      END IF ;
  END PROCESS comp;
END behavior;
```


Architecture Body Styles : Structural

```
ENTITY compare IS PORT (
      a, b:   IN  std_logic_vector(0 TO 3);
      equals: OUT std_logic);
END compare;
USE WORK.Gate-Library.ALL ;
ARCHITECTURE structure OF compare IS
  SIGNAL x : std_logic_vector (0 to 3) ;
BEGIN
  u0: xnor2 PORT MAP (a(0),b(0),x(0)) ;
  u1: xnor2 PORT MAP (a(1),b(1),x(1)) ;
  u2: xnor2 PORT MAP (a(2),b(2),x(2)) ;
  u3: xnor2 PORT MAP (a(3),b(3),x(3)) ;
  u4: and4  PORT MAP (x(0),x(1),x(2),x(3),equals) ;
END structure;
```

Mixing Architecture Styles

- The various styles may be mixed in one architecture.




```
ENTITY logic IS PORT (
    a,b,c: IN std_logic;
    f:     OUT std_logic);
END logic;

USE WORK.GateLibrary.ALL;
ARCHITECTURE archlogic OF logic IS
    SIGNAL d: std_logic;
BEGIN
    d <= a AND b;                -- Behavioral
    g1: nor2 PORT MAP (c, d, f); -- Structural
END archlogic;
```


Native Operators

- Logical
  - AND, NAND
  - OR, NOR
  - XOR, XNOR
  - NOT
- Relational
  - = (equal to)
  - /= (not equal to)
  - < (less than)
  - <= (less than or equal to)
  - > (greater than)
  - >= (greater than or equal to)

VHDL Statements

- There are two types of statements
  - Sequential
    - Though hardware is concurrent, it may be modeled with algorithms, by a series of sequential statements
    - By definition, sequential statements are grouped using a process statement.
  - Concurrent
    - Statements outside of a process are evaluated concurrently during simulation
    - Processes are concurrent statements

Concurrent Statements

- Concurrent statements include:
  - Boolean equations
  - Conditional/selective assignments
  - Instantiations

```
-- Examples of boolean equations
   x <= (a AND (NOT sel1)) OR (b AND sel1);
   g <= NOT (y AND sel2);

-- Examples of conditional assignments
   y <= d WHEN (sel1 = '1') ELSE c;
   h <= '0' WHEN (x = '1' AND sel2 = '0') ELSE '1';

-- Examples of instantiation
   inst: nand2 PORT MAP (h, g, f);
```

The Process Statement

- An architecture can contain multiple processes.
- Each process is executed concurrently
- Statements within a process are sequential statements-they execute sequentially during simulation
- The statements with in a process not be sequential after synthesis

The Process (cont)

```
label: PROCESS (sensitivity list)
-- variable declarations
BEGIN
-- sequential statements
END PROCESS label ;
```

- The process label and variable declarations are optional
- The process executes when one of the signals in the sensitivity list has an event (changes value).

Process (cont)

- Processes are executing or suspended (active or inactive/awake or asleep)
- A process typically has a sensitivity list
  - When a signal in the sensitivity list changes value, the process is executed by the simulator
  - e.g., a process with a clock signal in its sensitivity list becomes active on changes of the clock signal
- All signal assignments occur at the END PROCESS statement in terms of simulation
  - The process is then suspended until there is an event (change in value) on a signal in the sensitivity list

Selective Signal Assignment: select

- Assignment based on a selection signal
- WHEN clauses must be mutually exclusive
- Use a WHEN OTHERS to avoid latches
- Only one reference to the signal, only one assignment operator (<=)

```
WITH selection_signal SELECT
signal_name <= value_1 WHEN value_1,
               value_2 WHEN value_2,
               ...
               value_n WHEN value_n,
               value_x WHEN OTHERS;
```

Combinational Logic with select

- The same 4-1 multiplexer is shown below

```
WITH s SELECT
     x <= a when "00" ,
          b when "01" ,
          c when "10" ,
          d when others ;
```


Conditional Signal Assignment: when-else

- Signal is assigned a value based on conditions
- Any simple expression can be a condition
- Priority goes in order of appearance
- Only one reference to the signal, only one assignment operator (<=)
- Use a final ELSE to avoid latches

```
signal_name <= value_1 WHEN condition1 ELSE
               value_2 WHEN condition2 ELSE
               ...
               value_n WHEN conditionN ELSE
               value_x ;
```

Combinational Logic (cont)

The same 4-1 multiplexer is shown below

```
x <= a when (s = "00") else
     b when (s = "01") else
     c when (s = "10") else
     d ;
```


Combinational Logic (cont)

- The when conditions do not have to be mutually exclusive (as in with-select-when)
- A priority encoder is shown below

```
j <= w when (a = '1') else
     x when (b = '1') else
     y when (c = '1') else
     z when (d = '1') else
    '0' ;
```

Sequential Statements: if-then-else

- Used to select a set of statements to be executed
- Selection based on a boolean evaluation of a condition or set of conditions

```
IF condition(s) THEN
  do something;
ELSIF condition_2 THEN  -- optional
  do something different;
ELSE                    -- optional
  do something completely different;
END IF ;
```


Avoiding the latch within an if-then-else

- Absence of ELSE results in implicit memory
- 4-1 mux shown below

```
mux4_1: process (a, b, c, d, s)
      begin
            if  s = "00" then x <= a ;
            elsif  s = "01" then x <= b ;
            elsif  s = "10" then x <= c ;
            else x <= d ;
            end if;
      end process mux4_1 ;
```

Sequentional Statements: Case-When

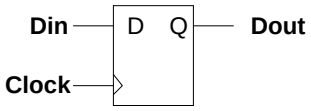
```
CASE selection_signal
  WHEN value_1_of_selection_signal  =>
      (do something)    -- set of statements 1
  WHEN value_2_of_selection_signal  =>
      (do something)    -- set of statements 2
...
  WHEN value_N_of_selection_signal  =>
      (do something)    -- set of statements N
  WHEN OTHERS  =>
      (do something)    -- default action
END CASE ;
```


The CASE Statement: 4-1 Mux

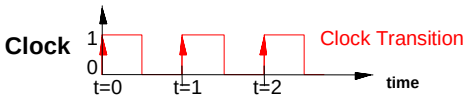
```
ARCHITECTURE archdesign OF design IS
    SIGNAL s: std_logic_vector(0 TO 1);
BEGIN
    mux4_1: PROCESS (a,b,c,d,s)
    BEGIN
          CASE s IS
                WHEN "00"  => x <= a;
                WHEN "01"  => x <= b;
                WHEN "10"  => x <= c;
                WHEN OTHERS => x <= d;
          END CASE;
    END PROCESS mux4_1;
END archdesign;
```

Synchronous Storage Elements

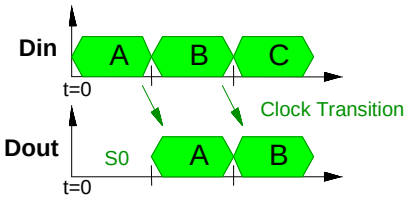
- Values change at times governed by clock



  - Clock
    - Input to circuit



  - Clock Event
    - Example: Rising edge
  - Flip/Flop
    - Transfers Value From D_{in} to D_{out} on Clock event

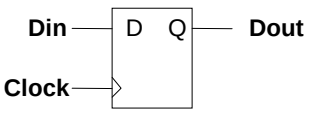


Edge Triggered Flop

```
entity flop is
  port (clk      : in std_logic;
        Din      : in std_logic;
        Dout     : out std_logic);
end flop;

architecture behavioral of flop is
begin
  flop:process(clk)
  begin
    if (clk='1' and clk'event) then
      Dout <= Din;
    end if;
  end process flop;

end behavioral;
```



Components

```
architecture structural of my_module is

 component flop32
   port(clk     : in  std_logic;
        Din     : in  std_logic_vector(31 downto 0);
        Dout    : out std_logic_vector(31 downto 0));
 end component;

   ... other components ...

 DataReg: flop32
   port map (clk     => clk,
             Din     => d_mod_in,
             Dout    => d_mod_out);

   ... other connections ...

end structural;
```


Finite State Machine (FSM)

- Terminology
  - Time: t
  - State: S
    - State variables: S = { S_1, S_2 … S_k }
    - Current state: S(t)
    - Next state: S(t+1)
  - State transition function δ()
    - Assigns next state
- State Transition
  - S(t+1) = δ(X , S(t))

State Declaration

```
type states is (init, pad, dout);
```


```
signal state, next_state : states;
```


State Declaration

- Binary Encoding

- Number of Flops = 2

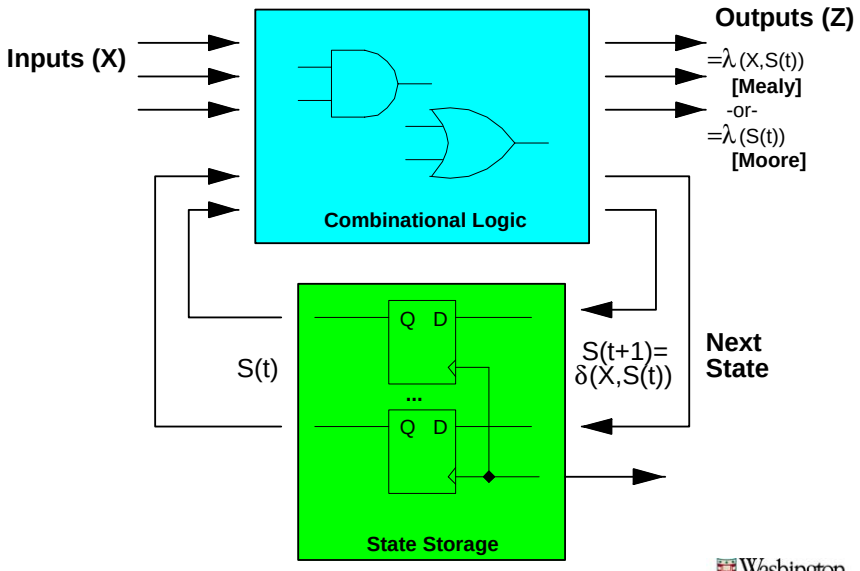
- One-Hot Encoding

- Number of Flops = 3

State Encoding

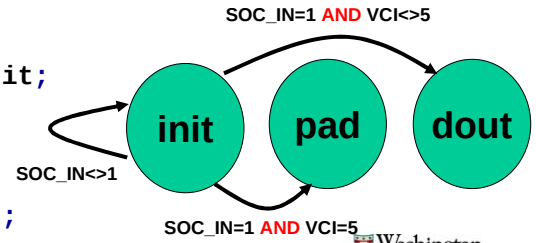
- Given state machine with K states
- Choose encoding to maintain state
  - Binary Encoding
    - Number of Flops: $O(\log_2 k)$
    - Minimizes number of Flops
  - One-Hot Encoding
    - Number of Flops: $O(k)$
    - Reduces the next-state logic
    - Uses fewer levels of logic cells
    - Enables high-speed state machines
  - Automatic Encoding
    - Feature of many synthesis tools

Implementation of the FSM



State Transition

```
state_trans:process(state, soc_in, cntr, data_in)
 begin
   case state is
     when init =>
       if (soc_in='1') then
         if (data_in(19 downto 4)=x"0005") then
           nxt_state <= pad;
         else
           nxt_state <= dout;
         end if;
       else
         nxt_state <= init;
       end if;
     ...
   end case;
 end process state_trans;
```

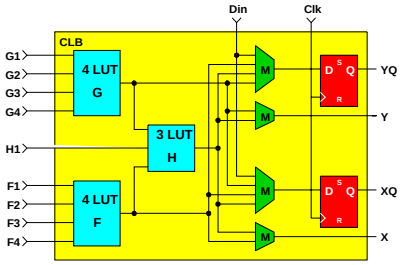


State Assignment Transition

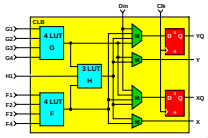
```
clkd: process(clk)
begin
  if (clk'event and clk='1') then
    if (reset='1') then
      state <= init;
    else
      state <= next_state;
    end if;
  end if;
end process clkd;
```

Technology Mapping

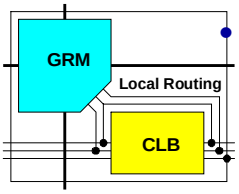
- Map function into library of gates
- Timing optimization



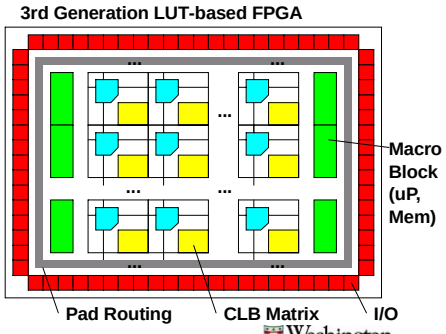
Reprogrammable Device Configuration



- CLB :
  - Primitive element of FPGA



- Routing Module :
  - Interconnection of Blocks



- FPGA :
  - Matrix of CLBs and Routing Modules

Final Design

- Physical Synthesis
  - Cell placement
  - Signal routing