Embedded Systems 1: Hardware Description Languages (HDLs) for Design

Davide Zoni PhD
email: davide.zoni@polimi.it
webpage: home.deib.polimi.it/zoni
Outline

- VLSI: History
  - From the single transistor to the HDL languages
  - Custom vs Standard Cells

- Hardware Description Languages
  - HDL vs Programming Language
  - SystemVerilog vs VHDL
  - Simulation and Synthesis Concepts

- Verilog for Design (subset)
  - Data-types, Module Structure
    Parametric Implementation
  - HDL Abstractions:
    - Gate, Data-flow, Behavioral
  - Always blocks, Hierarchical View
  - Combinatorial Logic
  - Sequential Logic and FSM

- Verilog for Simulation (subset) [next class]
  - Testbench
  - System tasks
    - $display, $monitor, $write,$display
  - Controlling the simulation
    - $finish, initial blocks
  - How the Simulation Flow Works

- Tools
  - HDL simulators
  - HDL synthesizers
  - Waveform Viewers

- Hands-on and Examples
  - Combinatorial
  - Sequential
  - FSM
  - Embedded Processor
Reference Books


The VLSI Evolution

Very Large Scale Integrated (VLSI) Circuits

- Late 1970s: the process of creating an integrated circuit made of thousands of transistors into a single chip.
- The Moore’s Law predict a double in the chip transistor count every 18 months
  - Nowadays almost any reasonable complex digital design follows in the VLSI category

Technology Libraries

- Full Custom Cells
  - Huge effort to design the ad-hoc cells for each design
- Semi-Custom Cells
- Standard Cells
  - Design the technology library once, then move the optimization effort to the circuit design stage up to the place&route

INTEGRATED Electronics (Intel)

- 1968: founded by Moore and Noyce with a focus on memory chips
- 1971: Intel 4004, 4-bit datapath @ 108KHz. 2300 transistors @ 10um
- 1972: Intel 8008, 8-bit datapath. 3500 transistors
- 1974: intel 8080, 2MHz 6000 transistors @ 6um
  - ...

The VLSI Evolution

Very Large Scale Integrated (VLSI) Circuits

- Late 1970s: the process of creating an integrated circuit made of thousands of transistors into a single chip.
- The Moore’s Law predict a double in the chip transistor count every 18 months
  - Nowadays almost any reasonable complex digital design follows in the VLSI category

Technology Libraries

- Full Custom Cells
  - Huge effort to design the ad-hoc cells for each design
- Semi-Custom Cells
- Standard Cells
  - Design the technology library once, then move the optimization effort to the circuit design stage up to the place&route

INTEGRATED Electronics (Intel)

- 1968: founded by Moore and Noyce with a focus on memory chips
- 1971: Intel 4004, 4-bit datapath @ 108KHz. 2300 transistors @ 10um
- 1972: Intel 8008, 8-bit datapath. 3500 transistors
- 1974: intel 8080, 2MHz 6000 transistors @ 6um
  - ...
The need for a better HW description methodology

**CURRENT STATUS:**
The increase in the transistor count in the same chip allows for:
- More functionalities
- More complex functions

**ISSUES:**
- The gate level design optimization (handmade) is not practical anymore
- (Possibly) design a custom cell for each gate of the design is not viable anymore

**NEW REQUIREMENTS:**
- A flexible, scalable and maintainable way to describe the hardware, also easing the optimization design stage
- Possibly a testing and verification language support can be a [huge] advantage
WHY HDLs? Hardware description language (HDL) is a device-independent representation of digital logic

- SystemVerilog is a hardware description (and verification) language
  - Actually it is more than this, while we focus on a subset of the entire specification.
    - Keep an eye on the complexity
    - We explore the widely used parts, that allows the design up to a complete SoC

- The History of Verilog:
  - 1983: Gateway Design Automation invented Verilog
  - 1990: Cadence bought Verilog. Transferred into the public domain and became a standard IEEE STD. 1364-1995 aka Verilog-95
  - Later versions:
    - Verilog 2001 (what we are considering in the rest of these classes)

- VHDL (VHSIC Hardware Description Language)
  - Published in 1987 with Dept. of Defence support as IEEE STD. 1076-1987

- Verilog and VHDL share the same HDL opportunities
  - Verilog is C-like
  - VHDL is Ada-like
Final Goals of These Classes

- Presenting the HDL basics to:
  - read and design simple combinatorial/sequential circuits and FSMs
  - Behavioral simulation of the design, only (post synthesis sim during ES1 projects)
  - From the specification to an HDL of the required design

- We are constrained to a subset of the SystemVerilog language
  - Reduce the complexity to start approaching HDLs (subset of Verilog 2001)
  - The focus is the parts that are really useful for real designs
  - This is not an HDL course

- A subset of the Verilog 2001 language is used
  - SystemVerilog and Verilog are used in the rest of this talk interchangeably with the clear meaning of Verilog 2001 subset
  - This is not limiting you in the offered functionalities, while the notation could be a little more verbose
The (Simplified) Hardware Design Flow

**Flow**

- **Verilog Code**
  ```verilog
  input a,b;
  output [1:0];
  assign sum={1'b0,a}+{1b'0,b};
  ```

- **Post Synthesis Netlist**

- **Post Mapped Code**
  G1 "and" n1 n2 n5
  G2 "and" n3 n4 n6
  G3 "or" n5 n6 n7

- **Post Place & Route Code**
  G1 "and_tlib" n1 n2 n5
  G2 "and_tlib" n3 n4 n6
  G3 "or_tlib" n5 n6 n7

**Possible Actions**

- **Behavioral simulation**
  - Semantic and syntax checks
  - No physical net/gate propagation delays

- **Post-synthesis simulation**
  - Semantic of the synthesized code
  - No physical net/gate propagation delays

- **Post-mapped simulation**
  - Semantic of the synthesized code
  - Physical gate propagation delays
  - No net propagation delays

- **Post-place and route simulation**
  - Semantic of the synthesized code
  - Physical gate propagation delays
  - No net propagation delays

**Netlist + accurate propagation delays**

```
Netlist + accurate propagation delays
Ready to go!
```
Verilog 2001: The Basic “Ingredients”
Verilog 2001: Data Types

- Vector of bits is the only data type we consider in these classes
  - No struct or float
  - Integer is a 32-bit vector
  - String can be declared as an n-bit vector where each char is 8-bit long
- Each bit of the vector can take on one of four values

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning Simulation</th>
<th>Meaning Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic zero</td>
<td>Logic zero</td>
</tr>
<tr>
<td>1</td>
<td>Logic one</td>
<td>Logic one</td>
</tr>
<tr>
<td>X</td>
<td>Unknown</td>
<td>Don’t care</td>
</tr>
<tr>
<td>Z</td>
<td>High impedance</td>
<td>High Impedance</td>
</tr>
</tbody>
</table>

- Set X in simulation where we want don’t care, thus helping the synthesis results.
- Moreover, it helps finding bugs
Bit literals

8’b0101_1111

- Binary literals
  - 8’b0000_0000
  - 8’b0xx0_1xx1
- Hexadecimal literals
  - 32’h0123_cdef
  - 16’haxxx
- Decimal literals
  - 32’d58
Verilog 2001: wire and reg

- **wire:**
  - used to denote a hardware net
    - wire [15:0] instruction;
    - Wire [7:0] byte_bus;

- **reg:**
  - It is a variable that can be used to implement both combinational and sequential logic
  - It does not identify a hardware register

- **NOTE:**
  - Verilog provides no type safety at all by connecting nets or regs.
  - Verilog only checks the width of the two connecting signals
Verilog 2001: Boolean Operators

- **Bitwise operators**: perform bit-sliced operations on vectors
  - \( \sim (4'b1010) = \{\sim 1, \sim 0, \sim 1, \sim 0\} = 4'b0101 \)
  - \( 4'b1010 \& 4'b0011 = 4'b0010 \)

- **Logical operators**: return one-bit result
  - \( a \&\& b \)

- **Reduction operators**: bit-wise operation on a single operand returning one-bit result
  - \( \& (4'b0101) = 0 \& 1 \& 0 \& 1 = 1'b0 \)

- **Comparison**: boolean test on two arguments

<table>
<thead>
<tr>
<th>Bitwise</th>
<th>Reduction</th>
<th>Logical</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>~a</td>
<td>&amp;a</td>
<td>!a</td>
<td>Relational</td>
</tr>
<tr>
<td>NOT</td>
<td>AND</td>
<td>NOT</td>
<td>(in)equality. Returns x when x or z in bits. Else 0 or 1, bit-wise comparison</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>~&amp;a</td>
<td>a &amp;&amp; b</td>
<td>a === b</td>
</tr>
<tr>
<td>AND</td>
<td>NAND</td>
<td>AND</td>
<td>a! == b</td>
</tr>
<tr>
<td>a | b</td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>OR</td>
<td>OR</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td>a ^ b</td>
<td>~</td>
<td></td>
<td>a |</td>
</tr>
<tr>
<td>XOR</td>
<td>NOR</td>
<td>OR</td>
<td>a === b</td>
</tr>
<tr>
<td>a~ ^ b</td>
<td>^</td>
<td>^</td>
<td>Case (in)equality. Returns 0 or 1 on a bit-by-bit comparison. x and z values matter</td>
</tr>
<tr>
<td>XNOR</td>
<td>XOR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Verilog 2001: The Module

- Modules are the basic building blocks in Verilog
  - Hardware blocks: made of synthesizable Verilog code
  - Testing blocks: made of non-synthesizable Verilog code
- Verilog designs consist of interconnected modules
  - A module can instantiate other modules as children
- NOTE: split between synthesizable VS non-synthesizable code
  - System tasks, delays using # char, tasks are not synthesizable
  - Several coding styles are not synthesizable: e.g. loops with a non-constant loop variable

```
module dut(in1,in2...,out1,out2...);
  input in1;
  input in2;
  output out;
  <impl of the module>
endmodule
```

The module’s behavior can be described in many different ways, but it should not matter from outside
Verilog 2001: How to Describe the Semantic of the Module
HDL: module body description

- **Behavioral**
  - High level description of a module semantic using all the features of the language: combination of always blocks and continuous assignments
  - **When:**
    - Non-synthesizable logic or complex module hierarchy
    - Let the synthesizer the maximum optimization opportunities

- **Data-Flow**
  - Continuous assignments, only. Description follows the data flow
  - **When:**
    - Is there a clear data flow structure to follow?

- **Gate-Level**
  - Use the basic gates only to describe the semantic of a module. No behavioral level constructs allowed
  - **When:**
    - Constraint to a specific implementation
module muxgate (a, b, out, outbar, sel);

input a, b, sel;
output out, outbar;
wire out1, out2, selb;
    and a1(out1, a, sel);
    not i1(selb, sel);
    and a2(out2, b, selb);
    or o1(out, out1, out2);
    not i2(outbar, out);
endmodule

Verilog supports for basic logic gates as primitives:
- and, nand, or, nor, xor, xnor, not, buf
- The net between each pair of gates is represented by means of a wire
A simple example: the multiplexerer (dataflow)

module muxdataflow (a, b, out, outbar, sel);

input a, b, sel;
output out, outbar;
assign out = sel ? a : b;
assign outbar = ~out;
endmodule

- Continuous assignments by using the assign keyword
  - models combinatorial logic in an easier way
  - The left side of the assign statement must be a wire
  - The right side can be both a reg or wire
- Dataflow operators
  - Conditional operator: (cond_expression) ? (value_if_true) : (value_if_false);
  - Arithmetic operator: +, -, (do not use *, **, / in synthesizable code)
  - Boolean operator: ~, &, |, ^ [unary and binary version]
- Nested conditional operator (4:1 mux)
  - assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0);
module muxbehavioral (a,b,out,
    outbar,sel);

input a,b,sel;

output reg out, outbar;

always@(a,b,sel)
beg
    if(sel)
        out = a;
    else
        out = b;
    outbar = ~out;
end
endmodule

Comments

• Anything assigned in an always block must be declared as reg

• always block is processed once whenever a signal in the sensitivity list changes value. Since Verilog-2001 the sensitivity list can be substituted with a * to get all the wire/variables to whom the always has to be “sensitive” at

• Statements in the always block are executed sequentially. (Order matters!)

• The begin/end keyword pair is used to group multiple statements within the same always block. Actually they are equivalent with the { / } in C/C++ code
module mux (a,b,out,outbar,sel);
  input a,b,sel;
  output reg out;
  output outbar;

  always@(a,b,sel)
  begin
    if(sel) out=a;
    else out=b;
  end
  assign outbar = ~out;
endmodule

• De-facto practice: **procedural blocks** and **continuous assignments** used side by side in the design of the same module

• Simulation viewpoint: **procedural blocks** and **continuous assignments** are evaluated in parallel
  • the order in the event processing is auto induced by the design, i.e. the always block is evaluated first with respect to the continuous assignment because of a dependency (by construction)

• Do not assign the same reg twice or more (overdrive error)
  • The same for the continuous assignments
The if-else Statement

```pascal
if [boolean_expr_1]
begin
  [procedural statement]+;
end
else if [boolean_expr_2]
begin
  [procedural statement]+;
end
. . .
else
begin
  [procedural statement]+;
end
```

- The boolean_expr_1 is evaluated first; if true its procedural statements are executed, otherwise the first else if expression is evaluated.
- It is possible to have multiple cascading else if.
- The if construct infers a priority routing network.
The *if-else* Statement: Example

```verilog
module prio_end_if (input [3:0] r, output reg [2:0] y);
  always@(*)
    if(r[3]==1'b1)
      y=3'b100;
    else if(r[2]==1'b1)
      y=3'b011;
    else if(r[1]==1'b1)
      y=3'b010;
    else if(r[0]==1'b1)
      y=3'b001;
    else
      y=3'b000;
endmodule
```

- The code checks the $r[4]$ first and eventually cascades to the subsequent `else if` statements one after the other until one of them matches or the `else` statement is reached.

- **NOTE:** both $r$ and $y$ are 4 values, i.e., 0, 1, Z, X.

- The final else statement allows to complete the if-else statement specification to prevent subtle, non-easily detectable bugs.
The case Statement

case [case_expr]

[item1]:
  begin
    [procedural statement]++;
  end

[item2]:
  begin
    [procedural statement]++;
  end

[item3]:
  begin
    [procedural statement]++;
  end

... 

default:
  begin
    [procedural statement]++;
  end
endcase

- The case statement is a multi-way decision statement that compares the [case_expr] with a number of items and execute the procedural statements of the first matching item.

- In the case of multiple matching items only the procedural statements of the first one are executed.

- It does not infer a priority network like the if-else statement.

- The default keyword is matched each time no item is matched. It is not mandatory to specify all the possible items.
module prio_enc_case(input [3:0] r,
    output reg [2:0] y
always@(*)
case(r)
    4'b1000, 4'b1001, 4'b1010, 4'b1011, 4'b1100, 4'b1101, 4'b1110, 4'b1111:
        y=3'b100;
    4'b0100, 4'b0101, 4'b0110, 4'b0111:
        y=3'b011;
    4'b0010, 4'b0011:
        y=3'b010;
    4'b0001:
        y=3'b001;
    4'b0000:
        y=3'b000;
edcase
endmodule

- The case statement can mimic the priority enforced by the if-else statement
  - it seems a stretch

- The 0 and 1 values are not the only values the r and items can assume (hints: X and Z)
  - Maybe a more flexible case statement can fit the task if the priority is necessary
  - The non complete specification can be dangerous... (discussed after)
**The casez and casex Statements**

- The **casez** statement is a multi-way decision statement that allows the `z` value and `?` character in the item expression as don't-cares:
  - It ignores each bit position with a `Z`
  - Use it for synthesis if necessary

- The **casex** statement is a multi-way decision statement that allows the `z` and `x` values as well as the `?` character in the item expression as don't-cares:
  - It ignores each bit position with a `Z` or `X`
  - Don't use the **casex** statement due to its subtleties in treating `x` values that can thus mask bugs
  - `x` value semantic is different between simulation and synthesis

```plaintext
```
casez (expr)           casex (expr)
```
```
3'bzzz: ...            3'bzxz: ...
3'b???: ...            3'b???: ...
3'b?zz: ...            3'b?xx: ...
endcase                 endcase
```

**EXAMPLE**

- Matching with multiple items is possible
  - The first coming is selected one, thus inferring a priority network
- `expr=3'bxxx`
  - No match in the casez
  - Always match in the casex
- `expr=3'bzzz`
  - Always match for both

**NOTE:**

- `casex` and `casez` use the identity operator `===` to compare the `case_espr` with each item
- `casex` and `casez` are symmetric since it does not matter if the special value/character appears in the `item` or in the `case_espr`
- `?` is a placeholder / character; `X` / `Z` are values
- The use of `?` in the `item` is preferred than `X` / `Z`
Inferred Unintended Latches in Comb. Circuits

module mux_wrong(input [1:0] sel, output reg [1:0] y)
always@(*)
    case(r)
        2'b00: y=2'b00;
        2'b01: y=2'b01;
        2'b10: y=2'b10;
    endcase
endmodule

module mux_correct(input [1:0] sel, output reg [1:0] y)
always@(*)
    case(r)
        2'b00: y=2'b00;
        2'b01: y=2'b01;
        2'b10: y=2'b10;
        default: y=2'bxx;
    endcase
endmodule

• What we have when sel == 2'b11?
  ➔ The old value. Is that what we intended to infer? (probably not!)
  ➔ We inferred an unintended latch to eventually keep the old value

• The default special case item can help preventing such a behavior
  ➔ Simulation: propagates the X and Z values thus eases the bug detection
  ➔ Synthesis: X values are optimized as treated as don’t-care

• The same for the if-else statement
  ➔ Use the else statement to catch all not listed possibilities
Use parameters to improve the design flexibility

module mux #(parameter W=2) 
  (input [1:0] sel,  
   input [W-1:0] a,  
   input [W-1:0] b,  
   input [W-1:0] c,  
   input [W-1:0] d,  
   output reg [W-1:0] y); 

always@(*)
  case(sel)
    2'b00: y=a;
    2'b01: y=b;
    2'b10: y=c;
    2'b10: y=d;
    default: y={W{1'bx}};
  endcase
endmodule

• The parameter keyword allows for a viable way to reuse the same design using
  ➔ Same behavior different signal widths
  ➔ Passing options to a design at the elaboration time

• NOTE: Sometimes the parameter alone is not enough to make the module flexible
  ➔ Hints: think about an N-input mux where N is a parameter. The sel signal width becomes parametric...
Nested modules and parameters

module mux #(parameter W=2)
  (input [1:0] sel,
   input [W-1:0] a, input [W-1:0] b,
   input [W-1:0] c, input [W-1:0] d,
   output reg [W-1:0] y);
  . . .//(defined as before)
endmodule

`define WGLOBAL 2
module parent(...)
  wire [`WGLOBAL-1:0] a,b,c,d,y;
  wire sel[1:0];
  mux #(.`W(`WGLOBAL))
    mux0(
      .a(a), .b(b), .c(c), .d(d),
      .sel(sel), .y(y)
    );
  ... //rest of the parent module logic
endmodule

- The parameter keyword allows for a viable way to reuse the same design:
  - Same behavior but different signal widths
  - Passing options to a design at the elaboration time

- NOTE: Sometimes the parameter alone is not enough to make the module flexible
  - Hints: think about an N-input mux where N is a parameter. The sel signal width becomes parametric ...
From Combinatorial
To Sequential Logic Circuits
Non-blocking VS Blocking Assignments

Beginning with $a=2$, $b=3$

```plaintext
... 
reg a[3:0];
reg b[3:0];
reg c[3:0];
always@(posedge clk)
begin
  a <= b;
  c <= a;
end
...
```

Afterwards $a=3$, $c=2$;  
[non-blocking ($<=$)]

```plaintext
... 
reg a[3:0];
reg b[3:0];
reg c[3:0];
always@(posedge clk)
begin
  a=b;
  c=a;
end
...
```

Afterwards $a=3$, $c=3$  
[blocking (=)]
Non-blocking VS Blocking Assignments

Blocking (=) and non-blocking (<=) assignments can be used within the always block, only

Differences:

- **Blocking Assignment**: evaluated (auto-ordered) one after the other
- **Non-blocking Assignment**: within the same always block the right side of each statement is frozen and it is used to update all the left side of the corresponding statement. Then, all the variables/signals are updated in parallel at once

Practical Usage Rules:

- The standard allows to mix and match **blocking** and **non-blocking** assignments together in the same always block
- In this course it is mandatory to use:
  - Combinatorial logic: blocking assignments only
  - Sequential logic: non-blocking assignments only
The flip-flop

module ff(input clk, input d, output reg q);
    always@(posedge clk)
    begin
        q<=d;
    end
endmodule

module ff_en(input clk, input en, input d, output reg q);
    always@(posedge clk)
    begin
        if(en)
            q<=d;
    end
endmodule
The flip-flop with (asynch/synch) reset

module ff_rst_a(input clk, input rst,  
                 input d, output reg q);
  always@(posedge clk, posedge rst)  
  begin  
    if(rst)  
      q<=d;  
  end  
endmodule

module ff(input clk, input rst,  
          input d, output reg q);
  always@(posedge clk)  
  begin  
    if(rst)  
      q<=d;  
  end  
endmodule
The Register

Standalone Component

module register #(parameter W=2) 
(
    input clk,
    input [W-1:0] d,
    output reg [W-1:0] q
);

    always@(posedge clk)
    begin
        q <= d;
    end
endmodule

As made of ff modules

module register #(parameter W=2) 
(
    input clk,
    input [W-1:0] d,
    output reg [W-1:0] q
);

    ff ff0(.clk(clk), .d(d), .q(q));
    ff ff1(.clk(clk), .d(d), .q(q));
endmodule

They share the same semantic...
More on the Non-blocking Assignments
... 
wire [2:0] a_i, b_i, c_i; 
reg [2:0] a, b, c;

always@(posedge clk) 
begin 
  a <= a_i;  
  b <= a+1;  
  c <= b+1; 
end 
...

- The order of the non-blocking assignments within the same always block does not matter

- The structure of the always makes the synthesizer to infer flip-flops
... 
wire [2:0] a_i, b_i, c_i;
reg [2:0] a, b, c;

always@(posedge clk)
begin
  a <= a_i;
  b <= b_i;
  c <= c_i;
end
assign b_i = a + 1;
assign c_i = b + 1;
...

• Same semantic; combinatorial logic is outside the always block
• The order of the non blocking assignment does not matter
...  
wire [2:0] a_i, b_i, c_i;
reg [2:0] a, b, c;

always@(posedge clk)
begin
  a <= a_i;
  b <= b_i;
  c <= c_i;
  assign b_i = a + 1;
  assign c_i = b + 1;
end

...
...wire [2:0] a_i, b_i, c_i;
reg [2:0] a, b, c;

always@(posedge clk)
a <= a_i;
always@(posedge clk)
b <= b_i;
always@(posedge clk)
c <= c_i;

assign b_i = a + 1;
assign c_i = b + 1;
...

• Same (CORRECT) behavior
  → Concurrent (sequential) always blocks, the same circuit is inferred
... wire [2:0] a_i, b_i, c_i; reg [2:0] a, b, c;
always@(posedge clk) begin
  a = a_i;
  b = b_i;
  c = c_i;
end
assign b_i = a + 1;
assign c_i = b + 1;
...

- **Syntactically correct** (it can be synthesized)
- **Semantically broken**
  - Blocking assignments do not reflect the intrinsic behavior of the sequential logic
    - The order of the statements matters!
  - Don’t use blocking assignments in sequential logic
- **IT IS FORBIDDEN IN THIS COURSE**
The simulator generates and processes the events following a strict policy

• **For a sequential circuit**
  1) update the variables/wire in the combinatorial constructs:
     - Continuous assignments
     - `always@(*)`
     - NOTE: Recursive actions can happen
  2) evaluate all the right hand of each non-blocking assignment (unordered), and defer the update of the left side of the statement
  3) update the left side of the statement

More details in the next class (SystemVerilog for Verification)
Finite State Machines
module fsm_light(input clk, 
        input button, 
        output reg light);

wire light_next;

always@(posedge clk)
    light <= light_next;

always@(*)
begin
    //prevent unintended latches
    //light_next = light;
    if(button)
        light_next=1'h1;
    else
        light_next=1'h0;
end
endmodule

Hello World FSM

- The FSM has two (unnamed) states: ON, OFF
- The datapath uses the input to compute the next state
- What about if button is X?

  light_next = light; but the semantic is lost
Finite State Machines (FSMs)

Mealy FSM
- Output
  - Depends on input and state
  - Not synchronized with the clock
  - Temporarily unstable output
  - Changes during transitions

Moore FSM
- Output
  - Depends on state only
  - Associated with FSM states
The Simple Vending Machine Example

Problem Specification:

- No change (sorry I’m too simple :p)
- Product delivery after inserting 0.3€. Single product
- Accepted Coins 0.1€ 0.2€. Single coin slot
- The Hard Reset input signal resets the vending machine
- Single idle cycle after completing one transaction
- No X at the FSM primary inputs (ideal scenario for now)
module fsm_moore(input clk, input rst, input ten_i, input twenty_i
             output open_o);

  reg [1:0] ss;
  localparam IDLE=2’b00, TEN=2’b01, TWENTY=2’b10, OPEN_CMD=2’b11;

  always@(posedge clk)
  begin
    if(rst) ss<=0;
    else
      begin
        case(ss)
          IDLE: if(ten_i) ss<=TEN;
               else if(twenty_i) ss<=TWENTY;
               else ss<=IDLE;
          TEN: if(ten_i) ss<=TWENTY;
               else if(twenty_i) ss<=OPEN_CMD;
               else ss<=TEN;
          TWENTY: if(ten_i) ss<=OPEN_CMD;
                   else if(twenty_i) ss<=OPEN_CMD;
                   else ss<=TWENTY;
          OPEN_CMD: ss<=IDLE; //send an impulse only
        endcase //endcase
      end //endif
    end //end always

  assign open = (ss==OPEN_CMD);
endmodule

**Simplifications**

- Leave one clock cycle before two transactions
  - (to not bother the state diagram with additional arrows)
- Assume no X can appear as primary input to the FSM. The X means don’t care in the state diagram
module fsm_moore(input clk, input rst, input ten_i, input twenty_i
  output reg open_o);

  reg [1:0] ss, ss_next;
  localparam IDLE=2’b00, TEN=2’b01, TWENTY=2’b10, OPEN_CMD=2’b11;

  always@(posedge clk)
    ss<=ss_next;
  always@(twenty_i, ten_i, ss, rst)
    begin
      if(rst) begin ss_next=0; open=0; end
      else
        begin
          case(ss)
            IDLE: begin
              open=0;
              if(ten_i) ss_next=TEN;
              else if(twenty_i) ss_next=TWENTY;
              else ss_next=IDLE;
            end
            TEN: begin
              open=0;
              if(ten_i) ss_next=TWENTY;
              else if(twenty_i) ss_next=OPEN_CMD;
              else ss_next=TEN;
            end
            TWENTY: begin
              open=0;
              if(ten_i) ss_next=OPEN_CMD;
              else if(twenty_i) ss_next=OPEN_CMD;
              else ss_next=TWENTY;
            end
            OPEN_CMD: begin ss_next=IDLE; open=1; end //send an impulse only
          endcase
        end
    end //endif
endmodule

Simplifications
- Not optimized
- Assume no X can appear as primary input to the FSM.
- The X means don’t care in the state diagram
Simple Vending Machine: Mealy FSM

module fsm_moore(input clk, input rst, input ten_i, input twenty_i
    output reg open_o);

    reg [1:0] ss, ss_next;
    localparam IDLE=2'b00, TEN=2'b01, TWENTY=2'b10, OPEN_CMD=2'b11;

    always@(posedge clk)
        ss<=ss_next;
    always@(twenty_i, ten_i, ss, rst)
        begin
            if(rst) begin ss_next=0; open=0; end
            else begin
                case(ss)
                    IDLE: begin
                        open=0;
                        if(ten_i) ss_next=TEN;
                        else if(twenty_i) ss_next=TWENTY;
                        else ss_next=IDLE;
                    end
                    TEN: begin
                        open=0;
                        if(ten_i) ss_next=TWENTY;
                        else if(twenty_i) begin ss_next=IDLE; open=1; end
                        else ss_next=TEN;
                    end
                    TWENTY: begin
                        open=0;
                        if(ten_i || twenty_i) begin ss_next=IDLE; open=1; end
                        else ss_next=TWENTY;
                    end
                endcase
            end
        end
    endmodule

Simplifications

- The Mealy FSM is faster and requires less states, in general compared with the equivalent Moore one
- Assume no X can appear as primary input to the FSM.
- The X means don’t care in the state diagram
An Rising Edge Detector (Moore)

The rising edge detector is a circuit that generates a short one-clock-cycle tick when the input signal changes from 0 to 1

POSSIBLE USAGE: identify the onset of a slow time-varying input signal

module edge_detector_moore
(input clk, input rst,
 input level,
 output reg tick);

localparam [1:0] ZERO=2'b00, EDGE=2'b01, ONE=2'b10;
reg [1:0] ss, ss_next;

// FSM state register update
always@(posedge clk,posedge rst)
  if(rst) ss<=zero;
  else ss<=ss_next;

cont’d

// combinational logic
always@(*)
begin
  ss_next=ss; tick=1’b0;
  case(ss)
    ZERO: if(level) ss_next=edge;
    EDGE: begin
      tick=1’b1;
      if(level) ss_next=one;
      Else ss_next= zero;
    end
    ONE: if(~level) ss_next=zero
    default: ss_next=zero;
  end
end
endmodule

cont’d here
The rising edge detector is a circuit that generates a short one-clock-cycle tick when the input signal changes from 0 to 1

POSSIBLE USAGE: identify the onset of a slow time-varying input signal

```verilog
module edge_detector_moore
(input clk, input rst,
 input level,
 output reg tick
);

localparam ZERO=1'b0, ONE= 1'b1;
reg [1:0] ss, ss_next;

// FSM state register update
always@(posedge clk,posedge rst)
  if(rst) ss<=zero;
  else ss<=ss_next;

cont’d here
// combinatorial logic
always@(*)
begin
  ss_next=ss; tick=1'b0;
  case(ss)
  begin
    ZERO: if(level) begin
      ss_next=ONE;
      tick=1'b1;
    end
    ONE: if(~level) ss_next=zero;
    default: ss_next=zero;
  end
  end
end
endmodule
```

```
cont’d
```
Tools
Behavioral Simulation and Waveform Analysis

- **CADENCE (ASIC flow)**
  - ncsim RTL simulator
  - Simvision

- **Xilinx (FPGA flow)**
  - Vivado
    - xvlog, xelab, compile and elaborate the design
    - xsim simulator with integrated waveform analyzer
  - ISE
    - GUI-based flow
    - The command-line tools are not user friendly

- **Open Source (Behavioral Simulation)**
  - Icarus verilog (Verilog 2001 simulator)
  - Gtkwave waveform analyzer