CS61C : Machine Structures

Lecture 4.2.1

FSMs and Verilog I

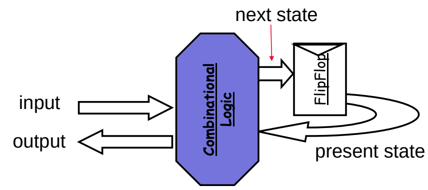
2004-07-14

Kurt Meinz

`inst.eecs.berkeley.edu/~cs61c`

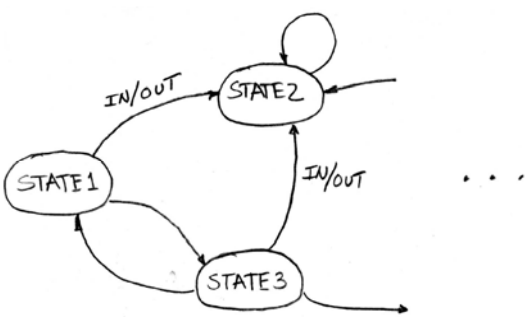
FSMs

- **With state elements, we can build circuits whose output is a function of inputs and current state.**



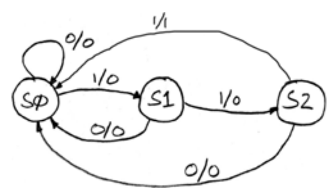
- **State transitions will occur on clock edges.**

Finite State Machines Introduction



Finite State Machine Example: 3 ones…

Draw the FSM…

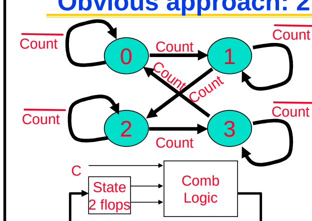


PS	Input	NS	Output
00	0	00	0
00	1	01	0
01	0	00	0
01	1	10	0
10	0	00	0
10	1	00	1

FSM Example 2

- **Two bit counter:**
  - **4 States: 0, 1, 2, 3**
  - **When input c is high, go to next state**
    - **(3->0)**
  - **When input is low, don't change state**

Obvious approach: 2 count bits for state

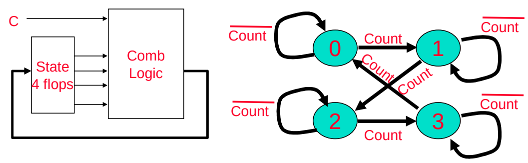


S_1	S_0	C	S_1'	S_0'
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

$$S_0' = (\overline{S_1} \cdot \overline{S_0} \cdot C) + (\overline{S_1} \cdot S_0 \cdot \overline{C}) + (S_1 \cdot \overline{S_0} \cdot C) + (S_1 \cdot S_0 \cdot \overline{C})$$
$$= (\overline{S_0} \cdot C) + (S_0 \cdot \overline{C})$$

$$S_1' = (\overline{S_1} \cdot S_0 \cdot C) + (S_1 \cdot \overline{S_0} \cdot \overline{C}) + (S_1 \cdot \overline{S_0} \cdot C) + (S_1 \cdot S_0 \cdot \overline{C})$$
$$= (\overline{S_1} \cdot S_0 \cdot C) + (S_1 \cdot \overline{C}) + (S_1 \cdot \overline{S_0})$$

One-Hot Encoding



- One Flip-flop per state
- Only one state bit = 1 at a time
- Much faster combinational logic
- Tradeoff: Size ⇔ Speed

$$S_0' = (S_0 \cdot \overline{C}) + (S_3 \cdot C)$$
$$S_1' = (S_1 \cdot \overline{C}) + (S_0 \cdot C)$$
$$S_2' = (S_2 \cdot \overline{C}) + (S_1 \cdot C)$$
$$S_3' = (S_3 \cdot \overline{C}) + (S_2 \cdot C)$$

Digital Design

- Our Conceptual Tools So Far:
  - Logic Schematics
  - Truth Tables
  - Boolean Algebraic Expressions
  - State Diagrams
- How are these tools used to design real hardware? …

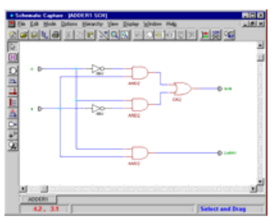
Digital Design – Tools - Schematics

☺Schematics are intuitive.

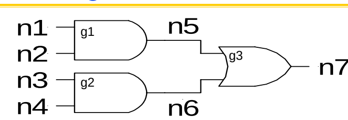
☺Somewhat physical.

☹Require a special tool (editor). →

☹Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow.



Digital Design – Tools - Netlists

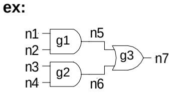


- Textual representation of logic blocks:
  - Give a name to all gates
  - Give a name to all wires
  - Standard representation of wire connections among gates …
    - Wire connections ←→ network ←→ nets
      - (Hence: "netlist")

Digital Design – Tools - Netlists

- A key data structure (or representation) in the design process is the "netlist":
  - Network List
- A netlist lists components and connects them with nodes:

ex:




```
g1 "and" n1 n2 n5
g2 "and" n3 n4 n6
g3 "or" n5 n6 n7
```

Alternative format:

```
n1 g1.in1
n2 g1.in2
n3 g2.in1
n4 g2.in2
n5 g1.out
g3.in1
n6 g2.out
g3.in2
n7 g3.out
g1 "and"
g2 "and"
g3 "or"
```


Digital Design – Tools - Netlists

- Netlist Pros:
  - Textual Layout
  - Imply physical organization
  - Automated tools can turn netlist into hardware layout
- Netlist Cons:
  - Lack of Abstraction

Digital Design – Tools - HDLs

- Hardware Description Languages:
  - Contain hierarchical netlist support
    - E.g. “Structural Verilog”
  - Contain support for logic testing, rough prototyping, architectural simulation
    - E.g. “Behavioral Verilog”
  - Other integrated design paradigms (and mappings to netlists)
    - E.g. “Verilog Dataflow”

Verilog Overview ()

- Verilog description composed of modules:

  module *Name (port list) ;*

  Declarations and Statements;

  endmodule
- Modules can have instantiations of other modules, or use primitives supplied by language
- Note that Verilog varies from C syntax, borrowing from Ada programming language at times (`endmodule`)

Verilog Overview (1/3)

- Structural Verilog:
  - Like netlists, but with support for module naming and instantiation.
    - E.g. build a 4->1 mux and instantiate many copies
    - Module ports = input and output interface
    - Same benefits as other OO systems.
  - For simulation purposes, includes notion of time (delay)
    - Useless for synthesis
    - Necessary for simulation

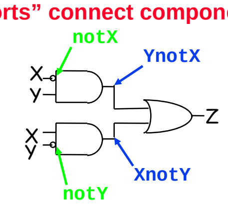
Example: Structural XOR (xor built-in,but..)

```
module my_xor(Z, X, Y);
 input X, Y;
 output Z;
 wire notX, notY,
  XnotY, YnotX;
 not
  (notX, X),
  (notY, Y);
 and
  (YnotX, notX, Y),
  (XnotY, X, notY);
 or
  (Z, YnotX, XnotY);
endmodule
```

which “ports” input, output

Default is 1 bit wide data

“ports” connect components



Note: order of gates doesn’t matter, since structure determines relationship

Verilog: Modules

- Now, other modules can instantiate our new xor module:

```
module 1-bit-adder(A, B, Ci, Co, S);
 input A, B, Ci;
 output Co, S;
 wire ttemp;
 my_xor Xor1 ( temp,       A,       B   ),
             Xor2 ( .X(temp),  .Y(Ci),  .Z(Co) );
 …
endmodule
```

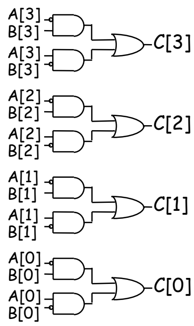

Verilog: Replication

- Often in hardware need many copies of an item, connected together in a regular way
  - Need way to name each copy
  - Need way to specify how many copies
- Replicated wires ➔ Bus!
- Specify a module with 4 XORs using existing module example …

Example: Replicated XOR in Verilog

```
module 4xor(C, A, B);
 input[3:0] A, B;
 output[3:0] C;
 my_xor My4XOR[3:0]
   (.X(A), .Y(B), .Z(C) );
endmodule
```

- Note: must give a name to new instance of xors (My4XOR)



Verilog - Structural

- If you were to $monitor an instance of our XORs you'd notice something funny …

```
x=0, y=0, z=0, exp=0, time=0
x=0, y=1, z=1, exp=1, time=100
x=1, y=0, z=1, exp=1, time=200
x=1, y=1, z=0, exp=0, time=300
```

- Expected value matches actual value, so Verilog works, but …
- 0 Propagation delay!!

Verilog big idea: Time

- A difference from normal prog. lang. is that time is part of the language
  - part of what trying to describe is when things occur, or how long things will take
- Determine time with #n : event will take place in n time units
  - structural: not #2(notX, X) says notX does not change until time advances 2 ns
  - Default unit is nanoseconds; can change

Structural XOR With Delay

```
module my_xor(Z, X, Y);
 input X, Y;
 output Z;
 wire notX, notY,
   XnotY, YnotX;
 not #2
  (notX, X),
  (notY, Y);
 and #3
  (YnotX, notX, Y),
  (XnotY, X, notY);
 or #3
  (Z, YnotX, XnotY);
endmodule
```


Verilog - Structural

```
x=0, y=0, z=x, exp=0, time=0
X=0, y=0, z=0, exp=0, time=8

x=0, y=1, z=0, exp=1, time=100
x=0, y=1, z=1, exp=1, time=108

x=1, y=0, z=1, exp=1, time=200
x=1, y=0, z=1, exp=1, time=208

x=1, y=1, z=1, exp=0, time=300
x=1, y=1, z=0, exp=0, time=308
```

☹Times not exact! – Really depend on transitions

Verilog

- CL can be done by wiring up gates
  - If we use abstraction and replication, we can control complexity
- What about state?
  - Could build D from NAND netlist … blech!
  - Verilog also provides ability to describe circuits by behavior rather than structure…

Example: Behavioral XOR in Verilog

```
module xorB(Z, X, Y);
 input X, Y;
 output Z;
 reg Z;
 always @ (X or Y)
    Z = X ^ Y; // ^ is C operator for xor
endmodule
```

- Unusual parts of above Verilog
  - "`always @ (X or Y)`" => whenever X or Y changes, do the following statement
  - "`reg`" is only type of behavioral data that can be changed in assignment, so must redeclare `Z` as `reg`
  - Default is single bit data types: X, Y, Z

Behavioral

- How about registers/flip-flops?
  - Rising clock edge? ("positive edge triggered")
  - Falling clock edge? ("negative edge triggered")
- Verilog Includes events "posedge", "negedge" to say when clock edge occurs …

Behavioral FFs

```
// Behavioral model of D FF:
// positive edge-triggered,
// synchronous active-high reset.
module DFF_SR (CLK,Q,D,RST);
 input D;
 input CLK, RST;
 output Q;
 reg Q;
 always @ (posedge CLK)
   if (RST) Q = 0; else Q = D;
endmodule
```

- On positive clock edge, either reset, or load with new value from D

Behavioral Clock

```
...
initial
 begin
  CLK = 1'b0;
  forever
   #5 CLK = ~CLK;
 end
...
```

New things:
- exec block once
- 1 bit in binary = 0
- while (true)

- No built in clock in Verilog, so specify one
  - Clock CLK above alternates forever in 10 ns period:
    5 ns at 0, 5 ns at 1,
    5 ns at 0, 5 ns at 1, …

Behavioral Adder

```
module add4 (S,A,B);
```

- a combinational logic block that forms the sum (S) of the two 4-bit binary numbers (A and B)
- Tutorial doesn't define this, left to the reader
- Write the Verilog for this module in a behavioral style now
  - Assume this addition takes 4 ns

Behavioral Adder

```
module add4 (S,A,B);
 input [3:0] A, B;
 output [3:0] S;
 reg S;
  always @(A or B)
    #4 S = A + B;
endmodule
```

- a combinational logic block that forms the sum of the two 4-bit binary numbers, taking 4 ns
- Above is behavioral Verilog

In conclusion

- Verilog allows both structural and behavioral descriptions, helpful in testing
- Syntax a mixture of C (operators, for, while, if, print) and Ada (begin… end, case…endcase, module …endmodule)
- Some special features only in Hardware Description Languages
  - # time delay, initial vs. always
- Verilog can describe everything from single gate to full computer system; you get to design a simple processor