ECE 551

Digital Design And Synthesis

Fall'23

Lecture 1

Instructor Introduction
Course Introduction
Verilog Basics
Structural Verilog
Statemachine & Bubble Diagram Example

1

Instructors

Eric Hoffman

- Teaching Faculty
- 25+ years industry experience doing Integrated Circuit & System design
  - 10 years at Intel
  - 7+ years at ZMD (Mixed signal, Analog/Digital IC's)
  - 9+ years as independent consultant
- Instructing experience:
  - ECE555, ECE554, ECE552, ECE551, ECE353, ECE352

Harish Batchu= TA

- harishbabu.batchu@wisc.edu

2

2

Instructor and TA Office Hours

- Eric Hoffman
  - 3615EH & MS Teams
  - Office Hours
    - Mon & Thurs: 2:30 to 4:00PM & Thurs 2:30 – 4:00 (3615EH). Sun: 9:30 – 10:30PM (MS Teams)
  - email = erichoffman@wisc.edu but use ece551_staff@g-groups.wisc.edu
- TA = Harish Batchu
  - Location = B555 & MS Teams
  - Tues 5:30 – 6:30PM (B555), Weds 6:30 – 8:00PM (MS Teams)
    - Available via email: **harishbabu.batchu@wisc.edu**
- UGSAs = Khailanii Slaton & Abhipriya Bansal
  - Help during in class exercises.
- Discussion Sessions hosted by Harish
  - Tuesday 6:45 – 7:45, in person in 2535EH
  - Thursday 6:45 – 7:45PM, via General Channel of MS Teams.
  - Some weeks discussion will be tutorials
    - In fact your first discussion will be ModelSim/QuestaSim Tutorial

3

3

Course Goals

- Provide knowledge and experience in:
  - Digital circuit design using a HDL (Verilog & SVerilog)
  - HDL simulation
  - Good practices in digital design verification
    - How to build self checking test benches
  - Synthesis of dataflow and behavioral designs
    - Optimizing hardware designs (timing, area, power)
  - Basic static timing analysis concepts
  - Introduce APR (Auto Place & Route)
  - Design tools commonly used in industry
- Teach you to be able to "think hardware" first, then code it in an HDL.

4

4

Course Structure (mostly inverted)

- Video Lectures (with subsequent quizzes)
- Hands on Exercises every week using: **ModelSim/Quartus/Synopsys**
- Exercises
  - Most individual, some as team of 2
  - Verilog DUT and testbench creation & simulation
  - Sometime mapping it to std cell (Synopsys)
- First few classes will be more quizzes on material because we do not have enough experience yet to do any meaningful "real work".

5

5

What You Should Already Know

- Principles of basic digital logic design (ECE 352)
  - Number representations (unsigned, signed, Hex & Binary)
  - Boolean algebra
  - Gate-level design
  - K-Map minimization
  - Finite State Machines
  - Basic datapath structures (adders, shifters, SRAM)
  - Some basic ModelSim and verilog from ECE352 AHW's
- How to log in to CAE machines and use a Linux shell

6

6

Course Website

- Canvas webpage:
- What the Website will have:
  - Lecture Notes (I will try to stay 1 week ahead of class)
  - Lecture Videos and Quizzes
  - In Class Exercise Descriptions with intro videos
  - Homework Assignments
  - Tutorials & Supplemental Information
  - Project Information
  - Exams Info

7

7

Course Materials

- Lecture Slides
- No Textbook Necessary
  - Can get by with the Standards & Lecture slides
- Standards
  - IEEE Std.1364-2001, IEEE Standard Verilog Hardware Description Language, IEEE, Inc., 2001.
  - IEEE Std 1364.1-2002, IEEE Standard for Verilog Register Transfer Level Synthesis, IEEE, Inc., 2002
- Tutorials on Canvas page for various CAD tools used
- Other useful readings

8

8

Evaluation and Grading

- Approximately: Graded on a curve. Class average grade will be around 3.25
  - 15% Homework
  - 11% Quizzes on video lectures (lowest 1 score dropped)
  - 10% In class exercise results (lowest 2 scores dropped)
  - 2% Cummings paper & Sutherland/Mills paper quizzes
  - 22% Project (4 person teams)
  - 20% Midterm
  - 20% Final
- Homework due 11:55PM on due date
  - **15% penalty** for each late period of 24 hours
  - Not accepted >48 hours after deadline

9

9

Class Project

- Work in groups of 4 students
- Design, model, simulate, synthesize, and test a complex digital system.
- Most important single component of grade & class
  - In class exercises focus on building components needed
  - Homework problems that build toward the project
  - Exams will have some questions project related
  - Final code review and testing will be detailed
- More details coming later in the course

10

10

Course Tools

- Industry-standard design tools:
  - Modelsim/QuestaSim HDL Simulation Tools (Mentor/Siemens)
  - Design Vision Synthesis Tools (Synopsys)
- Tutorials will be available for all tools
  - Modelsim tutorials next week, ***(only attend one of these)***
    - Monday 6:45 – 8:00 held online MS Teams
    - Tuesday 5:30 – 6:45 held in B555 *(in liu of Harish's office hours)*
    - Tuesday 6:45 – 8:00 held in B555 *(in liu of discussion)*
  - Design Vision tutorial a 6 to 7 weeks later
  - Tool knowledge will be required to complete homeworks
  - Harish will be a resource for help on tools

11

11

Semester Specific Video (TA, office hours times, tutorial times,…) ends here.

12

12

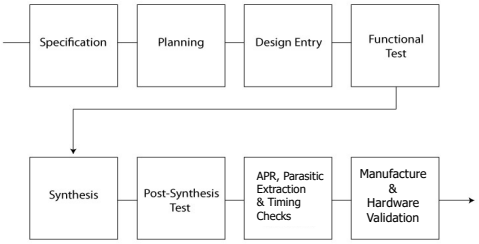
What You Should Get From This Class

- This class will teach you how to use Hardware Description Languages (HDLs) to design, verify, and realize digital logic
- You will learn how to synthesize HDLs into hardware using the same tools used in industry
- You will participate in the always enlightening process of working to design a digital system in a team environment
- By the end of this course, most of you should be qualified for an entry-level job or internship at an IC design firm (AMD, ARM, Intel, Micron, Nvidia, Qualcomm, …)

13

13

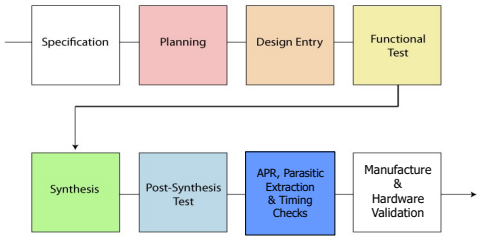
Digital Design Process



14

14

Description Phase



Implementation Phase

15

What is an HDL?

- This is not an HDL class...this is a digital design class.
- HDL = **H**ardware **D**escription **L**anguage
  - Allows for modeling & simulation (with timing) of digital designs
  - Can be synthesized into hardware (netlist) by synthesis tools (Synopsys, Ambit, FPGA compilers)
  - Two major standards in industry & academia
    - ✓ Verilog/Sverilog (Flexible, loose, more common in industry)
    - ✓ VHDL (Strongly typed, more common in defense and automotive)
    - ✓ Having used both I prefer Verilog. This course will use Verilog

16

16

What is an HDL? (continued)

```
module counter(clk,rst_n,cnt);
 input clk,rst_n;
 output [3:0] cnt;
 reg [3:0] cnt;
 always @(posedge clk) begin
  if (!rst_n)
    cnt <= 4'b0000;
  else
    cnt <= cnt+1;
 end
endmodule
```

- It looks like a programming language
- It is **NOT** a programming language
  - ✓ It is always critical to recall you are describing hardware
  - ✓ This codes primary purpose is to generate hardware
  - ✓ The hardware this code describes (a counter) can be simulated on a computer. In this secondary use of the language it does act more like a programming language.

17

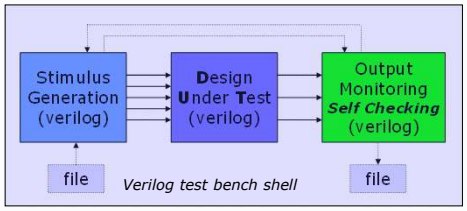
17

Simulating/Validating HDL

- The sad truth…
  - 10% design, 90% validation
  - If you do it right you will spend 9X more time testing/validating a design than designing it.

Testbenchs are written in verilog as well.

Testbench verilog is not describing hardware and can be thought of as more of a program.



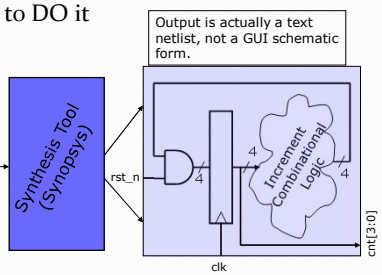
18

18

What is Synthesis?

- Takes a description of what a circuit DOES
- Creates the hardware to DO it

```
module counter(clk,rst_n,cnt);
  input clk,rst_n;
  output [3:0] cnt;
  reg [3:0] cnt;
  always @(posedge clk) begin
    if (~rst_n)
      cnt <= 4'b0000;
    else
      cnt <= cnt+1;
  end
endmodule
```



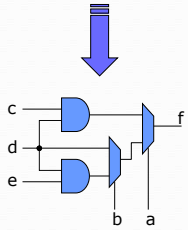
19

19

Synthesizing the Hardware Described

- All hardware created during synthesis
  - Even if **a** is true, still computing **d&e**
- Learn to understand how descriptions translated to hardware

```
if (a) f = c & d;
else if (b) f = d;
else f = d & e;
```



20

20

Why Use an HDL?

- Enables Larger Designs
  - More abstracted than schematics, allows larger designs.
    - Register Transfer Level Description
    - Wide datapaths (16, 32, or 64 bits wide) can be abstracted to a single vector
    - Synthesis tool does the bulk of the tedious repetitive work vs schematic capture
  - Work at transistor/gate level for large designs: cumbersome
- Portable Design
  - Behavioral or dataflow Verilog can be synthesized to a new process library with little effort (i.e. move from 45nm to 22nm process)

21

21

Why Use an HDL? (continued)

- Portable Design (continued)
  - Verilog written in ASCII text. The ultimate in portability. Much more portable than the binary files of a GUI schematic capture tool.
- Explore larger solution space
  - Synthesis options can help optimize (power, area, speed)
  - Synthesis options and coding styles can help examine tradeoffs
    - Speed
    - Power
    - area

22

22

Why Use an HDL? (continued)

- Better Validated Designs
  - Verilog itself is used to create the testbench
    - ✓ Flexible method that allows self checking tests
    - ✓ Unified environment
  - Synthesis tools are very good from the boolean correctness point of view
    - ✓ If you have a logic error in your final design there is a 99.999% chance that error exists in your behavioral code
    - ✓ Errors caused in synthesis fall in the following categories
      - ❑ Timing
      - ❑ Bad Library definitions
      - ❑ Bad coding style…sloppyness

23

23

Other Important HDL Features

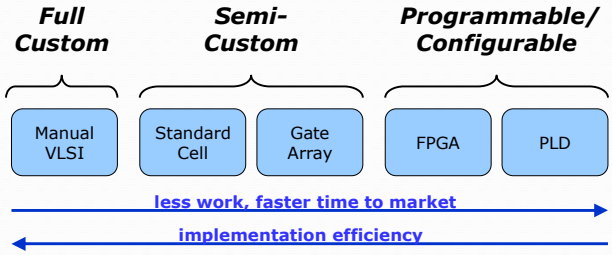
- Are highly portable (text)
- Are self-documenting (when commented well)
- Describe multiple levels of abstraction
- Represent parallelism
- Provides many descriptive styles
  - Structural
  - Register Transfer Level (RTL) or dataflow
  - Behavioral
- Serve as input for synthesis tools

24

24

Hardware Implementations

- HDLs can be compiled to semi-custom and programmable hardware implementations

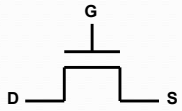


25

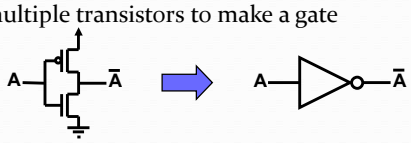
25

Hardware Building Blocks

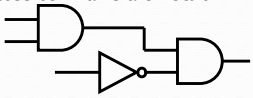
- Transistors are switches



- Use multiple transistors to make a gate



- Use multiple gates to make a circuit

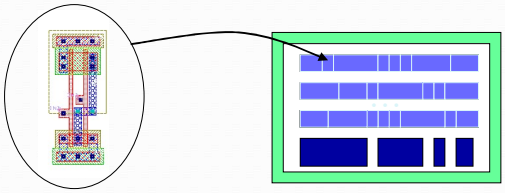


26

26

Standard Cells

- Library of common gates and structures (cells)
- Decompose hardware in terms of these cells
- Arrange the cells on the chip
- Connect them using metal wiring

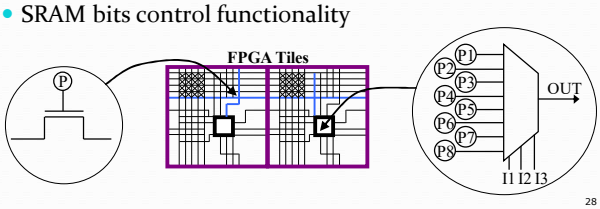


27

27

FPGAs

- "Programmable" hardware
- Use small memories as truth tables of functions
- Decompose circuit into these blocks
- Connect using programmable routing
- SRAM bits control functionality



28

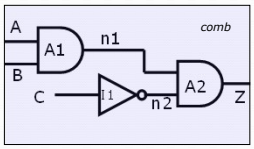
28

What is a Netlist?

- A netlist is a ASCII text representation of the interconnect of a schematic
- Many Standards Exist:
  - Spice Netlist
  - EDIF (Electronic Data Interchange Format)
  - Structural Verilog Netlist (this is what we will use)

```
module comb(Z,A,B,C);

  input A,B,C;
  output Z;
  wire n1, n2;

  and02d1 A1(n1,A,B);
  inv01d1  I1(n2,C);
  and02d2 A2(Z,n1,n2);

endmodule
```

=



29

29

Intro video ends here

30

30

Can Coding Affect Synthesis?

- Since HDLs try to abstract hardware design, do we even have to consider the hardware?
  - Good hardware design requires ability to analyze a problem to find simplifications
  - Multiple variables: throughput, area, latency, power
  - Finding an optimal hardware implementation is a computationally complex problem. The synthesis tools need guidance on where to start
- Optimization issues
  - `if (x != 0)` vs. `if((x <= -1) || (x >= 1))`
  - What hardware might this generate?

31

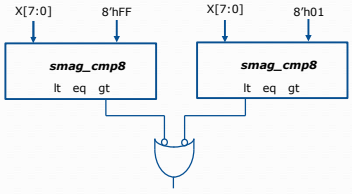
31

Can Coding Affect Synthesis?

if (x != 0) => If any bit of x is 1 then it is not zero.

Assume x[7:0] is 8-bit vector

if ((x <= -1) || (x>=1))

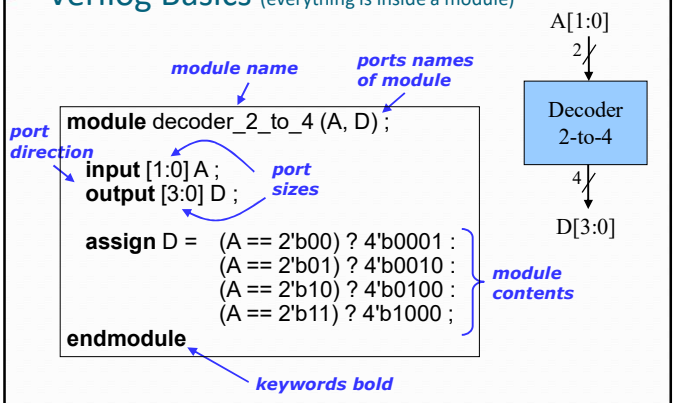


Yes…the synthesis tool is at least as smart as a 352 student and realizes that constants feeding into a signed magnitude comparator will simplify, but this is its starting point for optimizations.

32

32

Verilog Basics (everything is inside a module)



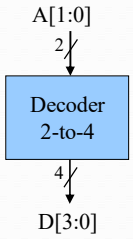

```
module decoder_2_to_4 (A, D) ;
   input [1:0] A ;
   output [3:0] D ;

   assign D =   (A == 2'b00) ? 4'b0001 :
                (A == 2'b01) ? 4'b0010 :
                (A == 2'b10) ? 4'b0100 :
                (A == 2'b11) ? 4'b1000 ;
endmodule
```

33

33

Verilog Basics (alternate port declaration style)

- Port details (width & direction) described inline with declaration in module




```
module decoder_2_to_4 (
   input [1:0] A,        // input to decode
   output [3:0] D        // one hot output
) ;

   assign D =   (A == 2'b00) ? 4'b0001 :
                (A == 2'b01) ? 4'b0010 :
                (A == 2'b10) ? 4'b0100 :
                (A == 2'b11) ? 4'b1000 ;
endmodule
```

34

34

Good HDL "Self Comments"

```
module xyz123 (A, B) ;
        input [3:0] A ;
        output B ;

        //module contents
endmodule
```

BAD

```
module doomsday_machine (
        input [3:0] crystals,
        output earthquake
);

        //module contents
endmodule
```

GOOD

35 35

35

Verilog Basics (declaring signal ports)

- Declare direction of port
  - **input**
  - **output**
  - **inout** (bidirectional)
- Scalar (single bit) - don't specify a size
  - **input** cin;
- Vector (multiple bits) - specify size using range (before signal)
  - Range is MSB to LSB (left to right)
  - Don't have to include zero if you don't want to... (**D[2:1]**)
  - **output** [7:0] OUT;
  - **input** [0:4] IN;

36

36

Verilog Basics (common mistake)

```
module mux4to1(
  input sel[1:0],
  input d_in[3:0],
  output d_out
);

  assign d_out = (sel==2'b00) ? d_in[0] :
                 (sel==2'b01) ? d_in[1] :
                 (sel==2'b10) ? d_in[2] :
                 d_in[3];

endmodule
```

What's wrong with this?

This is actually an array of 2 1-bit wide signals, not a single 2-bit wide signal.

Proper declaration is:
`input [1:0] sel;`

However, this part is correct. Bit selection of a vector does occur after the signal name.

Yes…that is kinda messed up and inconsistent.

37

Verilog Basics (declaring internal signals)

```
module mult_accum(
  input [7:0] A_op,B_op,
  input [15:0] accum,
  output [15:0] result
);

  wire [15:0] product;

  /// First multiply to form product ///
  assign product = A_op * B_op;      // infer HW multiplier

  /// Then accumulate to form result ///
  assign result = accum + product;

endmodule
```

Study this multiply accumulate module.

It has an internal signal (**product**) involved in the calculation.

This 16-bit vector had to be declared.

This is a purely combinational implementation (no flops). Product comes from a hardware multiplier (bunch of combinational logic).

It is declared as type `wire`. This is a net type used for anything driven by combinational logic (structural or dataflow verilog).

All our module inputs/outputs default to type wire.

38

Verilog Basics (declaring internal signals)

```
module mult_accum(
  input clk,
  input [7:0] A_op,B_op,
  input [15:0] accum,
  output [15:0] result
);

  reg [15:0] product;

  /// First multiply & flop to form product ///
  always @(posedge clk)
    product <= A_op * B_op;      // infer HW multiplier

  /// Then accumulate to form result ///
  assign result = accum + product;

endmodule
```

This version of ***mult_accum*** flops the result of the multiply before accumulating.

So now **product** represents the output of flops.

It has now been declared as type `reg`

Any signal assigned to in an `always` block had to be declared as type `reg`
(in old school verilog)

So…if we are inferring a flop use type `reg`, and if we are inferring combinational use type `wire`…OK, good I got it! *(not quite so clean cut)*

39

Verilog Basics (wire vs reg vs logic)

```
module mult_accum(
  input clk,
  input [7:0] A_op,B_op,
  input [15:0] accum,
  output logic [15:0] result
);

  logic [15:0] product;

  /// First multiply & flop to form product ///
  always @(posedge clk)
    product <= A_op * B_op;      // infer HW multiplier

  /// Then accumulate to form result ///
  always @(posedge clk)
    result <= accum + product;

endmodule
```

In general just default to declaring signals as type logic

System verilog introduced the type `logic` which is a superset of `reg` and `wire`.

Type `logic` can be used for combinational or flops, You should still know what you are making, but use of type `logic` can mitigate annoying errors.

Any signal assigned to in an `always` block had to be declared as type `reg` or `logic`

This module now flops the result too, so **result** had to be declared as type `logic` *(port inputs/outputs default to type `wire` if not specified)*

40

Module Coding Styles

- Modules can be specified different ways
  - Structural – connect primitives and modules
  - Dataflow– use continuous assignments
  - Behavioral – use initial and always blocks
- A single module can use more than one method!
- Dataflow & Behavioral will be covered in following lectures. Structural will be reviewed here. *(I say reviewed because you did some of this in 352).*

41

41

Structural

- A schematic in text form (i.e. A netlist)
- Build up a circuit from gates/flip-flops
  - Gates are primitives (part of the language)
  - Flip-flops themselves described behaviorally
- Structural design
  - Create module interface
  - Instantiate the gates in the circuit
  - Declare the internal wires needed to connect gates
  - Put the names of the wires in the correct port locations of the gates
    - For primitives, outputs always come first

42

42

Primitives

- No declarations - can only be instantiated
- Output port appears before input ports
- Optionally specify: instance name and/or delay (discuss delay later)

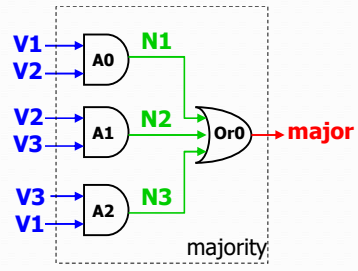
```
and N25 (Z, A, B, C);     // name specified
and #10 (Z, A, B, X),
        (X, C, D, E);     // delay specified, 2 gates
and #10 N30 (Z, A, B); // name and delay specified
```

43

43

Structural Example

```
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

wire N1, N2, N3;

and A0 (N1, V1, V2),
    A1 (N2, V2, V3),
    A2 (N3, V3, V1);

or  Or0(major, N1, N2, N3);

endmodule
```



44

44

Structural Example (continued)

```
module full_add (A, B, Ci, S, Co) ;

 input A, B, Ci ;
 output S, Co ;

 // Sum is XOR of inputs
 xor iSUM (S, A, B, Ci);

 // Co is the majority of inputs
 majority iCO(.V1(A), .V2(B), .V3(Ci),
              .major(Co));

endmodule
```

Built in primitives like and, or, nand, nor, xor assume output signal first followed by inputs.

Gates can be any width.

Instance name is optional for primitives but recommended

Instantiating another module is a form of structural verilog. Module name of block you are instantiating comes first.

When it is a unit you defined the instance name is not optional

Shown here is "named" connection. There is a "connect by reference order"…don't use it.

45

45

Structural Example (vectored instantiation)

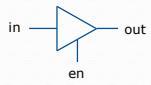
```
module RCA8(
  input      [7:0]   A,B,     // two 8-bit vectors to be added
  input              Cin,     // An optional carry in bit
  output     [7:0]   S,       // 8-bit Sum
  output             Cout     // and carry out
);

    ////////////////////////////////////////////////////////
    // Co of lesser significant form Cin of more significant //
    ////////////////////////////////////////////////////////
    logic [7:0] Carries;

    ////////////////////////////////////////////////
    // Implement Full Adder as structural verilog //
    ////////////////////////////////////////////////
    full_add iFA[7:0](.A(A), .B(B), .Cin({Carries[6:0],Cin}),
                      .S(S), .Cout(Carries));

    buf iBF(Cout,Carries[7]);

endmodule
```

8 instances at once

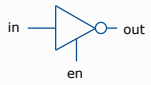
46

46

Structural: Tri-States



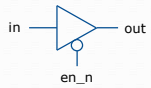
bufif1 iTRI(out,in,en);



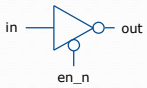
notif1 iTRINOT(out,in,en);

notif1 #1 iTRINOT(out,in,en);

(with delay of 1 time unit)



bufif0 iTRI(out,in,en);



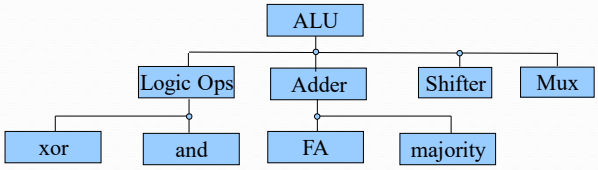
notif0 iTRI(out,in,en);

47

47

Structural: Build up Hierarchy

- Build up a module from smaller pieces
  - Other modules (which may contain other modules)
- Architecture: typically top-down
- Design & Verification: typically bottom-up



48

48

Structural: Proper Connections with Instantiation

- Positional or Connect by reference
  - Can be okay in some situations
    - Designs with very few ports
    - Interchangeable input ports (and/or/xor gate inputs)
  - Gets confusing for large #s of ports

```
module dec_2to4 (A, E_n, D);

 input [1:0] A;
 input E_n;
 output [3:0] D;
   .
   .
   .
endmodule
```

Partial code instatiating decoder

```
wire [1:0] X;
wire W_n;
wire [3:0] word;

// instantiate decoder
dec_2to4 iDEC (X,  W_n,  word);
```

X needs to be in **A** position

W_n in the **E_n** position

word in the **D** position

Structural: Proper Connections with Instantiation

- Explicit or Connect by name method
  - Helps avoid "misconnections"
  - Don't have to remember port order
  - Can be easier to read
  - `.<port name>(<signal name>)`

```
module dec_2to4 (A, E_n, D);

 input [1:0] A;
 input E_n;
 output [3:0] D;
   .
   .
   .
endmodule
```

Partial code instatiating decoder

```
wire [1:0] X;
wire W_n;
wire [3:0] word;

// instantiate decoder
dec_2to4 iDEC(.A(X), .E_n(W_n), .D(word));
```

NOTE: the named connections, order does not matter!

Imagine matching the order for a module with 20+ signal connections. You will see this for the project.

Empty Port Connections

- Example: `module dec_2to4(A, E_n, D);`
  - `dec_2to4 iDEC(.A(X),  .D(word));` // E_n is high impedence
  - `dec_2to4 iDEC(.A(X), .E_n(W_n), );` // Outputs D[3:0] unused.
- General rules
  - Empty input ports => high impedance state (z)
  - Empty output ports => output not used
- Specify all input ports anyway!
  - Z as an input is very bad...why?
- Helps if no connection to output port name but leave empty:
  - `dec_2_4_en DX(.A(X[3:2]), .E_n(W_n), .D());`

Why Know Structural Verilog?

- Code you write to be synthesized will almost all be dataflow or behavioral
- You will write your test bench primarily in behavioral
- What needs structural Verilog?
  - Building hierarchy (instantiating blocks to form higher level functional blocks)
  - Synthesis tools output structural verilog (gate level netlist). You need to be able to read this output.

End of video on verilog basics & structural verilog

53

53

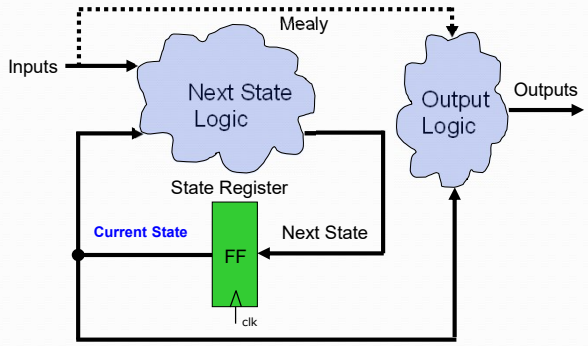
FSM Review

- Combinational and sequential logic
- Often used to generate control signals
- Reacts to inputs (including clock signal)
- Can perform multi-cycle operations
- Examples of FSMs
  - Counter
  - Vending machine
  - Traffic light controller
  - Bus Controller
  - Control unit of serial protocol (like RS232, I2C or SPI)

54

54

Mealy/Moore FSMs



55

55

FSMs

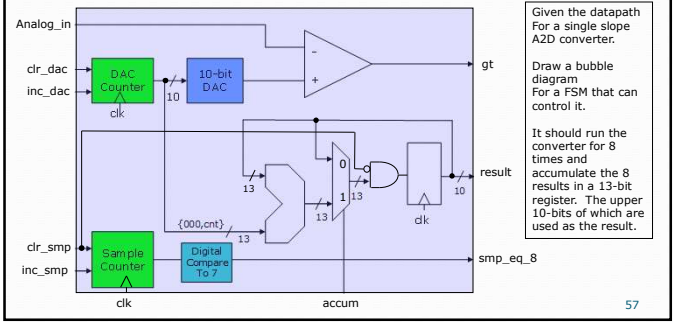
- Moore
  - Output depends only on current state
  - Outputs are synchronous (but not necessarily glitch free)
- Mealy
  - Output depends on current state and inputs
  - Outputs can be asynchronous
    - ✓Change with changes on the inputs
  - Outputs can be synchronous
    - ✓Register the outputs
    - ✓Outputs delayed by one cycle

56

56

Remember Bubble Diagrams?

- They can be useful. I sometimes will draw a bubble diagram first for a complex FSM. Then code it.



Given the datapath For a single slope A2D converter.

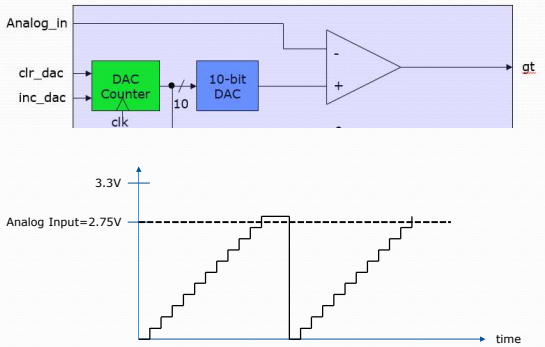
Draw a bubble diagram For a FSM that can control it.

It should run the converter for 8 times and accumulate the 8 results in a 13-bit register. The upper 10-bits of which are used as the result.

57

57

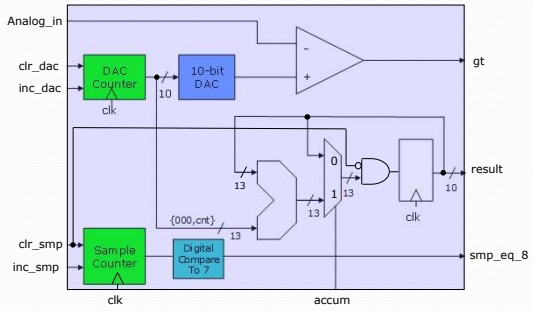
Single Slope A2D Converter:



58

58

Datapath Used to Average:

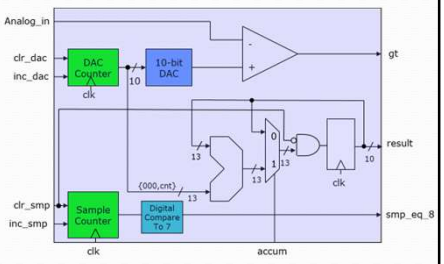


59

59

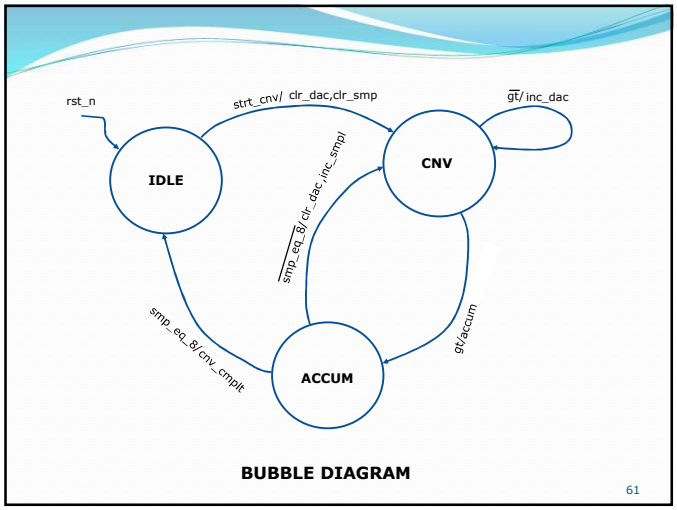
SM Inputs & Outputs

Inputs:	Outputs:
gt	clr_dac
smp_eq_8	inc_dac
strt_cnv	clr_smp
clk	inc_smp
rst_n	accum
	cnv_cmplt



60

60


rst_n
strt_cnv/ clr_dac,clr_smp
gt/ inc_dac
IDLE
CNV
smp_eq_8/ clr_dac,inc_smpl
smp_eq_8/ cnv_cmplt
gt/accum
ACCUM
BUBBLE DIAGRAM
61

61