

Digital Logic Circuit Design Using Verilog HDL

TAEJOON PARK

Department of Robotics Engineering



Verilog HDL

Introduction to Verilog HDL

What You Should Already Know

- **Principles of basic digital logic circuit design**
 - Number representations
 - Boolean algebra
 - Gate-level design
 - K-Map minimization
 - Sequential logic design
 - Basic arithmetic structures
 - ...

Overview of HDLs

- **Hardware description languages (HDLs)**
 - Are computer-based hardware programming languages
 - Allow modeling and simulating the functional behavior and timing of digital hardware
 - Synthesis tools take an HDL description and generate a technology-specific netlist
- **Two main HDLs used by industry**
 - **Verilog HDL** (C-based, industry-driven)
 - VHSIC HDL or **VHDL** (Ada-based, defense/industry/university-driven).

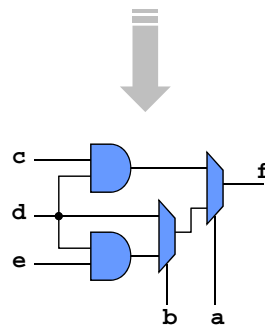
Synthesis of HDLs

- Takes a **description** of what a circuit **DOES**
- Creates the hardware to **DO** it
- HDLs may **LOOK** like software, but they're **NOT!**
 - NOT a program
 - Doesn't "run" on anything
 - Though we do **simulate** them on computers
 - Don't confuse them!
- Also **use HDLs to test the hardware** you create
 - This is more like software

Describing Hardware!

- **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;
```



Why Use an HDL?

- **More and more transistors can fit on a chip**
 - Allows larger designs!
 - Work at transistor/gate level for large designs: hard
 - Many designs need to go to production quickly
- **Abstract large hardware designs!**
 - Describe what you need the hardware to do
 - Tools then design the hardware for you
- **BIG CAVEAT**
 - Good descriptions → Good hardware
 - Bad descriptions → BAD hardware!

Why Use an HDL?

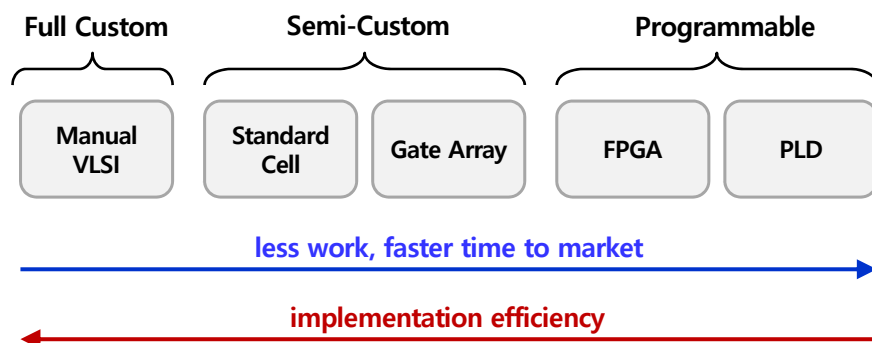
- **Simplified & faster design process**
- **Explore larger solution space**
 - Smaller, faster, lower power
 - Throughput vs. latency
 - Examine more design tradeoffs
- **Lessen the time spent debugging the design**
 - Design errors still possible, but in fewer places
 - Generally easier to find and fix
- **Can reuse design to target different technologies**
 - Don't manually change all transistors for rule change

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)
 - Behavioral
- Serve as input for synthesis tools

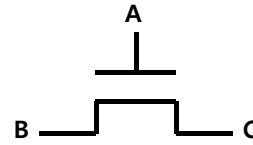
Hardware Implementations

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



Hardware Building Blocks

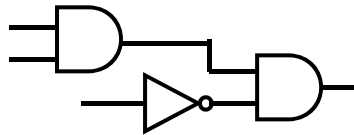
- Transistors are switches



- Use multiple transistors to make a gate

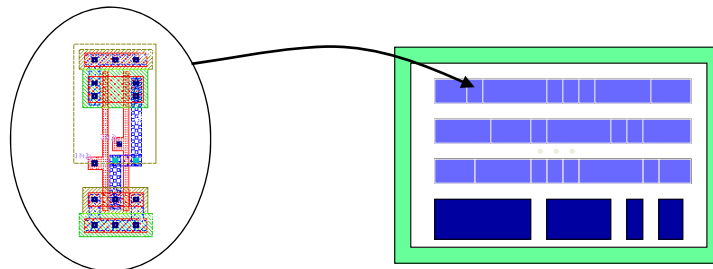


- Use multiple gates to make a circuit



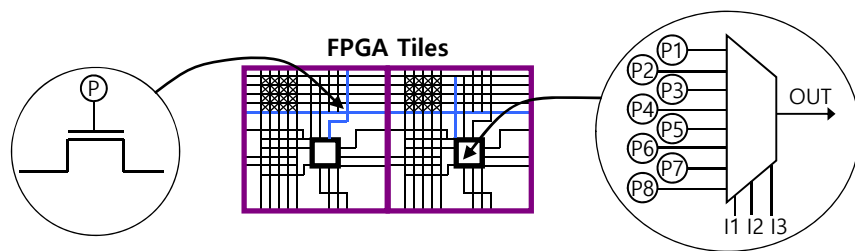
Standard Cells

- 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



FPGAs

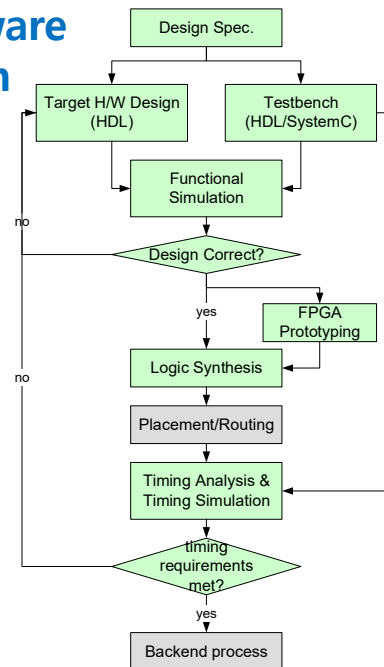
- **Programmable hardware**
- Use small memories as truth tables of functions
- Decompose circuit into these blocks
- Connect using programmable routing
- SRAM bits control functionality



디지털논리회로실험 : 2017-2

13

Hardware Design Flow



Hardware Design Flow

Thorough Understanding of
 - Multimedia & Networking algorithm
 - ARM RISC processor architecture
 - ARM programming

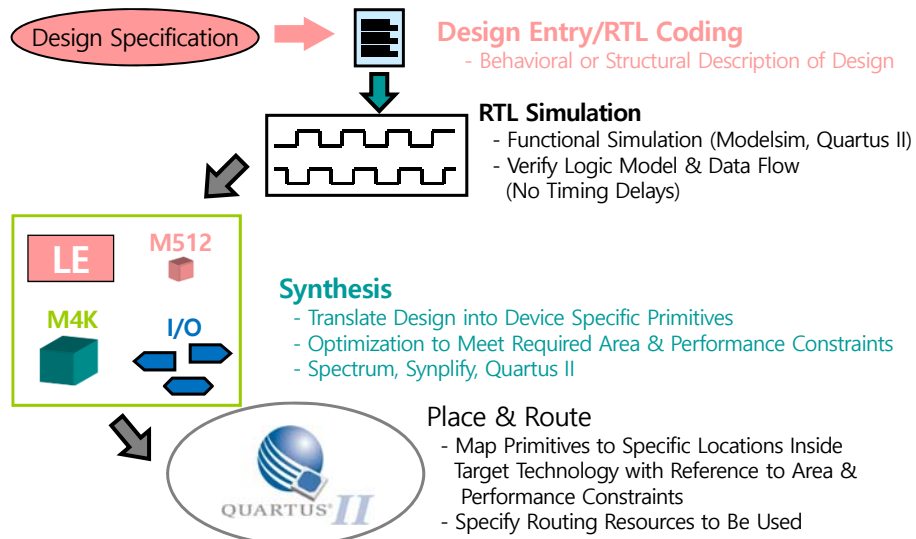
HDL Simulator	Cadence NC-Verilog/VHDL Verilog-XL MentorGraphics ModelSim
FPGA Tools & Devices	Quartus II / Synplify Xilinx Virtex-II XCV2000 Altera Cyclone II
Synthesis Tool	Synopsys Design Compiler
Timing Analysis Tool	Synopsys PrimeTime

Knowledge & CAD Tools for Hardware Design

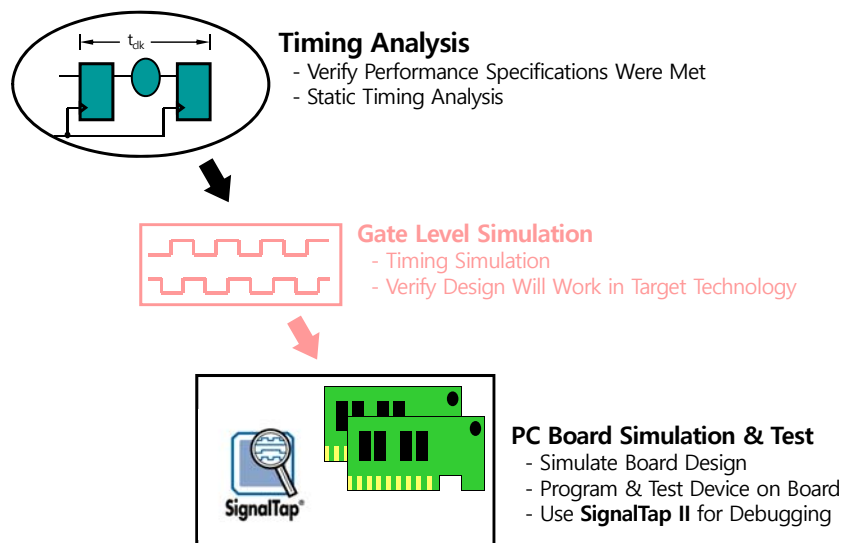
디지털논리회로실험 : 2017-2

14

FPGA Design Flow (1/2)



FPGA Design Flow (2/2)



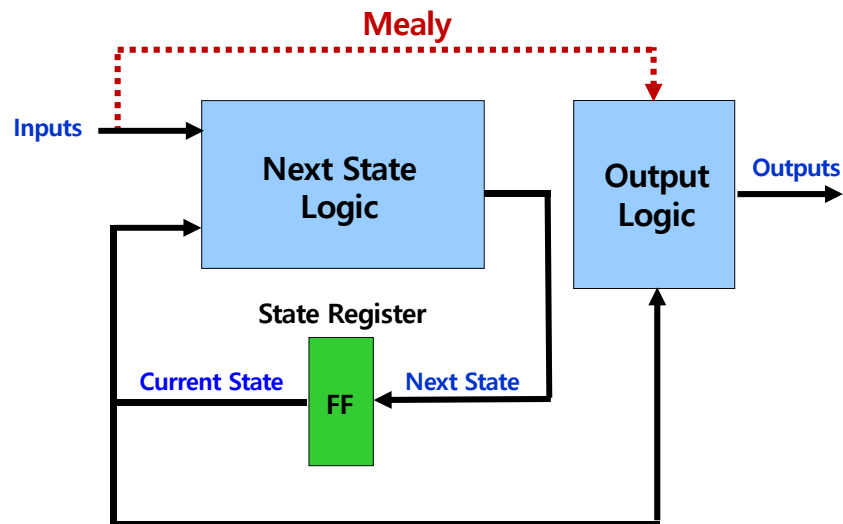
Boolean Algebra and K-maps

- We're abstracting hardware design ...
- Why do you need to understand hardware?
- Good hardware design requires ability to analyze a problem to find simplifications
 - Which may involve boolean equations, K-maps
- Why bother simplifying?
 - Easier to design/debug, speed up synthesis
 - Can have smaller/faster resulting hardware
 - Synthesis tool only knows what you tell it

FSM

- 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
 - Phone dialing

Mealy/Moore FSMs



FSMs

- **Moore**
 - Output depends only on **current state**
 - Outputs are synchronous
- **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

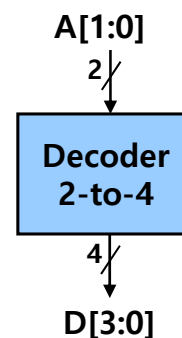
Verilog HDL

- In this class, we will use the **Verilog HDL**
 - Used in academia and industry
- VHDL is another common HDL
 - Also used by both academia and industry
- Many principles we will discuss apply to **any** HDL
- Once you can **“think hardware”**, you should be able to use any HDL fairly quickly

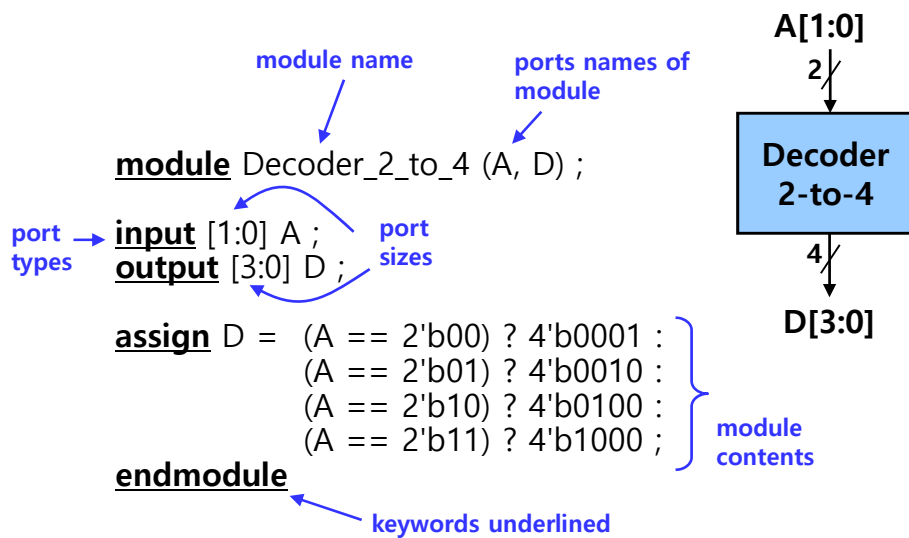
Verilog Module

- In Verilog, a circuit is 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
```



Verilog Module



Declaring A Module

- Can't use keywords as module/port/signal names
 - Choose a **descriptive** module name
- Indicate the ports (connectivity)
- Declare the signals connected to the ports
 - Choose **descriptive** signal names
- Declare any internal signals
- Write the internals of the module (functionality)

Declaring Ports

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

Module Styles

- Modules can be specified different ways
 - **Structural** : connect primitives and modules
 - **RTL** : use continuous assignments
 - **Behavioral** : use initial and always blocks
- A single module can use more than one method!
- **What are the differences?**

Structural

- **A schematic in text form**
- **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

Structural Example

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

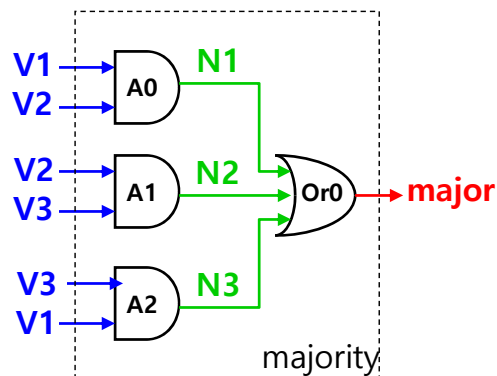
```
output major;  
input V1, V2, V3;
```

```
wire N1, N2, N3;
```

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

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

```
endmodule
```



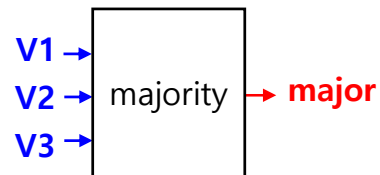
RTL Example

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

```
output major ;  
input V1, V2, V3 ;
```

```
assign major = V1 & V2  
              | V2 & V3  
              | V1 & V3;
```

```
endmodule
```



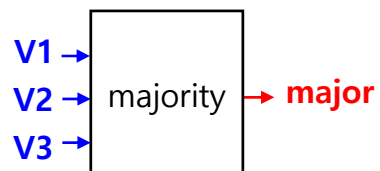
Behavioral Example

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

```
output reg major ;  
input V1, V2, V3 ;
```

```
always @(V1, V2, V3) begin  
    if (V1 && V2 || V2 && V3 || V1 && V3) major = 1;  
    else major = 0;  
end
```

```
endmodule
```



Full Adder : Structural Design

```
module half_add (X, Y, S, C);
```

```
input X, Y ;  
output S, C ;
```

```
xor SUM (S, X, Y);  
and CARRY (C, X, Y);
```

```
endmodule
```

```
module full_add (A, B, CI, S, CO) ;
```

```
input A, B, CI ;  
output S, CO ;
```

```
wire S1, C1, C2;
```

```
// build full adder from 2 half-  
// adders  
half_add PARTSUM (A, B, S1, C1),  
                SUM (S1, CI, S, C2);
```

```
// ... and an OR gate for the carry  
or CARRY (CO, C2, C1);
```

```
endmodule
```

Full Adder : RTL Design

```
module fa_rtl (A, B, CI, S, CO) ;
```

```
input A, B, CI ;  
output S, CO ;
```

```
// use continuous assignments
```

```
assign S = A ^ B ^ CI;  
assign CO = (A & B) | (A & CI) | (B & CI);
```

```
endmodule
```


Full Adder : Behavioral Design

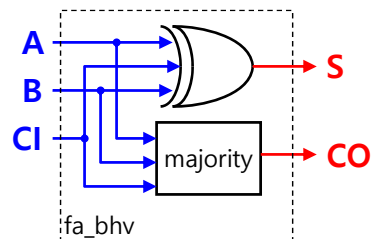
- Circuit “reacts” to given events (for simulation)
 - Actually list of signal changes that affect output

```
module fa_bhv (A, B, CI, S, CO) ;  
  
input A, B, CI;  
output S, CO;  
reg S, CO;           // explained in later lecture – “holds” values  
  
// use procedural assignments  
always@(A or B or CI)  
begin  
    S = A ^ B ^ CI;  
    CO = (A & B) | (A & CI) | (B & CI);  
end  
endmodule
```

Full Adder : Behavioral Design

- **In Simulation**
 - When A, B, or C change, S and CO are recalculated
- **In Reality**
 - **Combinational logic** – no “waiting” for the trigger
 - **Constantly computing** - think transistors and gates!
 - **Same hardware** created for behavioral and RTL examples

```
always@(A or B or CI)  
begin  
    S = A ^ B ^ CI;  
    CO = (A & B) | (A & CI) | (B & CI);  
end
```





Verilog HDL

Structural Verilog

Structural Basics

- **Build design up** from the gate/flip-flop/latch level
 - Flip-flops actually constructed using Behavioral
- Verilog provides a set of **gate primitives**
 - **and, nand, or, nor, xor, xnor, not, buf, bufif1**, etc.
 - Combinational building blocks for structural design
 - Known "behavior"
 - Cannot access "inside" description
- Can also model at the transistor level
 - Most people don't, we won't

Primitives

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

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

Syntax For Structural Verilog

- **First, declare the interface to the module**
 - Module keyword, module name
 - Port names/types/sizes
- **Next, declare any internal wires using “wire”**
 - wire [3:0] partialsum;
- **Then, instantiate the primitives/submodules**
 - Indicate which signal is on which port

Four-Value Logic

- **A single bit can have one of FOUR values**
 - 0 Numeric 0, logical FALSE
 - 1 Numeric 1, logical TRUE
 - x Unknown or ambiguous value
 - z No value (high impedance)
- **Why x?**
 - Could be a conflict, could be lack of initialization
- **Why z?**
 - Nothing driving the signal
 - Tri-states

Representing Numbers

- **Representation: `<size>'<base><number>`**
 - size number of **BITS** (regardless of base used)
 - base decimal, hex, octal, ...
 - number the actual value in the given base
- **Can use different bases**
 - Decimal (d or D) – default if no base specified!
 - Hex (h or H)
 - Octal (o or O)
 - Binary (b or B)
- **Size defaults to at least 32...**
 - You should specify the size explicitly!
 - Why create 32-bit register if you only need 5 bits?
 - May cause compilation errors on some compilers

Number Examples

Number	Decimal Equivalent	Actual Binary
4'd3	3	0011
8'ha	10	00001010
8'o26	22	00010110
5'b111	7	00111
8'b0101_1101	93	01011101
8'bx1101	-	xxx1101
-8'd6	-6	11111010

Numbers with MSB of **x** or **z** extended with that value

Datatypes

- Two categories
 - **Nets**
 - **Registers**
- **Only** dealing with **nets in Structural Verilog**
- Register datatype doesn't actually imply an actual register...
- Will discuss registers in Behavioral Verilog

Net Types

- **Wire (Default value for all signals)**
 - Most common, establishes connections
- Tri
 - Indicates it will be output of a tri-state
 - Basically same as "wire"
- supply0, supply1: ground & power connections
 - Can imply this by saying "0" or "1" instead
 - `xor xorgate(out, a, 1'b1);`
- wand, wor, triand, trior, tri0, tri1, trireg
 - Perform some signal resolution or logical operation
 - Not used in this course

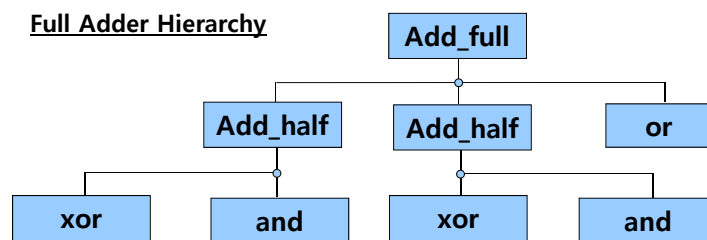
Hierarchy

- Any Verilog design you do will be a module
- This includes testbenches!
- Interface ("black box" representation)
 - Module name, ports
- Definition
 - Describe functionality of the block
 - Includes interface
- Instantiation
 - Use the module inside another module

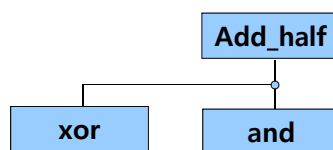
Hierarchy

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

Full Adder Hierarchy

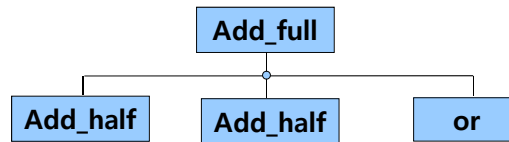


Add_half Module



```
module Add_half(c_out, sum, a, b);  
  output sum, c_out;  
  input a, b;  
  
  xor sum_bit(sum, a, b);  
  and carry_bit(c_out, a, b);  
endmodule
```

Add_full Module



```

module Add_full(c_out, sum, a, b, c_in) ;
    output sum, c_out;
    input a, b, c_in;
    wire w1, w2, w3;

    Add_half AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
    Add_half AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));
    or carry_bit(c_out, w2, w3);
endmodule
    
```

Can Mix Styles In Hierarchy!

```

module Add_half_bhv(c_out, sum, a, b);
    output reg sum, c_out;
    input a, b;
    always @(a, b) begin
        sum = a ^ b;
        c_out = a & b;
    end
endmodule
    
```

```

module Add_full_mix(c_out, sum, a, b, c_in) ;
    output sum, c_out;
    input a, b, c_in;
    wire w1, w2, w3;
    Add_half_bhv AH1(.sum(w1), .c_out(w2),
        .a(a), .b(b));
    Add_half_bhv AH2(.sum(sum), .c_out(w3),
        .a(c_in), .b(w1));
    assign c_out = w2 | w3;
endmodule
    
```


Port Connections

- **Positional or Implicit port connections**
 - Used for primitives (first port is output, others inputs)
 - Can be OK in some situations
 - Designs with very few ports
 - Interchangeable input ports (and/or/xor gate inputs)
 - Gets confusing for large #s of ports
- **Port connections by name**
 - Helps **avoid misconnections**
 - Don't have to remember port order
 - Can be easier to read
 - **.<port name>(<signal name>)**

Port Connections Example

- Variables – defined in upper level module
 - **wire** [3:2] X; **wire** W_n; **wire** [3:0] word;
- By position
 - **module** dec_2_4_en (A, E_n, D);
 - dec_2_4_en DX (X[3:2], W_n, word);
- By name
 - **module** dec_2_4_en (A, E_n, D);
 - dec_2_4_en DX (.E_n(W_n), .A(X[3:2]), .D(word));
- In both cases,
 - A = X[3:2], E_n = W_n, D = word

Module Port List

- Multiple ways to declare the ports of a module

```
module Add_half(c_out, sum, a, b);  
    output sum, c_out;  
    input a, b;  
    ...  
endmodule
```

```
module Add_half(output c_out, sum,  
               input a, b);  
    ...  
endmodule
```

Structural Design Tip

- If a design is complex, draw a **block diagram**!
- **Label the signals connecting the blocks**
- **Label ports on blocks** if not primitives/obvious
- Easier to double-check your code!
- Don't bother with 300-gate design...
- But if it's that big, probably should use hierarchy!

Concatenating Vectors

- Can "build" vectors using smaller vectors and/or scalar values
- Use the { } operator
- Example

```
module concatenate(out, a, b, c, d);  
  input [2:0] a;  
  input [1:0] b, c;  
  input d;  
  output [9:0] out;  
  
  assign out = {a[1:0], b, c, d, a[2]};  
  
endmodule
```

becomes
8-bit vector:
 $a_1a_0b_1b_0c_1c_0da_2$

Concatenating Vectors

- Can "build" vectors using smaller vectors and/or scalar values
- Use the { } operator
- Example

```
module add_concatenate(out, a, b, c, d);  
  input [7:0] a;  
  input [4:0] b;  
  input [1:0] c;  
  input d;  
  output [7:0] out;  
  
  add8bit(sum(out), .cout(), .a(a), .b({b,c,d}), .cin());  
  
endmodule
```

becomes
8-bit vector:
 $b_4b_3b_2b_1b_0c_1c_0d$

Arrays of Instances

- Need several instances with similar connections?
- Can create an array of instances!
- Works for **both primitives and modules**
- Syntax:
 <type> [<delay>] <array name> [<range>] (<ports>);
- Example:
 wire [7:0] a, b, out;
 and #4 AND8 [7:0] (out, a, b);

Simple Array Example 1

- Make an array of instances with each instance having same connections

```
module array_of_xor (y, a, b);  
    input [3:0] a, b;  
    output [3:0] y;  
    xor X3 (y[3], a[3], b[3]);           // instantiates 4 xor gates  
    xor X2 (y[2], a[2], b[2]);  
    xor X1 (y[1], a[1], b[1]);  
    xor X0 (y[0], a[0], b[0]);  
endmodule
```

```
module array_of_xor (y, a, b);  
    input [3:0] a, b;  
    output [3:0] y;  
  
endmodule
```

Simple Array Example 2

- Make an array of instances with each instance having same connections

```
module array_of_flops (q, data_in, clk, set, rst);
  input [7:0] data_in; // one per flip-flop
  input clk, set, rst; // shared signals
  output [7:0] q; // one per flip-flop
  /* instantiate 8 flip-flops to form an 8-bit register */
  flip_flop R7(q[7], data_in[7], clk, set, rst);
  flip_flop R6(q[6], data_in[6], clk, set, rst);
  flip_flop R5(q[5], data_in[5], clk, set, rst);
  flip_flop R4(q[4], data_in[4], clk, set, rst);
  flip_flop R3(q[3], data_in[3], clk, set, rst);
  flip_flop R2(q[2], data_in[2], clk, set, rst);
  flip_flop R1(q[1], data_in[1], clk, set, rst);
  flip_flop R0(q[0], data_in[0], clk, set, rst);
endmodule
```

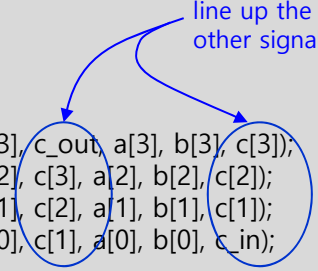
Complex Array Example

- Use an array to build a multi-bit adder
 - Caution: carry chain!

```
module 4_bit_adder(sum, c_out, a, b, c_in);
  input [3:0] a, b;
  input c_in;
  output [3:0] sum;
  output c_out;
  wire [3:1] c;

  Add_full M3(sum[3], c_out, a[3], b[3], c[3]);
  Add_full M2(sum[2], c[3], a[2], b[2], c[2]);
  Add_full M1(sum[1], c[2], a[1], b[1], c[1]);
  Add_full M0(sum[0], c[1], a[0], b[0], c_in);
endmodule
```

the carry signals don't
line up the way the
other signals do!

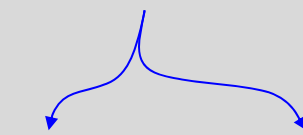


Complex Array Example

- Use an array to build a multi-bit adder
 - Caution: carry chain!

```
module 4_bit_adder(sum, c_out, a, b, c_in);  
  input [3:0] a, b;  
  input c_in;  
  output [3:0] sum;  
  output c_out;  
  wire [3:1] c;  
  
  Add_full M[3:0](sum, {c_out, c[3:1]}, a, b, {c[3:1], c_in});  
endmodule
```

use concatenation to form vectors!



Verilog HDL

RTL Verilog

RTL Basics

- **Higher-level of description than structural**
 - Don't always need to specify each individual gate
 - Can take advantage of **operators**
- **More hardware-explicit than behavioral**
 - Doesn't look as much like software
 - Frequently easier to understand what's happening
- Very easy to synthesize
 - Supported by even primitive synthesizers

Continuous Assignment

- **Implies structural hardware**
assign <LHS> = <RHS expression>;
- Example
wire out, a, b;
assign out = a & b;
- **If RHS result changes, LHS is updated with new value**
 - **Constantly** operating ("continuous"!)
 - It's **hardware!**
- Used to model **combinational logic and latches**

Full Adder : RTL Design

```
module fa_rtl (A, B, CI, S, CO) ;
```

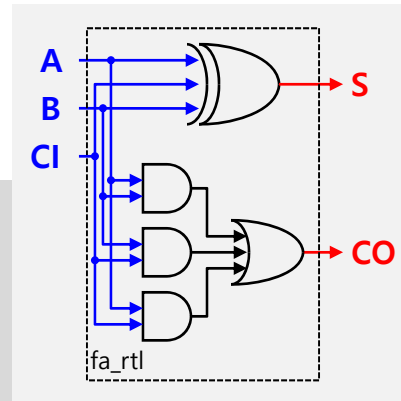
```
    input A, B, CI ;  
    output S, CO ;
```

```
    // use continuous assignments
```

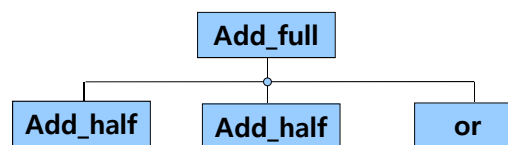
```
    assign S = A ^ B ^ CI;
```

```
    assign CO = (A & B) | (A & CI) | (B & CI);
```

```
endmodule
```



RTL And Structural Combined!



```
module Add_half(sum, cout, a, b);  
    output sum, cout;  
    input a, b;  
    assign sum = a ^ b;  
    assign cout = a & b;  
endmodule
```

```
module Add_full(c_out, sum, a, b, c_in) ;  
    output sum, c_out;  
    input a, b, c_in;  
    wire psum, c1, c2;  
    Add_half AH1(partsum, c1, a, b);  
    Add_half AH2(sum, c2, psum, c_in);  
    assign c_out = c1 | c2;  
endmodule
```

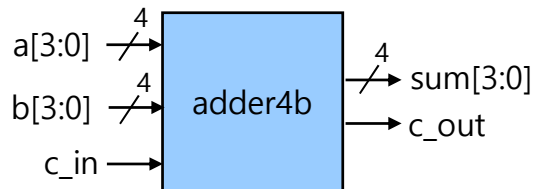

Continuous Assignment LHS

- Can assign values to:
 - Scalar nets
 - Vector nets
 - Single bits of vector nets
 - Part-selects of vector nets
 - Concatenation of any of the above
- Examples:
 - `assign out[7:4] = a[3:0] | b[7:4];`
 - `assign val[3] = c & d;`
 - `assign {a, b} = stimulus[15:0];`

Continuous Assignment RHS

- Use operators:
 - Arithmetic, Logical, Relational, Equality, Bitwise, Reduction, Shift, Concatenation, Replication, Conditional
 - Same set as used in Behavioral Verilog
- Can also be a pass-through!
 - `assign a = stimulus[16:9];`
 - `assign b = stimulus[8:1];`
 - `assign cin = stimulus[0];`
 - Note: "aliasing" is only in one direction
 - Cannot give 'a' a new value elsewhere to set stimulus[16:9]!

Example : adder4b



```
module adder4b (sum, c_out, a, b, c_in);
  input [3:0] a, b;
  input c_in;
  output [3:0] sum;
  output c_out;
  assign {c_out, sum} = a + b + c_in;
endmodule
```

Can Often Replace Primitive Arrays

- **Module/Instance Array**

```
module array_of_xor (output [3:0] y, input [3:0] a, b);
  xor Xarray [3:0] (y, a, b); // instantiates 4 xor gates
endmodule
```

- **Continuous Assign**

```
module xor_4bit (output [3:0] y, input [3:0] a, b);
  assign y = a ^ b; // instantiates 4 xor gates
endmodule
```

- **Can't replace array of FF's!**

- Why?
- No edge-triggering in continuous assignments!

Operators & Arithmetic

- **Much easier than structural!**

*	multiply	**	exponent
/	divide	%	modulus
+	add	-	subtract

- Some of these don't synthesis
- Also have unary operators +/- (pos/neg)
- Understand bit-size!
 - Can affect sign of result
 - Is affected by bit-width of BOTH sides
 $SUM[6:0] = a[3:0] + b[4:0]$

Example : Unsigned MAC Unit

- Design a multiply-accumulate (MAC) unit that computes
 $Z[7:0] = A[3:0] * B[3:0] + C[7:0]$
- It sets overflow to one, if the result cannot be represented using 8 bits.

```
module mac(output Z [7:0], output overflow,  
           input [3:0] A, B, input [7:0] C);
```

Solution : Unsigned MAC Unit

- **Solution**

```
module mac(output Z [7:0], output overflow,
           input [3:0] A, B, input [7:0] C);
  wire [8:0] P;
  assign P = A*B + C;
  assign Z = P[7:0];
  assign overflow = P[8];
endmodule
```

- **Alternative Method**

```
module mac(output Z [7:0], output overflow,
           input [3:0] A, B, input [7:0] C);
  assign {overflow, Z} = A*B + C;
endmodule
```

Operators

- Shift (<<, >>, <<<, >>>)
- Relational (<, >, <=, >=)
- Equality (==, !=, ===, !==)
 - ===, !== test x's, z's! ONLY USE FOR SIMULATION!
- Logical Operators (&&, ||, !)
 - Build clause for if statement or conditional expression
 - Returns single bit values
- Bitwise Operators (&, |, ^, ~)
 - Applies bit-by-bit!
- Watch ~ vs !, | vs ||, and & vs. &&

Operators

- Reduction (&, |, ^)
 - Unary!
 &(4'b0111), |(3'b010), ^(12'h502)
- Parentheses ()
 - Use to make calculations clear!
- Concatenation ({})
 - Assembles vectors
- Replication ({ })
 - // a is a 4-bit vector with the value of b
 - // as the value of each bit
 - wire [3:0] a = {4{b}};

Operators : Conditional

- Can do an "if else" assignment!
 <clause> ? <T exp> : <F exp>
- Examples:
 - assign mux_out = sel ? in1 : in0;
 - assign and2 = a ? b : 0;
 - assign xor2 = in1 ? ~in2 : in2;
 - assign triVal = sel ? in : 1'bz;
- Can nest the conditionals!
 assign trimux = trisel ? (muxsel ? a : b) : 1'bz;
- Frequently used for tristate, muxing
- Also used for complex combinational logic
- USE PARENTHESES WHEN NESTING!!!

Operator Precedence

Operators associate left to right (except conditional operator)

+ - ! ~ (unary)	Highest precedence
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != --- !---	
& ~&	
^ ~^ ~^	
~	
&&	
?: (conditional operator)	Lowest precedence

Example : Multiplexer

- Use the conditional operator and a single continuous assignment statement

```

module mux_8_to_1( output out,
                  input in0, in1, in2, in3, in4, in5, in6, in7,
                  input [2:0] sel);
    assign output =

endmodule

```

Implicit Continuous Assignments

- Can create an implicit continuous assign
- Goes in the wire declaration

```
wire [3:0] sum = a + b;
```
- Can be a useful shortcut to make code succinct, but doesn't allow fancy LHS combos

```
assign {cout, sum} = a + b + cin;
```
- Personal choice
 - You are welcome to use it when appropriate

Implicit Wire Declaration

- Can create an implicit wire
- When **wire is used but not declared**, it is implied

```
module majority(output out, input a, b, c);  
    assign part1 = a & b;  
    assign part2 = a & c;  
    assign part3 = b & c;  
    assign out = part1 | part2 | part3;  
endmodule
```

- **Lazy! Don't do it!**
 - Use explicit declarations
 - To design well, need to be "in tune" with design!

Latches

- Continuous assignments with feedback

```
module latch(output q_out, input data_in, enable);  
    assign q_out = enable ? data_in : q_out;  
endmodule
```

```
module latch_reset(output q_out, input data_in, enable, reset);  
    assign q_out = reset ? 0 : (enable ? data_in : q_out);  
endmodule
```

- How would we change these for 8-bit latches?
- How would we make the enable active low?



Verilog HDL

Behavioral Verilog

Behavioral Basics

- Use procedural blocks: **initial**, **always**
- These blocks contain series of statements
 - Abstract – works **SOMEWHAT like software**
 - Be careful to still remember **it's hardware!**
- **Parallel** operation **across** blocks
 - All blocks in a module operate simultaneously
- **Sequential or parallel** operation **within** blocks
 - Depends on the way the block is written
 - Discuss this in a later lecture
- LHS of assignments are **variables (reg)**

Types of Blocks

- **initial**
 - Behavioral block operates **ONCE**
 - Starts at time 0 (beginning of operation)
 - Useful for testbenches
 - Can **sometimes** provide initialization of memories/FFs
 - Often **better to use "reset" signal**
 - Inappropriate for combinational logic
 - Usually cannot be synthesized
- **always**
 - Behavioral block operates **CONTINUOUSLY**
 - Can use a **trigger list** to control operation; **@(a, b, c)**

initial Blocks

```
`timescale 1ns /1ns
module t_full_adder;
  reg [3:0] stim;
  wire s, c;

  // instantiate UUT
  full_adder(sum, carry, stim[2], stim[1], stim[0]);

  // monitor statement is special - only needs to be made once,
  initial $monitor("%t: s=%b c=%b stim=%b", $time, s, c, stim[2:0]);

  // tell our simulation when to stop
  initial #50 $stop;

  initial begin // stimulus generation
    for (stim = 4'd0; stim < 4'd8; stim = stim + 1) begin
      #5;
    end
  end
endmodule
```

all initial blocks start at time 0

single-statement block

multi-statement block enclosed by **begin** and **end**

always Blocks

- Operates continuously or on a trigger list
- Can be used with initial blocks
- **Cannot "nest"** initial or always blocks
- Useful example of continuous always block:

```
reg clock;
initial clock = 1'b0;
always #10 clock = ~clock;
```

- Clock generator goes in the **testbench**

always Blocks with Trigger Lists

- **Conditionally “execute”** inside of always block
 - Always block continuously operating
 - If trigger list present, continuously checking triggers
 - Any change on trigger (sensitivity) list, triggers block

```
always @(a, b, c) begin
    ...
end
```
- **Sounds like software! It isn't!**
 - This is how the simulator treats it
 - The hardware has the same resulting operation, but...
 - See examples in later slides to **see what is actually created**

Trigger Lists

- Uses **“event control operator”** @
- When net or variable in trigger list **changes**, always block is **triggered**

```
always @(a, b, c) begin
    a1 = a & b;
    a2 = b & c;
    a3 = a & c;
    carry = a1 | a2 | a3;
end
```

```
always @(in1, in0, sel) begin
    if (sel == 1'b0) out = in0;
    else out = in1;
end
```

```
always @(state, input) begin
    if (input == 1'b0) begin
        if (state != 2'b11)
            nextstate = state + 1;
        else
            nextstate = 2'b00;
    end else
        nextstate = state;
end
```

Event or

- Original way to specify trigger list
always @ (X1 or X2 or X3)
- In Verilog 2001 can use , instead of or
always @ (X1, X2, X3)
- Verilog 2001 also has * for combinational only
always @ (*)
 - Shortcut that includes all nets/variables used on RHS in statements in the block
 - Also includes variable used in if statements; if (x)
- You may be asked to specify inputs to trigger list without *

Example : Comparator

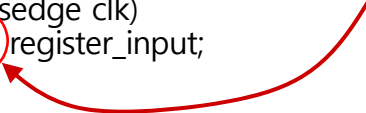
```
module compare_4bit_bhv (output reg A_lt_B, A_gt_B, A_eq_B,  
                        input [3:0] A, B);  
  
    always@(          ) begin  
  
    end  
  
endmodule
```

Edge Triggering

- A **negedge** is on the transitions
 - $1 \rightarrow x, z, 0$
 - $x, z \rightarrow 0$
- A **posedge** is on the transitions
 - $0 \rightarrow x, z, 1$
 - $x, z \rightarrow 1$
- Used for clocked (synchronous) logic

```
always @ (posedge clk)
    register <= register_input;
```

Different
assignment
operator!



Example : DFF

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

module dff_reset(output reg q, input d, clk, reset);
    always @(posedge clk, posedge reset) begin
        if (reset) q <= d; else q <= 0;
    end
endmodule
```

Combinational vs. Sequential

- **Combinational**
 - Not edge-triggered
 - **All "inputs" (RHS nets/variables) are triggers**
 - Does not depend on clock
- **Sequential**
 - Edge-triggered by clock signal
 - **Only clock (and possibly reset) appear in trigger list**
 - Can include combinational logic that feeds a FF or register

Synchronous/Asynchronous Reset

- **Synchronous** : triggered by **clock** signal

```
module dff_sync(output reg Q, input D, clk, rst);
  always @(posedge clk) begin
    if (rst) Q <= 1'b0;
    else Q <= D;
  end
endmodule
```


- **Asynchronous** : also triggered by **reset** signal

```
module dff_async(output reg Q, input D, clk, rst, en);
  always @(posedge clk, posedge reset) begin
    if (rst) Q <= 1'b0;
    else if (en) Q <= D;
  end
endmodule
```

Datatype Categories

- **Net**
 - Represents a physical wire
 - Describes structural connectivity
 - Assigned to in continuous assignment statements
 - Outputs of primitives and instantiated sub-modules
- **Variables**
 - Used in Behavioral procedural blocks
 - Can represent:
 - Synchronous registers
 - Combinational logic

Variable Datatypes

- **reg – scalar or vector binary values**  **Not necessarily a "register"!!!**
- integer – 32 or more bits
- time – time values represented in 64 bits (unsigned)
- real – double real values in 64 or more bits
- realtime - stores time as double real (64-bit +)
- Assigned value only within a behavioral block
- **CANNOT USE AS:**
 - Output of primitive gate or instantiated submodule
 - LHS of continuous assignment
 - Input or inout port within a module
- real and realtime initialize to 0.0, others to x...
 - Just initialize yourself!

Examples Of Variables

```

reg signed [7:0] A_reg;           // 8-bit signed vector register
reg Reg_Array[7:0]; // array of eight 1-bit registers
integer Int_Array[1:100];        // array of 100 integers
real B, Real_Array[0:5];         // scalar & array of 6 reals
time Time_Array[1:100];          // array of 100 times
realtime D, Real_Time[1:5];       // scalar & array of 5 realtimes
initial
begin
    A_reg = 8'h6;                 // Assigns all eight bits
    Reg_Array[7] = 1;             // Assigns one bit
    Int_Array[3] = -1;            // Assign integer -1
    B = 1.23e-4;                 // Assign real
    Time_Array[20] = $time;       // Assigned by system call
    D = 1.25;                    // Assign real time
end

```

wire vs. reg

- Same "value" used both as 'wire' and as 'reg'

```

module dff (q, d, clk);
    output reg q;           // reg declaration,
    input wire d, clk;      // wire declarations, since module inputs
    always @(posedge clk)   // why is q reg and d wire?
        q <= d;
endmodule

module t_dff;
    wire q, clk;           // now declared as wire
    reg d;                 // now declared as reg
    dff FF(q, d, clk);      // why is d reg and q wire?
    clockgen myclk(clk);
    initial begin
        d = 0;
        #5 d = 1;
    end
endmodule

```


Signed vs. Unsigned

- Net types and reg variables unsigned by default
 - Have to declare them as signed if desired!
reg signed [7:0] signedreg;
wire signed [3:0] signedwire;
- All bit-selects and part-selects are unsigned
 - A[6], B[5:2], etc.
- integer, real, realtime are signed
- System calls can force values to be signed or unsigned
reg [5:0] A = \$unsigned(-4);
reg signed [5:0] B = \$signed(4'b1101);

Operators with Real Operands

- Arithmetic
 - Unary +/-
 - + - * / **
- Relational
 - > >= < <=
- Logical
 - ! && ||
- Equality
 - == !=
- Conditional
 - ? :

Strings

- Strings are stored using properly sized registers

```
reg [12*8: 1] stringvar;           // 12 character string
stringvar = "Hello World";         // string assignment
```

- Uses ASCII values
- Unused characters are filled with zeros
- Strings can be copied, compared, concatenated
- Most common use of strings is in testbenches

Memories

- A memory is an array of n-bit registers

```
reg [15:0] mem_name [0:127];       //128 16-bit words
reg array_2D [15:0] [0:127];       // 2D array of 1-bit regs
```

- Can only access full word of memory

```
mem_name[122] = 35;                 // assigns word
mem_name[13][5] = 1;                // illegal – works in simulation
array_2D[122] = 35;                 // illegal – causes compiler error
array_2D[13][5] = 1;                // assigns bit
```

- Can use continuous assign to read bits

```
assign mem_val = mem[13];           // get word in slot 13
assign out = mem_val[5];             // get bit in slot 5 of word
assign dataout = mem[addr];
assign databit = dataout[bitpos];
```

Example : Memory

```
module memory(output reg [7:0] out, input [7:0] in, input [7:0] addr,
              input wr, clk, rst);

    reg [7:0] mem [0:255];
    reg [8:0] initaddr;

    always @ (posedge clk) begin
        if (rst) begin ← synchronous reset!
            for (initaddr = 0; initaddr < 256; initaddr = initaddr + 1) begin
                mem[initaddr] <= 8'd0;
            end
            end else if (wr) mem[addr] <= in; ← synchronous write
        end
        ← synchronous read
        always @(posedge clk) out <= mem[addr];
    endmodule
```

Procedural Assignments

- Used within a behavioral block (initial, always)
- Types
 - = Blocking assignment
 - <= Non-blocking assignment
- Assignments to variables:
 - reg
 - integer
 - real
 - realtime
 - time

Blocking Assignments

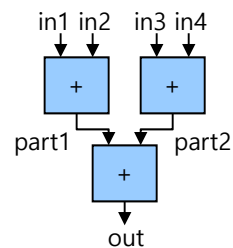
- Evaluated **sequentially**
- Works a lot like software → **danger!**
- Used for **combinational** logic

```

module addtree(output reg [9:0] out,
               input [7:0] in1, in2, in3, in4);

  reg [8:0] part1, part2;
  always @(in1, in2, in3, in4) begin
    part1 = in1 + in2;
    part2 = in3 + in4;
    out = part1 + part2;
  end
endmodule

```



Non-Blocking Assignments

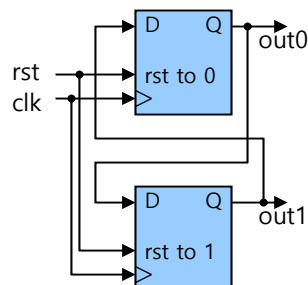
- Updated **simultaneously** if no delays given
- Used for **sequential** logic

```

module swap(output reg out0, out1, input rst, clk);

  always @(posedge clk) begin
    if (rst) begin
      out0 <= 1'b0;
      out1 <= 1'b1;
    end
    else begin
      out0 <= out1;
      out1 <= out0;
    end
  end
endmodule

```

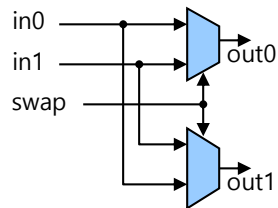


Swapping

- In blocking, need a "temp" variable

```
module swap(output reg out0, out1, input in0, in1, swap);
```

```
    reg temp;
    always @(*) begin
        out0 = in0;
        out1 = in1;
        if (swap) begin
            temp = out0;
            out0 = out1;
            out1 = temp;
        end
    end
endmodule
```



Blocking & Non-Blocking Example

```
reg [7:0] A, B, C, D;
always @(posedge clk)
begin
    A = B + C;
    B = A + D;
    C = A + B;
    D = B + D;
end
```

```
reg [7:0] A, B, C, D;
always @(posedge clk)
begin
    A <= B + C;
    B <= A + D;
    C <= A + B;
    D <= B + D;
end
```

- Assume initially that A=1, B=2, C=3, and D=4
- Note: Shouldn't use blocking with sequential! Why?**

Correcting The Example

```
reg [7:0] A, B, C, D;  
reg [7:0] newA, newB, newC, newD;  
always @(posedge clk) begin  
    A <= newA;  
    B <= newB;  
    C <= newC;  
    D <= newD;  
end  
always @(*) begin  
    newA = B + C;  
    newB = newA + D;  
    newC = newA + newB;  
    newD = newB + D;  
end
```

```
reg [7:0] A, B, C, D;  
always @(posedge clk)  
begin  
    A <= B + C;  
    B <= B + C + D;  
    C <= B + C + B + C + D;  
    D <= B + C + D + D;  
end
```

Why Not '=' In Sequential?

- **Yes, it can “work”, but...**
 - <= models pipeline stages better
 - = can cause problems if multiple always blocks
 - Order of statements is important with =
- **Use the style guidelines given!**
 - <= for sequential block
 - = for combinational block
 - Don't mix in same block!

Blocking vs. Non-Blocking (1/2)

- If behavior1 always first after reset, $y1 = y2 = 1$
- If behavior2 always first after reset, $y2 = y1 = 0$.
- Thus results are **order dependent** and ambiguous
- This is a **race condition**!

```
always @ (posedge clk or posedge rst) begin // behavior1
    if (rst) y1 = 0; // reset
    else y1 = y2;
end

always @ (posedge clk or posedge rst) begin // behavior2
    if (rst) y2 = 1; // preset
    else y2 = y1;
end
```

Blocking vs. Non-Blocking (2/2)

- Assignments for y1 and y2 occur in **parallel**
- $y1 = 1$ and $y2 = 0$ after reset
- Values swap each clock cycle after.
- **No race condition**!

```
always @ (posedge clk or posedge rst) begin // behavior1
    if (rst) y1 <= 0; // reset
    else y1 <= y2;
end

always @ (posedge clk or posedge rst) begin // behavior2
    if (rst) y2 <= 1; // preset
    else y2 <= y1;
end
```

Mixed Blocking/Non-Blocking

- **Don't ever mix!!!**
- Confusing to you... confusing to the simulator!

```
always @(posedge clk) begin
    b = a + a;
    c <= b + a;
    d <= c + a;
    c = d + a;
end

initial begin
    a = 3; b = 2; c = 1; d = 0;
end
```

Combinational/Sequential Verilog

- Can describe many circuits **several** ways
 - Even within "good practice"!
- Sometimes all equally good
- Circuit complexity can affect this choice
- Simple circuit
 - More likely to be able to use 0~2 always blocks
- Complex circuit (big FSM, etc.)
 - Can be good to separate combinational/sequential logic

Multiply-Accumulate (1/5)

```
module mac(output reg [15:0] out, input [7:0] a, b, input clk, rst);  
  
    always @(posedge clk) begin  
        if (rst) out <= 16'd0;  
        else    out <= out + (a * b);  
    end  
  
endmodule
```

Multiply-Accumulate (2/5)

```
module mac(output reg [15:0] out, input [7:0] a, b, input clk, rst);  
  
    reg [15:0] product;  
  
    always @(*) begin  
        product = a * b;  
    end  
  
    always @(posedge clk) begin  
        if (rst) out <= 16'd0;  
        else    out <= out + product;  
    end  
  
endmodule
```

Multiply-Accumulate (3/5)

```
module mac(output reg [15:0] out, input [7:0] a, b, input clk, rst);  
  
    reg [15:0] sum;  
    reg [15:0] product;  
  
    always @(*) begin  
        product = a * b;  
        sum = product + out;  
    end  
  
    always @(posedge clk) begin  
        if (rst) out <= 16'd0;  
        else    out <= sum;  
    end  
  
endmodule
```

Multiply-Accumulate (4/5)

```
module mac(output reg [15:0] out, input [7:0] a, b, input clk, rst);  
  
    reg [15:0] sum;  
    reg [15:0] product;  
  
    always @(*) product = a * b;  
    always @(*) sum = product + out;  
  
    always @(posedge clk) begin  
        if (rst) out <= 16'd0;  
        else    out <= sum;  
    end  
  
endmodule
```

Multiply-Accumulate (5/5)

```
module mac(output reg [15:0] out, input [7:0] a, b, input clk, rst);  
  
    wire [15:0] sum;  
    wire [15:0] product;  
  
    assign product = a * b;  
    assign sum = product + out;  
  
    always @(posedge clk) begin  
        if (rst) out <= 16'd0;  
        else    out <= sum;  
    end  
  
endmodule
```

Control Statements

- Behavioral Verilog looks a lot like software –
Danger!!
- Provides similar control structures
 - Not all these actually synthesize!
 - Can use them in testbenches

if... else if... else

- Operator ? for simple conditional assignments
- Sometimes need more complex behavior
- Can use **if** statement
 - **Does not** conditionally "execute" block of "code"
 - **Does not** conditionally create hardware!
 - It makes a **multiplexer** or similar logic
 - Generally:
 - Hardware for both paths is created
 - Both paths "compute" simultaneously
 - The result is selected depending on the condition

Potential Problems with if

- Can sometimes create long multiplexer chains

```
always @(select, a, b, c, d) begin
    out = d;
    if (select == 2'b00) out = a;
    if (select == 2'b01) out = b;
    if (select == 2'b10) out = c;
end
```

```
always @(a, b, c, d) begin
    if (a) begin
        if (b) begin
            if (c) out = d;
            else out = ~d;
        else out = 1;
    else out = 0;
end
```

Unintended Latches with if

- Always use **else** for combinational logic!
- Make sure all values in test either
 - In trigger list
 - Computed in same block but above test (temporary storage)

```
always @(en, a, b) begin
    if (en) c = a + b;           // latch!
end
What happens if en is 'x' in simulation?

always @(a, b) begin
    temp = a - b;
    if ((temp < 8'b0) && abs) // latch!
        out = -temp;
    else out = temp;
end
```

if Statement : Flip-Flop Set/Reset

- Does set or reset have priority?

```
module df_sr_behav (q, q_bar, data, set, reset, clk);

    input    data, set, clk, reset;
    output   q, q_bar;
    reg      q;

    assign q_bar = ~q;           // continuous assignment

    always @(posedge clk) begin // Flip-flop with synchronous set/reset
        if (reset) q <= 0;      // Active-high set and reset
        else if (set) q <= 1;
        else q <= data;
    end

endmodule
```

case Statements

- Verilog has three types of case statements:
 - case, casex, and casez
- Performs bitwise match of expression and case item
 - Both **must** have same bit-width to match!
- case
 - Can detect x and z! (only good for testbenches)
- casez
 - Uses z and ? as “don’t care” bits in case items and expression
- casex
 - Uses x, z, and ? as “don’t care” bits in case items and expression

“Don’t Care” Assignment with case

- Sometimes not all cases occur
- Can “**don’t care**” what is assigned

```
always@(state or x) begin
  case (state)                                // state is "expression"
    2'd0: next_state = 2'd1;                  // 2'd0 is case item
    2'd1: next_state = 2'd2;                  // ordering implied
    2'd2: if (x) next_state = 2'd0;
         else next_state = 2'd1;
    default: next_state = 2'dx;               // for all other states,
  endcase                                     // don't care what is
end                                           // assigned
```

- What does the state machine for this look like?
- What code is missing for this to be a complete FSM?

Mux with if...else if...else

```
module Mux_4_32_if (output [31:0] mux_out, input [31:0] data_3, data_2,  
                  data_1, data_0, input [1:0] select, input enable);  
    reg    [31: 0] mux_int;  
    // add the enable functionality  
    assign mux_out = enable ? mux_int : 32'bz;  
    // choose between the four inputs  
    always @ (data_3 or data_2 or data_1 or data_0 or select) begin  
        if (select == 0) mux_int = data_0; else  
        if (select == 1) mux_int = data_1; else  
        if (select == 2) mux_int = data_2; else  
            mux_int = data_3;  
    end  
endmodule
```

- What happens if we forget select in the trigger list?
- What happens if select is 2'bxx?

Mux with case

```
module Mux_4_32_case (output [31:0] mux_out, input [31:0] data_3, data_2,  
                    data_1, data_0, input [1:0] select, input enable);  
    reg    [31: 0] mux_int;  
    // add the enable functionality  
    assign mux_out = enable ? mux_int : 32'bz;  
    // choose between the four inputs  
    always @ (data_3 or data_2 or data_1 or data_0 or select)  
        case (select)  
            2'd0: mux_int = data_0;  
            2'd1: mux_int = data_1;  
            2'd2: mux_int = data_2;  
            2'd3: mux_int = data_3;  
        endcase  
endmodule
```

- Case statement implies priority unless use parallel_case pragma
- What happens if select is 2'bxx?

Encoder with if...else if...else

```
module encoder (output reg [2:0] Code, input [7:0] Data);
  always @ (Data) begin
    // encode the data
    if (Data == 8'b00000001) Code = 3'd0;
    else if (Data == 8'b00000010) Code = 3'd1;
    else if (Data == 8'b00000100) Code = 3'd2;
    else if (Data == 8'b00001000) Code = 3'd3;
    else if (Data == 8'b00010000) Code = 3'd4;
    else if (Data == 8'b00100000) Code = 3'd5;
    else if (Data == 8'b01000000) Code = 3'd6;
    else if (Data == 8'b10000000) Code = 3'd7;
    else Code = 3'bxxx; // invalid, so don't care
  end
endmodule
```

Encoder with case

```
module encoder (output reg [2:0] Code, input [7:0] Data);
  always @ (Data)
    // encode the data
    case (Data)
      8'b00000001 : Code = 3'd0;
      8'b00000010 : Code = 3'd1;
      8'b00000100 : Code = 3'd2;
      8'b00001000 : Code = 3'd3;
      8'b00010000 : Code = 3'd4;
      8'b00100000 : Code = 3'd5;
      8'b01000000 : Code = 3'd6;
      8'b10000000 : Code = 3'd7;
      default      : Code = 3'bxxx; // invalid, so don't care
    endcase
endmodule
```


Seven Segment Display (1/2)

```

module Seven_Seg_Display (Display, BCD, Blanking);
  output reg [6: 0]    Display;
  input          [3: 0]    BCD;
  input          Blanking;

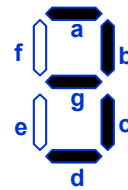
```

// Defined constants – can make code more understandable!

```

parameter BLANK  = 7'b111_1111;           // active low; abc_defg
parameter ZERO   = 7'b000_0001;
parameter ONE    = 7'b100_1111;
parameter TWO    = 7'b001_0010;
parameter THREE  = 7'b000_0110;
parameter FOUR   = 7'b100_1100;
parameter FIVE   = 7'b010_0100;
parameter SIX    = 7'b010_0000;
parameter SEVEN  = 7'b000_1111;
parameter EIGHT  = 7'b000_0000;
parameter NINE   = 7'b000_0100;

```



Seven Segment Display (2/2)

```

always @ (BCD or Blanking)
  if (Blanking) Display = BLANK;
  else begin
    case (BCD)
      4'd0:    Display = ZERO;
      4'd1:    Display = ONE;
      4'd2:    Display = TWO;
      4'd3:    Display = THREE;
      4'd4:    Display = FOUR;
      4'd5:    Display = FIVE;
      4'd6:    Display = SIX;
      4'd7:    Display = SEVEN;
      4'd8:    Display = EIGHT;
      4'd9:    Display = NINE;
      default: Display = BLANK;
    endcase
  end
endmodule

```

Register with Parallel Load

```
module Par_load_reg4 (Data_out, Data_in, load, clock, reset);
    input [3: 0] Data_in;
    input load, clock, reset;
    output reg [3: 0] Data_out;

    always @ (posedge reset or posedge clock) begin
        if (reset == 1'b1)
            Data_out <= 4'b0;
        else if (load == 1'b1)
            Data_out <= Data_in;
        end
    end
endmodule
```

Register File

```
module Register_File (Data_Out_1, Data_Out_2, Data_in, Read_Addr_1,
    Read_Addr_2, Write_Addr, Write_Enable, Clock);
    output [15:0] Data_Out_1, Data_Out_2;
    input [15:0] Data_in;
    input [2:0] Read_Addr_1, Read_Addr_2, Write_Addr;
    input Write_Enable, Clock;
    reg [15:0] Reg_File [0:7]; // 16-bit by 8-word memory declaration

    always @ (posedge Clock) begin
        if (Write_Enable)
            Reg_File [Write_Addr] <= Data_in;
        Data_Out_1 <= Reg_File[Read_Addr_1];
        Data_Out_2 <= Reg_File[Read_Addr_2];
    end
endmodule
```

- Are the reads and writes synchronous or asynchronous?