High-Level Design
Hardware Description Language
(Verilog)

C.K. Ken Yang
University of California at Los Angeles
yang@ee.ucla.edu

Overview

Reading

- W&H: Appendix A
- Verilog According to Tom
- Synopsys Handout

Introduction

The next level up in abstraction after discussing logic design and finite state machine is how to more efficiently describe them. To handle increasingly large and complex designs, we use a hardware description language instead of handling all the gates. These languages differ from algorithmic languages like Matlab or C in that they operate like hardware. Unless specified, blocks will operate concurrently much like functional hardware blocks. These blocks will interact through interfacing signals. In this lecture, we will focus on Verilog as the language of choice and introduce how we can use it to specify a system.

High-Level Design Issues

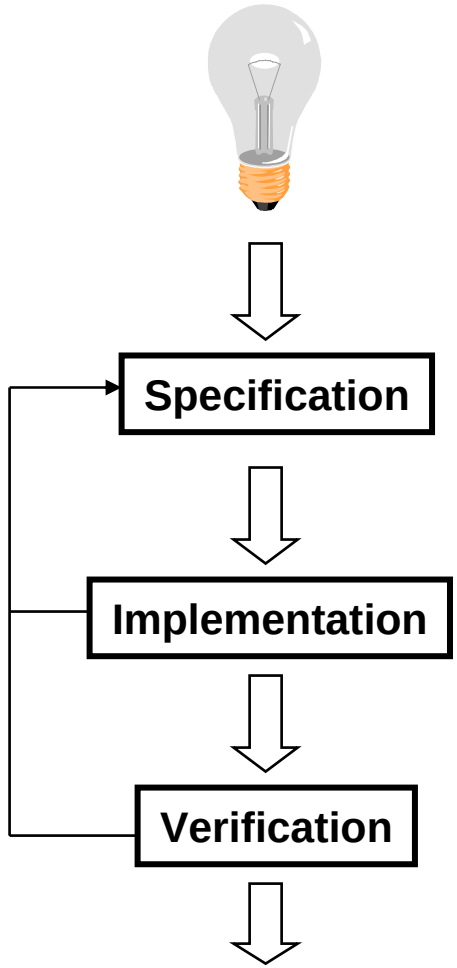
Many people think that design is a straight-forward logical process

- Start with the idea of what you need to build
- And then you build it

Real design is not like that

- Think you have an idea of what you need to build
- Through the design process you figure out what you really want to build
  - Need to validate basic idea early in the process
- What you build depends on the implementation capabilities and constraints
  - Implementation issues will change the specification

Need a language that helps with the real (interactive) design process



Hardware Description Languages

- Need a description level up from logic gates.
- Work at the level of functional blocks, not logic gates
  - Complexity of the functional blocks is up to the designer
  - A functional unit could be an ALU, or could be a microprocessor
- The description consists of functions blocks and their interconnections
  - Describe functional block (not predefined)
  - Support hierarchical description (function block nesting)
- To make sure the specification is correct, make it executable.
  - Run the functional specification and check what it does

Hardware Description Languages (HDLs)

There are many different systems for modeling and simulating hardware.

- **Verilog**
- **VHDL**
- L-language, M-language (Mentor)
- DECSIM (DEC)
- Aida (IBM / HaL)
- and many others

The two most standard languages are Verilog and VHDL.

- For this class (and many others) we will be using Verilog
- Given to UCLA for classes
- Runs on many machines (including in HP Computer Lab)
- Have both a simulator and synthesis tools that work with Verilog

Verilog from 20,000 Feet

- Verilog Descriptions look like programs:

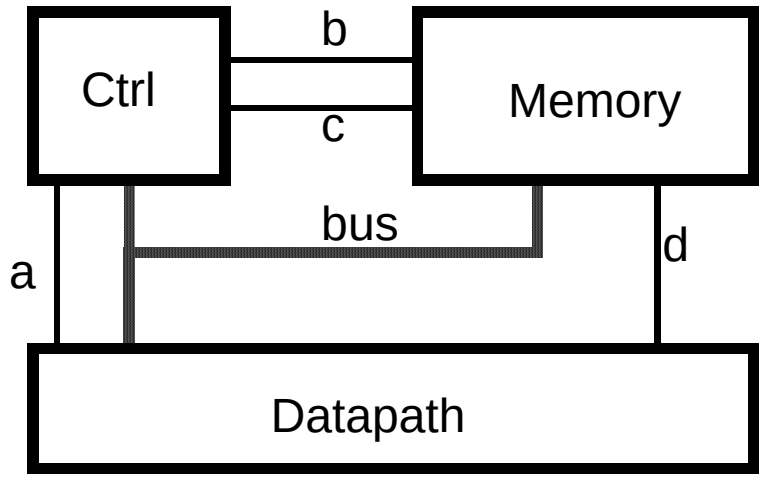
C / Pascal	Verilog
Procedures/Functions	Modules
Procedure parameters	Ports
Variables	Wires / Regs

- Block structure is a key principle
  - Use hierarchy/modularity to manage complexity

- But they aren't 'normal' programs
  - Module evaluation is concurrent. (Every block has its own "program counter")
  - Model is really communicating blocks

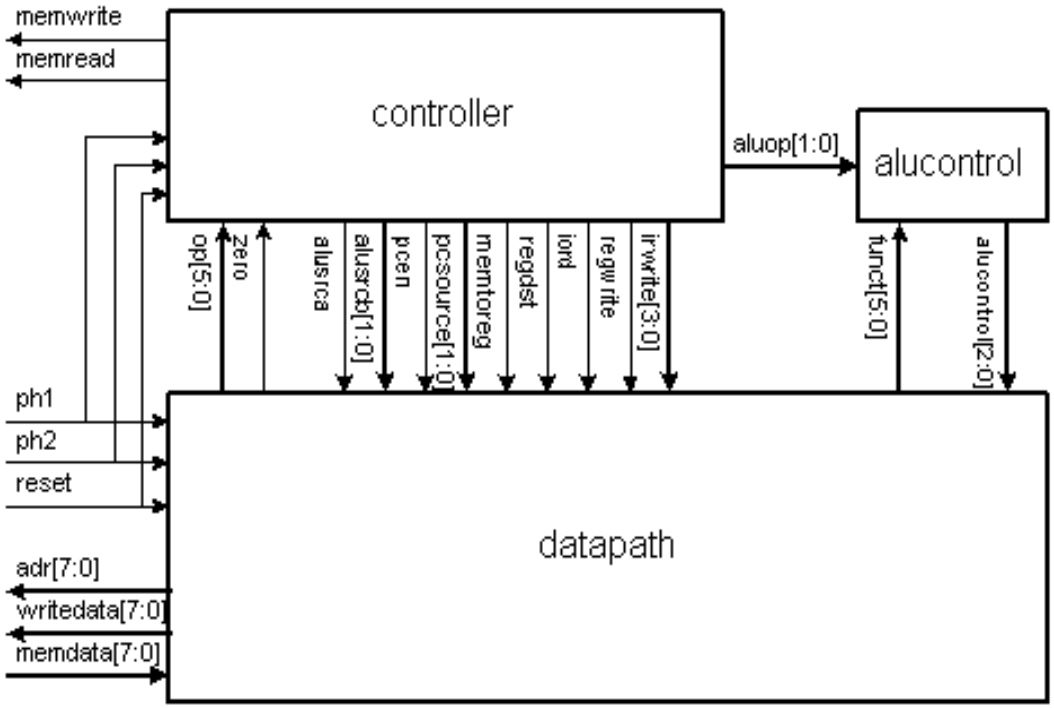
Verilog (or any HDL) View of the World

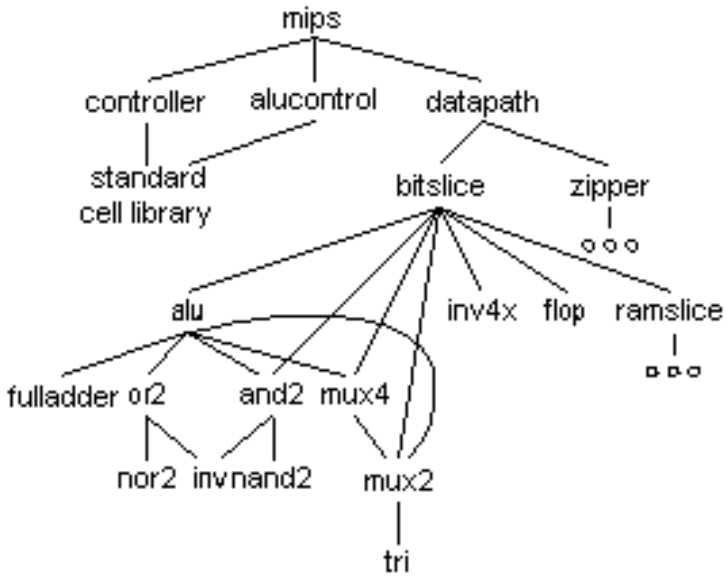
- A design consists of a set of communicating modules
- There are graphic inputs devices for Verilog, but we will not use them
- Instead we will use the text method. Label the wires, and pass them between modules as you would parameters in function calls.
  - Wires are input/output nets for a module.



Physical and Verilog Partition

- Handy to use the same partition in Verilog and physically.
  - Partition is hierarchical
  - Allows different implementation method for the two blocks.
  - Example: MIPS processor





Ways to Describe A Function

- Structural
  - Consists only of module calls
- Declarative
  - Concurrently executed combinational logic
- Procedural
  - Sequentially executed program
    - A state machine (with storage)
    - Or combinational logic
- Functional
  - Function calls
  - not mapped to hardware so we ignore this

Structural Description

```
module system;
    wire [7:0] bus_v1, const_s1;
    wire [2:0] regSpec_s1, regSpecA_s1,
          regSpecB_s1;
    wire [1:0] opcode_s1;
    wire Phi1, Phi2, writeReg_s1,
          ReadReg_s1,nextVector_s1
    clkgen clkgen(Phi1, Phi2);
    datapath datapath(Phi1, Phi2,
          regSpec_s1, bus_v1,
          writeReg_s1, readReg_s1);
    controller controller1(Phi1, Phi2,
          regSpec_s1, bus_v1, const_s1,
          writeReg_s1, readReg_s1,
          opcode_s1, regSpecA_s1, regSpecB_s1,
          nextVector_s1);
    patternsource patternsource(Phi1,
          Phi2,nextVector_s1, opcode_s1,
          regSpecA_s1, regSpecB_s1, const_s1);

ModuleName InstanceName (wires);
```

- In this example the instance name and the module name are the same, except for controller1.

- Compose a module out of module calls.
  - Specify components, and wiring
- Maps a physical structure into verilog.
  - Example is one shown earlier.
- Possible heirarchically
  - List of functions.
  - List of sub-functions.
  - List of gates
  - List of transistors.
- Typically don’t need to go below a gate level list.
  - And only near the end of a design.

Example: Gate-Level Structural Verilog

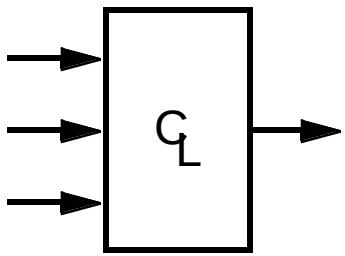
```
module V_mux_4 (in1, in2, out1, sel);
input [3:0] in1, in2;
input sel;
output [3:0] out1;
    MX2X1 I_0 (.A(in1[0]), .B(in2[0]), .Y(out1[0]),
   .S0(sel));
    MX2X1 I_1 (.A(in1[1]), .B(in2[1]), .Y(out1[1]),
   .S0(sel));
    MX2X1 I_2 (.A(in1[2]), .B(in2[2]), .Y(out1[2]),
   .S0(sel));
    MX2X1 I_3 (.A(in1[3]), .B(in2[3]), .Y(out1[3]),
   .S0(sel));
endmodule
```

- This is commonly the output of synthesized logic that has been assigned to logic gates.
  - The node names use a special convention to identify specific port of the logic cell (even if logically equivalent)
  - The *A*-input of the MUX corresponds to *in1*

Declarative Statements

Provides the logical relations between inputs and outputs.

- Assign outputs to be some function of the inputs (continuously)
  - Key word is `assign`
- Models a piece of combinational logic
- Uses a C-like expression syntax
- Denoted by keyword assign



Examples (all execute in parallel):

```
assign    nor = ~(b | c);
assign    a = x & y, o = x | y;
assign    sum[4:0] = a[3:0] + b[3:0];
assign    out = (Sel) ? in1: in2; //conditional
```

Outputs are wires, and can be a single bit or multiple bits.

- It is good practice to declare all variables even though Verilog allows undeclared single bit wires.

Declarative Order of Execution

- Even though declarative statements are still executed in a particular order.
  - Verilog has an internal event linked list.
  - There is no guarantee of that order

```
assign   out = aaa;
assign   out = bbb;
```

  - This yields a warning and is not allowed.
- Don’t assume any particular order.
  - Each statement is occuring concurrently.

```
assign   x = aaa;
assign   aaa = bbb;
```

  - In C, it matters the order of the above statements but not in Verilog.

Procedural Statements

- Still need flow control statements
  - This type of control statement implies **sequential ordering**
  - keyword `always` provides functionality of a tiny program that executes sequentially.
- Inside an `always` block, can use standard control flow statements:
  - `if (<conditional>) then <statements> else <statements>;`
  - `case (<var>) <value>: <statements>; … default: <statements>`
  - Case statements are actually prioritized
    - The second case entry can't happen unless the first does not match.
    - May not be what the actual hardware implies – especially when cases are mutually exclusive.
    - Need additional directives (parallel-case) to indicate this. More later.
- Statements can be compound (use `begin` and `end` to form blocks)
- Example:

```
always @ (Activation List…stuff we still need to talk about)
    begin
    // more than 1 statement allowed inside here
            if (x==y) then
                    out= in1
            else
                    out = in2;
    end
```

Always Block Issues

- Two issues with always blocks
  - unset outputs
    - Are all outputs given a value with an explicit assignment statement at the end of the block?
    - If not, then it is unset.
    - Similar to switch logic.
    - If the output is always set, then the always block is no different from a combinational logic.
  - activation list
    - Determines when to execute the always block.

Unset Outputs

- Occur when an output of the block is not set on all the paths through the code.
- Example:

```
always @ (Activation List…stuff we still need to talk about)
    begin
    // more than 1 statement allowed inside here
            if (x==y) then
                    out= in1
    // not else so if x!= y then out is unset.
    end
```

  - In Verilog, this creates storage
    - The value of the output remains the previous value.
    - Similar to dynamic storage, except in synthesized result, it appears as an explicit FF or latch.
- Is this storage what we want?
  - Be careful to not build storage elements when you don't intend to.
- Since the outputs of a `always` blocks MIGHT act as storage elements
  - Left-hand sides of expressions in a `always` blocks must be declared as registers (regs).
    - Note, that does not mean the synthesized result contains registers.
    - Output is set on all paths so there is no storage.

Intentionally Creating Storage in Verilog

- To make a simple latch in Verilog is easy. Just make the output of an `always` block not get set when you want to hold its value.
- Example:

```
reg myout; //a latch
always @ (stuff we still need to talk about)
        if (Enable) then
            myout = in;
```

  - When `Enable` is high, the output `myout` is updated
  - When `Enable` is low, `myout` will hold its last value.
  - This is like the simple pass transistor latch in Lecture 6.

- In this example, `myout` would need to be declared a register, because it is the LHS of an expression in an `always` block.

Activation List

- The last tricky part about the `always` block is the activation list.
- Activation List
  - Tells the simulator when to run this block
  - Allows the user to specify when to run the block and makes the simulator more efficient.
    - If not sensitized to every input, you get a storage element
  - But also enables subtle errors to enter into the design.
- Two forms of activation list in Verilog:
  - `@(signalName or signalName or …)`
    - Evaluate this block when any of the named signals change (either positive or negative change)
  - `@(posedge signalName);` or `@(negedge signalName);`
    - Makes an edge triggered flop. Evaluates only on one edge of a signal.
    - Can have `@(posedge signal1, posedge signal2)`
      - Implied OR (not AND) because edges are singular events
      - Not used in this class because difficult to map to an actual gate.

Activation Lists

- Example:

```
always @ (Enable or In)
    if (Enable) then
    out=In;
always @ (x or y or in1 or in2) //combinational logic
    begin
    if (x==y) then
        out= in1
    else
        out = in2;
    end //same as out = (x==y) ? in1 : in2;
```

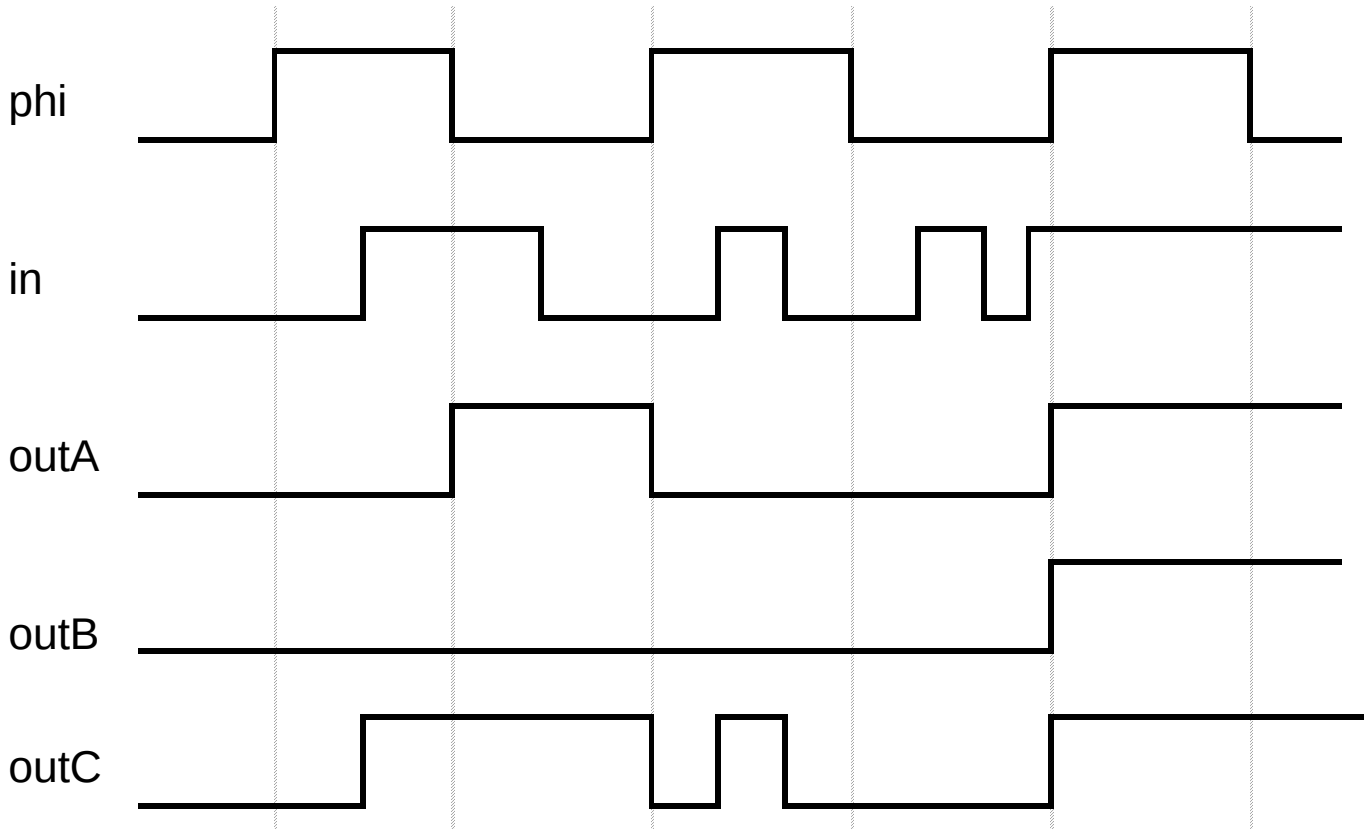
- To represent Combinational Logic
  - The activation lists must contain **everything** on the RHS of the expressions (and both side of conditionals).
  - Otherwise, there is implied storage.
- Beware, if an always block has **no** activation list (or # delay statements), then the simulator goes into an infinite loop.

Activation Errors - Examples

```
always @(phi)
outA =in;
```

```
always @(phi)
if(phi) outB = in;
```

```
always @(phi or in)
if(phi) outC = in;
```



Procedural Order of Execution

- Be careful of the sequential nature. C-like behavior
- Case 1

```
always @(posedge clock) begin
   q2=q1;
   q1=q0;
end
```

- Case 2

```
always @(posedge clock) begin
   q1=q0;
   q2=q1;
End
```

- Case 3 – Which one is this case more similar to?

```
always @(posedge clock) begin
   q1=q0;
end
always @(posedge clock) begin
   q2=q1;
end
```

Non-Blocked Assignment

- A newer feature of Verilog helps by eliminating the order of evaluation.
  - Instead of "=" ; known as a blocking assignment
    - Blocks future action until RHS is updated.
  - Use "<=“; known as non-blocking assignment
    - All LHS are changed first before the RHS is updated.

```
always @ (posedge clock)
   begin
        a[0] <= inp;
        a[1] <= a[0];
        a[2] <= a[1];
        a[3] <= a[2];
   end
```

- The above is equivalent to `a[3:0]={a[2:0],inp};`
- If we had used "=" instead of "<=", then `a = 4{inp};`

`Initial` Block

- This is another type of procedural block
  - Does not need an activation list
  - It is run just once, when the simulation starts.

- Used to do extra stuff at the very start of simulation
  - Initialize simulation environment
  - Initialize design
    - This is usually only used in the first pass of writing a design.
    - Beware, real hardware does not have initial blocks.
  - Allows testing of a design (outside of the design module)
- Best to use `initial` blocks only for non-hardware statements (like `$display` or `$gr_waves`)

Summary of Verilog Variables

- There are two types of "physical" variables in Verilog:
  - Wires (all outputs of `assign` statements must be wires)
  - Regs (all outputs of `always` blocks must be regs)
- Both variables can be used as inputs anywhere
  - Can use regs or wires as inputs (RHS) to `assign` statements

    ```
    assign bus = LatchOutput + ImmediateValue
    ```

    - `bus` must be a wire, but `LatchOutput` can be a reg
  - Can use regs or wires as inputs (RHS) in `always` blocks

    ```
    always @ (in or clk)
    if (clk) out = in
    ```

    - `in` can be a wire, `out` must be a reg
- Module outputs are typed can be either regs or wires.

```
module div_ctrl(ctl1,ctl2,dp1, clock, reset, start);
        output  ctl1,ctl2;
        input   dp1, clock, reset, start;
```

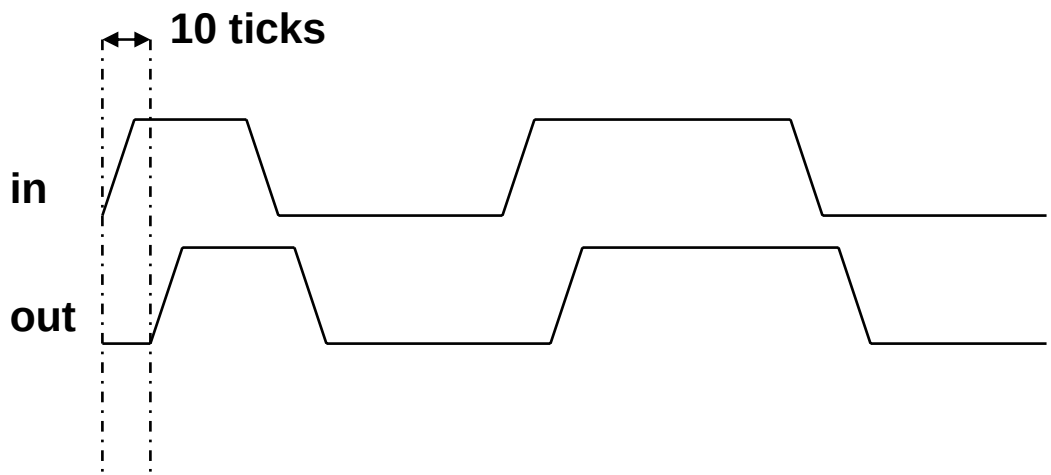
- `Integer` and `real` do not map into hardware.
  - Useful for initial functional description but not for implementation.

Delays in Verilog

- Verilog simulated time is in “units” or “ticks”.
  - Simulated time is unrelated to the wall-clock to run the simulator.
  - Simulated time models the time in the modeled machine
    - When the computer completes with all the “events” that occur at the current simulated time
    - The computer increases time until another signal is scheduled to change values.
- User must specify delay values explicitly to Verilog
  - `# delayAmount`
    - When the simulator sees this symbol, it stops “evaluating”, and pause `delayAmount` of simulated time (# of ticks).
    - Delays are often used to model the delay in functional units.
    - Can be tricky to use properly
  - We will design our logic to have zero (or unit) delay.
    - The standard cell library we use can annotate delay information.

Declarative Delay Control

- A way to specifying delay of a signal
- Make out a delayed version of the input (by 10 ticks)
  - `assign #10 out = in;`
  - Delayed assignment.
- Anywhere else to put delay is not allowed
  - `assign out = #10 in;` //is not allowed



Procedural Delay Control

- Procedural delay control is a little tricky

```
always @(phi or in)
   #10 if (phi) then out = in;
```

  - Wait 10 ticks after either input changes, then checks to see if phi == 1, and then updates the output.
    - Delayed evaluate
- If you wanted to sample the input when it changed, and then delay updating the output:

```
always @(phi or in)
   if (phi) then out = #10 in;
```

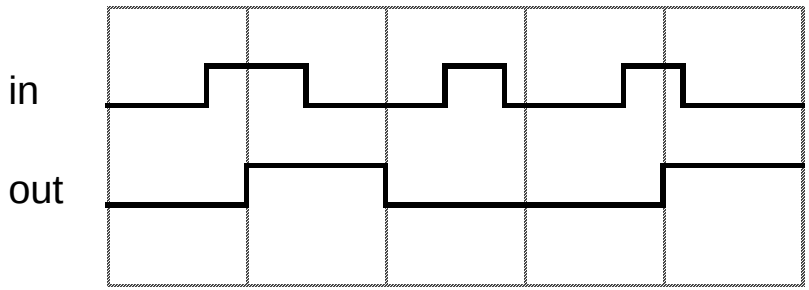
  - This code runs the code when the inputs change, and just delays the update of the output for 10 ticks.
    - Delayed assignment
  - An always block is not reactivated until every line of code is completed.
    - So while waiting for the delayed event, new inputs are ignored.

Delay Control

- Example

```
always
   #100 out = in;
```

- Since the `always` does not have an activation, it runs all the time.
  - As a result every 100 time ticks the output is updated with the current version of the input.



- Delay control is used most commonly for clock or pattern generation;

```
always
   #100 out = ~out;
```

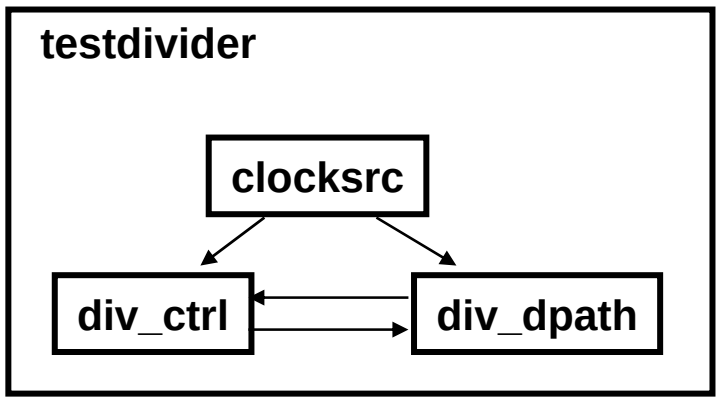
Verilog Code for a State Machine

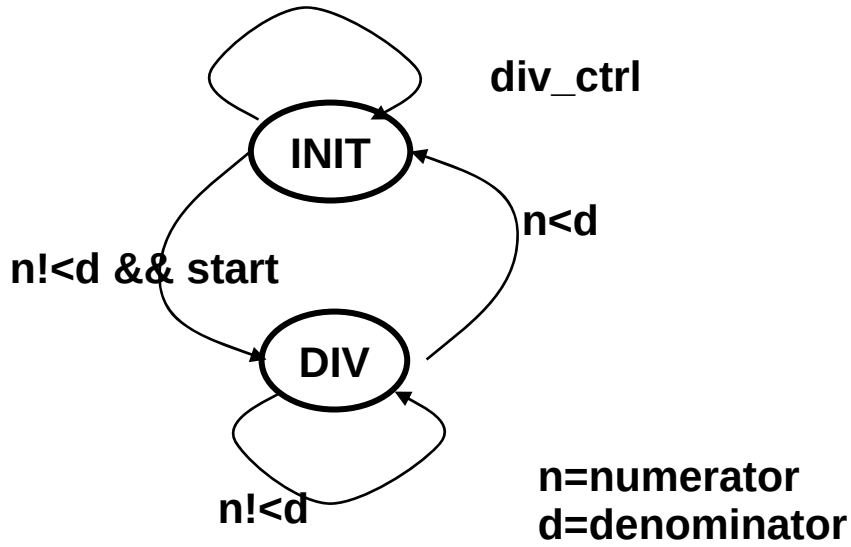
- State Transition diagrams convert nicely to always blocks
  - Use `case` statement to get into the correct state
  - Use another `case`, or `if - then - else` to deal with the inputs
  - At the end of every choice, set the next state, and the outputs
- Must be cautious about not creating any accidental storage elements.
  - A very constrained way of writing Verilog can avoid accidental flipflops
  - Often helps to make an `always` block be only combinational logic
    - Uses *currentState* and the inputs
    - Produces *nextState* and the outputs
  - Then use a separate `always` block for the storage
  - Easier to make sure that the “logic” block does not have any accidental latches in it.

Verilog Example – Divider

- Example of a serial Divider using subtract-compare algorithm.
  - Start indicates the starting of subtracting the denominator from the numerator until numerator is less.
  - The divider is broken into two parts: div_ctrl, div_dpath

- Uses flipflop based clocking for the entire machine





Code for Divider – 1

```
module div_ctrl(ctl1,ctl2,dp1, clock, reset, start);
        output  ctl1,ctl2;
        input   dp1, clock, reset, start;

reg     state_s, state_v;
wire    n_less_d;

parameter
        INIT = 1'b0,
        DIV = 1'b1;

always @(posedge clock) begin
        state_s = state_v;
end
always @(state_s or reset or start or n_less_d) begin
    if (reset) begin
        state_v = INIT;
    end
    else begin
        case (state_s)
        INIT:
            if (start)
                if (n_less_d)
                    state_v = INIT;
                else
                    state_v = DIV;
            else
                state_v = INIT;
        DIV:
            if (n_less_d)
                state_v = INIT;
            else
                state_v = DIV;
        endcase
    end
end
assign ctl1 = state_s;
assign n_less_d = dp1;
Endmodule

module div_dpath(ctl1, ctl2, dp1, num_in, den_in,
     quot, rem, clock);
    output      dp1;
    input       ctl1,ctl2, clock;
    input       [15:0] num_in,den_in;
    output      [15:0] quot, rem;

    reg         [15:0] num_s, den_s, tmp_quot_s,
     quot, rem;
    wire        overflow;
    wire        [15:0] num_v,den_v,tmp_quot_v,
     quot_v, rem_v;
    wire        [15:0] comp_v;
    wire        count, n_less_d;
```

Flipflop

Logic

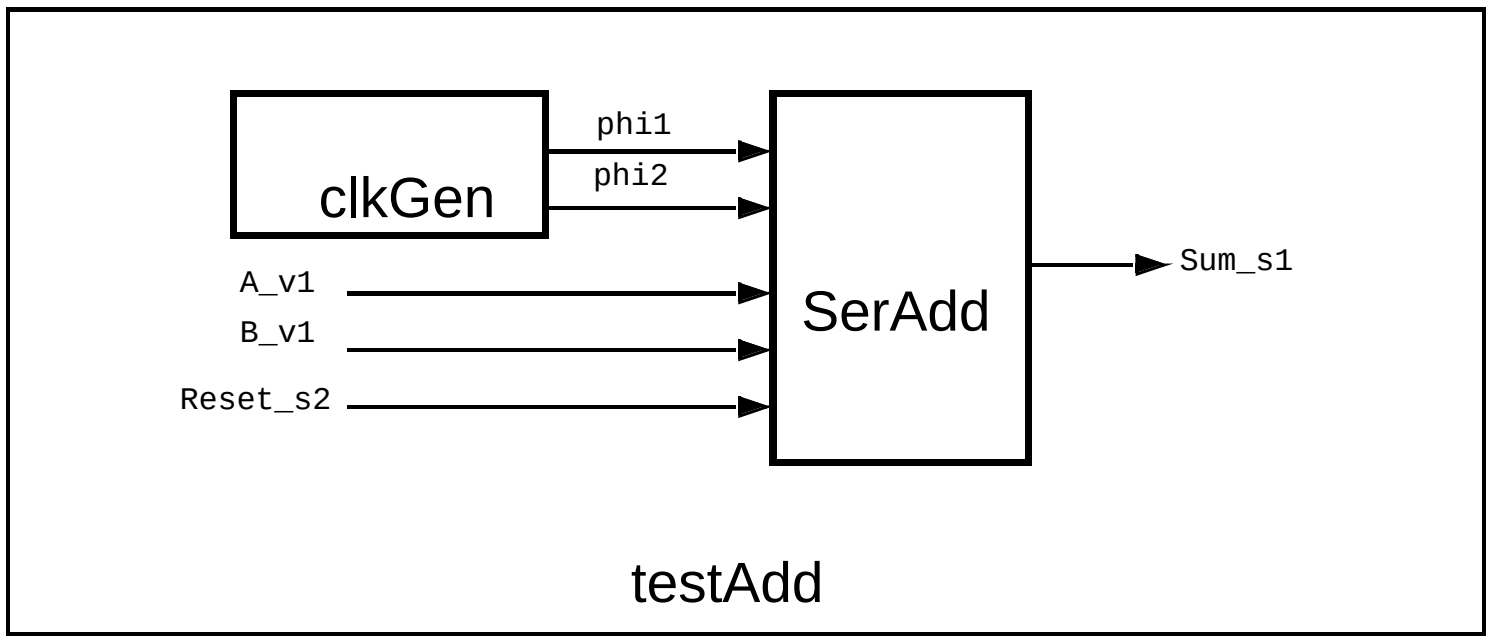
Logic

Code for Divider – 2

```
always @(posedge clock) begin
    num_s = num_v;
    den_s = den_v;
    quot = quot_v;
    tmp_quot_s = tmp_quot_v;
    rem = rem_v;
end
assign #1 {overflow,num_v} = count ? num_s - den_s :
     {1'b0,num_in};
assign #1 den_v = count ? den_s : den_in;
assign #1 tmp_quot_v = count ? tmp_quot_s + 1 : 16'b0;
assign #1 quot_v = count ? tmp_quot_s : quot;
assign #1 rem_v = count ? num_s : rem;
assign n_less_d = num_v[15];
// you can also do another subtraction to look ahead.
//      assign comp_v = num_v - den_s;
//      assign n_less_d = comp_v[15];
//      quot_v = count ? tmp_quot_v : quot;
// yet another alternative if you use a 2nd subtraction
     is to
//      introduce another state into the state machine
     to save the quot
//      register.

assign count = ctl1;
assign dp1 = n_less_d;
endmodule

module clocksrc (out);
    output out;
    reg clk;
    initial
        clk = 1'b0;
    always #100
        clk = ~clk;
    assign out = clk;
endmodule

module testdivider;
    reg         [15:0]  num_in,den_in;
    reg         reset, start;
    wire        clock,dp1,ctl1,ctl2;
    wire        [15:0]  quot, rem;

    clocksrc    clkmod(clock);
    div_dpath   dpath(ctl1, ctl2, dp1, num_in,
     den_in,quot, rem, clock);
    div_ctrl    ctrl(ctl1, ctl2, dp1, clock,reset,
     start);

    initial begin
        #1      reset = 1'b0;
                start = 1'b0;
                $dumpvars(2,testdivider);
        #10     reset = 1'b1;
                num_in = 16'b0000000000001111;
                den_in = 16'b0000000000000100;
        #400    reset = 1'b0;
                start = 1'b1;
        #600    num_in = 16'b0000000000011111;
        #600    start = 1'b0;
        #100000 $finish;
    end
endmodule
```

Verilog Example - SerAdd

- Example of a serial Adder called `serAdd` that is called by a top-level module called `testAdd`
- Uses latch-based 2 phase clocking
- They are separate just to isolate the real hardware from the shell we are using for illustration.



Code for SerAdd – 1

```
// serAdd.v -- 2 phase serial adder module
module serAdd(Sum_s1, A_v1, B_v1, Reset_s2,
              phi1, phi2);
output Sum_s1;
input   A_v1, B_v1, phi1, phi2, Reset_s2;

reg Sum_s1;
reg A_s2, B_s2, Carry_s1, Carry_s2;

always @(phi1 or A_v1)
        if (phi1)
                A_s2 = A_v1;

always @(phi1 or B_v1)
        if (phi1)
                B_s2 = B_v1;

always @(A_s2 or B_s2 or Reset_s2 or Carry_s2 or phi2)
        if (phi2)
            if (Reset_s2) begin
               Sum_s1 = 0;
               Carry_s1 = 0;
             end
            else begin
               Sum_s1   = A_s2 + B_s2 + Carry_s2;
               Carry_s1 = A_s2 & B_s2 |
                          A_s2 & Carry_s2 |
                          B_s2 & Carry_s2;
            end

always @(Carry_s1 or phi1)
        if (phi1)
                Carry_s2 = Carry_s1;

endmodule
```

Latch (phi1)

Latch (phi2)

Latch (phi1)

```
// testAdd.v -- serial adder test vector generator


// 2 phase clock generator

module clkGen(phi1, phi2);
output phi1,phi2;
reg phi1, phi2;

initial
        begin
                phi1 = 0;
                phi2 = 0;
        end

always
        begin
                #100
                        phi1 = 0;
                #20
                        phi2 = 1;
                #100
                        phi2 = 0;
                #20
                        phi1 = 1;
        end
endmodule

/*
The above clock generator will produce a clock with
a period of 240 units of simulation time.
*/
```

Code for SerAdd – 2

```
/* // test module for the adder
module testAdd; // top level

wire     A_v1, B_v1;
reg      Reset_s2;

serAdd serAdd(Sum_s1, A_v1, B_v1, Reset_s2, phi1,
phi2);

/*
 The serial adder takes inputs during phi1
 and produces _s1 outputs during phi2.
 The _s1 output corresponds to the addition of
 the inputs at the previous falling edge of phi1
*/

clkGen clkGen(phi1,phi2);


reg [5:0] tstVA_s1, tstVB_s1;
reg [6:0] accum_Sum;

initial
 $gr_waves("phi1",phi1,"phi2",phi2,
            "Reset_s2",Reset_s2,"A_v1",A_v1,
            "B_v1",B_v1,"Sum_s1",Sum_s1,
            "Carry_s1",serAdd.Carry_s1,
            "accum_Sum",accum_Sum);

/*
Since SerAdd is a serial adder, we put in the
operands one bit at a time, and accumulate the
output one bit at a time.
 */
assign A_v1 = tstVA_s1[0];
assign B_v1 = tstVB_s1[0];

always @(posedge phi1) begin
                #10
                release A_v1;
                release B_v1;
end

always @(posedge phi2) begin
                #10
                force A_v1 = 1'b0;
                force B_v1 = 1'b0;
end

initial begin
        Reset_s2 = 1;
        tstVA_s1 = 6'b01000;
        tstVB_s1 = 6'b11010;
        accum_Sum = 0;
        @(posedge phi1)
                #50 Reset_s2 = 0;
end

always @(negedge phi1) begin
        $display ("A_v1=%h, B_v1=%h,
                   sum_s1=%h, time=%d",
                   A_v1, B_v1, Sum_s1,$time);
        accum_Sum = accum_Sum << 1 | Sum_s1;
        $display ("tstVA=%h, tstVB=%h,
                  sum_s1=%h,accum_Sum=%h\n",

tstVA_s1,tstVB_s1,Sum_s1,accum_Sum);
end

always @(posedge phi2) begin
 #15
        if (~Reset_s2) begin
            tstVA_s1 = tstVA_s1 >> 1;
            tstVB_s1 = tstVB_s1 >> 1;
           if (tstVA_s1 == 0 && tstVB_s1 == 0) begin
                        #800 $stop;
            end
        end
end
endmodule
```

Code for SerAdd – 1 (Separate Latches)

```
//SerAdd code with isolated latches
module serAdd(Sum_s1, A_v1, B_v1, Reset_s2, phi1,
     phi2);
     output  Sum_s2;
     input   A_v1, B_v1, phi1, phi2, Reset_s2;

reg  Sum_s1, A_s2, B_s2, Carry_s1, Carry_s2;
reg  Sum_v2, Carry_v2;

always @(phi1 or A_v1)
     if (phi1) A_s2 = A_v1;
always @(phi1 or B_v1)
     if (phi1) B_s2 = B_v1;
always @(phi1 or Carry_s1)
     if (phi1) Carry_s2 = Carry_s1;

always @(phi2 or Carry_v2)
     if (phi2) Carry_s1 = Carry_v2;
always @(phi2 or Sum_v2)
     if (phi2) Sum_s1 = Sum_v2;

always @(A_s2 or B_s2 or Reset_s2 or Carry_s2)
     if (Reset_s2) begin
        Sum_v2 = 0;
        Carry_v2 = 0;
     end
     else begin
        Sum_v2 = A_s2 + B_s2 + Carry_s2;
        Carry_v2 = A_s2 & B_s2 |
             A_s2 & Carry_s2 |
             B_s2 & Carry_s2;
     end
endmodule
```

Latch (phi1)

Latch (phi2)

Logic

Logic

Either code style is fine. The strict version **guarantees** mapping/binding to the latches and logic.

Verilog Summary

- An HDL provides a means for the user to specify a design at a higher level than just gates.
  - This lecture addresses mostly form and not content
    - How to represent combinational logic and state machines
  - We can now use this tool to specify any machine with state.
- A good question to ask is
  - “What should my code look like?”
  - “Are there certain styles of hardware that are easier to understand / build / test?”
  - This gets back to the question of abstractions, and is really asking whether there are some hardware abstractions that work well.
- We’ve talked about partitioning of the problem as a good approach
  - Finite State Machines
  - Dataflows

Synthesis

Logic Synthesis and Place & Route

1. Convert HDL into logic gate netlist (standard cell library).
   - For us, synopsys – synthesis tool
   - Translate HDL
   - Optimize logic
   - Map into gates
2. Place the gates and connect between them
   - For us, Silicon Ensemble – place and route tool
   - Place cells within specified area constraint
   - Connect between cells with minimum distance

- Ultimate goal is speed/area/power.
  - Typically need to iterate the design
    - Identify critical nets each time to optimize.

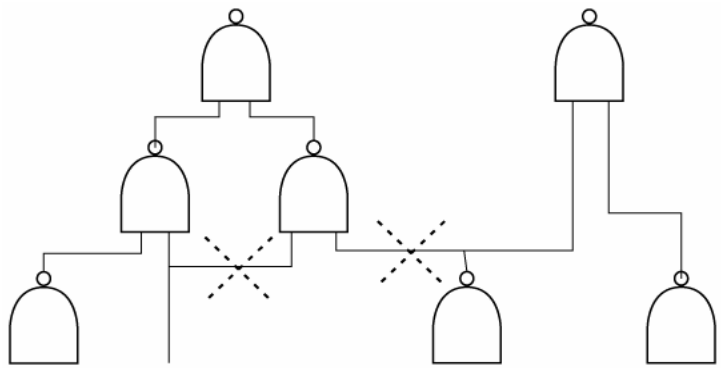
Guts of a Synthesis Tool

- Parsing VHDL
  - Language is quite simple.
- Logic Optimization
    - Reduce logic (2-level logic reduction)
    - Factorization
    - Merging
    - If specified as a FSM, determine states and optimal state assignments.
- Mapping into a standard cell library
  - Many mapping possible, depends on the size of the library.
    - Can be quite slow
  - Can use functional blocks in library to synthesize bigger logical functions
    - Entire adder/multiplier etc.

Gate Binding

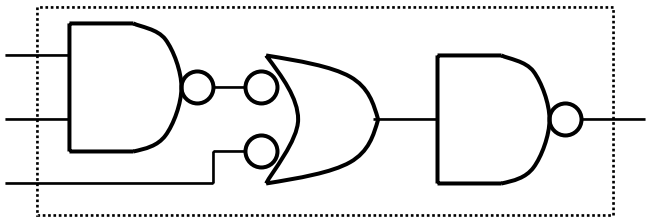
- Binding – the mapping from a boolean equation into gates in a standard cell library.
  - A.k.a. technology mapping
  - Many possible bindings.
  - Pattern matching problem
    - Minimize number of gates

- Example gate-binding methodology
  - Write Boolean network in canonical NAND form.
  - If reconvergent or splitting, break it into simple trees.



Gate-Binding Example

- Write each library gate in canonical NAND form.
  - Example: And-Or-Invert



- Assign cost to each library gate
  - Speed/Area/Power
- If network is a tree, can use dynamic programming to select minimum-cost cover of network by library gates

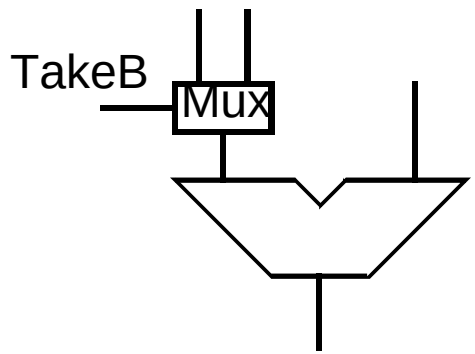
Helping Synthesis

- In an ideal world it shouldn't matter how you write the Verilog
  - Optimization in the CAD tools will find the best solution
- But the world is not ideal, yet (and progress is slow)
  - Logic optimizers tend to use your module decomposition
    - Need to partition.
    - Choose which part to custom design (structural verilog)
  - Tools use your code as a starting point
    - Your structure is not completely eliminated during logic optimization. (probably good)
    - Finding a "good" way to think about the problem is key.
  - Experience helps with finding a good architecture/algorithm to start with.
- Like optimizing compliers for C, logic optimizers are good for local optimizations, but don't expect them to rewrite your code and change your algorithm.

Branches in Computers

- In a computer, the address of the next instruction can depend on the result of a previous branch instruction. Assume that the preceding branch has a compare in it, and this signal is called *TakeBranch*. Your job is to speed up the PC adder, since it is on the critical path after the branch comparison.

```
if (TakeBranch) then
  PCBus_s2 = PCreg_s2 + Disp_s2;
else
  PCBus_s2 = PCreg_s2 +1;
```

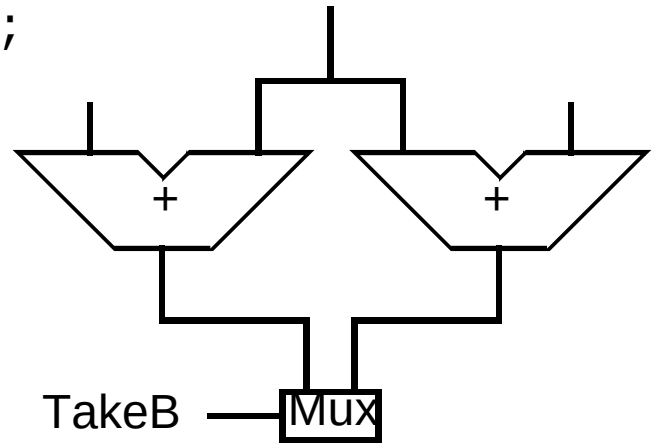


- Since TakeBranch arrives late, the add must be fast

Faster Branches

- Change the code to do the adds and compares in parallel
  - Then use the comparison to do a “late select”
    - Takes two adders, but is much faster

```
PCDisp_s2 = PCreg_s2 + Disp_s2;
PCNext_s2 = PCreg_s2 + 1;
  if (TakeBranch) then
       PCBus_s2 = PCDisp_s2;
  else
       PCBus_s2 = PCNext_s2;
```

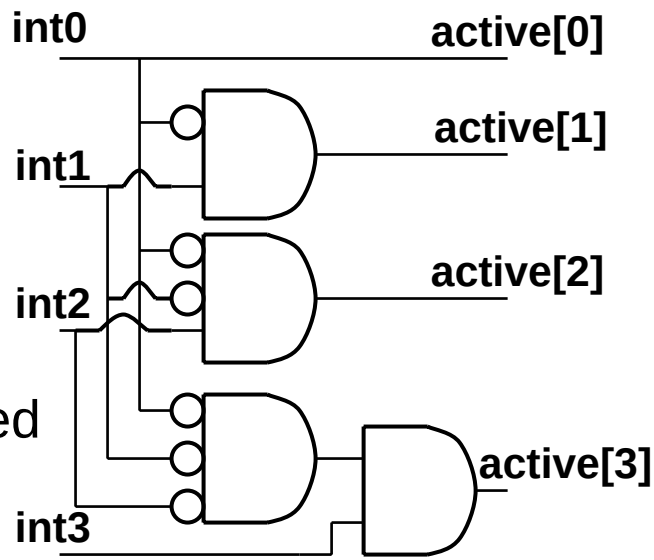


- Explicitly defining the intermediate signals help determine the position of the MUX.

If-then-elseif Implies Priority

- Example: Priority Encoder for interrupts

```
always @(int0 or int1 or int2 or int3) begin
  active[3:0] = 4'b0000;
  if      (int0) active[0] = 1'b1;
  else if (int1) active[1] = 1'b1;
  else if (int2) active[2] = 1'b1;
  else if (int3) active[3] = 1'b1;
end
```



- Using "case" is safer and can be controlled

Case Statements

- Don’t forget the default state, otherwise storage is created

```
always @(active[0] or active[1] or active[2] or active[3]) begin
  case ({active[0], active[1], active[2], active[3]})
    4’b1000: temp = 3’b010;
    4’b0100: temp = 3’b011;
    4’b0010: temp = 3’b100;
    4’b0001: temp = 3’b101;
    4’b0000: temp = 3’b000;
   default: temp = 3’bxxx;
  endcase
 end
```

- Use `// synopsys full_case` helps avoid storage
- Case also implies priority
  - Use `// synopsys parallel_case` to avoid implying any priority, when the states are mutually exclusive.

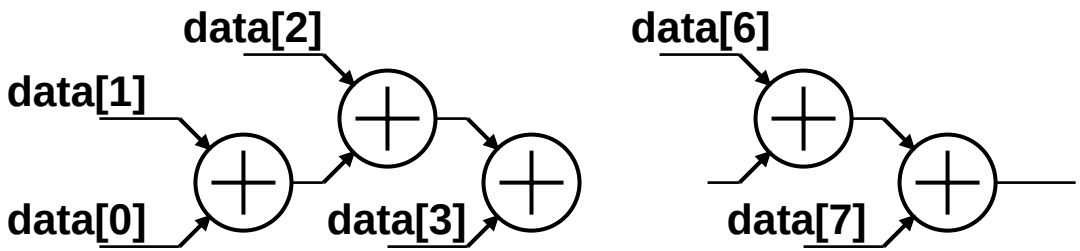
For-Loop Synthesis

- Loops are unrolled in implementation

```
always @(a or b) begin
  for (i = 0 ; i <= 3; i = i + 1)
    out[i] = a[i] & b[3-i];
end
```

  - Results in 4 AND gates
- Dependencies are maintained (parity checker for data[])

```
sum = 0;
for (i = 0 ; i < 8; i = i + 1)
    sum = sum + data[i];
odd_parity = sum[0];
```

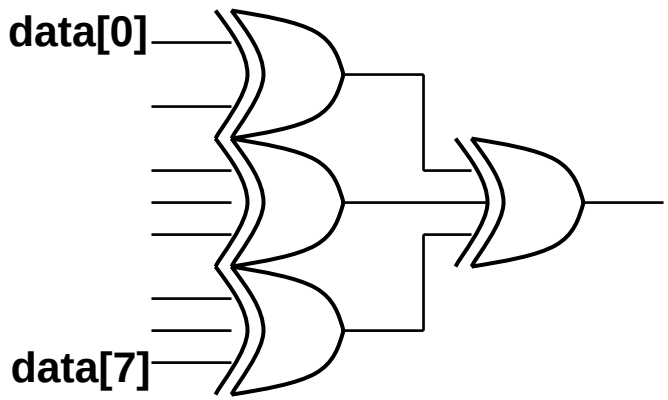


Example: Re-Coding the Parity

- Use XORs instead for the bit-wise operation

```
always @(data) begin
  odd_parity = ^data;
end
```

- Synopsys will optimize or you can structure it by hand



Local Optimization with Verilog Syntax

- Watch ordering of variables and parenthesis
  - How you order your computation will set the actually logic
    - a + b + c + d implies 3 serial adders
    - (a+b) + (c+d) implies a tree adder
  - Need to make sure you choose which you want.
- Slow Input - if you have an input signal that comes in late
  - Write your logic so that this signal has little logic in its path
    - If *AEqualB* is late
    - TakeBranch = (bunch of logic) & AEqualB;
  - The optimization will generally do this for you, but sometimes the signals cross partition boundaries, and reordering the variable becomes harder to do
- Local changes can help the synthesis but it can also be eliminated by synthesis. It depends on the level of “optimization”

Synthesis Summary

- Synthesis tools are like compilers
  - Allow the user to work at a higher level
  - Show you what the details look like (maybe)
- Use tools to understand the parts that need extra work
  - Like a profile of a program
  - Optimize the parts that don't meet the constraints
  - Don't improve what is not broken
- The role of tools are partitioned: need to iterate within each partition.
- Tools leverage your creativity
  - Not a substitute for thinking
  - Need to compete with others using the same tools!