Old Exam Question

What is the maximum operating frequency for this circuit?
Assume the following delay characteristics:
XOR intrinsic delay:  4 ns
AND intrinsic delay:  3 ns
Flipflop clock to Q delay: 8 ns
Flipflop setup time:  2 ns
Fanout delay, one destination: 1 ns
Fanout delay, two destinations: 2 ns
Output loading delay: 10 ns
Parallel Registers

Setup Time ($t_{su}$) is the amount of time before a clock edge that data needs to be stable.

Clock to Q delay ($t_{pd}$) is how long it takes from a clock edge until data are available at the Q outputs.
Serial Registers
F_{\text{max}} = 1/27\text{ns} \approx 37 \text{ MHz}
Setup, Hold and Critical Path Analysis

• Critical (longest) path analysis is central to circuit development.
• Writing Verilog is NOT programming. It is circuit development.
• Register setup time is part of the longest path.
• Register hold time is not.
Setup and Hold

Setup Time $t_{su}$ is the amount of time before the clock edge that data need to be stable.

Hold time $t_{hd}$ is the amount of time after the edge that data need to remain stable.
Setup and Hold

• Setup and hold times are characteristics of a particular device (register).
• They are not a function of any combinational logic that may be driving or being driven by the register.
Critical Path

Critical path is the path that limits a circuits operating speed.

It is not necessarily the longest path, as some paths do not need to operate on every clock cycle (multicycle paths).
Multicycle Path

Hardware multiplier may take a dozen or so cycles.

State decode needs to run in one cycle: may be the critical path, even though it is much shorter than multiplier delay.
Critical Path

Critical path limit is how much stuff can be done between active (usually rising) clock edges.

Data must be stable at FF1 input at the start. This means that the measurement starts AFTER the setup time of FF1 has passed.
Critical Path

Critical path summation starts at one rising (active) edge of a clock. When the clock edge arrives, the flipflop starts changing. It finishes changing after its Clock to Q delay. At that point, a new value is available at Q1.

Then the signal propagates through whatever combinational logic is in the path to the next flipflop.

Finally the new signal arrives at the second flipflop. But that flipflop has a Setup time. Thus the signal must arrive enough before the next edge that all the propagation delay (Clock to Q, combinational logic PLUS setup all fit into one clock period.)
Critical Path

Thus a path between registers consists of

– 1. Clock to Q delay of source register
– 2. Propagation delay of combinational devices and wiring in the path
– 3. Setup time (internal propagation delay of destination register)
Hold Time

Registers also have a thing called Hold Time.

Why doesn’t this figure into the critical path?
Hold Time

• Hold time is the amount of time a signal must remain stable AFTER the active clock edge.

• Hold time is never a problem on long paths. If a signal spends most of a clock cycle just getting to the next register on time, it will certainly still be there when the clock arrives.
Hold Time

Now take the case of a shift register: no logic between stages.
Assume short Clock to Q delay—say 1 femtosecond.
Also assume 1 fs wire delay
Because of Clock Skew, clock arrives at FF2 1 ns after FF1.
Result: Hold Violation. Data from FF1 are gone before they can be latched into FF2.
Hold Time

• Because of internal clock line delay of a flipflop, there can be a hold violation even if there is zero or negative clock skew.

• When the data path between registers is not devoid of logic gates, there will almost always be enough delay to slow down the data path so that the clock path will be shorter. Thus no hold violation.

• Hold time is not a factor in calculating path length. It does not add anything at all to data delay path.
Latency

- Latency is the amount of time a piece of data takes to traverse the entire design unit.
- It is usually measured in clock cycles.
- Designers may trade off latency in clock cycles for clock period: to get a piece of data through a unit (ALU, for example) adding pipelining increases latency but allows a faster clock to be used.
Operating frequency can be increased by adding register stages (increasing latency) between adders. Wallace trees are used for multi-operand addition, which can be part of a hardware multiplier.
Lab 6

- Lab 6 calls for setting delays with Specify blocks.
- This is not the normal and accepted use of Specify blocks.
- Their proper usage will be covered later, in the modeling section.
- Ignore delay specifications.
Lab 6, Supply 0 and 1

• Do not use supply0 and supply1 nets.
• Instead, simply set the inputs to 1'b0 or 1'b1, as appropriate.
Lab 6 Clarification

• In Lab 6, there *can* be a carry in to the LSB bit position.
• The C0 input should not be tied low.
• There is no mux on the second bit because even though it does need to wait for the first to generate a carry the output would not be speeded up by pre-calculating two sums and then selecting one.
Conditional Compilation

- Conditional compilation uses compiler directives `ifdef, `ifndef, `elsif and `endif.
- They all work with `define directives.
- Tells the Verilog compiler to skip or not skip portions of your code.
- `ifndef is a Verilog 2001 addition which is supported by NC Verilog.
ifdef Example

`ifdef RTL
    wire y = a & b;
else
    `else
    and #1 (y,a,b);
endif
ifdef Example

define TEST_MODE
 ifdef TEST_MODE
 tb_with_error tb_csa();
 else
 tb_csa tb_csa();
 endif
Force and Release

• force and release may be used in test benches to change values of registers and nets in a design.

• They may NEVER be used in synthesizable code.

• Since they will be used to drive values in a design under test from a test bench, hierarchical names must be used.
module DFF(DATA_IN, DATA_OUT, CLK);
  input DATA_IN, CLK;
  output DATA_OUT;
  reg DATA_OUT;
  always @(posedge CLK) begin
    DATA_OUT <= DATA_IN;
  end
endmodule
Parameterized DFF Model

//Bit 0 down to 0? Is that legal?
module DFF(DATA_IN, DATA_OUT, CLK);
    parameter WIDTH = 1;
    input CLK;
    input [WIDTH – 1 : 0] DATA_IN;
    output [WIDTH – 1 : 0] DATA_OUT;
    reg [WIDTH – 1 : 0] DATA_OUT;
    always @(posedge CLK) begin
        DATA_OUT <= DATA_IN;
    end
endmodule
Force Register Demo

```
`timescale 1 ns / 1 ns
module forcedemo();

reg DATA_IN, CLK;
wire DATA_OUT;
DFF UUT(DATA_IN, DATA_OUT, CLK);

initial begin
    CLK = 1'b0;
    forever #10 CLK = ~CLK; //rising edges on odd 10's of ns.
end

initial begin
    DATA_IN = 1'b1;
    #35 force UUT.DATA_OUT = 1'b0;
    #30 release UUT.DATA_OUT;
end
endmodule
```
Force Flipflop

CLK

Data In

Data Out

Release is here

Force in middle of cycle, asynchronous

Flipflop does not re-evaluate until rising edge
Combinational Adder, Wire and Reg

module FA2(A, B, CIN, S2, COUT2);
  input A, B, CIN;
  output S2, COUT2;
  wire S2, COUT2;
  assign S2 = A ^ B ^ CIN;
  assign COUT2 = (A&B) | (A&CIN) | (B&CIN);
endmodule

module FA1(A, B, CIN, S1, COUT1);
  input A, B, CIN;
  output S1, COUT1;
  reg S1, COUT1;
  
  always @(A or B or CIN) begin
    S1 = A ^ B ^ CIN;
    COUT1 = (A&B) | (A&CIN) | (B&CIN);
  end
endmodule
initial begin
    A = 1'b1; B = 1'b1; CIN = 1'b0;
    #35 force FA1.S1 = 1'b1; force FA2.S2 = 1'b1;
    #30 release FA1.S1; release FA2.S2;
end
Force/Release demonstration: Continuous assignment resumes normal value \((t = 65)\). Always block (register variables) does not because its inputs do not change. Flipflop resumes normal operation at clock edge following release \((t = 70)\).
Text is Misleading

• Text page 90, point 2: “The assignment expression is evaluated as soon as one of the right-hand side operands changes and the value is assigned to the left-hand-side net.

• This is WRONG. Continuous assignments are continuously evaluated and will respond immediately to a release after a force.
Looping Statements

• Verilog has four types of looping constructs:
  – while, for, repeat and forever
• All such constructs can only be used inside of an *initial* or *always* block.
While Loop

• A *while* loop executes a statement (or block of statements) as long as its expression is true.

• If the expression start out false, the statements are not executed.
while Example

reg [7:0] tempreg;
reg [3:0] count;
initial ..... 
begin 
  count = 0;
  ..... 
  while (tempreg)
    begin
      if (tempreg[0]) count = count + 1;
      tempreg = tempreg >> 1; // Shift right
    end
end
end
What is the Test Condition?

• Previous example tests tempreg.
• While (tempreg): while tempreg is true.
• What does that mean? Tempreg is eight bits. How can it be true or false?
• Guess?
• Or maybe test (simulate) and know for sure.
module tb_whileloop();
    reg [7:0] tempreg;
    reg [3:0] count;
    integer loopcount;
    initial
    begin
        count = 0;
        tempreg = 8'bx;
        loopcount = 0;
        while (tempreg)
            begin
                #10 //delay to show running counts
                if (tempreg[0]) count = count + 1;
                tempreg = tempreg >> 1; // Shift right
                loopcount = loopcount + 1;
                $display("tempreg = %b, count = %d, loopcount = %d", tempreg, count, loopcount);
            end
        end
endmodule
Loop is Never Entered

- Tempreg stays 8’bx forever.
- Count and Loopcount are never incremented.
module tb_whileloop();
  reg [7:0] tempreg;
  reg [3:0] count;
  integer loopcount;
  initial
    begin
      count = 0;
      tempreg = 8'b1xxxxxxx;
      loopcount = 0;
      while (tempreg)
        begin
          #10 //delay to show running counts
          if (tempreg[0]) count = count + 1;
          tempreg = tempreg >> 1; // Shift right
          loopcount = loopcount + 1;
          $display("tempreg = %b, count = %d, loopcount =
          %d", tempreg, count, loopcount);
        end
    end
endmodule
Tempreg Initialized to 1xxxxxxxx

run 100 ns

# tempreg = 01xxxxxxx, count = 0, loopcount = 1
# tempreg = 001xxxxx, count = 0, loopcount = 2
# tempreg = 0001xxxx, count = 0, loopcount = 3
# tempreg = 00001xxx, count = 0, loopcount = 4
# tempreg = 000001xx, count = 0, loopcount = 5
# tempreg = 0000001x, count = 0, loopcount = 6
# tempreg = 00000001, count = 0, loopcount = 7
# tempreg = 00000000, count = 1, loopcount = 8
Tempreg Initialized to xxxxxxx1

run
# tempreg = 0xxxxxxx, count = 1, loopcount = 1
Conclusion

• While means while at least one bit of the control is equal to 1.
• It does not mean while all bits are not equal to 0.
Corollary

• While (!Variable) means while each and every bit is 0.
• It does not mean ambiguous (i.e. X).
• Examples that will not enter the loop:
  – tempreg = 8’b10000000;
  – tempreg = 8’xxxxxxxxx;
  – tempreg = 8’0000000x;
While Loop

*while* loops execute until the condition is no longer true.

```
while (CNT < MAX_CNT) begin
    $display ("CNT = %d", count);
    CNT = CNT + 1;
    force uut.MEM[CNT] = CNT;
    release uut.MEM[CNT];
end
```

If CNT is not less than MAX_CNT when the line is first hit, the loop will not be entered at all.
While Loops in Test Fixtures

While loops are rarely useful in synthesizable code but are often used in test benches.

```plaintext
integer i, j; //need to initialize i and j: not shown on slide.
while (i <= MAX_OUTER) begin
    //set one set of parameters for the outer loop
    while (j <= MAX_INNER) begin
        //do all sorts of tricky test stuff
        j = j + 1;
    end
    i = i + 1;
end
```
Integers in Synthesizable Code

• Previous example shows integers being used as loop control variables.
• OK for both test fixtures and circuit descriptions.
• Integers are rarely used as design objects because their definition is vague: at least 32 bits, but may be more. Use regs of a specific width instead.
do...while Loop

- do while loops do bottom testing, so they are always entered at least once.
- Verilog DOES NOT SUPPORT do while loops.
- SystemVerilog superset of Verilog does.
- Implication: variable initialization needs to be done outside of while loops in case a loop will never be entered.
For Loop

The **for** loop is equivalent to the following:

```plaintext
<initialization>;
while (<condition>);
    begin
        loop ;
        <operation>;
    end
```
For Loop

Example 1: Check for some (error) condition

```plaintext
for (index = 0; index < size; index = index + 1)
    if (val[index] === 1'b1)
        $display("found an x");
```

Example 2: Initialize a memory

```plaintext
for (i = size - 1; i >= 0; i = i - 1)
    memory[i] = 0;
```
Useful ‘for’ Loop?

reg [2:0] i; //note use of reg for loop count variable, not integer.

reg[3:0] out;

always @(a or b) begin

//is this mixing blocking & non-blocking assignment operators?
    for (i = 0; i <= 3; i = i + 1)
        out[i] = a[i] & b[i];
end
Useful? Not Really

It’s the same as:

    wire [3:0] a, b, out;
    assign out = a & b;

Will the previous one synthesize the three-bit index register?
Another ‘for’ Loop

always @(posedge CLK or negedge RESET) begin
    if (!RESET) count <= 0;
    else begin
        count <= 0; temp <= data;
        //Cycling through all index values takes place in just one clock cycle
        for (index = 0; index < width; index = index + 1) begin
            count <= count + temp[0];
            temp <= temp >> 1;
        end
    end
end
More Efficient ‘for’ Loop?

always @(posedge CLK or negedge RESET) begin
  if (!RESET) count <= 0;
  else begin
    count <= 0; temp <= data;
    //only execute loop if at least one bit remains set
    for (index = 0; |temp; index = index + 1) begin
      if (temp[0]) count <= count + 1;
      temp <= temp >> 1;
    end
  end
end
Dynamic Loop Control

- Previous example uses a dynamic loop control.
- The loop can not be “unrolled” by the compiler.
- Therefore a synthesizer will not be able to generate logic for it.
- The second example is NOT synthesizable because loop length is data dependent.
For Loop

Example: tricky stuff that is legal Verilog.
Advisable?

```verilog
reg [7:0] A,B,C,D;
initial
begin
    A = 12;       //if A== 0 the loop will not be executed
    B = 2;
    C = 0;
    for ( D = B; A; C = D)
        begin
            D = D + C;
            A = A - B;
        end
end
```
Verilog vs. C “for” Loops

• Syntactically very similar EXCEPT:
  – Verilog does not support ++/-- operations
• i.e.
  for (i = 0; i < LIMIT; i++) in C becomes
  for (i = 0; i < LIMIT; i = i + 1) in Verilog
Loop Tip

• Minimize the logic in a loop.
  – All logic in the loop will be generated for each loop iteration.

• Some thought may show that operations do not need to be done each time.

• Remember to always “Think Hardware.”
Loop Logic Minimization Example

/*This one will make 8 adders*/
DONE = 1’b0;
ADDR = BASE_ADDR;
for (I=1;I<=8;I=I+1)
    if (IRQ(I) & !DONE) begin
        ADDR = ADDR + OFFSET[I];
        DONE = 1’b1;
    end

/*This one will make only one adder*/
TEMP_OFFSET = 5’d0;
for (I=1;I<=8;I=I+1)
    if (IRQ[I])
        TEMP_OFFSET = OFFSET[I];

ADDR = BASE_ADDR +
TEMP_OFFSET;
Forever Loops

• Loops forever, or until a $stop or $finish task is encountered.
• Primarily used for generating clocks.
• Can only be used within an initial or always block.
• A forever loop must be the last statement in a block, as anything after one will never run.
forever Loop

• A forever loop executes a statement (or block of statements) until the simulation ends.

• A forever loop should be the last item in a procedural block. No statements should follow it.
Forever Example

//A 50 ns clock oscillator that starts after 1000 nanoseconds
`timescale 1 ns / 1 ns
(module statement, etc…..)
initial begin
  clk = 1’b0;
  #1000 forever #25 clk = ~clk;
end

This is a closed loop. Any code in the initial block following the loop will never run.
Faster Loop?

```verilog
reg clk;

initial begin
  clk = 1'b0;
  forever begin
    #10 clk = 1'b1;
    #10 clk = 1'b0;
  end
end
```

This construct may take less simulation time, as it does not require the simulator to evaluate and negate the current state of clk.
repeat Loops

- Repeat loops execute a fixed number of times.
- The terminating condition must be fixed at the start of the loop. It will not be re-evaluated during execution.
- Must be used only within initial or always blocks.
Repeat Example

initial begin

    repeat (128) begin

        $display(“Count = %d”, CNT);

        CNT = CNT + 1;

    end

end
Another Repeat Example

initial begin

    CNT = 4;
    repeat (CNT)
        CNT = CNT + 1;

end

How many times will this loop run?
Regs and Wires

• Quick and dirty rule of thumb:
  – If a signal is assigned a value inside an ‘initial’ or an ‘always’ block, it must be a reg.
  – Otherwise it is a wire.

• Good enough for almost all Verilog design, not very satisfactory from a theoretical standpoint.
Major Data Types

• Nets
  – Represent physical connection between devices.

• Registers
  – Originally intended to represent storage devices. Language has evolved.

• Parameters
  – Used to declare run time constants.
Default Logic Type

• Undeclared variables default to single bit wires.
• This can be changed with a directive
  – `default_nettype <type>`
• Some recommend setting no default, so every variable must be declared:
  – `default_nettype none`
• Must be set outside of any module.
Nets

- Nets are continuously driven by the devices that drive them.
- Verilog automatically propagates a new value onto a net when the drivers of the net change value.
Types of Nets

- Nets have different types to model design-specific and technology-specific functionality.

<table>
<thead>
<tr>
<th>Net Type</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire, tri</td>
<td>For standard interconnection wires</td>
</tr>
<tr>
<td></td>
<td>(it is also the default for undeclared nets)</td>
</tr>
<tr>
<td>wor, triior</td>
<td>For multiple drivers that are Wired-ORed</td>
</tr>
<tr>
<td></td>
<td>(for ECL technology)</td>
</tr>
<tr>
<td>wand, triand</td>
<td>For multiple drivers that are Wired-ANDed</td>
</tr>
<tr>
<td></td>
<td>(for open collector technology)</td>
</tr>
<tr>
<td>trireg</td>
<td>for nets with capacitive storage</td>
</tr>
<tr>
<td>tri1</td>
<td>For nets which pull up when not driven</td>
</tr>
<tr>
<td>tri0</td>
<td>For nets which pull down when not driven</td>
</tr>
<tr>
<td>supply1</td>
<td>For power</td>
</tr>
<tr>
<td>supply0</td>
<td>For ground</td>
</tr>
</tbody>
</table>
Types of Nets

- wire is used for single driver connections while tri is used for connections that have multiple sources. The difference is mainly for documentation purposes.

- There is nothing in Verilog that prevents a wire from being assigned high impedance.

  ```verilog
  wire [15:0] tribus;
  assign tribus = 16’hz;
  ```
Nets Declaration

- Net declaration format:

  `<net type> <range>? <delay_spec>? <<net_name> <,net_name>*>`

  `<net type>` is the type of the net.

  `<range>` is the vector range [msb:lsb].

  `<delay_spec>` specifies the delay associated with the net.

  `<net_name>` is the net name. It could also be a list of names.

Examples:

  `tri [15:0] busb; // A 16-bit tri-state bus`
  `wire [0:31] w1, w2; // Two 32-bit wires with msb labeled 0`
Continuous Assignments

• A continuous assignment is used to drive a value onto a net.

• A continuous assignment replaces gates in modeling combinational logic circuits and describes it at a higher level of abstraction.

• A continuous assignment statement starts with the keyword `assign`. The `assign` statement has the following syntax:

\[
\text{assign } <\#\text{delay}>?<\text{strength}>?<\text{net_name}>=<\text{expression}>
\]

• continuous assignments are outside of procedural blocks.
Continuous Assignments

• The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register when used outside of an initial or always block.

• Continuous assignments are always active. The assignment expression is evaluated, as soon as one of the right-hand-side operands changes, and its value is assigned to the left-hand-side net.

• The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
Continuous Assignments

• Example 1:
  
  wire [2:0] c, a;
  assign #6 c = {a[0], r1, r2};

• Example 2:

  wire and_net, or_net, a1, a2, b1, b2;
  assign #3 and_net = a1 & a2; // Data Flow
  and #3 (and_net, a1, a2); // Structural
  assign #3 or_net = b1 | b2; // Data Flow
  or #3 (or_net, b1, b2); // Structural

  The above two statements could be combined together:
  assign #3 and_net = a1 & a2, or_net = b1 | b2;
Continuous Assignments

Example3:
wire c_out, c_in;
wire [3:0] sum, a, b;
assign \{c_out, sum[3:0]\} = a[3:0] + b[3:0] + c_in;

Example4:
reg a4, a3, a2, a1;
wire [4:0] f;
assign f[4:2] = \{a4, a3, a2\};
Implicit Continuous Assignments

- Continuous assignment could also be done implicitly as part of the net declarations.

- Example 1:

  wire #8 and_net = a1 & a2, or_net = b1 | b2;

  This equivalent to the following two statements:

  wire and_net, or_net;

  assign #8 and_net = a1 & a2, or_net = b1 | b2;
Implicit Continuous Assignments

Example 1:

wire [7:0] buf_out = reg1;

wire #7 even = ^buffer; odd = ~^buffer;
Logic Conflict Resolution with Net Data Types

### Wire/tri

<table>
<thead>
<tr>
<th>a \ b</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>z</td>
</tr>
</tbody>
</table>

### Wand/triand

<table>
<thead>
<tr>
<th>a \ b</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>z</td>
</tr>
</tbody>
</table>

### Wor/trior

<table>
<thead>
<tr>
<th>a \ b</th>
<th>0</th>
<th>1</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>z</td>
</tr>
</tbody>
</table>
Wired OR

- Wired-OR logic was the way emitter-coupled logic (ECL) worked.
- Obsolete now: advantage was speed, but CMOS has surpassed it.
- Poor noise immunity, high power consumption and non-standard voltage supplies made ECL unattractive.
Wired AND

- Wired AND is still useful.
- Not used inside of ICs, but is used on the periphery (I/O cells).
- Allows a variable number of devices to be plugged into one line.
- TTL open collector, CMOS open drain are wired AND logic.
If any interrupt request goes high, the processor input is dragged low: wired AND.

Open-collector gates, 74LS05 or similar
’05 Open-Collector Output
Open Collector, Open Drain

- Open collector (TTL) and open drain (MOS) gates may have their outputs shorted together.
- Same technique would cause catastrophic failure with normal gates.
- HC12 includes some open drain outputs on the serial communications link. Other pins may not be connected directly to multiple drivers.
Data Type Reg

• Regs are used in behavioral modeling and in generating stimuli.
• Regs are assigned values using behavioral constructs.
• Regs hold their values until new values are assigned.
• Misleading name: reg does not imply register.
Holding a Value

- Regs *do* hold a value until updated.
- This does not imply that a flipflop will be synthesized for every variable of type reg.
- Properly-written Verilog code will have all reg values updated as needed to prevent memory effect from causing simulation/synthesis mismatches.
Reg cout, sum;
always @(a or cin)
    {cout, sum} = a + b + cin;

This will ignore any changes on b:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>cin</th>
<th>cout</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Incomplete Sensitivity List

• Incompletely-specified sensitivity list would appear to cause memory (latched values).
• This is not the case. Combinational sensitivity lists are ignored by synthesizers.
• Flipflops are inferred from edge constructs in sensitivity lists.
• Level-sensitive latches will be dealt with shortly.
* In Sensitivity Lists

Instead of putting all the right-hand side variables of combinational blocks into a sensitivity list, you can just put a * there.

//Note dual use of * symbol
always @(*) PROD = A * B * C * D;
always @(*) {cout, sum} = A + B + C;
Continuous Assignment Registers

• Not strictly illegal.
• Poor technique. Don’t do it.

reg BADSTYLE;
always @(A or B) assign BADSTYLE = A & B ;
Reg vs. Wire

- Wires must be continuously driven to maintain a value.
- Regs hold previous value until something causes them to be re-evaluated.
- Names are archaic: an artifact of how the language grew. Verilog started out as a verification tool. Synthesis came later.
- **REG DOES NOT IMPLY FLIPFLOP (REGISTER).**
## Registers Types

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>Unsigned integer variable of varying bit width. It is closely associated with hardware.</td>
</tr>
<tr>
<td>integer</td>
<td>Signed integer variable, at least 32-bit wide. Stored as 32 signed 2’s complement number. Integers are used to manipulate quantities that are not regarded as hardware.</td>
</tr>
<tr>
<td>real</td>
<td>Signed floating-point variable, double precision.</td>
</tr>
<tr>
<td>realtime</td>
<td>Double-precision floating point variable for reporting simulation time.</td>
</tr>
<tr>
<td>time</td>
<td>Unsigned integer variable, 64-bit wide. It is used to store and manipulate simulation time quantities.</td>
</tr>
</tbody>
</table>
Registers Declaration

Register declaration format:

\[ <\text{register\_type}> <\text{range}>? <<\text{register\_name}>,\text{register\_name}>* > \]

where:

- \(<\text{register\_type}>\) Is the type of register.
- \(<\text{range}>\) Is the vector range. It is legal for \text{reg} type only not for integer, real or time.
- \(<\text{register\_name}>\) Is the register name or list of names.

Examples:

- \text{reg} [3:0] \text{R1}; \ // \ A 4-bit register from msb to lsb.
- \text{reg} [15:0] \text{R1, R2}; \ // \ Two 16-bit registers
Signed Registers

- Verilog includes a signed register data type:
  - `reg signed [63:0] PRODUCT;`
- Integers are also signed registers:
  - `integer i; /* usually 32-bit signed value, implementation-dependent */`
- Signed registers must be used when preserving sign bit (arithmetic shift). Integers should be avoided in synthesizable code, as size is not defined.
Vector Registers

• It just means multi-bit registers.
  – reg [15:0] data_bus;

• Vector part select
  – Normally done with explicit bit selects:
    • data_bus[15:8];
    • SIGN_BIT <= SUM[7];
    • Variable Vector Part Select is also part of the language.
Variable Part Select

• Highly-unusual construct, but legal.
  – HIGH_BYTE <= DATA_BUS[31-:8];
  – Takes bits 31:24 and sends them to variable HIGH_BYTE

• Can use variables in the selector:
  – index <= index – 1;
  – HIGH_BYTE <= BUS[(index*8)+:8] <= 8’b0;
Part Select Gotcha

• Part select uses an operator (+ or -) before the separator (:

• If the operator is inadvertently placed after the operator, it becomes a unary operator.
  – Take bits 23 to 31
  – HIGH_BYTE <= DATA_BUS[23+:8];
  – Take bits 23 through 8
  – HIGH_BYTE <= DATA_BUS[23:+8];
Arrays

• Arrays are allowed in Verilog for all standard data types. (Pre-Verilog 2001 only register type variables were allowed to be arrays).

• Arrays of vectors can also be created.

• Array elements are accessed by:

  \(<\text{array\_name}> [\text{<subscript>}]\)
Array of registers

- In Verilog the syntax for an array of registers is:
  
  ```verilog
  reg [msb:lsb] <register_name> [first_addr:last_addr];
  ```

- where:
  - `msb and lsb` determine the word size of the register.
  - `register_name` is the name of the register array
  - `first_addr and last_addr` determine the depth of the array
  - You can declare word size and register depth with any legal expression.
Arrays

Examples:

- integer count [0:7]; // An array of 8 integer variables (32-bit wide)
- reg bool[31:0] ; // Array of 32 one-bit boolean register variables
- time chk_pt [1: 100 ];  // Array of 100 time checkpoint variables
  // 64-bit wide
- reg [4:0]port_id[0:7]; //Array of 8 port_ids; each port_id is 5 bits

Examples of accessing array elements :

- chk_pt[100]  //100th time check point value
- port_id[3] //4th element of port_id array. This is a 5-bit
module tribuf8bit (out, in, enable);
output [7:0] y;
input [7:0] a;
input en; wire [7:0] y, a;
wire en;
//array of 8 Verilog tri-state primitives each bit of the
//vectors is connected to a different primitive instance
bufif1 u[7:0] (y, a, en);
endmodule
module tribuf64bit (out, in, enable);
output [63:0] out;
input [63:0] in;
input enable;
wire [63:0] out, in;
wire enable;
/*array of 8 8-bit tri-state buffers; each instance is connected to 8-bit part selects of the 64-bit vectors; The scalar enable line is connected to all instances.*/
tribuf8bit i[7:0] (out, in, enable);
endmodule
Arrays & Vectors

• A vector is a single element that is n-bits wide. Arrays are multiple elements that are each one or more bits wide.

• Vector:
  – reg [7:0] DATA_BUS;//an eight-bit vector

• Array:
  – //an array of eight single-bit variables.
  – reg boolean [7:0];
Memory Declaration

• In Verilog, a memory is modeled by an array of registers.

Examples:

• reg [15:0] MEM [0:1023]; // 16-bit x 1K memory array

• reg [7:0] PREP ['hFFFE:'hFFFF]; // 8 x 2 memory array

• reg [0:wordsize-1] MEM3 [memsize-1:0]; // Memory array declared using parameters
Memory Addressing

- A memory element is addressed by giving the location as the address to a previously defined memory array.

- To address a part of a memory word, store the memory location addressed in a temporary register of the correct width.
Memory Addressing

• reg [8:1] memory [0:255]; // declare memory called memory

• reg [8:1] mem_word; // temp register called mem-word

To display contents of the 6th memory location:

• $displayb (memory[5]);

To display the msb of the 6th memory word:

• mem_word = memory[5];
• $displayb (mem_word[8]);
Multi-Dimensional Arrays

• New version of Verilog supports multi-dimensional arrays.

• A cubic memory could be declared:
  – reg [3:0] MEM [7:0] [7:0];
  – 8 x 8 array of four bit wide data

• Cubic memory usage:
  – always @(posedge WR) MEM[A][B] = DIN;
  – always @(A or B) DOUT = MEM[A][B];
Array Partial Select

Verilog 2001 adds an array partial select.

The assignment below to out2 takes the high byte of element 100/7 from the 32-bit 256 x 16 array array2.

This enhancement to the language may not be supported by all tools.

```verilog
//select the high-order byte of one word in a 2-dimensional array of 32-bit reg variables
reg [31:0] array2 [0:255][0:15];
wire [7:0] out2 = array2[100][7][31:24];
```
Array Partial Select, The Old Way

• Needs to be done in two commands:
  – First, select an array element
  – Second, select a piece of the element

• Example:
  – reg [7:0] MEM [63:0]; // 8 x 64 register file
  – reg [7:0] MY_BYTE:
  – reg MY_BIT;
  – MY_BYTE = MEM[35]; // Select array element #35
  – MY_BIT = MY_BYTE[5]; // Grab 6th bit.
Loading a Memory Array

There are several ways to load a memory. One way is to assign values to each word of the memory array. This method is useful when the value of a particular word of the memory needs to be changed or when all the words of the memory need to be initialized to the same value.
module scaleable_mem (address, data, ..);

  parameter addr_bits = 8;              //size of address bus
  parameter wordsize = 8;               //width of a word
  parameter memsize = (1 << addr_bits); //size of memory=2**8
  input [addr_bits -1:0] address;
  reg [wordsize - 1:0] memory [0:memsize -1]; //address bus
  reg [wordsize - 1:0] mem_word;          //memory declaration
  integer i;

  initial

    begin

      for ( i=0 ; i < memsize ; i = i+1 )
        memory [i] = 8'b0;          //initialize memory to all 0’s

      ... 

    end

  ... 

  initial

    mem_word = memory[address];       //read from a specific location

endmodule
Loading a Memory

Another way to load a memory is through Verilog systems tasks $readmemx, where x is the radix.

$readmemh(“memfile.txt”, UUT.MY_MEM);
$readmemb(“memfile.txt”, UUT.MY_MEM);

Only binary and hexadecimal radices are used.

This is for loading a test file into a RAM/ROM for simulation. It is not synthesizable.
Loading a Memory Array

Use $readmemb and $readmemh system tasks. A memory array can be initialized by writing into the memory the contents of a text file. The $readmemb and $readmemh system tasks read from a file and write into the memory.

$readmem<base>("<filename>", <mem-name>, <start>?, <finish>?);

where:
<base> is b if the values are binary numbers.
    is h if the values are hexadecimal numbers.
<filename> is the file from which the memory array is to be loaded.
<mem-name> is the name of the memory to be loaded.
<start> and <finish> are the addresses of the memory that are to be loaded. By default, <start> is the first address and <finish> is the last address of the memory array.
Loading a Memory Array

module scaleable_mem (address, data, ..);

    parameter addr_bits = 8;               //size of address bus
    parameter wordsize = 8;                //width of a word
    parameter memsize = (1 << addr_bits);  //size of memory=2**8
    input [addr_bits-1:0] address;         //address bus

    reg [wordsize-1:0] memory [0:memsize - 1];  //memory declaration
                                                // reg [7:0] memory [0:255];
    initial
    begin
        $readmemb("mem_file.txt", memory);    //load memory data from a file
    end
endmodule
Loading a Memory Array

reg [7:0] memory [0:255];
initial
begin
    $readmemb("mem_file.txt", memory, 16);   //Load up
    $readmemb("mem_file.txt", memory, 16, 128); // Load up towards 128
    $readmemb("mem_file.txt", memory, 128, 1);  //Load down toward 1
end
File Format for $readmemb and $readmemh

- The values are interpreted as binary or hexadecimal numbers depending on the system task being called.
- Use underscores ( _ ) to increase readability.
- Unknown value (x, X), and the high impedance (z, Z) can be used.
- Both single line and multiline comments are supported.
- You can assign a value to a particular address. When addresses are used in the data file, the format is:

  @<hex_address>

- The hexadecimal address is case **Insensitive**.
- No white space is allowed between the @ and the number.
- When the system task encounters an address specification, it loads subsequent data starting at that memory address.

**Note:** Verilog does not tell you if a nonexistent address is loaded.
**File Format for $readmemb and $readmemh**

```vhdl
$readmemb("mem_file.txt", memory);
```

**UNIX Text File**

`mem_file.txt`

- **0000_0000**
- **0110_0001 0011_0010**
  - //addresses 3-255 are not defined
- **@100**
- **1111_1100**
  - /* addresses 257-1022 are not defined */
- **@3FF**
- **1110_0010**

**Declared Memory Array**

```
reg [0:7] memory [0:1023]
```

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Value (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>01100001</td>
<td>1</td>
</tr>
<tr>
<td>00110010</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>11111100</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>11100010</td>
<td>1023</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

119
Unspecified Bits in $readmemb

Suppose this is the contents of a text file read with $readmemb:

0 00 0000 00000000
1 11 1111 11111111

What would be put into the memory?
module rdmem;
integer i;
reg [7:0] mem [7:0];
initial $readmemb("memfile.txt", mem);
initial for (i=0; i<8;i=i+1)
  $display("Address = %d Data = %b", i, mem[i]);
endmodule
Unspecified Bits in $readmemb

Address = 0 Data = 00000000
Address = 1 Data = 00000000
Address = 2 Data = 00000000
Address = 3 Data = 00000000
Address = 4 Data = 00000001
Address = 5 Data = 00000011
Address = 6 Data = 00001111
Address = 7 Data = 11111111
Modeling Bi-directional Ports

- The bidirectional port needs to be declared using the keyword `inout`.
- Inout ports follow these rules:
  - The `inout` port must be driven by a `net` and not a register.
  - An `inout` port must drive a `net` and not a register.
Modeling Bidirectional Ports

• Logic needs to be built around the *inout* port to ensure proper operation. Therefore, when the port is acting as an input port, the output logic is disabled and when it is acting as an output port, the input logic is disabled.

• *inout* ports follow the port connection rules. No stimulus can be applied to the inout port through a register data type, thus, the port needs to be buffered.

```
assign data = write ? data_reg : 8’bz;
```
A Structural Modeling of a Bidirectional Ports

module inout_port(io_port, rd_nwr);
    inout io_port;
    input rd_nwr;
    bufifl bl (io_port, data, rd_nwr); //output
    bufif0 b2 (data, io_port, rd_nwr); //input
.
endmodule
A Data flow Modeling of a Bidirectional Ports

module inout_port(io_port, rd_nwr);
    inout io_port;
    input rd_nwr;
    assign io_port = rd_nwr ? data: 'bz; //output
    assign data = rd_nwr ? 'bz : io_port; //input
.
endmodule
module inout_port(io_port, rd_nwr);
    inout [7:0] io_port;
    input rd_nwr;
    reg [7:0] out_reg;
    wire [7:0] io_port;
    assign io_port = rd_nwr ? out_reg: ‘bz; //read
    always @ (negedge rd_nwr)
        out_reg = io_port; //only a single write
        //is made for each
        //negedge rd_nwr
endmodule