Lecture 3 – System Timing and Case Studies
Summary: Verilog for Combinational Logic

- **Two alternatives:**
  - assign statement – simple logic equation
  - always block – allows complex program to describe function

- Each assign and always block compiles into a component
  - Combinational function with some inputs and outputs

- **All components operate in parallel, continuously**
  - If any input changes, the function is recomputed
  - This may change the output
  - Which will cause inputs of some components to change

- Just like a circuit made up of gates!
Verilog for

- for is similar to C
- for statement is executed at compile time
  - result is all that matters, not how result is calculated

```verilog
// simple encoder
module encode
    (input [7:0] A, // 8-bit input vector
     output reg [2:0] Y); // 3-bit encoded output
    integer i; // Temporary variables for program only
    reg [7:0] test;

    always @(*) begin
        test = 8b'00000001;
        Y = 3'bX;
        for (i = 0; i < 8; i = i + 1) begin
            if (A == test) Y = i;
            test = test << 1;
        end
    end
endmodule
```
Another Behavioral Example

- Combinational block that computes Conway’s Game of Life rule

```verilog
module life
  (input      self,
   input [7:0] neighbors,
   output reg  out);
  integer      count;
  integer      i;

  always @(*) begin
    count = 0;
    for (i = 0; i<8; i = i+1) count = count + neighbors[i];
    out = 0;
    out = out | (count == 3);
    out = out | ((self == 1) & (count == 2));
  end
endmodule
```
Case Study – Division by Constant

- e.g. gray = (red + blue + green)/3

- Division is very expensive in general
  - Area and Delay
  - Much more so than multiplication

- Convert division to multiplication
  - Multiply by the reciprocal
  - e.g. (red + blue + green) * 0.33
  - Floating-point??
    - Also expensive!

- Key idea: multiply & divide by $2^n$ is FREE
RGB to Grayscale

- $Y = 0.3R + 0.59G + 0.11B$
- $1024 \times 0.3 = 307.2$
- $1024 \times 0.59 = 604.16$
- $1024 \times 0.11 = 112.64$

- $Y = (307R + 604G + 113B) >> 10$;

- This works for multiplying/dividing with any number with fractions
  - Scale then re-scale
Converting Division to Multiplication

- Increase precision until it’s good enough
  - FPGA has 18x18 multipliers – almost free
- Division by a variable?
  - Table lookup of reciprocal
  - Does not scale to large numbers
  - Use iterative solutions
Creating a Table in Verilog

///////////  Sin Wave ROM Table /////////////
always @(*) begin
  case(SIN_Cont)
    0 : Sin_Out = 0 ;
    1 : Sin_Out = 4276 ;
    2 : Sin_Out = 8480 ;
    3 : Sin_Out = 12539 ;
    4 : Sin_Out = 16383 ;
    5 : Sin_Out = 19947 ;
    6 : Sin_Out = 23169 ;
    7 : Sin_Out = 25995 ;
    8 : Sin_Out = 28377 ;
    9 : Sin_Out = 30272 ;
   10 : Sin_Out = 31650 ;
   11 : Sin_Out = 32486 ;
   12 : Sin_Out = 32767 ;
   13 : Sin_Out = 32486 ;
   14 : Sin_Out = 31650 ;
   15 : Sin_Out = 30272 ;
   16 : Sin_Out = 28377 ;
   17 : Sin_Out = 25995 ;
   18 : Sin_Out = 23169 ;
   19 : Sin_Out = 19947 ;
   20 : Sin_Out = 16383 ;
    21 : Sin_Out = 12539 ;
    22 : Sin_Out = 8480 ;
    23 : Sin_Out = 4276 ;
    24 : Sin_Out = 0 ;
    25 : Sin_Out = 61259 ;
    26 : Sin_Out = 57056 ;
    default : Sin_Out = 0 ;
  endcase
end
Summary: Verilog for Sequential Logic

- **Use** `always @(posedge clk) blocks`
  - Executes only when clock “ticks” (rising edge)
  - All assignments are registered
    - Use `<=` (delayed assignment – implements parallel sample/hold)

- **Styles**
  - **One posedge clk block**
    - Computation and registers
    - Not general
  - **Separate register block and combinational logic block**
    - More general, more like hardware
    - Less intuitive
Shift Register Example

```verilog
// 4 register shift register
module shiftReg
    (input    CLK,
     input    reset,    // initialize registers
     input    shift,
     input    [7:0] Din,  // Data input for load
     output   [7:0] Dout);
    reg    [7:0] D0, D1, D2, D3;
    assign Dout = D0;
    always @(posedge CLK) begin
        if (reset) begin
            D0 <= 0; D1 <= 0; D2 <= 0; D3 <= 0;
        end else if (shift) begin
            D3 <= Din; D2 <= D3; D1 <= D2; D0 <= D1;
        end
    end
endmodule    // shiftReg
```
FIR Filter Example

module fir
    (input    CLK,
     input    reset, // initialize registers
     input    [7:0] Din, // Data input for load
     output   reg [7:0] Dout);
    reg [7:0]   D0, D1, D2, D3;
    localparam C0 = 4, C1 = 3, C2 = 2, C3 = 1;

    always @(posedge CLK) begin
        if (reset) begin
            D0 <= 0; D1 <= 0; D2 <= 0; D3 <= 0;
        end else begin
            D3 <= Din; D2 <= D3; D1 <= D2; D0 <= D1;
            Dout <= C0 * D0 + C1 * D1 + C2 * D2 + C3 * D3;
        end
    end
endmodule // fir
Case Study: Wavelet Computation #1

localparam INIT=4, State0=0, State1=1, State2=2, State3=3;
reg [2:0] state; // State register

// This shift register captures the input
reg [7:0] in0, in1, in2, in3;
// These hold data while data is shifting in
reg [7:0] D0, D1, D2, D3;

// We use this shift register to delay the input valid bit by
// up to 8 clock cycles to generate the output valid bit
reg [8:0] valid;
assign validOut = valid[5]; // 5 cycles for pipe to fill

// Valid data enters as soon as reset is turned off
wire validIn = ~reset;

always @(posedge clk) begin
  if (reset) begin
    valid = 0;
    state <= INIT;
  end else begin
    // Constantly shift input data
    in0 <= in1;
in1 <= in2;
in2 <= in3;
in3 <= dataIn;
    valid <= { valid[7:0], validIn }
  endcase
end // case: state
end

end // case (state)
Block Diagram for Wavelet Filter
Wavelet with Split Block Style

localparam INIT=4, State0=0, State1=1, State2=2, State3=3;
reg [2:0] state, nextState;  // State register

// Registers
always @(posedge clk) begin
  if (reset) begin
    valid <= 0;
    state <= INIT;
  end else begin
    // Input shift register
    in0 <= in1;
in1 <= in2;
in2 <= in3;
in3 <= dataIn;
    valid <= { valid[7:0], validIn };  // Shift left
    if (state == State3) begin
      D0 <= in0;
      D1 <= in1;
      D2 <= in2;
      D3 <= in3;
    end
    state <= nextState;
  end // else: !if(reset)
end // always @(posedge clk)

// Combinational logic
always @(*) begin
  case (state)
    INIT: begin
      nextState = State0;
      end
    State0: begin
      dataOut = D0-D1;  // Q0
      nextState = State1;
      end
    State1: begin
      dataOut = D2-D3;  // Q1
      nextState = State2;
      end
    State2: begin
      dataOut = D0+D1+D2+D3;  // Q2
      nextState = State3;
      end
    State3: begin
      dataOut = D0+D1-D2-D3;  // Q3
      nextState = State0;
      end // case: State3
  endcase // case(state)
end

localparam INIT=4, State0=0, State1=1, State2=2, State3=3;
reg [2:0] state;        // State register
reg [9:0] D0D1, D2D3;   // Temp registers

// This shift register captures the input
//        oldest         newest
reg [7:0] in0, in1, in2, in3;

reg [8:0] valid;
assign validOut = valid[3];

// Valid data enters as soon as reset is turned off
wire validIn = ~reset;

always @ (posedge clk) begin
  if (reset) begin
    valid = 0;
    state <= INIT;
  end else begin
    in0 <= in1;
in1 <= in2;
in2 <= in3;
in3 <= dataIn;
valid <= { valid[7:0], validIn };
case (state)
  INIT: begin
    state <= State0;
  end
  State0: begin
    dataOut <= D0D1 + D2D3;  // Q2
    state <= State1;
  end
  State1: begin
    dataOut <= D0D1 - D2D3;  // Q3
    state <= State2;
  end
  State2: begin
    dataOut <= in1 - in2; // Q0: D0 - D1
    D0D1 <= in1 + in2;     // D0 + D1
    state <= State3;
  end
  State3: begin
    dataOut <= in2 - in3; // Q1: D2 - D3
    D2D3 <= in2 + in3;     // D2 + D3
    state <= State0;
  end // case: State3
endcase // case(state)
end
endmodule
Block Diagram for Complex Wavelet
module camera(
  input  clk, // clock
  input  reset, // reset
  input [11:0] CCD_DATA, // Input from camera
  input  CCD_FVAL, // In Frame signal
  input  CCD_LVAL, // In Line signal
  input  iSTART, // Start button
  input  iEND, // Stop button
  output [11:0] pixelOut, // Output pixel data
  output reg [15:0] colAddr, // Pixel address
  output reg [15:0] lineAddr, // Line address
  output newFrame, // Done with frame
  output pixelValid // Pixel valid
);

// Control signals
assign isNewPixel = CCD_FVAL && CCD_LVAL;
assign isNewFrame = fvalPrev && !CCD_FVAL;
assign isNewLine = lvalPrev && !CCD_LVAL;

assign pixelOut = CCD_DATA_L1;
assign pixelValid = isNewPixel && outputEnable;
assign newFrame = isNewFrame && outputEnable;

// Latch signals
always @(posedge clk) begin
  fvalPrev <= CCD_FVAL;
  lvalPrev <= CCD_LVAL;
  CCD_DATA_L0 <= CCD_DATA;
  CCD_DATA_L1 <= CCD_DATA_L0;
end

// Calculate addresses
always @(posedge clk) begin
  if (reset || isNewLine) colAddr <= 0;
  else if (isNewPixel) colAddr <= colAddr + 1;
  if (reset || isNewFrame) lineAddr <= 0;
  else if (isNewLine) lineAddr <= lineAddr + 1;
end

// Implement START/STOP picture
always @(posedge clk) begin
  if (reset || iSTART) outputEnableSignal <= 1;
  else if (iEND) outputEnableSignal <= 0;
  if (isNewFrame) outputEnable <= outputEnableSignal;
end

// Registers
reg fvalPrev;
reg lvalPrev;
reg [11:0] CCD_DATA_L0;
reg [11:0] CCD_DATA_L1;

// Control signals
reg outputEnableSignal;
reg outputEnable;
wire isNewPixel;
wire isNewFrame;
wire isNewLine;
Case Study: Camera Input

// Calculate addresses
always @(posedge clk) begin
  if (reset || isNewLine) colAddr <= 0;
  else if (isNewPixel) colAddr <= colAddr + 1;
  else if (isNewLine) colAddr <= colAddr - 1;
end

// Implement START/STOP picture
always @(posedge clk) begin
  if (reset || iSTART) outputEnableSignal <= 1;
  else if (iEND) outputEnableSignal <= 0;
  else if (isNewLine) outputEnable <= outputEnableSignal;
end
Case Study: Camera Input

// Control signals
assign isNewPixel = CCD_FVAL && CCD_LVAL;
assign isNewFrame = fvalPrev && !CCD_FVAL;
assign isNewLine = lvalPrev && !CCD_LVAL;

assign pixelOut = CCD_DATA_L1;
assign pixelValid = isNewPixel && outputEnable;
assign newFrame = isNewFrame && outputEnable;

// Latch signals
always @(posedge clk) begin
  fvalPrev <= CCD_FVAL;
  lvalPrev <= CCD_LVAL;
  CCD_DATA_L0 <= CCD_DATA;
  CCD_DATA_L1 <= CCD_DATA_L0;
end
Case Study: Camera Input

// Control signals
assign isNewPixel = CCD_FVAL && CCD_LVAL;
assign isNewFrame = fvalPrev && !CCD_FVAL;
assign isNewLine = lvalPrev && !CCD_LVAL;

assign pixelOut = CCD_DATA_L1;
assign pixelValid = isNewPixel && outputEnable;
assign newFrame = isNewFrame && outputEnable;

// Latch signals
always @(posedge clk) begin
    fvalPrev <= CCD_FVAL;
    lvalPrev <= CCD_LVAL;
    CCD_DATA_L0 <= CCD_DATA;
    CCD_DATA_L1 <= CCD_DATA_L0;
end
Design Problem – Run-Length Encoder

- 7-bit input stream
- 8-bit output stream
  - high-order bit == 0: Data value
  - high-order bit == 1: Repeat count of previous data value
- Valid bit set when 8-bit output is data or count
RLE Design

- Split design into datapath and control
- Datapath
  - Registers for data values, count
  - Multiplexors
- Control
  - Keep track of what’s happening
  - clear count, increment count, send data value, send count
- Control will be an FSM
Start with Datapath

- We need to know what to control

- FSM inputs
  - eq

- FSM outputs
  - clr, inc, valid, cnt,
FSM Controller
Verilog For State Machines

- State machine has two parts
  - State register
  - Combinational Logic
    - Next state function
    - Output function
- Each in a different always block
RLE Module – Datapath + Control

module rleFSM (clk, reset, eq, clr, inc, valid, cnttag);
  input clk, reset;
  input eq; // current data value == previous data value
  output clr; // clear count value (0 means 2, . . .)
  output inc; // increment count value (clr overrides)
  output valid; // output value is valid
  output cnttag; // select the count for the output value

  // Use parameter to define symbolic states
  localparam START1 = 0, START2 = 1, SENDING = 2, COUNTING = 3;
  reg [1:0] state, // current state
            nextState; // next state

  always @(posedge CLK) begin
    if (reset)
      state <= START1;
    else
      state <= nextState;
  end
always @(*) begin
    // Set defaults
    valid = 0; inc = x; clr = x; cnttag = x;
case (state)
    START1:
        nextState = START2;
    START2:
        nextState = SENDING;
    SENDING: begin
        valid = 1;
cnttag = 0;
    if (eq) begin
        nextState = COUNTING;
    end else begin
        nextState = SENDING;
    end
end
COUNTING: begin
    if (eq) begin
        clr = 0;
    inc = 1;
        nextState = COUNTING;
    end else begin
        valid = 1;
cnttag = 1;
        nextState = SENDING;
    end
end
Combinational Logic Design

- We can translate a Boolean function into logic gates
  - AND, OR, INVERT

- e.g. Homework problem
  - \( g_0 = r_0 \)
  - \( g_1 = r_1 \cdot r_0' \)
  - \( g_2 = r_2 \cdot r_0' \cdot r_1' \)
  - \( g_2 = r_3 \cdot r_0' \cdot r_1' \cdot r_2' \)
  - \( g_2 = r_4 \cdot r_0' \cdot r_1' \cdot r_2' \cdot r_3' \)
  - \( g_2 = r_5 \cdot r_0' \cdot r_1' \cdot r_2' \cdot r_3' \cdot r_4 \)

- This is a “scan” computation from parallel computing
Reduce and Scan Operators

- **Reduce**: Reduce a vector of values to a single value
  - e.g. sum, max, min, ...

- **Scan**: Convert a vector of values to a new vector
  - output$_k$ is reduction applied to the first k inputs
  - e.g. sum, max, min, ...

- Reduce can be done in $O(\log n)$ time
  - Simple n-ary tree

- Scan can also be done in $O(\log n)$ time
  - Double the time for Reduce
  - A “down” tree and an “up” tree
A Real Example of Scan

- Histogram equalization – contrast enhancement
  - (see Wikipedia)

- Step 1 – Create a histogram of all pixel values
  - Bin 0 – 255

- Step 2 – Compute the Cumulative Distribution Function
  - \( \text{cdf}[i] = \sum_{j=0}^{i} \text{bin}[j] / \text{totalPixelCount} \)

- Step 3 – Convert pixels using the cdf
  - \( \text{out} = 255 \times \text{cdf}(\text{in}) \)

- CDF can be computed in \( O(\log) \) time
  - For large dynamic range, this can be important
Histogram Equalization

- a) Histogram and cdf
- b) Equalized histogram and cdf
Carry-Lookahead Adder

- Recall: $\text{Sum}_i = A_i \text{ xor } B_i \text{ xor } C_i$
  - where $C_i$ is the carry in from the next bit position
- If we can compute carry using scan, we can compute add in $O(\log)$ time
- Problem: Parallel scan only works for associative functions
  - Carry is not associative
- We will use two other functions that are associative
  - Carry Generate (G)
  - Carry Propagate (P)
Carry-Lookahead Adder

- Generate – true, if an adder “block” generates a carry regardless of the carry in
- Propagate – true, if an adder “block” propagates a carry in to the carry out
- Example: 1-bit adder

- Example: 2-bit adder

- Associativity: Combining Generates and Propagates
Carry-Lookahead Adder

- \( C_i = G_{i:0} + Cin \times P_{i:0} \)
- That is, if we compute \( G_{i:0} \) and \( P_{i:0} \) fast, then we can compute \( C_i \) fast
- And we can use parallel scan (aka parallel prefix) to compute \( G_{i:0} \) and \( P_{i:0} \) fast
System Timing

- Register Timing Constraints
  - Setup time
  - Hold time

- Clock Generation and Distribution
  - Clock skew
  - Multiple clocks
  - PLLs & DLLs

- Asynchronous communication
  - Not all modules can have the same clock
  - Distance and too much clock skew

- Synchronous communication
  - Source-synchronous signaling
Registers

- Sample data using clock
- Hold data between clock cycles
- Computation (and delay) occurs between registers
there is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized
Typical timing specifications

- Positive edge-triggered D flip-flop
  - setup and hold times
  - minimum clock width
  - propagation delays (low to high, high to low, max and typical)

![Diagram of D flip-flop timing]

- $T_{su} = 2\text{ns}$
- $T_{h} = 1\text{ns}$
- $T_{su} = 2\text{ns}$
- $T_{h} = 1\text{ns}$
- $T_{w} = 7\text{ns}$
- $T_{plh} = [2,4]\text{ns}$
- $T_{phl} = [1,3]\text{ns}$

All measurements are made from the clocking event that is, the rising edge of the clock.
Synchronous System Model

- Register-to-register operation
- Perform operations during transfer
- Many transfers/operations occur simultaneously
System Clock Frequency

- Register transfer must fit into one clock cycle
  - \( \text{reg } t_{pd} + \text{C.L. } t_{pd} + \text{reg } t_{su} < T_{clk} \)
- Use maximum delays
- Find the “critical path”
  - Longest register-register delay
Short Paths

- Can a path have too little delay?
  - Yes: Hold time can be violated
  - \( t_{pd} > t_h \)
  - Use min delay (contamination delay)

- Fortunately, most registers have hold time = 0
  - But there can still be a problem! Clock skew…
Clock Skew

- Cannot make clock arrive at registers at the same time
- If skew > 0:
  - \( t_{pd} > t_h + t_{skew} \)
- Clock skew can cause system failure
  - Can you fix this after you’ve fabbed the chip?
Clock Skew

- Cannot make clock arrive at registers at the same time
- If skew > 0:
  - \( t_{pd} > t_h + t_{skew} \)
- Clock skew can cause system failure
  - Can you fix this after you’ve fabbed the chip?
Clock Skew

- If skew < 0:
  - $t_{clk} > reg \ t_{pd} + CL \ t_{pd} + reg \ t_{SU} + |t_{skew}|$
  - Can you fix this after fab?
Clock Skew

- If skew < 0:
  - $t_{clk} > reg \ t_{pd} + CL \ t_{pd} + \ reg \ t_{SU} + |t_{skew}|$
  - Can you fix this after fab?
Clock Skew

- Correct behavior assumes that all storage elements sample at exactly the same time.
- Not possible in real systems:
  - clock driven from some central location
  - different wire delay to different points in the circuit
- Problems arise if skew is of the same order as FF contamination delay
- Gets worse as systems get faster (wires don't improve as fast)
  - 1) distribute clock signals against the data flow
  - 2) wire carrying the clock between two communicating components should be as short as possible
  - 3) try to make all wires from the clock generator be the same length => clock tree
Altera FPGA PLL

Figure 2-16. Cyclone II PLL  Note (1)
What About External Inputs?

- Internal signals are OK
  - Can only change when clock changes

- External signals can change at any time
  - Asynchronous inputs
  - Truly asynchronous
  - Produced by a different clock

- This means register may sample a signal that is changing
  - Violates setup/hold time
  - What happens?
Sampling Asynchronous Inputs
Sampling Rate

- How fast does your sample clock need to be?
Sampling Rate

- How fast does your sample clock need to be?
  - $f(\text{clkB}) > f(\text{clkA})$
  - $f(\text{clkB}) > 2 \times f(\text{data})$ (Nyquist)
Another Problem with Asynchronous inputs

- What goes wrong here?

- What is the fix?
Think About the Reset Signal!

- Reset input is typically asynchronous
  - Which edge do we care about?
  - Does it matter if the registers have synchronous or asynchronous resets?
- Reset is broadcast to all registers in system
- What happens?

- Register input should be single-registered just like every other asynchronous input
More Asynchronous inputs

- What is the problem?
- What is the fix?
Important Rule!

- Exactly one register makes the synchronizing decision
More Asynchronous inputs

- Can we input asynchronous data values with several bits?
More Asynchronous inputs

- How can we input asynchronous data values with several bits?
What Went Wrong?

- Each bit has a different delay
  - Wire lengths differ
  - Gate thresholds differ
  - Driver speeds are different
  - Register delays are different
    - Rise vs. Fall times
  - Clock skews to register bits
- Bottom line – “data skew” is inevitable
  - aka Bus Skew
  - Longer wires => More skew
- What is the solution??
Sending Multiple Data Bits

- Must send a “clock” with the data – Source-Synchronous
  - Waits until data is stable
    - “Sample window”
  - De-skewing delay
- $f(\text{clkB}) > 2 f(\text{clkA})$
Crossing Clock Domain Boundaries is Tricky

- Direct sampling of multiple asynchronous bits is impossible
  - We can reliably sample only a single bit

- But even worse: Synchronization Failure [Metastability]
  - Non-zero probability that our system will fail
  - And there is nothing we can do about
    - Although we can reduce the probability to an extremely low value
Synchronization failure

- Occurs when FF input changes close to clock edge
  - the FF may enter a metastable state – neither a logic 0 nor 1 –
  - it may stay in this state an indefinite amount of time
  - this is not likely in practice but has some probability

small, but non-zero probability that the FF output will get stuck in an in-between state

oscilloscope traces demonstrating synchronizer failure and eventual decay to steady state
Guarding against synchronization failure

- Key idea – give the register as long as possible to make up its mind
  - Probability of failure cannot be reduced to 0, but low enough that we don’t care
- Cascade two registers
- Use very fast registers
- Slow down the clock
  - Slows communication rate
Calculating probability of failure

For a single synchronizer

\[
\text{Mean-Time Between Failure (MTBF)} = \exp \left( \frac{t_r}{\tau} \right) \div \left( T_0 \times f_c \times f_a \right)
\]

where a failure occurs if metastability persists beyond time \( t_r \)

- \( t_r \) is the resolution time – slack time in the clock period for settling
  - \( T_{\text{clk}} - (t_{\text{pd}} + T_{\text{CL}} + t_{\text{setup}}) \)
- \( f_c \) is the frequency of the FF clock
- \( f_a \) is the frequency of asynchronous events on the input of the FF
- \( T_0 \) and \( \tau \) are constraints that depend on the FF's electrical characteristics (e.g., gain or steepness of curve)
- Must add probabilities from all synchronizers in system
  \[
  \frac{1}{\text{MTBF}_{\text{system}}} = \sum \frac{1}{\text{MTBF}_{\text{synch}}}
  \]
Altera Stratix-3 Example Data

Measured Vs. Predicted MTBFs

- Predicted Typical
- Predicted Worst Case
- Measured (1.1V)
- Measured (1.02V)

MTBF (s)

Clock Frequency (MHz)

225 275 325 375 425 475