Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 – Introductory Digital Systems Laboratory

Problem Set 3

Problem Set Issued: March 2, 2007
Problem Set Due: March 14, 2007

Problem 1: Critical Path Timing Analysis

From Lecture 9, the critical path is:

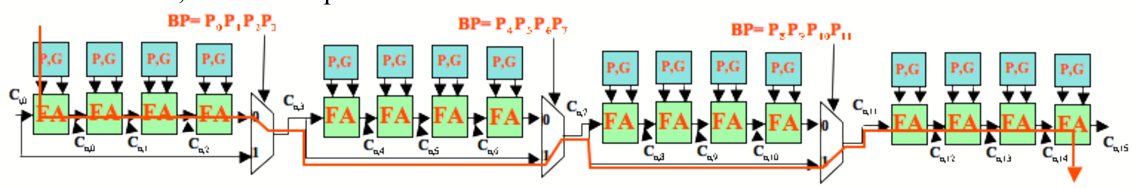

Figure 1. Critical path for the carry bypass adder

For each 4-bit carry bypass adder the critical path for generation of the carry out bit must go through one P, G unit (1 unit) and four full adders (4 units) for a total of 5 units.

Each BP signal BP, BP2, BP3, etc… is generated in parallel and equally affects the critical path so we only need to add the contribution of generating the carry out bit for a 4-bit adder once.

For the critical path computation we consider the path originating from the leftmost 4-bit adder because it must bypass the most 4-bit adder units (i.e. travel through the most 2:1 multiplexers). For the case shown above we pass through three 2:1 multiplexers (3 units).

Finally, the critical path is dependent on the computation of the most significant sum bit (S_{15}) which is a function of the propagate and carry-in bit ($S_{15} = P_{15}$ xor $C_{i,15}$). $C_{i,15}$ is a function of the final 4-bit adder so the critical path must pass through an additional 4 full adders (4 units).

Adding up the critical path we have 5 + 3 + 4 = 12 units. In summary, that is 5 units for the first 4-bit adder, 3 units for the 2:1 multiplexers, and 4 units for the final sum bit, which is a function of $C_{i,15}$.

Problem 2: Twos Complement Multiplier

a) One solution is to conditionally convert `x` and `y` from 2's complement to sign magnitude, multiply the two results, and then conditionally convert back. The condition can be performed using an `XOR` of each bit of the signal with the MSB of the signal, and then adding the MSB to the signal as shown below.

```
module mult8x8(x,y,z);
input [7:0] x,y;
```

```
    output [15:0] z;

    wire sign;
    wire [7:0] a,b;
    wire [15:0] c;

    assign a = ({8{x[7]}} ^ x) + x[7];
    assign b = ({8{y[7]}} ^ y) + y[7];
    assign c = a*b;
    assign sign = x[7]^y[7];
    assign z = ({16{sign}} ^ c) + sign;

    endmodule
```

We can test the multiplier by running it on a range of input values. You then take a look at the result and see if the answers are correct. One possible test bench could be:

```
`timescale 1ns / 1ps

module mult8x8_tb;

  reg [7:0] x;
  reg [7:0] y;
  wire [15:0] z;

// uncomment the module you wish to test
// mult8x8 m8x8 (x,y,z);
// signed_mult8x8 m8x8(x,y,z);

  integer    i;
  integer    j;

  initial
    begin
      #100;
      x = 0;
      y = 0;

      for (i = 0; i < 16; i = i + 1)
     begin
       x = i;
       for (j = -8; j < 8; j = j + 1)
         begin
           y = j;
           #50;
           $display("%d * %d = %d",x, y, z);
         end
     end

      $finish;
    end

endmodule
```

Here is the corresponding waveform:

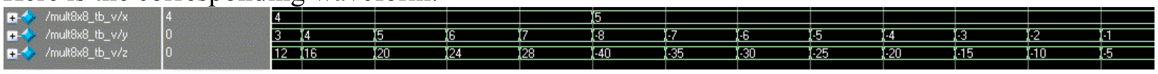

Figure 2. Waveform for mult8x8_tb

b) Using the signed modifier only works in some implementations of Verilog. Luckily, the Xilinx tools have incorporated this functionality.

```
module signed_mult8x8(x,y,z);
  input signed [7:0] x,y;
  output signed [15:0] z;

  assign z = x * y;

endmodule
```

Problem 3: Generating Block RAMs

a) Here are the steps needed to generate a 16x16 BRAM.

Right click in the "Sources" window and select the "New Source…" option. This will open a new window where you can name your module and say what kind of file you would like it to be.

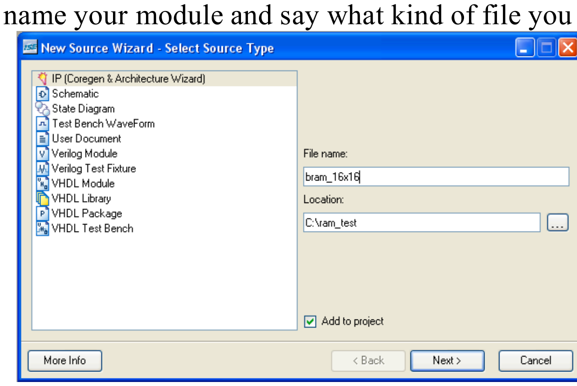

Figure 3. New Source window

Click next to open up the core selection window and choose "Memories & Storage Elements/RAMs & ROMs/Single Port Block Memory v6.2".

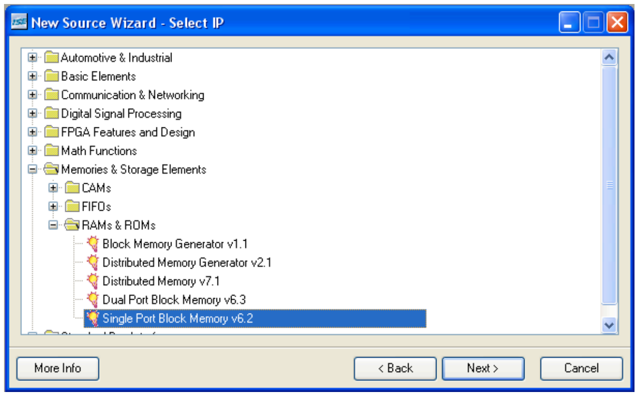

Figure 4. Core selection window

Click next, and then click finish. The core generator window will now appear. The default settings will do for this application. You only need to change the width and depth to be 16 and 16 respectively.

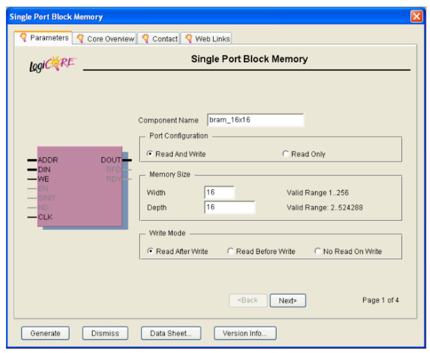

Figure 5. Core generation window

Click the button that says "Generate". The module that you created should now appear in your "Sources " window.

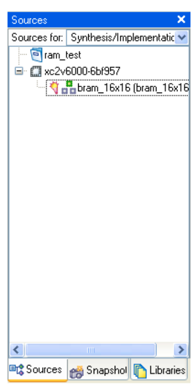

Figure 6. Sources in Project window with the generated core

b) After completing part a, you can now write a testbench as you normally would. Here is one possible testbench that writes 0x6363 to location 5, reads from location 12, and then reads from location 5 to show that the data was written correctly.

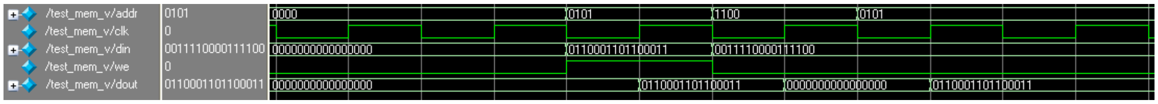

Figure 7. Screenshot of 16x16 BRAM testbench

```
module test_mem_v;

      // Inputs
      reg [3:0] addr;
      reg clk;
      reg [15:0] din;
      reg we;

      // Outputs
      wire [15:0] dout;

// Instantiate the Unit Under Test (UUT)
      bram_16x16 uut (
            .addr(addr),
            .clk(clk),
            .din(din),
            .dout(dout),
            .we(we)
      );

      always #5 clk <= ~clk;
      initial begin
            // Initialize Inputs
            addr = 0;
            clk = 0;
            din = 0;
            we = 0;

            // Wait 100 ns for global reset to finish
            #100;
            // falling edge at multiples of 10,
            // therefore the values below will be
            // settled by the time the rising edge comes
            addr = 4'h5;
            din = 16'h6363;
            we = 1'b1;
            #10;
```

```
        // we've satisfied the hold time so
        // _we_ can be deasserted and the address
        // can be modified
        we = 0;
        addr = 4'hc;
        din = 16'h3c3c;
        #10;
        // check to see if our data was written
        addr = 4'h5;

    end

endmodule
```

Problem 4: Introduction to Video

a) Here is the code for a video controller:

```
// This module provides control signals to the ADV7125
// such that the resolution is 640x480 and the refresh
// rate is 75Hz.

// hsync is active low: high for 640 pixels of active video,
//                      high for 16 pixels of front porch,
//                      low for 96 pixels of hsync,
//                      high for 48 pixels of back porch

// vsync is active low: high for 480 lines of active video,
//                      high for 11 lines of front porch,
//                      low for 2 lines of vsync,
//                      high for 32 lines of back porch

module vga (pixel_clock, reset, hsync, vsync, sync_b,
         blank_b, pixel_count, line_count);

  input pixel_clock; // 31.5 MHz pixel clock
  input reset; // system reset
  output hsync; // horizontal sync
  output vsync; // vertical sync
  output sync_b; // hardwired to Vdd
  output blank_b; // composite blank
  output [9:0] pixel_count; // number of the current pixel
  output [9:0] line_count; // number of the current line
  // 640x480 75Hz parameters

  parameter    PIXELS = 800;
  parameter    LINES = 525;
  parameter    HACTIVE_VIDEO = 640;
  parameter    HFRONT_PORCH = 16;
  parameter    HSYNC_PERIOD = 96;
  parameter    HBACK_PORCH = 48;
  parameter    VACTIVE_VIDEO = 480;
  parameter    VFRONT_PORCH = 11;
  parameter    VSYNC_PERIOD = 2;
  parameter    VBACK_PORCH = 32;

  // current pixel count
  reg [9:0] pixel_count = 10'b0;
  reg [9:0] line_count = 10'b0;
```

```
// registered outputs
reg          hsync = 1'b1;
reg          vsync = 1'b1;
reg          blank_b = 1'b1;
wire         sync_b; // connected to Vdd

wire pixel_clock;
wire [9:0] next_pixel_count;
wire [9:0] next_line_count;

always @ (posedge pixel_clock)
  begin
    if (reset)
   begin
     pixel_count <= 10'b0;
     line_count <= 10'b0;
     hsync <= 1'b1;
     vsync <= 1'b1;
     blank_b <= 1'b1;
   end
    else
   begin

     pixel_count <= next_pixel_count;
     line_count <= next_line_count;

     hsync <=
         (next_pixel_count < HACTIVE_VIDEO + HFRONT_PORCH) |
         (next_pixel_count >= HACTIVE_VIDEO+HFRONT_PORCH+
                              HSYNC_PERIOD);


     vsync <=
         (next_line_count < VACTIVE_VIDEO+VFRONT_PORCH) |
         (next_line_count >= VACTIVE_VIDEO+VFRONT_PORCH+
                             VSYNC_PERIOD);

     // this is the and of hblank and vblank
     blank_b <=
         (next_pixel_count < HACTIVE_VIDEO) &
         (next_line_count < VACTIVE_VIDEO);


   end
  end

// next state is computed with combinational logic
assign next_pixel_count = (pixel_count == PIXELS-1) ?
                          10'h000 : pixel_count + 1'b1;

assign next_line_count = (pixel_count == PIXELS-1) ?
                         (line_count == LINES-1) ? 10'h000 :
                         line_count + 1'b1 : line_count;
```

```
  // since we are providing hsync and vsync to the display, we
  // can hardwire composite sync to Vdd.
  assign sync_b = 1'b1;

endmodule
```

b) Here is a screenshot of what your waveform should look like

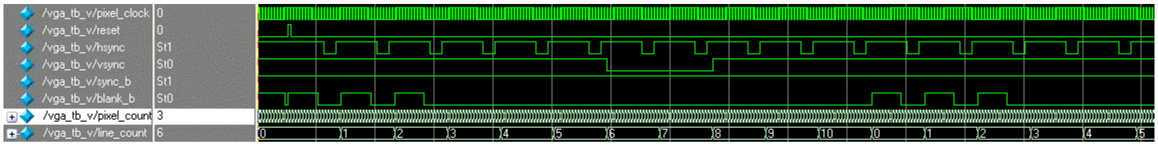

Figure 8. VGA testbench waveform

The verilog code that was used to produce this waveform and test the VGA module is this:

```
`timescale 1ns / 1ps
module vga_tb_v;

      // Inputs
      reg pixel_clock;
      reg reset;


      // Outputs
      wire hsync;
      wire vsync;
      wire sync_b;
      wire blank_b;
      wire [9:0] pixel_count;
      wire [9:0] line_count;

      // Instantiate the Unit Under Test (UUT)
      vga uut (
            .pixel_clock(pixel_clock),
            .reset(reset),
            .hsync(hsync),
            .vsync(vsync),
            .sync_b(sync_b),
            .blank_b(blank_b),
            .pixel_count(pixel_count),
            .line_count(line_count)
      );
      // define smaller parameters
      // so that simulation runs in a
      // reasonable amount of time
      defparam    uut.PIXELS = 18;
      defparam    uut.LINES = 11;
      defparam    uut.HACTIVE_VIDEO = 10;
      defparam    uut.HFRONT_PORCH = 2;
      defparam    uut.HSYNC_PERIOD = 4;
      defparam    uut.HBACK_PORCH = 2;
      defparam    uut.VACTIVE_VIDEO = 3;
      defparam    uut.VFRONT_PORCH = 3;
      defparam    uut.VSYNC_PERIOD = 2;
      defparam    uut.VBACK_PORCH = 3;

      always #5 pixel_clock <= ~pixel_clock;
```

```
    initial begin
        // Initialize Inputs
        pixel_clock = 0;
        reset = 0;

        // Wait 100 ns for global reset to finish
        #100;
        #5;
        reset = 1;
        #10;
        reset = 0;
        #2000;

    end

endmodule
```

c) There are multiple ways to implement the checkerboard pattern. You can count how many lines and pixels have occurred for example. If you divide the screen into 10 regions of 64x48 pixels, then you can use the sixth bit of the pixel count to change the order that colors are output by the logic that generates the rows across the screen. The code below takes a different approach to demonstrate how you can use "for" loops to generate module descriptions for you. Here we just enumerate the regions where the select bit of a mux is a 1 or a 0, then use that bit to choose the output from the second set of logic. You might be able to do something like this to generate different on-screen parts of your pong lab.

```
module checkerboard(pixel, line, red, green, blue);
  input [9:0] pixel, line;
  output [7:0] red, green, blue;

  reg [7:0]    red, green, blue;
  reg flip;

  parameter    WIDTH = 640;
  parameter    HEIGHT = 480;
  parameter    ROW_HEIGHT = 96;
  parameter    COL_WIDTH = 128;

  integer      i,j;

  always @ (pixel or line or flip)
    begin

      flip = 0;

      for (j = 0; j < 10; j = j + 1)
        begin
          if ((j*ROW_HEIGHT/2 <= line) &&
              (line < (j+1)*ROW_HEIGHT/2))
          begin
            flip = ((j%2)==0);
          end
        end

      for (i = 0; i < 10; i = i + 1)
        if ((i*COL_WIDTH/2 <= pixel) &&
            (pixel < (i+1)*COL_WIDTH/2))
        begin
          {red, green, blue} = flip ?
                          (((i%2)==0) ? 24'h000000 : 24'hffffff):
                          (((i%2)==0) ? 24'hffffff : 24'h000000);
        end
      end
endmodule // checkerboard
```