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

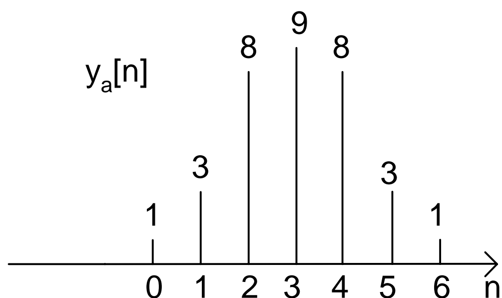
Problem Set 3
Solutions

Problem Set Issued: March 3, 2004
Solutions Issued: March 19, 2004

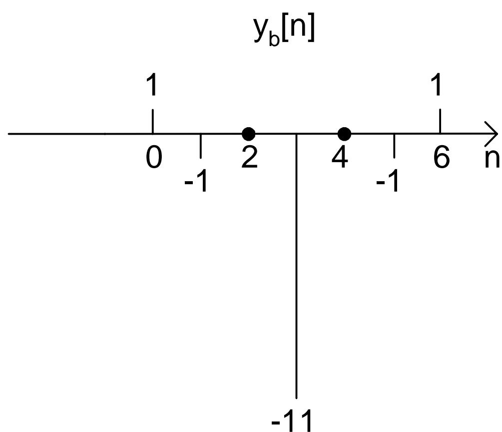
Problem 1 – Convolution

Part (a)

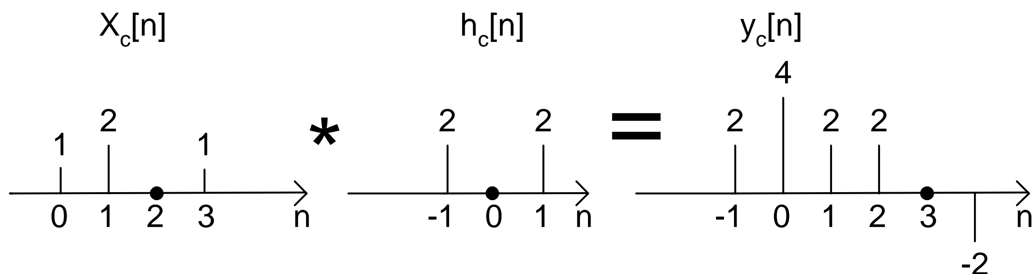
We can answer this problem using the convolution sum or the flip and shift method.



Part (b)



Part (c)



Part (d)

For this part we consider a multiply-accumulate module of the form similar to the one to be used in Lab 3.

The multiplication of two eight-bit numbers requires 16 output bits. The fact that we are using a 10-tap filter means that we will need to accumulate ten 16-bit numbers. The summation of ten product terms affects the width of the adder by requiring the addition of the binary representation of 10 (4'b1010) with the width of the multiplier output (16 bits). Thus we have 16 + 4 = 20 bits.

Problem 2 – Two's Complement Multiplier

Several methods of designing an 8x8 multiplier are possible for this problem. The easiest approach is to use unsigned multiplication using the * multiply operation in Verilog and then accounting for the sign on the MSB. The Verilog code and screen capture for this approach are provided below. In this solution we present both a blocking solution (asynchronous) and a non-blocking (synchronous) solution. The synchronous solution pipelines the logic from the asynchronous solution. There is a delay of four clock cycles associated with using the synchronous multiplier. Please note that we account for both zero representations in magnitude representation by checking for this condition on the input y.

Other approaches are possible for answering this question. One possible alternative was to implement an 8x8 Baugh-Wooley multiplier based off the 4x4 implementation presented in Lecture 8.

Asynchronous Multiplier

```
//8x8 Multiplier (Two's Complement x Magnitude)
module twos_compliment_multiplier (x, y, z);
	input [7:0] x, y;
	output [15:0] z;
	wire [7:0] x_neg, y_neg;
	wire [15:0] z_pos, z_neg, z;

	assign x_neg = (x == 8'b0) ? 8'b0: (~x + 1);
	assign y_neg = (y == 8'b0 || y == 8'b10000000) ? 8'b0: (~y + 1);
	assign z_pos = (x[7] ? x_neg : x) * (y[7] ? y_neg : y);
	assign z_neg = (z_pos == 8'b0) ? 8'b0 : (~z_pos + 1);
	assign z = (x[7] ^ y[7]) ? z_neg : z_pos;
endmodule
```

Synchronous Multiplier

```
module synch_twosmult (clk, reset, x, y, z);
input clk, reset;
input [7:0] x, y;
output [15:0] z;
reg [7:0] x_d0, y_d0, x_d1, y_d1, x_d2, y_d2, x_d3, y_d3; //delay registers
reg [7:0] x_neg, y_neg;
reg [15:0] z_pos, z_neg, z_pos_d1;
reg [15:0] z;

always @ (posedge clk or negedge reset)   begin
   x_d0 <= x;
   y_d0 <= y;
   x_d1 <= x_d0;
   y_d1 <= y_d0;
   x_d2 <= x_d1;
   y_d2 <= y_d1;
   x_d3 <= x_d2;
   y_d3 <= y_d2;
   z_pos_d1 <= z_pos;

     if (x_d0 == 8'b0) begin
        x_neg <= 8'b0;
     end else begin
        x_neg <= ~x_d0 + 1;
               end
               if (y_d0 == 8'b0 || y_d0 == 8'b10000000) begin
        y_neg <= 8'b00000000;
     end else begin
        y_neg <= ~y_d0 + 1;
               end

               if (x_d1[7] == 1'b1 & y_d1[7] == 1'b1) begin
                           z_pos <= x_neg * y_neg;
               end else if (x_d1[7] == 1'b1 & y_d1[7] == 1'b0) begin
                           z_pos <= x_neg * y_d1;
               end else if (x_d1[7] == 1'b0 & y_d1[7] == 1'b1) begin
                           z_pos <= x_d1 * y_neg;
               end else if (x_d1[7] == 1'b0 & y_d1[7] == 1'b0) begin
                           z_pos <= x_d1 * y_d1;
              end

              if (z_pos == 16'b0) begin
                  z_neg <= 16'b0;
              end else begin
                           z_neg <= (~z_pos + 1);
              end

              if (x_d3[7] ^ y_d3[7]) begin
                           z <= z_neg;
              end else begin
                           z <= z_pos_d1;
              end
   end
endmodule
```

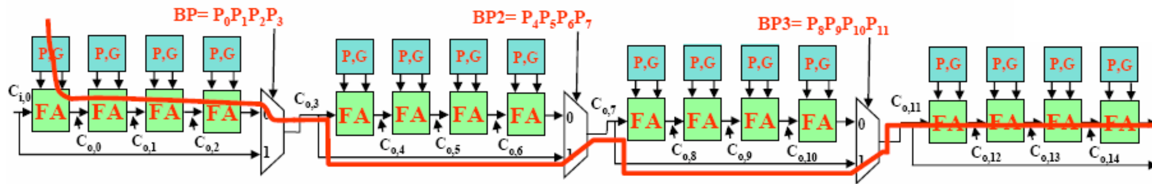
Testbench (for Synchronous Multiplier)

```
module twosmult_tb;

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

synch_twosmult mult1
(
.clk(clk),
.reset(reset),
.x(x),
.y(y),
.z(z)
);

always #50 clk = ~clk;

initial begin
        x = 0;
        y = 0;
        clk = 0;
        reset = 0;
        #100
        reset = 1;
        #100
        x = 8'd0;
        y = 8'd50;
        #100
        x = 8'd1;
        y = 8'd50;
        #100
        x = 8'b11111111; // x = -1
        y = 8'd50;
        #100
        x = 8'b01111111;
        y = 8'b01111111;
        #100
        x = 8'b11111111;
        y = 8'b01111111;
        #100
        x = 8'd50;
        y = 8'b10000000;
end
endmodule
```

/twosmult_tb/clk
/twosmult_tb/reset
/twosmult_tb/x 0 1 -1 127 -1 50
/twosmult_tb/y 0 50 127 128
/twosmult_tb/z 0 50 -50 16129 -127 0

Note that we handle the Magnitude Case where y = 8'b10000000. There is a delay of four clock cycles using this multiplier.

Problem 3 – Critical Path Timing Analysis



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… are generated in parallel and equally affect 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 multiplexors). For the case shown above we pass through three 2:1 multiplexors (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 multiplexors, and 4 units for the final sum bit computation which is a function of $C_{i,15}$.

Problem 4 - FIFO Design and Major/Minor FSMs

A parameterized RAM was created using MaxPlusII's MegaWizard using the following memory initialization file.

```
-- MEMORY INITIALIZATION FILE
-- EXAMPLE DATA FOR AN 8x8 ROM

WIDTH = 8;   % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE %
DEPTH = 8;   % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL VALUE %

ADDRESS_RADIX = HEX;  % Address and data radixes are optional, default is
hex %
DATA_RADIX = HEX;     % Valid radixes = BIN,DEC,HEX or OCT  %
```

```
CONTENT BEGIN
        0       :       07;  % ADDRESS :  VALUE %
        1       :       06;
        2       :       05;
        3       :       04;
        4       :       03;
        5       :       02;
        6       :       01;
        7       :       00;
       8      :      10;
       9      :      20;
       A      :      30;
       B      :      40;
       C      :      50;
       D      :      60;
       E      :      70;
       F      :      80;
END;

-- SHORTCUTS FOR SPECIFYING CONTENTS
--   [0..FF]    :       0;   % Range--Every address from 0 to FF = 0%
--      D       :       7;      % Single address--Address D = 7 %
--      6       :       9 C 8;  % Range starting from specific address--%
-- If there are multiple values for the same address only the last value
is used
```

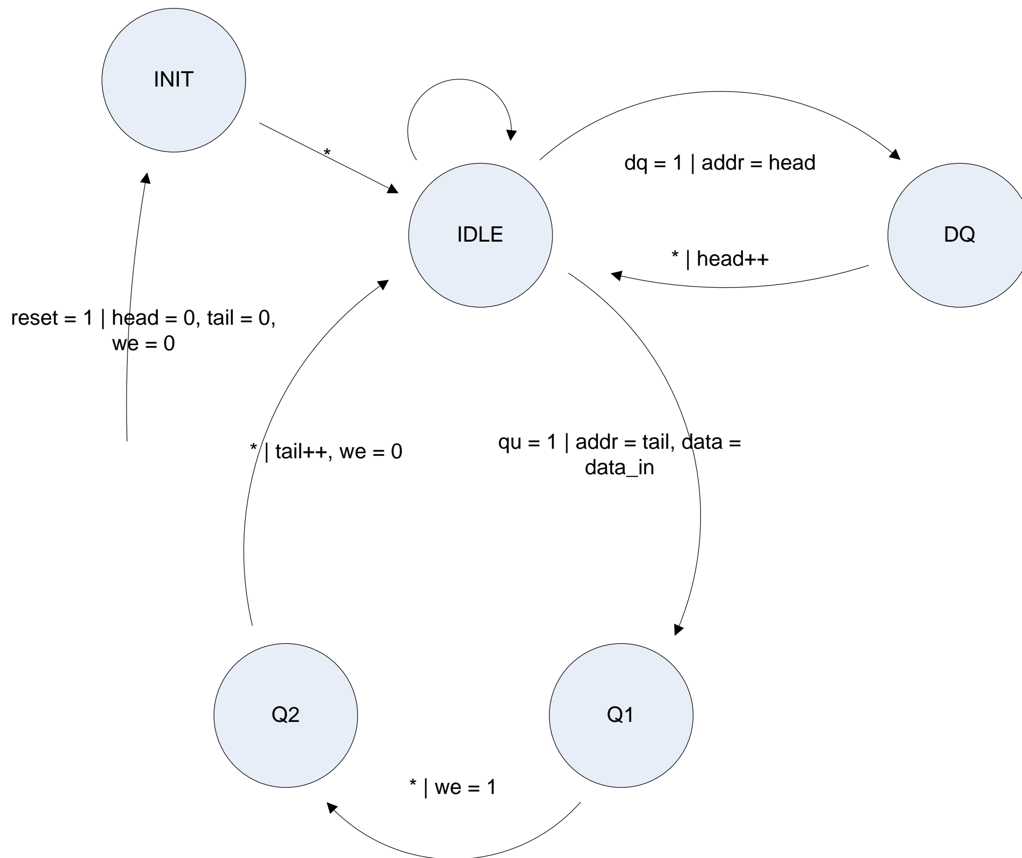
The following module was automatically generated:

```
module ram8x16 (address,inclock,we,data,q);

       input  [3:0]  address;
       input    inclock;
       input    we;
       input  [7:0]  data;
       output [7:0]  q;

       wire [7:0] sub_wire0;
       wire [7:0] q = sub_wire0[7:0];

       lpm_ram_dq   lpm_ram_dq_component (
                           .address (address),
                           .inclock (inclock),
                           .data (data),
                           .we (we),
                           .q (sub_wire0));
       defparam
              lpm_ram_dq_component.lpm_width = 8,
              lpm_ram_dq_component.lpm_widthad = 4,
              lpm_ram_dq_component.lpm_indata = "REGISTERED",
              lpm_ram_dq_component.lpm_address_control = "REGISTERED",
              lpm_ram_dq_component.lpm_outdata = "UNREGISTERED",
              lpm_ram_dq_component.lpm_file = "U:/ps3.mif",
              lpm_ram_dq_component.lpm_hint = "USE_EAB=ON";


endmodule
```

The FSM for our FIFO controller consists of five states, one for initialization of internal variables, one for an idle state that waits for queue or dequeue operations, two for the queue operation, and one for the dequeue operation.

The queue operation requires two states since it entails a write—one state to setup address and data and one state to pulse write enable. The dequeue operation—just a read—requires one state to setup the address.

The following are a state transition diagram and the Verilog code implementation.




```
module FIFO (clk, reset, qu, dqu, data, out, tail, head, addr, we,
tail_int, head_int);

	input clk, qu, dqu, reset;
	input [7:0] data;
	output [7:0] out;
	output [3:0] head, tail, addr, tail_int, head_int;
	output we;

	reg [7:0] out_ram;
	reg [7:0] data_ram;
	reg [3:0] addr, addr_int, head, head_int, tail, tail_int;
	reg we, we_int, out_oen, out_oen_int;
	reg [2:0] state, next;

	parameter INIT = 0;
	parameter IDLE = 1;
	parameter Q1 = 2;
	parameter Q2 = 3;
	parameter DQ = 4;

	assign out = (out_oen) ? out_ram : 8'bZ;

	ram8x16 myram (
		.address(addr),
		.inclock(clk),
		.we(we),
		.data(data_ram),
		.q(out_ram)
	);
```

```
	always @ (posedge clk) begin
		if (reset) begin
			state <= INIT;
		end
		else state <= next;

		addr <= addr_int;
		tail <= tail_int;
		head <= head_int;
		we <= we_int;
		out_oen <= out_oen_int;

		if (next == Q1) data_ram <= data;
	end


	always @ (state or dqu or qu or reset) begin
		tail_int = tail;
		head_int = head;
		we_int = 0;
		addr_int = addr;
		out_oen_int = out_oen;


		if (state == INIT) begin
			tail_int = 0;
			head_int = 0;
			addr_int = 0;
			out_oen_int = 0;
			next = IDLE;
		end
		else if ((state == IDLE) && (qu)) begin  // do a queue
operation
			addr_int = tail;
			out_oen_int = 0;
			next = Q1;
		end
```

```
else if ((state == IDLE) && (dqu)) begin  // do a dequeue operation
                        addr_int = head;
                        out_oen_int = 1;
                        next = DQ;
                end
                else if (state == Q1) begin
                        we_int = 1;
                        next = Q2;
                end
                else if (state == Q2) begin  // incr the tail after a queue
                        tail_int = tail + 1;
                        next = IDLE;
                end
                else if (state == DQ) begin  // incr the head after a
dequeue
                        head_int = head + 1;
                        next = IDLE;
                end
                else
                        next = IDLE;
        end


endmodule
```

In the Verilog code, we instantiate the RAM we created in our earlier part. There are two internal variables to keep track of the tail and the head of the FIFO. On a queue operation, the address (tail) and data are first established, and write enable is pulsed in the next cycle. We return to the idle state on the next cycle while incrementing our tail pointer. Note that we rely on the contamination time of the combinational logic to hold our address. On a dequeue operation, the address (head) is established and it takes one cycle for the data to appear at the output. Upon returning to the idle state, we increment our head pointer. Also, as an optional feature the output is tristated during queue operations.

The following is the simulation waveform from MaxPlusII.