COE 405, Term 131

Design & Modeling of Digital Systems

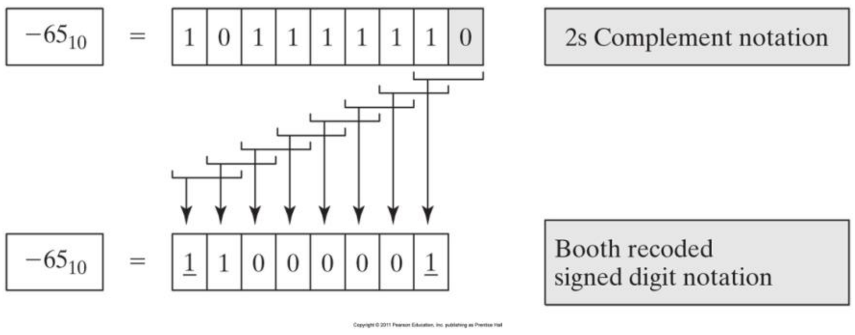
HW# 4

Due date: Thursday, Nov. 21

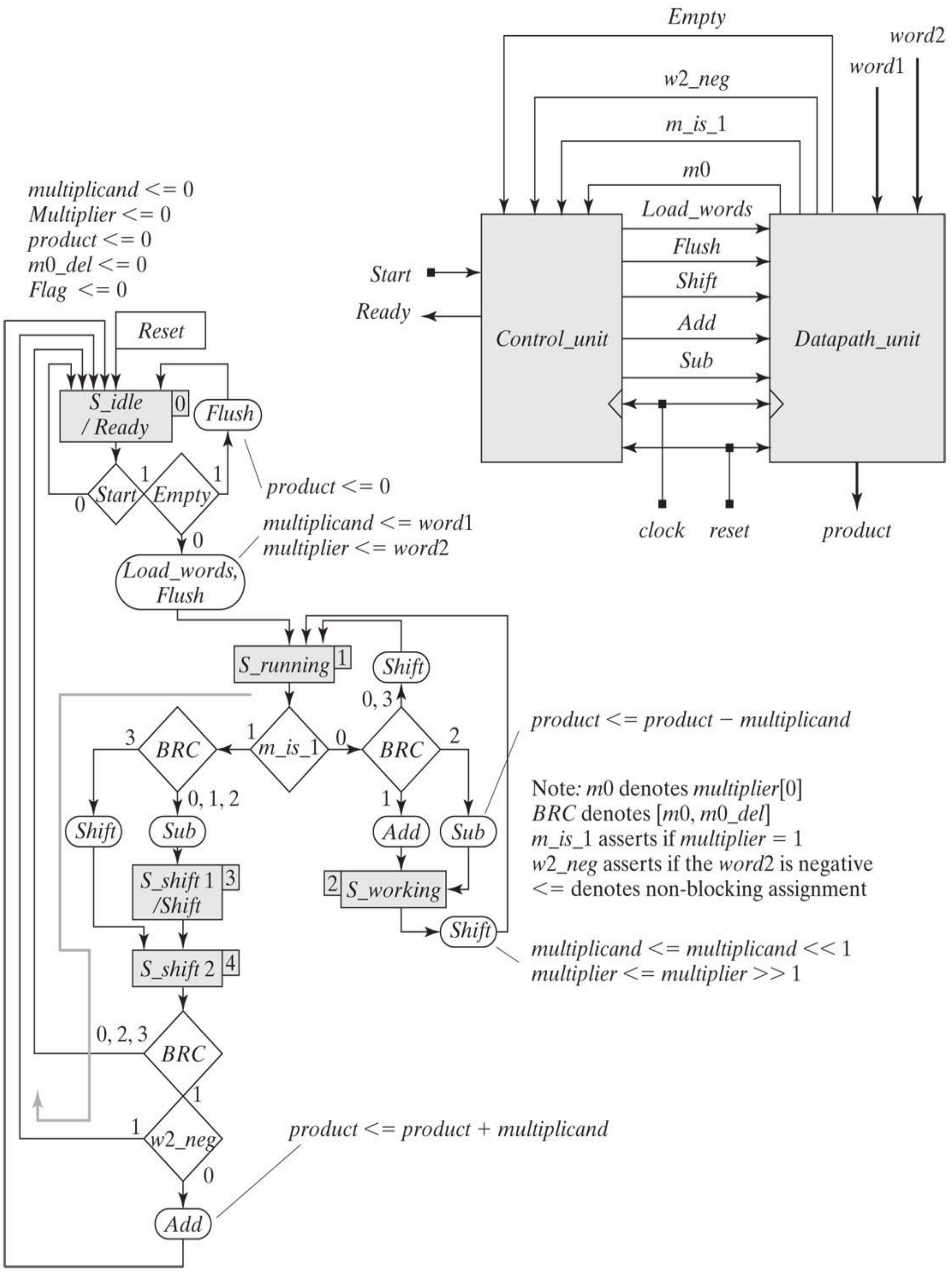
Q.1. Various algorithms have been developed to improve the performance of sequential multipliers and to simplify their circuitry. Booth's recoding algorithm is widely used because it has a simple hardware realization, requires less silicon area and can speed sequential multiplication significantly. Multipliers that use Booth's algorithm recode the bits of the multiplier to reduce the number of additions required to complete the cycle of multiplication. Only the multiplier is recoded; the multiplicand is left unchanged. Booth algorithm is applicable to positive numbers and to negative numbers in 2's complement representation. The key to Booth's algorithm is that it skips over strings of 1's in the multiplier and replaces a series of additions by one addition and one subtraction. For example, the word 1111_0000 is equivalent to 2^8-1-(2^4-1)=$2^8 - 2^4$ = 256-16=240. The table below summarizes the recoding rules.

m_i	m_{i-1}	BRC$_i$	Value	Status
0	0	0	0	String of 0s
0	1	1	+1	End of string of 1s
1	0	$\underline{1}$	−1	Begin string of 1s
1	1	0	0	Midstring of 1s

The algorithm reads from the LSB to the MSB and the value of two successive bits (m_i, m_{i-1}) determines the Booth recoded multiplier bit, BRC$_i$. As the algorithm reads two successive bits, the present and the immediate past, it forms and uses BRC$_i$ to determine whether to add or subtract before skipping to the next bit. The first step of the algorithm is seeded with a value of 0 to the right of the LSB of the word. If the signed digit $\underline{\mathbf{1}}$ is encountered, a subtraction operation is performed (i.e. an appropriately shifted copy of the 2's complement of the multiplicand is added to the product). The process encodes the first encountered 1 as a $\underline{\mathbf{1}}$, skips over any successive 1's until a 0 is encountered. That 0 is encoded as **1** to signify the end of a string of 1's, and then the process continues. As an example of Booth recoding, the encoding of -65=1011_1111 is shown below.


-65_{10}
=
1 0 1 1 1 1 1 1 0
2s Complement notation
-65_{10}
=
$\underline{1}$ 1 0 0 0 0 0 $\underline{1}$
Booth recoded
signed digit notation
Copyright © 2011 Pearson Education, Inc. publishing as Prentice Hall

A block diagram and an ASMD chart for the booth multiplier are given below. The machine is efficient as it does not waste time doing needless operations, such as multiplying by 0 or multiplying after the last 1 in the multiplier has been found. When word1 or word2 are 0, *Empty* signal is asserted. When the value of the multiplier register is 1, the machine's action depends on whether this last bit of 1 is possibly due to word2 having been the 2's complement code corresponding to a negative number (i.e., the MSB of word2 was 1). Moreover, when *m_is_1* is asserted, there are only two possible values of BRC: 2 and 3. In the former case, the usual subtraction must be performed. *Sub* is asserted and the state moves from *S_running* to *S_shift1*, where BRC is now 1. If word2 was negative, no further action is required; otherwise, a final addition must be executed. *Shift* is asserted in *S_Shift1* to align the multiplicand for the final addition, then the state moves to *S_Shift2*, where *Add* is asserted. The latter case must be handled differently, because the condition that BRC is 3 in *S_running* with *m_is_1* asserted could be due to an intermediate string of 1's or to a terminating string of 1's in word2. A terminating string of 1's corresponds to a negative 2's complement multiplier and dictates that *Add* not be asserted in *S_shift2*. There is no way to distinguish between these two cases without setting a flag in the datapath to indicate that word2 is negative and passing the result as a status signal to the controller. The machine uses a flag register in the datapath to form an additional status signal, *w2_neg*, to indicate that the data pattern of word2 has a negative value.



Note: $m0$ denotes *multiplier*[0]
BRC denotes [$m0$, *m0_del*]
m_is_1 asserts if *multiplier* = 1
w2_neg asserts if the *word2* is negative
<= denotes non-blocking assignment

(i) Show the design of the data-path and control unit of a 4-bit sequential Booth multiplier.

```
module BMultiplier #(parameter word_size=4)(
output [2*word_size-1: 0] product,
output Ready,
input  [word_size-1:0] word1, word2,
input  Start, clock, reset);

BMultiplier_CU  M1 (Load_words, Flush, Shift, Add, Sub, Ready, Start, m0,
m0_del,
m_is_1, w2_neg, Empty, clock, reset);

BMultiplier_DPU #(word_size) M2 (product, Empty, w2_neg, m_is_1, m0,
m0_del,word1, word2,
Load_words,Flush, Shift, Add, Sub, clock, reset);

endmodule


module BMultiplier_DPU #(parameter word_size=4)(
output reg [2*word_size-1: 0] product,
output Empty, w2_neg, m_is_1, m0,
output reg m0_del,
input [word_size-1:0] word1, word2,
input Load_words,Flush, Shift, Add, Sub, clock, reset);

reg [2*word_size-1:0] multiplicand;
reg [word_size-1:0] multiplier;
reg Flag;

assign m0 = multiplier[0];
assign w1z = ~| word1;
assign w2z = ~| word2;
assign Empty = w1z | w2z;
assign m_is_1 = (~|multiplier[word_size-1:1])&multiplier[0];
assign w2_neg = Flag;

always @ (posedge clock, posedge reset)
  if (reset) Flag <= 0;
  else if (Load_words) Flag <= word2[word_size-1];

always @ (posedge clock, posedge reset)
  if (reset) m0_del <= 0;
  else m0_del <= multiplier[0];

always @ (posedge clock, posedge reset)
  if (reset) product <= 0;
  else if (Flush) product <= 0;
  else if (Add)
    product <= product + multiplicand;
  else if (Sub)
    product <= product - multiplicand;

always @ (posedge clock, posedge reset)
  if (reset) multiplicand <= 0;
  else if (Load_words) multiplicand <= {{word_size{word1[word_size-1]}},word1};
  else if (Shift) multiplicand <= multiplicand << 1;

always @ (posedge clock, posedge reset)
  if (reset) multiplier <= 0;
  else if (Load_words) multiplier <= word2;
  else if (Shift) multiplier <= multiplier >> 1;

endmodule



module BMultiplier_CU  (output reg Load_words,
Flush, Shift, Add, Sub, Ready,
input Start, m0, m0_del, m_is_1, w2_neg, Empty, clock, reset);
wire [1:0] BRC;

assign BRC = {m0, m0_del};

parameter S_idle = 3'b000, S_running=3'b001, S_working=3'b010,
S_shift1=3'b011, S_shift2=3'b100;

reg [2:0]  state, next_state;

  always @(posedge clock, posedge reset)
   if (reset) state <= S_idle;
   else state <= next_state;

  always @(state, Start, BRC, m_is_1, Empty, w2_neg) begin
   next_state=S_idle;
   Load_words=0; Flush=0; Add=0; Sub=0; Shift=0; Ready=0;
   case (state)
      S_idle: begin
          Ready=1;
          if (Start) begin
            if (Empty) begin
                    next_state=S_idle; Flush=1; end
            else begin
                    next_state=S_running; Load_words=1; Flush=1; end
            end
```

```
         else next_state=S_idle;
      end
      S_running:
     if (m_is_1)
      case (BRC)
         0,1,2: begin Sub=1;next_state=S_shift1; end
         3: begin Shift=1;     next_state=S_shift2; end
         endcase
        else
         case (BRC)
         0,3: begin Shift=1;  next_state=S_running; end
         1: begin Add=1;      next_state=S_working; end
         2: begin Sub=1;      next_state=S_working; end
         endcase
      S_working: begin Shift=1; next_state=S_running; end
      S_shift1: begin Shift=1; next_state=S_shift2; end
      S_shift2:
         case (BRC)
      0,2,3:  next_state=S_idle;
      1: if (w2_neg) next_state=S_idle;
         else begin Add=1; next_state=S_idle; end
      endcase
    endcase
  end
endmodule
```

(ii) Model the circuit in Verilog and verify its correct functionality by simulation for the following values:

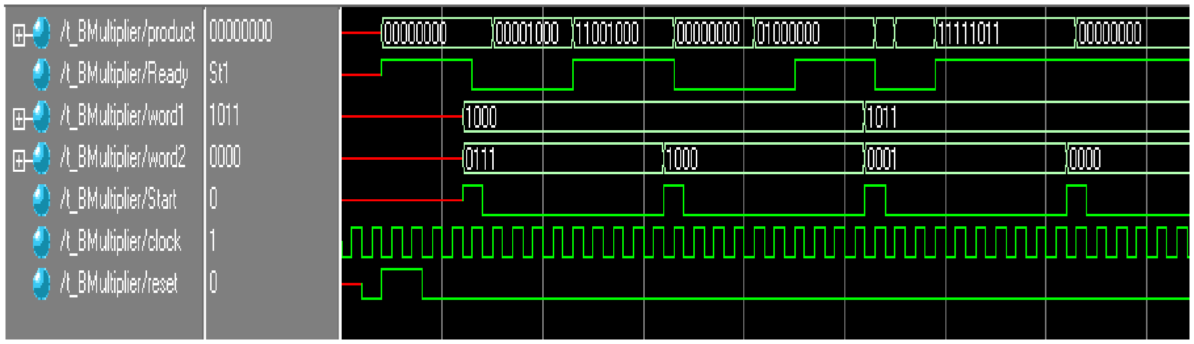
- Multiplicand=-8, multiplier=7
- Multiplicand=-8, multiplier=-8
- Multiplicand=-5, multiplier=1
- Multiplicand=-5, multiplier=0

```
module t_BMultiplier ();

wire [7: 0] product;
wire Ready;
reg  [3:0] word1, word2;
reg  Start, clock, reset;

BMultiplier M1 (product, Ready, word1, word2, Start, clock, reset);
initial #450 $finish;
initial begin clock = 0; forever #5 clock = ~clock; end
initial fork
#10 reset = 0; // Power-up reset
#20 reset = 1;
```

```
#40 reset = 0;
#60 word1 = 'b1000;
#60 word2 = 'b0111;
#60 Start = 1;
#70 Start = 0;
#160 word1 = 'b1000;
#160 word2 = 'b1000;
#160 Start = 1;
#170 Start = 0;
#260 word1 = 'b1011;
#260 word2 = 'b0001;
#260 Start = 1;
#270 Start = 0;
#360 word1 = 'b1011;
#360 word2 = 'b0000;
#360 Start = 1;
#370 Start = 0;
join
endmodule
```



(iii) Implement the circuit using Xilinx FPGA board and verify its correct functionality for the following values: (**1% Bonus**)

- Multiplicand=-8, multiplier=7
- Multiplicand=-8, multiplier=-8
- Multiplicand=-5, multiplier=1
- Multiplicand=-5, multiplier=0