Digital Systems Design ELEC373/473

Multipliers

Prof J.S. Smith
Room A515;
E-mail: j.s.smith@liv.ac.uk

Multipliers

- The next section looks at implementing multipliers in digital logic.
- For a 32 bit x 32 bit multiplication how wide (how many bits) could the result be?

MULTIPLIERS (unsigned)

- Paper and pencil example (unsigned):
- Multiplicand 1000
  Multiplier 1001

```
Multiplicand        1000
Multiplier          1001
                ---------
                    1000
                   0000
                  0000
                 1000
                ---------
Product          1001000
```

- m bits multiply by n bits = (m+n) bits product
- The multiplier’s LSB is checked. If it is

  0 => place 0 in the sub-product

  1 => place a copy of the multiplicand in the sub-product shifted by the appropriate number of bits

- 4 versions of multiply hardware & algorithm will be presented with successive refinement.

Unsigned Multiplication

AB_i called a "partial product" $\longrightarrow$

$$\underbrace{\begin{array}{ccccccc}
 & & & A_3 & A_2 & A_1 & A_0\\
 & & \times & B_3 & B_2 & B_1 & B_0\\ \hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0\\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+\ A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\ \hline
\end{array}}$$

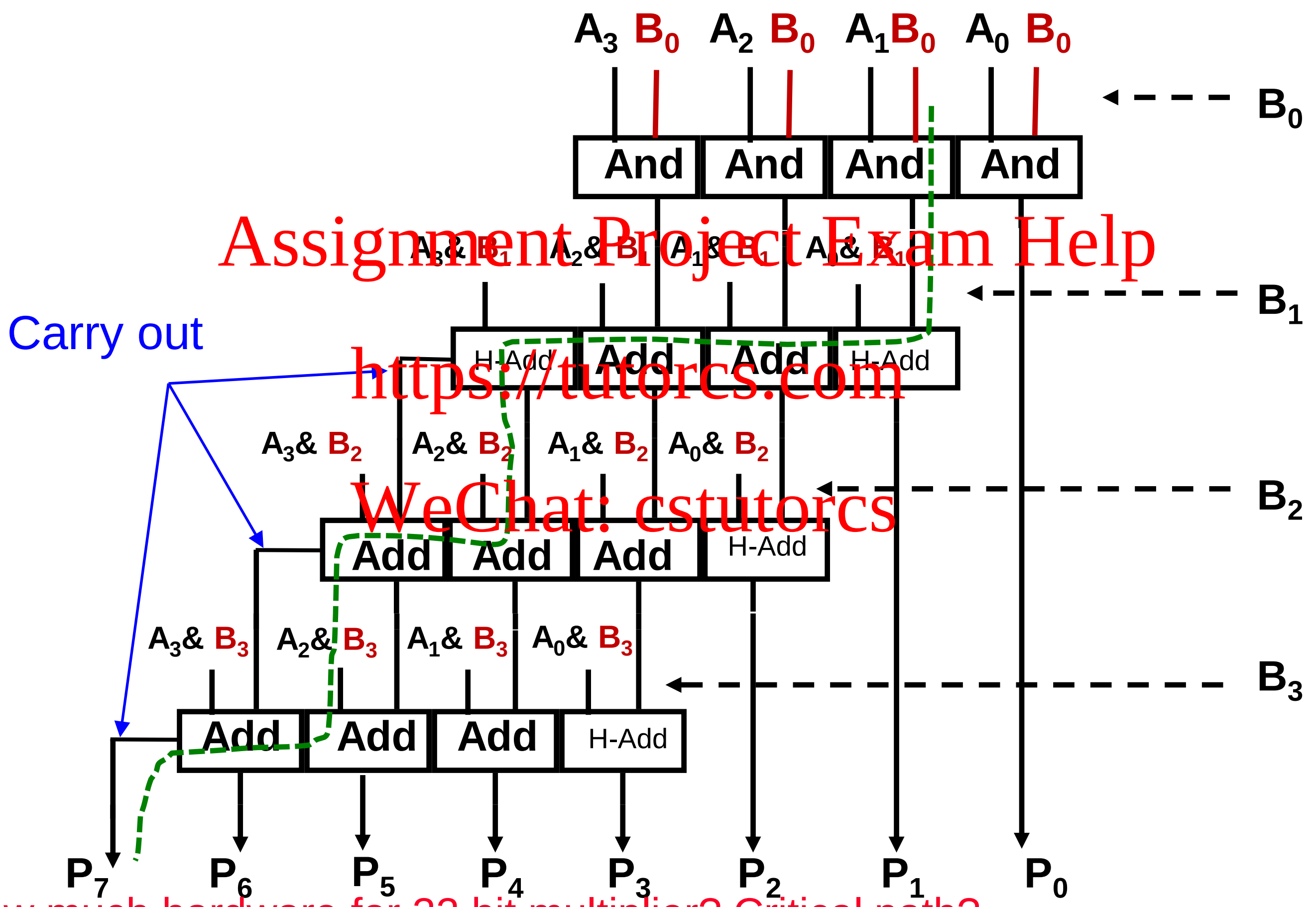
Multiplying N-bit number by M-bit number gives (N+M)-bit result

Easy part: forming partial products

(just an AND gate since B_I is either 0 or 1)

Hard part: adding M N-bit partial products

Unsigned Combinational (Parallel) Multiplier



- Q: How much hardware for 32 bit multiplier? Critical path?
- ((31 × 31)-1) 1-Bit Full Adders + 32 1-bit Half Adders+(32 × 32) 2-input AND gates
- Maximum t_{pd} = (32 + (2 × 30)) × (Adder delay) +(1 × (AND gate delay))

Problems

- For a 32bit adder the propagation delay is

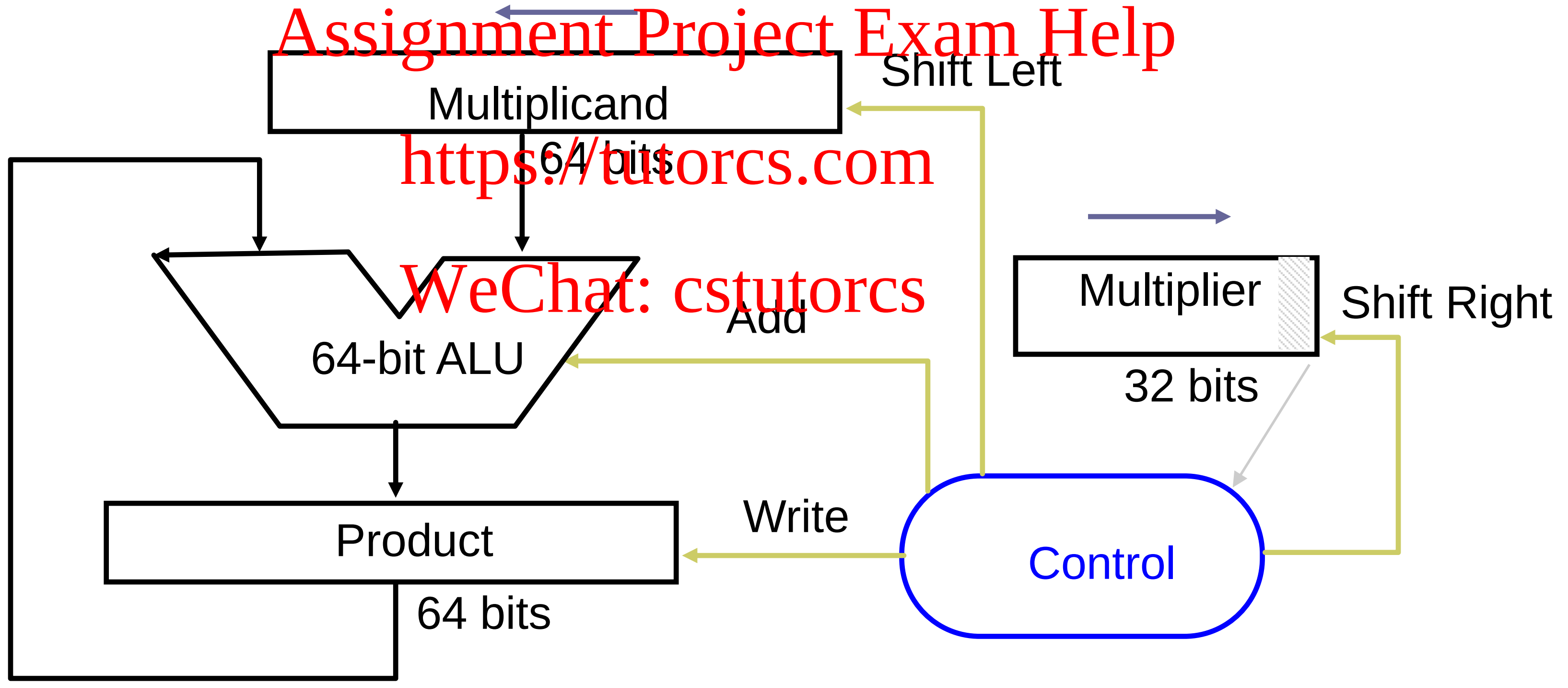
  Maximum t_{pd} = (32 + (2 × 30)) × (Adder delay) +(1 × (AND gate delay))

- In a sequential (clocked) system this would reduce the maximum clock speed as the system need to wait for the signals to propagate before the clock can be applied.

- One solution is to go for a sequential system rather than a combinational logic system.

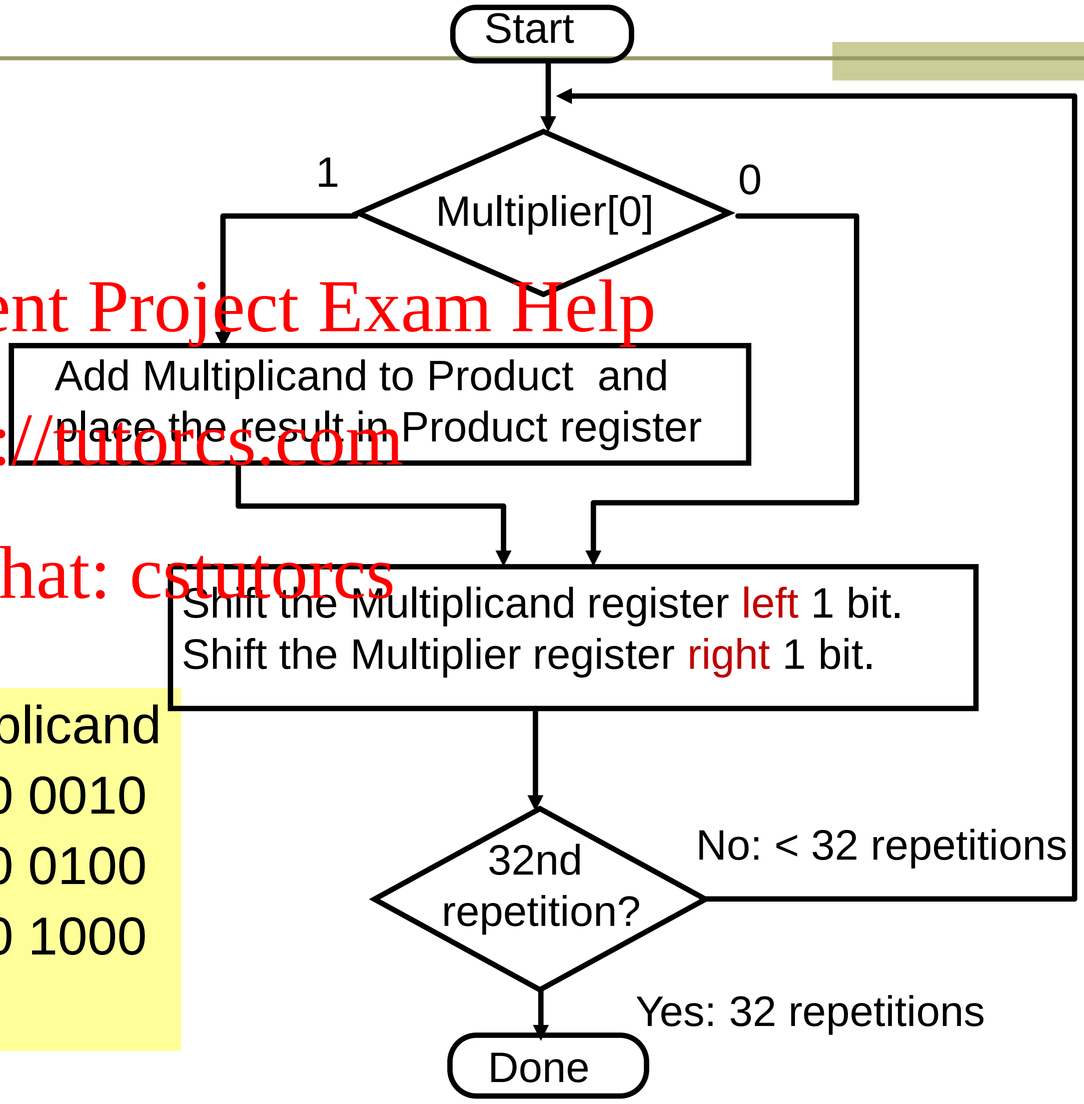
Unsigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control

Multiply Algorithm (version 1)


Start
Multiplier[0]
1
0
Add Multiplicand to Product and place the result in Product register
Shift the Multiplicand register left 1 bit.
Shift the Multiplier register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

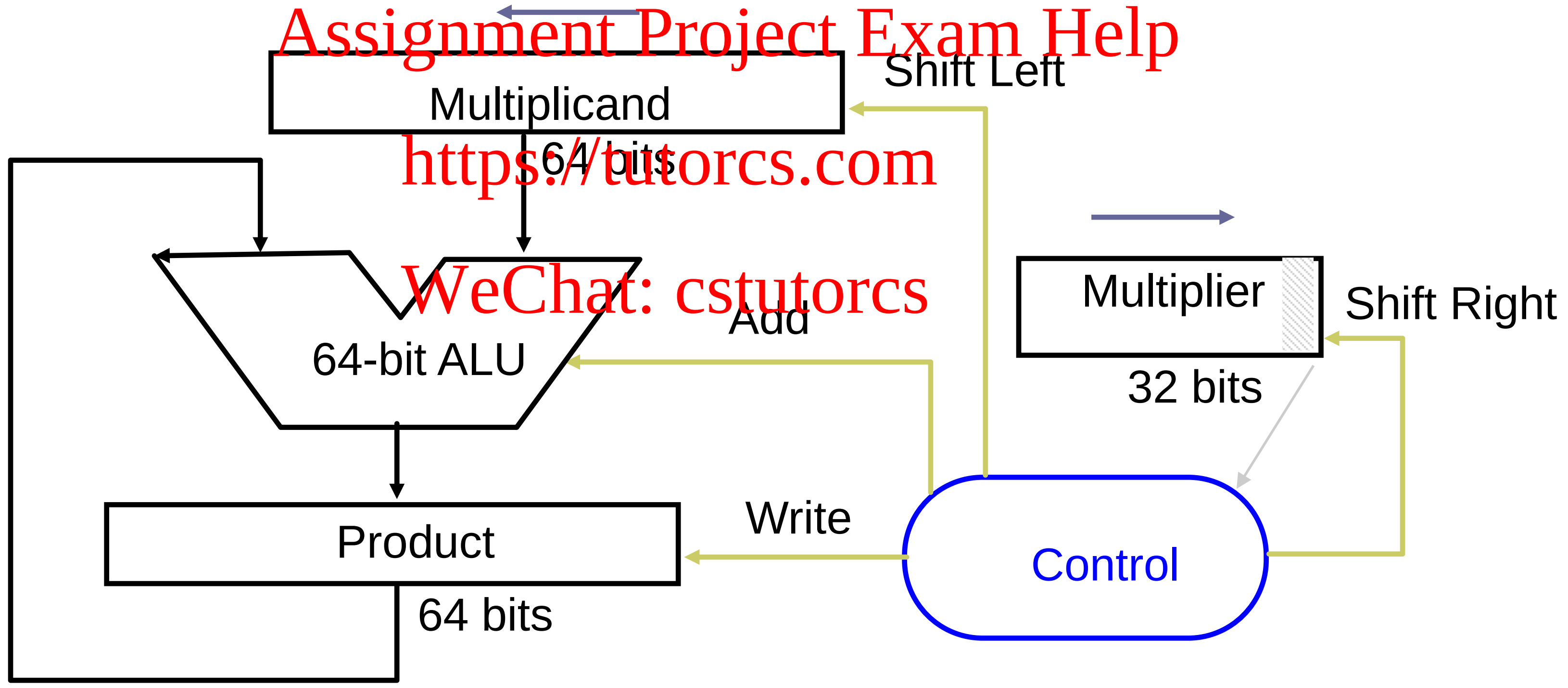
Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		

Observations on 32-bit Multiplier (version 1)

- 1 clock per cycle => 32×2=64 clocks per multiply
- Half of the bits in the multiplicand are always 0
  => 64-bit adder is wasted
- 0's are inserted in the right of Multiplicand as it is shifted => least significant bits of the Product register never changed once formed
- Instead of shifting multiplicand to left, shift product to right?

Unsigned shift-add multiplier (version 1)

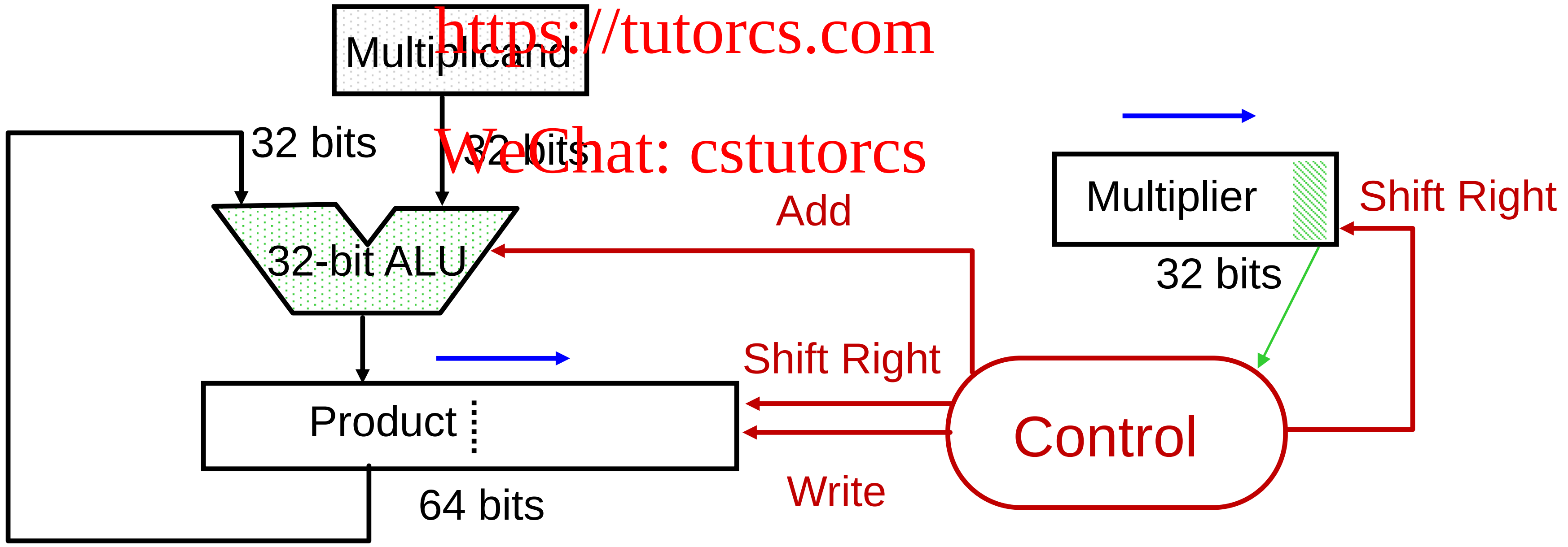
- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control

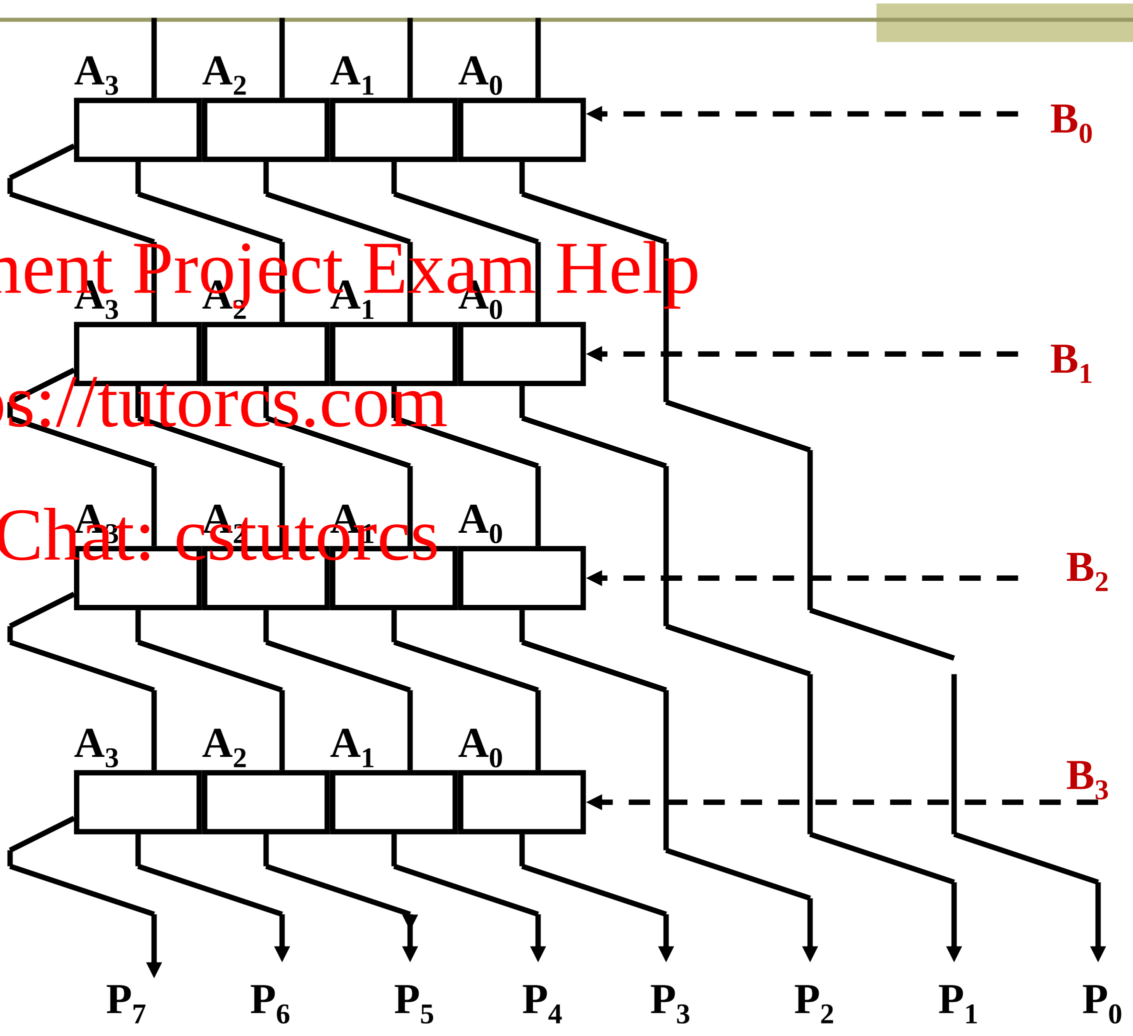
Multiply Hardware (version 2)

- 32-bit Multiplicand Reg, 32 -bit ALU, 64-bit Product Reg, 32-bit Multiplier Reg



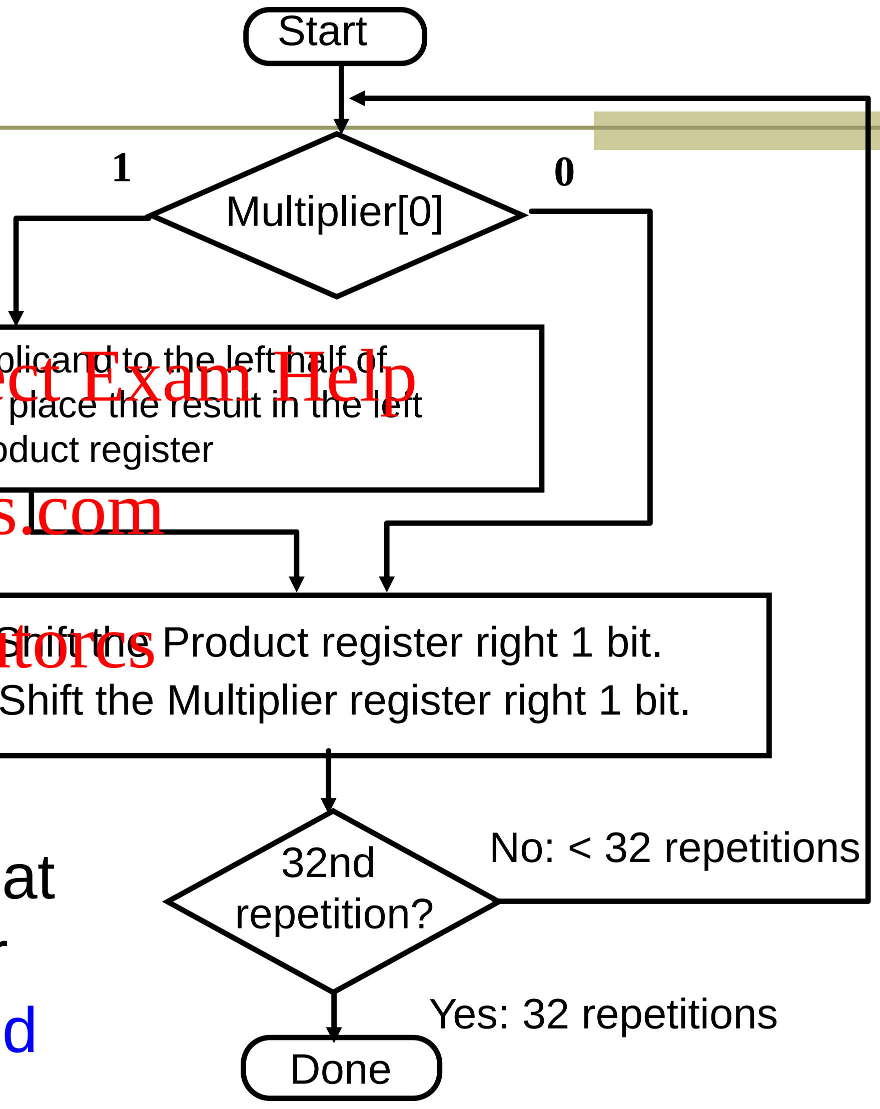
What’s going on?

Multiplicand stay’s still and Product moves right



Multiply Algorithm (version 2)

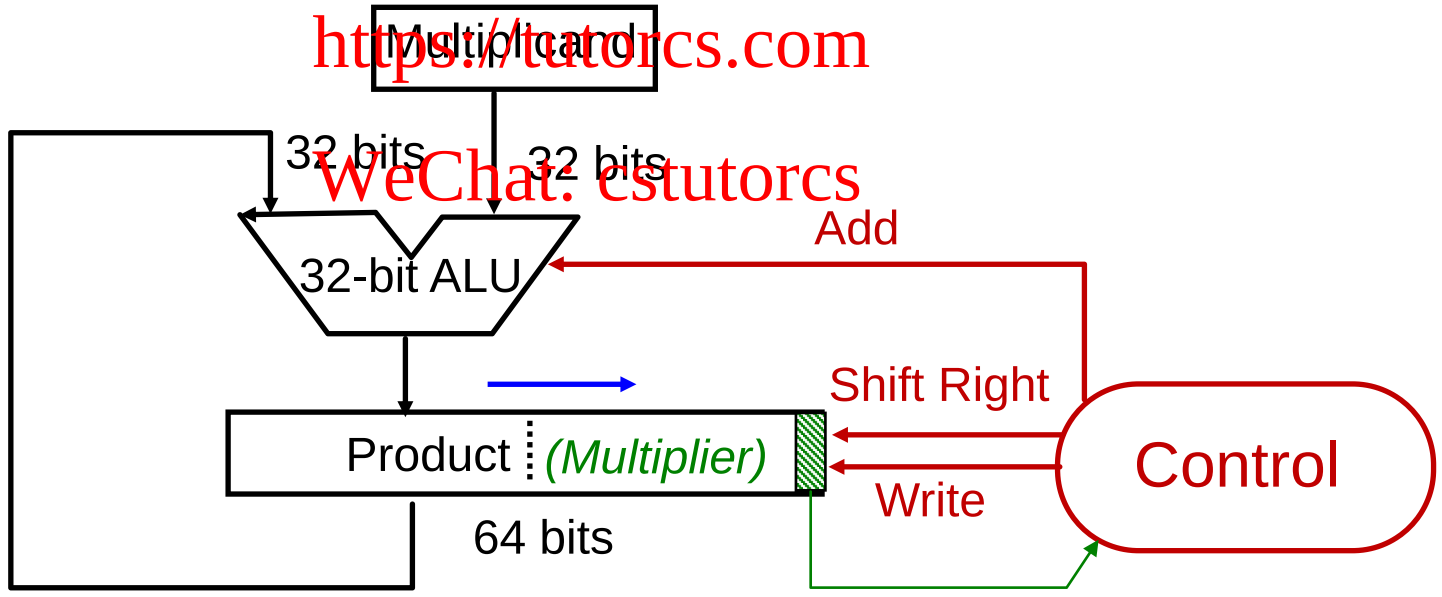
Product	Multiplier	Multiplicand
0000 0000	0011	0010
0010 0000		
0001 0000	0001	0010
0011 0000	0001	0010
0001 1000	0000	0010
0000 1100	0000	0010
0000 0110	0000	0010


Start
Multiplier[0]
1
0
Add multiplicand to the left half of product & place the result in the left half of Product register
Shift the Product register right 1 bit.
Shift the Multiplier register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

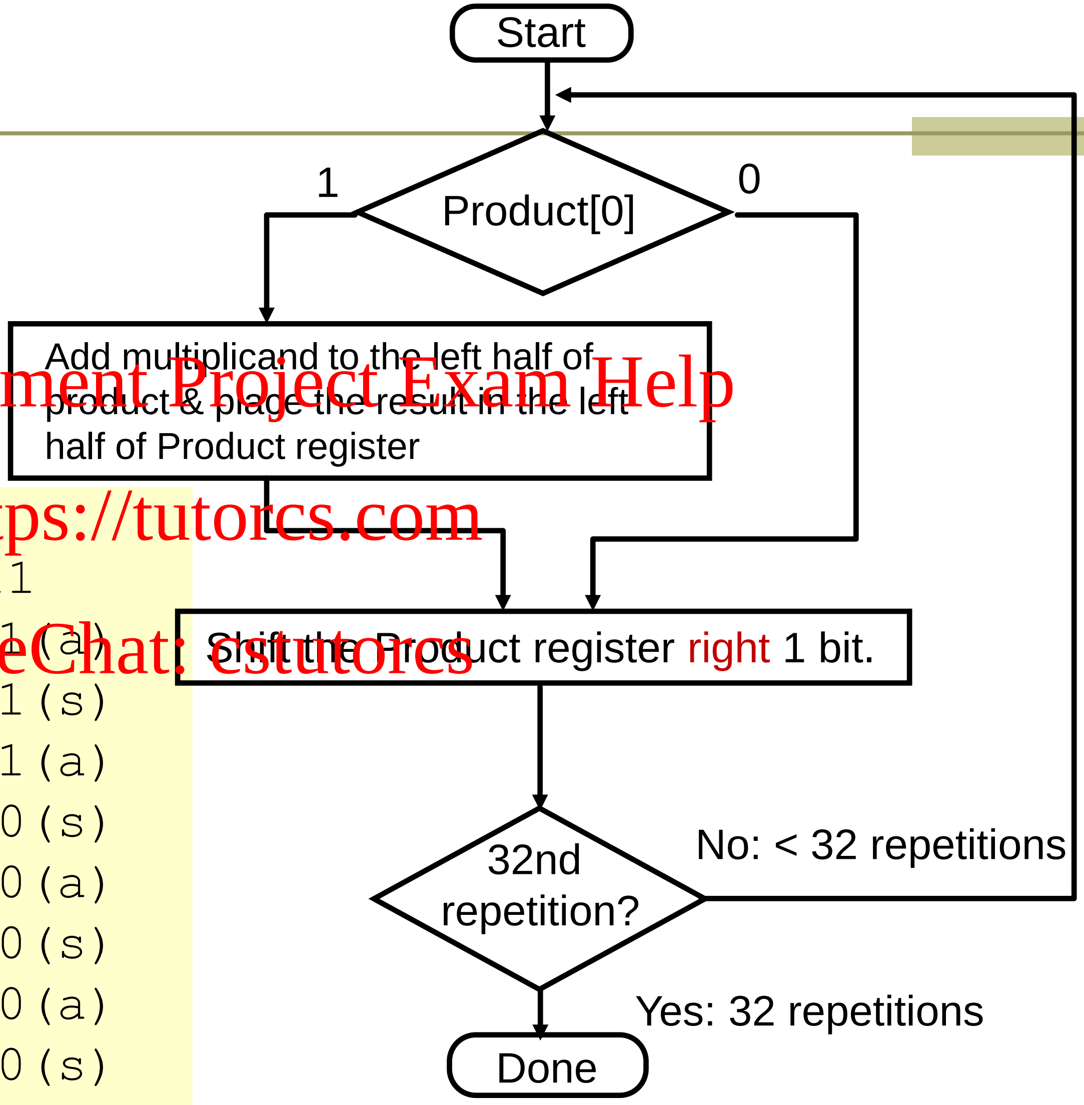
Product register wastes space that exactly matches size of multiplier
=> combine Multiplier register and Product register

Multiply Hardware (version 3)

- 32-bit Multiplicand Reg, 32 -bit ALU, 64-bit Product Reg, (No Multiplier Reg)



Multiply Algorithm (version 3)


Start
Product[0]
1
0
Add multiplicand to the left half of product & place the result in the left half of Product register
Shift the Product register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Multiplicand	Product
0010	0000 0011
0010	0010 0011(a)
0010	0001 0001(s)
0010	0011 0001(a)
0010	0001 1000(s)
0010	0001 1000(a)
0010	0000 1100(s)
0010	0000 1100(a)
0010	0000 0110(s)

Observations on Multiply (version 3)

- Less registers because Multiplier & Product are combined
- What about signed multiplication?
  1. The easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps)
  2. Apply definition of 2's complement
     1. need to sign-extend partial products and subtract at the end
  3. Booth's Algorithm is an elegant way to multiply signed numbers using the same hardware as before and saving cycles
     1. can handle multiple bits at a time

Motivation for Booth's Algorithm

Example 2 x 6 = 0010 x 0110:

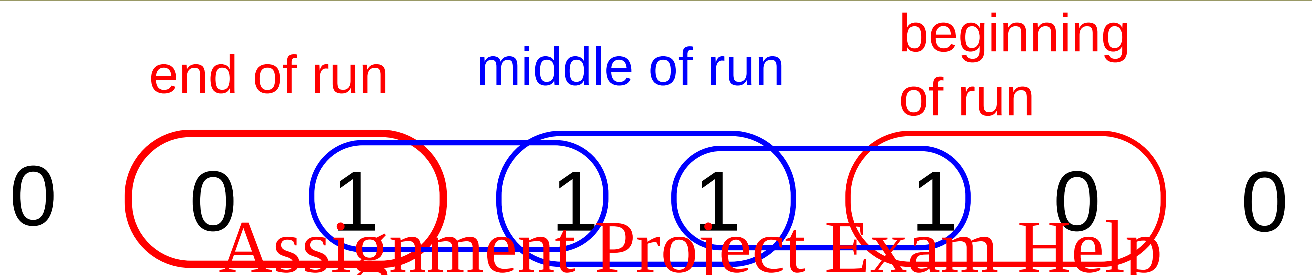
```
           0010
   x       0110
   ------------
   +       0000   shift (0 in multiplier)
   +      0010    add (1 in multiplier)
   +     0010     add (1 in multiplier)
   +    0000      shift (0 in multiplier)
   ------------
       00001100
```

ALU with add and subtract operations gets the same result in more than one way:

```
6        = – 2 + 8
0110     = – 0010 + 1000 = 11110 + 01000
         (-2+8) x2 = -2x2 + 8x2 = 12
```

```
           0010   (multiplicand)
   x       0110   (multiplier)
   ------------
           0000   shift (0 in multiplier)
   –      0010    sub (first 1 in multiplier)
         0000     shift (mid string of 1s) .
   +    0010      add (prior step had last 1)
   ------------
       00001100
```

Booth’s Algorithm



Current Bit	Bit to the Right	Explanation	Example	Operation
1	0	Begins run of 1s	0001111000	sub + shift
1	1	Middle of run of 1s	0001111000	shift
1	1	Middle of run of 1s	0001111000	shift
1	1	Middle of run of 1s	0001111000	shift
0	1	End of run of 1s	0001111000	add + shift
0	0	Middle of run of 0s	0001111000	shift

Originally for Speed (when shift was faster than add)

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one

Example (2 × 7)

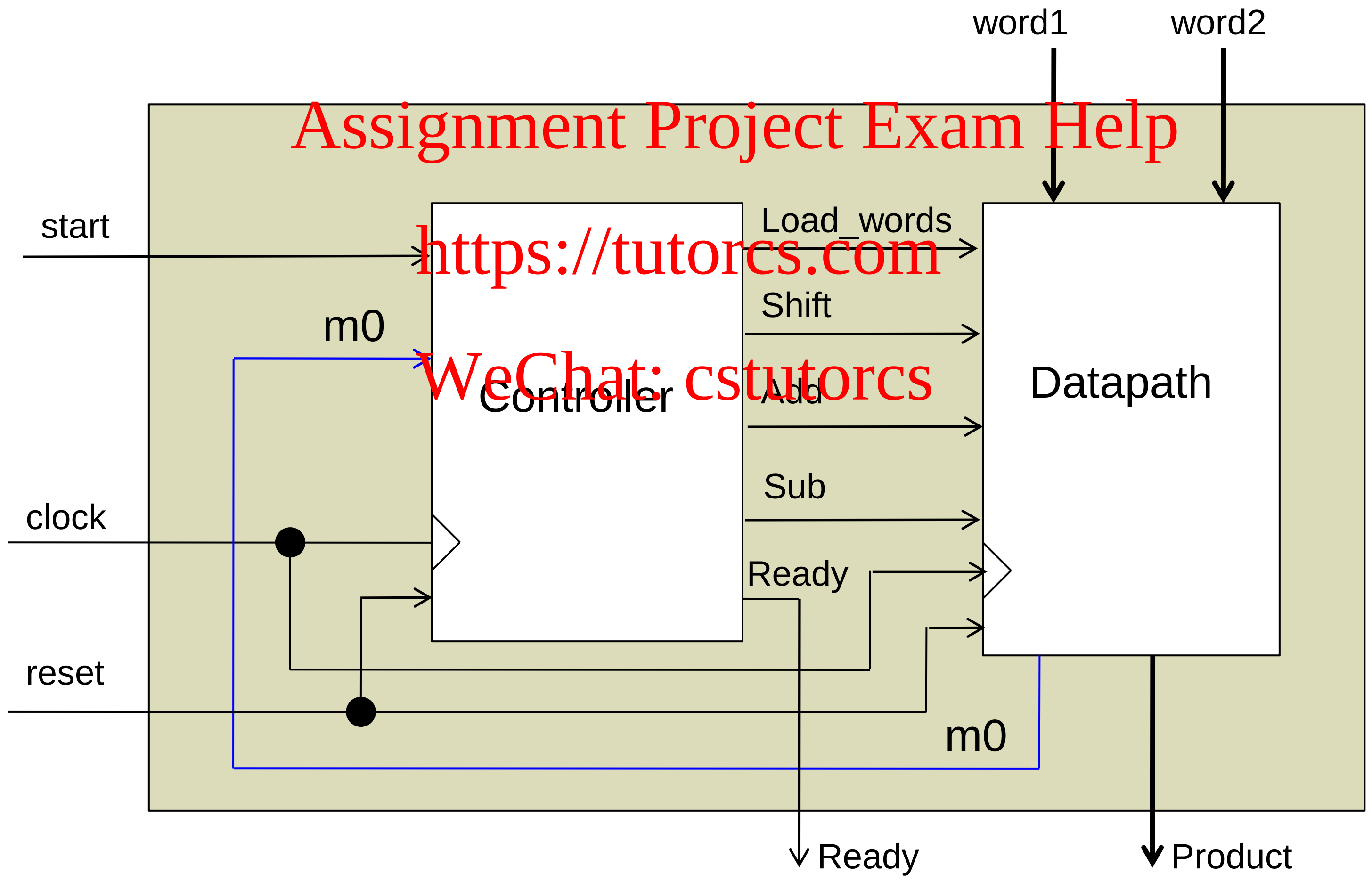
Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	subtract
1a. P = P - m	1110	+ 1110	
		1110 0111 0	shift P (sign extend)
1b.	0010	1111 0011 1	nop, shift (sign extend)
2.	0010	1111 1001 1	nop, shift (sign extend)
3.	0010	1111 1100 1	add
4a.	0010	+ 0010	
		0001 1100 1	shift (sign extend)
4b.	0010	0000 1110 0	done

$(1110)_2 = 14$

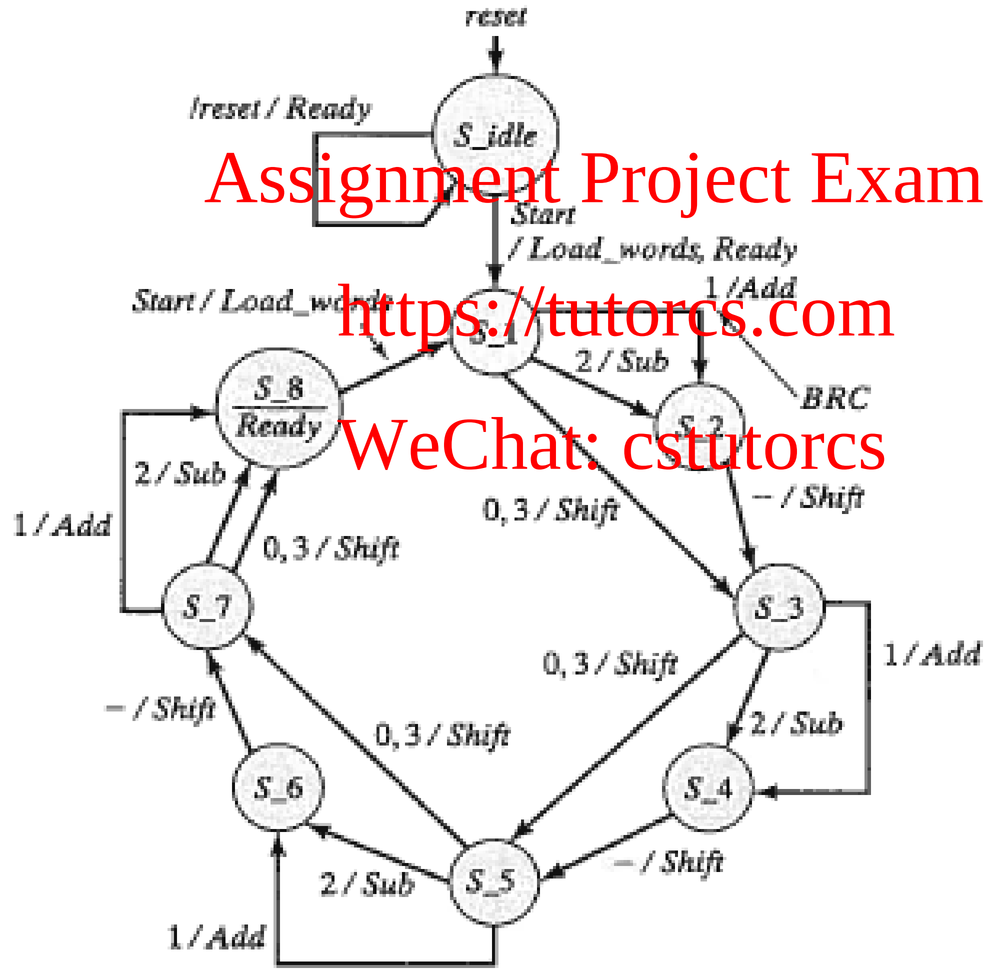
Example (2 × (-3))

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 110 1 0	10 -> sub
1a. P = P - m	1110	+ 1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1	01 -> add
		+ 0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 101 1 0	10 -> sub
		+ 1110	
3a.	0010	1110 1011 0	shift
3b.	0010	1111 010 1 1	11 -> nop
4a.		1111 0101 1	shift
4b.	0010	$\boxed{1111\ 1010\ 1}$	done

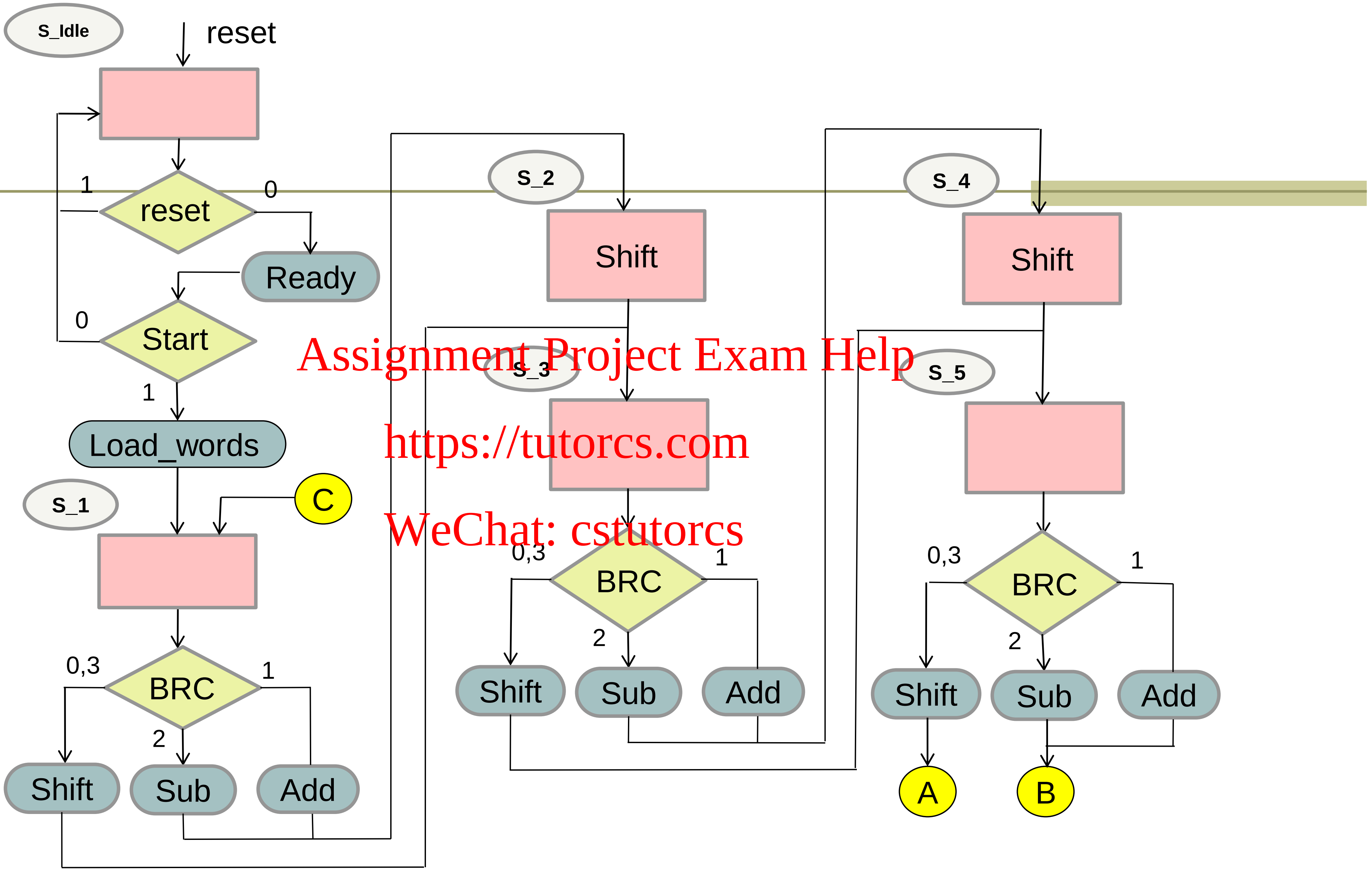
Structural Units of Booth's Multiplier

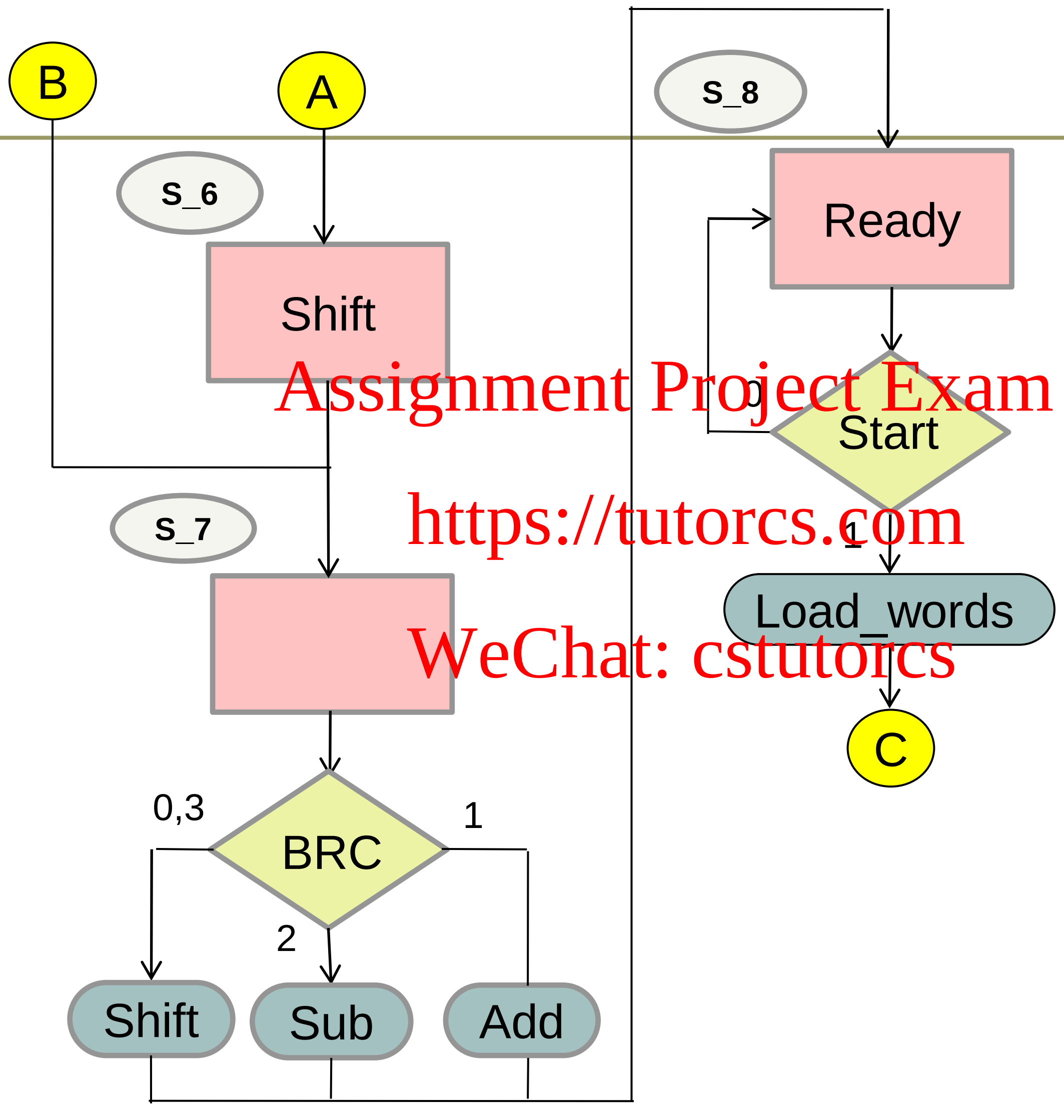


State Transition Graph (STG) for a 4-bit Booth Sequential Multiplier

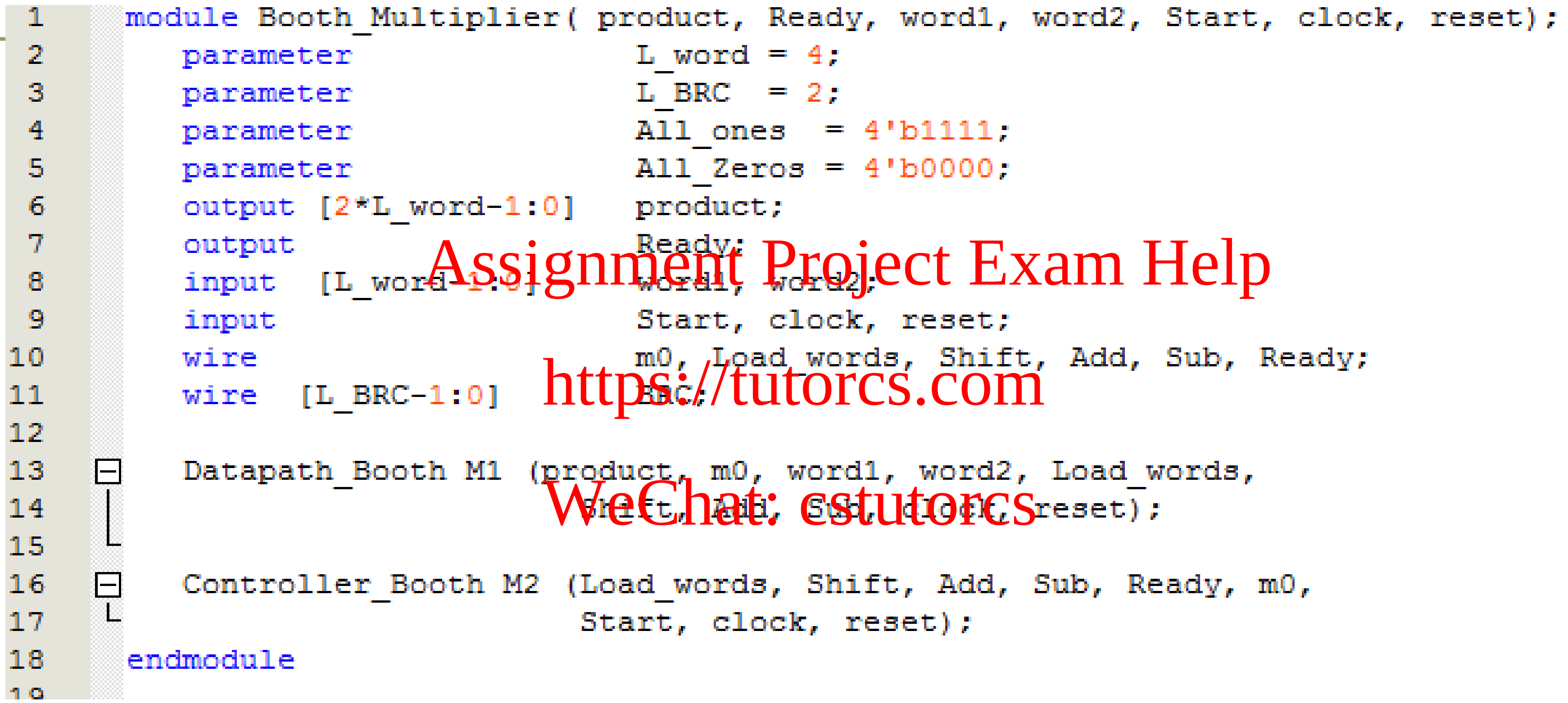


The value of two successive bits (mi, mi-1) determines the Booth recoded multiplier bit, BRCi.


S_Idle
reset
1
reset
0
Ready
0
Start
1
Load_words
S_1
C
0,3
BRC
1
2
Shift
Sub
Add
S_2
Shift
S_3
0,3
BRC
1
2
Shift
Sub
Add
S_4
Shift
S_5
0,3
BRC
1
2
Shift
Sub
Add
A
B

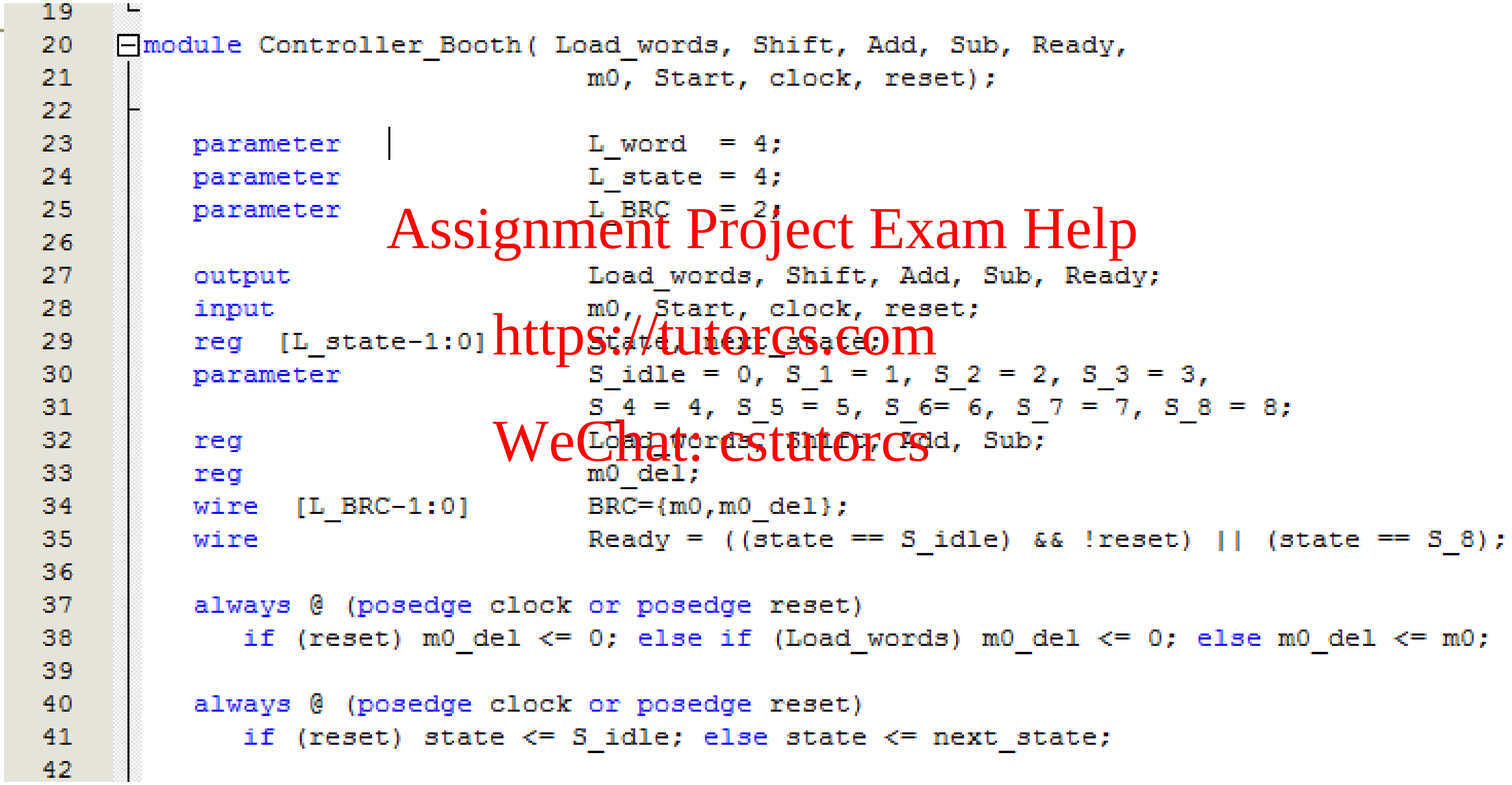

B
A
S_6
Shift
S_7
BRC
0,3
1
2
Shift
Sub
Add
S_8
Ready
0
Start
1
Load_words
C

Verilog Code for Booth’s Algorithm (1)



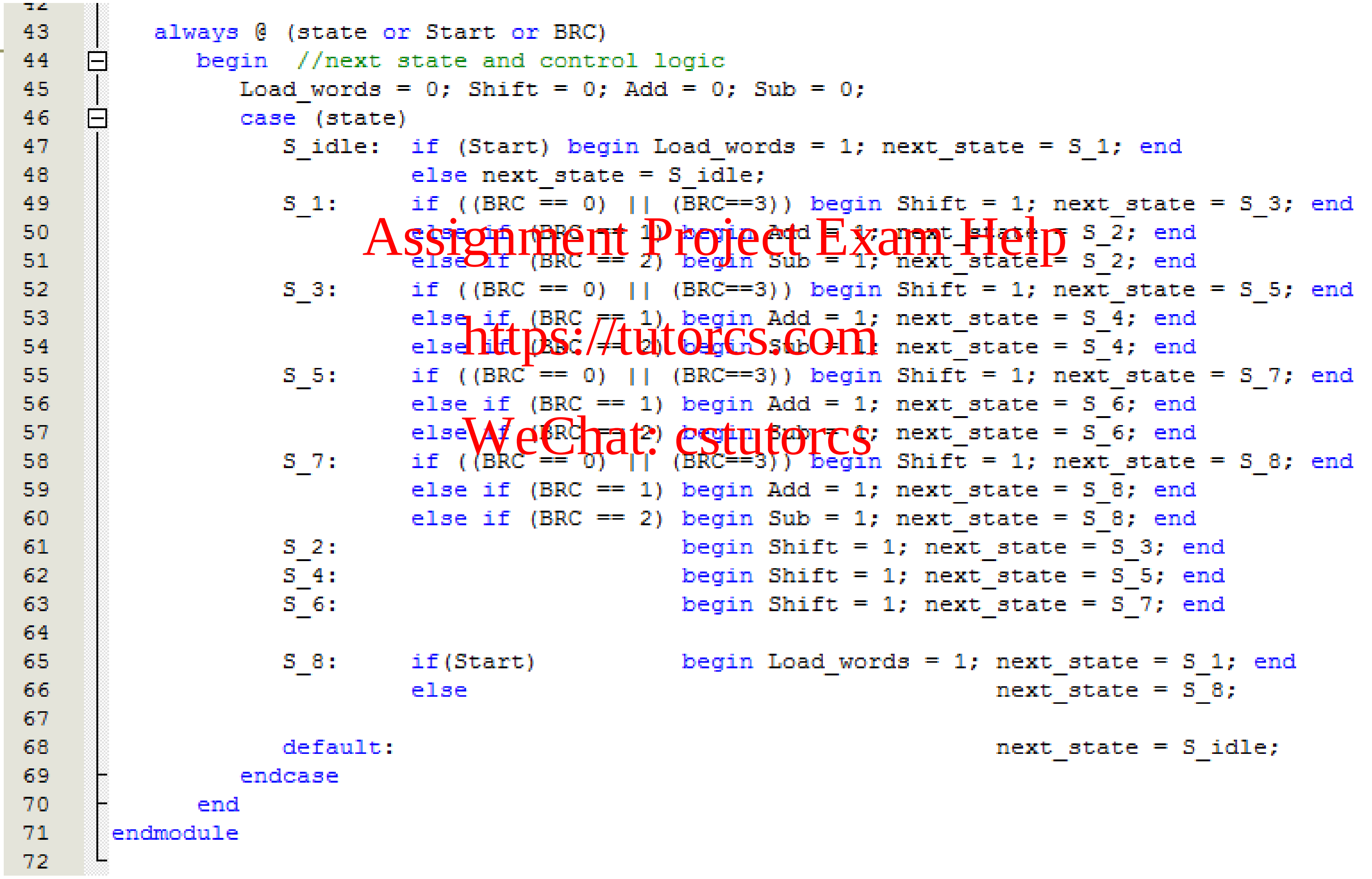
```
module Booth_Multiplier( product, Ready, word1, word2, Start, clock, reset);
   parameter                L_word = 4;
   parameter                L_BRC  = 2;
   parameter                All_ones   = 4'b1111;
   parameter                All_Zeros = 4'b0000;
   output  [2*L_word-1:0]   product;
   output                   Ready;
   input   [L_word-1:0]     word1, word2;
   input                    Start, clock, reset;
   wire                     m0, Load_words, Shift, Add, Sub, Ready;
   wire   [L_BRC-1:0]       BRC;

   Datapath_Booth M1  (product, m0, word1, word2, Load_words,
                          Shift, Add, Sub, clock, reset);

   Controller_Booth M2  (Load_words, Shift, Add, Sub, Ready, m0,
                          Start, clock, reset);
endmodule
```

Verilog Code for Booth's Algorithm (2)

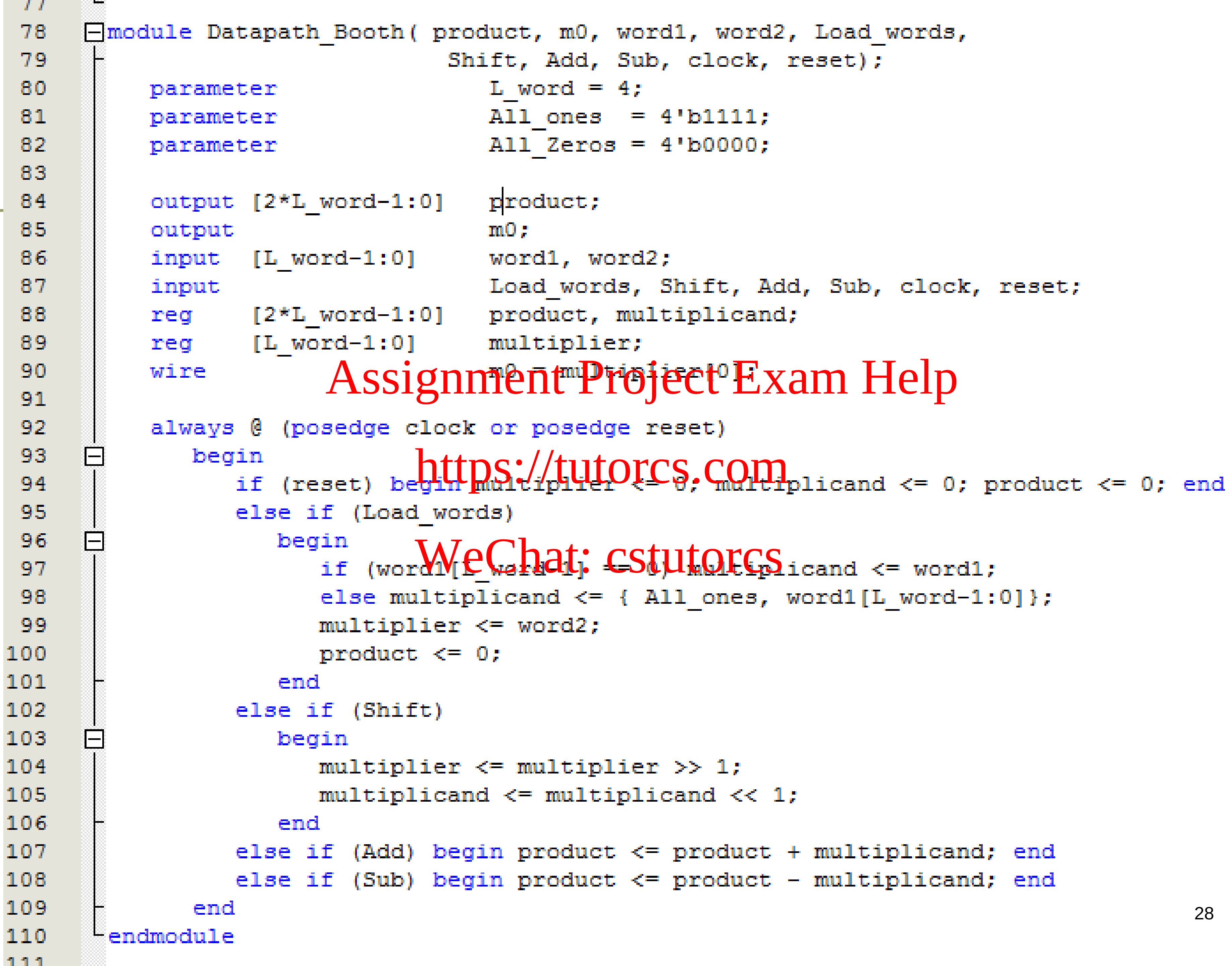


```verilog

module Controller_Booth( Load_words, Shift, Add, Sub, Ready,
                         m0, Start, clock, reset);

  parameter                L_word  = 4;
  parameter                L_state = 4;
  parameter                L_BRC   = 2;

  output                   Load_words, Shift, Add, Sub, Ready;
  input                    m0, Start, clock, reset;
  reg   [L_state-1:0]      state, next_state;
  parameter                S_idle = 0, S_1 = 1, S_2 = 2, S_3 = 3,
                           S_4 = 4, S_5 = 5, S_6= 6, S_7 = 7, S_8 = 8;
  reg                      Load_words, Shift, Add, Sub;
  reg                      m0_del;
  wire  [L_BRC-1:0]        BRC={m0,m0_del};
  wire                     Ready = ((state == S_idle) && !reset) || (state == S_8);

  always @ (posedge clock or posedge reset)
     if (reset) m0_del <= 0; else if (Load_words) m0_del <= 0; else m0_del <= m0;

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

```

Verilog Code for Booth’s Algorithm (3)

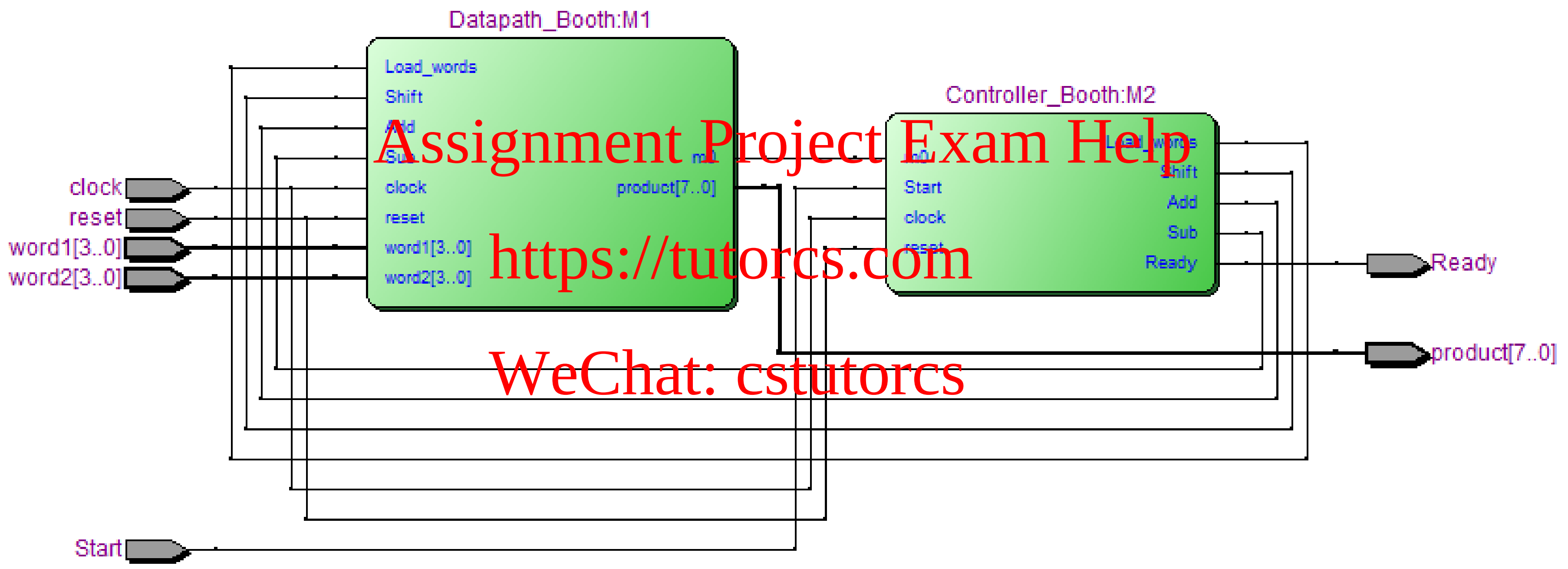


```verilog
   always @ (state or Start or BRC)
      begin  //next state and control logic
         Load_words = 0; Shift = 0; Add = 0; Sub = 0;
         case (state)
            S_idle:  if (Start) begin Load_words = 1; next_state = S_1; end
                     else next_state = S_idle;
            S_1:     if ((BRC == 0) || (BRC==3)) begin Shift = 1; next_state = S_3; end
                     else if (BRC == 1) begin Add = 1; next_state = S_2; end
                     else if (BRC == 2) begin Sub = 1; next_state = S_2; end
            S_3:     if ((BRC == 0) || (BRC==3)) begin Shift = 1; next_state = S_5; end
                     else if (BRC == 1) begin Add = 1; next_state = S_4; end
                     else if (BRC == 2) begin Sub = 1; next_state = S_4; end
            S_5:     if ((BRC == 0) || (BRC==3)) begin Shift = 1; next_state = S_7; end
                     else if (BRC == 1) begin Add = 1; next_state = S_6; end
                     else if (BRC == 2) begin Sub = 1; next_state = S_6; end
            S_7:     if ((BRC == 0) || (BRC==3)) begin Shift = 1; next_state = S_8; end
                     else if (BRC == 1) begin Add = 1; next_state = S_8; end
                     else if (BRC == 2) begin Sub = 1; next_state = S_8; end
            S_2:                        begin Shift = 1; next_state = S_3; end
            S_4:                        begin Shift = 1; next_state = S_5; end
            S_6:                        begin Shift = 1; next_state = S_7; end

            S_8:     if(Start)          begin Load_words = 1; next_state = S_1; end
                     else                                     next_state = S_8;

            default:                                          next_state = S_idle;
         endcase
      end
endmodule
```

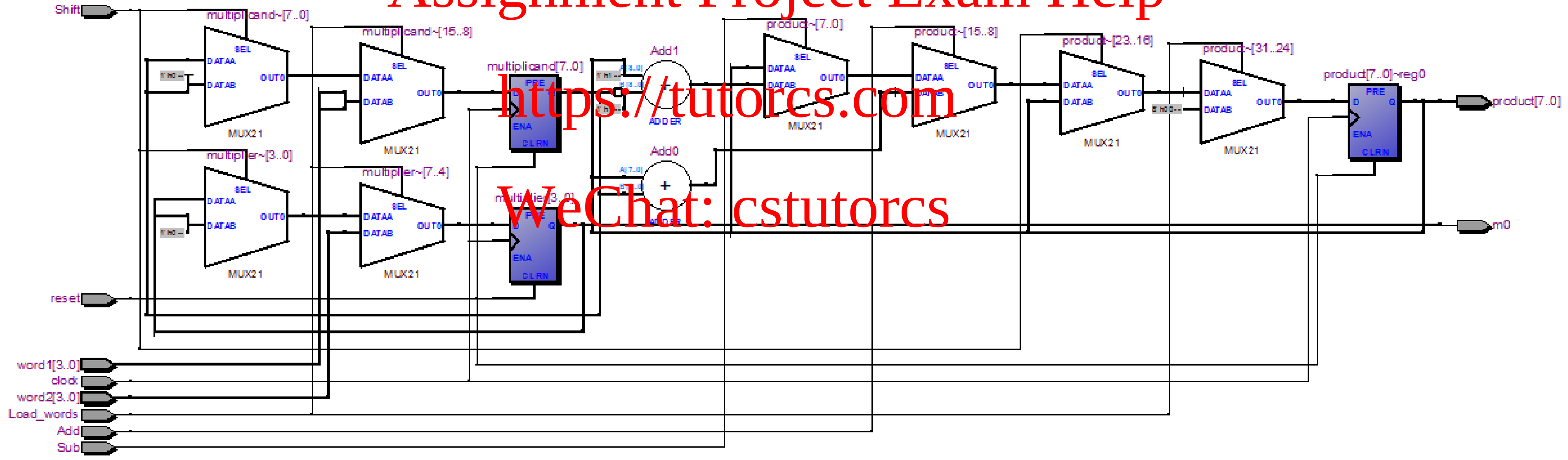


```
module Datapath_Booth( product, m0, word1, word2, Load_words,
                        Shift, Add, Sub, clock, reset);
   parameter                 L_word = 4;
   parameter                 All_ones  = 4'b1111;
   parameter                 All_Zeros = 4'b0000;

   output [2*L_word-1:0]     product;
   output                    m0;
   input  [L_word-1:0]       word1, word2;
   input                     Load_words, Shift, Add, Sub, clock, reset;
   reg    [2*L_word-1:0]     product, multiplicand;
   reg    [L_word-1:0]       multiplier;
   wire                      m0 = multiplier[0];

   always @ (posedge clock or posedge reset)
      begin
         if (reset) begin multiplier <= 0; multiplicand <= 0; product <= 0; end
         else if (Load_words)
            begin
               if (word1[L_word-1] == 0) multiplicand <= word1;
               else multiplicand <= { All_ones, word1[L_word-1:0]};
               multiplier <= word2;
               product <= 0;
            end
         else if (Shift)
            begin
               multiplier <= multiplier >> 1;
               multiplicand <= multiplicand << 1;
            end
         else if (Add) begin product <= product + multiplicand; end
         else if (Sub) begin product <= product - multiplicand; end
      end
endmodule
```

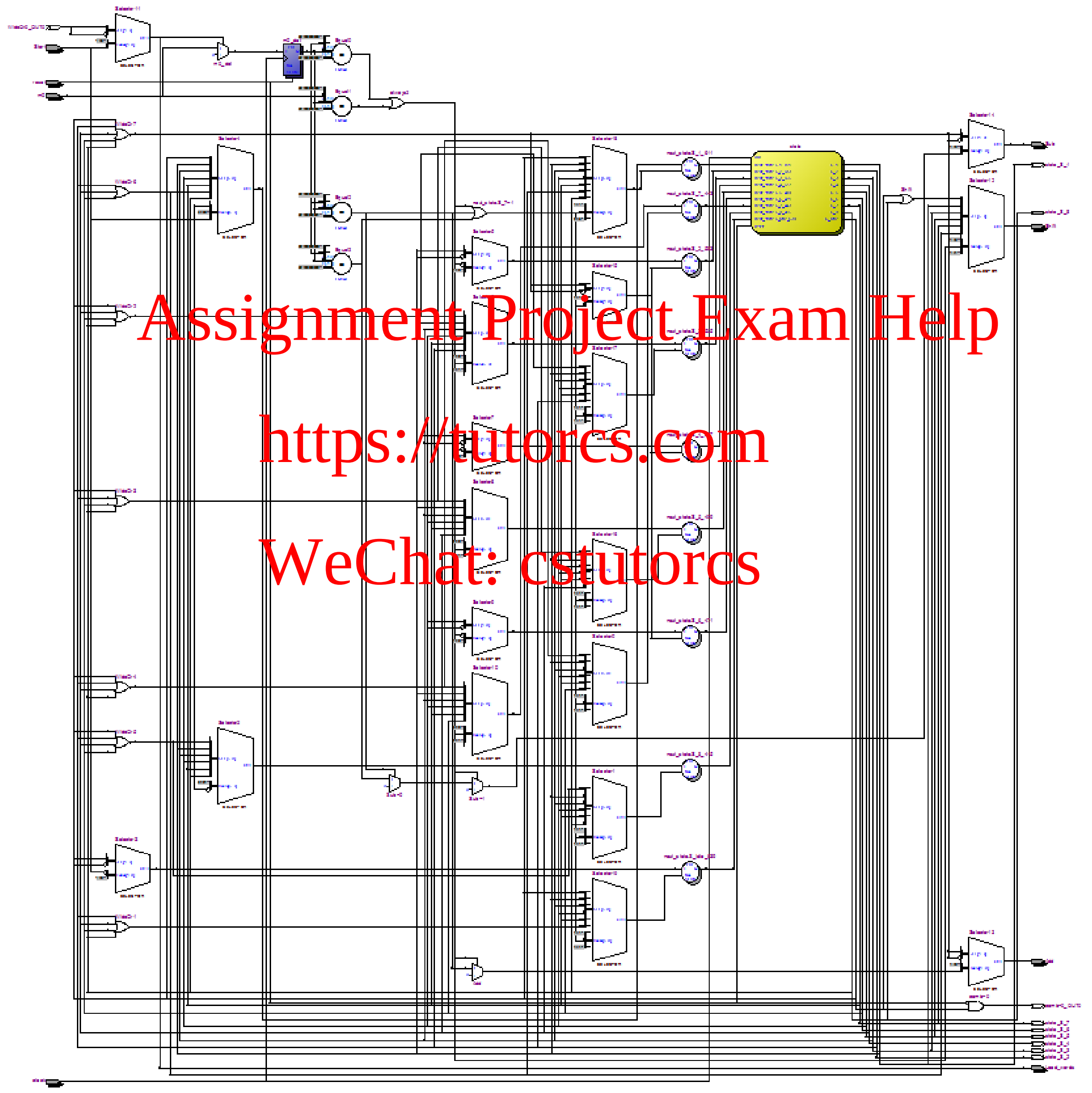
RTL View of Booth's Multiplier



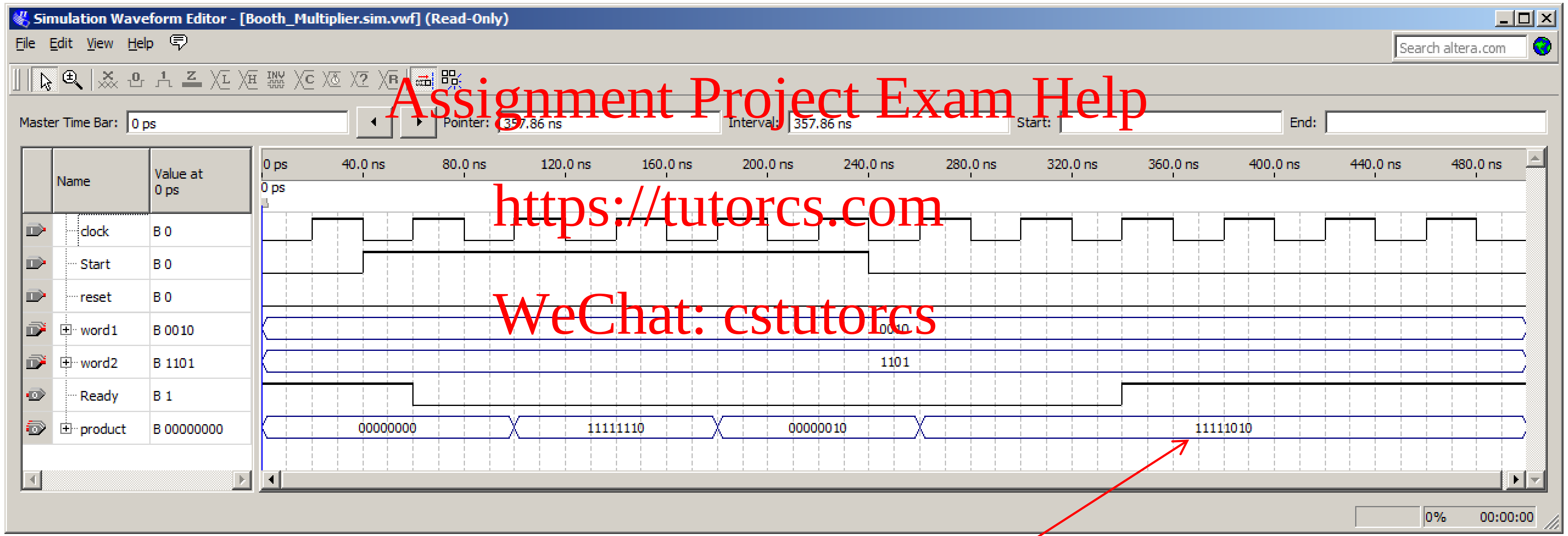
The Datapath



The Controller

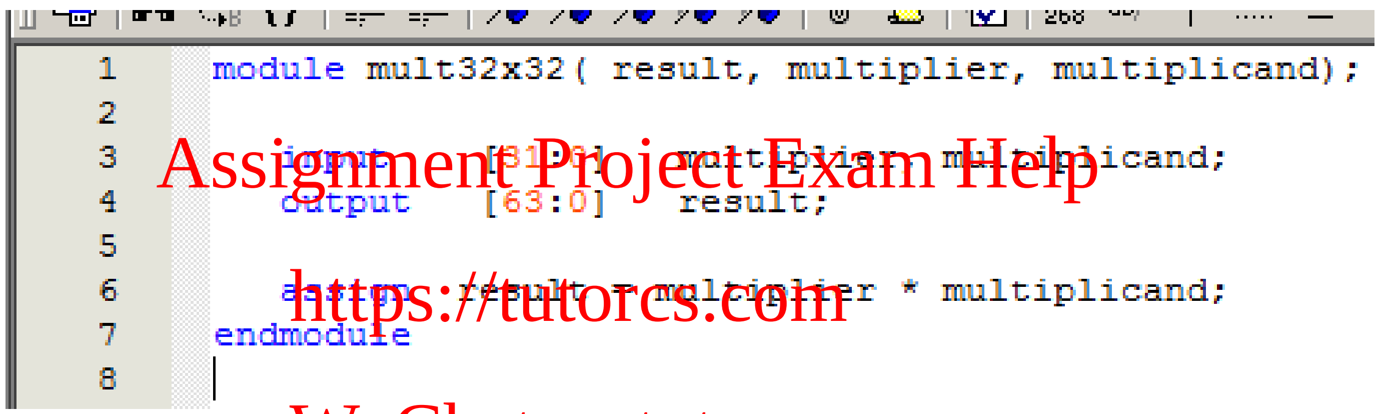


Booth’s Simulation

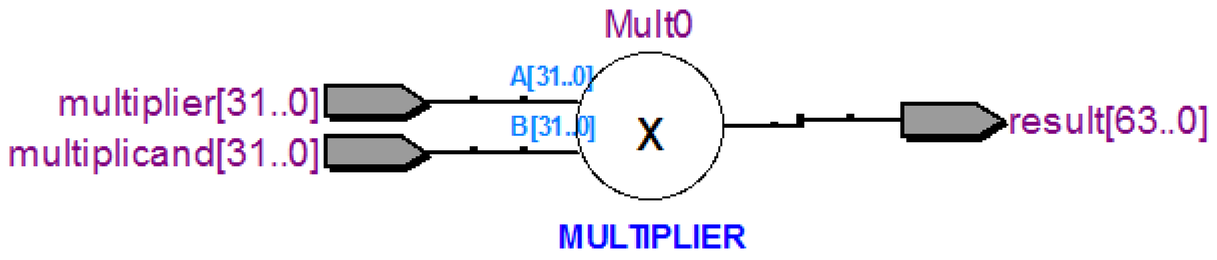


one input is negative: 0010 X 1101 = 11111010 (2 * -3 = -6)

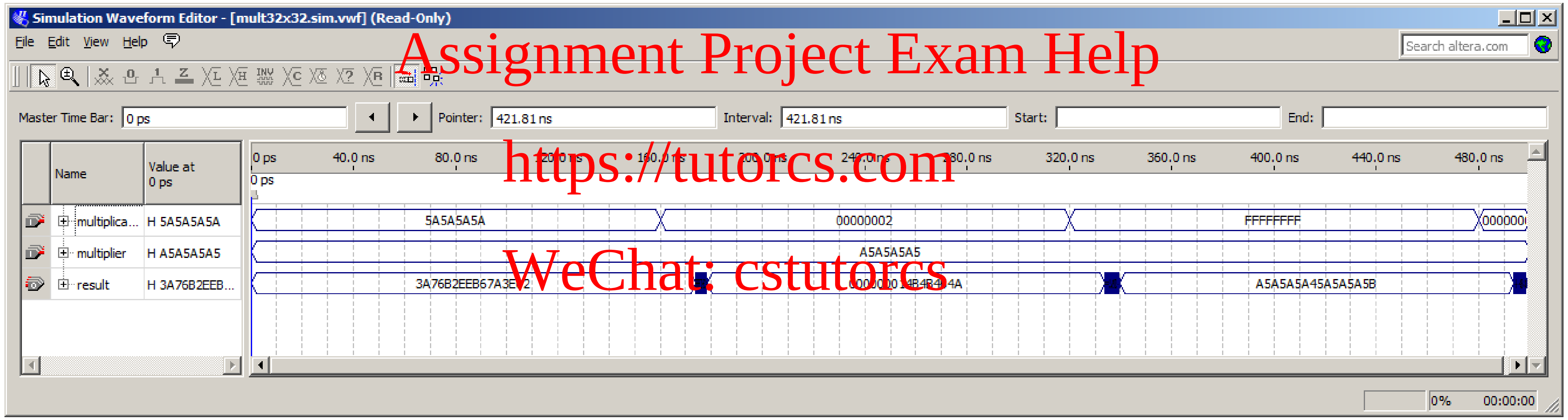
Verilog Quartus implementation



```
module mult32x32( result, multiplier, multiplicand);

    input     [31:0]    multiplier, multiplicand;
    output    [63:0]    result;

    assign result = multiplier * multiplicand;
endmodule
```

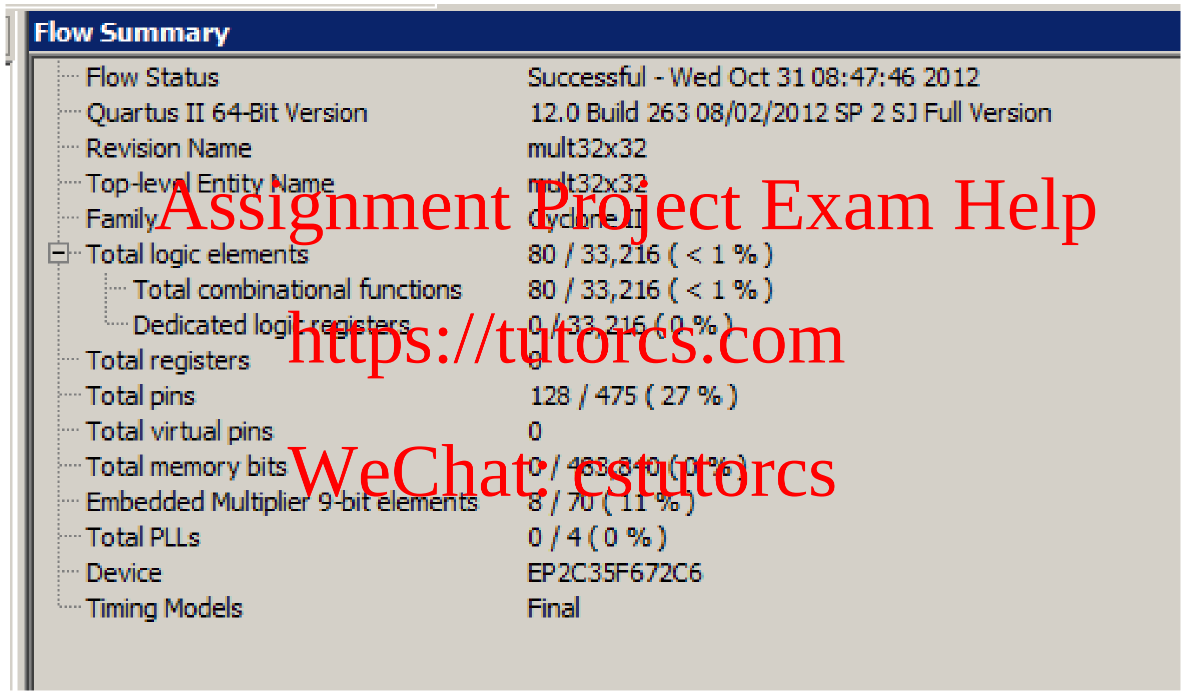


Simulation results



Propagation time is about 18ns. Is that what you would expect for a purely combination circuit?

Resources used

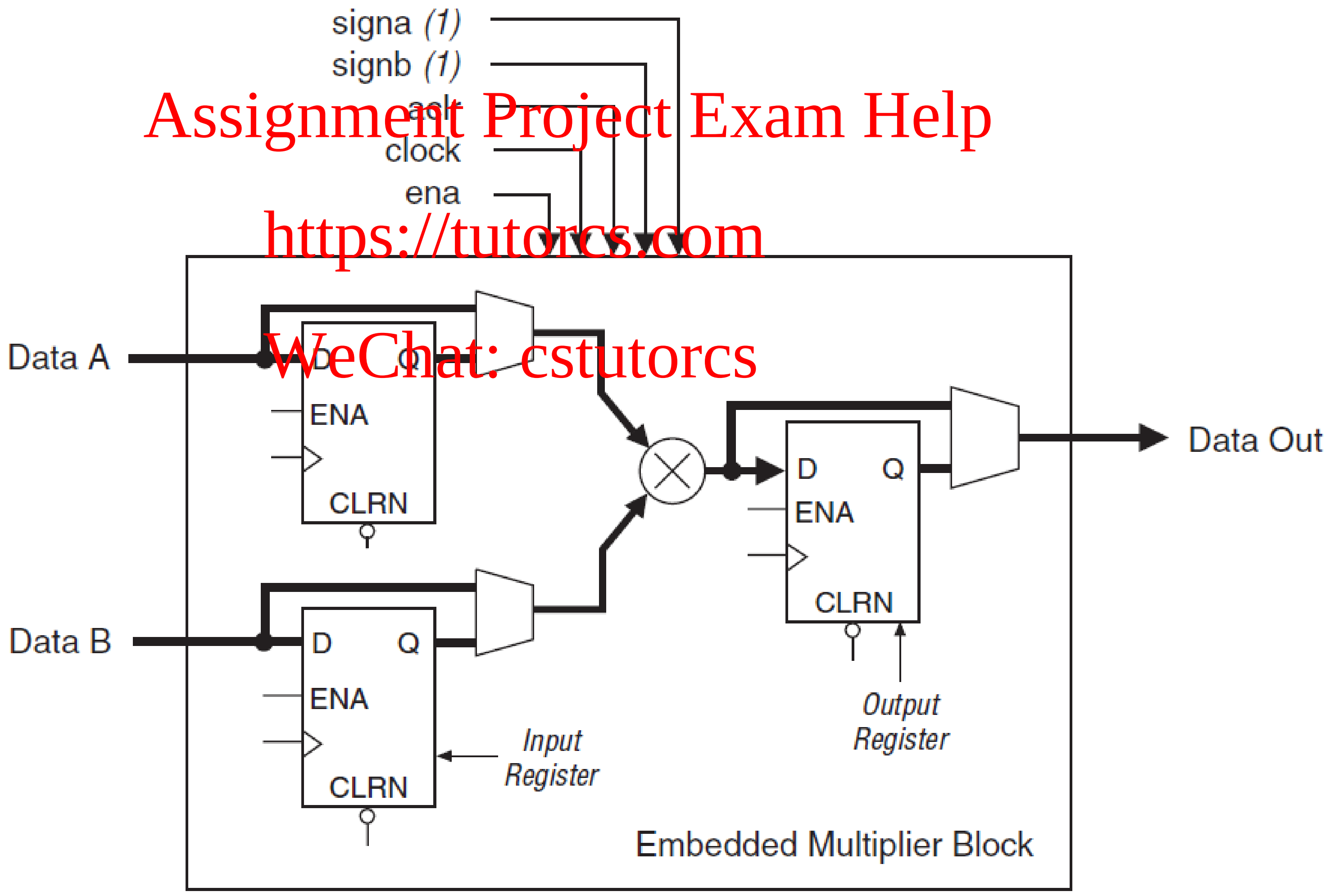


Flow Summary

Flow Status	Successful - Wed Oct 31 08:47:46 2012
Quartus II 64-Bit Version	12.0 Build 263 08/02/2012 SP 2 SJ Full Version
Revision Name	mult32x32
Top-level Entity Name	mult32x32
Family	Cyclone II
Total logic elements	80 / 33,216 (< 1 %)
Total combinational functions	80 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	128 / 475 (27 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	8 / 70 (11 %)
Total PLLs	0 / 4 (0 %)
Device	EP2C35F672C6
Timing Models	Final

Note the use of 8 embedded 9 bit multipliers

Altera's Embedded Multiplier


signa (1)
signb (1)
aclr
clock
ena
Data A
Data B
D
Q
ENA
CLRN
Input Register
Output Register
Data Out
Embedded Multiplier Block