Signed / Unsigned MPY

DD Lab8

助教:徐瑋程、林冠翰、胡祐嘉、劉宸彥

Department of Electrical Engineering and SoC Research Center National Chung Cheng University

Outline

- 課程目的
- Unsigned MPY 架構
- Signed MPY 原理與架構
- Lab作業
- 課程評分

課程目的

經過了先前的實驗課，我們已經了解如何透過多個全加器設計RCA及CLA架構，而在數位系統導論課程中也學習了無號乘法器與有號乘法的原理，我們將會帶大家複習上述內容，並讓大家學習：

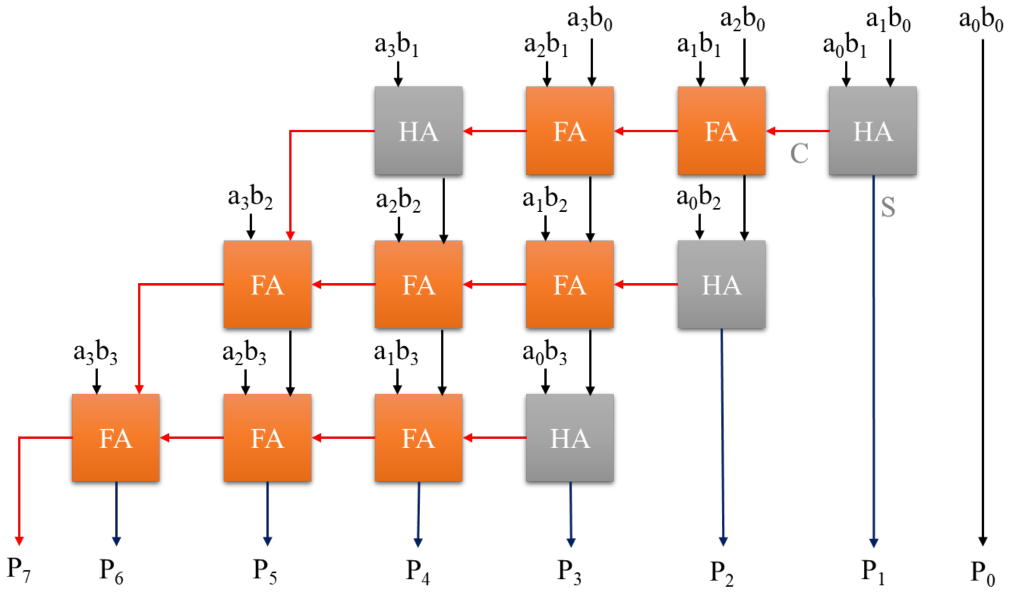
- 透過全加器與半加器設計無號乘法器（Array Multiplier）
- 透過Separate sign handling、Sign extension、Baugh-Wooley algorithm與Booth recoding設計有號數乘法器

Array Multiplier

- Array Multiplier 是一個計算方式與直式乘法相似的硬體架構，這邊以4-bit MPY作為範例，總共需要16個and，4個HA，8個FA來實現。
- 先將被乘數與各個乘數的bit相乘得到該部分的乘積，再透過加法器將前文得出的部分乘積與carry bit加總得到乘法結果。

					a_3	a_2	a_1	a_0
				x)	b_3	b_2	b_1	b_0
					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			
	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

- 直式乘法示意圖



- Array Multiplier架構圖

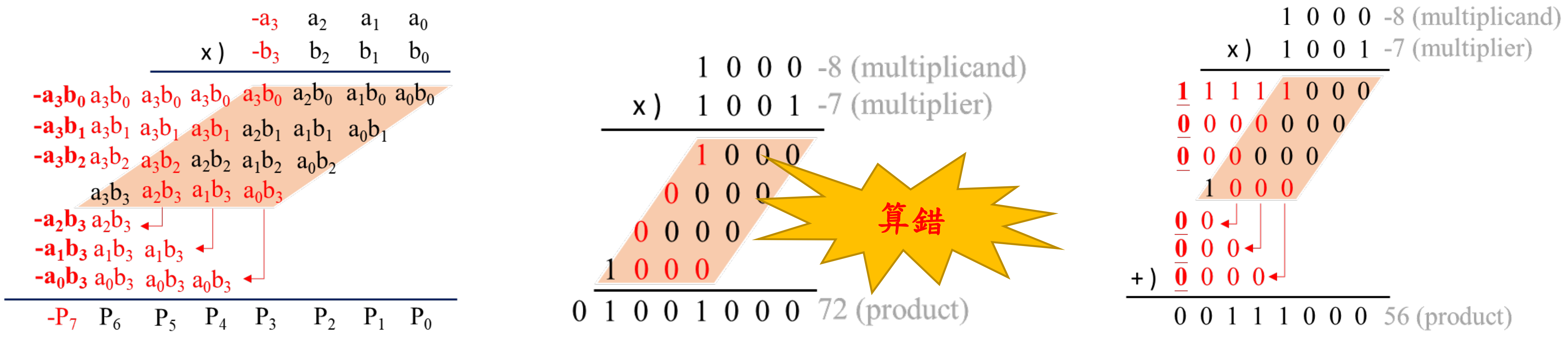
Separate Sign Handling

- 可以透過判斷兩個input是否為負數，若為負數則對其做二補數轉換
  - Ex: *if (a[3] == 1) Multiplier = ~a + 1*

- 若兩個input其中一個為負數，則乘積為負，需要將運算結果做二補數轉換，我們透過對兩個input的sign bit做 XOR 來判斷最後運算結果的正負
  - Ex: *sign = a[3] ^ b[3]*
    if (sign== 1) product = ~product + 1

Sign Extension

- 在4-bit Array multiplier進行有號數乘法會發現答案不對，因為有號數中的MSB (最高位元) 代表著正負號，若其為負 (MSB = 1)，則代表中間計算結果也會是負值，在後續進行加法時若直接相加則會將值的bit數擴展進而影響到答案正確性
- 可透過將乘到MSB的值進行Sign Extension至8bit使值在運算過程表示正確，使得後續加法的結果即為正確的乘積



Baugh-Wooley Algorithm

P = A x B ，A與B皆為n bit 有號數

A = $-a_{n-1}\ a_{n-2}\ \ldots\ldots\ a_1\ a_0$

B = $-b_{n-1}\ b_{n-2}\ \ldots\ldots\ b_1\ b_0$

$$P = \left(-a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i\right) \times \left(-b_{n-1} \cdot 2^{n-1} + \sum_{j=0}^{n-2} b_j \cdot 2^j\right)$$

$$= a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j}$$

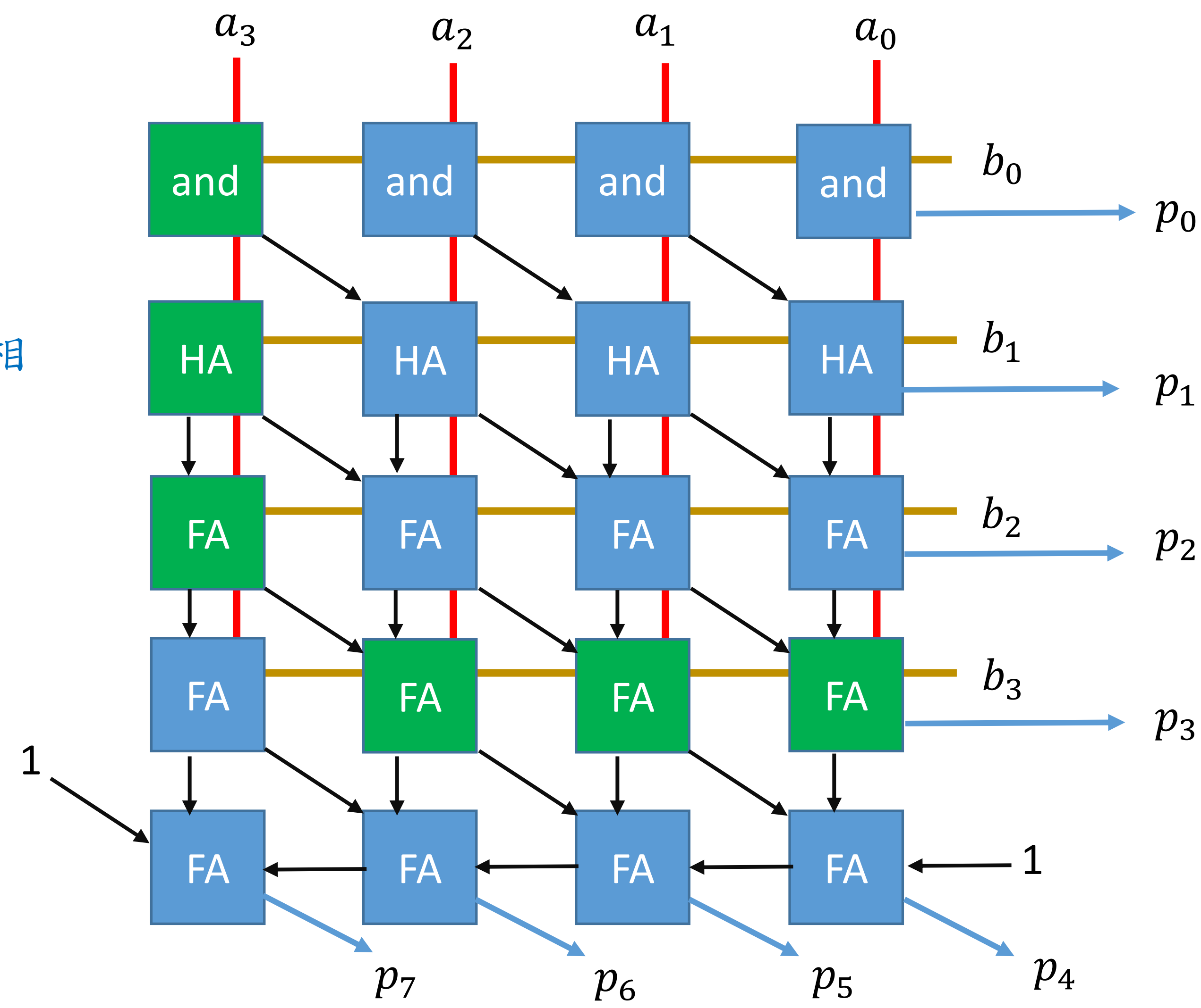
$$-2^{n-1} \cdot \sum_{i=0}^{n-2} a_i \cdot b_{n-1} \cdot 2^i - 2^{n-1} \cdot \sum_{j=0}^{n-2} a_{n-1} \cdot b_j \cdot 2^j$$

$$= a_{n-1} \cdot b_{n-1} \cdot 2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i \cdot b_j \cdot 2^{i+j}$$

$$+2^{n-1} \cdot \sum_{i=0}^{n-2} \overline{a_i \cdot b_{n-1}} \cdot 2^i + 2^{n-1} \cdot \sum_{j=0}^{n-2} \overline{a_{n-1} \cdot b_j} \cdot 2^j - 2^{2n-1} + 2^n$$

Baugh-Wooley Algorithm

綠框為$a_i b_j$取反相



Baugh-Wooley Algorithm

經過推導可以得知，要將無號數乘法改為有號數乘法只要將所有與sign bit做and的乘積反相以及在p的 $(2n-1)$ 列與p的 $(n+1)$ 列就可以得到正確的有號數乘法結果，如下圖

$$
\begin{array}{ccccccccc}
 & & & & & -a_3 & a_2 & a_1 & a_0 \\
 & & & & \text{X} & -b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & & \overline{a_3b_0} & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & \overline{a_3b_1} & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & \overline{a_3b_2} & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & & a_3b_3 & \overline{a_2b_3} & \overline{a_1b_3} & \overline{a_0b_3} & & & \\
+ & 1 & & & 1 & & & & \\
\hline
 & -p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

e.g.

$$
\begin{array}{cccccccccl}
 & & & & & 1 & 0 & 0 & 0 & \text{-8 (multiplicand)} \\
 & & & & \text{x} & 1 & 0 & 0 & 1 & \text{-7 (multiplier)} \\
\hline
 & & & & & 0 & 0 & 0 & 0 & \\
 & & & & 1 & 0 & 0 & 0 & & \\
 & & & 1 & 0 & 0 & 0 & & & \\
 & & 1 & 1 & 1 & 1 & & & & \\
+ & 1 & & & 1 & & & & & \\
\hline
 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \text{56 (product)}
\end{array}
$$

Booth Recoding

現有一乘數B與被乘數A，在乘數右方加一bit 0 後判斷$b_0 b_{0-1}$，依照其結果做三種運算，後判斷$b_1 b_0$，以此類推，直到b_i為MSB為止，三種運算分別為+A、-A、+0

b_i	b_{i-1}	operation
0	0	0
0	1	+A
1	0	-A
1	1	0

e.g.

1 0 0 0	-8 (multiplicand)
x 1 0 0 1 0	-7 (multiplier)
0 0 0 0 1 0 0 0	+8 (-A)
1 1 1 1 0 0 0	-8 (+A)
0 0 0 0 0 0	0 (0)
+ 0 1 0 0 0	+8 (-A)
0 0 1 1 1 0 0 0	56 (product)

Radix-4 Booth Recoding

現有一乘數B與被乘數A，在乘數右方加一bit 0 後判斷$b_1b_0b_{0-1}$，依照其結果做五種運算，後判斷$b_3b_2b_1$，以此類推，直到b_{i+1}為MSB為止，五種運算分別為+A、+2A、-A、-2A、+0

b_{i+1}	b_i	b_{i-1}	operation
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

e.g.

	1 0 0 0	-8 (multiplicand)
x	1 0 0 1 0	-7 (multiplier)
	1 1 1 1 1 0 0 0	-8 (010: A)
+	0 1 0 0 0 0	+16 (100: -2A)
	0 0 1 1 1 0 0 0	56 (product)

教材說明

教材內容

- 4bit Array Multiplier與以及對應的testbench
- 4bit Sign Extension Multiplier及對應的testbench
- 4bit booth Multiplier及對應的testbench

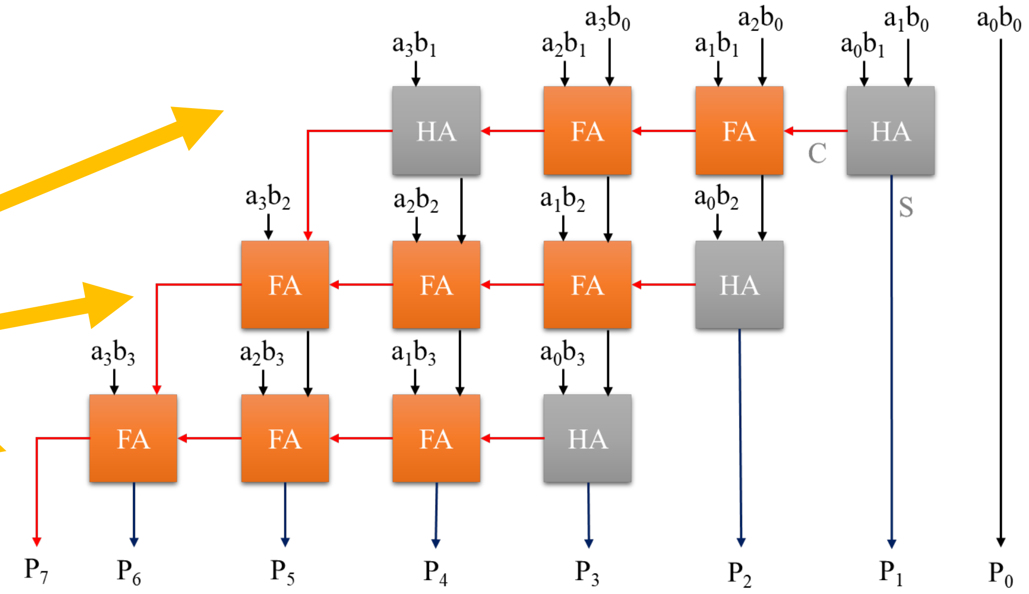
教材說明 - 4bit Array Multiplier

```
module MPY(clk, a, b, p);
    input clk;
    input [3:0] a, b;
    output [7:0] p;
    wire [3:0] ab0, ab1, ab2, ab3;
    wire [2:0] carry;
    wire [3:0] s0, s1, s2;

    arrand and0(a, b[0], ab0);
    arrand and1(a, b[1], ab1);
    arrand and2(a, b[2], ab2);
    arrand and3(a, b[3], ab3);

    HA2FA2 HA2FA2_u1(clk, ab1, ab0[3:1], carry[0], s0);
    HA1FA3 HA1FA3_u1(clk, ab2, {carry[0],s0[3:1]}, carry[1], s1);
    HA1FA3 HA1FA3_u2(clk, ab3, {carry[1],s1[3:1]}, carry[2], s2);

    assign p[0] = ab0[0];
    assign p[1] = s0[0];
    assign p[2] = s1[0];
    assign p[6:3] = s2;
    assign p[7] = carry[2];
endmodule
```



依照架構接線

教材說明 - Sign Extension

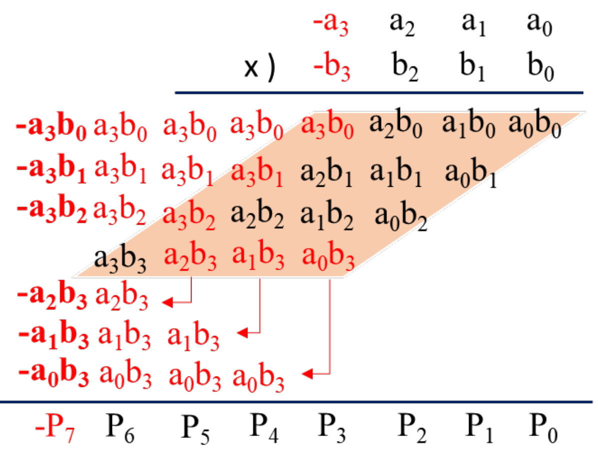
```
module MPY(clk, a, b, product);
    input clk;
    input [3:0] a, b;
    wire [3:0] ab0, ab1, ab2, ab3;
    wire [7:0] add0, add1, add2, add3;
    wire [7:0] ext0, ext1, ext2, ext3;
    wire [7:0] sum0, sum1, sum2, sum3, sum4;
    output [7:0] product;

    arrand and0(a, b[0], ab0);
    arrand and1(a, b[1], ab1);
    arrand and2(a, b[2], ab2);
    arrand and3(a, b[3], ab3);

    assign add0 = {{4{ab0[3]}}, ab0};
    assign add1 = {{3{ab1[3]}}, ab1, 1'b0};
    assign add2 = {{2{ab2[3]}}, ab2, 2'b0};
    assign add3 = {1'b0, ab3, 3'b0};
    assign ext0 = {{4{ab3[0]}}, 4'b0};
    assign ext1 = {{3{ab3[1]}}, 5'b0};
    assign ext2 = {{2{ab3[2]}}, 6'b0};

    adder adder1(clk,add0,add1,sum0);
    adder adder2(clk,sum0,add2,sum1);
    adder adder3(clk,sum1,add3,sum2);
    adder adder4(clk,sum2,ext0,sum3);
    adder adder5(clk,sum3,ext1,sum4);
    adder adder6(clk,sum4,ext2,product);

endmodule
```



將乘到MSB的值進行Sign Extension使值在運算過程正確

教材說明 - Booth Recoding (1 / 2)

```
module booth_add(a, b, ab);
    input [3:0] a;
    input [1:0] b;
    wire signed [4:0] a_ext;
    output [4:0] ab;

    assign a_ext = {a[3],a};
    assign ab = (b==2'b01) ? a_ext:
                (b==2'b10) ? -a_ext:
                             5'b0;
endmodule
```

判斷$b_i b_{i-1}$決定運算

b_i	b_{i-1}	operation
0	0	0
0	1	+A
1	0	-A
1	1	0

教材說明 - Booth Recoding (2 / 2)

```
booth_add booth1(a, {b[0],1'b0}, add0);
booth_add booth2(a, b[1:0], add1);
booth_add booth3(a, b[2:1], add2);
booth_add booth4(a, b[3:2], add3);

assign add0_ext = {{3{add0[4]}},add0};
assign add1_ext = {{2{add1[4]}},add1};
assign add2_ext = {add2[4],add2};

HA1FA6 HA1FA6_u1(clk, add0_ext[7:1], add1_ext, s0);
HA1FA5 HA1FA5_u1(clk, s0[6:1], add2_ext, s1);
HA1FA4 HA1FA4_u1(clk, s1[5:1], add3, s2);

assign p[0] = add0_ext[0];
assign p[1] = s0[0];
assign p[2] = s1[0];
assign p[7:3] = s2;
endmodule
```

判斷

位移

Array MPY

算出結果

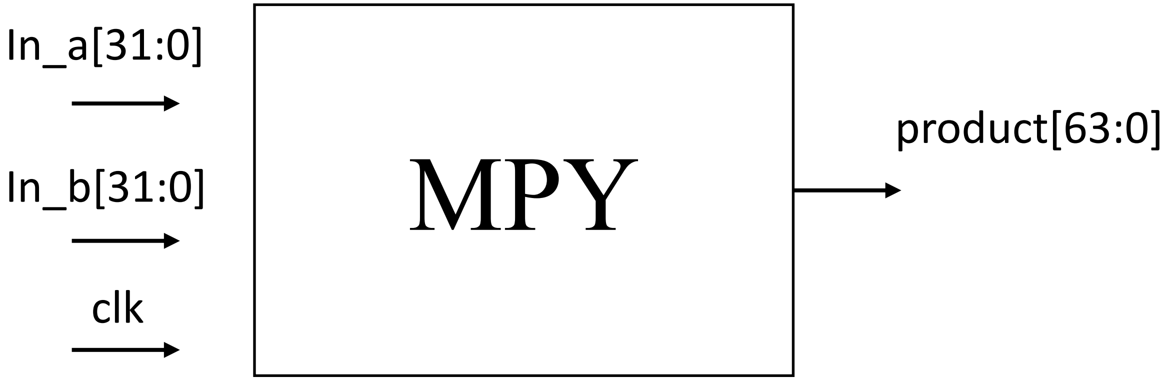
判斷$b_i b_{i-1}$直到b_i為MSB

e.g.

	1 0 0 0	-8 (multiplicand)
x	1 0 0 1 0	-7 (multiplier)
	0 0 0 0 1 0 0 0	+8 (-A)
	1 1 1 1 0 0 0	-8 (+A)
	0 0 0 0 0 0	0 (0)
+	0 1 0 0 0	+8 (-A)
	0 0 1 1 1 0 0 0	56 (product)

LAB作業

- 透過上述方法之一，實作一32bit有號數乘法器。



設計的乘法器皆有2個input及1個output，請同學自行設計testbench驗證結果是否正確。

課程評分

透過上述方法之一，實作一32bit有號數乘法器。
Demo時將使用助教提供的Testbench進行運算，共10筆測資，每筆10%。
並會於Demo時詢問選用架構及原因，嚴禁抄襲。

使用Verilog 語法直接進行相乘視同作弊，此次LAB成績以0分計算。

記得填寫意見回饋表，否則不予以計分