Lecture 6 Arithmetic

SIGNED AND UNSIGNED ADDITION AND MULTIPLICATION

Negative Integers

- Thus far we have only addressed positive integers. What about negative numbers?

- 3 ways to represent negative integers in binary:
  - Sign Magnitude
  - One's complement
  - Two's complement

- Example: Consider +23 –23
  - Sign Magnitude 0 10111 1 10111
  - One's complement 0 10111 1 01000
  - Two's complement 0 10111 1 01001

Sign bit

Why do we all use 2's complement?

- 2's complement makes subtraction easy
- Represent negative number, –x as $2^n - x$
- All arithmetic is done modulo 2^n so no adjustments are necessary
  - $x + (-y) = x + (2^n - y) \pmod{2^n}$
  - consider 4-bit numbers

$$v = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

$$\begin{aligned} 4 - 3 &= 4 + (16 - 3) \pmod{16} \\ &= 4 + (15 - 3 + 1) \\ &= 0100 + (1111 - 0011) + 0001 \\ &= 0100 + 1100 + 0001 \\ &= 0001 \end{aligned}$$

Signed data types (after Verilog-2001)

- Type casting
  - A = $signed(B) //sign extension of sign bit
  - A = $unsigned(B) //zero fill B and assign it to A
- Signed Based Values
  - 3'sh2 //3-bit signed hex value
  - -3'sh4 //3-bit signed hex value
  - A decimal number is always signed

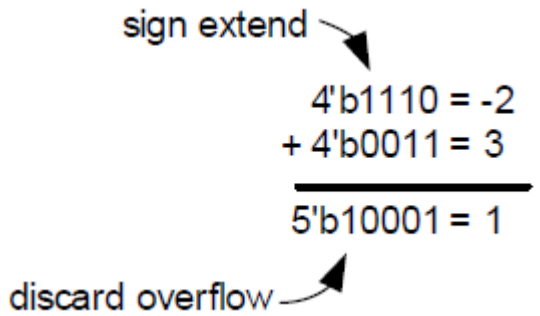
```
logic signed [5:0] a;
logic [2:0] b;
b = 3'b101;
a = $signed(101);  //a = 111_101;
a= $unsigned(101); //a = 000_101;
```

Decimal Value	Signed Representation
3	3'b011
2	3'b010
1	3'b001
0	3'b000
-1	3'b111
-2	3'b110
-3	3'b101
-4	3'b100

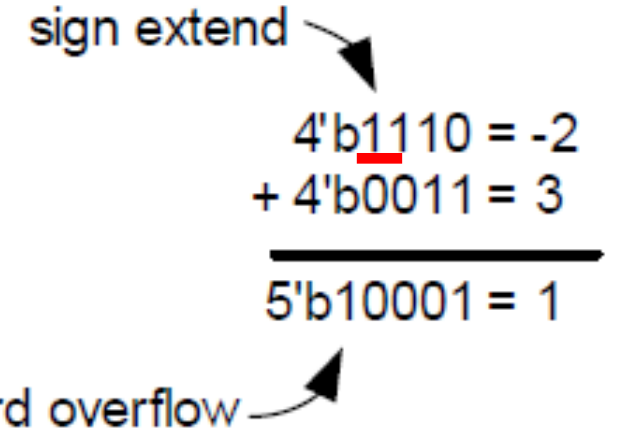
2's complement

無號數及有號數的乘加運算電路 (DIY Sign Extension in Verilog)

- 設計一個電路計算a * b + c，
  - mode=0時，採用unsigned operation，
  - mode=1時，採用signed operation



Sign Extension in Verilog for Addition




```
module add_signed_1995 (
       input [2:0] A,
       input [2:0] B,
       output [3:0] Sum
       );
       assign Sum = {A[2],A} + {B[2],B};
endmodule // add_signed_1995
```

Mixed signed/unsigned operation is regarded as unsigned
use `$signed` for type casting

You have to deal with sign extension by yourself in Verilog 1995
Use signed for auto handling

```
module add_signed_2001 (
       input signed [2:0] A,
       input signed [2:0] B,
       output signed [3:0] Sum
       );
       assign Sum = A + B;
endmodule // add_signed_2001
```

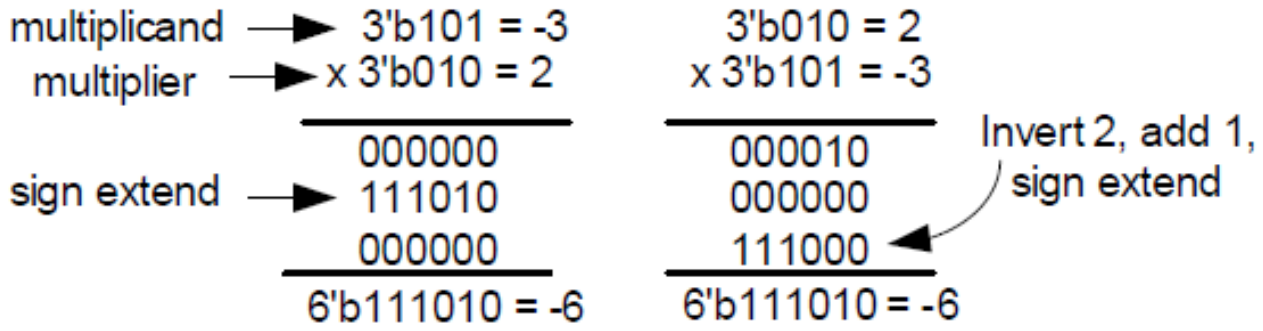
```
module add_carry_signed_final (
       input signed [2:0] A,
       input signed [2:0] B,
       input carry_in,
       output signed [3:0] Sum
       );
assign Sum = A + B + $signed({1'b0,carry_in});
endmodule // add_carry_signed_2001
```

carry_in is unsigned, add "0" before type casting

http://www.tumbush.com/published_papers/Tumbush%20DVCon%2005.pdf

Sign Extension in Verilog for Multiplication

```
module mult_signed_1995 (
input [2:0] a,
input [2:0] b,
output [5:0] prod
);
  wire [5:0] prod_intermediate0;
  wire [5:0] prod_intermediate1;
  wire [5:0] prod_intermediate2;
  wire [2:0] inv_add1;
  assign     prod_intermediate0 = b[0] ? {{3{a[2]}}, a}     : 6'b0;
  assign     prod_intermediate1 = b[1] ? {{2{a[2]}}, a, 1'b0} : 6'b0;
  // Do the invert and add1 of a.
  assign     inv_add1 = ~a + 1'b1;
  assign     prod_intermediate2 = b[2] ? {{1{inv_add1[2]}}, inv_add1, 2'b0} : 6'b0;
  assign     prod = prod_intermediate0 + prod_intermediate1 + prod_intermediate2;
endmodule
```



MSB is sign bit, appy 2's complement

Sign Extension in Verilog for Multiplication

```
module mult_signed_2001 (
       input signed [2:0] a,
       input signed [2:0] b,
       output signed [5:0] prod
       );
       assign prod = a*b;
endmodule
```

```
module mult_signed_unsigned_2001 (
       input signed [2:0] a,
       input [2:0] b,
       output signed [5:0] prod
       );
       assign prod = a*$signed({1'b0,b});
endmodule
```

Careful on mixed sign and unsigned operations

b is unsigned, add “0” before type casting so as a positive number

Summary: Sign Extension in Verilog

- Math operations are context-determined
  - Signed arithmetic is done if all operands are signed
  - ***But,…*** *If any operand in the context is unsigned, then unsigned arithmetic is done*

```
logic signed [3:0] a, b;
logic              ci;
logic signed [4:0] sum;
sum = a + b + ci;
```

Gotcha!

Unsigned adder, even though `a`, `b` and `sum` are signed
- Size context is largest vector on left and right sides
  - `a`, `b` and `ci` are extended to 5 bits before being added
- Sign context is only the operands on right side
  - `ci` is unsigned, so the context is unsigned
  - `a` and `b` are converted to unsigned and zero extended

```
logic signed [3:0] a, b;
logic signed       ci;
logic signed [4:0] sum;
sum = a + b + ci;
```

Gotcha!

Signed adder that **subtracts** carry in
- `a`, `b` and `ci` are signed, so the context is signed
- if `ci` is `1`, sign extending to 5 bits gives `11111` (binary)
- As a signed value, `11111` (binary) is `-1`!

- ***To avoid this Gotcha…***
  - Engineers must know Verilog's context-operation rules!

```
sum = a + b + signed'({1'b0,ci});
```

http://www.sutherland-hdl.com/papers/2006-SNUG-Boston_standard_gotchas_presentation.pdf

實際乘法怎麼做

- 簡單結論
  - assign a = b * c; //注意所需輸出a 精確度 m-bit * n-bit = (m+n) bit
- 那不同加法架構做法呢?
  - 交給synthesis tool根據面積速度功耗限制 自動決定
- 那我可以知道tool到底幫我選了何種架構嗎?
  - Design compiler: report_resources

Bitwidth Problem

- Rule
  - During the evaluation of an expression, interim results shall take the size of the largest operand (in case of an assignment, this also includes the left-hand side)

```
reg [15:0] a, b; // 16-bit regs
reg [15:0] sumA; // 16-bit reg
reg [16:0] sumB; // 17-bit reg

sumA = a + b; // expression evaluates using 16 bits
sumB = a + b; // expression evaluates using 17 bits
```

```
logic [15:0] a, b, answer; // 16-bit regs
assign answer = (a + b) >> 1; //will not work properly
//中間值計算只用16bit, carry bit 會在 做 >>1 前 就被忽略
assign answer = (a + b + 0) >> 1; //will work correctly
//better way
logic [16:0] answer_w;
assign answer_w =  (a + b) >> 1; //wider bitwidth declaration
////////////////////////////////////////////////
//assume only needs top 4-bits results
logic [3:0] c, d, answer_msb;
logic [7:0] tmp_w;
assign answer_msb = a * b; //only the lower 4bits remained, error
assign tmp_w = a * b;
assign answer_msb = tmp_w[7:4]; //correct
```

Overflow

- Normal Condition (沒有Overflow)
  - (+6) + (-3) = (+3)

```
  0110
   101
------
  1001
```

錯

故意使用4 bit的+6與3 bit的-3相加，若直接將兩個signed值相加，答案為-7

```
  00110
  11101
--------
 100011
```

對

因為4 bit與3 bit相加，結果可能進位到5 bit，正確的作法是將4 bit的+6做signed extension到5 bit，且3 bit的-3也要做signed extension到5 bit後，然後才相加，若最後進位到6 bit，則不考慮6 bit的值·

- Boundary Condition (正Overflow)
  - (+7) + (+3) = (+10)

```
  0111
   011
------
  1010
```

錯

```
  00111
  00011
-------
  01010
```

對

故意使用4 bit的+7與3 bit的+3相加，若直接將兩個signed值相加，答案為-6

+7與+3必須做signed extension才能相加，這樣才能得到正確答案+10·

+10必須動到5 bit才能顯示，若輸出的值域為4 bit，只能-8 ~ +7，+10很顯然已經正overflow了

若只能以4 bit表示，因為是正的，MSB必須是0(SUM[3]=0)，所以若MSB是1就表示由進位而來，也就是正overflow了(此例的SUM[3]為1，所以已經正overflow)，再加上因為目前運算結果為5 bit，且是正，所以SUM[5]必須為0。

也就是說，若SUM[5]=0且SUM[4]=1時，為正overflow，所以01010對於4 bit來說，是正overflow。

- Boundary Condition (負Overflow)
  - (-5) + (-4) = (-9)

```
  1011
   100
 ------
  1111
```

錯

```
  11011
  11100
 -------
  110111
```

對

故意使用4 bit的-5與3 bit的-4相加，若直接將兩個signed值相加，答案為-1

-5與-4一樣必須做signed extension才能相加，這樣才能得到正確答案-9·進位到6 bit的1要捨去

-9必須動到5 bit才能顯示，若輸出的值域是4 bit，只能-8 ~ +7，-9很顯然已經是負overflow了·
若只能以4 bit表示，因為是負的，MSB必須是1(SUM[3]=1)，所以若MSB是0就表示由進位而來，也就是負overflow了(此例的SUM[3]為0，所以已經負overflow)，再加上因為目前運算結果為5 bit，且是負，所以SUM[5]必須為1·

也就是說，若SUM[5]為1且SUM[4]為0時，為負overflow，所以10111對於4 bit來說，是負overflow

- m bit + m bit => (m+1) bit
- m bit + n bit => (m+1) bit，其中n < m
  - m bit與n bit都必須先做signed extension到(m+1) bit才能相加
  - 若結果有到(m+2) bit則忽略之，實際的結果為(m+1) bit
- 若Sum[m+1] ^ Sum[m]為1，表示有overflow
  - 若Sum[m+1]為0且Sum[m]為1，則為正overflow
  - 若Sum[m+1]為1且Sum[m]為0，則為負overflow