

Lecture 3 – Boolean Arithmetic

Dr Tianxiang Cui

Outline

- Binary Arithmetic
- Representing Negative Numbers
- Overflow
- Adder

Learning Outcome

- To be able to perform binary arithmetic
- To be able to understand different signed number representations
- To be able to understand overflow and its conditions
- To be able to implement half adder and full adder in HDL

Numbers

- Various symbols have been used over the ages to represent numbers
 - Roman numerals (I,II,III,...)
 - Arabic numbers (1,2,3,...)
- All encode a quantity
- We use the decimal system for counting
- But we also have counting systems using other bases
 - Time, eggs...
- Computers just use a different encoding based on two symbols



Binary Counting

Decimal	Binary	Decimal	Binary	Decimal	Binary
0	0	6	110	12	1100
1	1	7	111	13	1101
2	10	8	1000	14	1110
3	11	9	1001	15	1111
4	100	10	1010	16	10000
5	101	11	1011	17	10001

- The 1 in binary behaves like 9 in decimal
- Result of $1 + 1$ is 0, carrying a 1 to the next digit

Binary to Decimal

- Each binary digit corresponds to a power of 2:

Place	7 th	6 th	5 th	4 th	3 rd	2 nd	1 st	0 th
Weight	2^7 = 128	2^6 = 64	2^5 = 32	2^4 = 16	2^3 = 8	2^2 = 4	2^1 = 2	2^0 = 1

- Where the digit is 1, we add the corresponding weight
- Example: convert $1100\ 1010_2$ into decimal

$$\begin{aligned} 1100\ 1010_2 &= 1 \times 128 + 1 \times 64 + 0 \times 32 + 0 \times 16 \\ &\quad + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 128 + 64 + 8 + 2 = 202_{10} \end{aligned}$$

Decimal to Binary

- Repeatedly divide by 2, until we reach 0
- The **right/left**-most binary digit is the **first/last** remainder
- E.g. $101_{10} = 1100101_2$

101	Remainder
50	1
25	0
12	1
6	0
3	0
1	1
0	1

- Example: convert 163_{10} into binary

Decimal to Binary (look-up table)

- $87 = 64$ ($64 = 2^6$, the biggest 2^n that 87 is divisible by) + 23 (remainder)
- $87 = 64 + 16$ ($16 = 2^4$, the biggest 2^n that 23 is divisible by) + 7 (remainder)
- $87 = 64 + 16 + 4$ ($4 = 2^2$, the biggest 2^n that 7 is divisible by) + 3 (remainder)
- $87 = 64 + 16 + 4 + 2$ ($2 = 2^1$, the biggest 2^n that 3 is divisible by) + 1 (remainder)
- $87 = 64 + 16 + 4 + 2 + 1$ ($1 = 2^0$, the biggest 2^n that 1 is divisible by) + 0 (remainder)
- Stop when remainder = 0

Decimal to Binary (look-up table)

$$87 = 2^6 + 2^4 + 2^2 + 2^1 + 2^0$$

$$87 = 1*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$$

$$87 = (1010111)_2$$

- Usually faster than recursive division by 2
- 2 **low cost** processes [lookup and subtract] **better than** 1 **high cost** process [divide]
- Very important principle in CS

Binary Numbers

- Often written out with leading zeros, up to a certain number of bits usually a multiple of eight (one byte)
- If a computer receives $10011 = 19$, in an 8 bit system, this is what gets stored in a register:

00010011

- In a 32 bit system, this is what gets stored (32 binary digits):

[illegible]

Binary, Octal and Hexadecimal

- **Octal** is a **base 8** system, **Hexadecimal** is a **base 16** system
- Both of these are powers of two – often used to compress binary
- Each octal digit equates to **three** consecutive bits of a binary number
- Each hex digit equates to **four** consecutive bits
- Binary 00111011 | Decimal 59 | Hex 3B | Octal 73

Binary, Octal and Hexadecimal

BINARY	HEXADECIMAL	OCTAL	DECIMAL
0 0 0 0	0	0	0
0 0 0 1	1	1	1
0 0 1 0	2	2	2
0 0 1 1	3	3	3
0 1 0 0	4	4	4
0 1 0 1	5	5	5
0 1 1 0	6	6	6
0 1 1 1	7	7	7
1 0 0 0	8	10	8
1 0 0 1	9	11	9
1 0 1 0	A	12	10
1 0 1 1	B	13	11
1 1 0 0	C	14	12
1 1 0 1	D	15	13
1 1 1 0	E	16	14
1 1 1 1	F	17	15

Binary, Octal and Hexadecimal

- Binary numbers are founded on base 2:

$$(10011)_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

- In general, let the following be a string of digits

$$\mathbf{x} = x_n x_{n-1} \dots x_0$$

- The decimal value of \mathbf{x} in base b , denoted $(\mathbf{x})_b$ is defined as follows:

$$(x_n x_{n-1} \dots x_0) = \sum_{i=0}^n x_i \cdot b^i$$

Binary Addition

- First recap how decimal addition works
 - Add each column together from right
 - If bigger than 9 [biggest decimal digit] , we carry over into the next column
- Binary addition is the same, except we carry if the value is greater than 1 [biggest binary digit]

Decimal Addition

$$\begin{array}{r} + 5783 \\ 2456 \\ \hline \end{array}$$

$$\begin{array}{r} + 5783 \\ 2456 \\ \hline 9 \end{array}$$

$$\begin{array}{r} 1 \\ + 5783 \\ 2456 \\ \hline 39 \end{array}$$

$$\begin{array}{r} 11 \\ + 5783 \\ 2456 \\ \hline 239 \end{array}$$

$$\begin{array}{r} 11 \\ + 5783 \\ 2456 \\ \hline 8239 \end{array}$$

Binary Addition

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \end{array}$$

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline 0 \ 1 \end{array}$$

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline 1 \end{array}$$

$$\begin{array}{r} 1 \\ + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline 0 \ 0 \ 1 \end{array}$$

Binary Addition

$$\begin{array}{r} 11 \\ + 00010101 \\ \hline 0001 \end{array}$$

$$\begin{array}{r} 111 \\ + 00010101 \\ \hline 110001 \end{array}$$

$$\begin{array}{r} 111 \\ + 00010101 \\ \hline 10001 \end{array}$$

$$\begin{array}{r} 111 \\ + 00010101 \\ \hline 1110001 \end{array}$$

Representing Negative Numbers

- So far, unsigned numbers
 - How are negative numbers represented on a computer?
- What we use in decimal notation
 - +/– and 0, 1, 2, . . .
- Such a representation is called **sign and magnitude**
- For binary numbers – define **leftmost** bit to be the **sign**
 - $0 \Rightarrow +$, $1 \Rightarrow -$
 - Rest of bits can be numerical value of number
 - Hence, only seven bits are left in a byte (apart from the sign bit), the magnitude can range from 0000000 (0) to 1111111 (127)
- Problems?

One's Complement

- Alternatively, a system known as **one's complement** can be used to represent negative numbers
- A negative binary number is the bitwise **NOT** applied to it — the "**complement**" of its positive counterpart
- E.g. the ones' complement form of 00101011 (43_{10}) becomes 11010100 (-43_{10})
- Still has two representations of 0: 00000000 (+0) and 11111111 (-0)
- The range of signed numbers using one's complement is represented by $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$ and ± 0
 - A conventional eight-bit byte is -127_{10} to $+127_{10}$ with zero being either 00000000 (+0) or 11111111 (-0)

Excess- n

- **Excess- n** , also called offset binary or biased representation, uses a pre-specified number n as a biasing value
- A value is represented by the unsigned number which is n greater than the intended value
- Therefore 0 is represented by n , and $-n$ is represented by the all-zeros bit pattern
- E.g. Excess-3
 - 0 is represented by 0011 (3)
 - +1 is represented by 0100 (4), +2 is represented by 0101(5)...
 - -1 is represented by 0010 (2), -2 is represented by 0001 (1)
 - -3 is represented by 0000 (0)

Two's Complement

- The **two's complement** of an N -bit binary number is defined as the complement with respect to 2^N
 - It is the result of subtracting the number from 2^N
 - $-x$ is represented as $2^N - x$
- There's a quicker way to calculate $2^N - x$:
 - $x + (\text{1's complement of } x) = 2^N - 1$ (all 1 bits)
 - $2^N - x = (\text{1's complement of } x) + 1$
 - Take the bitwise inverse (**NOT**) of x , then add 1 to result
- An N -bit two's-complement numeral system can represent every integer in the range $-(2^{N-1})$ to $+(2^{N-1} - 1)$
 - One's complement: $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$
- The sum of a number and its two's complement will always equal 0 (the last digit is ignored)
 - The sum of a number and its one's complement will always equal -0 (all 1 bits)

Two's Complement

- To get the negative version of a number
 - Invert the bits
 - Add 1
- So, if we want -29
 - 29 = 0001 1101
 - Invert 1110 0010
 - Add 1 1110 0011
- Try -30

Two's Complement: Alternative View

- Assume an 8-bit two's-complement numeral system

8-Bit Two's Complement ($-128 \leq x < 127$)

Bit	MSB							LSB
	7 th	6 th	5 th	4 th	3 rd	2 nd	1 st	0 th
Weight	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- What does 0000 0001 represent?
- What does 1111 1111 represent?
- What does 0101 1011 represent?
- How do we represent -2 in binary?

Example of 4-Bit Signed Encodings

Sign and Mag.		Ones' Comp.		Excess-3		Two's Comp.	
1111	-7	1000	-7	0000	-3	1000	-8
1110	-6	1001	-6	0001	-2	1001	-7
1101	-5	1010	-5	0010	-1	1010	-6
1100	-4	1011	-4	0011	0	1011	-5
1011	-3	1100	-3	0100	+1	1100	-4
1010	-2	1101	-2	0101	+2	1101	-3
1001	-1	1110	-1	0110	+3	1110	-2
1000	-0	1111	-0	0111	+4	1111	-1
0000	+0	0000	+0	1000	+5	0000	0
0001	+1	0001	+1	1001	+6	0001	+1
0010	+2	0010	+2	1010	+7	0010	+2
0011	+3	0011	+3	1011	+8	0011	+3
0100	+4	0100	+4	1100	+9	0100	+4
0101	+5	0101	+5	1101	+10	0101	+5
0110	+6	0110	+6	1110	+11	0110	+6
0111	+7	0111	+7	1111	+12	0111	+7

Signed Extension

- Sign extension is the operation of increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value
 - This is done by appending digits to the **most significant** side of the number, following a procedure dependent on the particular signed number representation used
- For example, if six bits are used to represent the number "00 1010" (decimal +10) and the sign extend operation increases the word length to 16 bits, then the new representation is simply "**0000 0000 0000** 1010" – padding the left side with **0**s
- If ten bits are used to represent the value "11 1111 0001" (decimal -15) using two's complement, and this is sign extended to 16 bits, the new representation is "**1111 1111 1111** 0001 – padding the left side with **1**s

Overflow

- 0111 0110 + 1101 0101

Long Addition in Binary

	0	1	1	1	0	1	1	0	= 118
+	1	1	0	1	0	1	0	1	= 213
	1	1	1	1	0	1	0	0	Carry
	1	0	1	0	0	1	0	1	= 331

Overflow

- One issue in computer arithmetic is dealing with finite amounts of storage, such as 8-bit register
- Overflow occurs when the result of an operation is too large to be stored
- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated
- For a signed number, overflow occurs when
 - Adding **two positives** gives **a negative**
 - Or, adding **two negatives** gives **a positive**
 - Or, subtract **a negative** from **a positive** gives **a negative**
 - Or, subtract **a positive** from **a negative** gives **a positive**

Overflow Conditions

- One way to detect overflow is to check whether the sign bit is consistent with the sign of the inputs when the two inputs are of the same sign – if you added two positive numbers and got a negative number, something is wrong, and vice versa
- Overflow conditions for addition and subtraction are summarized as:

Operation	Operand A	Operand B	Result
A + B	+ve	+ve	-ve
A + B	-ve	-ve	+ve
A - B	+ve	-ve	-ve
A - B	-ve	+ve	+ve
Overflow conditions for addition and subtraction			

Ariane 5

- In 1996, the European Space Agency's Ariane5 rocket was launched for the first time... and it exploded 40 seconds after liftoff
- It turns out the Ariane5 used software designed for the older Ariane4
 - The Ariane4 stored its horizontal velocity as a 16-bit signed integer
 - But the Ariane5 reaches a much higher velocity, which caused an overflow in the 16-bit quantity
- The overflow error was never caught, so incorrect instructions were sent to the rocket boosters and main engine



Binary Multiplication by Base

- Take advantage of the fact that any time you multiply a number by it's base you just add 0 to the end
- Decimal $12 * 10 = 12\mathbf{0}$
- Octal $14 * 10 = 14\mathbf{0}$
- Binary $1100 * 10 = 1100\mathbf{0}$

Shift Operations

- Shift operations shift a word a number of places to the left or right
- Bits which are shifted out, just disappear
- E.g. on 4 bits: 1011 shifted left results in 0110 and shifted right results in 0101
- For unsigned numbers if no bit disappears, shift left corresponds to multiplication by 2
 - E.g. 0011 (3) shifted left is 0110 (6)
- For unsigned numbers, shift right corresponds to division by 2, ignoring the remainder
 - E.g. 0101 (5) shifted right is 0010 (2)

Shift Operations

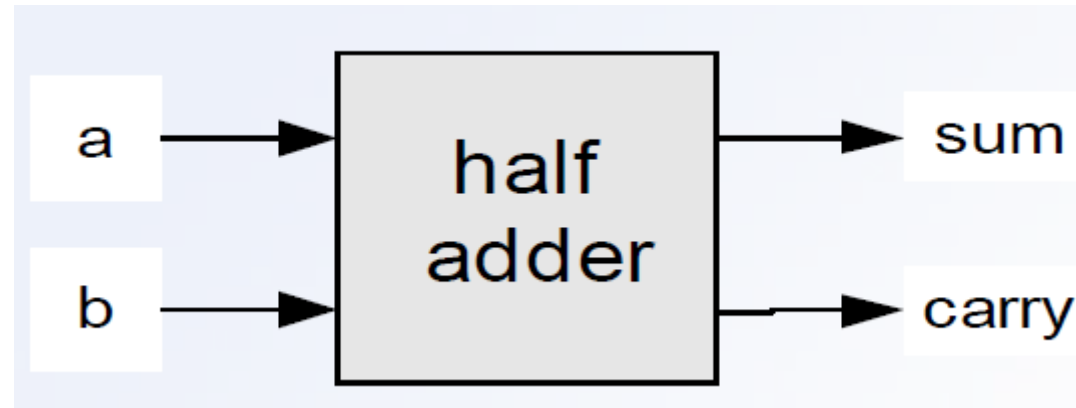
- On two's complement shift left is multiplication by 2, if there is no overflow
 - E.g. 1100 (-4) shifted left results in 1000 (-8)
- Shift right doesn't correspond to division by 2 on negative numbers
 - E.g. 1100 (-4) shifted right results in 0110 (6)
- Arithmetical shift right performs sign extension when shifting
 - E.g. 1100 (-4) shifted right arithmetical results in 1110 (-2)

Adder

- Build an Adder:
 - Half adder: adds two bits
 - Full adder: adds three bits
 - Adder: adds two integers

Half Adder

- Add **two** single binary digits and provide the **output** plus a **carry value**
- It has two inputs, called A(a) and B(b), and two outputs S (sum) and C (carry)



Half Adder

- Least significant bit in the addition is called sum ($a+b$)
- Most significant bit is called carry (carry of $a+b$)

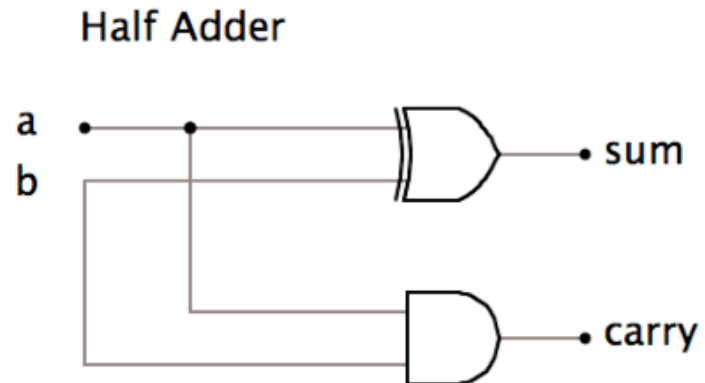
a	b	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- **Never has a situation when sum and carry are both 1**

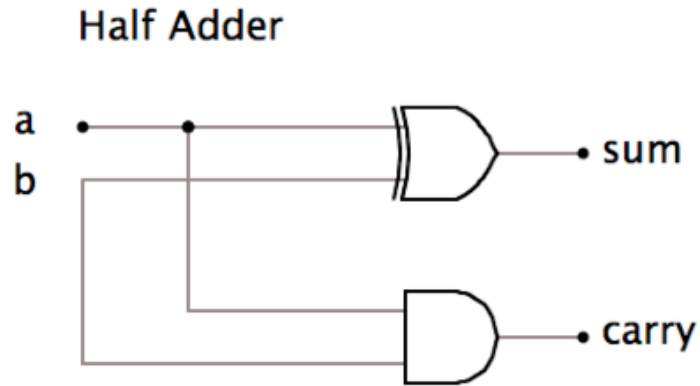
Half Adder

a	b	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- The common representation uses a XOR and a AND gate



Half Adder in HDL



```
CHIP HalfAdder {  
  IN a, b; // 1-bit inputs  
  OUT sum, // Right bit of a + b  
  carry; // Left bit of a + b  
  PARTS:  
    Xor(a=a, b=b, out=sum);  
    And(a=a, b=b, out=carry);  
}
```

Full Adder

- Add **three** single binary digits and provide the **output** plus a **carry value**
- It has three inputs, called A, B and Carry(in), and two outputs S (sum) and Carry(out)



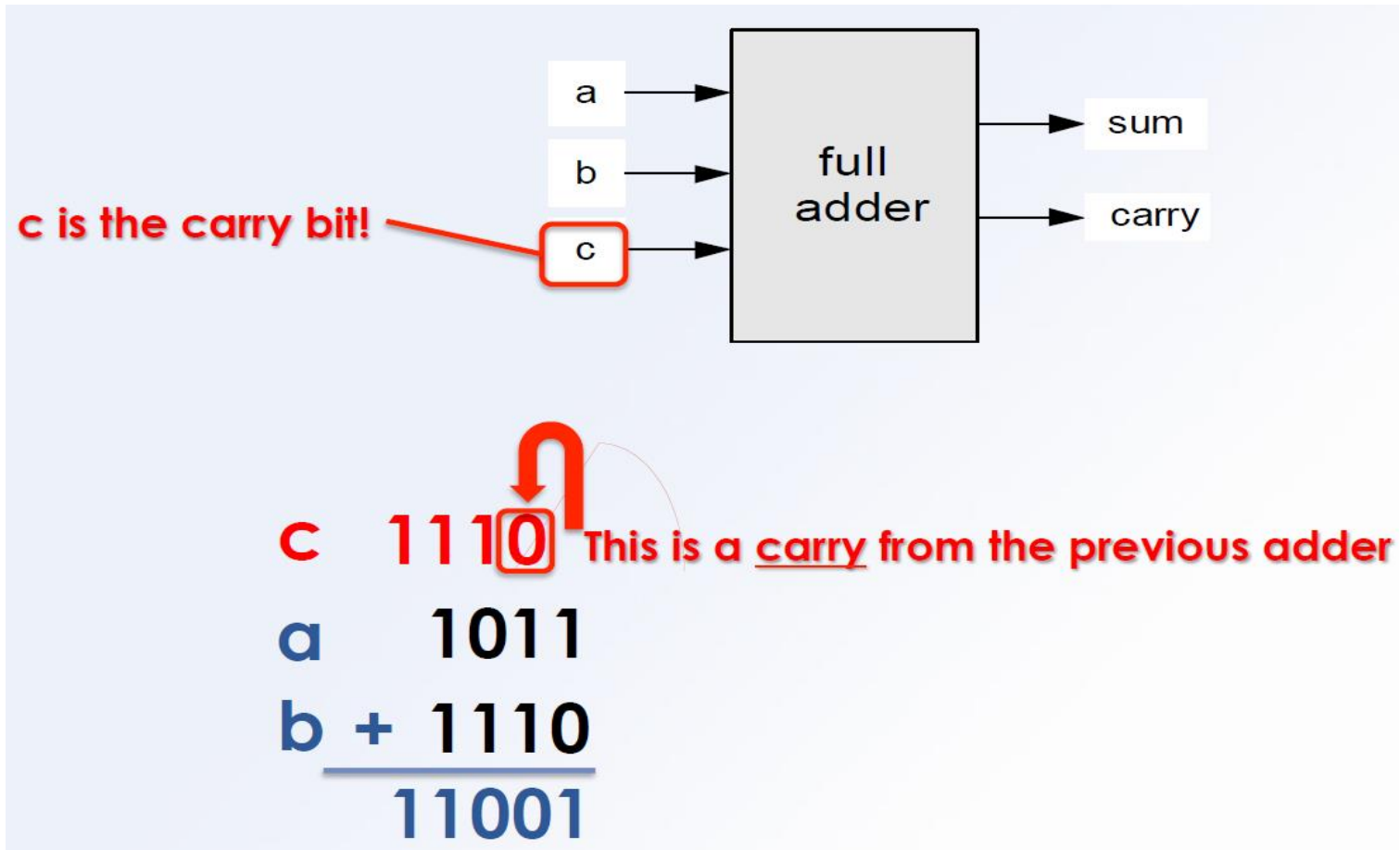
Full Adder

- Least significant bit in the addition is called sum ($a+b+c_{in}$)
- Most significant bit is called carry(out) (carry of $a+b+c_{in}$)

a	b	Carry(in)	Carry(out)	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

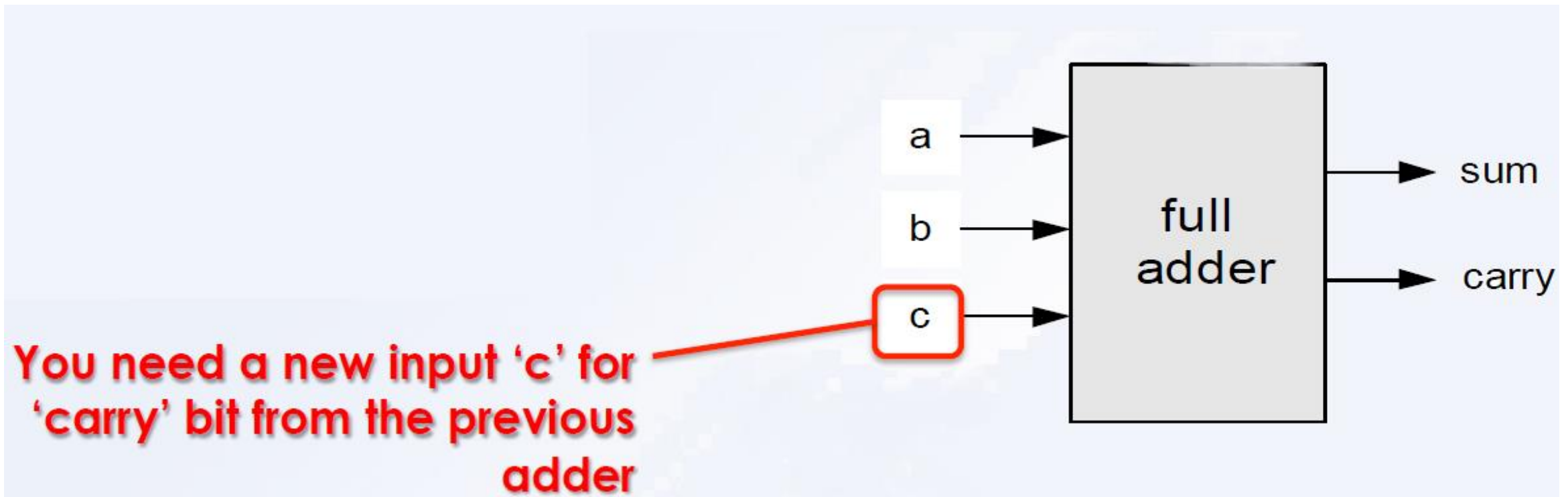
Full Adder

- Computes sum – the least significant bit of $a + b + c$
- Carry the most significant bit of $a + b + c$



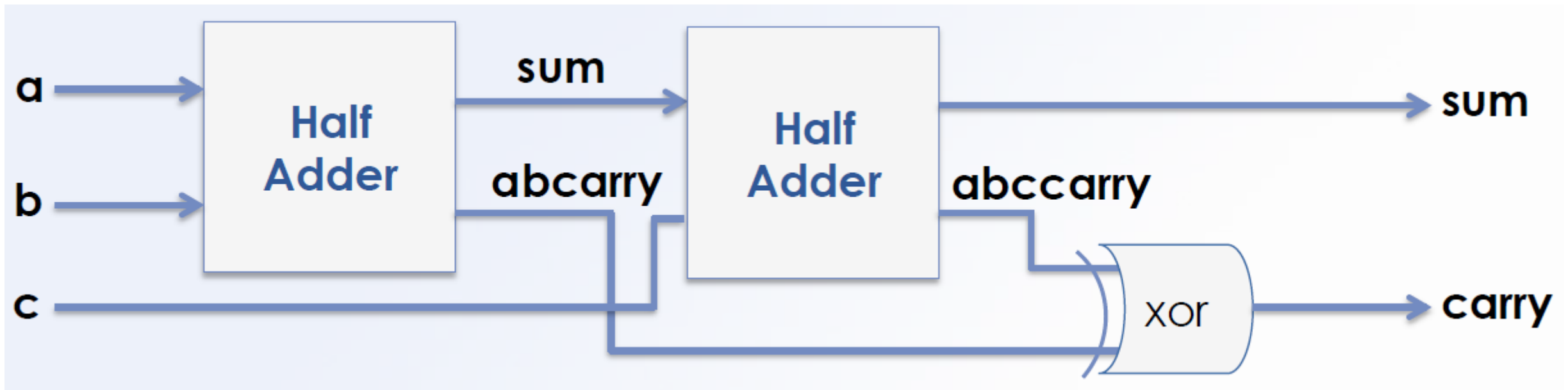
Full Adder

- Carry(in) from the previous adder is needed to bring the carry to the next bit, which makes a full adder



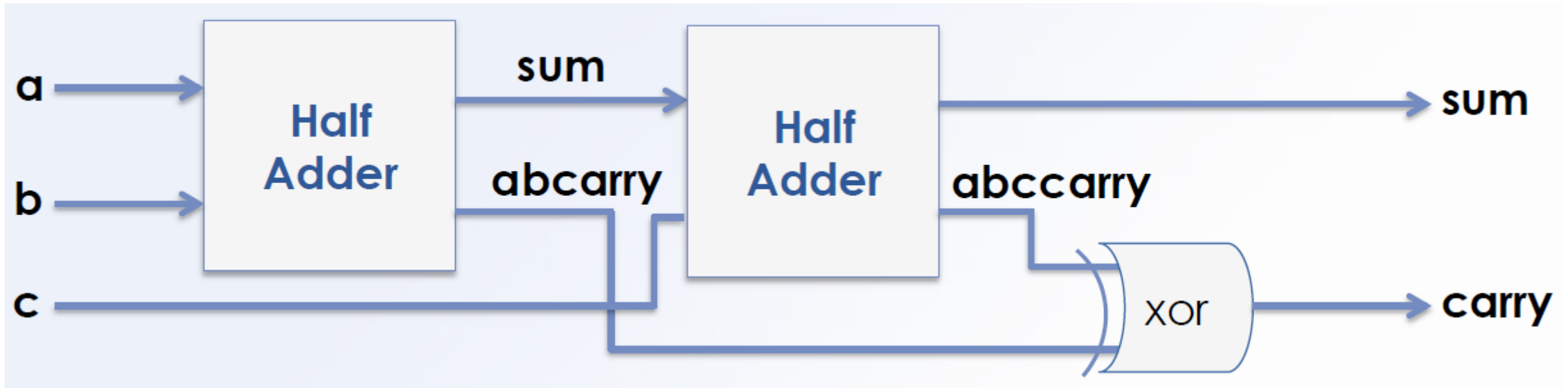
Full Adder: Implementation

- Use two half adders to build a full adder



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Full Adder in HDL



```
CHIP FullAdder {  
  IN a, b, c; // 1-bit inputs  
  OUT sum, // Right bit of a + b + c  
  carry; // Left bit of a + b + c  
  PARTS:  
    HalfAdder(a=a, b=b, sum=absum, carry=abcarry);  
    HalfAdder(a=absum, b=c, sum=sum, carry=abccarry);  
    Xor(a=abcarry, b=abccarry, out=carry);  
}
```

Summary

- Binary Arithmetic
 - Convert binary to decimal, decimal to binary, etc.
- Representing Negative Numbers
 - Sign and magnitude
 - One's complement
 - Excess-n
 - Two's complement
- Overflow
 - Overflow conditions
- Adder
 - Half adder
 - Full adder

Lab 2

Given: Nand

Goal: Build the following gates:

Elementary logic gates

- Not
- And
- Or
- Xor
- Mux
- DMux

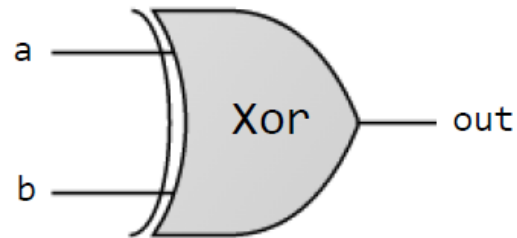
16-bit variants

- Not16
- And16
- Or16
- Mux16

Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

Chip Building Materials



outputs 1 if $a \neq b$

Xor.cmp

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

The contract:

When running your Xor.hdl on the supplied Xor.tst, your Xor.out should be the same as the supplied Xor.cmp

Xor.hdl

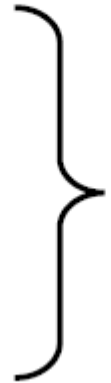
```
CHIP Xor {  
  IN  a, b;  
  OUT out;  
  
  PARTS:  
    // Put your code here.  
}
```

Xor.tst

```
load Xor.hdl,  
output-file Xor.out,  
compare-to Xor.cmp,  
output-list a b out;  
set a 0, set b 0, eval, output;  
set a 0, set b 1, eval, output;  
set a 1, set b 0, eval, output;  
set a 1, set b 1, eval, output;
```

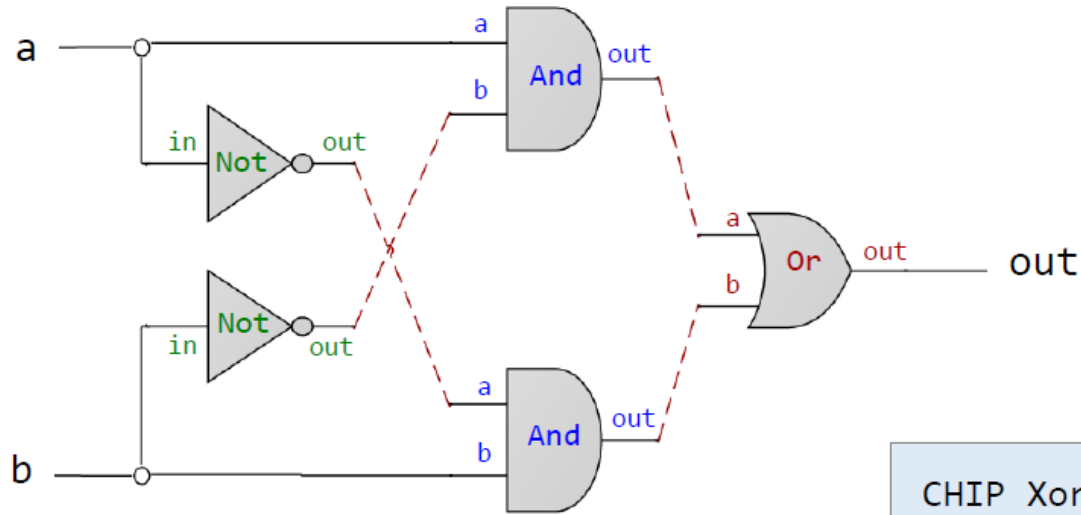
More Resources

- Text editor (for writing your HDL files)
- HDL Survival Guide
- Hardware Simulator Tutorial
- nand2tetris Q&A forum



All available in: www.nand2tetris.org

Hack Chipset API




```
CHIP Xor {  
  IN a, b;  
  OUT out;  
  
  PARTS:  
    Not (in= , out=);  
    Not (in= , out=);  
    And (a= , b= , out=);  
    And (a= , b=b , out=);  
    Or (a= , b= , out=);  
}
```


Hack Chipset API

```
Add16 (a= ,b= ,out= );
ALU (x= ,y= ,zx= ,nx= ,zy= ,ny= ,f= ,no= ,out= ,zr= ,ng= );
And16 (a= ,b= ,out= );
And (a= ,b= ,out= );
Aregister (in= ,load= ,out= );
Bit (in= ,load= ,out= );
CPU (inM= ,instruction= ,reset= ,outM= ,writeM= ,addressM= ,pc= );
DFF (in= ,out= );
DMux4Way (in= ,sel= ,a= ,b= ,c= ,d= );
DMux8Way (in= ,sel= ,a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= );
Dmux (in= ,sel= ,a= ,b= );
Dregister (in= ,load= ,out= );
FullAdder (a= ,b= ,c= ,sum= ,carry= );
HalfAdder (a= ,b= ,sum= ,carry= );
Inc16 (in= ,out= );
Keyboard (out= );
Memory (in= ,load= ,address= ,out= );
Mux16 (a= ,b= ,sel= ,out= );
Mux4Way16 (a= ,b= ,c= ,d= ,sel= ,out= );
Mux8Way16 (a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= );
```

```
Mux8Way (a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= );
Mux (a= ,b= ,sel= ,out= );
Nand (a= ,b= ,out= );
Not16 (in= ,out= );
Not (in= ,out= );
Or16 (a= ,b= ,out= );
Or8Way (in= ,out= );
Or (a= ,b= ,out= );
PC (in= ,load= ,inc= ,reset= ,out= );
PCLoadLogic (cinstr= ,j1= ,j2= ,j3= ,load= ,inc= );
RAM16K (in= ,load= ,address= ,out= );
RAM4K (in= ,load= ,address= ,out= );
RAM512 (in= ,load= ,address= ,out= );
RAM64 (in= ,load= ,address= ,out= );
RAM8 (in= ,load= ,address= ,out= );
Register (in= ,load= ,out= );
ROM32K (address= ,out= );
Screen (in= ,load= ,address= ,out= );
Xor (a= ,b= ,out= );
```

Built-in Chips

```
CHIP Foo {  
  IN ...;  
  OUT ...;  
  
  PARTS:  
  ...  
  Mux16(...)   
  ...  
}
```

Q: What happens if there is no Mux16.hdl file in the current directory?

A: The simulator invokes, and evaluates, the built-in version of Mux16 (if such exists).

- The supplied simulator software features built-in chip implementations of all the chips in the Hack chip set
- If you don't implement some chips from the Hack chipset, you can still use them as chip-parts of other chips:
 - Just rename their given stub files to, say, Mux16.hdl1
 - This will cause the simulator to use the built-in chip implementation.

Best Practice Advice

- Try to implement the chips in the given order
- If you don't implement some chips, you can still use them as chip parts in other chips (the built in implementations will kick in)
- You can invent new, “helper chips”; however, this is not required: you can build any chip using previously built chips only
- Try to use as few chip parts as possible