

CS/EE 320 Computer Organization and Assembly Language Spring 2025

Shahid Masud

Lecture 9

- Logic and Arithmetic Instructions in Assembly Language
- Binary Numbers, Logic Design and Boolean Algebra
- Basic digital logic functions Invert, AND, OR, NAND, NOR
- Half-Adder (without Carry-In)
- 1-bit Full Adder (with Carry-In)
- Making 2's Complement Subtractor from Full-Adder
- Design of 1-bit ALU containing AND gate, OR gate, Full Adder
- Quiz 2 next lecture

MIPS Arithmetic Instructions



Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow detected
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow detected
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow detected
	add unsigned	addu \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three operands; overflow undetected
	subtract unsigned	subu \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three operands; overflow undetected
	add immediate unsigned	addiu \$s1,\$s2,100	$\$s1 = \$s2 + 100$	+ constant; overflow undetected
	move from coprocessor register	mfc0 \$s1,\$epc	$\$s1 = \epc	Copy Exception PC + special regs
	multiply	mult \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit signed product in Hi, Lo
	multiply unsigned	multu \$s2,\$s3	Hi, Lo = $\$s2 \times \$s3$	64-bit unsigned product in Hi, Lo
	divide	div \$s2,\$s3	Lo = $\$s2 / \$s3$, Hi = $\$s2 \bmod \$s3$	Lo = quotient, Hi = remainder
	divide unsigned	divu \$s2,\$s3	Lo = $\$s2 / \$s3$, Hi = $\$s2 \bmod \$s3$	Unsigned quotient and remainder
	move from Hi	mfhi \$s1	$\$s1 = \text{Hi}$	Used to get copy of Hi
	move from Lo	mflo \$s1	$\$s1 = \text{Lo}$	Used to get copy of Lo

MIPS Logical Instructions



Logical	AND	AND	$\$s1, \$s2, \$s3$	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	OR	OR	$\$s1, \$s2, \$s3$	$\$s1 = \$s2 \mid \$s3$	Three reg. operands; bit-by-bit OR
	NOR	NOR	$\$s1, \$s2, \$s3$	$\$s1 = \sim (\$s2 \mid \$s3)$	Three reg. operands; bit-by-bit NOR
	AND immediate	ANDi	$\$s1, \$s2, 100$	$\$s1 = \$s2 \& 100$	Bit-by-bit AND with constant
	OR immediate	ORi	$\$s1, \$s2, 100$	$\$s1 = \$s2 \mid 100$	Bit-by-bit OR with constant
	shift left logical	sll	$\$s1, \$s2, 10$	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl	$\$s1, \$s2, 10$	$\$s1 = \$s2 \gg 10$	Shift right by constant

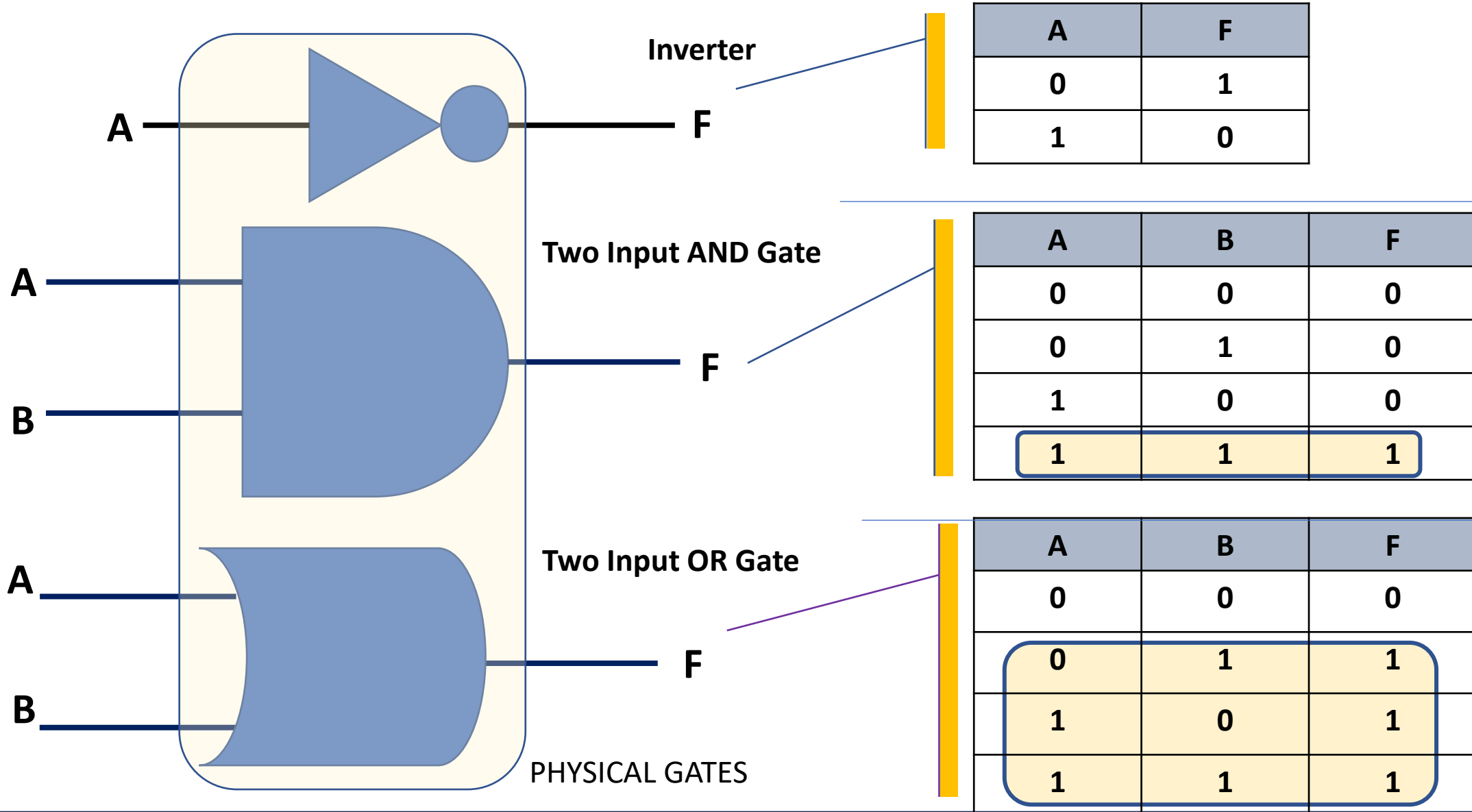
- Operations on integers
 - Addition and subtraction
 - Multiplication and division
 - Dealing with overflow
- Floating-point real numbers (later)
 - Representation and operations

Basic Logic Operations in CPU

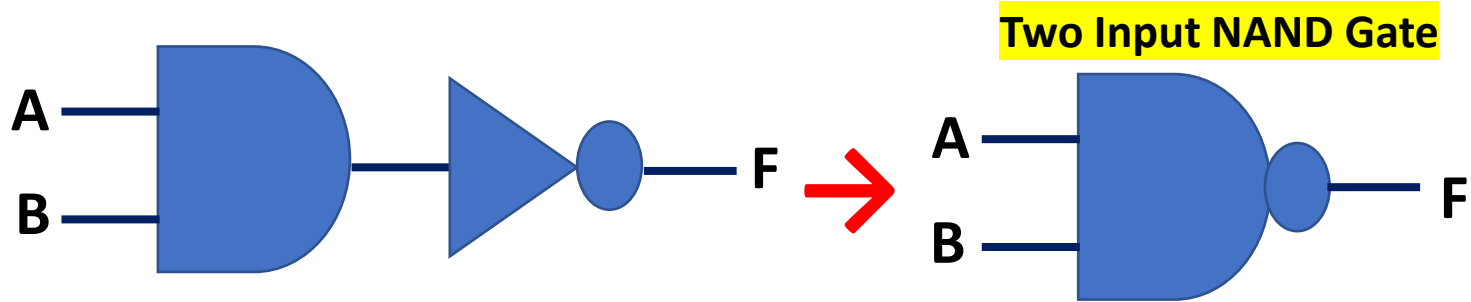


- Functions Through Logic Gates
 - NOT
 - AND
 - OR
 - NOR
 - NAND
 - XOR

Gates (Primary Gates)

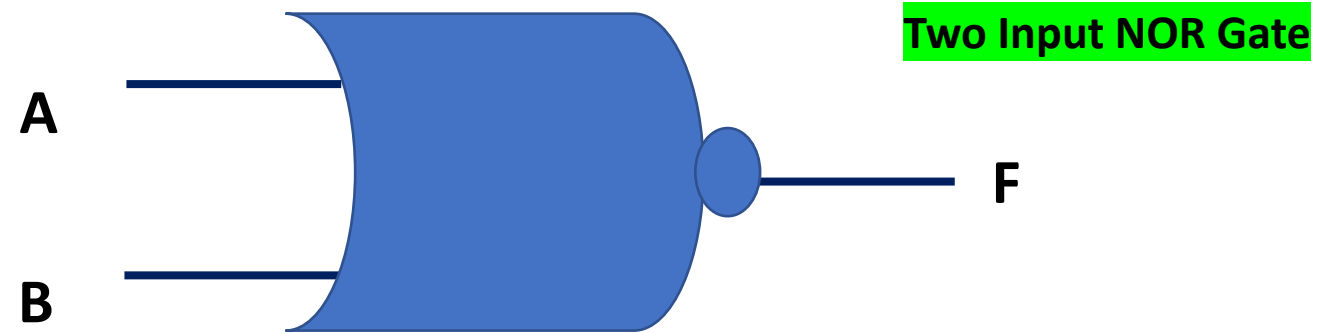


Gates (Compound Gates)



AND Gate followed by Inverter

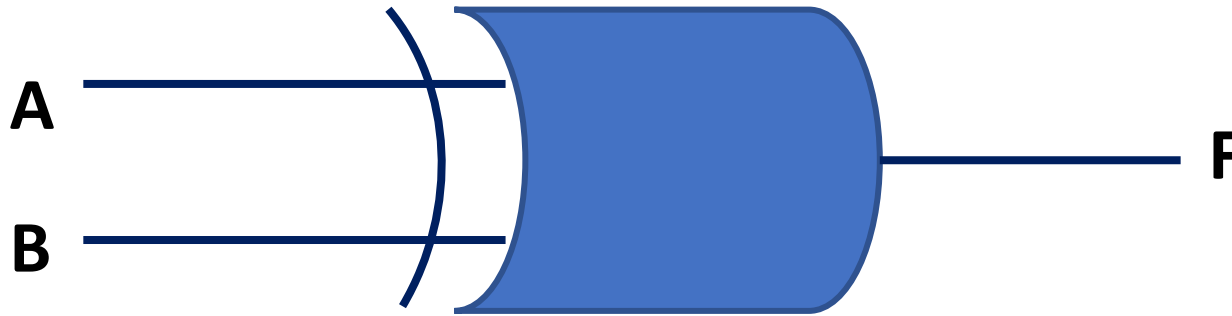
A	B	F
0	0	1
0	1	1
1	0	1
1	1	0



OR Gate followed by Inverter

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

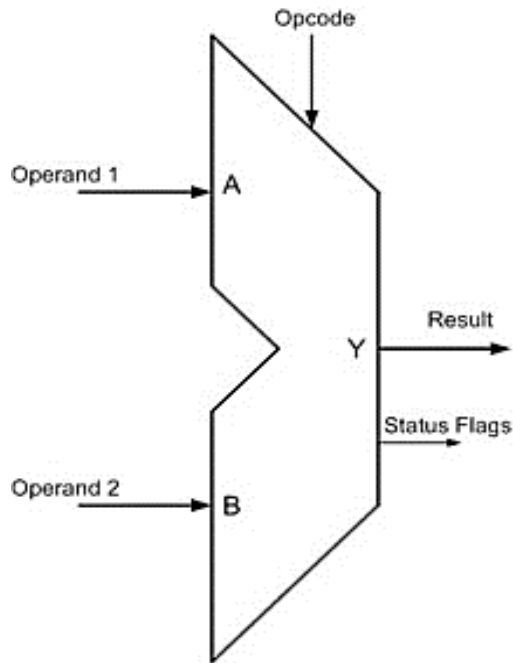
Two Input Exclusive OR Gate



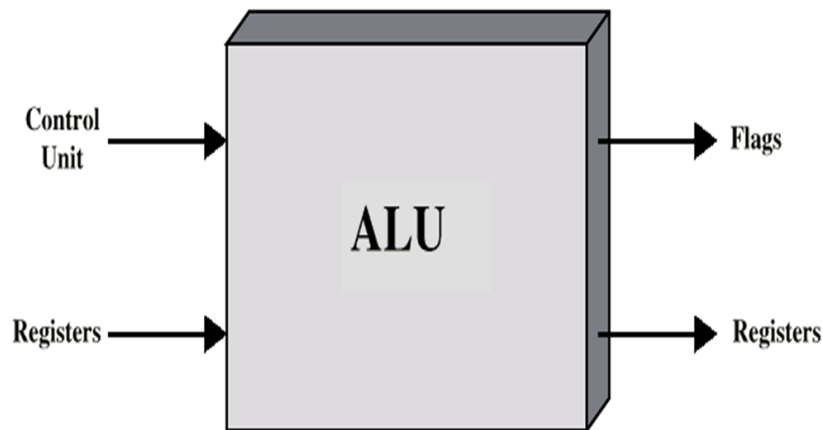
$$F = \bar{A}B + A\bar{B}$$

$$F = A \oplus B$$

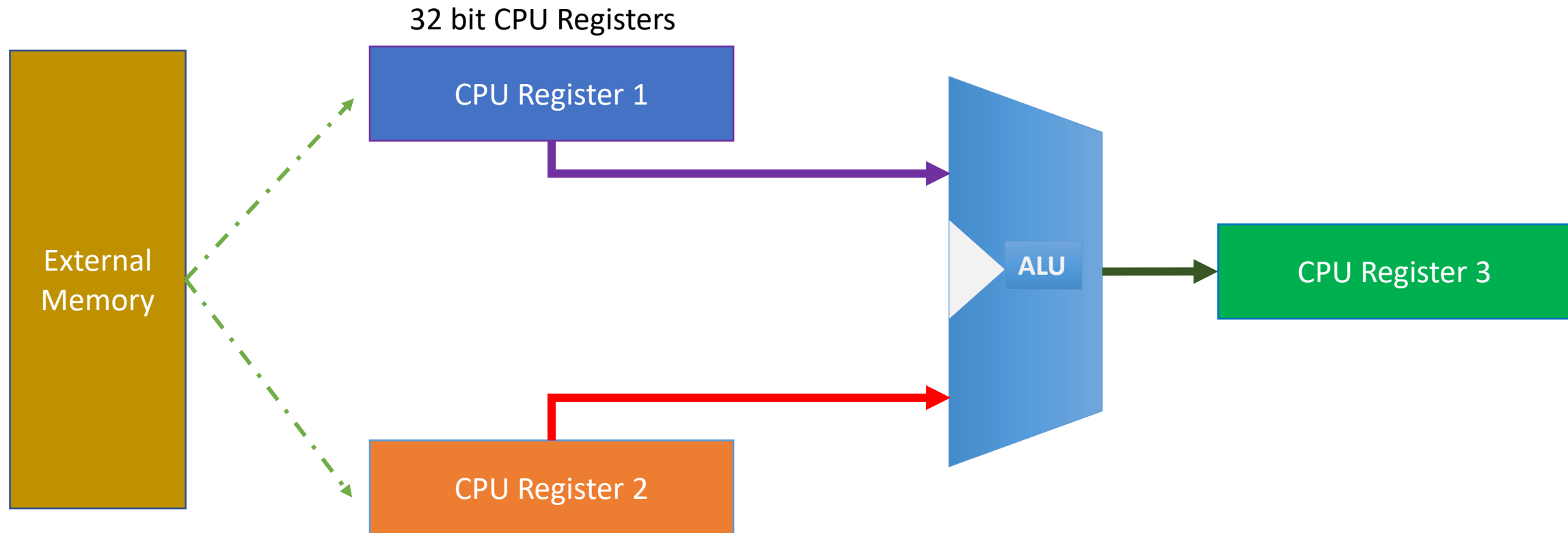
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0



- Does all calculations
- Everything else in the computer is there to service this unit
- Handles integers
- May handle floating point (real) numbers
- May be separate FPU (maths co-processor)
- May be on chip separate FPU (Eg. 486DX)



ALU in **Load/Store** Architecture



- ALU does calculations with binary numbers
- Decimal number system
 - Uses 10 digits (0,1,2,3,4,5,6,7,8,9)
 - In decimal system, a number 84, e.g., means
$$84 = (8 \times 10) + 4$$
 - $4728 = (4 \times 1000) + (7 \times 100) + (2 \times 10) + 8$
 - Base or radix of 10 \rightarrow each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position
 - E.g. $83 = (8 \times 10^1) + (3 \times 10^0)$
 - $4728 = (4 \times 10^3) + (7 \times 10^2) + (2 \times 10^1) + (8 \times 10^0)$

- Uses only two digits, 0 and 1
- It is base or radix of 2
- Each digit has a **value** depending on its **position**:
 - $10_2 = (1 \times 2^1) + (0 \times 2^0) = 2_{10}$
 - $11_2 = (1 \times 2^1) + (1 \times 2^0) = 3_{10}$
 - $100_2 = (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4_{10}$
 - $1001.101_2 = (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) = 9.625_{10}$

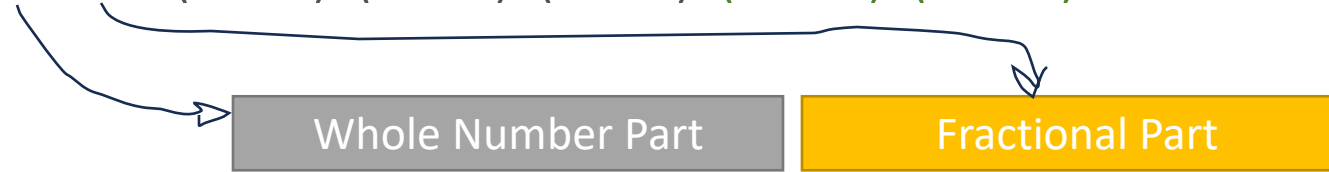
Decimal to Binary conversion



- Integer and fractional parts are handled separately,
 - Integer part is handled by repeating division by 2
 - Fractional part is handled by repeating multiplication by 2
- E.g. convert decimal 11.81 to binary
 - Integer part 11
 - Fractional part .81

- Fractional values, e.g.

- $472.83 = (4 \times 10^2) + (7 \times 10^1) + (2 \times 10^0) + (8 \times 10^{-1}) + (3 \times 10^{-2})$



- In general, for the decimal representation of
- $X = \{ \dots x_2 x_1 x_0 . x_{-1} x_{-2} x_{-3} \dots \}$

$$X = \sum_i x_i 10^i$$

Decimal to Binary conversion example



- e.g. 11.81 to 1011.11001 (approx)
 - $11/2 = 5$ remainder 1
 - $5/2 = 2$ remainder 1
 - $2/2 = 1$ remainder 0
 - $1/2 = 0$ remainder 1
 - Binary number 1011
 - $.81 \times 2 = 1.62$ integral part 1
 - $.62 \times 2 = 1.24$ integral part 1
 - $.24 \times 2 = 0.48$ integral part 0
 - $.48 \times 2 = 0.96$ integral part 0
 - $.96 \times 2 = 1.92$ integral part 1
 - Binary number .11001 (approximate)

- Compact representation of bus information
- Use 16 digits, (0,1,3,...9,A,B,C,D,E,F)
- $1A_{16} = (1_{16} \times 16^1) + (A_{16} \times 16^0)$
 $= (1_{10} \times 16^1) + (10_{10} \times 16^0) = 26_{10}$
- Convert group of four binary digits to/from one hexadecimal digit,
 - 0000=0; 0001=1; 0010=2; 0011=3; 0100=4; 0101=5; 0110=6; 0111=7; 1000=8; 1001=9; 1010=A; 1011=B; 1100=C; 1101=D; 1110=E; 1111=F;
- e.g.
 - 1101 1110 0001. 1110 1101 = DE1.DE

- Only have 0 & 1 to represent everything
- Positive numbers stored in binary
 - e.g. 41=00101001
- No minus sign
- No period
- **How to represent negative number?**
 - Sign-Magnitude
 - Two's compliment

Sign-Magnitude Representation



- Add one extra bit on MSB to represent sign
- Left most bit is sign bit
- 0 means positive
- 1 means negative
- $+18 = 00010010$
- $-18 = 10010010$
- Problems
 - Need to consider both sign and magnitude in arithmetic
 - Two representations of zero (+0 and -0)

Two's Complement Representation



- $+3 = 00000011$
- $+2 = 00000010$
- $+1 = 00000001$
- $+0 = 00000000$
- $-1 = 11111111$
- $-2 = 11111110$
- $-3 = 11111101$

2's Complement Number System



- One representation of zero
- Arithmetic works easily, use same hardware as positive numbers
- Negating is fairly easy (2's complement operation)
 - 3 = 00000011
 - Boolean **complement** gives 11111100
 - Add +1 to LSB 11111101

Range of 2's Complement Numbers



- 8 bit 2's complement
 - $+127 = 01111111 = 2^7 - 1$
 - $-128 = 10000000 = -2^7$
- 16 bit 2's complement
 - $+32767 = 01111111 11111111 = 2^{15} - 1$
 - $-32768 = 10000000 00000000 = -2^{15}$

Different word lengths in 2's Complement



- Positive number pack with leading zeros
- $+18 =$ 00010010 (in 8 bits)
- $+18 =$ 00000000 00010010 (in 16 bits)
- Negative numbers pack with leading ones
- $-18 =$ 10010010 (in 8 bits)
- $-18 =$ 11111111 10010010 (in 16 bits, notice sign bit replication)
- i.e. pack higher bits with MSB (sign bit)

Negation Special Case 1



- 0 = 00000000
- Bitwise not 11111111
- Add 1 to LSB +1
- Result 1 00000000
- **Overflow** is ignored, so:
- - 0 = 0 OK!

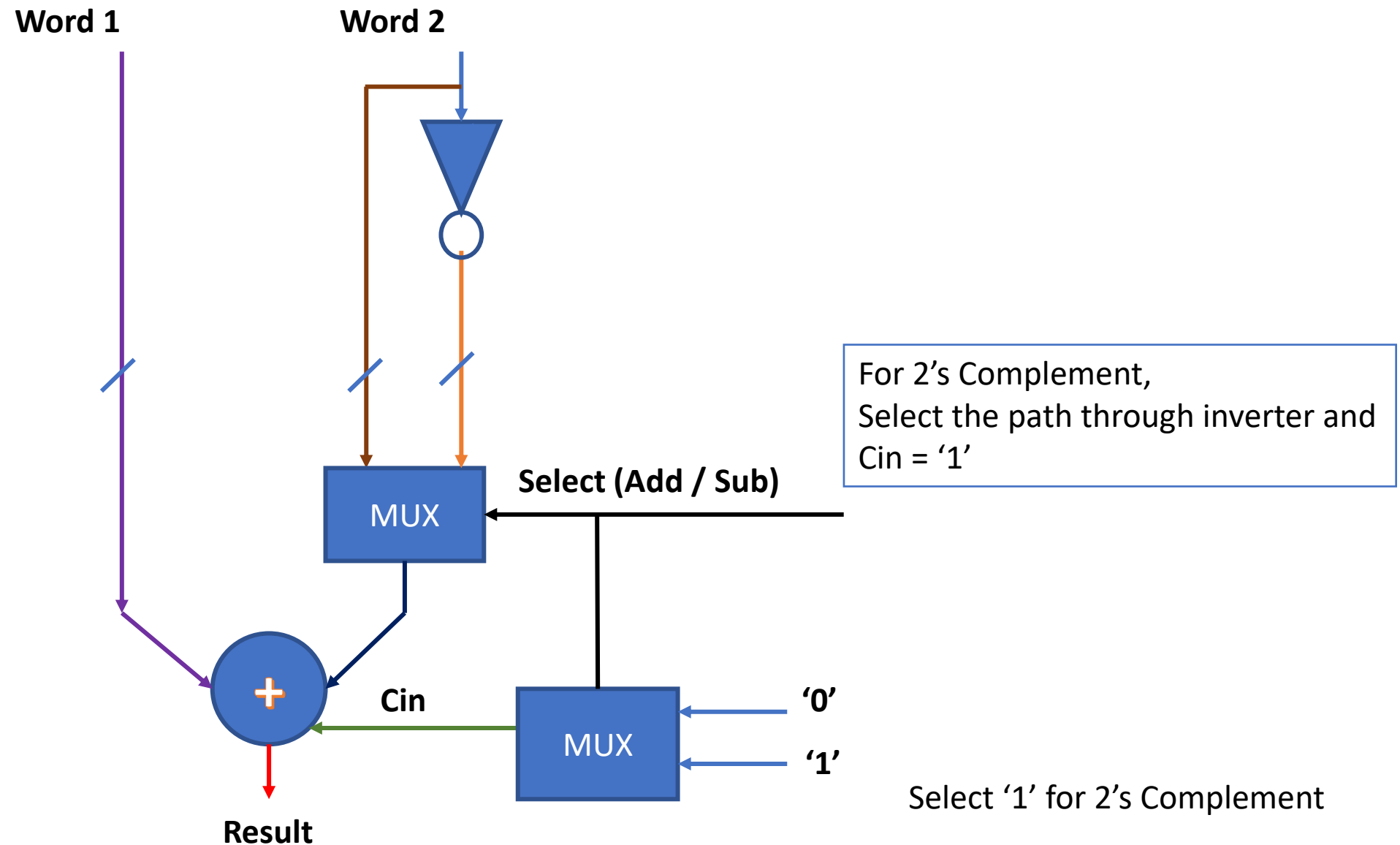
Negation Special Case 2



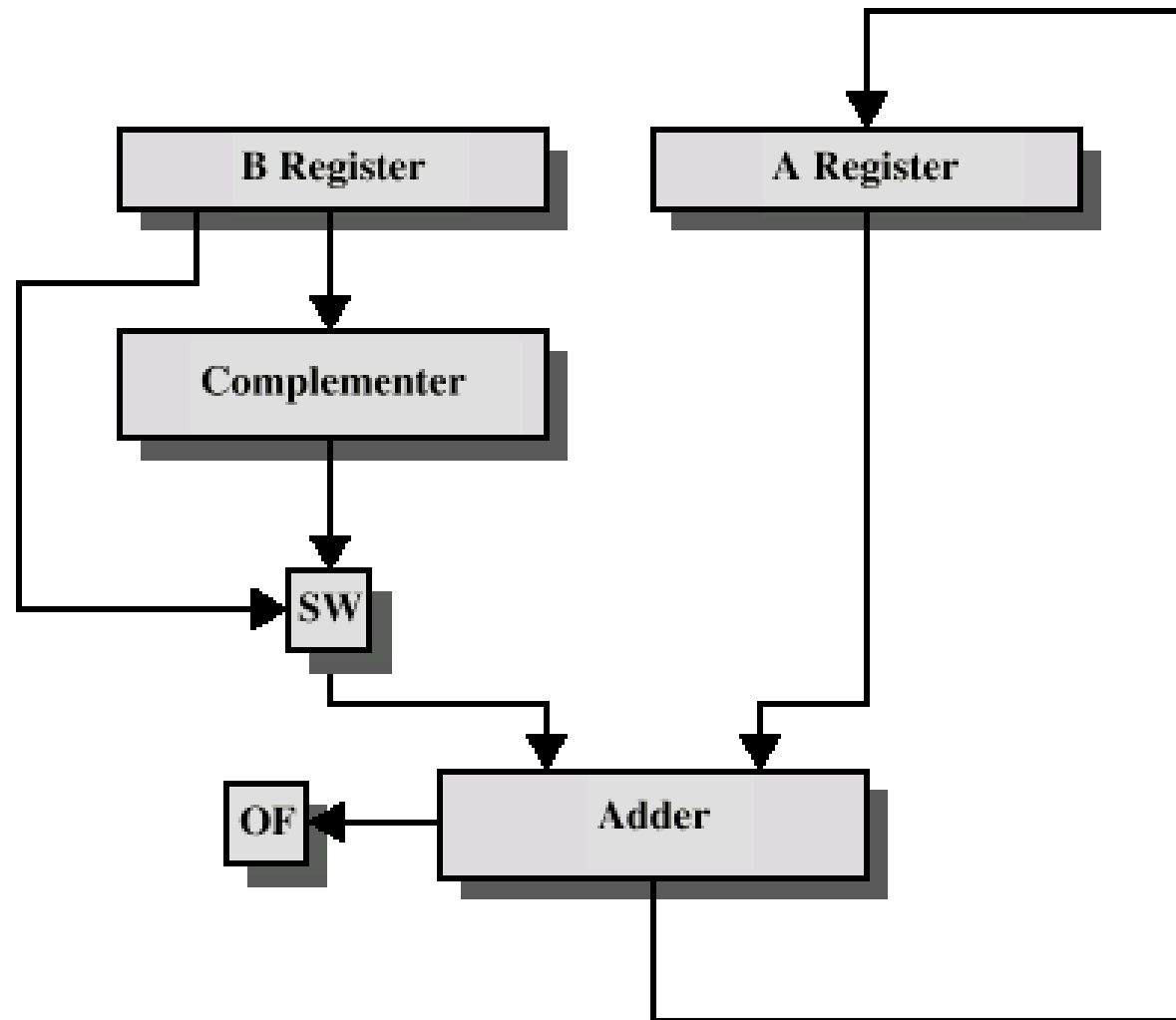
- $-128 = 10000000$
- bitwise not 01111111
- Add 1 to LSB $+1$
- Result 10000000
- So:
- $-(-128) = -128$ **Problem here!**
- Monitor MSB (sign bit)
- It should change during negation
- >> There is no representation of $+128$ in this case. (no $+2^n$)

- Normal binary addition
- $0011 \quad 0101 \quad 1100$
- $+0100 \quad +0100 \quad +1111$
- -----
- $0111 \quad \underline{1}001 = \text{overflow} \quad \underline{1}1011$
- Monitor sign bit for overflow (sign bit change as adding two positive numbers or two negative numbers.)
- Subtraction: Take twos compliment of subtrahend then add to minuend
 - i.e. $a - b = a + (-b)$
- **So we only need addition and complement circuits**

Combined Adder and 2's Complement Subtraction



Hardware for Addition and Subtraction

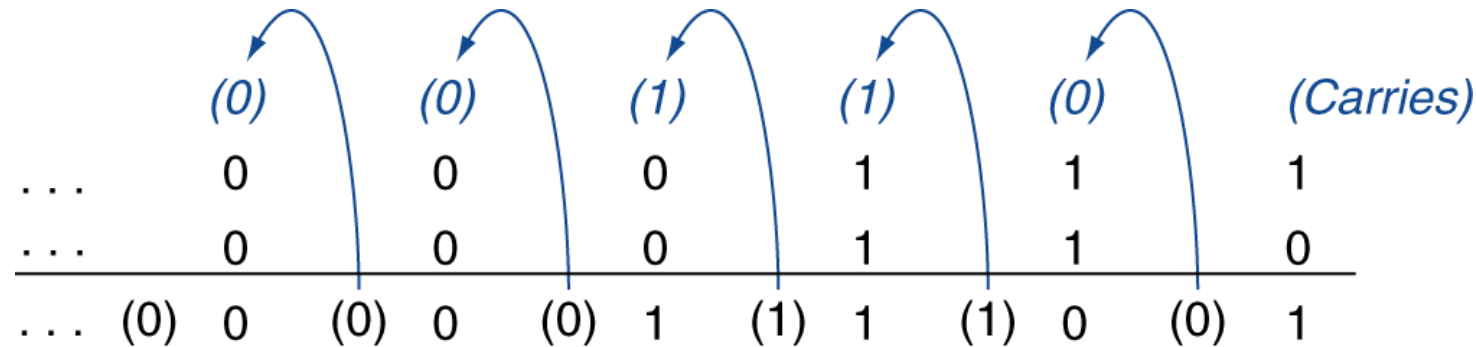


OF = overflow bit

SW = Switch (select addition or subtraction)

Integer Addition - Overflow

- Example: $7 + 6$



- **Overflow** if result is out of range
 - Adding +ve and -ve operands, no overflow
 - Adding two +ve operands
 - Overflow if result sign is 1
 - Adding two -ve operands
 - Overflow if result sign is 0

Integer Subtraction - Overflow



- Add negation of second operand
- Example: $7 - 6 = 7 + (-6)$
 - +7: 0000 0000 ... 0000 0111
 - -6: 1111 1111 ... 1111 1010
 - +1: 0000 0000 ... 0000 0001
- **Overflow** if result is out of range
 - Subtracting two +ve or two -ve operands, no overflow
 - Subtracting +ve from -ve operand
 - Overflow if result sign is 0
 - Subtracting -ve from +ve operand
 - Overflow if result sign is 1

Overflow in Addition / Subtraction

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

$\begin{array}{r} 1001 = -7 \\ + 0101 = 5 \\ \hline 1110 = -2 \end{array}$ <p>(a) $(-7) + (+5)$</p>	$\begin{array}{r} 1100 = -4 \\ + 0100 = 4 \\ \hline 10000 = 0 \end{array}$ <p>(b) $(-4) + (+4)$</p>
$\begin{array}{r} 0011 = 3 \\ + 0100 = 4 \\ \hline 0111 = 7 \end{array}$ <p>(c) $(+3) + (+4)$</p>	$\begin{array}{r} 1100 = -4 \\ + 1111 = -1 \\ \hline 11011 = -5 \end{array}$ <p>(d) $(-4) + (-1)$</p>
$\begin{array}{r} 0101 = 5 \\ + 0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$ <p>(e) $(+5) + (+4)$</p>	$\begin{array}{r} 1001 = -7 \\ + 1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$ <p>(f) $(-7) + (-6)$</p>

Figure 10.3 Addition of Numbers in Twos Complement Representation

$\begin{array}{r} 0010 = 2 \\ + 1001 = -7 \\ \hline 1011 = -5 \end{array}$ <p>(a) $M = 2 = 0010$ $S = 7 = 0111$ $-S = 1001$</p>	$\begin{array}{r} 0101 = 5 \\ + 1110 = -2 \\ \hline 10011 = 3 \end{array}$ <p>(b) $M = 5 = 0101$ $S = 2 = 0010$ $-S = 1110$</p>
$\begin{array}{r} 1011 = -5 \\ + 1110 = -2 \\ \hline 11001 = -7 \end{array}$ <p>(c) $M = -5 = 1011$ $S = 2 = 0010$ $-S = 1110$</p>	$\begin{array}{r} 0101 = 5 \\ + 0010 = 2 \\ \hline 0111 = 7 \end{array}$ <p>(d) $M = 5 = 0101$ $S = -2 = 1110$ $-S = 0010$</p>
$\begin{array}{r} 0111 = 7 \\ + 0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$ <p>(e) $M = 7 = 0111$ $S = -7 = 1001$ $-S = 0111$</p>	$\begin{array}{r} 1010 = -6 \\ + 1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$ <p>(f) $M = -6 = 1010$ $S = 4 = 0100$ $-S = 1100$</p>

Figure 10.4 Subtraction of Numbers in Twos Complement Representation ($M - S$)

- Some languages (e.g., C) ignore overflow
 - Use MIPS addu, addui, subu instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
 - Use MIPS add, addi, sub instructions
 - On overflow, invoke exception handler
 - Save PC in exception program counter (EPC) register
 - Jump to predefined handler address
 - mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu`, no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

- ❖ Carry is useful when adding (subtracting) **unsigned integers**
 - ✧ Carry indicates that the **unsigned sum is out of range**
- ❖ Overflow is useful when adding (subtracting) **signed integers**
 - ✧ Overflow indicates that the **signed sum is out of range**
- ❖ Range for 32-bit unsigned integers = 0 to $(2^{32} - 1)$
- ❖ Range for 32-bit signed integers = -2^{31} to $(2^{31} - 1)$
- ❖ Example 1: Carry = 1, Overflow = 0 (NO overflow)

	11111	1			1	11		1
	1000	0100	0000	0000	1110	0001	0100	0001
+	1111	1111	0000	0000	1111	0101	0010	0000
<hr/>								
	1000	0011	0000	0001	1101	0110	0110	0001

Unsigned sum is out-of-range, but the Signed sum is correct

Another Example of Carry and Overflow



❖ Example 2: Carry = 0, Overflow = 1

	01111	1			11	1		
	0010	0100	0000	0100	1011	0001	0100	0100
+	0111	1111	0111	0000	0011	0101	0000	0010
<hr/>								
	1010	0011	0111	0100	1110	0110	0100	0110

Unsigned sum is correct, but the Signed sum is out-of-range

❖ Example 3: Carry = 1, Overflow = 1

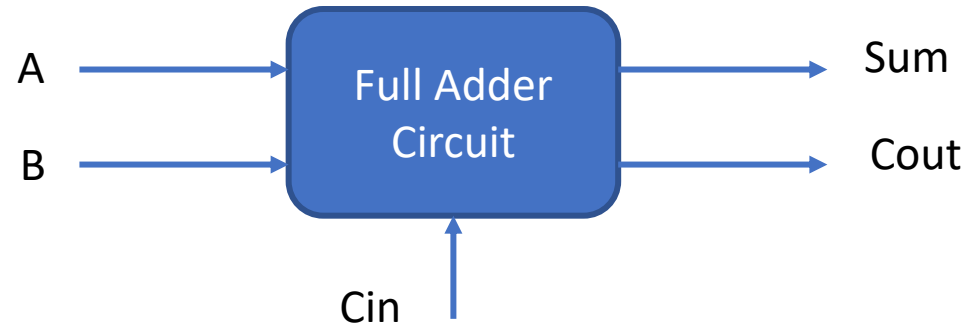
	1	11	1			11	1	
	1000	0100	0000	0100	1011	0001	0100	0100
+	1001	1111	0111	0000	0011	0101	0000	0010
<hr/>								
	0010	0011	0111	0100	1110	0110	0100	0110

Both the Unsigned and Signed sums are out-of-range

Logic Circuit of Adder / Subtractor

1 Bit Full Adder Circuit

Truth table represents **behaviour** of inputs and outputs



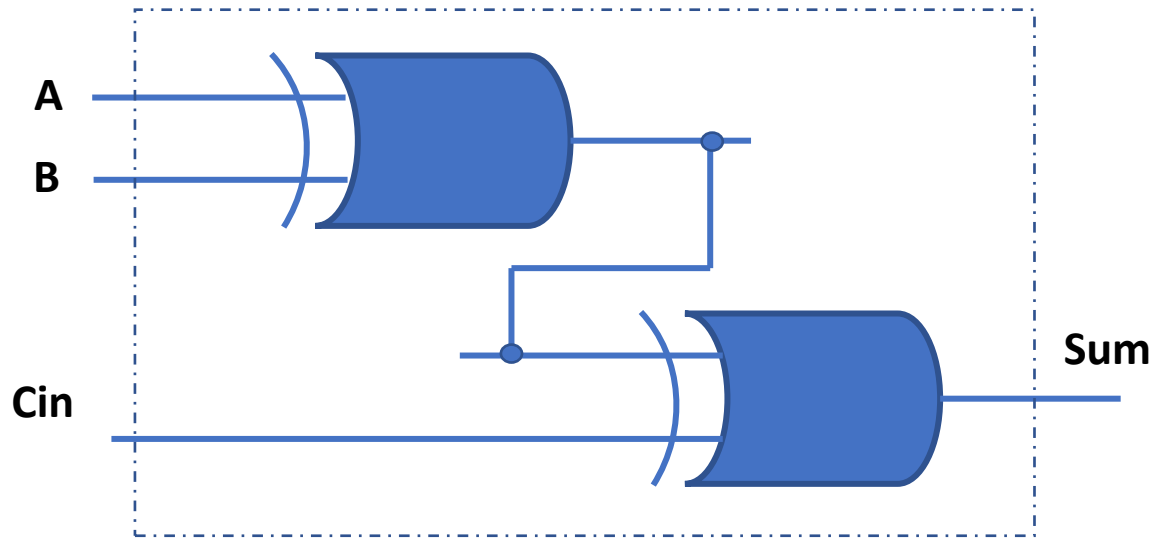
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Boolean Expression:

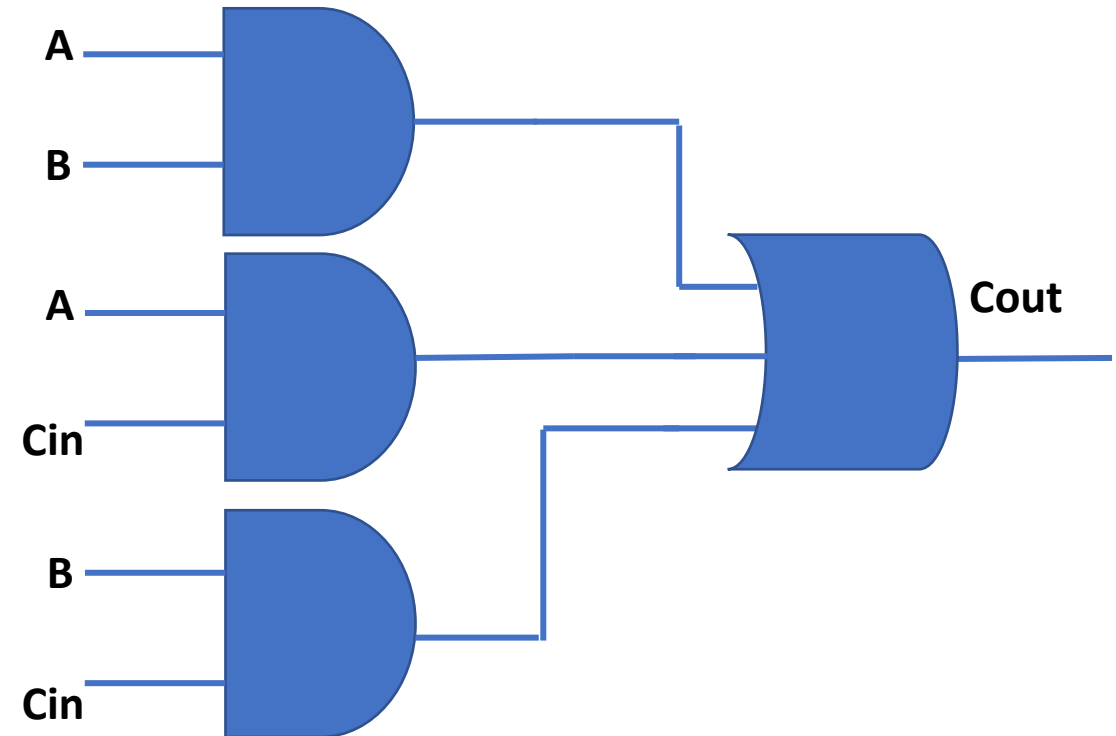
$$\text{Sum } S_i = X_i \oplus Y_i \oplus C_i$$

$$C_{i+1} = X_i \cdot Y_i + C_i (X_i \oplus Y_i)$$

Full Adder Implementation using Logic Gates



$$Sum = A \oplus B \oplus Cin$$

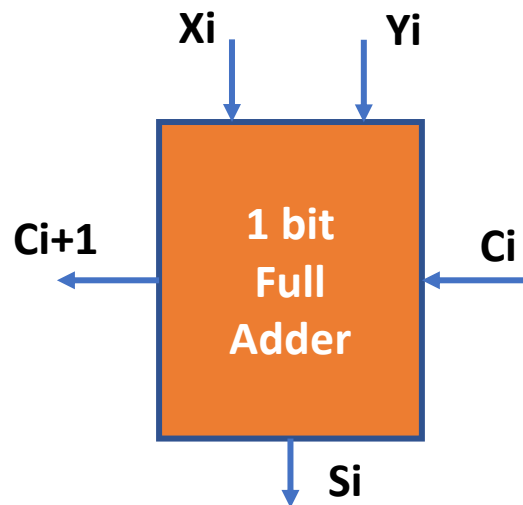


$$Cout = AB + ACin + BCin$$

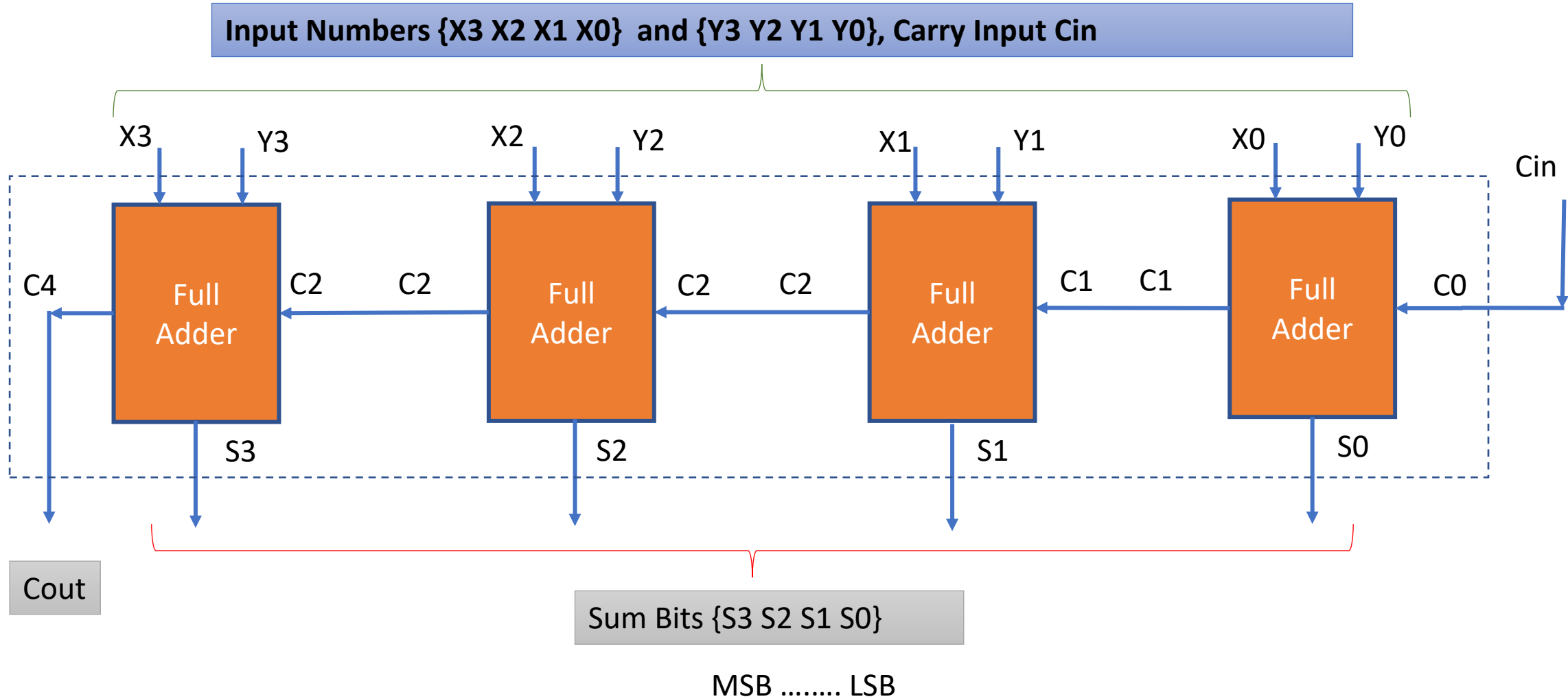
Array of Adders?

Usually, data types defined in any program are 8 bits, 16 bits or 32 bits

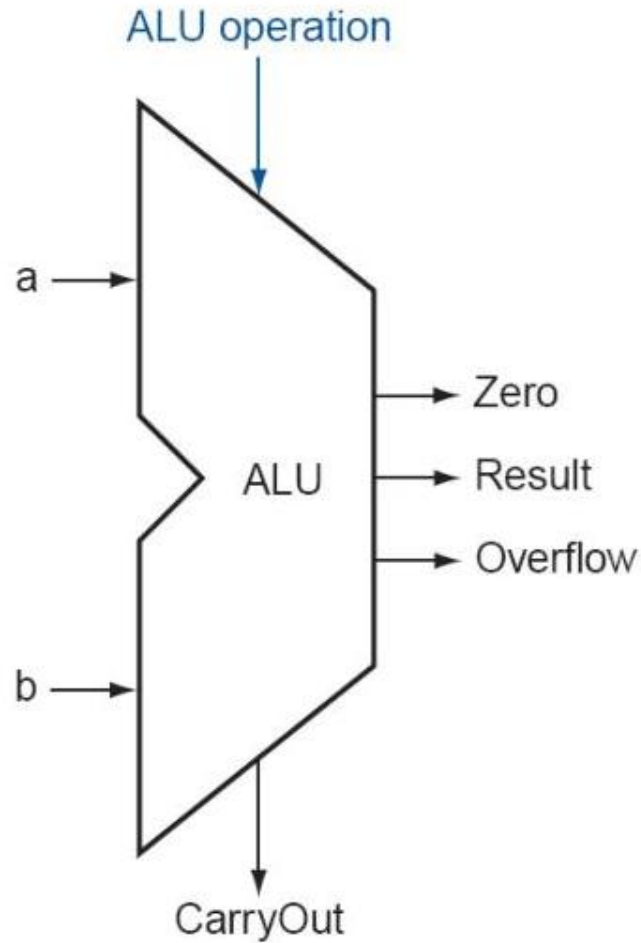
We need array of arithmetic and logic circuits to perform ALU operation on CPU registers



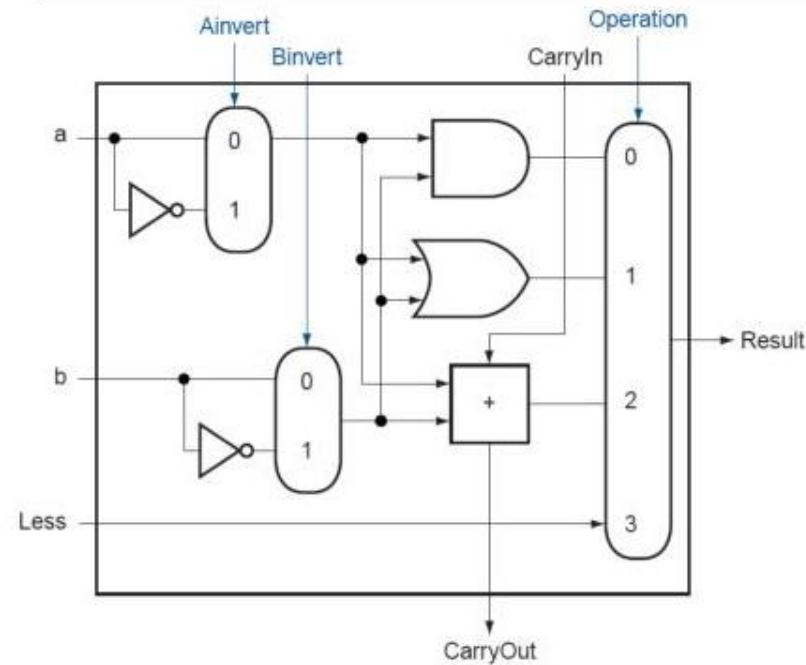
Multiple Bits – Ripple Carry Adder



Simple ALU Architecture



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR



- P&H Textbook Chapter 3
- Search Appendix C P&H Textbook online; this contains useful background material on digital logic, ALU and related stuff