CS/EE 320 Computer Organization and Assembly Language Spring 2025

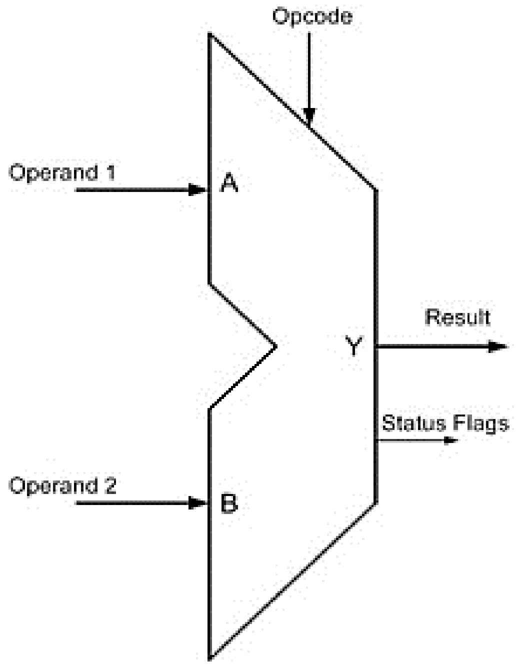
Shahid Masud

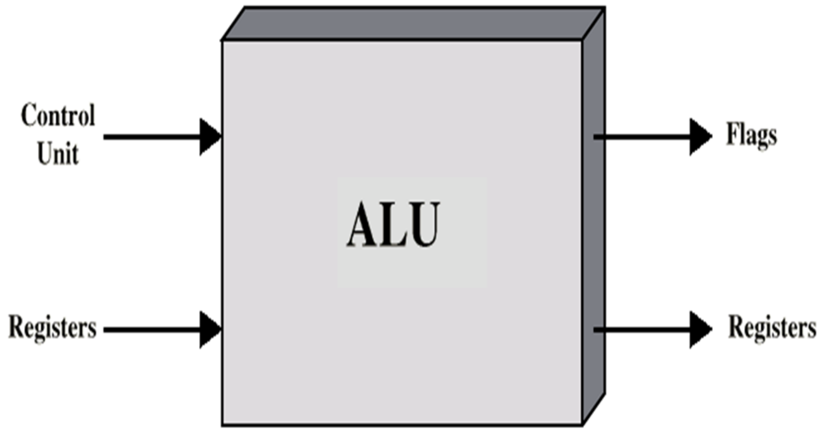
Lecture 9

Topics

- Logic and Arithmetic Instructions in Assembly Language -**ALU**
- 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

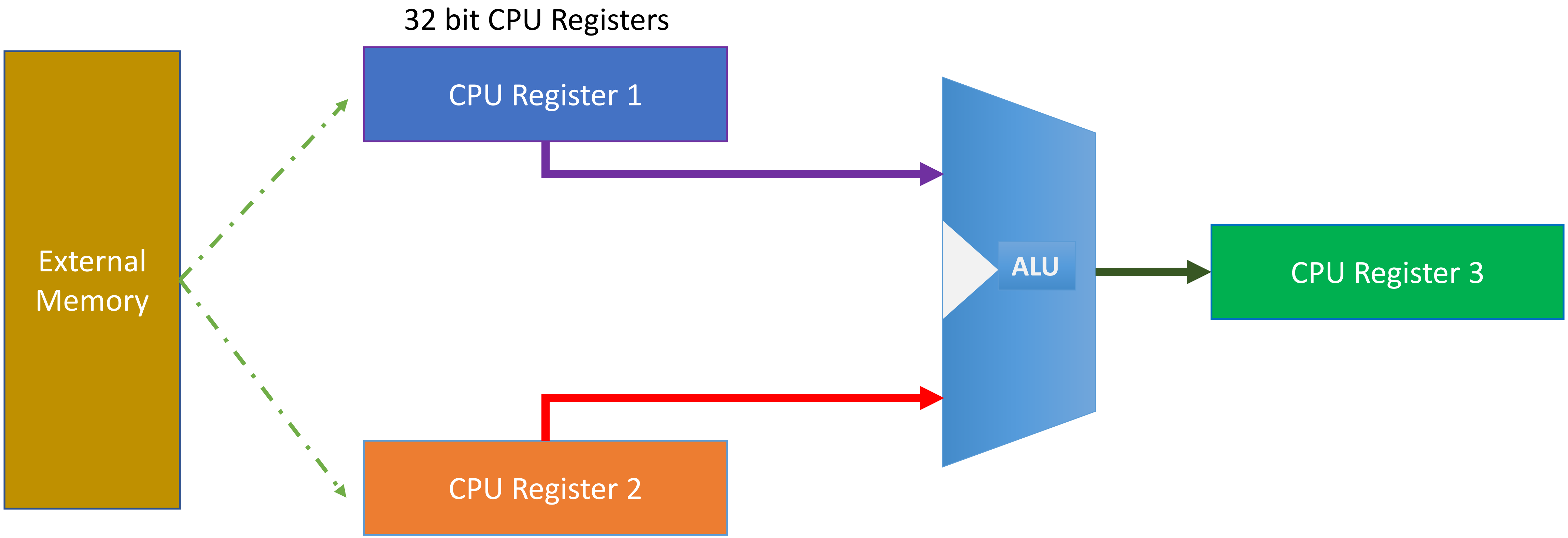
Arithmetic & Logic Unit





- 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



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 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = 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 | $s3	Three reg. operands; bit-by-bit OR
	NOR	NOR $s1,$s2,$s3	$s1 = ~ ($s2 |$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 | 100	Bit-by-bit OR with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant

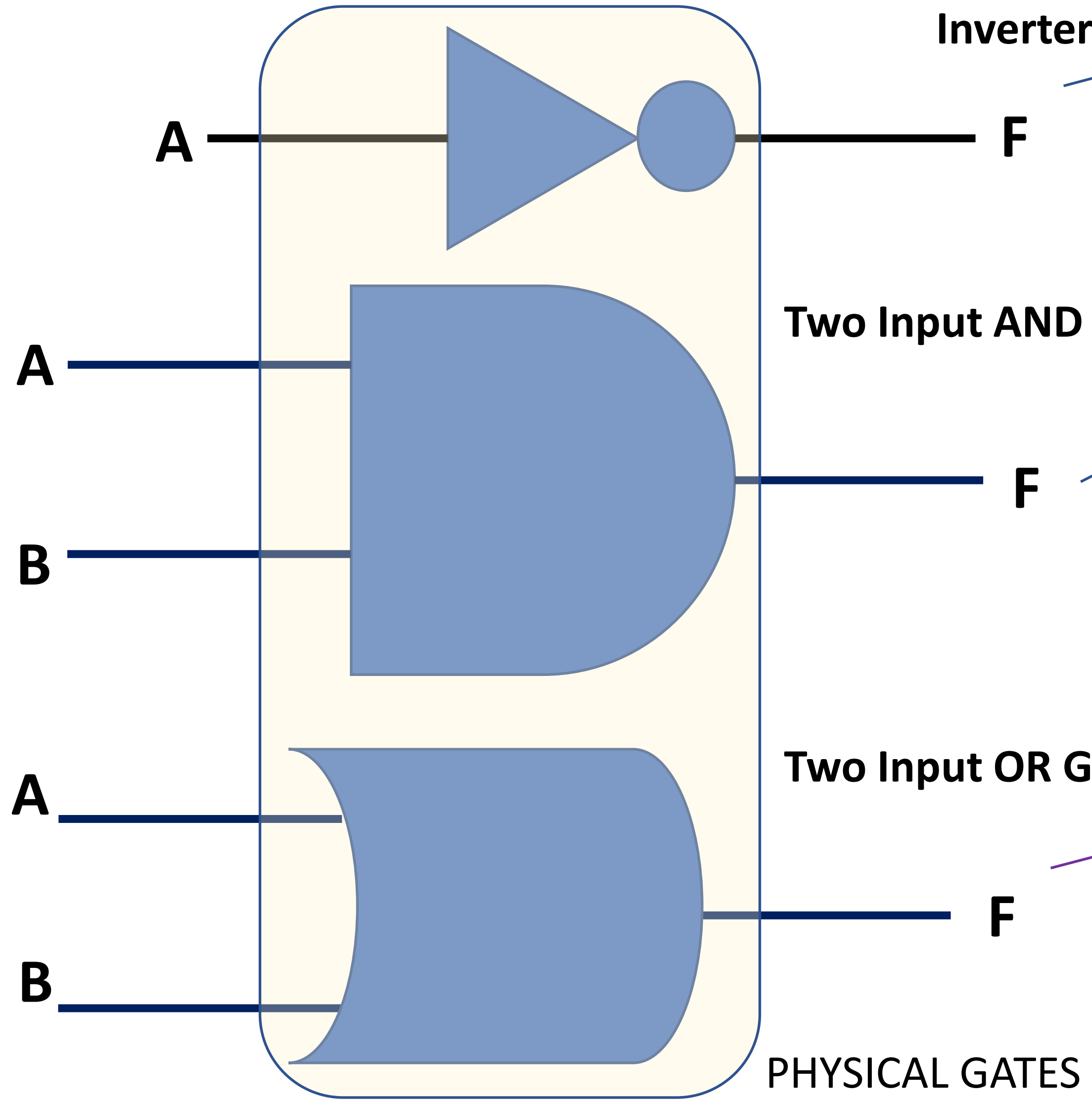
Arithmetic for Computers

- 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)



Inverter

A	F
0	1
1	0

Two Input AND Gate

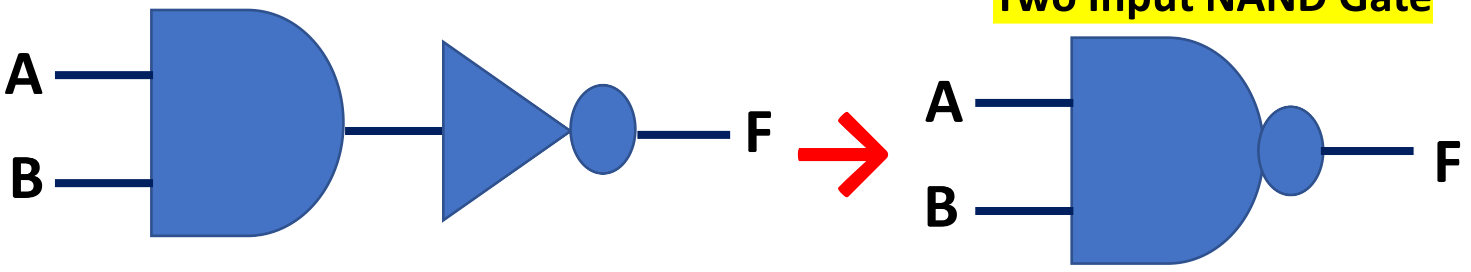
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Two Input OR Gate

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

Gates (Compound Gates)

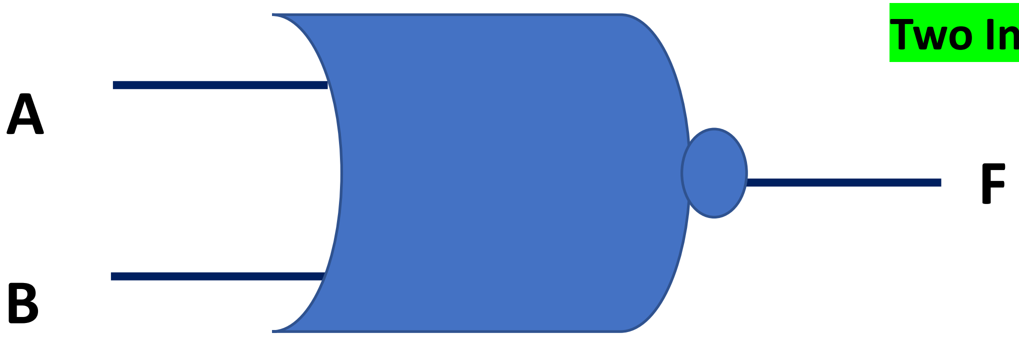
Two Input NAND Gate



AND Gate followed by Inverter

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

Two Input NOR Gate

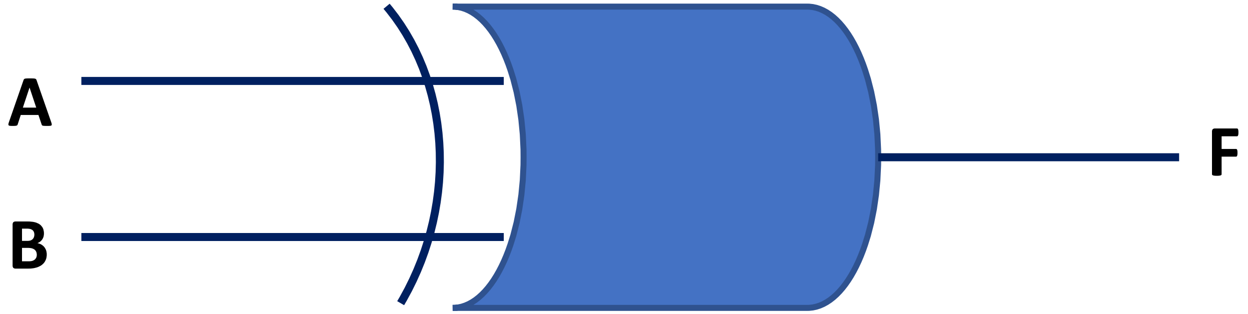


OR Gate followed by Inverter

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

Complex Gates – the XOR and XNOR

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

Number Systems

- 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 = (8x10) + 4
  - 4728 = (4x1000)+(7x100)+(2x10)+8
  - Base or radix of 10 → 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 = $(4 \times 10^3)+(7 \times 10^2)+(2 \times 10^1)+(8 \times 10^0)$

Binary Number System

- Uses only two digits, 0 and 1
- It is base or radix of 2
- Each digit has a **value** depending on its **position**:
  - $10_2 = (1x2^1)+(0x2^0) = 2_{10}$
  - $11_2 = (1x2^1)+(1x2^0) = 3_{10}$
  - $100_2 = (1x2^2)+ (0x2^1)+(0x2^0) = 4_{10}$
  - $1001.101_2 = (1x2^3)+(0x2^2)+ (0x2^1)+(1x2^0)$
    $+(1x2^{-1})+(0x2^{-2})+(1x2^{-3}) = 9.625_{10}$

Decimal to Binary conversion

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

Decimal number to Binary

- Fractional values, e.g.
  - $472.83 = (4x10^{2})+(7x10^{1})+(2x10^{0})+(8x10^{-1})+(3x10^{-2})$

Whole Number Part	Fractional Part

  - In general, for the decimal representation of
  - $X = \{\ldots\ x_2x_1x_0.x_{-1}x_{-2}x_{-3}\ \ldots\ \}$

    $X = \sum_i x_i 10^i$

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
  - .81x2 = 1.62 integral part 1
  - .62x2 = 1.24 integral part 1
  - .24x2 = 0.48 integral part 0
  - .48x2 = 0.96 integral part 0
  - .96x2 = 1.92 integral part 1
  - Binary number .11001 (approximate)

Hexadecimal Notation

- Compact representation of bus information
- Use 16 digits, (0,1,3,…9,A,B,C,D,E,F)
- $1A_{16} = (1_{16} \times 16^{1})+(A_{16} \times 16^{o})$
  $= (1_{10} \times 16^{1})+(10_{10} \times 16^{0})=26_{10}$
- 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

Integer Representation +/- numbers

- 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 = **0**0010010
- -18 = **1**0010010
- Problems
  - Need to consider both sign and magnitude in arithmetic
  - Two representations of zero (+0 and -0)

Two’s Compliment Representation

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

2’s Compliment Number System

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

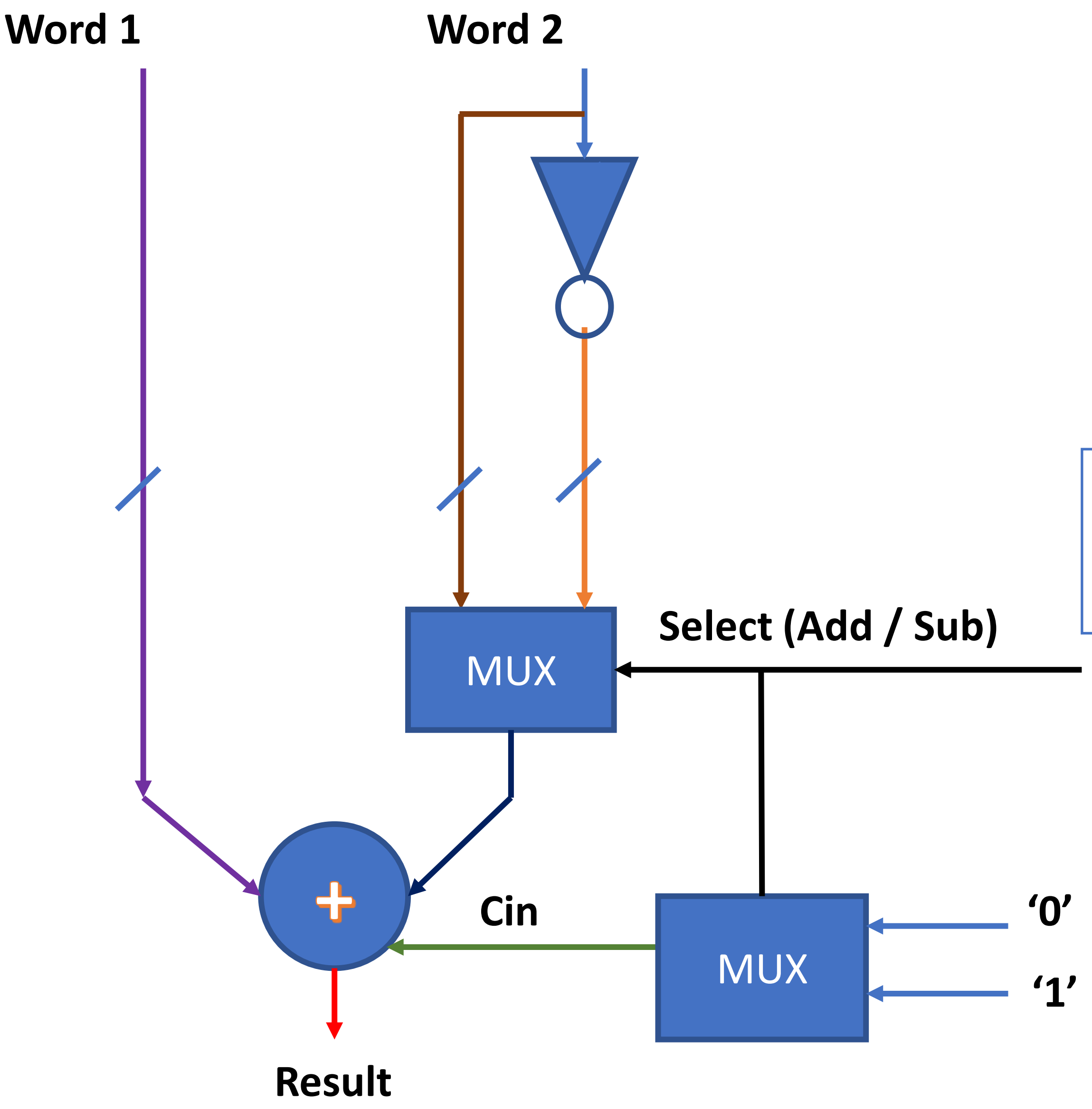
Range of 2's Compliment Numbers

- 8 bit 2's compliment
  - +127 = 01111111 = 2^7 -1
  - -128 = 10000000 = -2^7
- 16 bit 2's compliment
  - +32767 = 011111111 11111111 = 2^{15} - 1
  - -32768 = 100000000 00000000 = -2^{15}

Different word lengths in 2's Compliment

- 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)

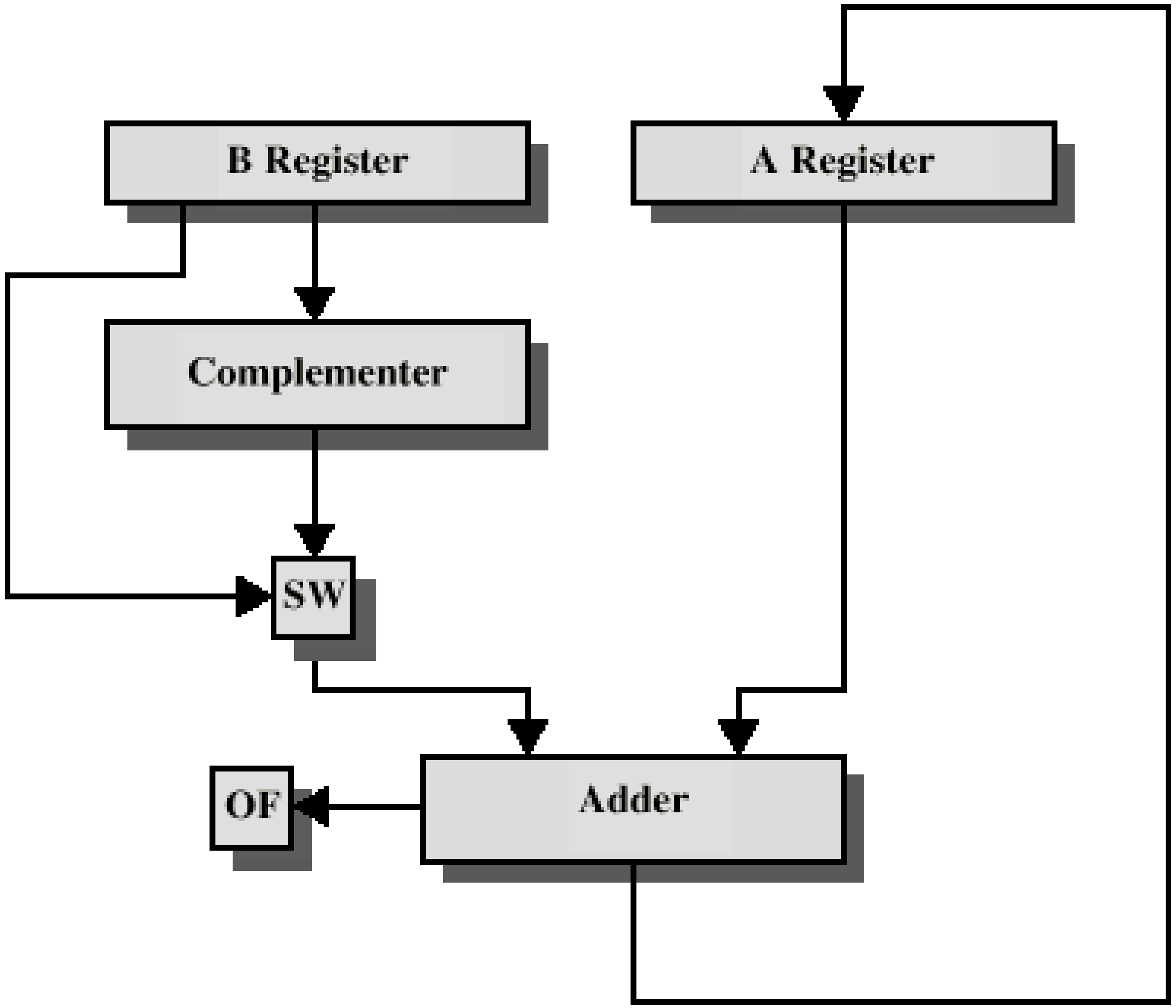
Combined Adder and 2’s Complement Subtraction



For 2’s Complement,
Select the path through inverter and Cin = ‘1’

Select ‘1’ for 2’s Complement

Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

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 $+2^n$)

Addition and Subtraction

- Normal binary addition
- 0011 0101 1100
- +0100 +0100 +1111
- -------- ---------- ------------
- 0111 1001 = overflow 11011
- 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**

Integer Addition - Overflow

- Example: 7 + 6

		(0)		*(0)*		*(1)*		*(1)*		*(0)*		*(Carries)*
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- **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.

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Figure 10.3 Addition of Numbers in Twos Complement Representation

0010 = 2 +1001 = −7 1011 = −5 (a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 +1110 = −2 10011 = 3 (b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7 (c) M = −5 = 1011 S = 2 = 0010 −S = 1110	0101 = 5 +0010 = 2 0111 = 7 (d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = Overflow (e) M = 7 = 0111 S = −7 = 1001 −S = 0111	1010 = −6 +1100 = −4 10110 = Overflow (f) M = −6 = 1010 S = 4 = 0100 −S = 1100

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

CPU Dealing with Overflow

- 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

MIPS Instructions and Overflow

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 vs Overflow

- 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
 ----------------------------------------
  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
 ----------------------------------------
  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

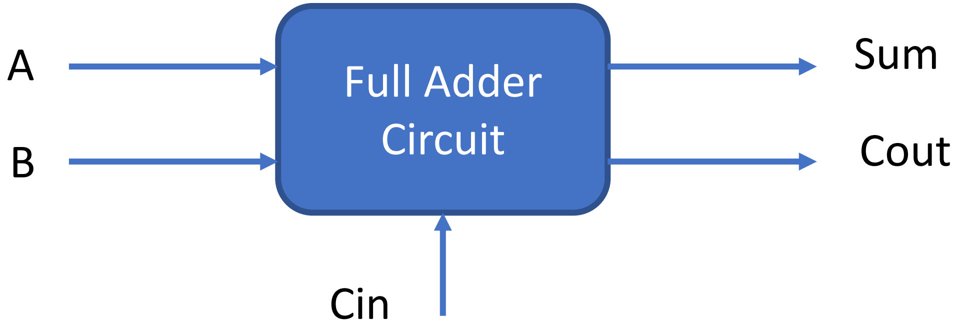
```
 1  11 1                      11    1
  1000 0100 0000 0100 1011 0001 0100 0100
+ 1001 1111 0111 0000 0011 0101 0000 0010
 ----------------------------------------
  0010 0011 0111 0100 1110 0110 0100 0110
```

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



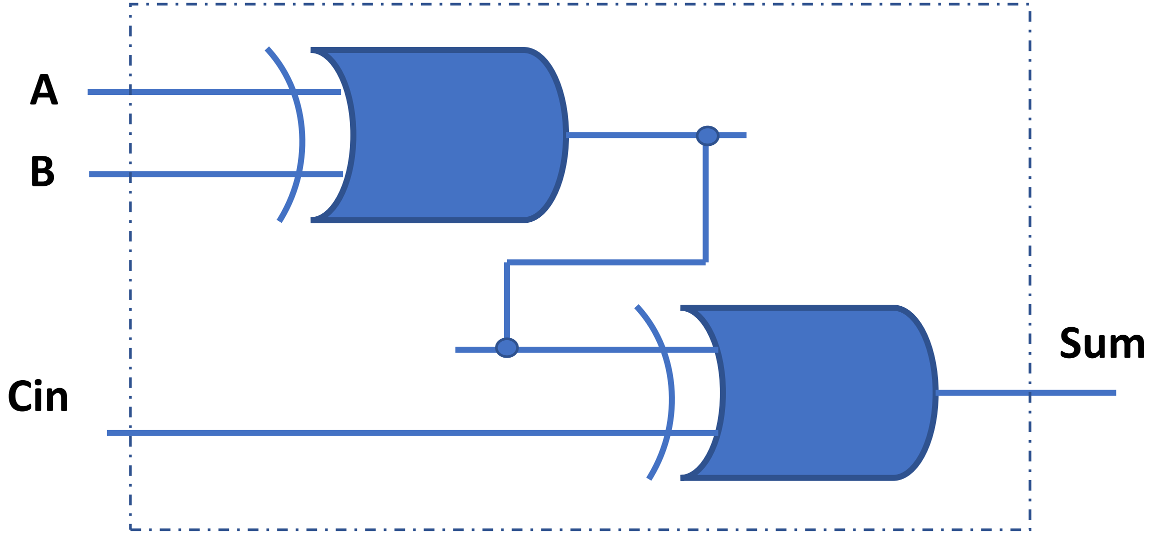
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Boolean Expression:

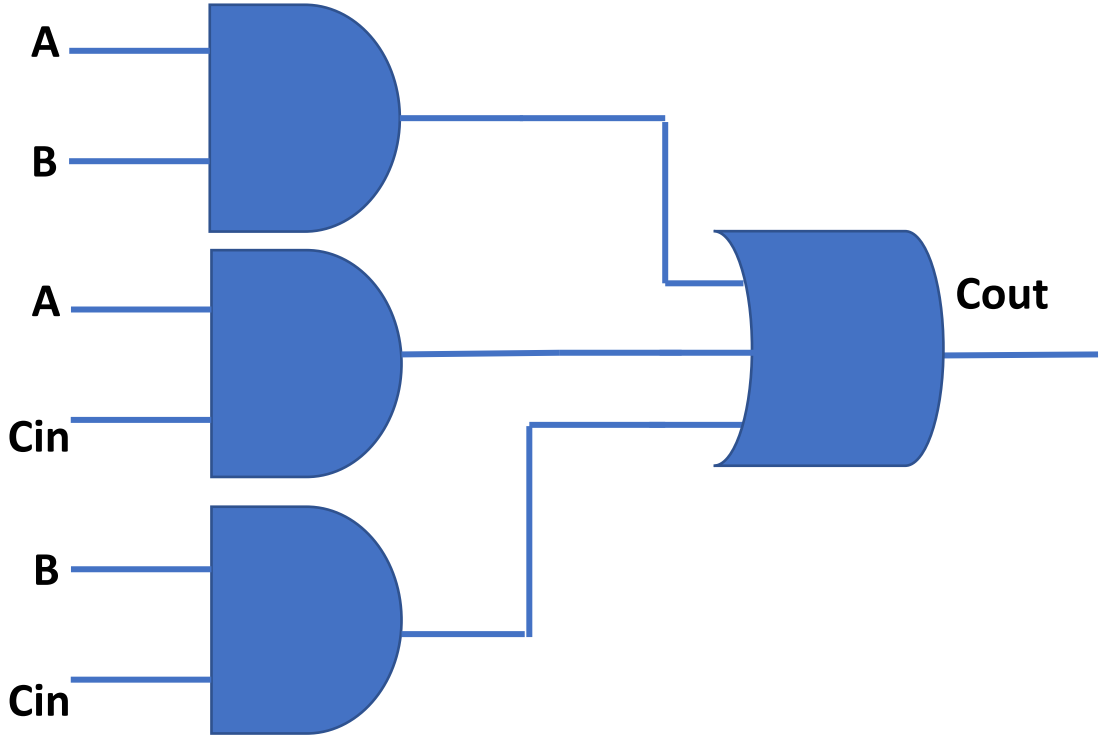
Sum $Si = Xi \oplus Yi \oplus Ci$

$Ci+1 = Xi.Yi + Ci\,(Xi \oplus Yi)$

Full Adder Implementation using Logic Gates



Sum = A⊕B ⊕Cin

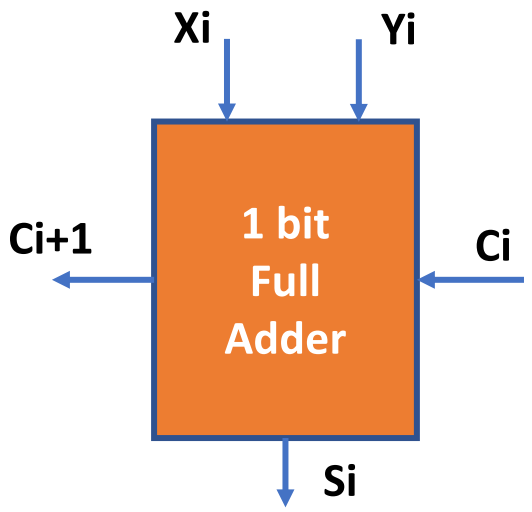


Cout = $AB + ACin + BCin$

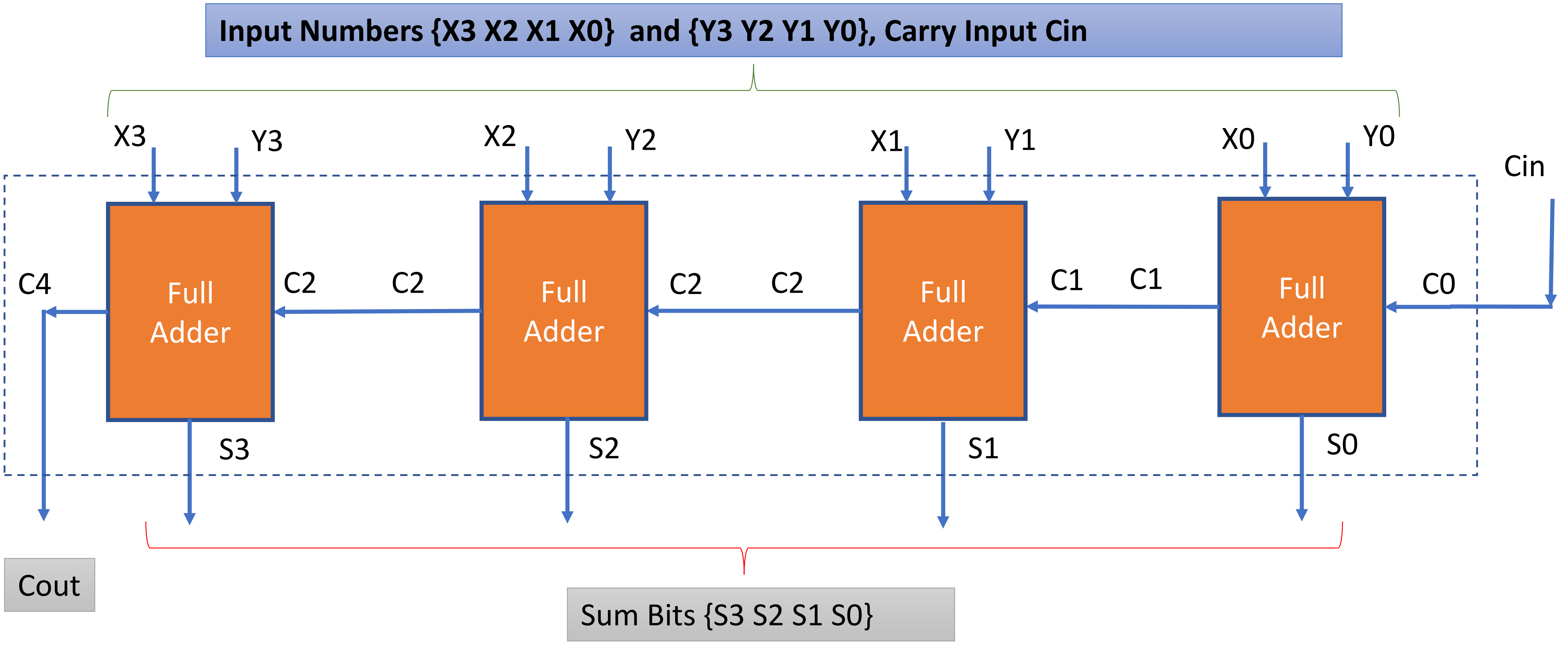
Array of Adders?

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

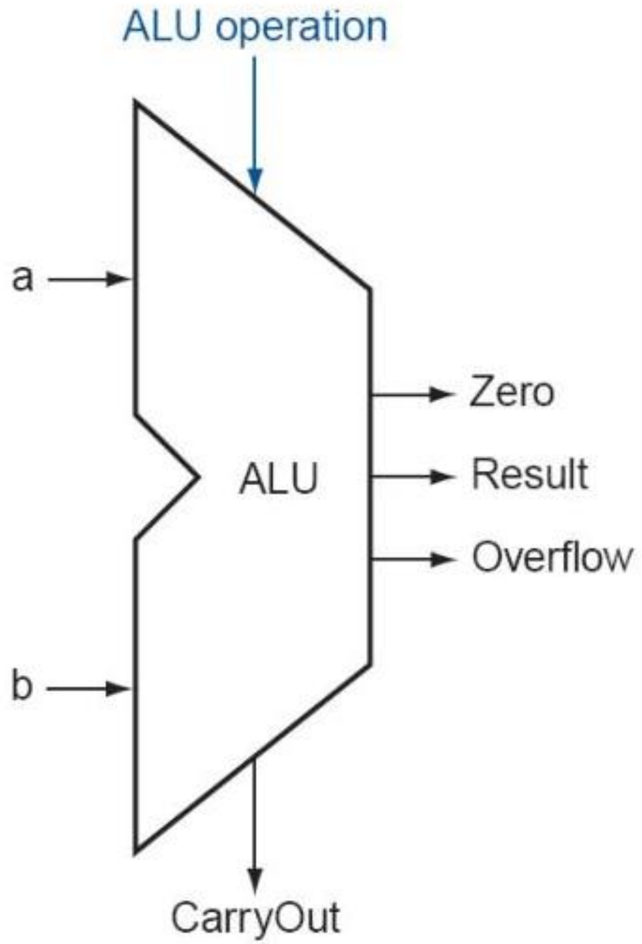


Multiple Bits – Ripple Carry Adder

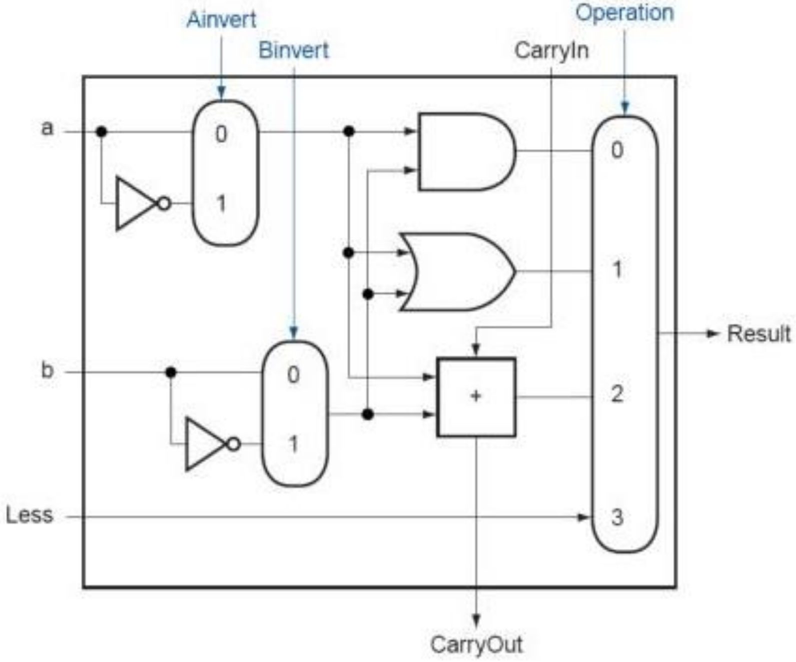


MSB …….. LSB

Simple ALU Architecture



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



Readings

- P&H Textbook Chapter 3

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