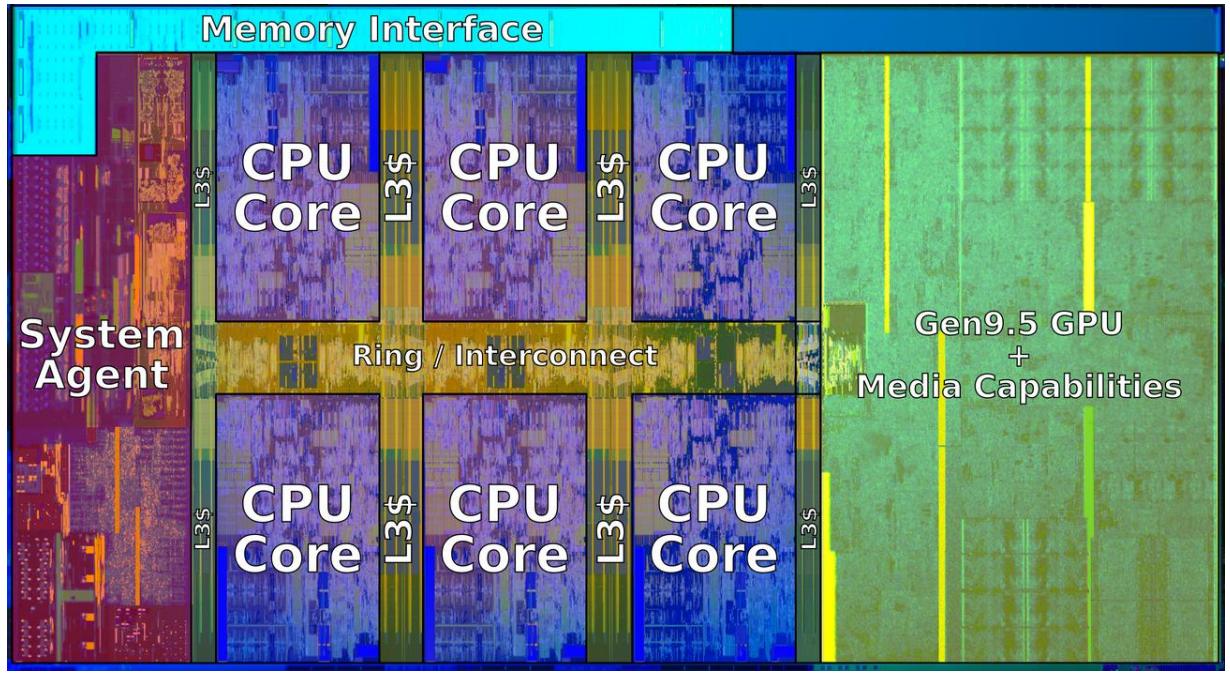
COE 530
Quantum Computer And Architecture

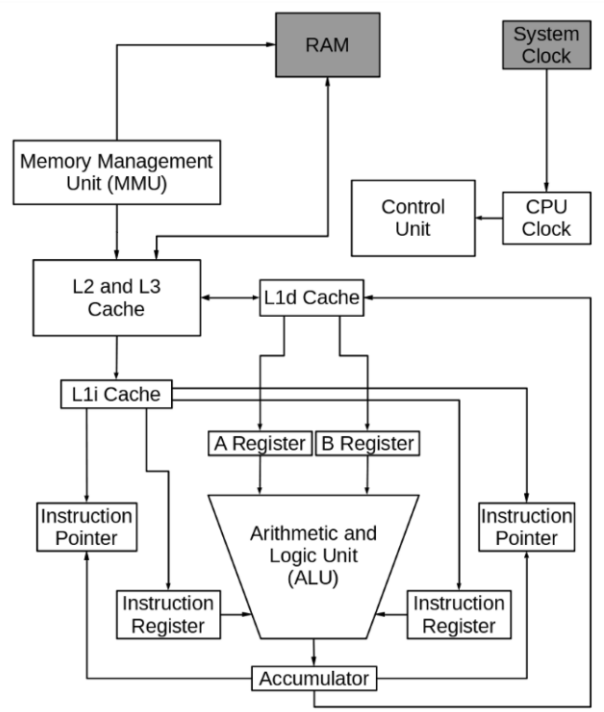
Lecture 2

Classical Computer System II

ALU

State of the art CPU



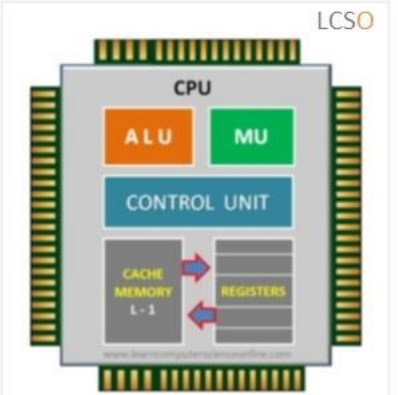


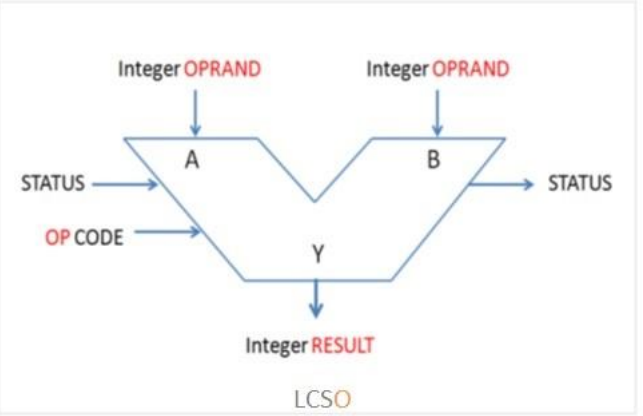
https://www.redhat.com/sysadmin/cpu-components-functionality

This slide is from CMU course

Arithmatic and Logic Unit (ALU)

- CPU executes computer programs, which contains set of machine instructions
- Machine instructions are set of arithmatic and logical operations
- ALU is a combinational digital circuit that performs pre-identified set of arithmatic and logical operations, example
  - Addition
  - Subtraction
  - Multiplication
  - AND
  - OR
  - Etc





https://www.learncomputerscienceonline.com/arithmetic-logic-unit/

In general, ALU takes two inputs (OPRANDS) and an instruction (OP CODE) and produces the RESULTS

Binary Addition

- Suppose we want to add two numbers 11 and 11
  - Recall that $11=1*2^3+0*2^2+1*2^1+1*2^0=(1011)_2$

    $14= 1*2^3+1*2^2+1*2^1+0*2^0=(1110)_2$

- We know that 11+11=25, but how is it done in binary?

$$\begin{array}{r} 1011 \\ +\ 1110 \\ \hline \end{array}$$

Binary Addition

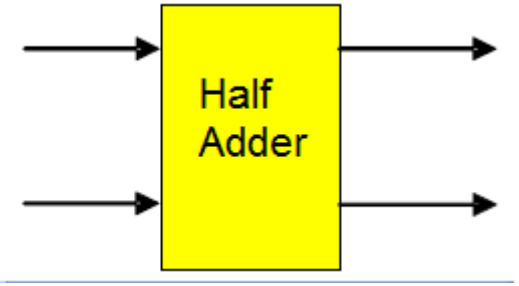
- Binary addition is used frequently

- "Addition" Development:
  - Half-Adder (HA), a 2-input bit-wise addition functional block
  - Full-Adder (FA), a 3-input bit-wise addition functional block
  - Ripple Carry Adder, an iterative array to perform binary addition
  - Carry-Look-Ahead Adder (CLA), Speeds up performance by generating carries from the input numbers directly to avoid carry ripple delay

Half-Adder

- A half adder (HA) is an arithmetic circuit that is used to add two bits. The block diagram of HA is shown. It has two inputs and two outputs.
- A 2-input (no carry input), 1-bit width binary adder that performs the following computations:

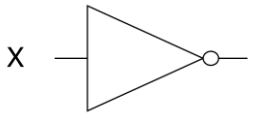
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0



- A half adder adds two bits to produce two bits: S & C
- The output is expressed as a sum bit , S and a carry bit, C
- The half adder can be specified as a truth table for S and C

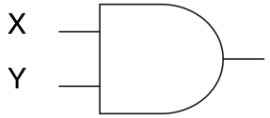
INPUTS		OUTPUTS	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Review of Boolean Algebra



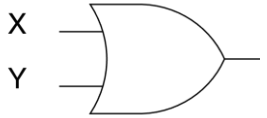
X	Z
0	1
1	0

NOT
Truth Table
$Z = \bar{X}$



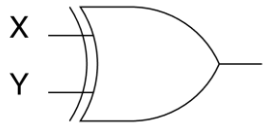
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

OR
Truth Table
$Z = XY$



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

AND
Truth Table
$Z = X + Y$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

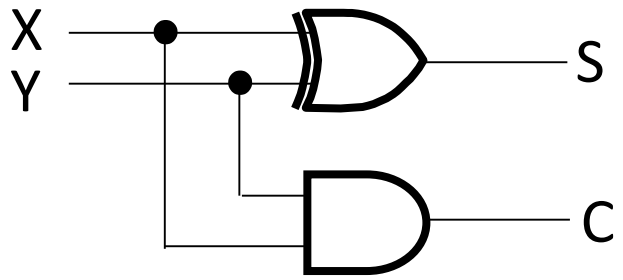
XOR
Truth Table
$Z = X \oplus Y$

Implementations: Half-Adder

- The most common half adder implementation is:

INPUTS		OUTPUTS	
X	**Y**	**C**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = X \otimes Y$$
$$C = XY$$



Transistors as Physical Logic Gates

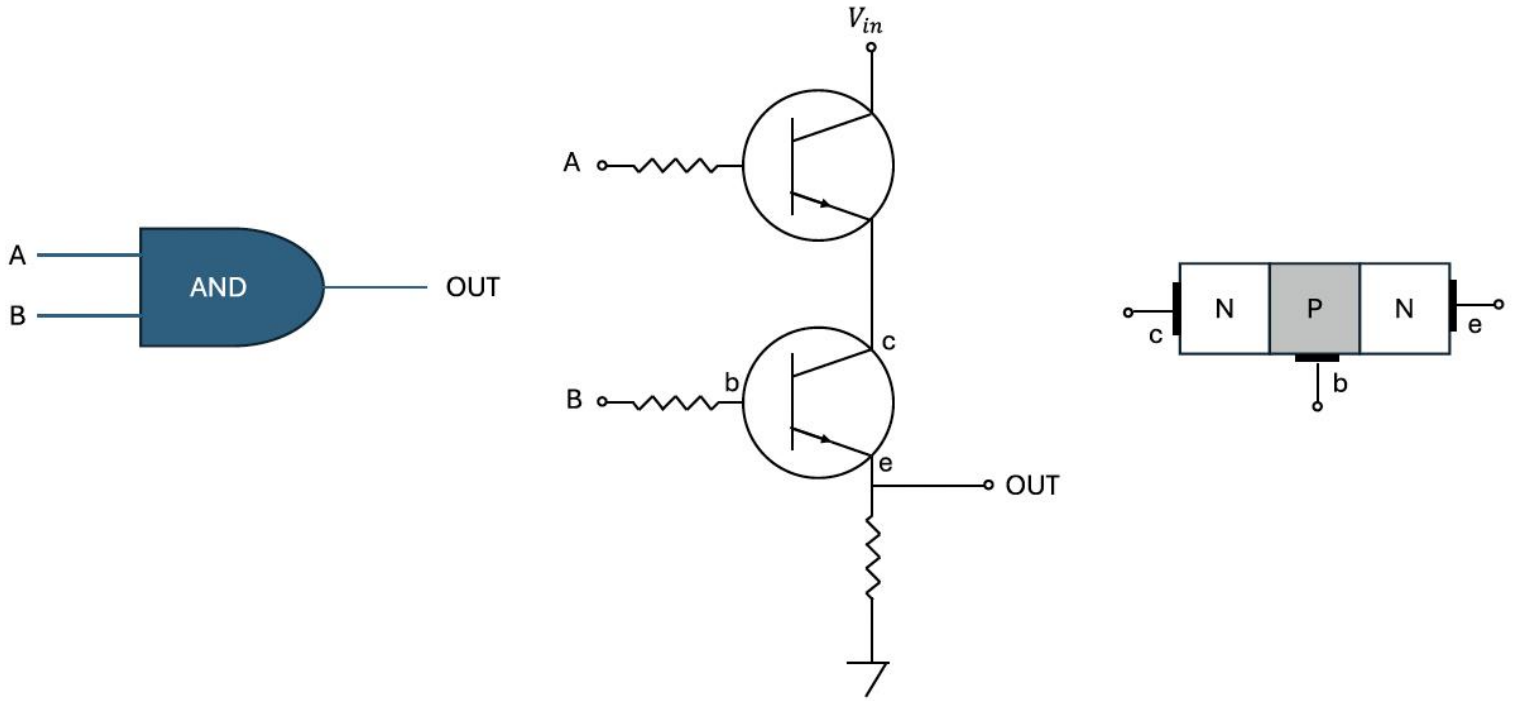
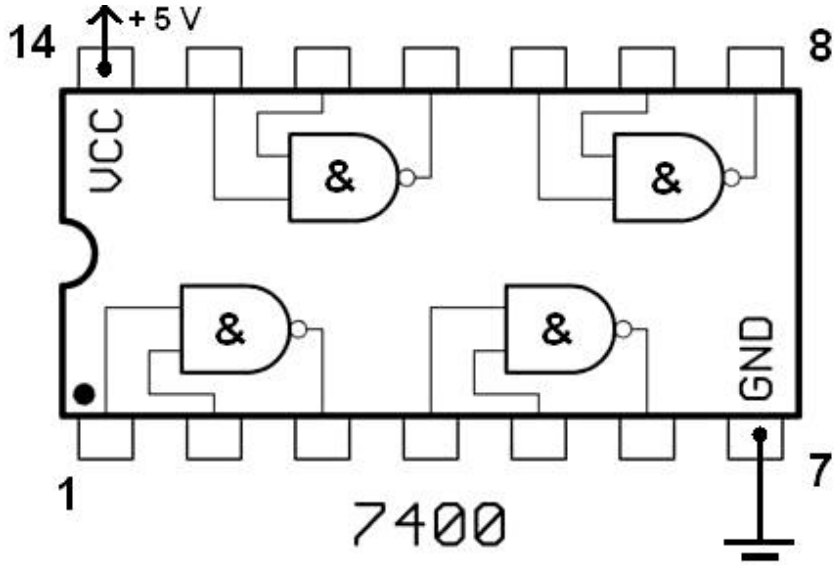
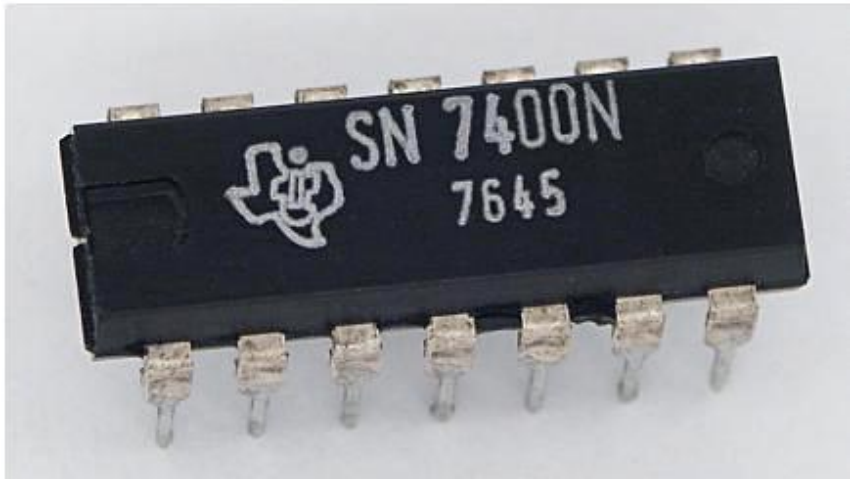
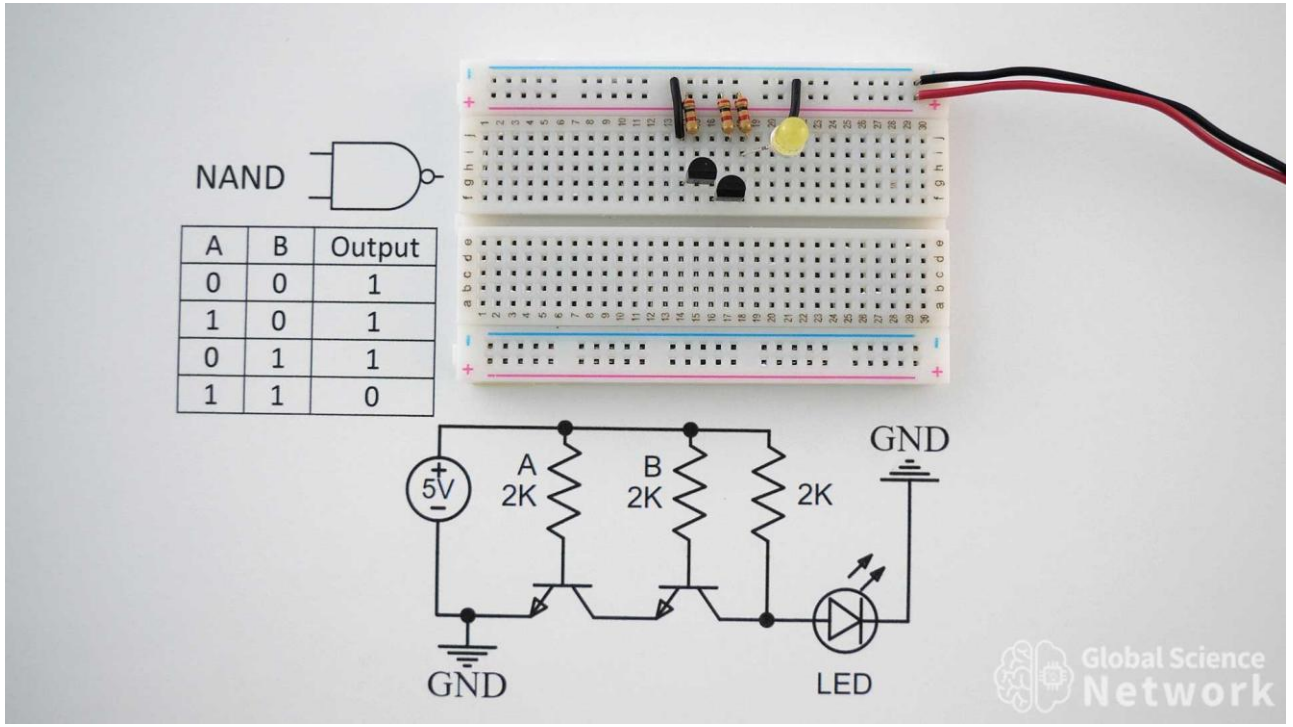


Fig. 4.2 Left: AND gate. Center: AND gate implemented using bipolar junction transistors of NPN type. Right: Schematic of a bipolar junction transistor of NPN type with labeled doped regions

NAND Gate as Hardware







Full-Adder

- A full adder is similar to a half adder but includes a carry-in bit from lower stages. Like the half-adder, it computes a sum bit, S and a carry bit, C.
- For a carry-in (Z) of 0, it is the same as the half-adder:

Z	0	0	0	0
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

- For a carry- in (Z) of 1:

Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

Exercise

- What is the circuit for Full-Adder?

Full Adder Truth Table

X	Y	Z	C	S
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

- We need a way to write functions from truth table?

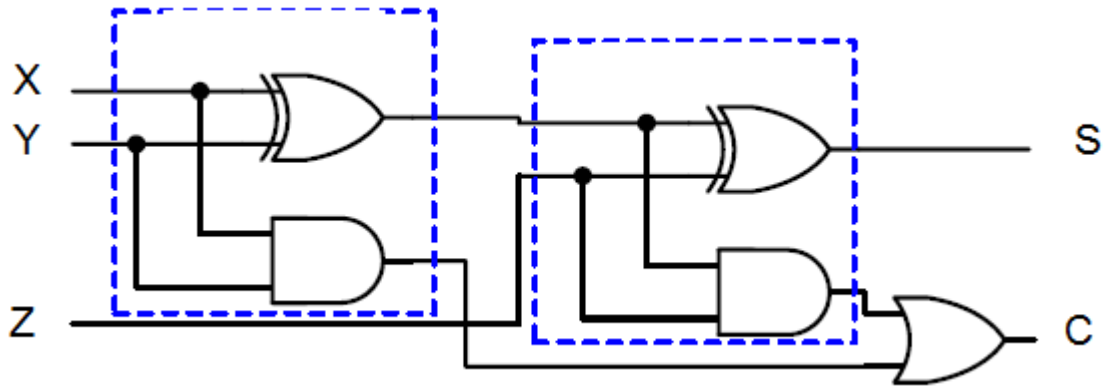
Equations: Full-Adder

- The Boolean functions for the sum and carry outputs can be manipulated to simplify the circuit, as shown below:

$$S = \overline{X}\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z} + XYZ$$
$$= X \oplus Y \oplus Z$$
$$= (X \oplus Y) \oplus Z$$

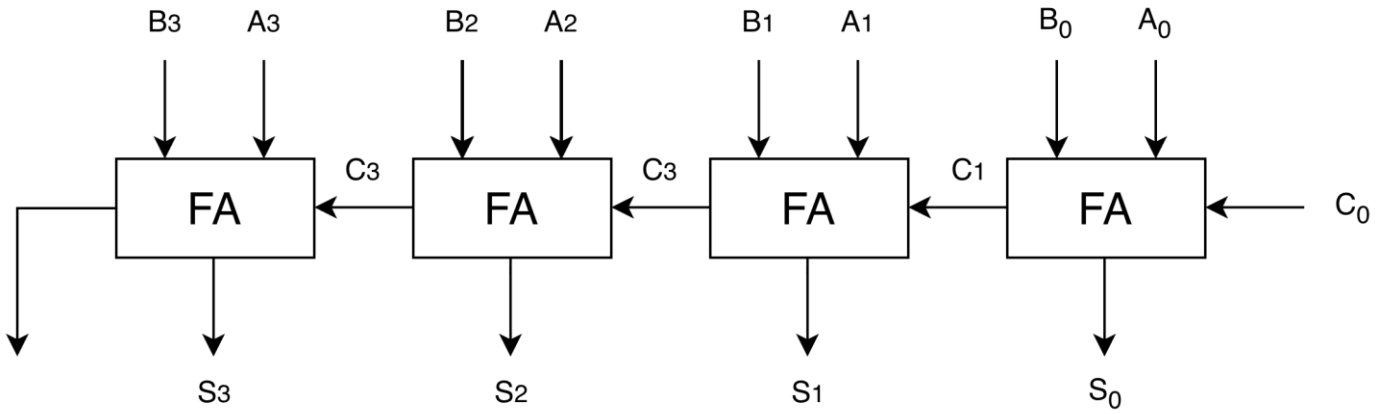
$$C = XY + X\overline{Y}Z + \overline{X}YZ$$
$$= XY + Z(X\overline{Y} + \overline{X}Y)$$
$$= XY + Z(X \oplus Y)$$

- Thus the full adder can be implemented using two half adders and an OR gate as shown below:



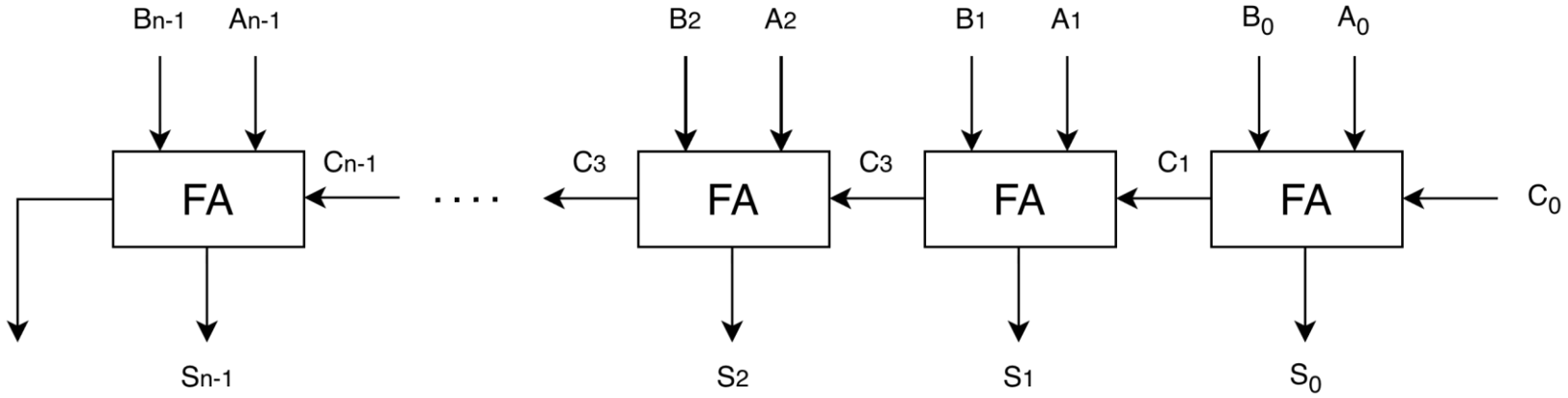
4-bit Ripple-Carry Adder (RCA)

- A 4-bit Ripple-Carry Adder made from four 1-bit Full Adders:



$$\begin{array}{r} C_0 \\ B_3B_2B_1B_0 \\ +\ A_3A_2A_1A_0 \\ \hline C_4S_3S_2S_1S_0 \end{array}$$

- N-bit Ripple-Carry Adder:



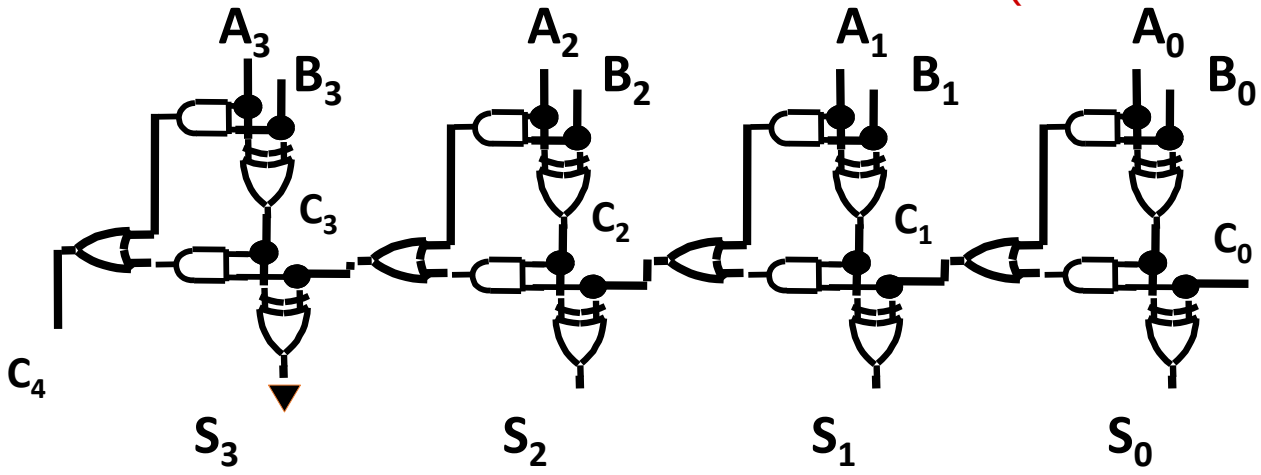
$$\begin{array}{r} C_0 \\ B_{n-1}\ldots B_2B_1B_0 \\ +\ A_{n-1}\ldots A_2A_1A_0 \\ \hline C_nS_{n-1}\ldots S_2S_1S_0 \end{array}$$

4-bit RCA: Carry Propagation & Delay*

- One problem with the addition of binary numbers is the length of time taken to propagate the ripple carry from the least significant bit to the most significant bit.
- Gate-level propagation path for the 4-bit

  ripple carry adder (XOR = 3 gate delays)

Critical (longest) delay path
(Worst case scenario)



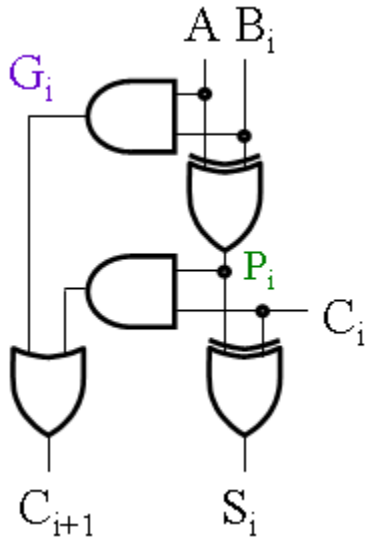
Longest Total Carry Ripple Delay from inputs to S3 = 3 + 3 + 2(n-1) gate delays
where n is the number of stages (= 4 here) → 12 gate delays

Carry Lookahead Adder (CLA)*

- Defining the equations for the Full Adder in terms of the P_i and G_i:

$$P_i = A_i \oplus B_i \qquad G_i = A_i B_i$$

$$S_i = P_i \oplus C_i \qquad C_{i+1} = G_i + P_i C_i$$

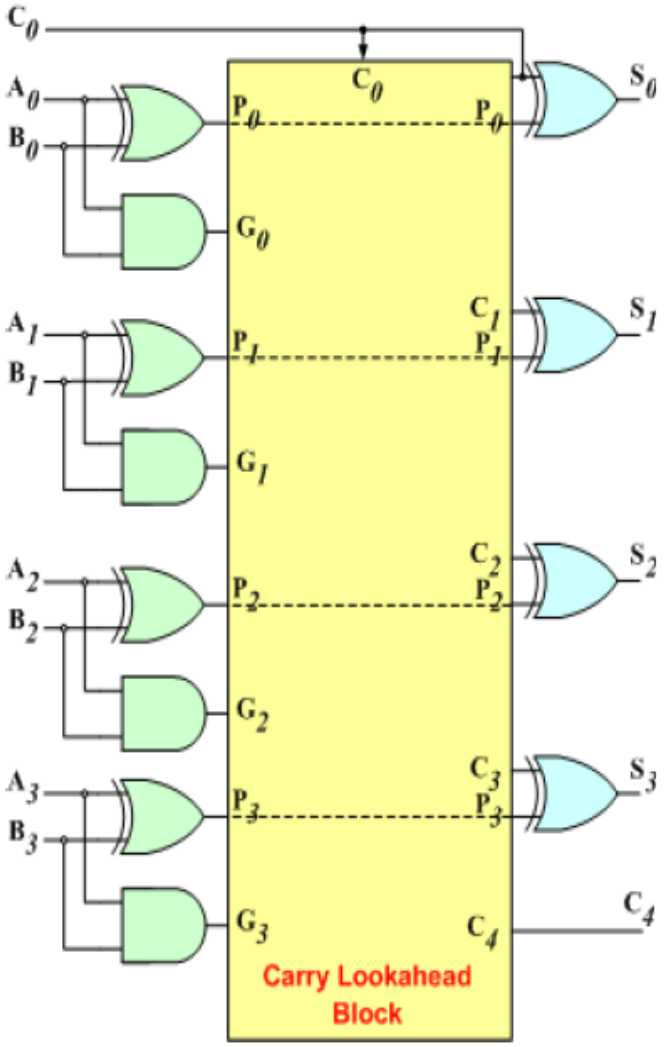


- In the ripple carry adder:
  - G_i, P_i, C_i and S_i are obtained locally at each bit cell of the adder (i.e. limited to that bit)
- In the carry lookahead adder, in order to reduce the length of the ripple carry chain, C_i is changed to a more global function spanning multiple cells

CLA Block*

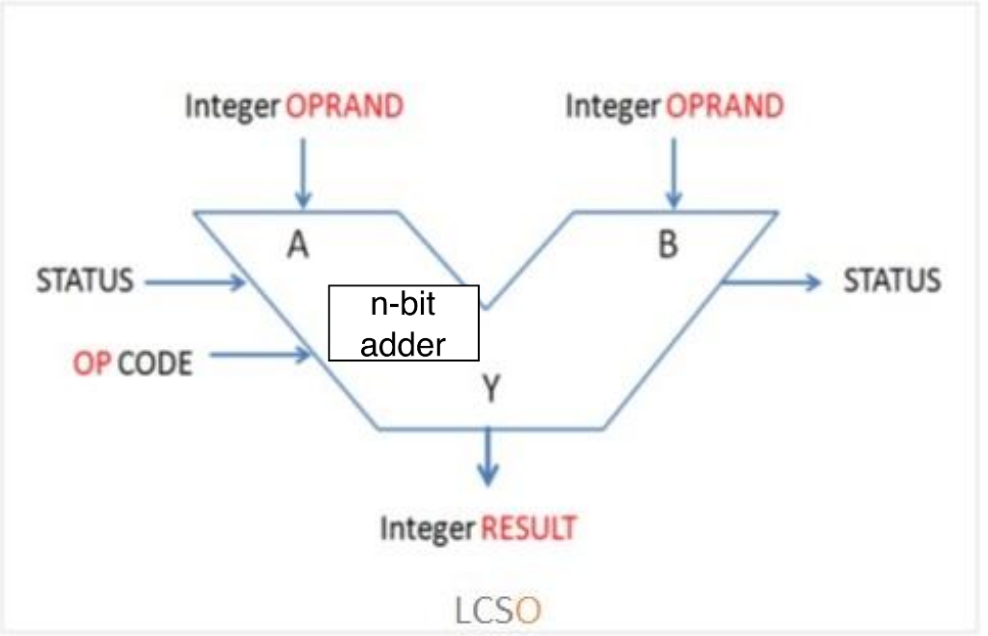
- The 4-bit carry look-ahead (CLA) adder consists of 3 levels of logic:
  - **First level:** Generates all the P & G signals. Four sets of P & G logic (each consists of an XOR gate and an AND gate)
  - **Second level:** The Carry Look-Ahead (CLA) logic block which consists of four 2-level implementation logic circuits. It generates the carry signals (C1, C2, C3, and C4)
  - **Third level:** Four XOR gates which generate the sum signals (Si) (Si = Pi ⊕ Ci), (S0, S1, S2, and S3)
- The delay of CLA is 8 gates
  (down from 12 gates)
- How?

  Beyond the scope of this course



Back to ALU

- We implemented a circuit to add two n-bit numbers in the ALU
- How can we construct a a circuit to subtract two 4-bit numbers using 4-bit Ripple Carry Adder?



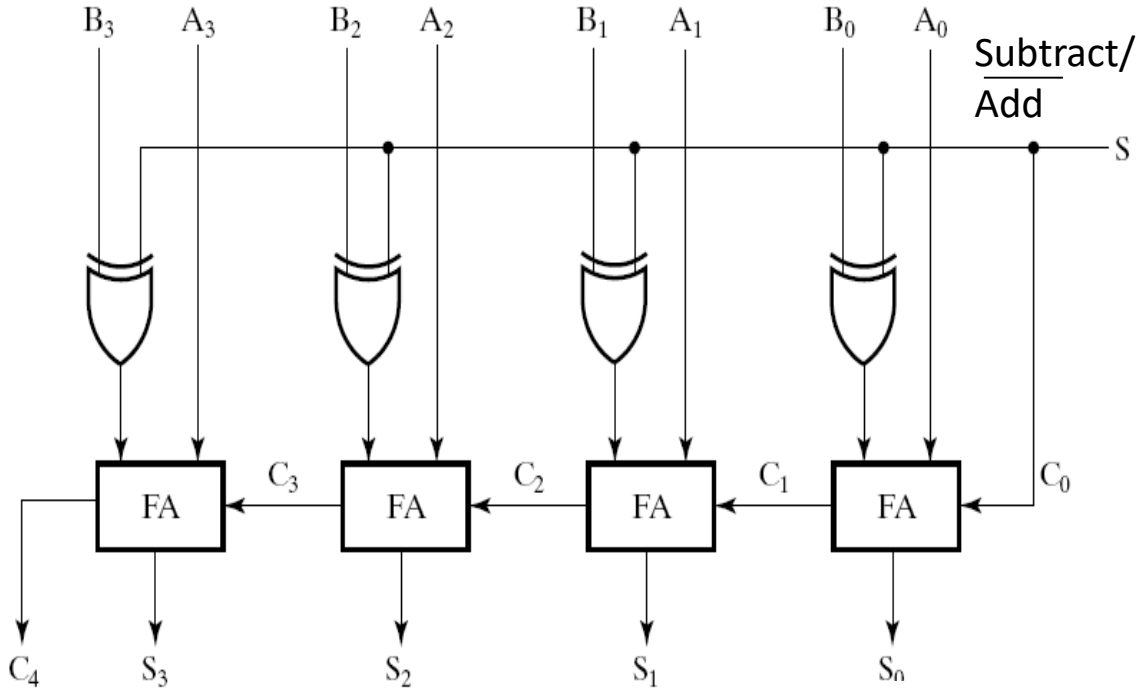
Adder/Subtractor for Signed 2's Complement

- One circuit computes A + B or A – B:
- Subtraction is done by the addition of the 2's Complement
- For S = 0 (add):

  B is passed through to the adder unchanged
- For S = 1 (subtract):

  → 2's complement of B is obtained using XORs to form the 1's comp + 1 applied to C_0 of

  1st stage→ 2's comp. added to A



No correction Needed

One’s Complement Representation*

- Positive numbers are represented using normal binary equivalent .
- Negative numbers are represented by the 1's complement (complement) of the normal binary representation of the magnitude.
- Example:
  - +9 is represented as 01001
  - -9 is represented as 10110 (obtained by complementing the binary representation of 9).

8 bit ones' complement

Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	+0	0
00000001	1	1
...	...	...
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−127	128
10000001	−126	129
10000010	−125	130
...	...	...
11111101	−2	253
11111110	−1	254
11111111	−0	255

Why is it called “one’s complement?”*

- Complementing a single bit is equivalent to subtracting it from 1.

  0’ = 1, and 1 - 0 = 1 1’ = 0, and 1 - 1 = 0

- Similarly, complementing each bit of an n-bit number is equivalent to subtracting that number from 2^n-1.
- For example, we can negate the 5-bit number 01101.
  - Here n=5, and 2^n-1 = 31_{10} = 11111_2.
  - Subtracting 01101 from 11111 yields 10010:

$$\begin{array}{r} 1\ 1\ 1\ 1\ 1 \\ -\ 0\ 1\ 1\ 0\ 1 \\ \hline 1\ 0\ 0\ 1\ 0 \end{array}$$

One’s Complement Addition*

- Example 1: Adding 0111 (+7) + 0111 (+7)
  - The result is 1110 and the carry out is 0, hence an overflow has occurred (adding two positive numbers resulted in a negative number, -1 in this case).
- Example 2: Subtracting 0001 (+1) – 0111 (+7)
  - 0001 (+1) + 1000 (-7) = 1001 (-6)
  - The result is 1001 with no end carry. The result represents -6 (the correct result) and no further addition is required.
- Example 3: Subtracting 0111 (+7) - 0001 (+1)
  - 0111 (+7) + 1110 (-1) = 0101 (+5) with carry out=1
  - The result is 0101 with an end carry of 1. This carry has to be added to the previous result and yields 0110 (+6), the correct answer).

Two’s Complement Representation*

- Positive numbers are represented using normal binary equivalent.
- Negative numbers are represented by the 2's complement of the normal binary representation of the magnitude.
- The 2's complement of a binary number equals its 1's complement + 1.
- Another easy way to obtain the 2's complement of a binary number:
  - Start at the LSB, leaving all the 0s unchanged, look for the first occurrence of a 1. Leave this 1 unchanged and complement all the bits after it.

8 bit two's complement

Binary value	Two's complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
...	...	...
01111110	126	126
01111111	127	127
10000000	−128	128
10000001	−127	129
10000010	−126	130
...	...	...
11111110	−2	254
11111111	−1	255

Computing the 2's Complement*

starting value	00100100_2 = +36
step1: Invert the bits (1's complement)	11011011_2
step 2: Add 1 to the value from step 1	+ 1_2
sum = 2's complement representation	11011100_2 = -36

2's complement of 11011100_2 (-36) = 00100011_2 + 1 = 00100100_2 = +36

The 2's complement of *A* is the **negative of *A***

Another way to obtain the 2's complement:

Start at the least significant 1

Leave all the 0s to its right unchanged

Complement all the bits to its left

Binary Value

= 00100**1**00 (least significant 1)

2's Complement

= 11011**1**00

Two’s Complement Representation*

- To find the value of a number represented in two’s complement, check the sign bit
  - If the sign bit is 0, the number is positive and find its value using weighted method
  - If the sign is 1, the number is negative and you can find value in two ways
    - Take the 2’s complement, you will get the corresponding positive value, then add a minus sign to it
- Example: Find the value of the 4-bit number 1011
  - Take the 2’s complement-> 0101 = +5; thus 1011 represents -5

Two's Complement Addition*

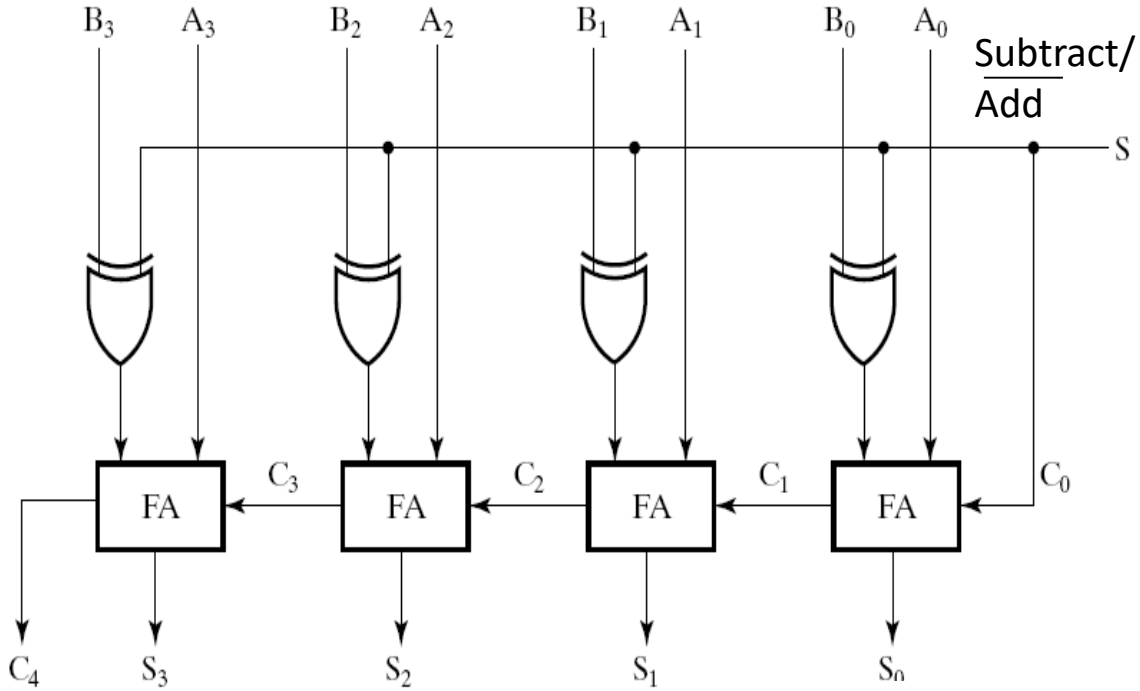
- Example 1: Adding 0111 (+7) + 0111 (+7)
  - The result is 1110 (-2), hence an overflow has occurred (adding two positive numbers resulted in a negative number).
- Example 2: Subtracting 0001 (+1) - 0111 (+7)
  - 0001 (+1) - 0111 (+7) = 0001 (+1) + 1001 (-7) = 1010 (-6)
  - The result represents -6 in 2's complement (the correct result) and no further operation is required.
- Example 3: Subtracting 0111 (+7) – 0001 (+1)
  - 0111 (+7) – 0001 (+1) = 0111 (+7) + 1111 (-1) = 0110
  - The result is 0110 (+6, the correct result). Since the last two carries being 1 no overflow has occurred and no further operations are required.

Adder/Subtractor for Signed 2's Complement -- Revisited

- One circuit computes A + B or A – B:
- Subtraction is done by the addition of the 2's Complement
- For S = 0 (add):

  B is passed through to the adder unchanged
- For S = 1 (subtract):

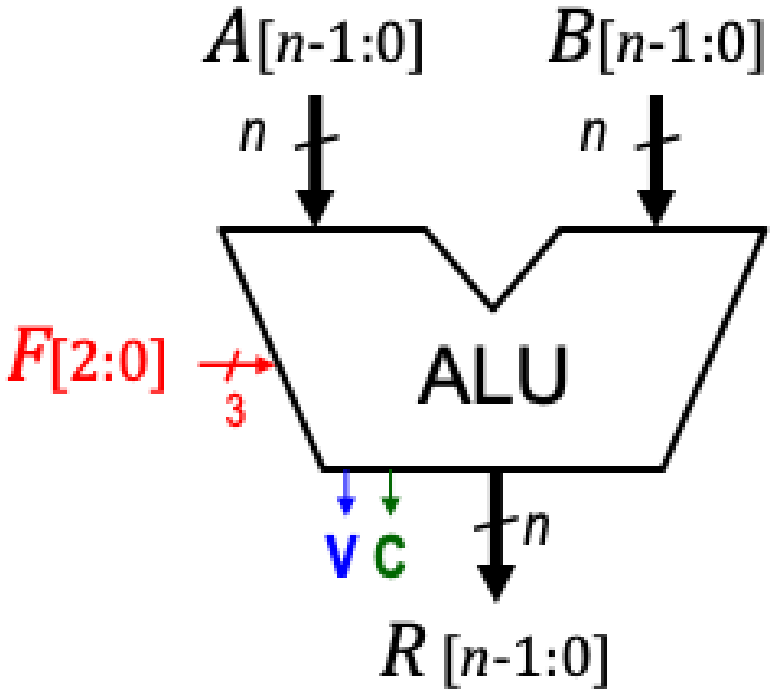
  → 2's complement of B is obtained using XORs to form the 1's comp + 1 applied to C_0 of 1st stage → 2's comp. added to A



No correction Needed

Arithmetic and Logic Unit (ALU)

- ALU performs arithmetic and logic functions

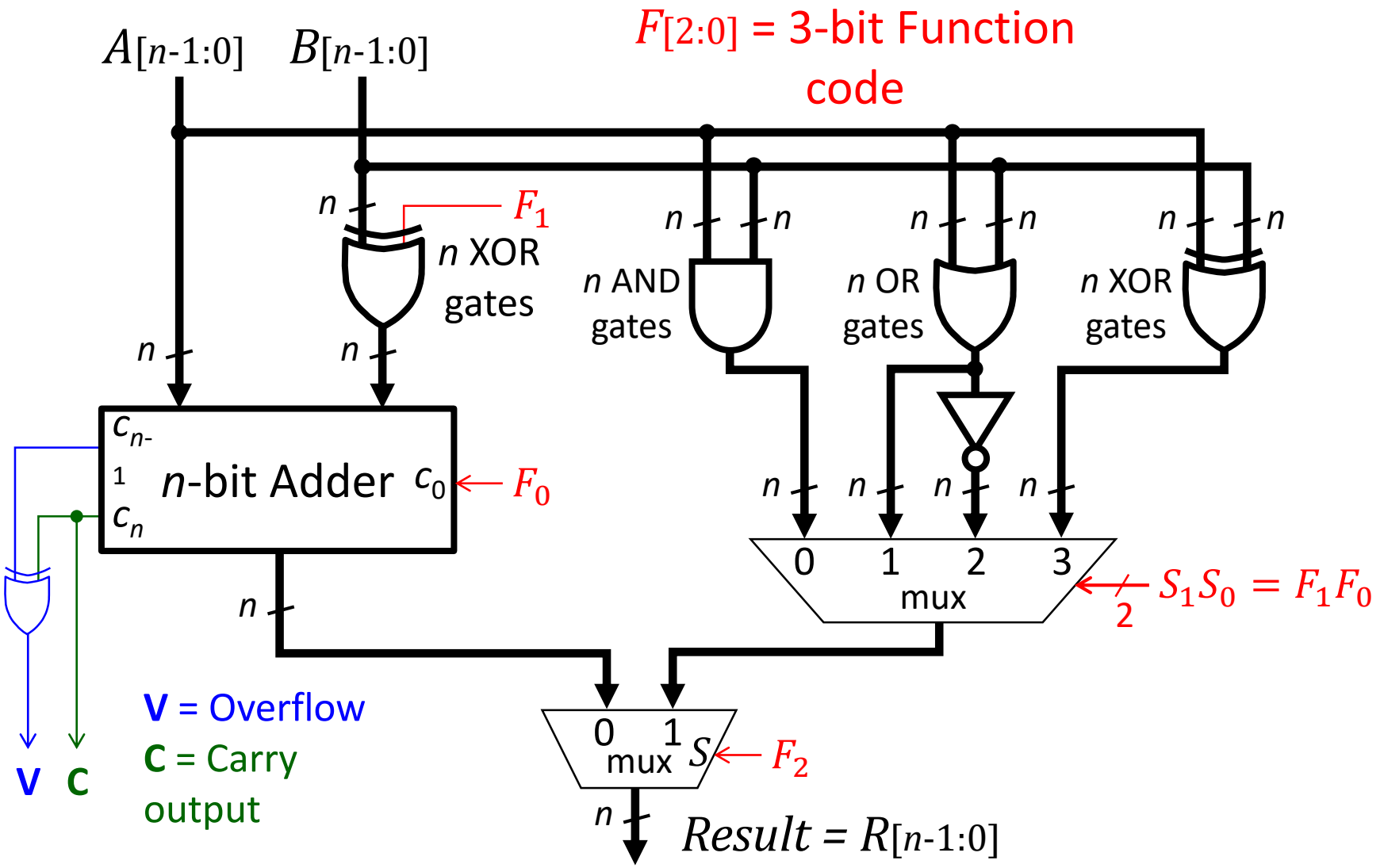


- We will design an ALU with 8 functions
- The function F is coded with 3 bits as follows:

OP CODE	ALU Result	Function	ALU Result
F = 000 (ADD)	R = A + B	F = 100 (AND)	R = A & B
F = 001 (ADD + 1)	R = A + B + 1	F = 101 (OR)	R = A | B
F = 010 (SUB – 1)	R = A – B – 1	F = 110 (NOR)	R = ~(A | B)
F = 011 (SUB)	R = A – B	F = 111 (XOR)	R = (A ^ B)

ALU with Addition, Subtraction, AND, OR< XOR

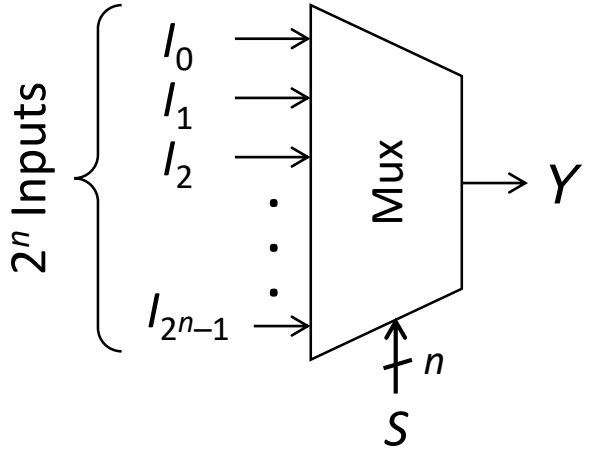
Function	ALU Result	Function	ALU Result
F = 000 (ADD)	R = A + B	F = 100 (AND)	R = A & B
F = 001 (ADD + 1)	R = A + B + 1	F = 101 (OR)	R = A | B
F = 010 (SUB – 1)	R = A – B – 1	F = 110 (NOR)	R = ~(A | B)
F = 011 (SUB)	R = A – B	F = 111 (XOR)	R = (A ^ B)



Multiplexers: 2^n-to-1

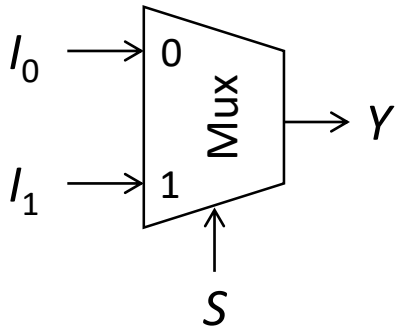
- A multiplexer (MUX) selects information from one of 2^n input line and directs it toward a single output line.
- A typical multiplexer has:
  - 2^n information inputs ($I_{(2^n - 1)}$, … I_0) (to select from)
  - 1 Output Y (to select to)
  - *n* select control (address) inputs (S_{n-1}, … S_0) (to select with)

MUX selection circuits can be duplicated m times (with the same selection controls in parallel) to provide m-wide data widths



2-to-1 MUX

- The single selection variable S has two values:
  - S = 0 selects input I_0
  - S = 1 selects input I_1
- 3-input K-map optimization gives the output equation:



S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

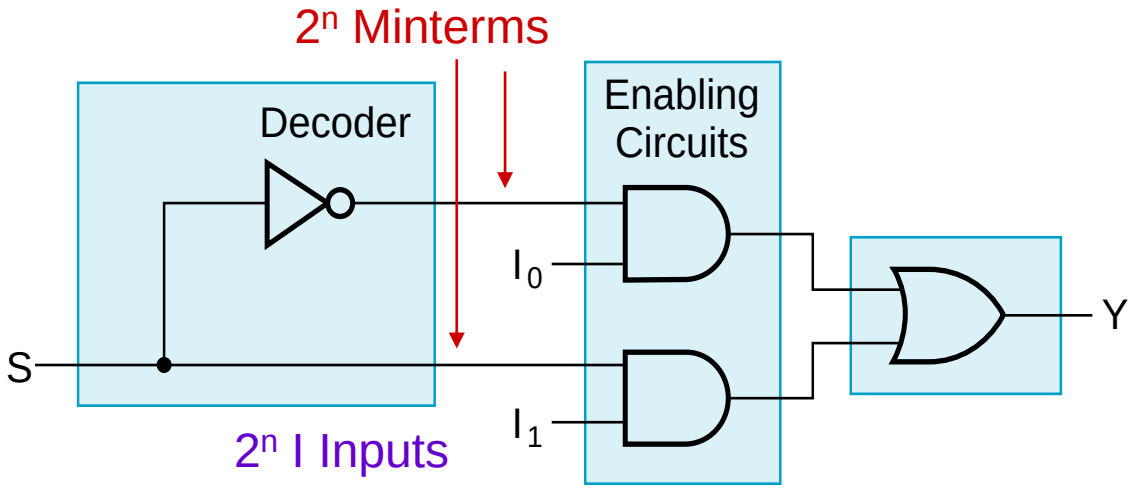
- The circuit:
- Can be seen As:

  1-to-2 decoder

  + Enabling

  + Selection

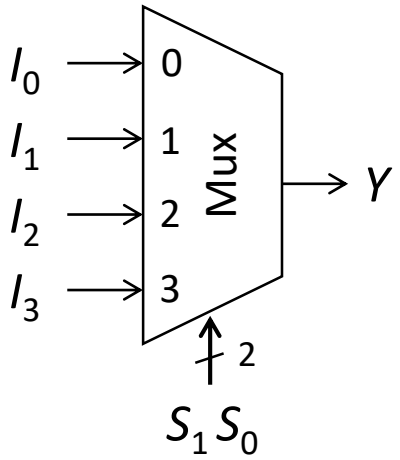
$$Y = \overline{S}I_0 + SI_1$$



4-to-1 MUX

- 4-to-1 Multiplexer

if (S_1S_0 == 00) $Y = I_0$;
else if (S_1S_0 == 01) $Y = I_1$;
else if (S_1S_0 == 10) $Y = I_2$;
else $Y = I_3$;



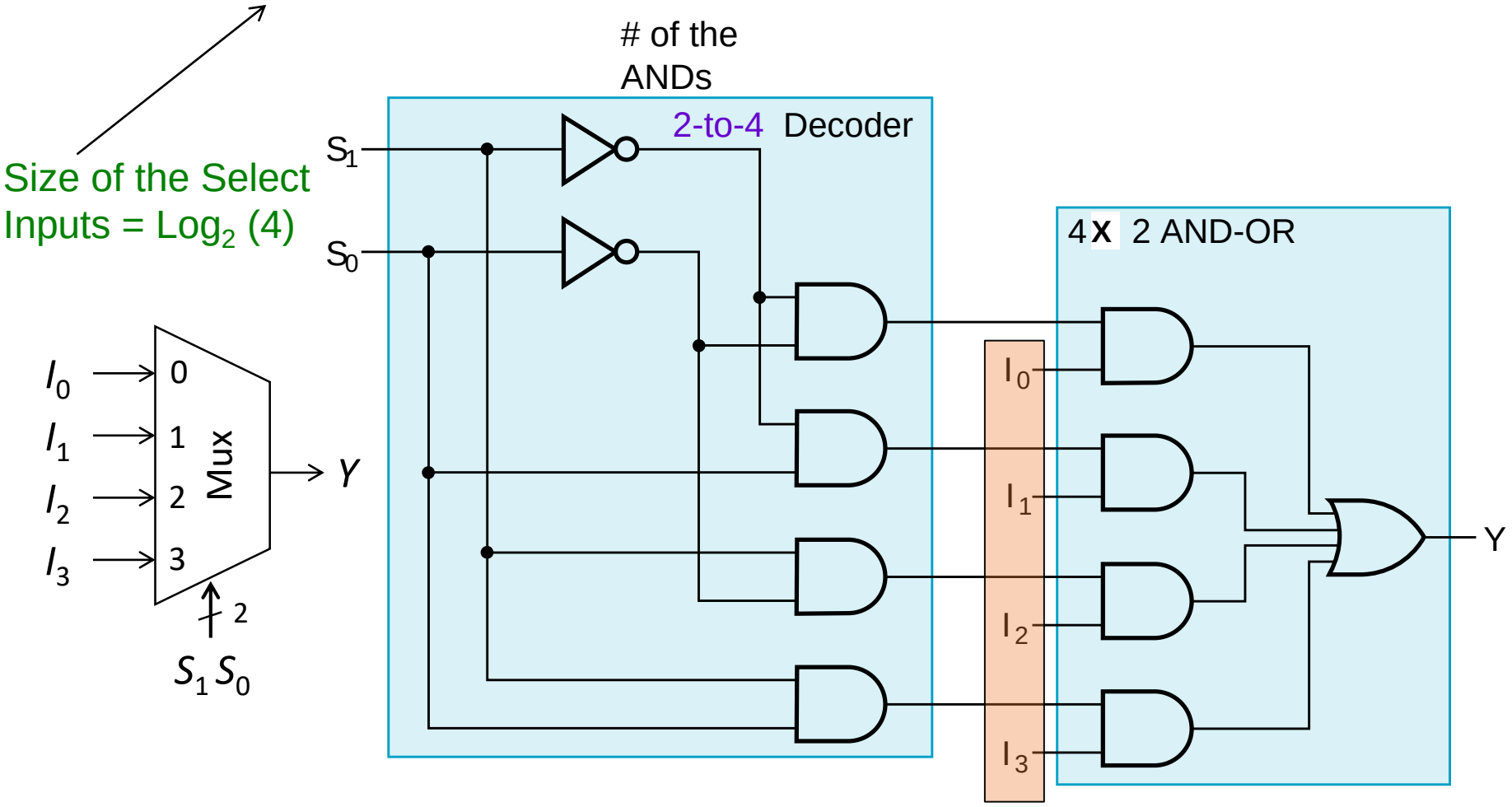
Inputs						Output
S_1	S_0	I_0	I_1	I_2	I_3	Y
0	0	0	X	X	X	$0 = I_0$
0	0	1	X	X	X	$1 = I_0$
0	1	X	0	X	X	$0 = I_1$
0	1	X	1	X	X	$1 = I_1$
1	0	X	X	0	X	$0 = I_2$
1	0	X	X	1	X	$1 = I_2$
1	1	X	X	X	0	$0 = I_3$
1	1	X	X	X	1	$1 = I_3$

Logic expression:

$$Y = I_0\, S_1' S_0' + I_1\, S_1' S_0 + I_2\, S_1 S_0' + I_3\, S_1 S_2$$

4-to-1 MUX

- Using 2-to-4 decoder +4 2-input AND + 4-input OR for Enabling/Selection

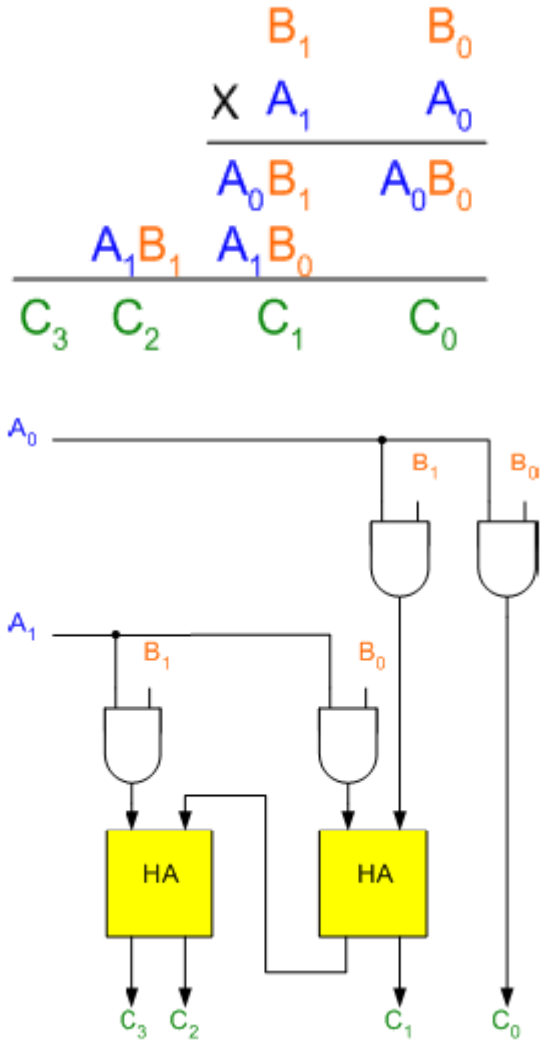


How About Multiplication?*

- Multiplication of binary numbers is performed in the same way as with decimal numbers
- The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit
- The result of each such multiplication forms a partial product. Successive partial products are shifted one bit to the left
- The product is obtained by adding these shifted partial products

Binary Multiplier*

- Example 1: Consider an example of multiplication of two numbers, say A and B (2 bits each), C = A x B.
- The first partial product is formed by multiplying the B_1B_0 by A_0. The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.
- The second partial product is formed by multiplying the B_1B_0 by A_1 and is shifted one position to the left.

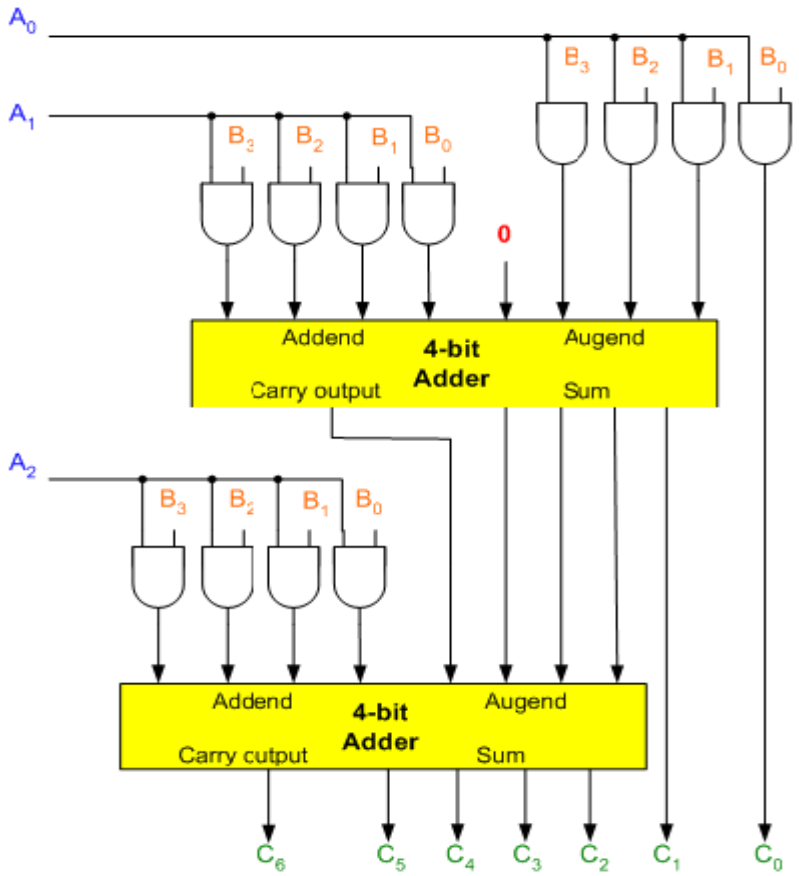


Binary Multiplier*

- Example 2: Consider the example of multiplying two numbers, say A (3-bit number) and B (4-bit number).

$$
\begin{array}{rcccccc}
 & & & B_3 & B_2 & B_1 & B_0 \\
 & & & \times & A_2 & A_1 & A_0 \\
\hline
 & & A_0B_3 & A_0B_2 & A_0B_1 & A_0B_0 & \\
 & A_1B_3 & A_1B_2 & A_1B_1 & A_1B_0 & & \\
A_2B_3 & A_2B_2 & A_2B_1 & A_2B_0 & & & \\
\hline
C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0
\end{array}
$$

Since J = 3 and K = 4, 12 (J x K) AND gates and two 4-bit ((J - 1) K-bit) adders are needed to produce a product of seven (J + K) bits.



Full-fledge ALU

- We can implement a full-fledge ALU that can perform ALL arithmetic and logical operations
- Each operation will have a unique OP CODE

  Addition = ADD

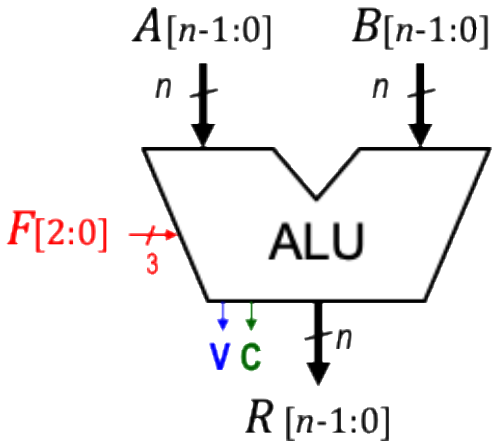
  Subtraction = SUM

  Multiplication = MUL

  Division = DIV

  AND = AND

  So on
- The value of OP CODE depends on the implementation of the ALU



OP CODE	ALU Result	Function	ALU Result
F = 0000 (ADD)	R = A + B	F = 1000 (AND)	R = A & B
F = 0001 (ADD + 1)	R = A + B + 1	F = 1001 (OR)	R = A | B
F = 0010 (SUB – 1)	R = A – B – 1	F = 1010 (NAND)	R = ~(A & B)
F = 0011 (SUB)	R = A – B	F = 1011 (NOR)	R = ~(A | B)
F=0100 (MUL)	R = A * B	F = 1100 (XOR)	R = A ⊕ B
F=0101 (POW)	$R = A^B$	F = 1101 (XNOR)	R = ~(A ⊕ B)
F=0110 (DIV)	R = A / B	F = 1110 (SHR)	R >> 1
F=0111 (SQRT)	$R = A^{1/B}$	F = 1111 (SHL)	R << 1

Can We Construct a Quantum ALU?

- Yes!
- The key is to find an equivalency of each digital gate (NOT, AND, OR)

- Whether it is useful or not, we will debate this later.