Computer Architecture

Trần Trọng Hiếu

Information Systems Department

Faculty of Information Technology

VNU - UET

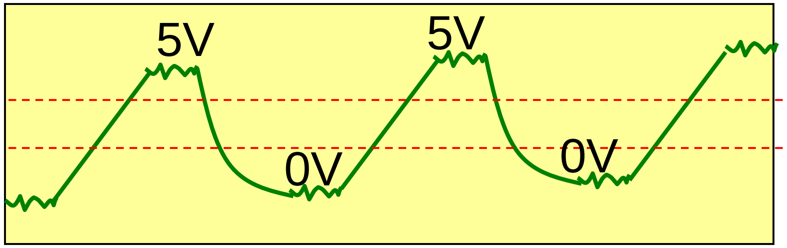
hieutt@vnu.edu.vn

Fundamentals

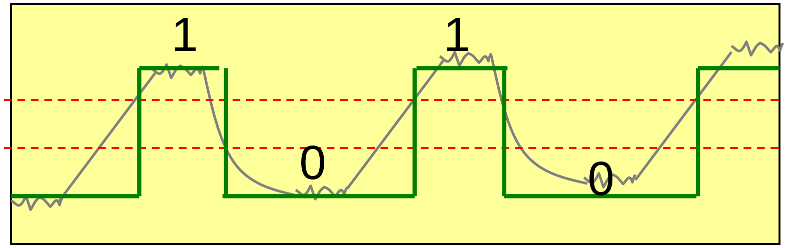
Boolean Algebra

Digital Representation

- Digital is an abstraction of analog voltage
  - Voltage is a continuous, physical unit
    - Typically ranging from 0 to 5 volts on PCs
  - Digital logic abstracts it to 2 distinct levels
    - “1” or positive (typically 2.5 V or greater)
    - “0” or negative (typically less than 1 volt)
    - Eases design and manufacturing



Analog Voltage Waveform



Digital Voltage Waveform

Digital Processing

- Combine “1”s and “0”s in different ways
  - To generate more “1”s and “0”s
    - This is finally what a computer really does
- Need a well defined mechanism
    - Ease design & development of circuits
  - **Boolean Algebra**
    - Mathematical framework for processing “1”s & “0”s
    - Based on simple, scalable primitive operations
      - Easy to realize using basic hardware components
    - Scales to reason about complex operations
    - Leads to information processing
      - When combined with suitable interpretations

Axioms of Boolean Algebra

- Two Boolean constants
  - “1” or “true”
  - “0” or “false”
- Boolean variables
  - An unknown Boolean value
    - Can be “1” or “0” (but not both at the same time)
  - Represented using symbols (or alphabets)
    - Examples: “X”, “A”, “B”, “α”, “β”
- 3 primary Operators
  - NOT (unary operator)
  - AND (Binary operator)
  - OR (Binary operator)

Truth Table

- Tabulates results of operators
  - Involves *n* variables
  - Consists of 2^n rows
    - Each row has a unique combination of “1” and “0” for the *n* variables
  - Used to define result of primary operators
    - NOT, AND, & OR

NOT Operator

- NOT operator inverts the value of a variable
  - Given a Boolean variable A
  - NOT operation is represented as $\overline{A}$
- NOT is described by the following truth table:

A	$\overline{A}$
0	1
1	0

AND Operator

- AND operator
  - Binary operator: Uses 2 operands
  - Result is a "1" only if both operands are "1"
  - AND operation is represented as AB or A•B
    - Where A and B are two Boolean variables
- AND is described by the following truth table:

A	B	A•B
0	0	0
0	1	0
1	0	0
1	1	1

OR Operator

- OR operator
  - Binary operator: Uses 2 operands
  - Result is a “1” if any one of the operand is a “1”
  - OR operation is represented as A+B
    - Where A and B are two Boolean variables
- OR is described by the following truth table:

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Exercises

- If A=1 and B=0, What is
  - A+B =
  - AB =
  - AA =
  - B+B =
  - $\overline{A}$ =

Boolean Expression

- Combination of operands & operators
  - Examples
    - A+A_
    - (A+B)•1
    - (A•0)+(B•0)
    - A•1+B•1
    - What are the results of the above expressions if A=1, B=0
  - Operator precedence
    - Inner most parentheses
    - NOT
    - AND
    - OR

Boolean Equations & Truth Tables

- Illustrate Truth Table for $A+\overline{B}\bullet C$

A	B	C	$\overline{B}\bullet C$	$A+\overline{B}\bullet C$
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Laws of Boolean Algebra

- Identity law
  - $A + 0 = A$
  - $A \cdot 1 = A$
  - $\overline{\overline{A}} = A$
- Zero and One laws
  - $A + 1 = 1$
  - $A \cdot 0 = 0$
- Inverse laws
  - $A + \overline{A} = 1$
  - $A \cdot \overline{A} = 0$
- Idempotent law: $AA=A$ $A+A=A$

Laws of Boolean Algebra (Contd.)

- Given Boolean variables A, B, & C
  - Commutative laws
    - $A + B = B + A$
    - $A \cdot B = B \cdot A$
  - Associative laws
    - $A + (B + C) = (A + B) + C$
    - $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
  - Distributive laws
    - $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
    - $A+(B \cdot C) = (A+B) \cdot (A+C)$
  - DeMorgan's laws
    - $\overline{(A \cdot B)} = \overline{A} + \overline{B}$
    - $\overline{(A+B)} = \overline{A} \cdot \overline{B}$

Verification of Laws (1)

- Using Truth Tables
  - Identity Law
    - A + 0 = A

A	A + 0
0	0
1	1

Verification of Laws (2)

- Distributive law
  - A • (B + C) = (A • B) + (A • C)

A	B	C	A•(B+C)	(A•B)+(A•C)
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

More Boolean Operators

- Other commonly used Boolean Operators
  - Convenient when implementing logic operations using electronic components
- NAND
  - $(\overline{A \bullet B})$
- NOR
  - $(\overline{A+B})$

English to Logic Conversion

- Straightforward strategy
    - Use common sense
  - Identify independent clauses
    - Look for “and” & “or” clauses in sentences
  - Identify primary inputs
  - Work logic out for each independent clause
  - Connect them back together
  - Optimize the final equation
    - We will not deal with optimizations in this course
  - Verify using Truth Table

Example 1

- Output is 1 only if the two primary inputs are zero
  - Let the primary inputs be A and B
  - Output is 1 in the following cases
    - A=0 & B=0 => $\overline{A} \cdot \overline{B}$
  - The above equation can be rewritten as:
    - $\overline{(A+B)}$

Example 2

- Output is 1 only if the two primary inputs are different
  - Let the inputs be A and B
  - Output is 1 in the following cases
    - A=0, B=1 => $\overline{A}\bullet B$
    - A=1, B=0 => $A\bullet\overline{B}$
  - Combining the above two cases gives:
    - $(\overline{A}\bullet B)+(A\bullet\overline{B})$
    - This operation is called Exclusive-OR or XOR
      - It is frequently used
      - Represented as $A\oplus B$

Truth Table for XOR

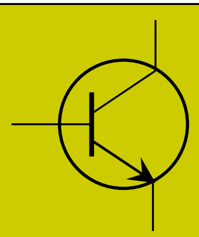
- XOR is a frequently used operation
  - It is important to remember its operation

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

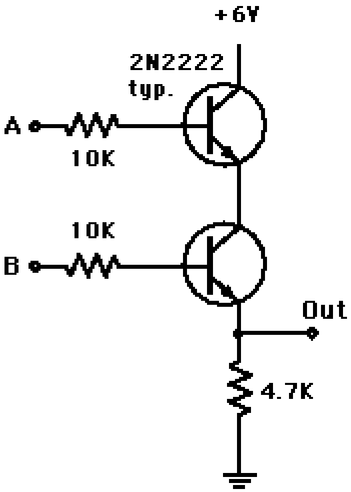
Digital Logic Circuits

Basic transitor gates

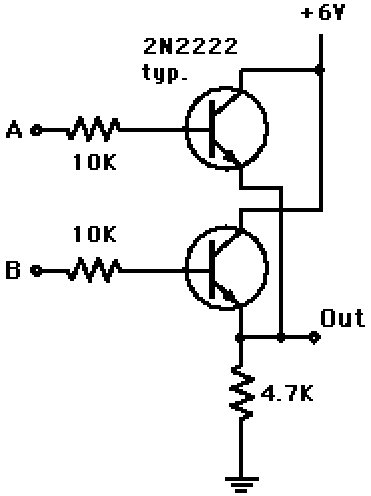
- Interconnected set of Transistors called Circuits
  - Transistors are Electronic Switches
    - Turn “On” or “Off”
      - Depending on input voltages
  - Used to implement Boolean expressions



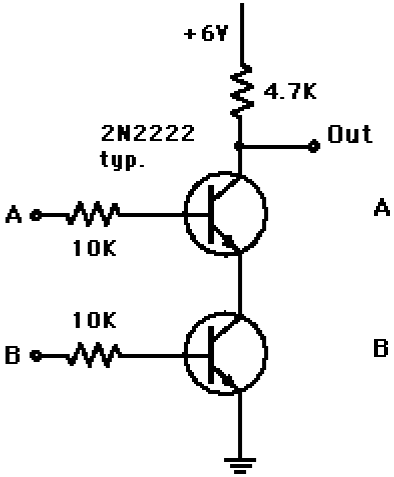
Transistor



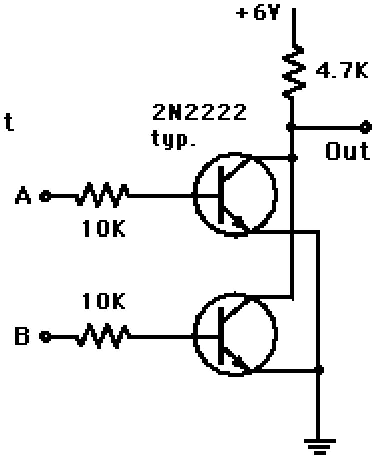
AND



OR



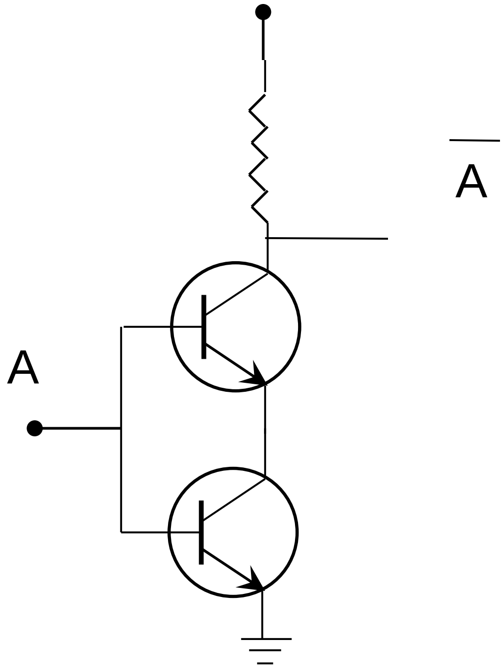
NAND



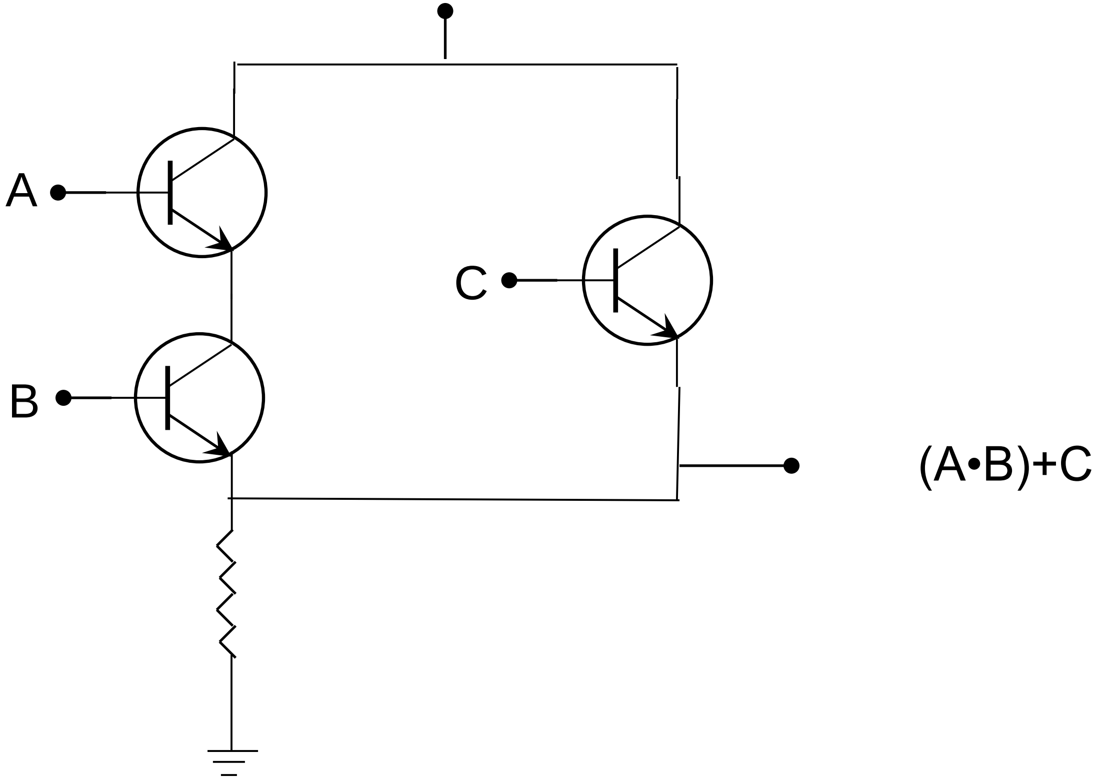
NOR

Question

- How the NOT gate is constructed?
  - A.A=A
  - NOT(A.A)=NOT A
  - A NAND A = NOT A



Expression construction

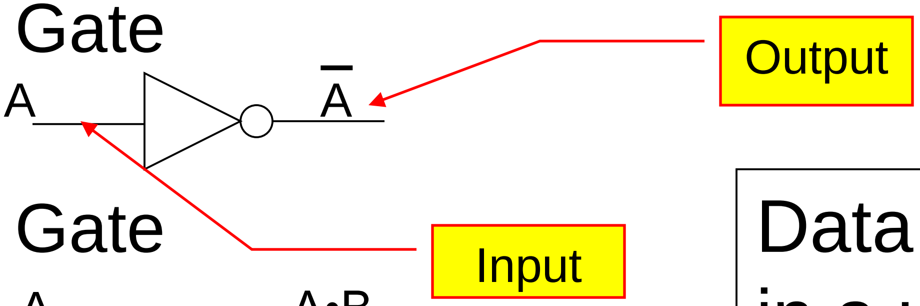


Logic Gates

- Developing large circuits is complex
  - Drawing many transistors is cumbersome
  - Makes the circuit diagram unwieldy
    - Hard to illustrate and comprehend
- Solution: Logic Gates
  - Abstract notation for common logic circuits
    - Functionally similar to set of transistors
  - Simpler to develop and use

Basic Gates

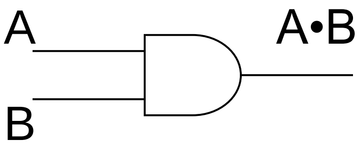
- Corresponding to basic operations in Boolean Algebra
  - NOT Gate



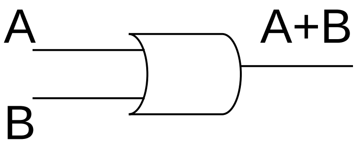
Output

Input

  - AND Gate



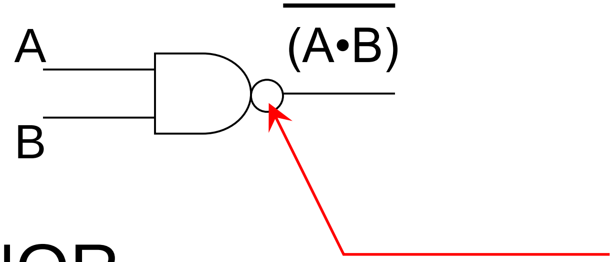
  - OR Gate



Data always flows in a unidirectional manner from *inputs* to *outputs* of the logic gates!

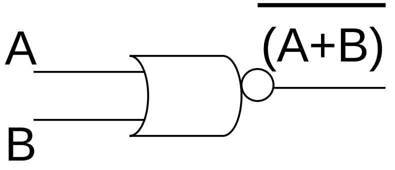
Commonly used Gates

- Other commonly used gates
  - NAND

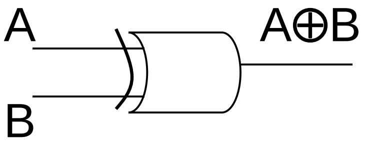


The circle (or bubble) indicates inversion or **NOT** operation. You may add this circle (or bubble) at the output or input of any gate!

  - NOR

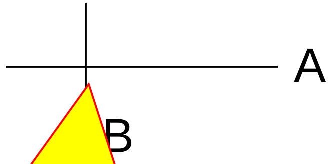


  - XOR

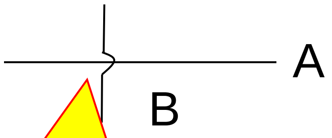


Equations to Circuits

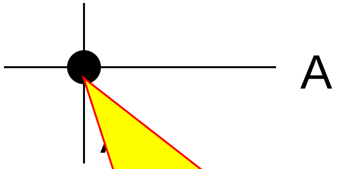
- Convert Boolean equations to Logic Circuits
  - Logic circuits drawn on paper are often also called Schematics
- Straightforward process
  - Convert each operator to a Logic Gate
- Suitably connect inputs and output
  - Pay attention to crossing lines versus connected lines



No relationship between A & B.



No relationship between A & B. This is preferred!



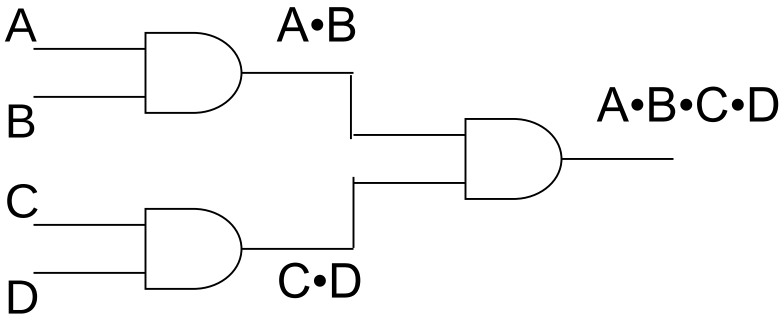
These two lines/wires are the same!

Example 1

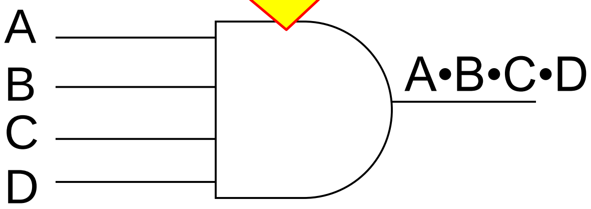
- A•B•C•D
  - (A•B)•(C•D)

There are a few aspects to consider when using the shortcut version:

1. All gates must be the same
2. Input to output transformation must be straightforward



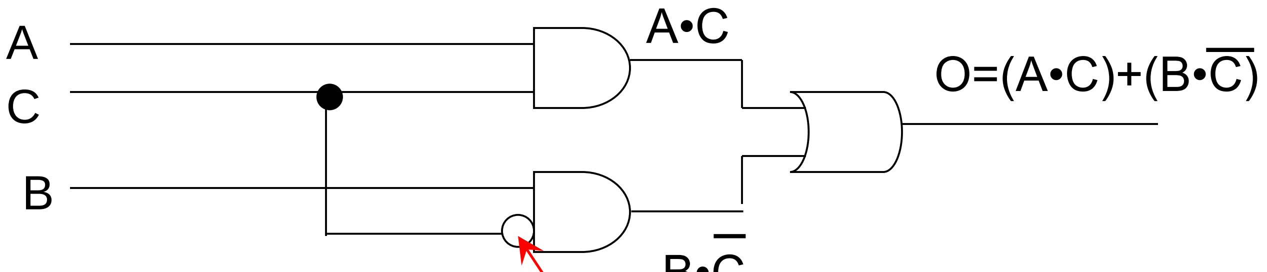
Standard Version (With 2-input Gates)



Shortcut Version (*n*-Input Gates)

Example 2

- Output O is
  - *A*, if *C*=1
  - *B*, if C=0
- Solution
  - $O = (A \bullet C)+(B \bullet \overline{C})$



Shortcut for NOT operation.

General Selection Logic

- Develop a circuit to select 1 of the given 5 inputs
  - Let the inputs be A, B, C, D, & E
  - Assign unique combinations of 1s and 0s to identify each Input
    - Given n inputs you need k bits such that 2^k>=n
    - In this case n=5 and therefore k=3
    - Let selection variables be s_1, s_2, and s_3

S_1	S_2	S_3	O
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E

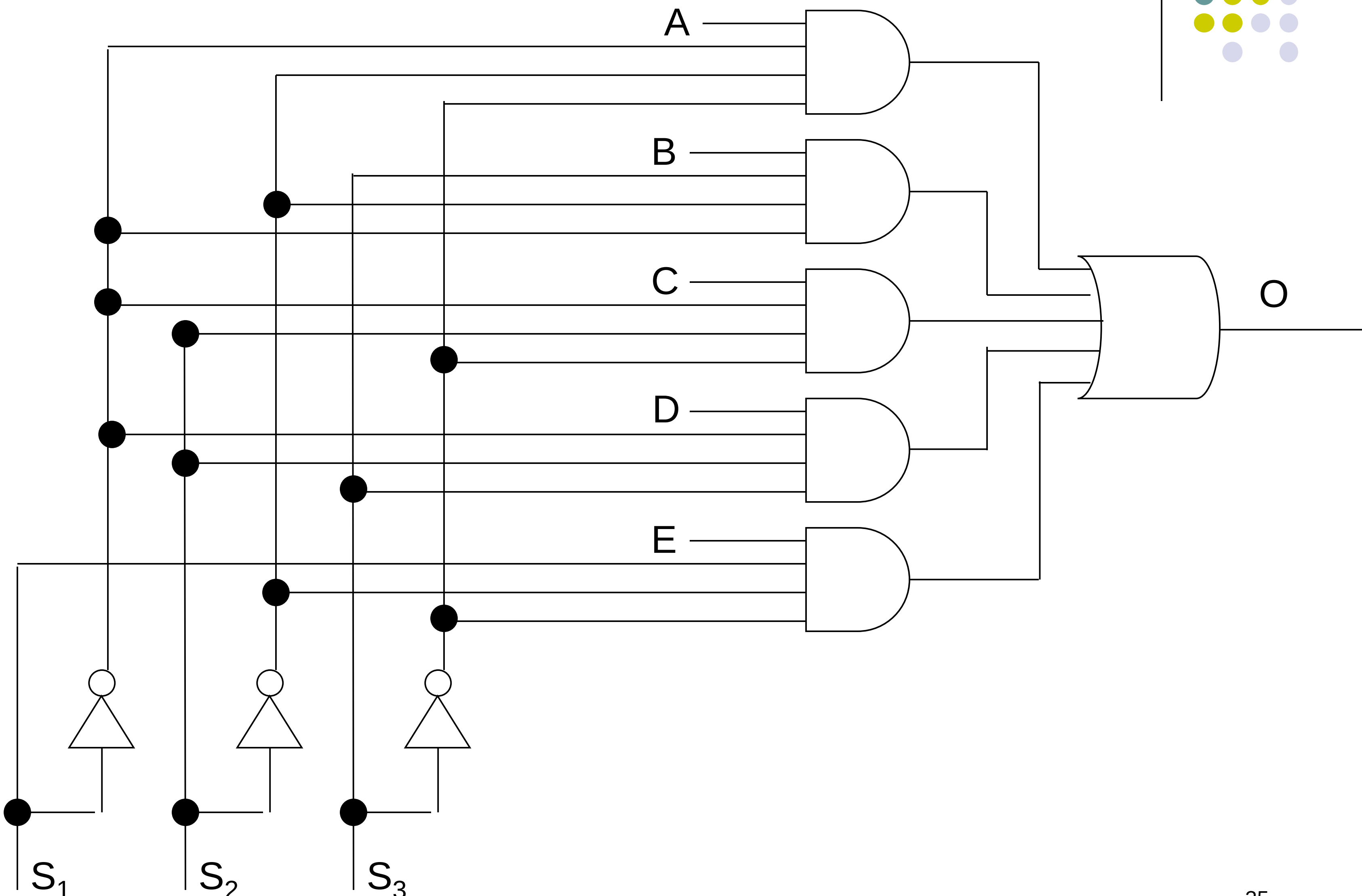
General Selection (Cont.)

- Boolean equation for the example:
  - $O=(A\bullet\overline{S_1}\bullet\overline{S_2}\bullet\overline{S_3})+(B\bullet\overline{S_1}\bullet\overline{S_2}\bullet S_3)+(C\bullet\overline{S_1}\bullet S_2\bullet\overline{S_3})+(D\bullet S_1\bullet\overline{S_2}\bullet S_3)+$
    $(E\bullet S_1\bullet\overline{S_2}\bullet\overline{S_3})$

S_1	S_2	S_3	O
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E

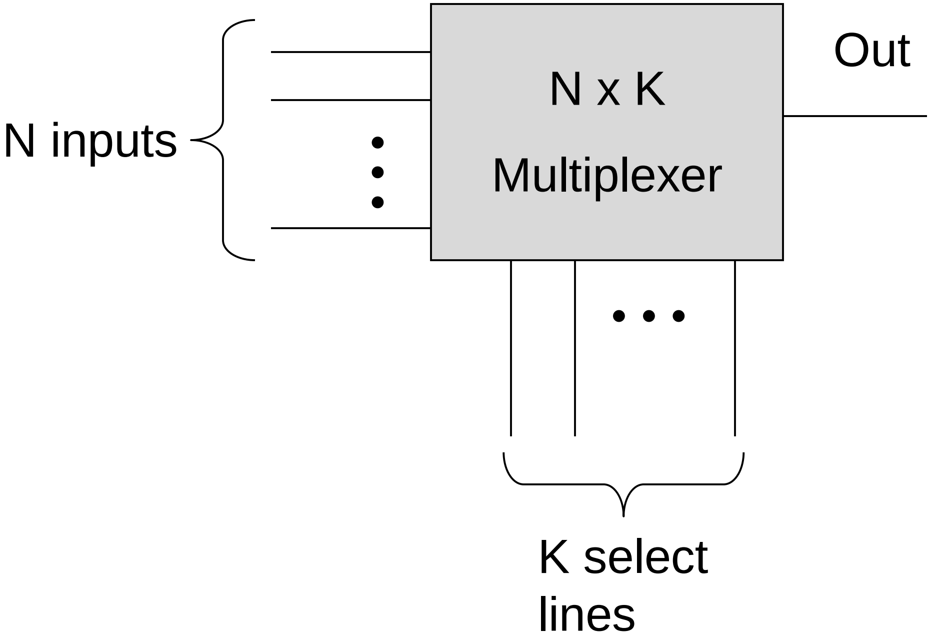
Draw the circuit?

Logic Circuit for Selector



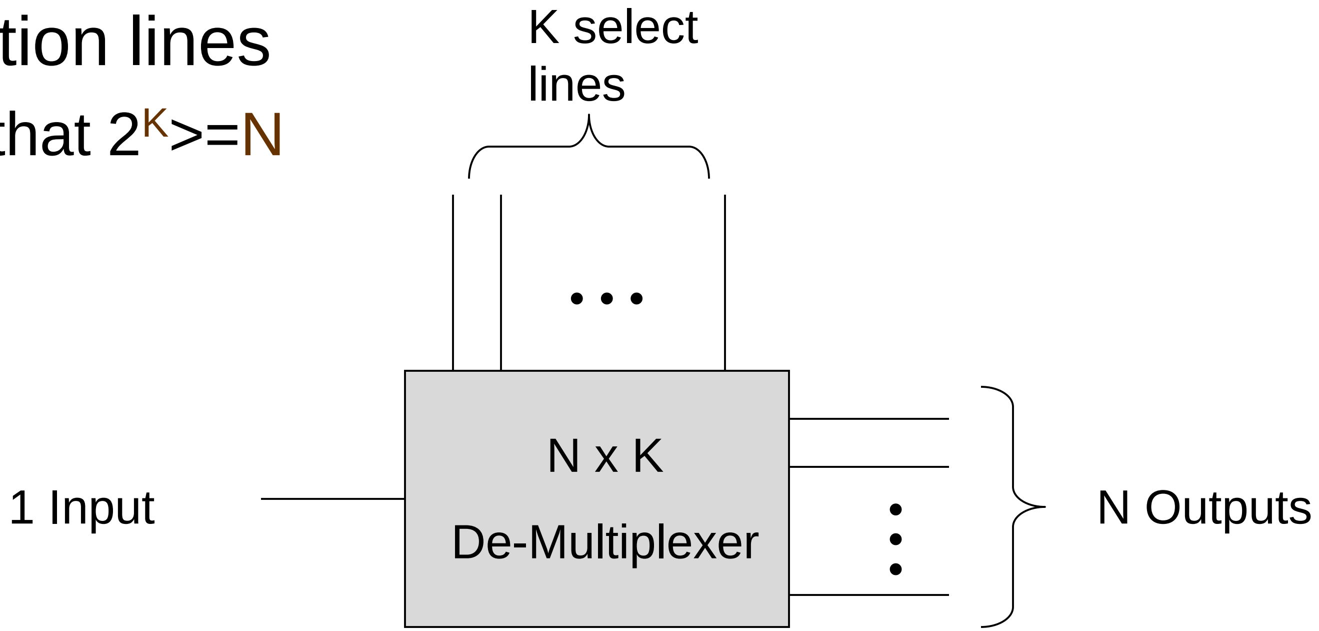
Multiplexer (Mux)

- Select 1 given N circuits are called Multiplexers
  - Have N inputs
  - K selection lines
    - Such that $2^k >= N$
  - 1 output line



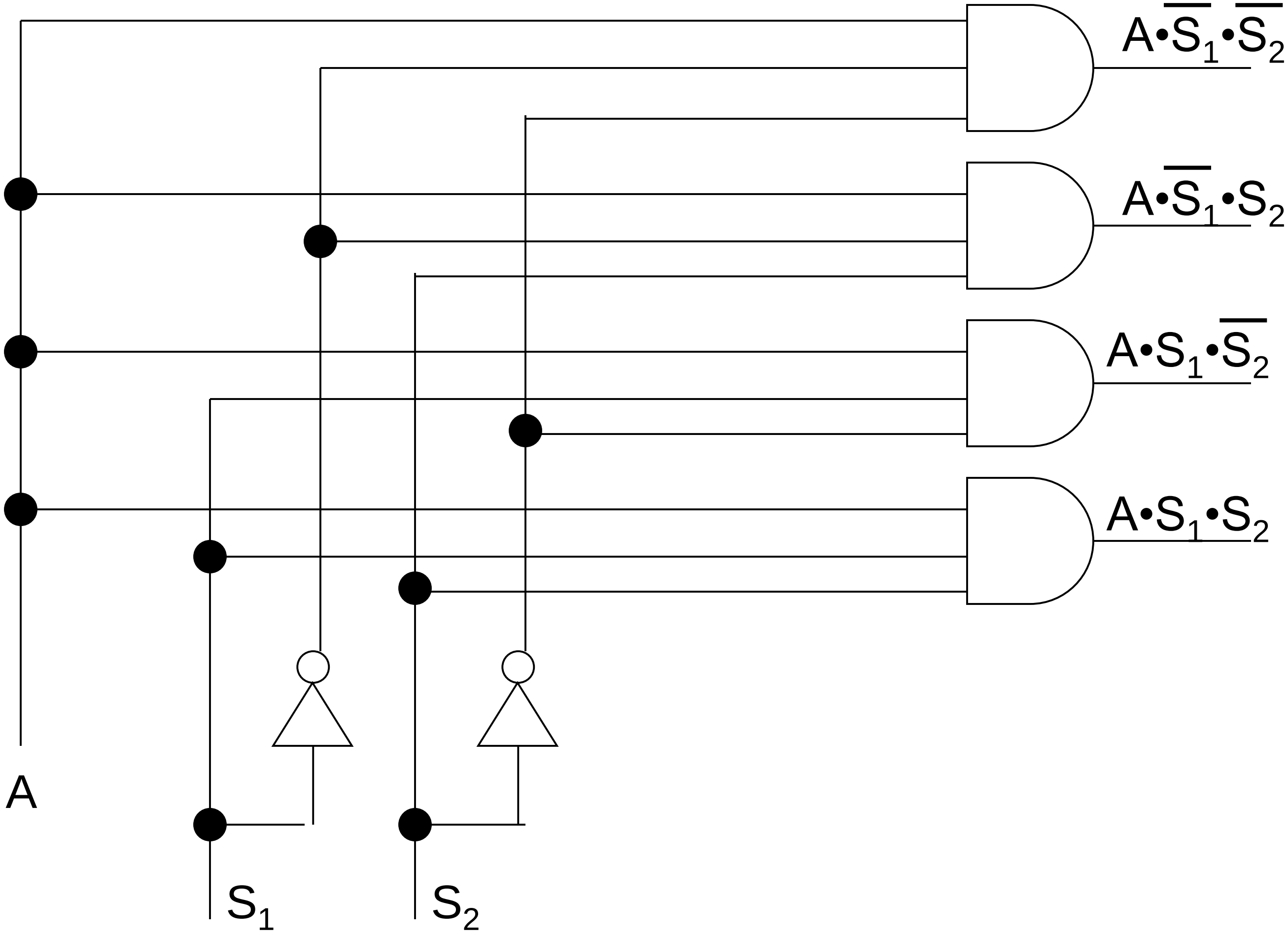
De-Multiplexer (DeMux)

- Move 1 input bit to selected output line
  - 1 Input
  - N Output lines
  - K selection lines
    - Such that $2^K >= N$



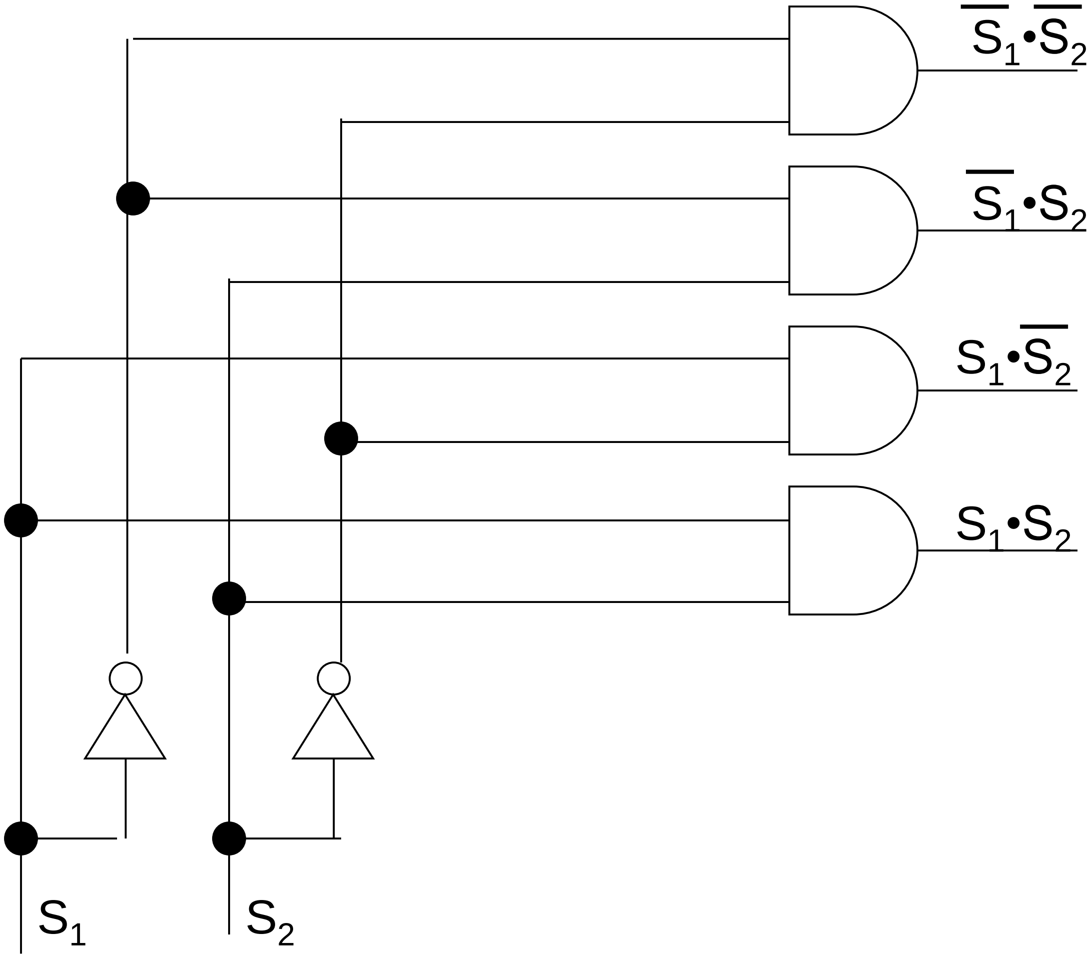
De-Multiplexer Logic Circuit

- 1 X 4 De-Multiplexer


$A \bullet \overline{S_1} \bullet \overline{S_2}$
$A \bullet \overline{S_1} \bullet S_2$
$A \bullet S_1 \bullet \overline{S_2}$
$A \bullet S_1 \bullet S_2$
A
S_1
S_2

Decoder Logic Circuit

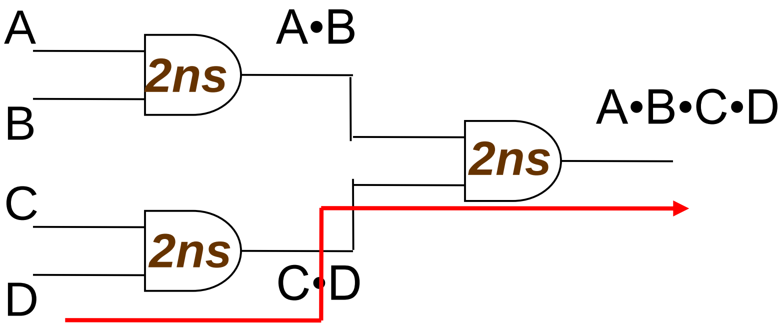
- A sample decoder


$\overline{S_1} \bullet \overline{S_2}$
$\overline{S_1} \bullet S_2$
$S_1 \bullet \overline{S_2}$
$S_1 \bullet S_2$
S_1
S_2

Timing

- Gates take time to work
  - Outputs don’t stabilize for some time
    - Stabilization time is usually in nanoseconds
- Gate delays compound in circuits
  - Final output is not ready until all gates are stable
    - Propagation delay
      - Time taken for changes at the input to propagate to output
  - Typically, the longest path from input to output
    - This is often called the “critical path” in a circuit

Example

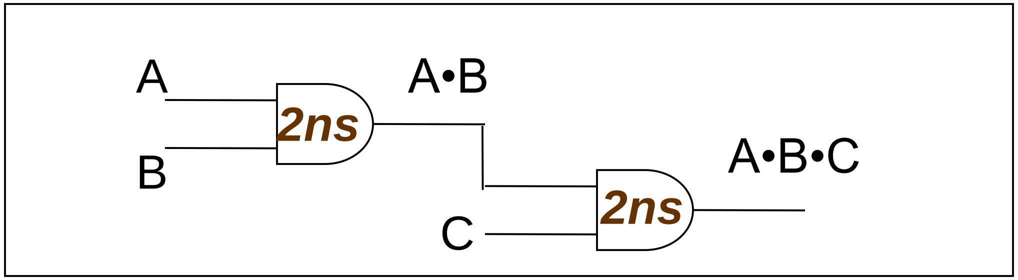


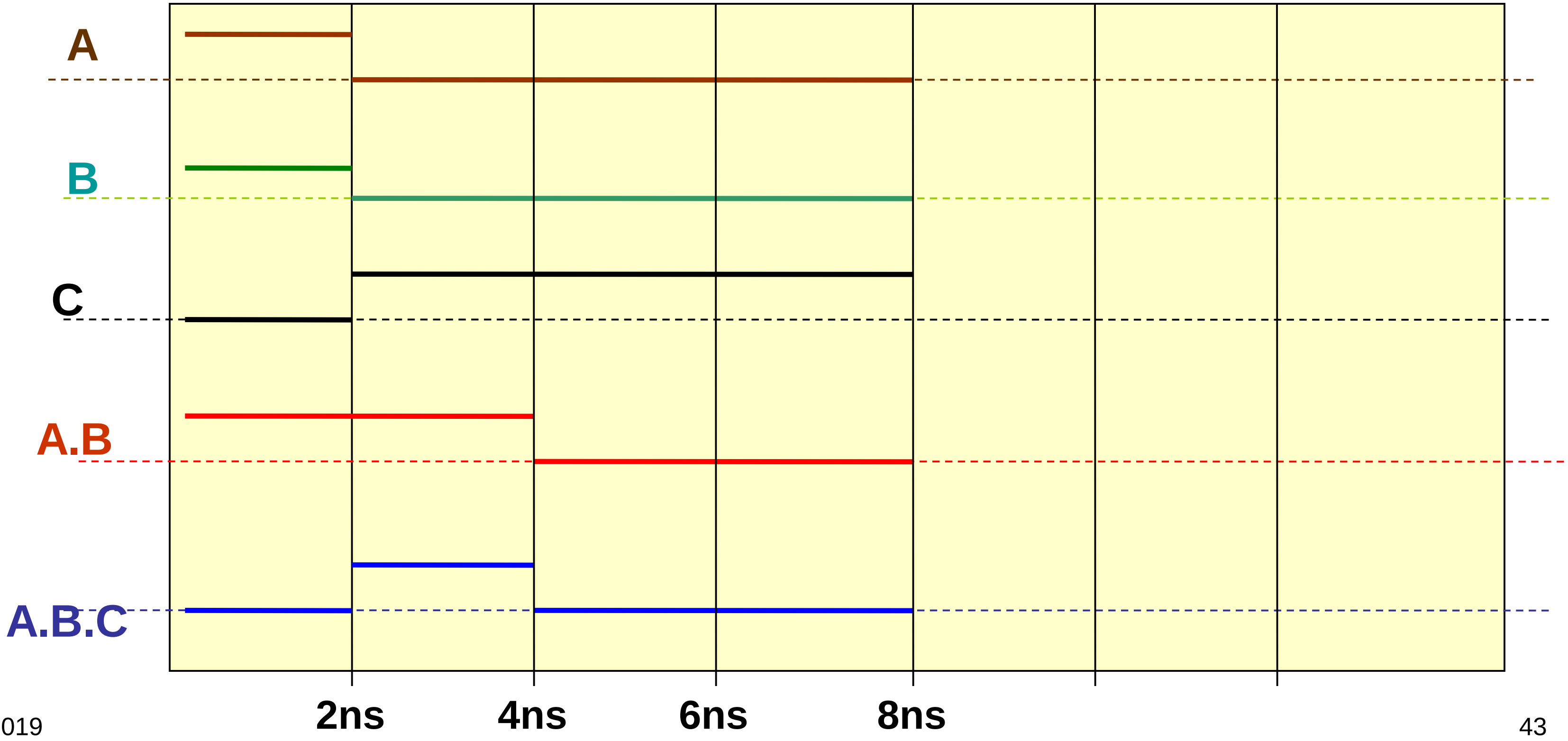
Total delay = 4ns

Timing Diagrams

- Illustrate change in inputs & outputs in a circuit with respect to time
  - In the form of a graph
    - Time on X-axis
    - Selected inputs / outputs on the Y-axis

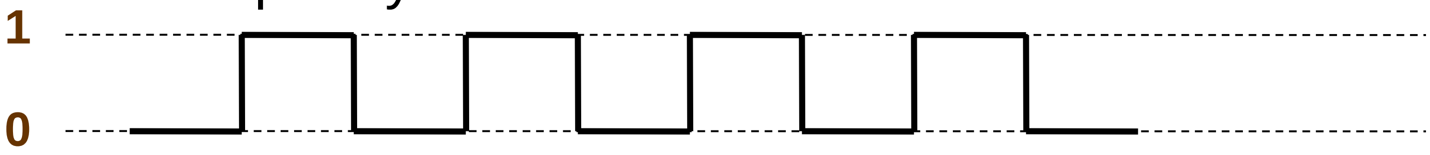
Timing Diagram Example





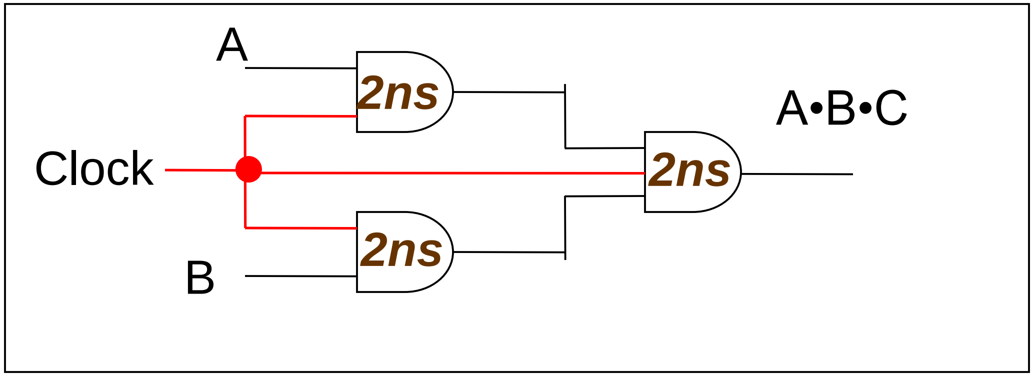
Clocks

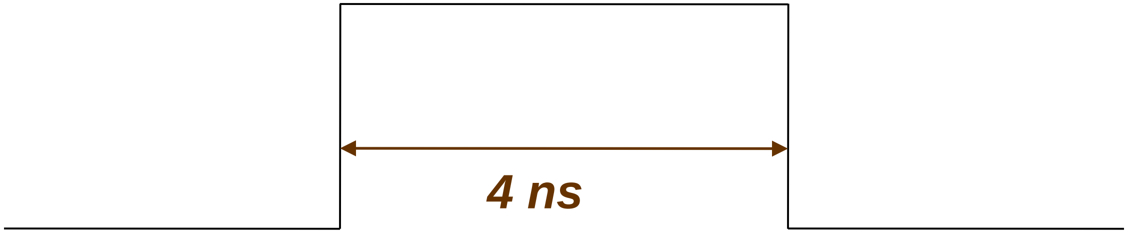
- Delays require careful timing
    - Otherwise results will be incorrect or garbled
  - Particularly when multiple inputs are to be processed
- I/O is synchronized using a Clock
  - Clock is a alternating sequence of 1 and 0
    - With a given periodicity or frequency
      - Frequency = 1/Period
    - Frequency is determined by the gate delays and circuit complexity



Clock Example

- Clocked I/O
  - Minimum clock period = 4ns
  - Maximum Frequency = 1/4ns = 250 MHz



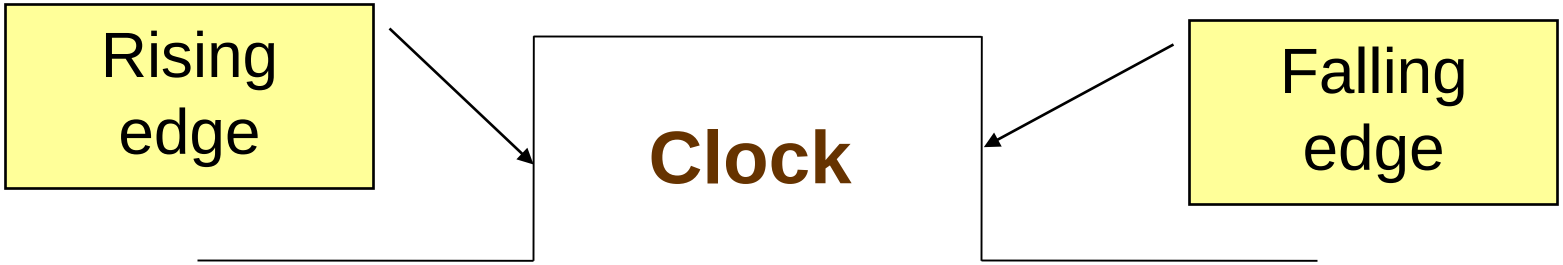


Bus clock example

- A computer with
  - 64 bit data bus
    - A read/write operation on RAM takes 4 cycles
    - Bus clock is 800Mhz (1Mhz=10^6Hz)
- What is data transfer rate (in MBps)?
    - Number of transfer per second=$800.10^6/4=2.10^8$
    - One transfer has 8 bytes
    - Transfer rate=$8.2.\ 10^8 = 5^8.2^{12}$ Bps≈1526MBps

Triggering

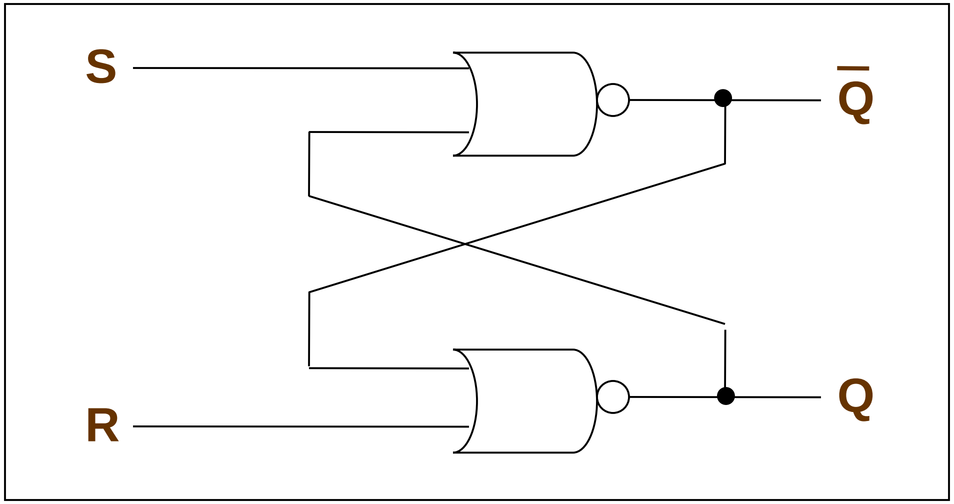
- Clocks transitions are used in different ways
  - Level triggering
    - When clock is in a given state
  - Edge triggering
    - Raising edge triggered
      - When the clock is in transition from 0 → 1
    - Falling edge triggered
      - When the clock is in transition from 1 → 0



Latches

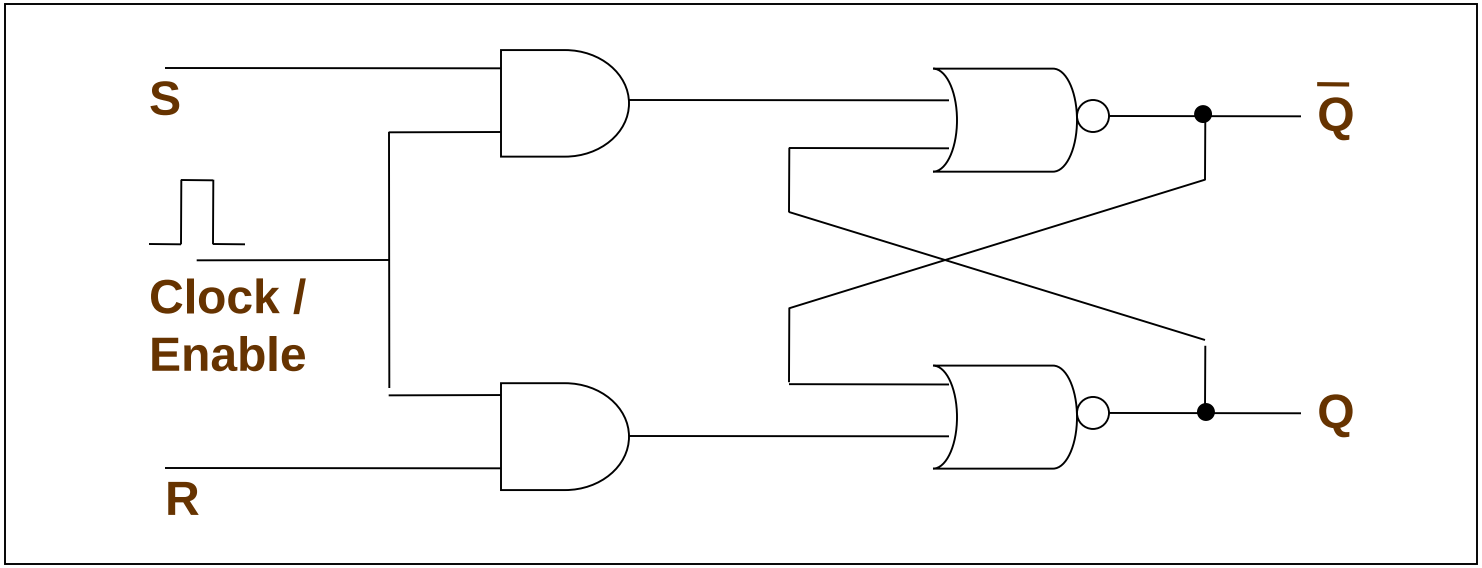
- Latches maintain state
  - Can be set to a specific value
  - Output of latches does not change even after Inputs change to 0!
- Fundamental units for storage
  - Building blocks for memory
- Latches always store data when the clock is at a fixed level
  - Hence they are also called as level triggered device

Set-Reset (SR) Latch



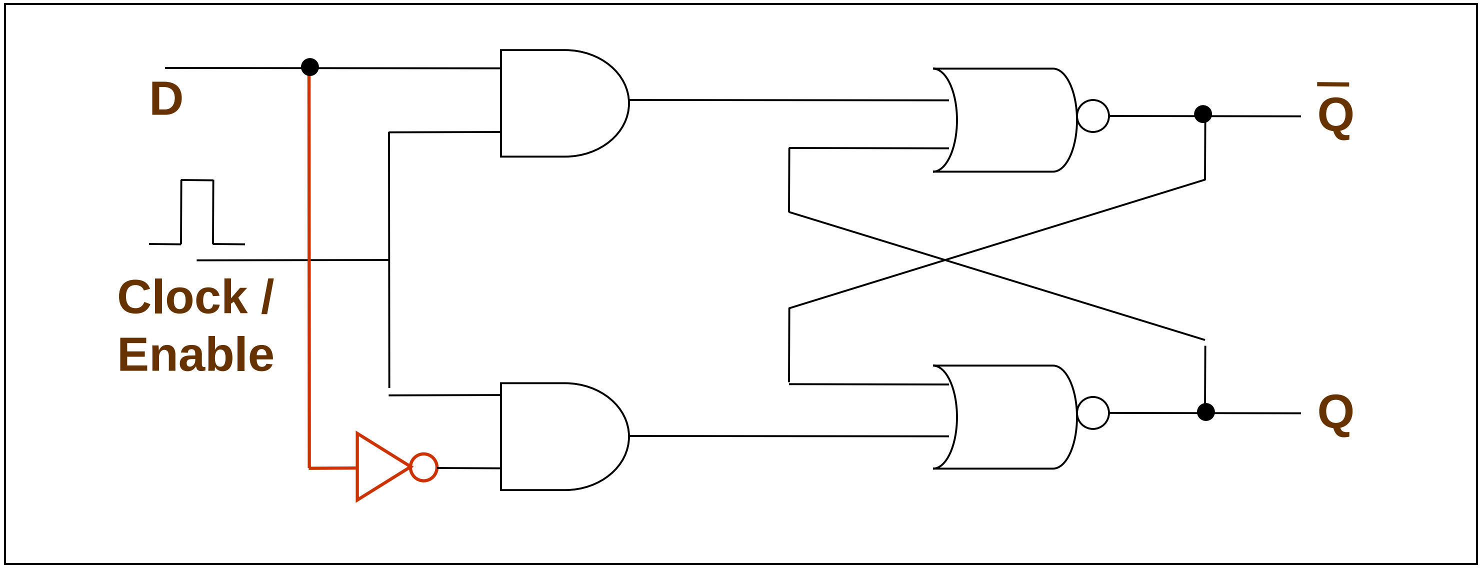
S	R	Q
0	0	No change
1	0	1
0	1	0
1	1	Unstable

Clocked S-R Latch



- Latch stores (or changes) value only when clock is high
  - Clock must be at logic level 1 to store data in the latch.
  - Data can be read at any time

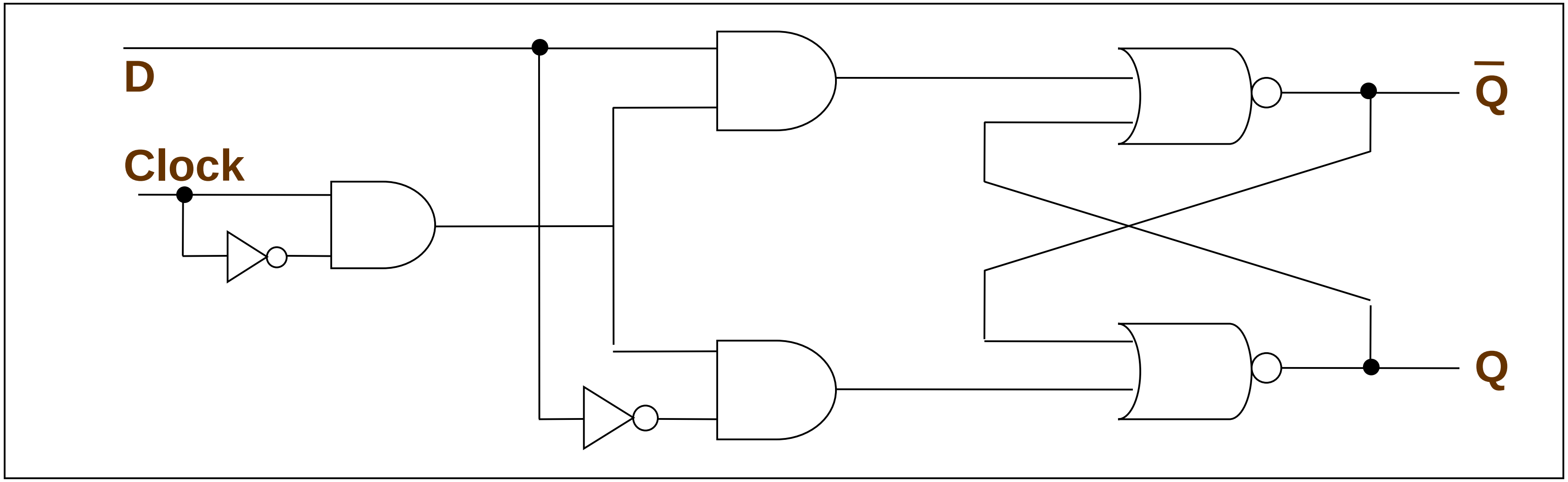
D-Latch



- Advantages over S-R Latch
  - Single input to store 1 or 0
  - Avoid spurious input of S=1 and R=1

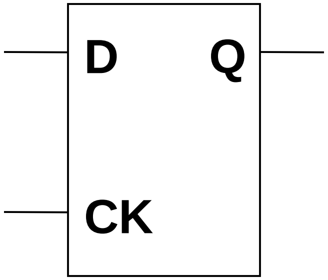
D-Flip Flop

- An edge triggered D-Latch is a D-Flip Flop

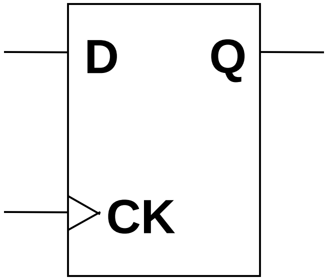


- Stores data only on raising edge
  - Changes at input at other times is ignored
    - Suitable clock frequency permits data to be stored only after inputs have settled
    - Data can be read at any time!

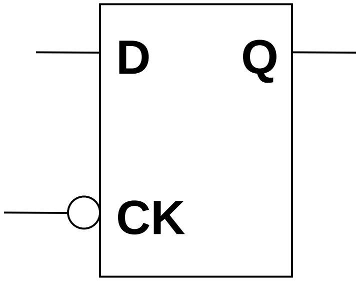
Abstract Representations



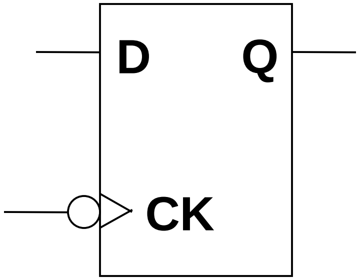
D-Latch
(Positive Level Triggered)



D-Flip Flop
(Rising Edge Triggered)



D-Latch
(Negative Level Triggered)



D-Flip Flop
(Falling Edge Triggered)

Asserted: Terminology

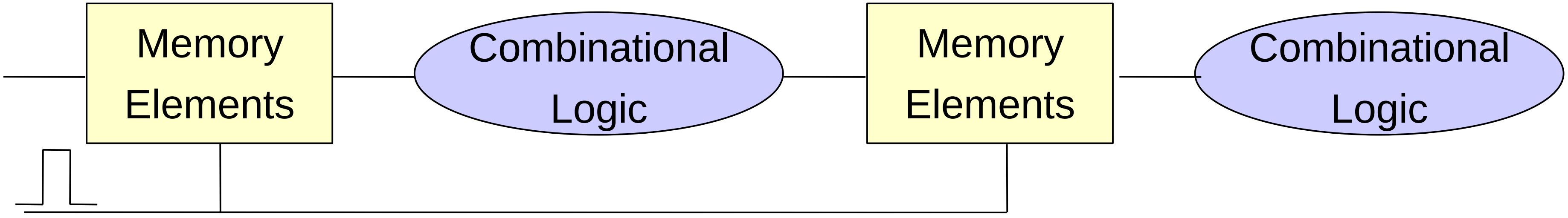
- Flip Flops use positive or negative logic
  - Same concept applies to other devices
  - In order to ease discussion the term “asserted” is used
    - Positive logic
      - A “1” triggers the working of the device
    - Negative logic
      - A “0” triggers the working of the device

Sequential Logic Circuits

- Involve one or more memory elements
  - Output depends on value in memory element
    - Typically based on earlier computations or history
  - Opposite of combinatory logic circuits
      - Also known as Combinatorial logic circuits
      - Circuits we have been dealing with so far
    - Does not include a memory element
    - Outputs depend purely on primary inputs

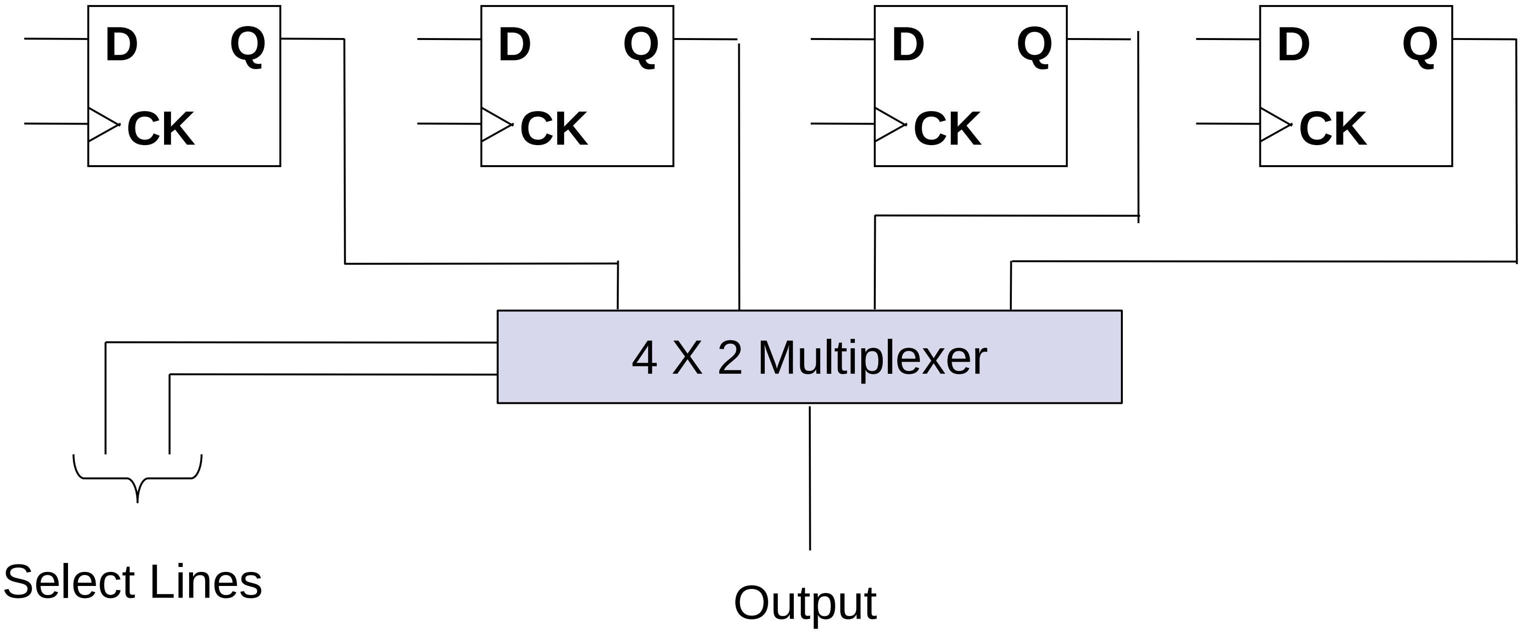
Typical Sequential Circuits

- Clocks control timings
  - Ensure values are not stored when they are transient
    - Have to wait for the signals to stabilize
  - State elements store values between computations



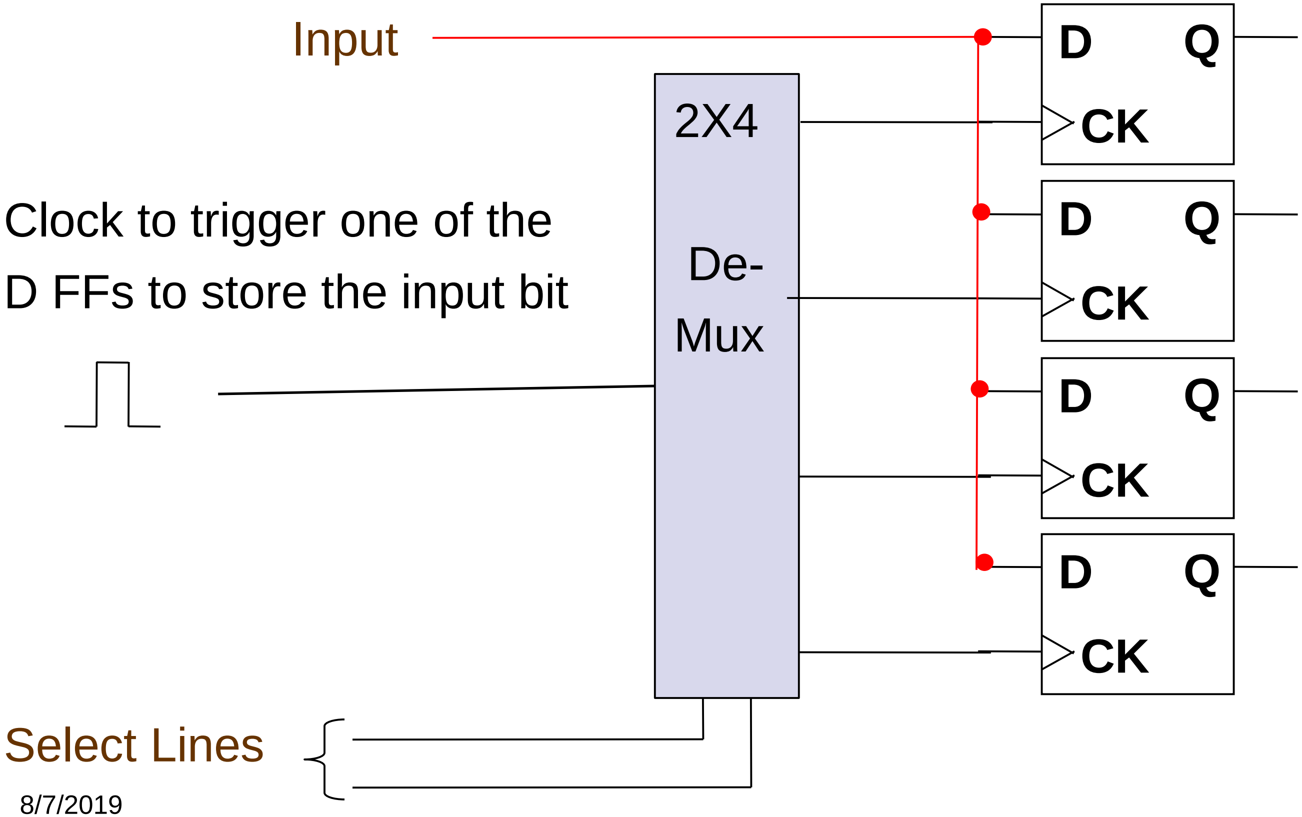
Circuit to read a Bit

- Given 4 Flip Flops, develop a logic circuit to select and read a given Flip Flop.



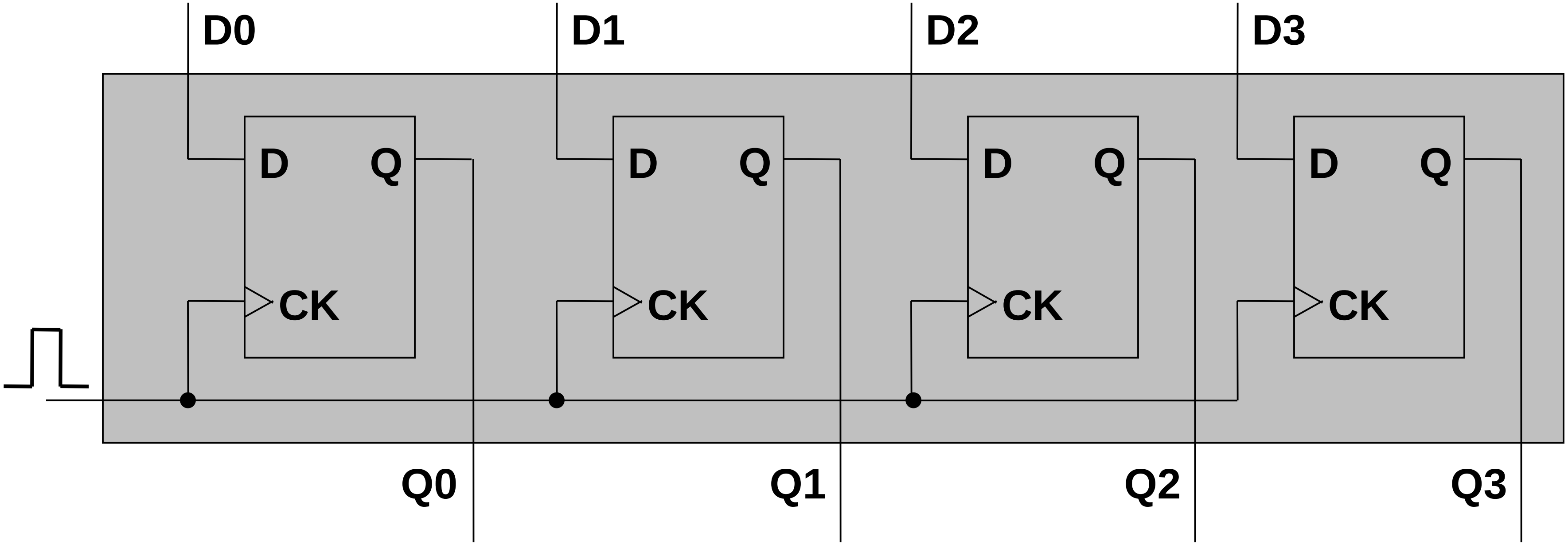
Circuit to write a Bit

- Given 4 Flip Flops, develop a logic circuit to select and change data in a given Flip Flop.



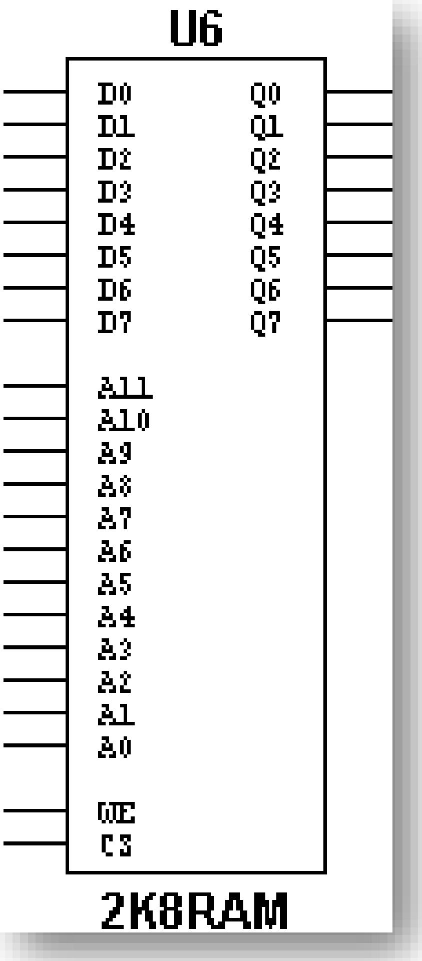
Word

- A fixed number of D-Flip Flops
  - Usually powers of 2 (2, 4, 8, 16, 32, 64)
  - Operate as a single unit
    - Store/Read n-bits at a time

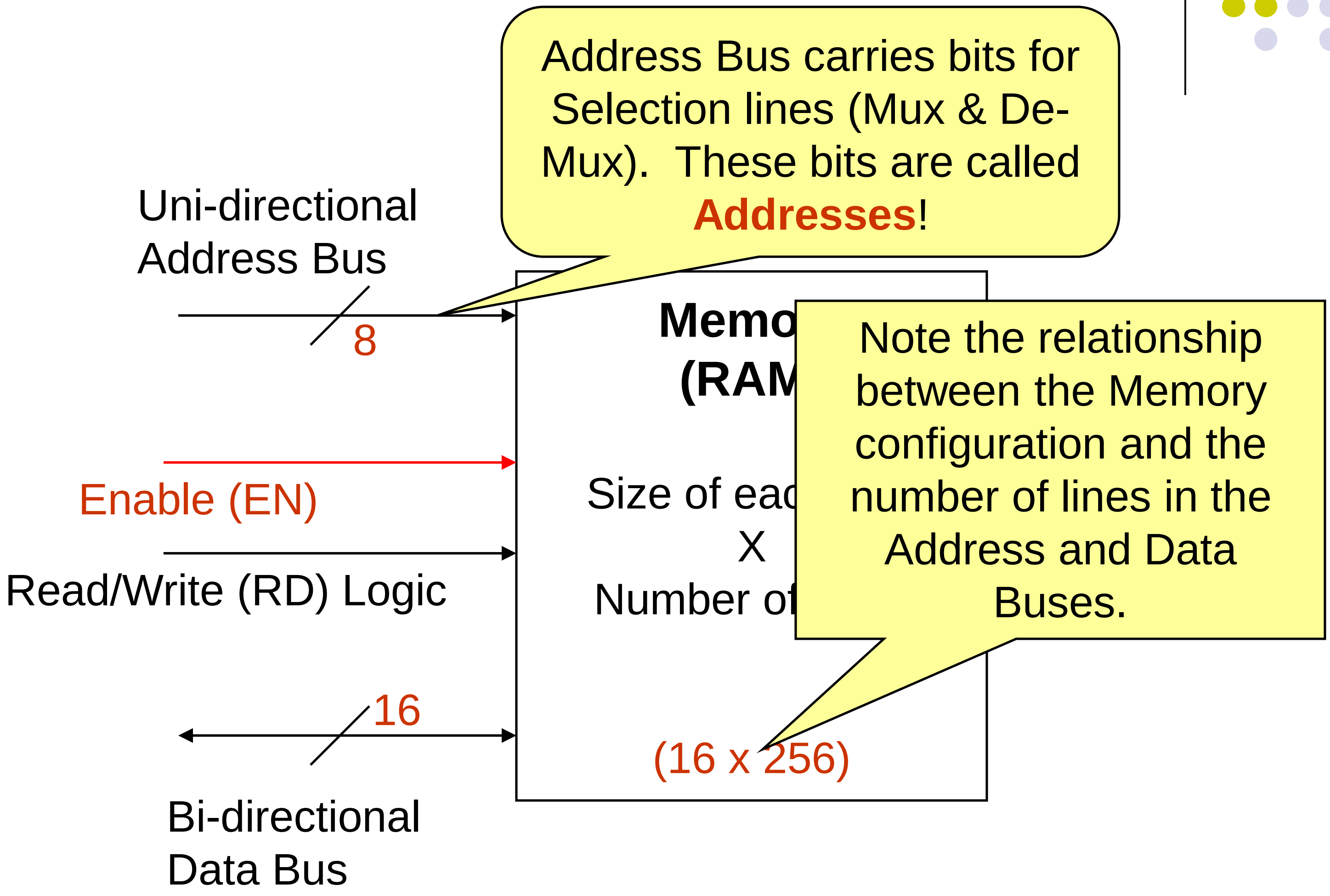


Random Access Memory (RAM)

- RAM is the common form of main memory that is store data and programs in modern computers.
  - It is typically designed as a collection of flip flops the previous slide
    - However fabrication technology is different to reduc improve transistor densities
  - Terminology:
    - Lines that carry input or output data are referred to or data bus
    - The select lines associated with the Mux and DeMu the address bus
      - The selection data is called address
      - In programming terminology it is called a pointer or a refer



Abstract Notation for Memory



Address Bus carries bits for Selection lines (Mux & De-Mux). These bits are called **Addresses**!

Uni-directional Address Bus

8

Memo

(RAM

Note the relationship between the Memory configuration and the number of lines in the Address and Data Buses.

Enable (EN)

Size of eac

X

Number of

Read/Write (RD) Logic

16

(16 x 256)

Bi-directional Data Bus

Number Representation

Its all 1s and 0s!

- Computers operate using bits
  - Everything is ultimately a “1” or a “0”
- Mind Bender
  - If everything is a “1” or a “0” then how does the computer:
    - Show and edit video?
    - Compose and play music?
    - Display and modify pictures?
    - Learn and recognize speech?

Magic of Interpretation

- The key is interpretation
  - Different sequence of 1s and 0s are assumed to convey different meanings to different devices
  - Example: What does bits 1011 imply
    - Letter “A” on the keyboard
    - A number (11) to the CPU
    - Color red to the video card
    - Music note F# to the sound card
    - Sector number 11 to the hard disk drive

Standards

- Interpretations can be problematic
  - Different interpretations can lead to confusion
  - Problems with portability & interoperability
- Solution: Standards
  - Standard ways to represent data
  - Different data types need different standards
    - Integers have different representation versus floating point numbers
    - Numbers have different representation than characters and strings
      - Depending on needs
      - Efficiency of hardware implementation

Integer representation

- Several standards are available
  - Mirrors standard mathematical representation of numbers for most part
  - Subdivided into 2 main categories
  - Unsigned integer representation
    - Standard binary number representation
  - Signed integer representation
    - Sign-bit Magnitude representation
    - 1’s Complement representation
    - 2’s Complement representation

Background on number representation

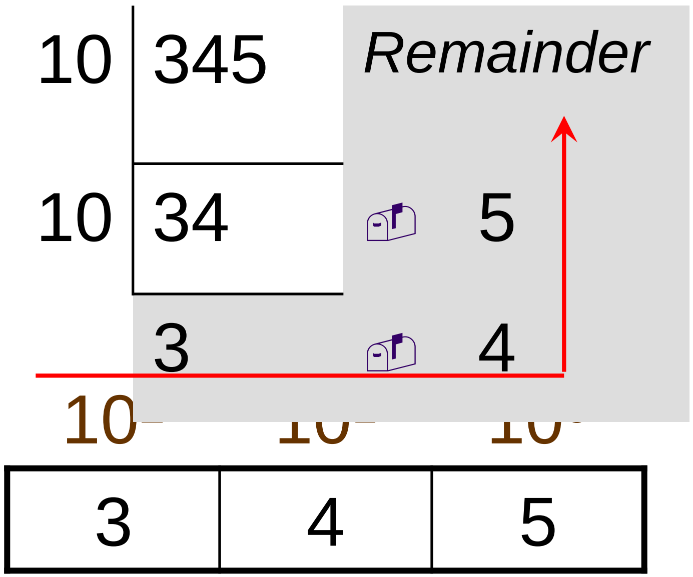
- Number representation are based on mathematical representation of decimal numbers
  - Uses 10 symbols (0 through 9) to represent 10 different values
  - Uses powers of 10 and a sequence of above symbols to represent larger values

10^3	10^2	10^1	10^0
0	3	8	6

Think Time

- Standard number conversions

Break 345 into Units, tens, & hundredths place



10 | 345

10 | 34 — 5

3 — 4

Remainder

10^2	10^1	10^0
3	4	5

Combine the digits shown below into a number

10^2	10^1	10^0
3	4	5

Value=$(3*10^2)+(4*10^1)+(5*10^0) = 300+40+5 = 345$

Binary number representation

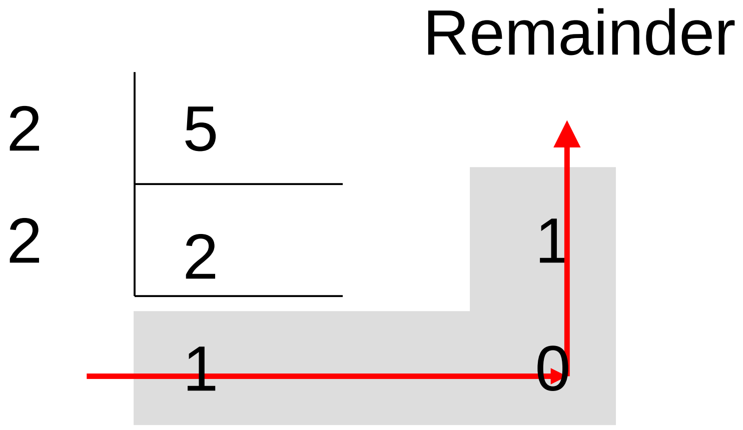
- In Binary system there are only 2 symbols
    - “1” and “0”
  - Yields base-2 representation
  - This enables representation of 2 unique values
    - Namely 0_2 = 0_{10} and 1_2=1_{10} d
      - Notice the use of base values for numbers!
  - Analogous to the decimal system, larger numbers are represented using sequence of “1”s and “0s”
    - Each position indicates a specific power of 2

Example binary number:

2^2	2^1	2^0
1	0	1

Unsigned Decimal to Unsigned Binary Conversion

- Performed through successive division by 2
  - Until quotient becomes 1
  - Writing remainders in reverse order
- Example convert 5_{10} to binary



Remainder

2 | 5

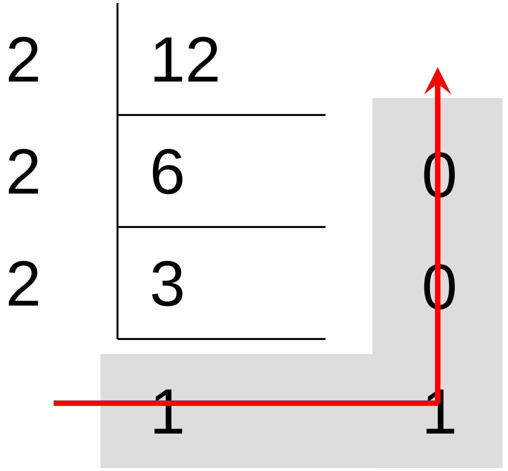
2 | 2 1

1 0

$$\boxed{5_{10} = 101_2}$$

Example 2

- Convert 12 to binary



2 | 12

2 | 6 0

2 | 3 0

1 1

$$\boxed{12_{10} = 1100_2}$$

Fixed Size Representation

- Numbers are represented using a fixed number of bits
  - Typically a word is used
    - A word can be 8, 16, 32, or 64 bits depending on how a given machine is designed.
  - Representation mechanism
    - Convert decimal to binary
    - Pad binary with leading 0s to fit given word size

Example

- Represent 12_{10} as a 8-bit binary number
  - Solution:
    - Convert 12_{10} to binary which is 1100_2
    - Now 1100_2 as 8-bit number = 00001100_2
      - Padding 4 leading 0s to make it 8-bits wide
- Represent 39_{10} as a 5-bit binary number
  - Solution:
    - Convert 39_{10} to binary which is 100111_2
      - Cannot fit 6-bits into 5-bits position!
    - Drop left-most digits as necessary
      - Result = 00111_2

Binary to Decimal Conversion

- Multiply by powers of 2 and add
  - Powers of 2 increase from left to right!
- Example: Convert 110_2 to decimal

2^2	2^1	2^0
1	1	0

$$\text{Decimal} = (1*2^2) + (1*2^1) + (0*2^0)$$

$$= 4 + 2 + 0 = 6$$

$$\boxed{110_2 = 6_{10}}$$

Binary to Decimal Conversion (Example 2)

- Example: Convert 11010_2 to decimal
  - Value = $(1*2^4)+(1*2^3)+(0*2^2)+(1*2^1)+(0*2^1)$

    = 16 + 8 + 0 + 2 + 0

    = 26
  - $11010_2 = 26_{10}$
- Tip for verification
  - Even valued binary numbers have 0 at the end
  - Odd valued binary numbers have 1 at the end

Table of Values

Decimal (Base-10)	*Binary (Base-2)*
0_{10}	0_2
1_{10}	1_2
2_{10}	10_2
3_{10}	11_2
4_{10}	100_2
5_{10}	101_2
6_{10}	110_2
7_{10}	111_2
8_{10}	1000_2
9_{10}	1001_2

Range of numbers

- Given a unsigned binary number with K positions
  - Minimum number = 0
  - Maximum number = 2^K-1

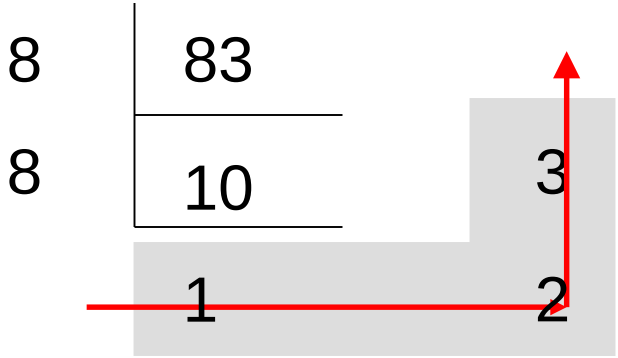
- Example if K = 8
  - With 8 bits, maximum number = 2^8-1 = 255

Octal representation

- Octal representation is to the base-8
  - Uses symbols “0” through “7” to represent 8 different values
  - Uses sequence of symbols for larger numbers.
- Convenient for representation of larger numbers
  - Easy to convert between Octal & binary
  - Requires fewer display places on hardware devices

Decimal to Octal Conversion

- Performed through successive division by 8
    - Similar in philosophy to other conversions
  - Until quotient becomes less than 8
  - Writing remainders in reverse order
- Example: Convert 83_{10} to Octal



8 | 83

8 | 10 3

1 2

$\boxed{83_{10} = 123_8}$

Octal to Decimal Conversion

- Example: Convert 567_8 to decimal
  - Value = $(5*8^2)+(6*8^1)+(7*8^0)$

    = 320 + 48 + 7

    = 375
  - $567_8 = 375_{10}$

Octal to Binary Conversion

- Simply write 3-bit binary representation for each digit as if it was a decimal digit
  - 3-bits are needed to represent 8 different values (0 to 7) for each digit in the octal representation
  - In a left to right manner
- Example: Convert 123_8 to binary
  - $1_8 = 001_2$
  - $2_8 = 010_2$
  - $3_8 = 011_2$
  - $123_8 = 001010011_2$

Binary to Octal Conversion

- Organize bits in binary number in sets of 3
  - From right to left manner
  - Write decimal value for each set of 3 bits
    - Range of values will be from 0 to 7
- Example: Convert 10110000_2 to Octal
  - 10 110 000
  - 2 6 0
  - 260_8

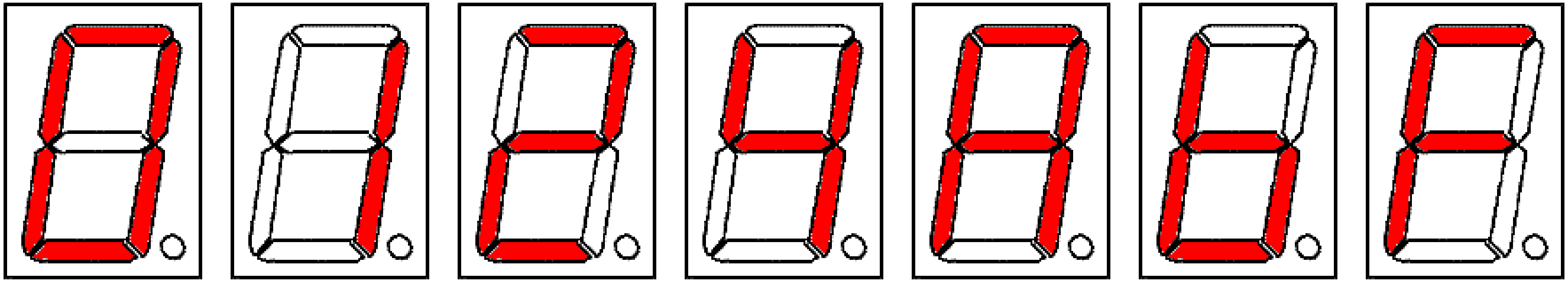
Hexadecimal Representation

- Hexadecimal (or Hex) uses Base-16
  - Requires 16 symbols for values 0 to 15
    - Uses standard numerals for 0 to 9
    - Uses A through F for values 10 to 15

Dec	***Hex***	***Dec***	***Hex***
0_{10}	0_{16}	8_{10}	8_{16}
1_{10}	1_{16}	9_{10}	9_{16}
2_{10}	2_{16}	10_{10}	A_{16}
3_{10}	3_{16}	11_{10}	B_{16}
4_{10}	4_{16}	12_{10}	C_{16}
5_{10}	5_{16}	13_{10}	D_{16}
6_{10}	6_{16}	14_{10}	E_{16}
7_{10}	7_{16}	15_{10}	F_{16}

Motivation for Hex

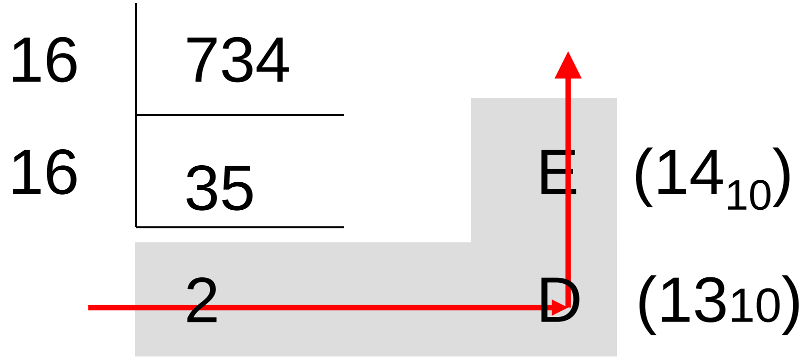
- Primarily for display purposes
  - Originally intended for 7-segment displays



  - Still used in conventional computers for display of binary information
    - Such as memory addresses etc.

Decimal to Hex Conversion

- Performed through successive division by 16
    - Similar in philosophy to other conversions
  - Until quotient becomes less than 16
  - Writing remainders in reverse order
- Example: Convert 734_{10} to Hex



16 | 734

16 | 35 E (14_{10})

2 D (13_{10})

$734_{10} = 2DE_{16}$

Hex to Decimal Conversion

- Example: Convert $A8F_{16}$ to decimal
  - Value $= (A*16^2)+(8*16^1)+(F*16^0)$

    $= (10*16^2)+(8*16^1)+(15*16^0)$

    $= 2560 + 128 + 15$

    $= 2703$
  - $A8F_{16} = 2703_{10}$

Hexal to Binary Conversion

- Simply write 4-bit binary representation for each digit as if it was a decimal digit
  - 4-bits are needed to represent 16 different values (0 to F) for each digit in the hexal representation
  - In a left to right manner
- Example: Convert $1C3_{16}$ to binary
  - $1_{16} = 0001_2$
  - $C_{16} = 1100_2$
  - $3_{16} = 0011_2$
  - $1C3_{16} = 000111000011_2$

Binary to Hexal Conversion

- Organize bits in binary number in sets of 4
  - From right to left manner
  - Write decimal value for each set of 4 bits
    - Range of values will be from 0 to F
- Example: Convert 100110000_2 to Hexal
  - 1   0011   0000
  - 1   3   0
  - 130_{16}

Table of Equivalent Values

Dec.	Binary	Octal	Hexal	Dec.	Binary	Octal	Hexal
0_{10}	0_2	0_8	0_{16}	8_{10}	1000_2	10_8	8_{16}
1_{10}	1_2	1_8	1_{16}	9_{10}	1001_2	11_8	9_{16}
2_{10}	10_2	2_8	2_{16}	10_{10}	1010_2	12_8	A_{16}
3_{10}	11_2	3_8	3_{16}	11_{10}	1011_2	13_8	B_{16}
4_{10}	100_2	4_8	4_{16}	12_{10}	1100_2	14_8	C_{16}
5_{10}	101_2	5_8	5_{16}	13_{10}	1101_2	15_8	D_{16}
6_{10}	110_2	6_8	6_{16}	14_{10}	1110_2	16_8	E_{16}
7_{10}	111_2	7_8	7_{16}	15_{10}	1111_2	17_8	F_{16}

Binary Addition

Computer Architecture

Review of Addition

- Addition of decimal numbers
  - Proceeds from lowest to highest powers
    - That is from left to right
  - Each digit is added
    - If result is below 10, the digit is written as is
    - If the result is above 9, it is broken into two parts
      - Sum: Digit which results from adding the numbers
      - Carry: Digit for use in the next higher power column.

Example

- Add 78_{10} with 99_{10}

1	1		
0	7	8	+
0	9	9	
1	7	7	

Binary Addition

- Proceeds in a similar fashion as conventional addition
  - Bits are added from right to left
    - Sum is 1 or 0
    - Carry is 1 or 0
  - Results from addition of bits is illustrated in the adjacent truth table.

A	*B*	C_{in}	*Sum*	C_{out}
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

Binary Addition: Example 1

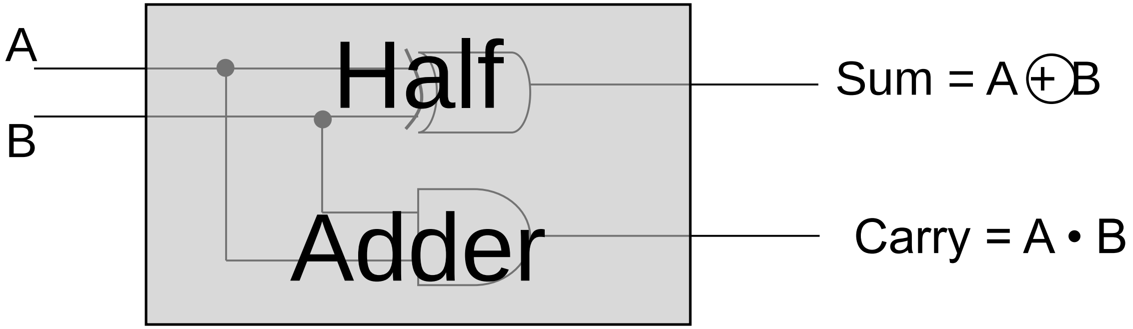
- Add 1011_2 to 1101_2

1	1	1	1		
0	1	0	1	1	
0	1	1	0	1	+
1	1	0	0	0	

Logic Circuit to add 2 Bits: Half Adder

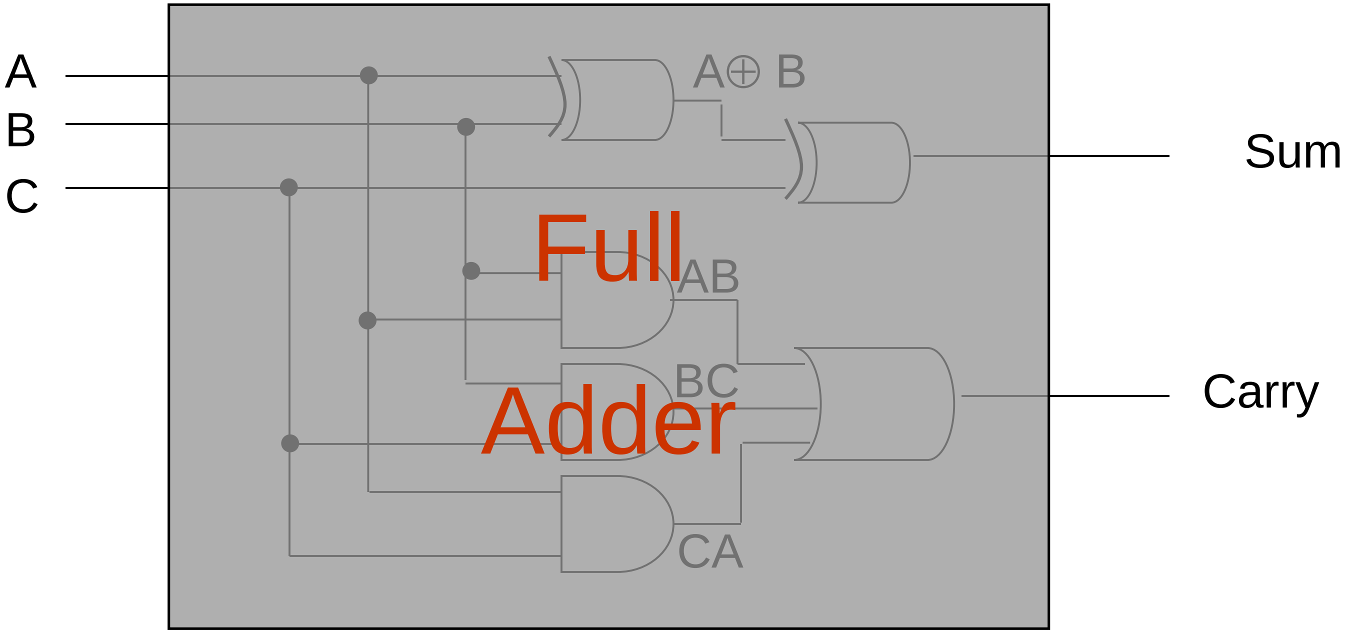
- Truth table for sum & carry bits are as follows.
- Given the truth table
  - Sum = $A \oplus B$
  - $C_{out} = A \bullet B$

A	***B***	***Sum***	***C_{out}***
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1



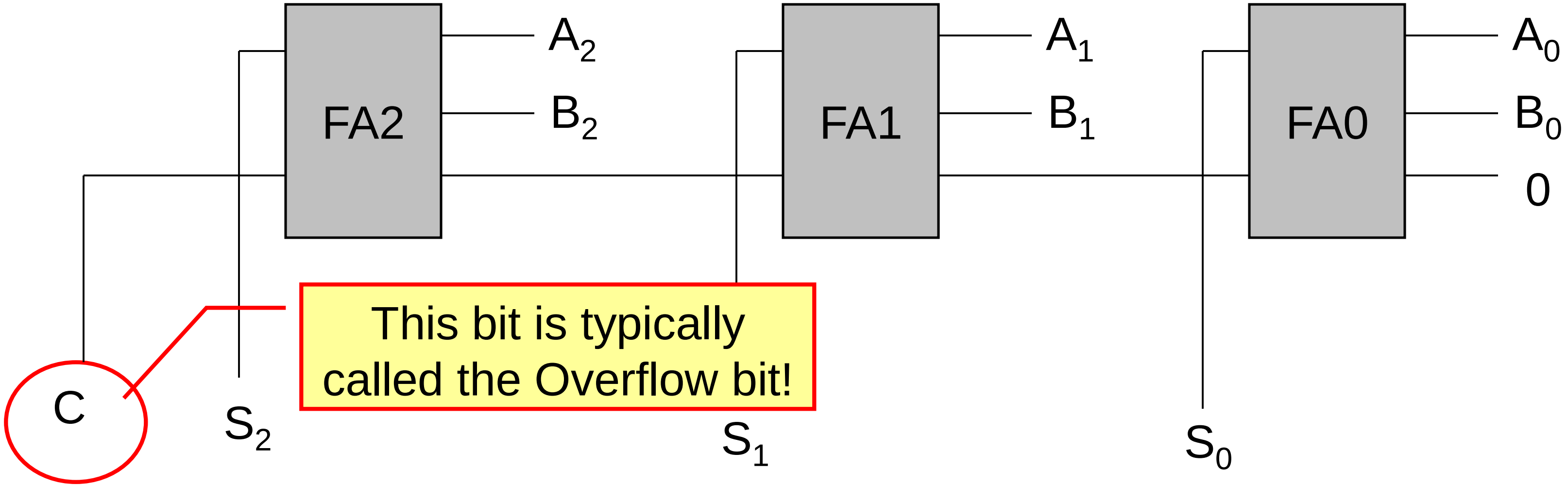
Logic Circuit to add 3 bits: Full Adder

- Refer to truth table shown earlier
  - Sum = $A \oplus B \oplus C$
  - Carry = (A•B) + (B•C) + (C•A)



Ripple Carry Adder

- Several full adder's can be cascaded to add multiple-bits!
  - Circuit reflects the way binary numbers are added
  - The following circuit adds 2 3-bit binary numbers namely $A_2A_1A_0$ and $B_2B_1B_0$ to yield result $C_3C_2C_1C_0$



This bit is typically called the Overflow bit!

Ripple Carry Adder

- To add 2 *n*-bit numbers you need *n* Full Adders
  - First carry to the circuit is 0
    - A half adder could be used here instead
  - Each carry-out (except the last) is fed to the carry-in of the next stage
    - Note that inputs are organized from lowest to highest power of 2
  - Circuit generates n+1 bits as result of addition
    - Last carry (highest power) is called an overflow bit
      - Because it does not fit in *n*-bits (which is typically what is expected to happen in these circuits).
    - If a carry is present in the n+1th bit it is called an overflow condition!
    - Microprocessors typically provide a mechanism to detect overflows!

Unsigned Representations

- Unsigned number representations
  - Binary (base-2)
  - Octal (base-8)
  - Hexadecimal (base-16)
- Given n bits
  - Range: 0 to 2^n-1 decimal numbers

Need for Signed Representations

- Unsigned representations cannot represent negative numbers
  - Such as: -2, -51 etc.
- However negative numbers are frequently used
  - Need a representation for positive & negative numbers – that is, signed numbers

Standard Signed Representations

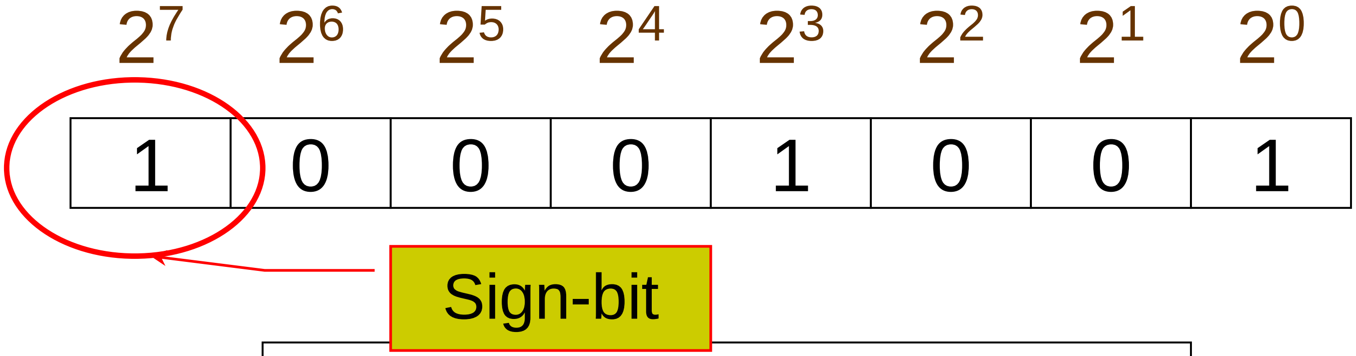
- Signed numbers are represented using 3 different standards
  - Sign-bit Magnitude (SBM) representation
  - 1’s Complement Representation
  - 2’s Complement Representation
    - This is the representation that is used by computers today!

Strategy for Signed Representation

- General strategy is as follows:
  - All representations assume fixed size
    - 8, 16, 32, or 64 bits operated on as a single unit-*word*
  - Given n bits (unsigned range: 0 to 2^n-1)
    - Break the range into two halves
  - One half represents positive numbers
    - 0 to 2^{n-1}-1 corresponds to positive numbers
      - Decimal value: 0 to 2^{n-1}-1
  - Another half represents negative numbers
    - 2^{n-1} to 2^n-1 corresponds to negative numbers
      - Decimal range: -2^{n-1} to -1

Sign-Bit Magnitude (SBM) Representation

- Uses left-most bit called sign bit to indicate sign
  - Sign bit = 0 implies positive number
  - Sign bit = 1 implies negative number
- Example:



2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	1	0	0	1

Decimal value = -9_{10}

SBM Examples (1)

- Represent 10_{10} using 8-bit SBM
  - Converting to binary: $10_{10} = 1010_2$
  - 8-bit SBM representation: 00001010_2
    - Sign bit is 0 to indicate positive value.
- Represent -15_{10} using 8-bit SBM
  - Converting to binary: $15_{10} = 1111_2$
  - 8-bit SBM representation: 10001111_2
    - Sign bit is 1 to indicate negative value

SBM Examples (2)

- Convert 8-bit SBM 00000101_2 to Decimal
  - The sign bit (left most bit) is 0 indicating positive value
  - Converting 0000101_2 to decimal we get 5_{10}
  - 00000101_2 in SBM = 5_{10}
- Convert 8-bit SBM 10001101_2 to Decimal
  - The sign bit (left most bit) is 1 indicating negative value!
  - Converting 0001101_2 to decimal we get 13_{10}
  - Result: $10001101_2 = -13_{10}$

1's Complement Representation

- There are a few drawbacks with SBM
  - There are 2 different representations for zero
    - +0 (00000000) and -0 (10000000)
    - Logic circuits for addition and subtraction of binary numbers are complicated as they have to handle sign bit separately.
- 1's Complement Representation
  - Fixed size representation
  - Most significant bit is reserved as sign-bit
  - Positive numbers are represented using standard binary notation
  - Negative numbers are represented by inverting all bits in the standard binary notation

Example of 1's Complement

- Represent 12_{10} using 8-bit 1's Complement
  - Converting to binary: $12_{10} = 1100_2$
  - 8-bit 1's complement representation: 00001100_2
    - Sign bit is 0 to indicate positive value.
- Represent -13_{10} using 8-bit SBM
  - Converting to binary: $13_{10} = 1101_2$
  - 8-bit representation: 00001101_2
  - Since original decimal number was negative each bit in 8-bit representation is inverted!
  - 1s Complement representation = **1**1110010
    - Note: Sign bit becomes **1** to indicate negative value

Example of 1's Complement

- Convert 1's complement 00001111_2 to decimal
  - Sign bit is 0 to indicating positive value.
  - Converting 00001111_2 to decimal we get 15_{10}
  - Final result: $+15_{10}$
- Convert 1's complement 11110101_2 to decimal
  - Sign bit is 1 indicating negative value.
  - First invert all the bits to get: 00001010_2
  - Convert above binary to decimal to get 10_{10}
  - Final result: -10_{10}

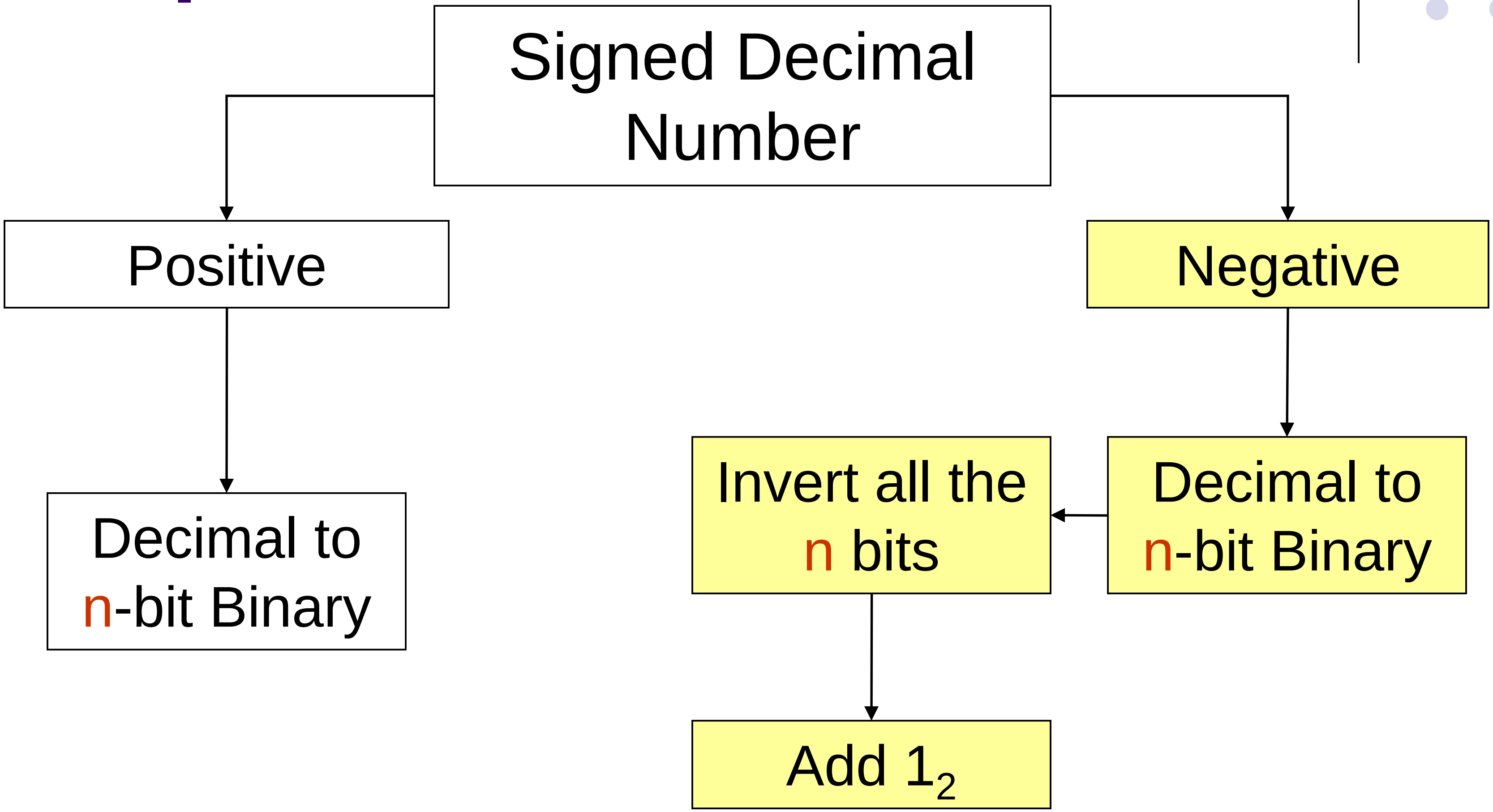
1’s Complement

- Still has 2 different representations for 0
  - +0 (00000000) and -0 (11111111)
- However, A – A operations can be easily represented
  - A – A = A + (-A) = 11111111 (which is effectively 0 in 1’s complement)

2’s Complement

- Overcomes the limitations of SBM and 1’s complement representation!
  - Fixed size representation
  - Enables subtraction of numbers via addition!
  - Also reserves a special sign bit (left most bit)
    - 0 (sign bit) indicates positive numbers
    - 1 (sign bit) indicates negative numbers
  - Conversion to and from 2’s complement requires similar steps
    - Several of them are identical to 1’s complement

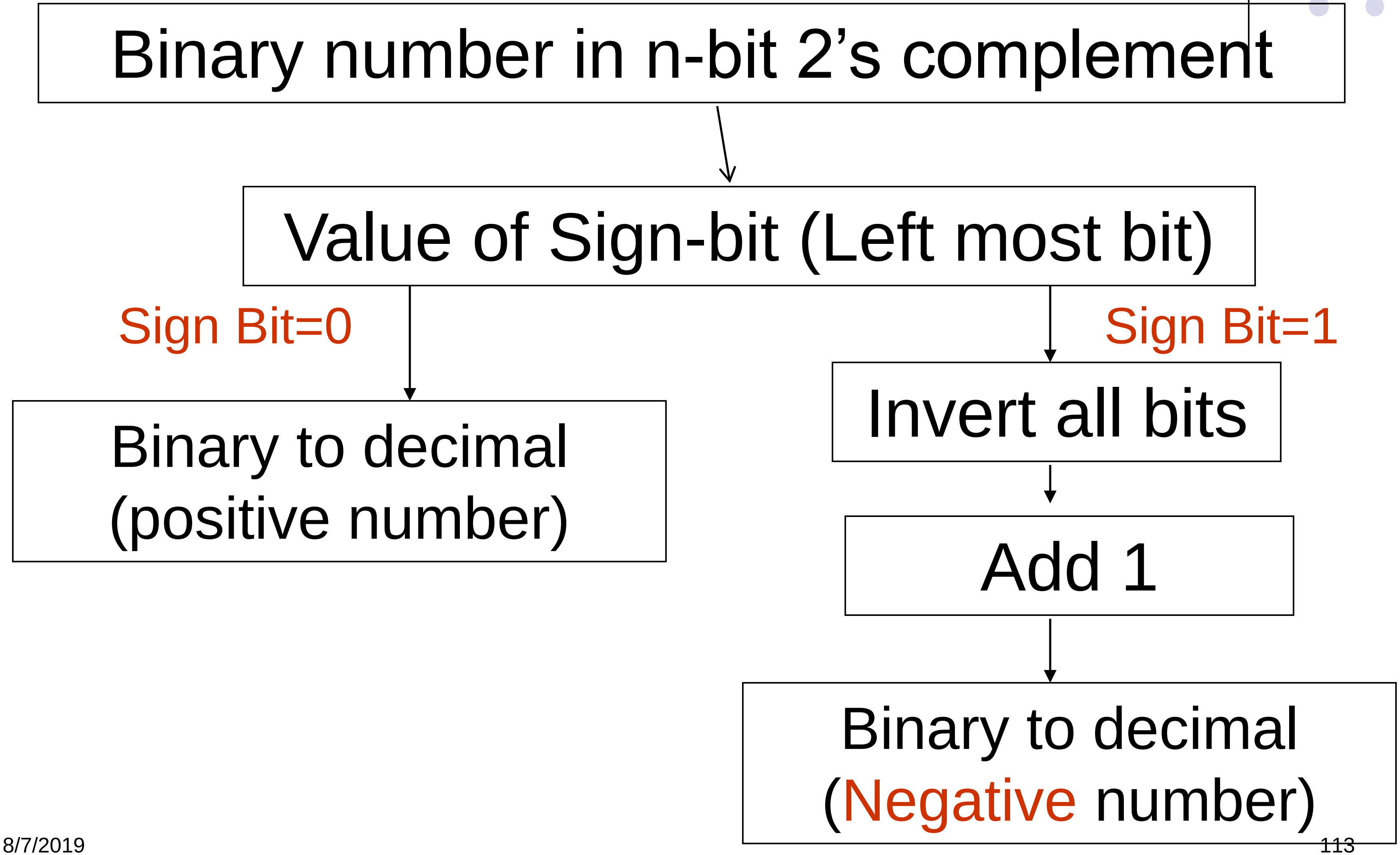
Convert Decimal to 2’s Complement



2’s Complement Example

- Represent $+20_{10}$ using 8-bit 2’s Complement
  - Since number is positive simply represent $+20_{10}$ as a 8-bit binary number!
  - $+20_{10} = 00010100_2$
- Represent -18_{10} using 8-bit 2’s complement
  - Since number is negative we need to do 1’s complement and add 1_2
  - Step 1: Convert 18_{10} to 8-bit binary = 00010010_2
  - Step 2: Invert bits = 11101101_2
  - Step 3: Add $1_2 = 11101110_2$
  - Final result: $-18_{10} = 11101110_2$ in 8-bit 2’s complement

2’s Complement to Decimal



Binary number in n-bit 2’s complement

Value of Sign-bit (Left most bit)

Sign Bit=0

Binary to decimal (positive number)

Sign Bit=1

Invert all bits

Add 1

Binary to decimal (Negative number)

Example

- Convert 8-bit 2’s complement to decimal
  - Case 1: 00001010_2
    - Sign bit is 0 indicating positive number
    - Simply convert binary to decimal to get 10_{10}
  - Case 2: 11111010_2
    - Sign bit is 1 indicating negative number
    - Step 1: Invert all bits to get 00000101_2
    - Step 2: Add 1_2 to get 00000110_2
    - Convert binary to decimal to get -6_{10}
      - Note the negative sign on decimal number!

Subtraction using 2's Complement

- Perform $5_{10} – 3_{10}$ using 4-bit 2's complement

$5_{10} – 3_{10} = 5_{10} + (-3_{10})$

$5_{10} = 0101_2$

$-3_{10} = (0011_2 \xrightarrow{\text{INV}} 1100_2 + 1_2) = 1101_2$

$5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2$

$0010_2 = 2_{10}$

Subtraction using 2’s Complement

- Perform $2_{10} – 4_{10}$ using 4-bit 2’s complement

$2_{10} – 4_{10} = 2_{10} + (-4_{10})$

$2_{10} = 0010_2$

$-4_{10} = (0100_2 \xrightarrow{\text{INV}} 1011_2 + 1_2) = 1100_2$

$2_{10} + (-4_{10}) = 0010_2 + 1100_2 = 1110_2$

1110_2 is 2’s complement result and because it is negative (sign bit is 1) it needs to be converted

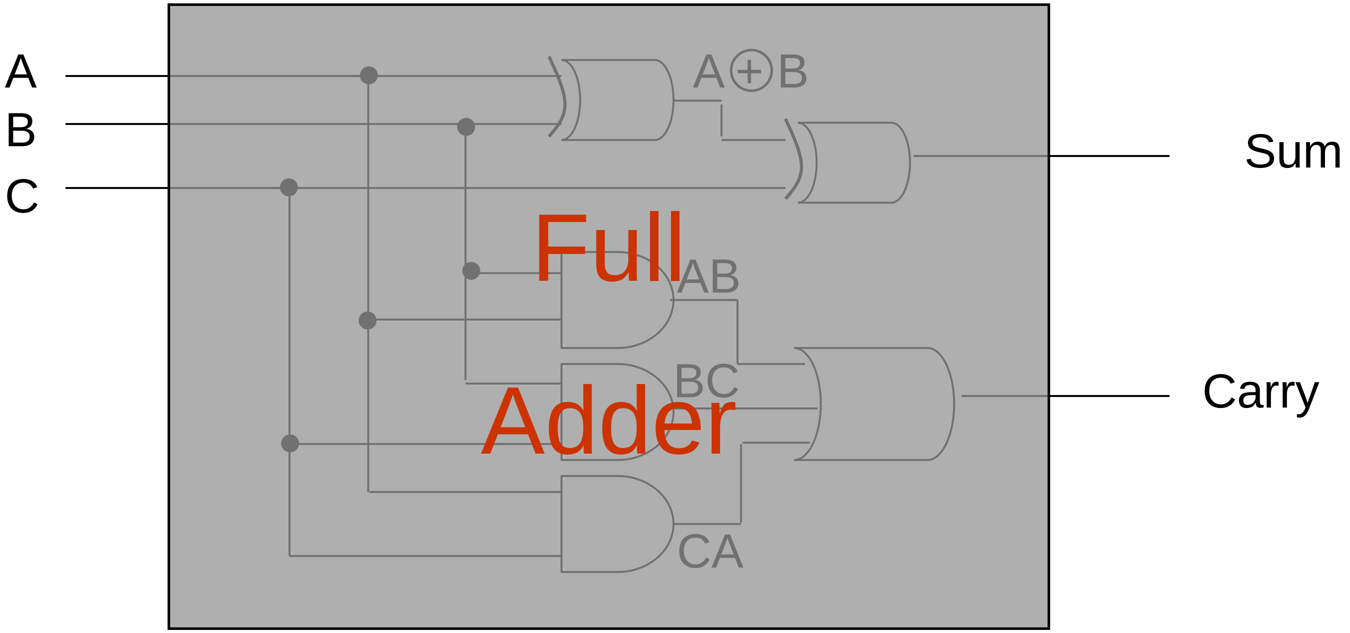
$1110_2 \xrightarrow{\text{INV}} 0001_2 + 1_2 = 0010_2 = -2_{10}$

Subtraction Circuit

- Circuit to subtract two 3-bit 2’s complement numbers
  - Recollect that subtraction is performed via addition in 2’s complement representation.
    - Addition is performed using Full Adder modules
    - Cascaded together in the form of Ripple Carry Adder
  - One of the numbers have to be converted to 2’s complement representation
    - Invert all the bits
    - Add 1_2

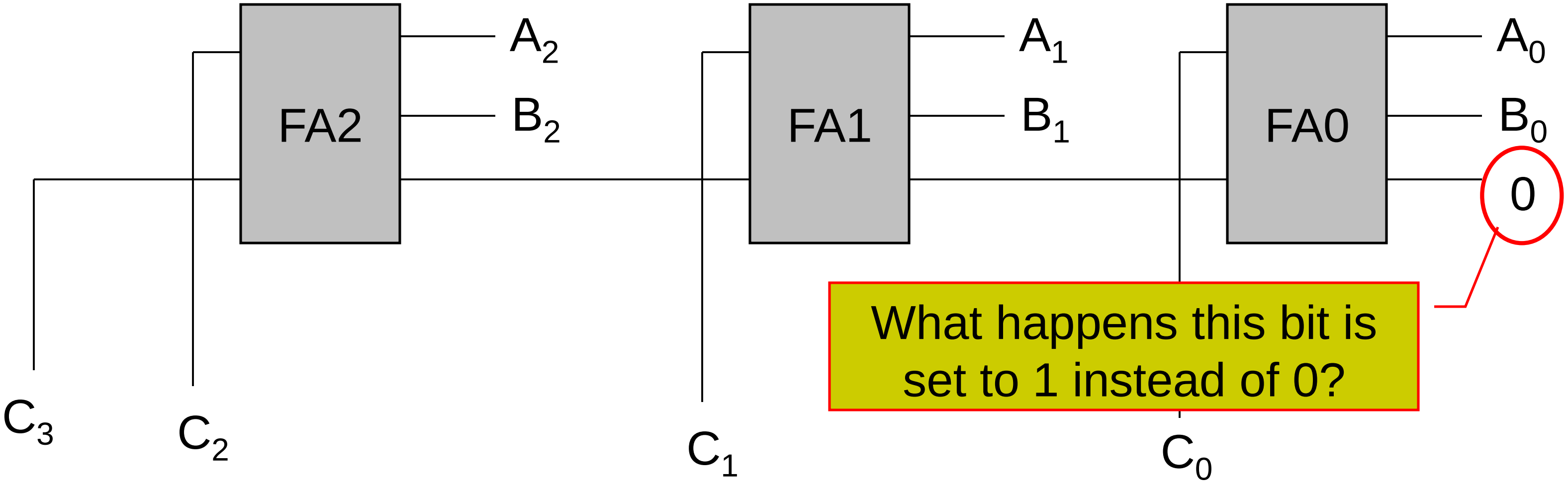
Logic Circuit to add 3 bits: Full Adder

- Refer to truth table shown earlier
  - Sum = $A \oplus B \oplus C$
  - Carry = (A•B) + (B•C) + (C•A)



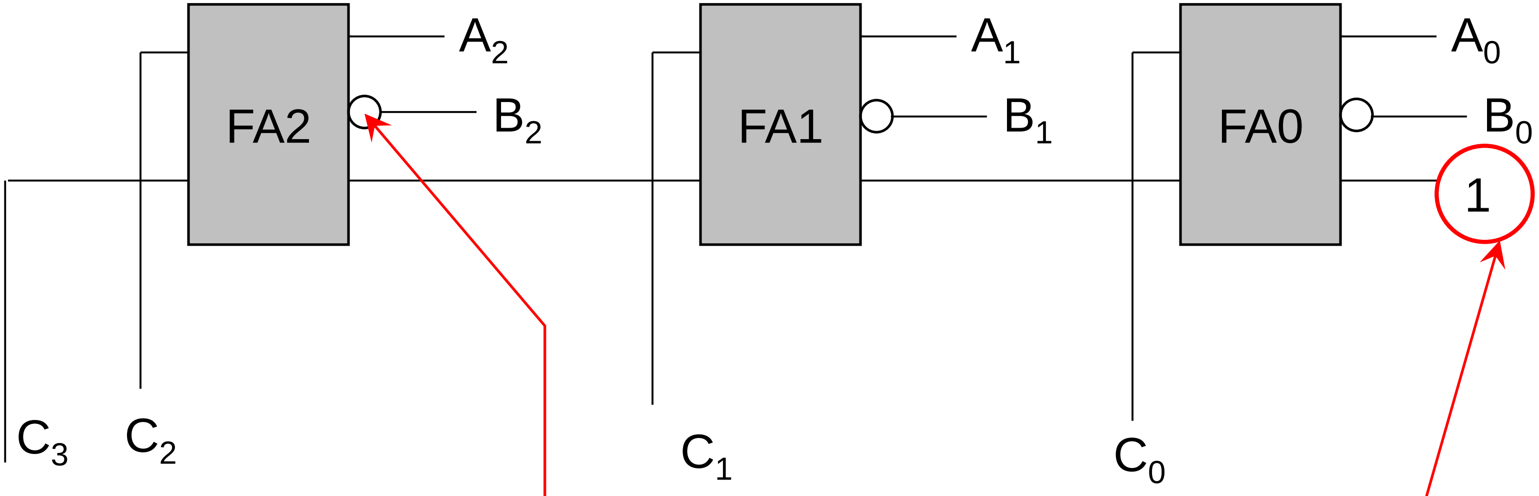
Ripple Carry Adder

- Several full adder’s can be cascaded to add multiple-bits!
  - Circuit reflects the way binary numbers are added
  - The following circuit adds 2 3-bit binary numbers namely $A_2A_1A_0$ and $B_2B_1B_0$ to yield result $C_3C_2C_1C_0$



What happens this bit is set to 1 instead of 0?

Subtraction Circuit



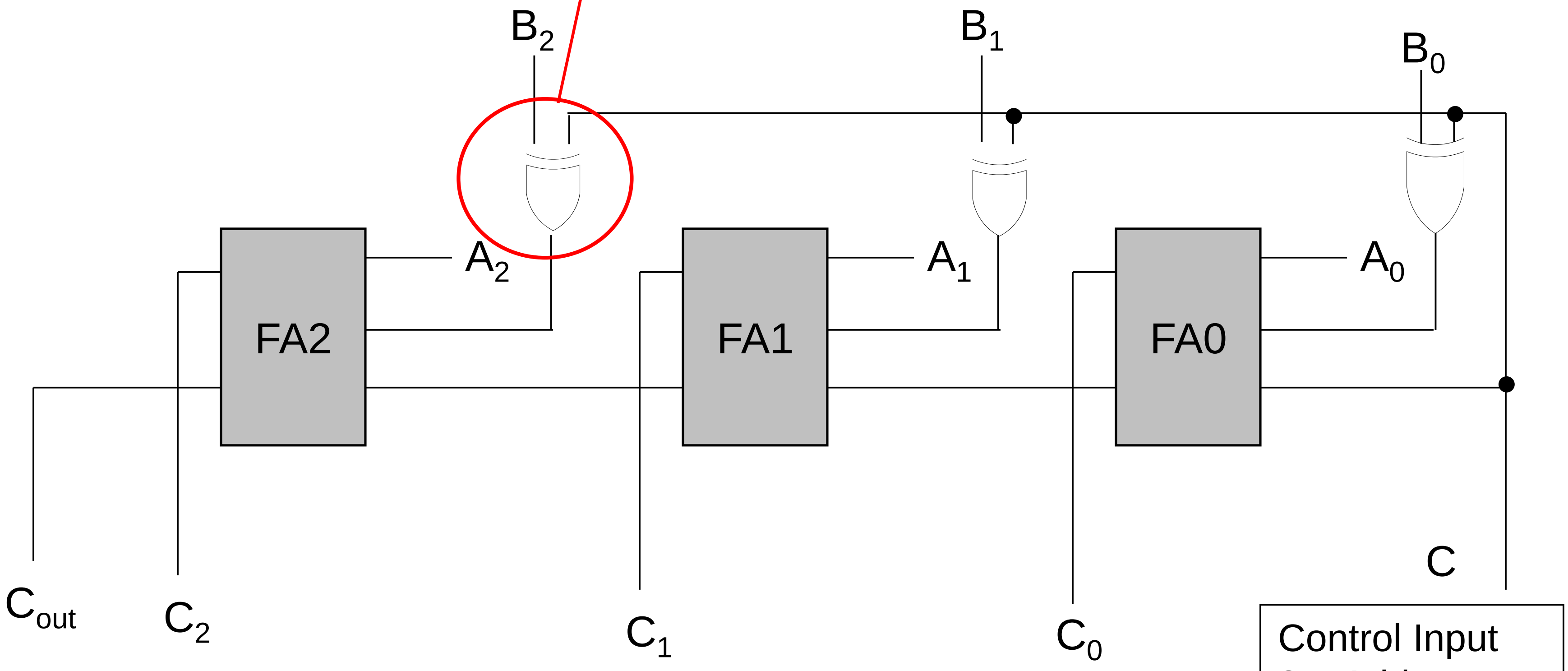
Each bit is inverted (Step 1 of Converting to 2's Complement)

Add 1 to inverted B_n bits to generate 2's complement!

Add/Subtract Circuit

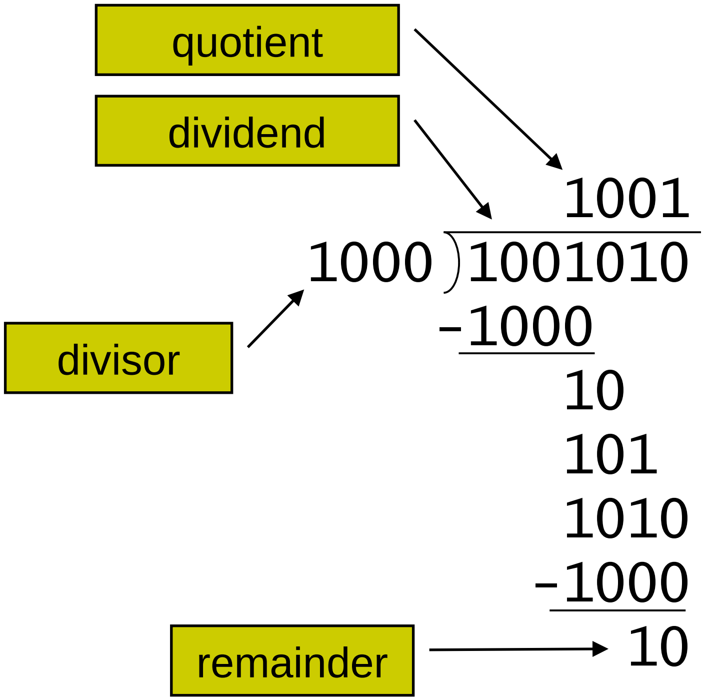
- Circuit to Add or Subtract two 3-bit 2's complement number

The XOR gate inverts the bits only when control input is 1



Control Input
0 = Add
1 = Subtract

Division


quotient

dividend

divisor

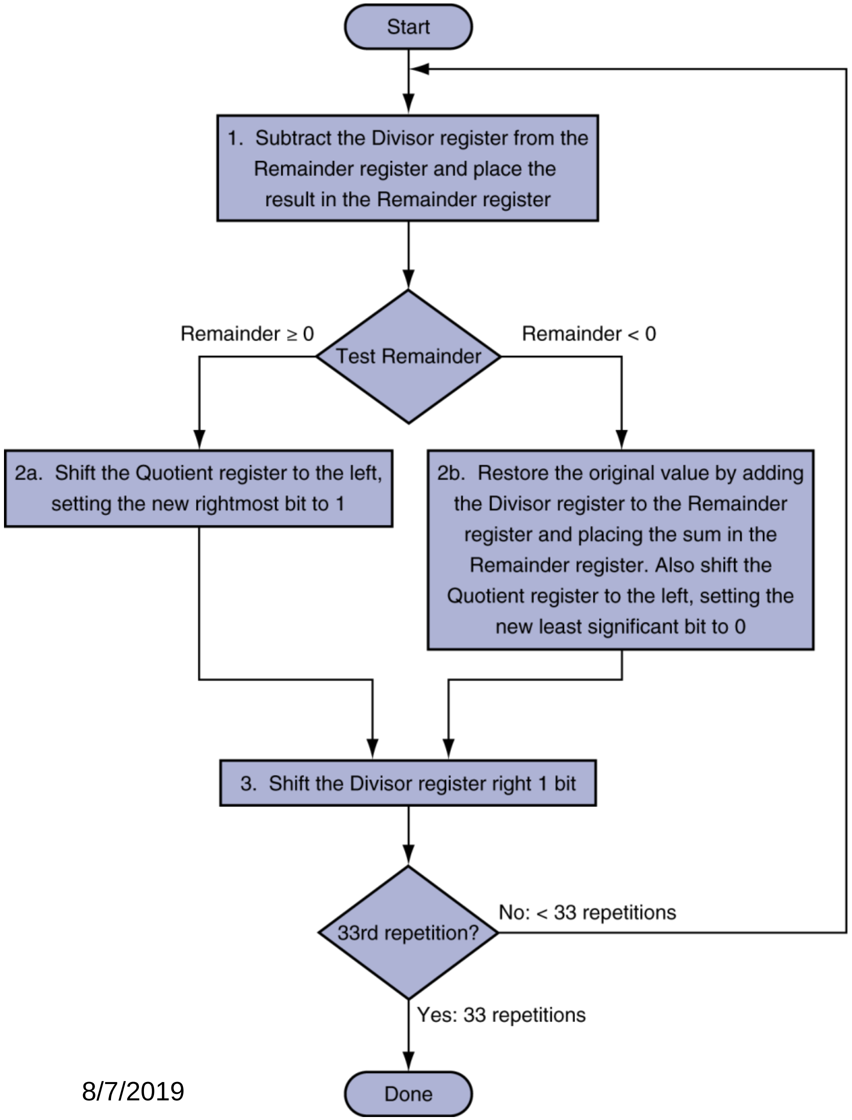
remainder

```
           1001
1000 ) 1001010
      -1000
           10
           101
           1010
          -1000
             10
```

n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware



Start

1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register

Test Remainder

Remainder ≥ 0

Remainder < 0

2a. Shift the Quotient register to the left, setting the new rightmost bit to 1

2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0

3. Shift the Divisor register right 1 bit

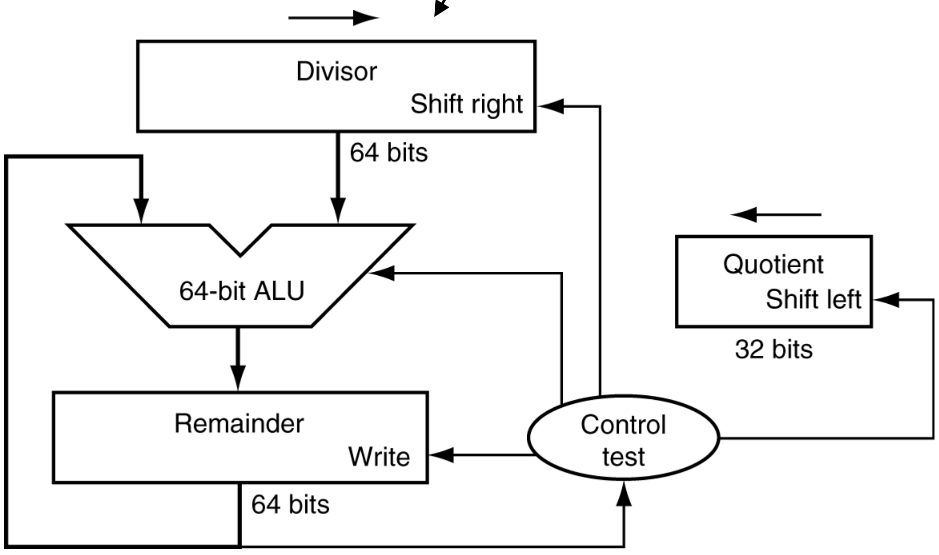
33rd repetition?

No: < 33 repetitions

Yes: 33 repetitions

Done

Initially divisor in left half



Initially dividend

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Still require multiple steps

Multiply

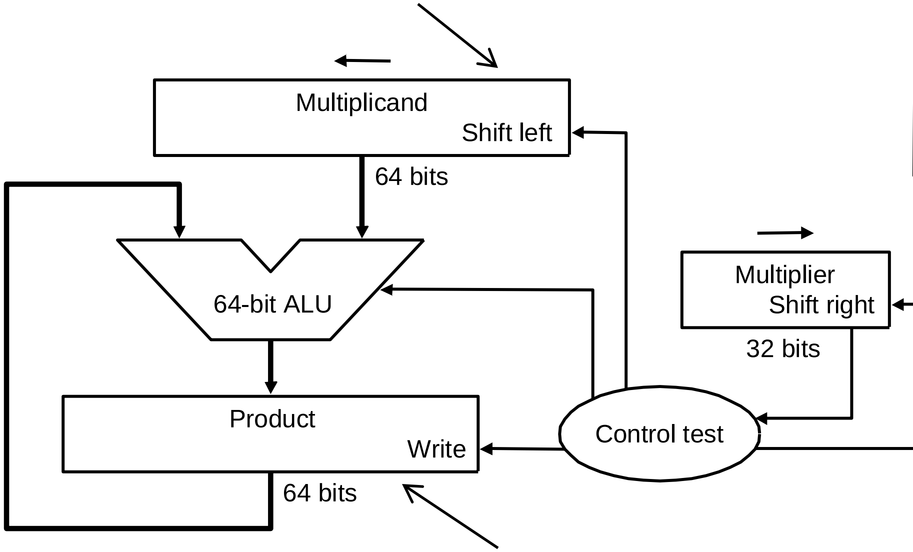
- Grade school shift-add method:

```
Multiplicand        1000
Multiplier        x 1001
                  ------
                   1000
                    0000
                  0000
                1000
              ----------
Product       01001000
```

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - multiplier bit 1 => copy multiplicand (1 x multiplicand)
  - multiplier bit 0 => place 0   (0 x multiplicand)

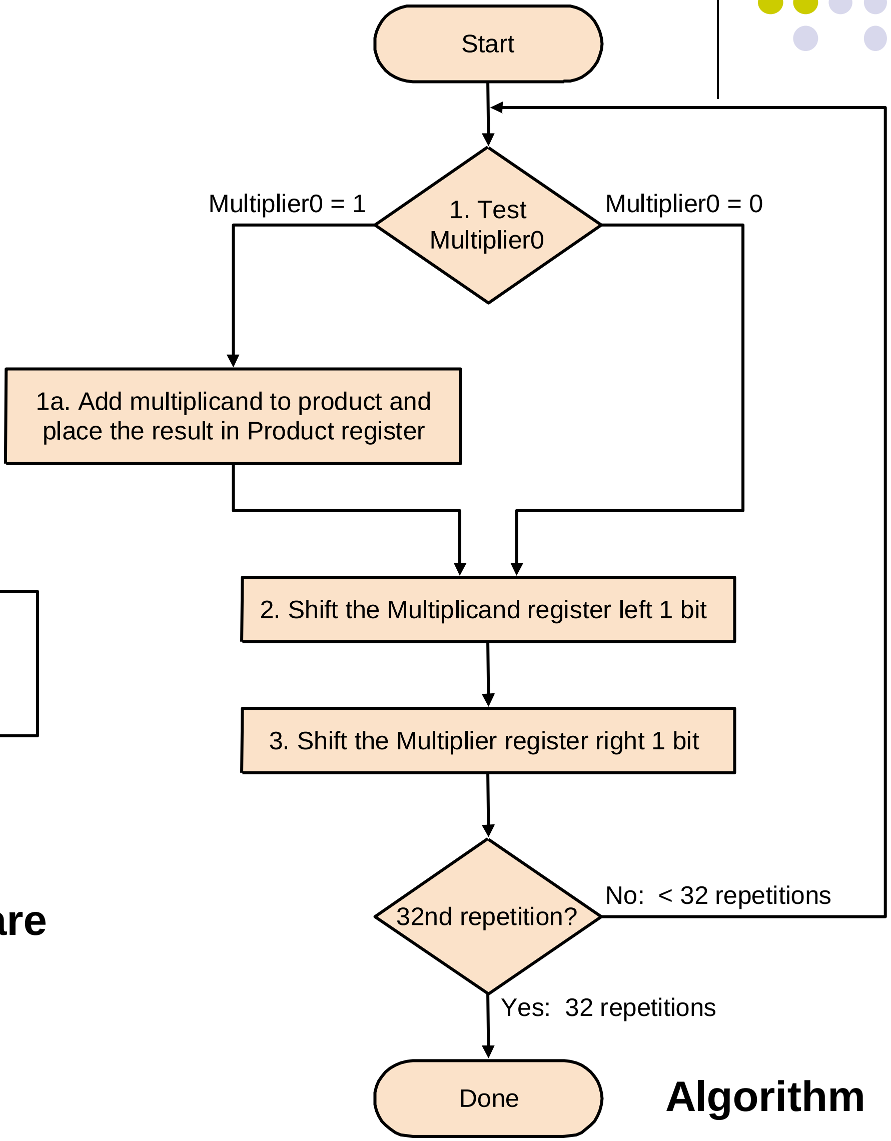
Shift-add Multiplier Version

32-bit multiplicand starts at right half of multiplicand register



Product register is initialized at 0

Multiplicand register, product register, ALU are 64-bit wide; multiplier register is 32-bit wide


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Algorithm

Multiplication

- This is left for further reading

Real numbers

Real numbers

- Decimal real numbers
  - $13.234 = 1*10^{1}+3*10^{0}+2*10^{-1}+3*10^{-2}+4*10^{-3}$
- Binary real numbers
  - $101.111 = 1*2^{2}+1*2^{0}+1*2^{-1}+1*2^{-2}+1*2^{-3}=5.875$
- Decimal to binary
  - 4.68=100.?
  - 100.1010111
  - Just approximately
  - 4.6796875

0.68*2=1.36	1
0.36*2=0.72	0
0.72*2=1.44	1
0.44*2=0.88	0
0.88*2=1.76	1
0.76*2=1.52	1
0.52*2=1.04	1

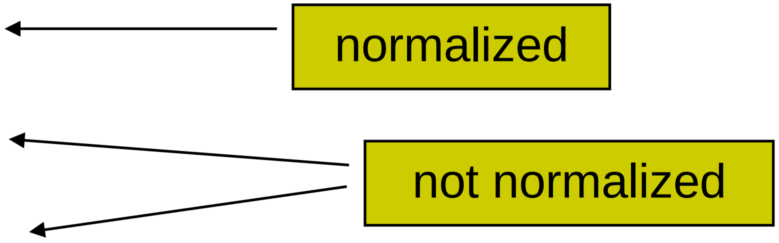
…

Fixed point real numbers

- Representation
  - A number of bits is used to represent the integral part
  - The rest represents the fraction value
- The hardware is less costly
- The precision is not high
- Suitable for some special-purpose embedded processors

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$



normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significant: $1.0 \leq |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    ⇒ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    ⇒ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^{1} \times 1.1_{2} \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_{2}$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_{2}
    - Double: –1 + 1023 = 1022 = 01111111110_{2}
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = 10000001_2 = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^{S} \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision

- Denormal with fraction = 000...0

$$x = (-1)^{S} \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
  - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^2
- 4. Round and renormalize if necessary
  - 1.002×10^2

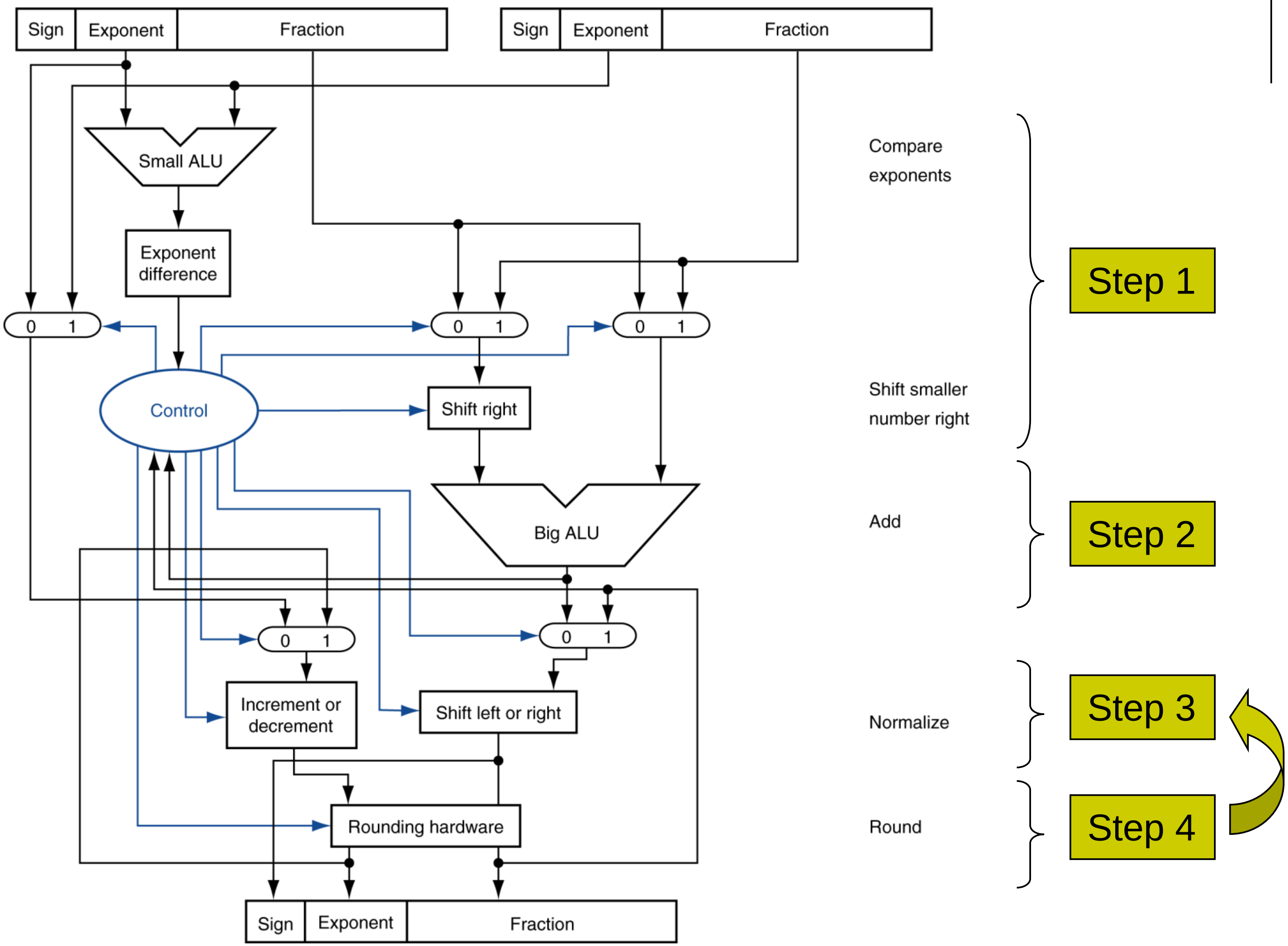
Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



Compare exponents

Shift smaller number right

Step 1

Add

Step 2

Normalize

Step 3

Round

Step 4

Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \;\Rightarrow\; 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

ALU & CPU

Introducing ALU

- ALU: Arithmetic & Logic Unit
  - Performs arithmetic operations
    - Addition
    - Subtraction
  - Performs logic operations
    - AND: A•B
    - OR: $\overline{A}$+B
    - NOT: A
  - Desired operation/result is chosen based on a selection logic.

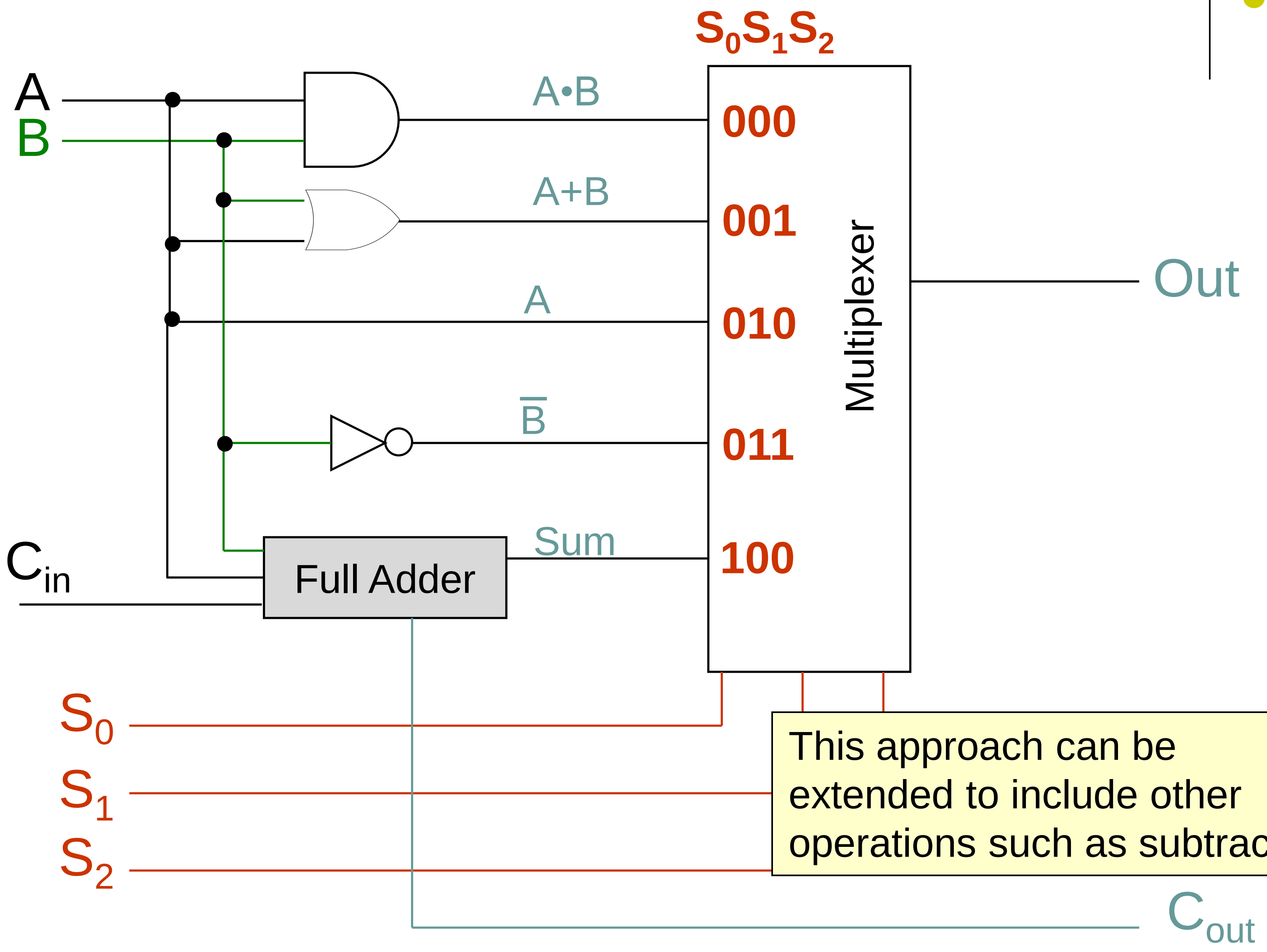
Exercise

- Develop a circuit that accepts 3 inputs (say A, B, and C_{in}) and generates 2 outputs (say Out and C_{out}) depending on 3 control inputs S_1,S_2, and S_3 as shown in the truth table:

S_0	S_1	S_2	Out	C_{out}
0	0	0	A•B	X
0	0	1	A+B	X
0	1	0	A	X
0	1	1	$\overline{B}$	X
1	0	0	$A \oplus B \oplus C_{in}$	$AB + BC_{in} + AC_{in}$

X = Don’t care (can be 1 or 0)

Solution: 1-bit ALU



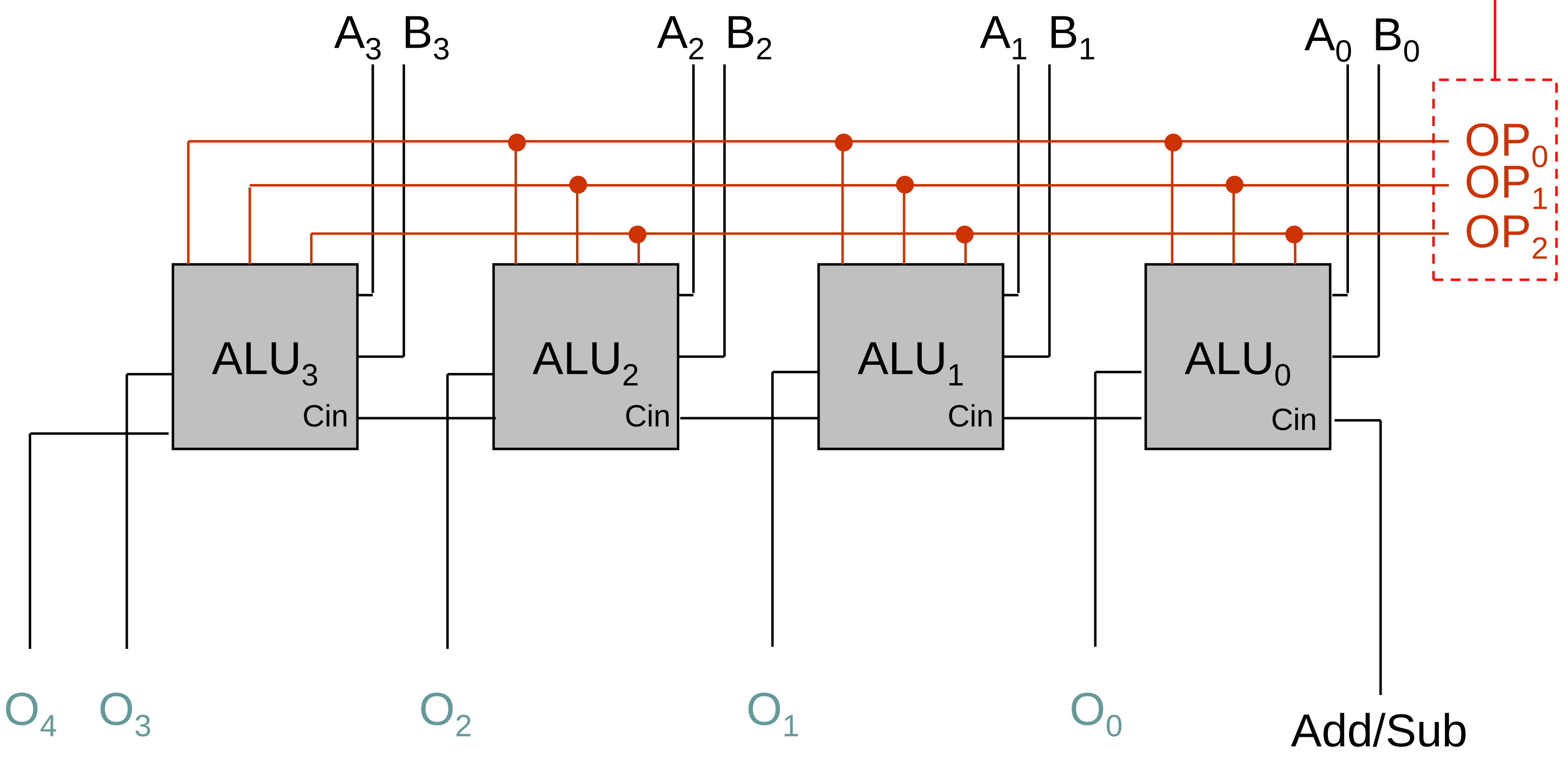
This approach can be extended to include other operations such as subtraction.

n-bit ALU

- n-bit ALU
  - Repeat 1-bit ALU n times
  - All 1-bit ALUs get the same selection lines
  - Carry out (C_{out})from one stage is wired to carry-in of next state
    - Similar to how a ripple carry adder (or subtraction circuit) is wired
  - Performs operation on n-bits *simultaneously*

Example 4-bit ALU

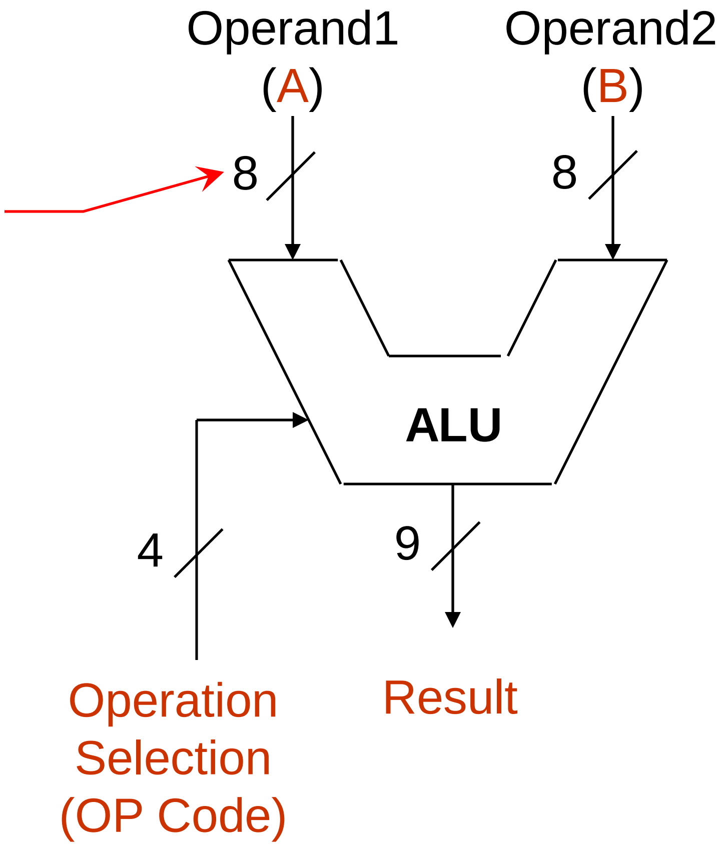
Selection lines simply renamed to operation code (OPCode) lines



ALU Notation

- An ALU is denoted using the following graphical notation.

Indicates number of bits on each line. The size of each operand is typically the same and corresponds to the *word* size.



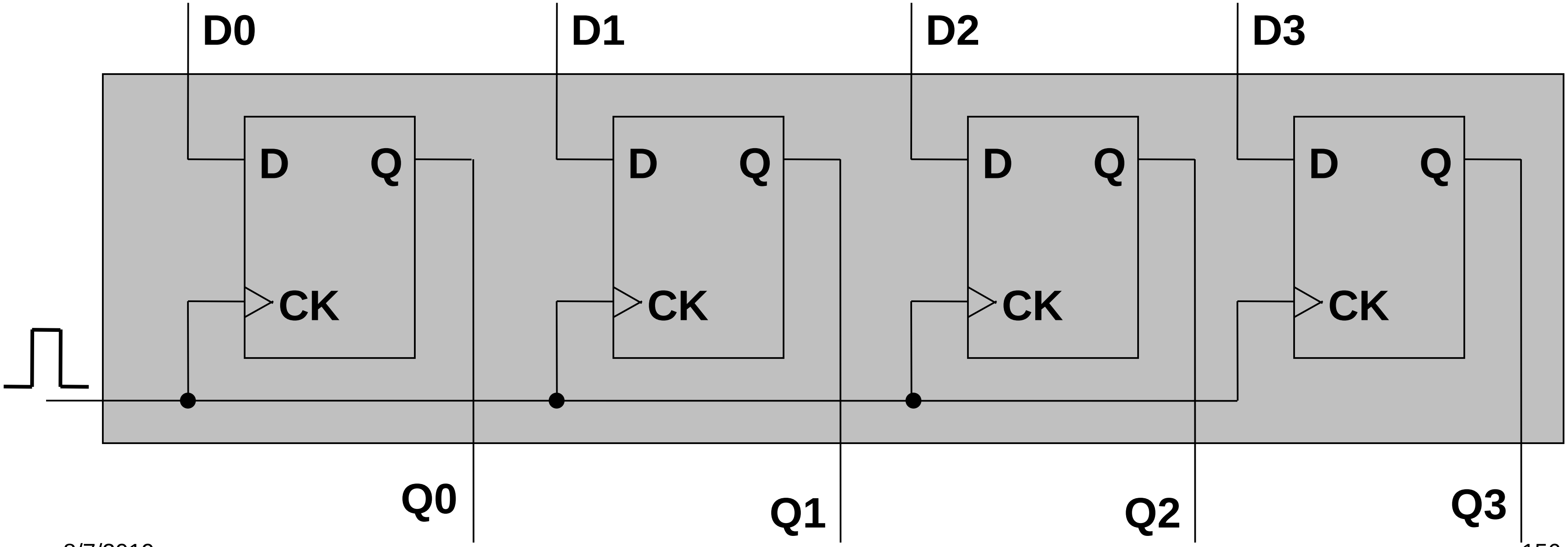
OP Code	Result
0_{10}	A•B
1_{10}	A+B
2_{10}	$\overline{A}$
3_{10}	B
…	…

Thought Experiment

- Where do the inputs to the ALU come from?
  - The two operands
  - The operation the ALU needs to perform
- Similarly where do the outputs from the ALU go?
- Solution: Registers!
  - The fundamental storage units.

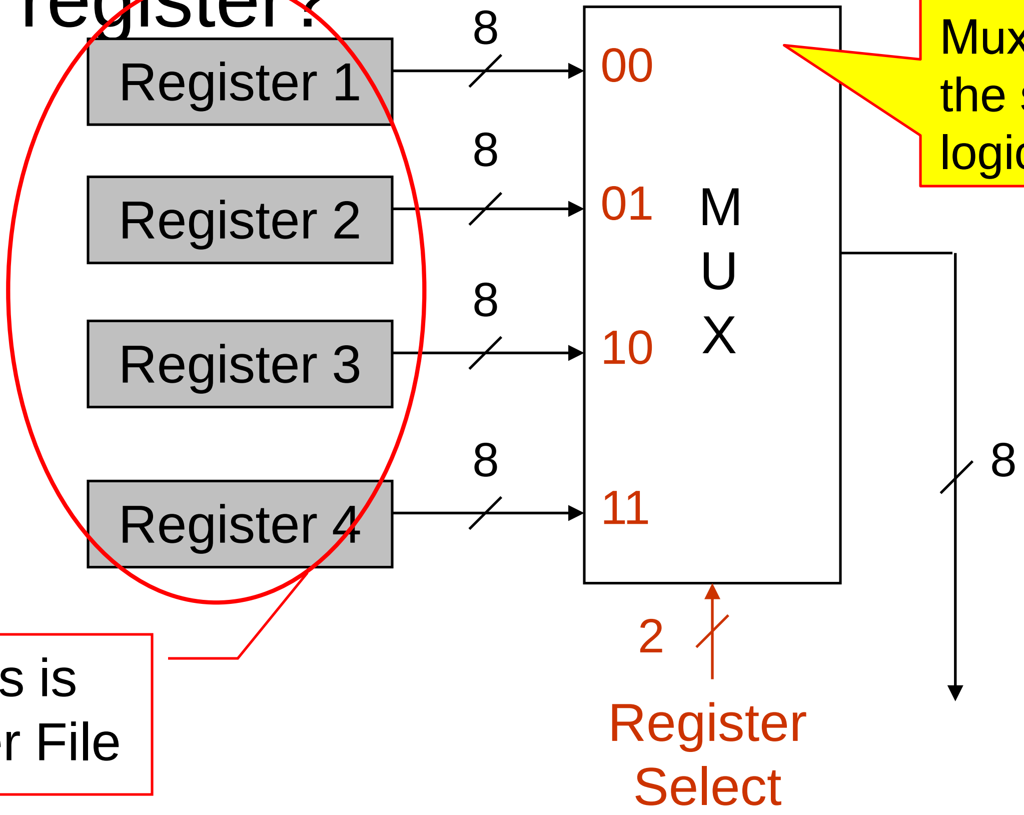
Registers Revisited

- Fixed number of D-Flip Flops to form a Word
  - Corresponding to size of ALU operands!
  - Operate as a single unit
    - Store/Read n-bits at a time



Selecting a Register

- Given n Registers how to read da f specific register?

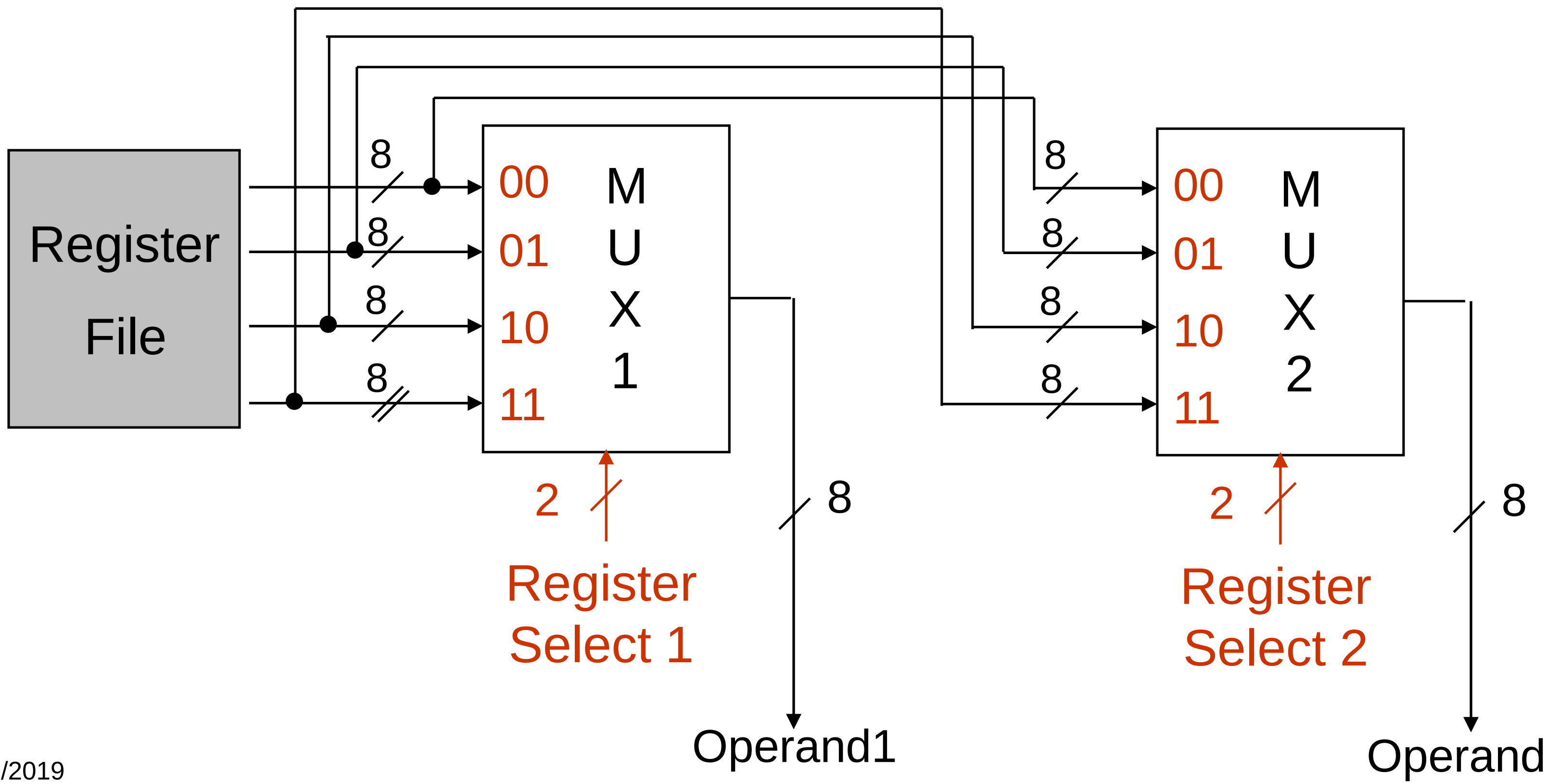


This is not a single Mux but a set of 8 Mux's all having the same selection logic!

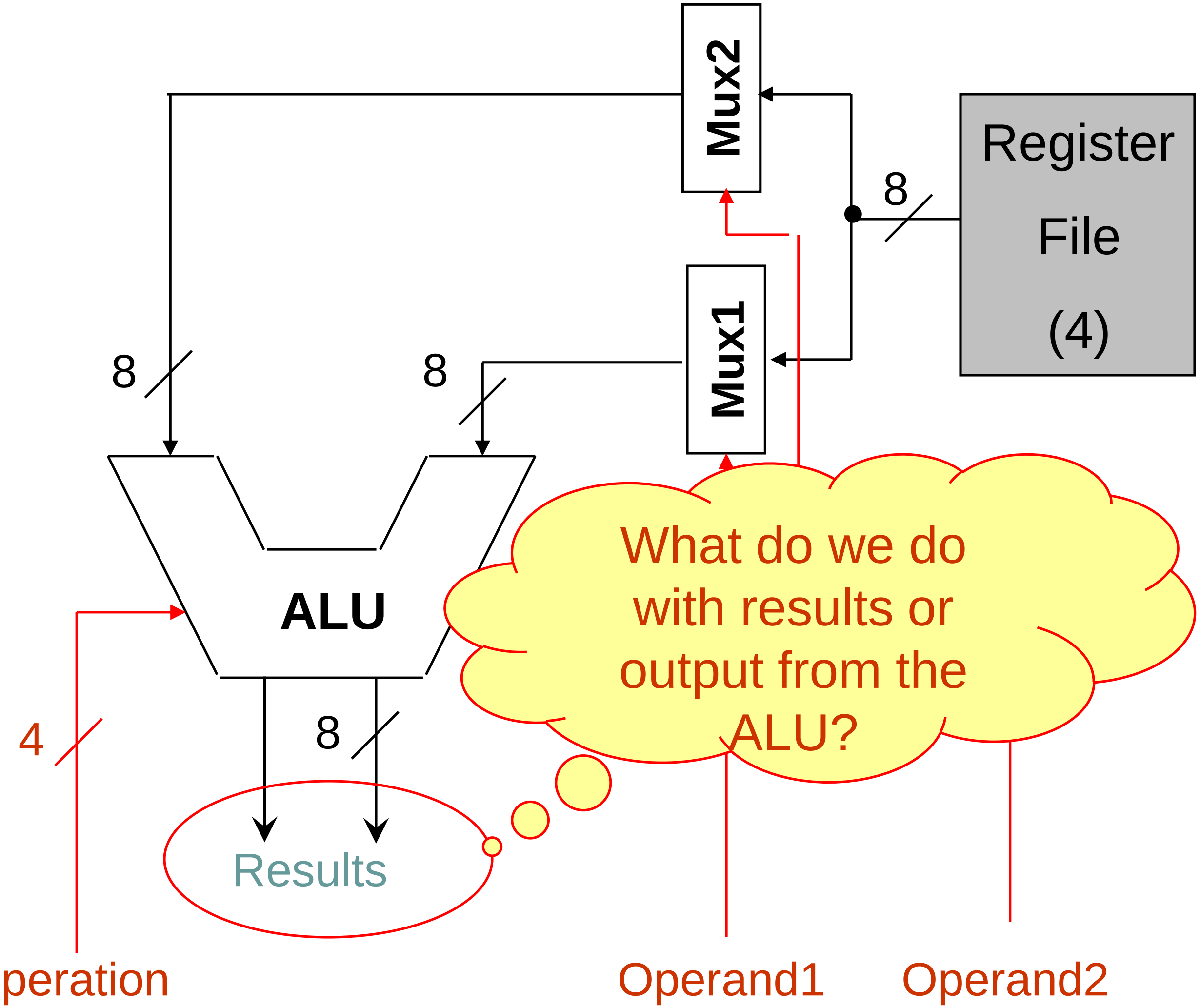
Set of registers is called a Register File

Selecting 2 Registers

- ALU needs 2 operands to work!
  - How to select 2 registers from a Register File?



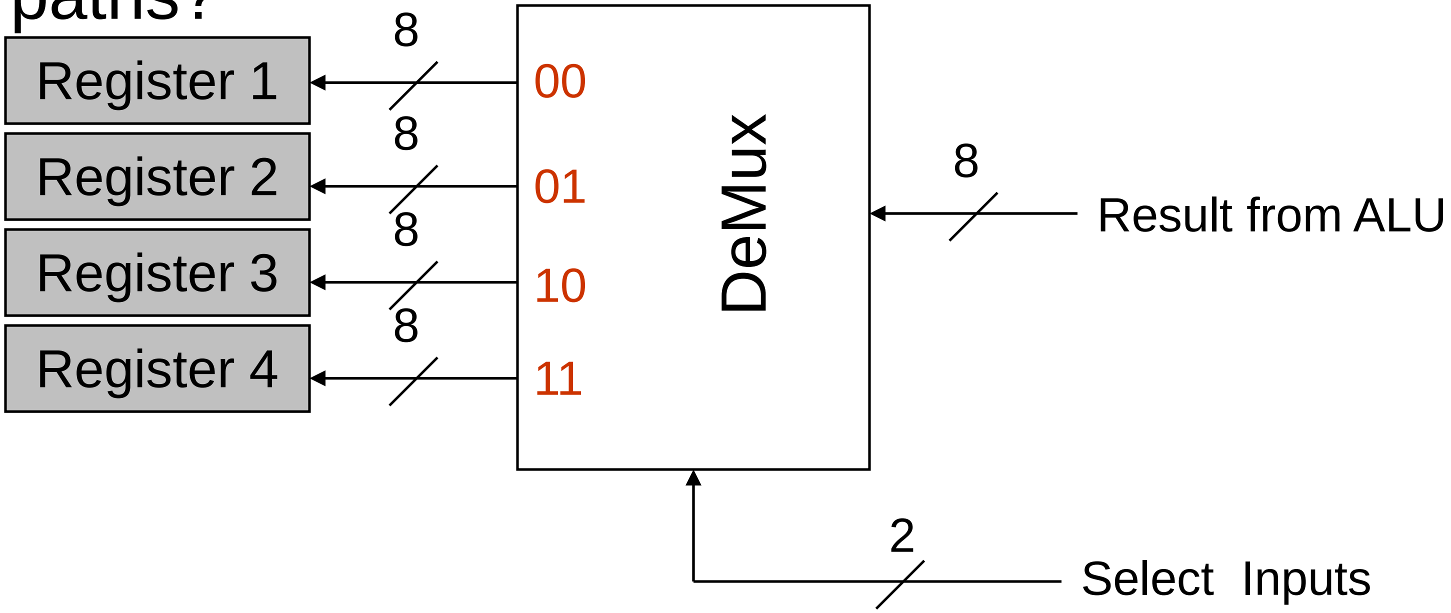
ALU with Inputs



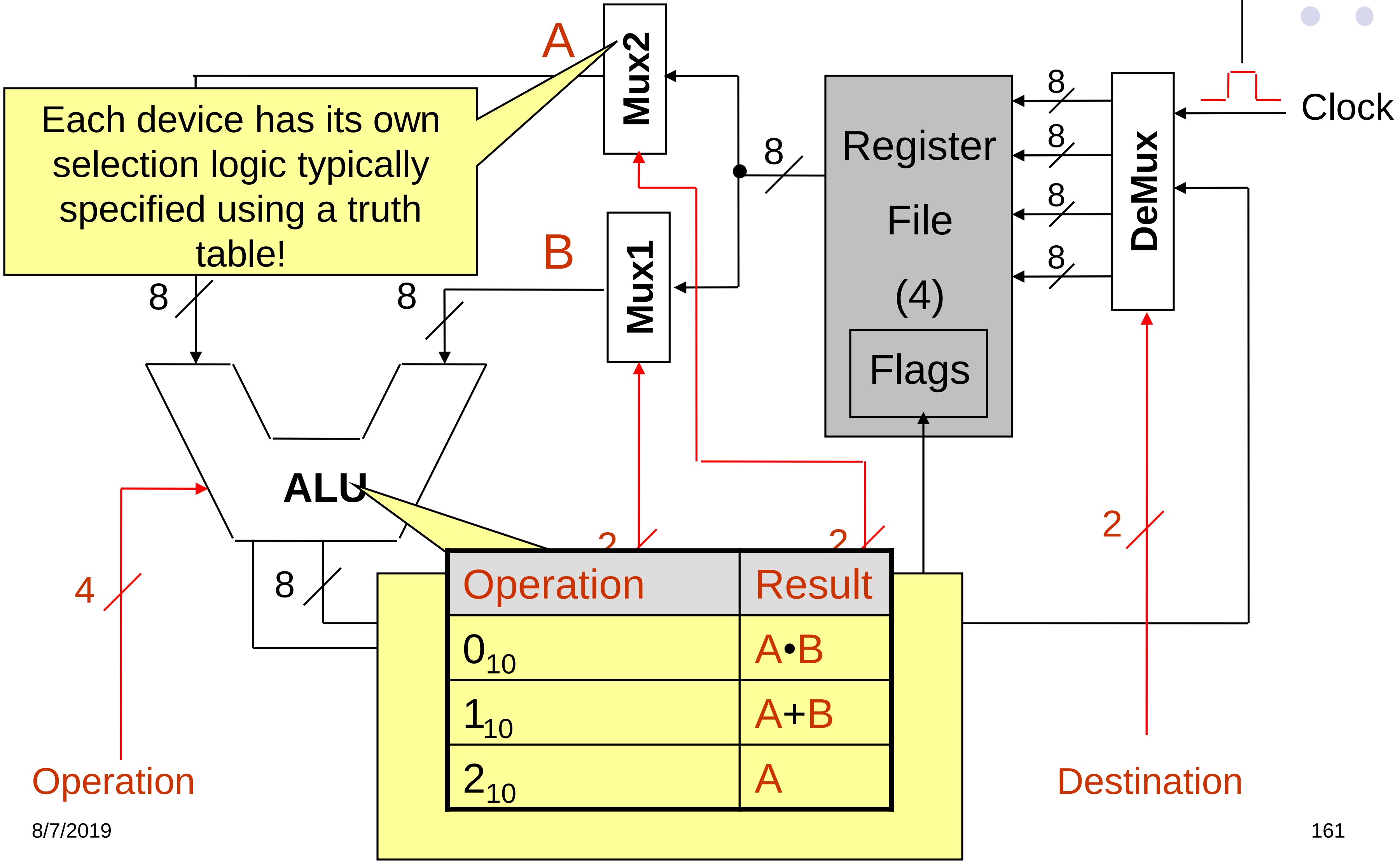
What do we do with results or output from the ALU?

ALU Output

- Solution: Put it into a register!
- OK, so do we do that?
  - That is, Given 1 input how to route it on n different paths?



Handling ALU Output



Each device has its own selection logic typically specified using a truth table!

Operation	Result
0_{10}	A•B
1_{10}	A+B
2_{10}	A

Data path

- The ALU and associated components constitute the Data Path
  - Includes Registers, Multiplexers and any other device associated with ALU operations
  - All operands are typically the same size
    - Register sizes match with size of operand
  - Size of operands are associated with CPU
    - 32-bit processor (ALU uses 32-bit operands)
    - 64-bit processor (ALU uses 64-bit operands)

Handling Constant Values

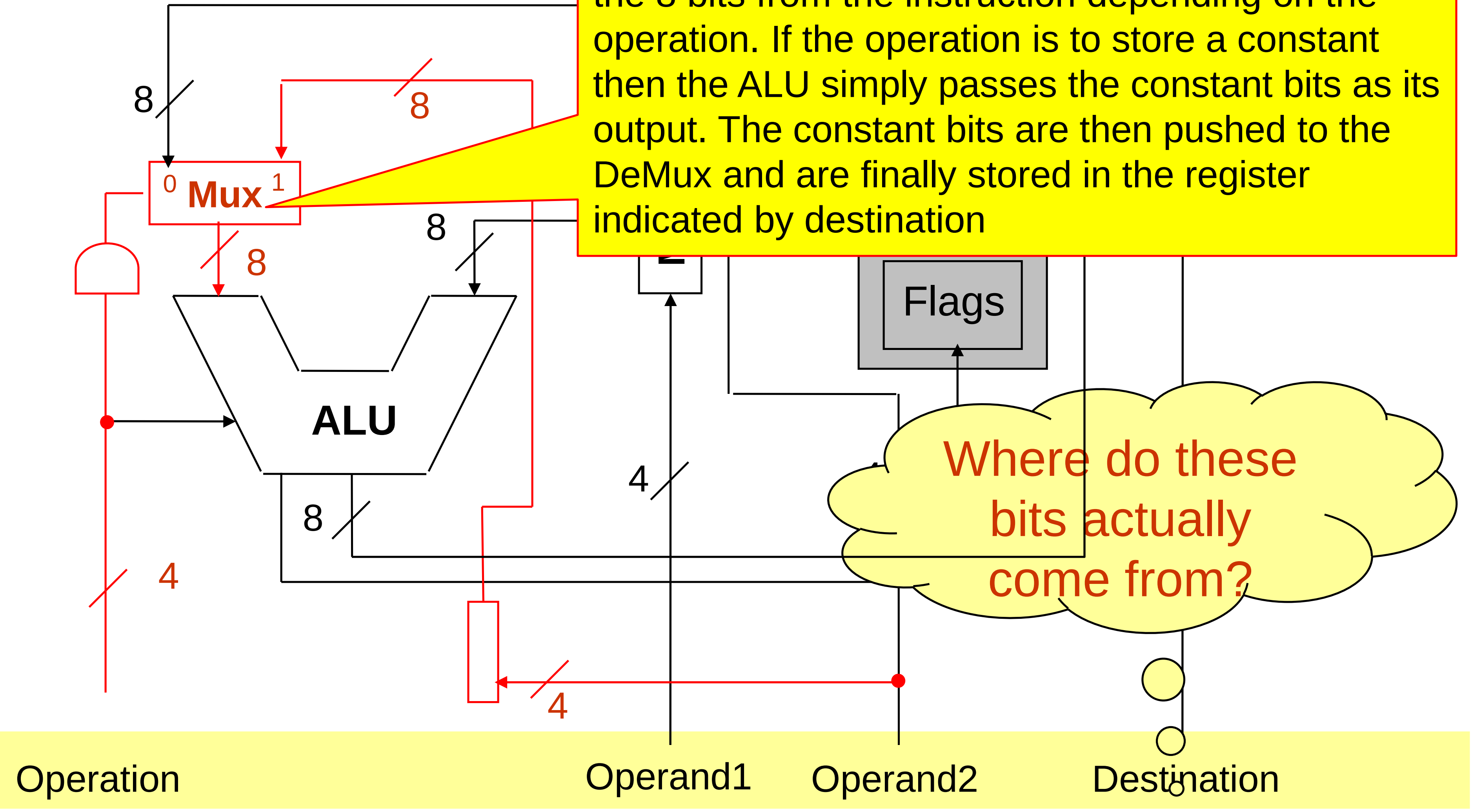
- Earlier data path did not permit initialization of registers with constant values
  - Limited constant values could be achieved using operations supported by ALU.
    - Even that was pretty convoluted!
- Solution: Add instruction to initialize register
  - With a constant value
  - Typically, the constant value is embedded as a part of the instruction.
    - By reusing as many bits as possible for this task

Implementation Strategy

- Fix code for constant value initialization
  - Have to use a code that is not already used by ALU
    - In our case, let’s set it to 1111 for our ALU
  - If initialization is detected, use constant bits in instruction to initialize a register.
    - Need 8-bits to hold constant value for 8-bit CPU
      - For this we can reuse register selection input bits to double up as constant bits in this instruction as we are not using registers for any operation.
    - Need 2-bits to select 1 of 4 destination registers

Handling Constants

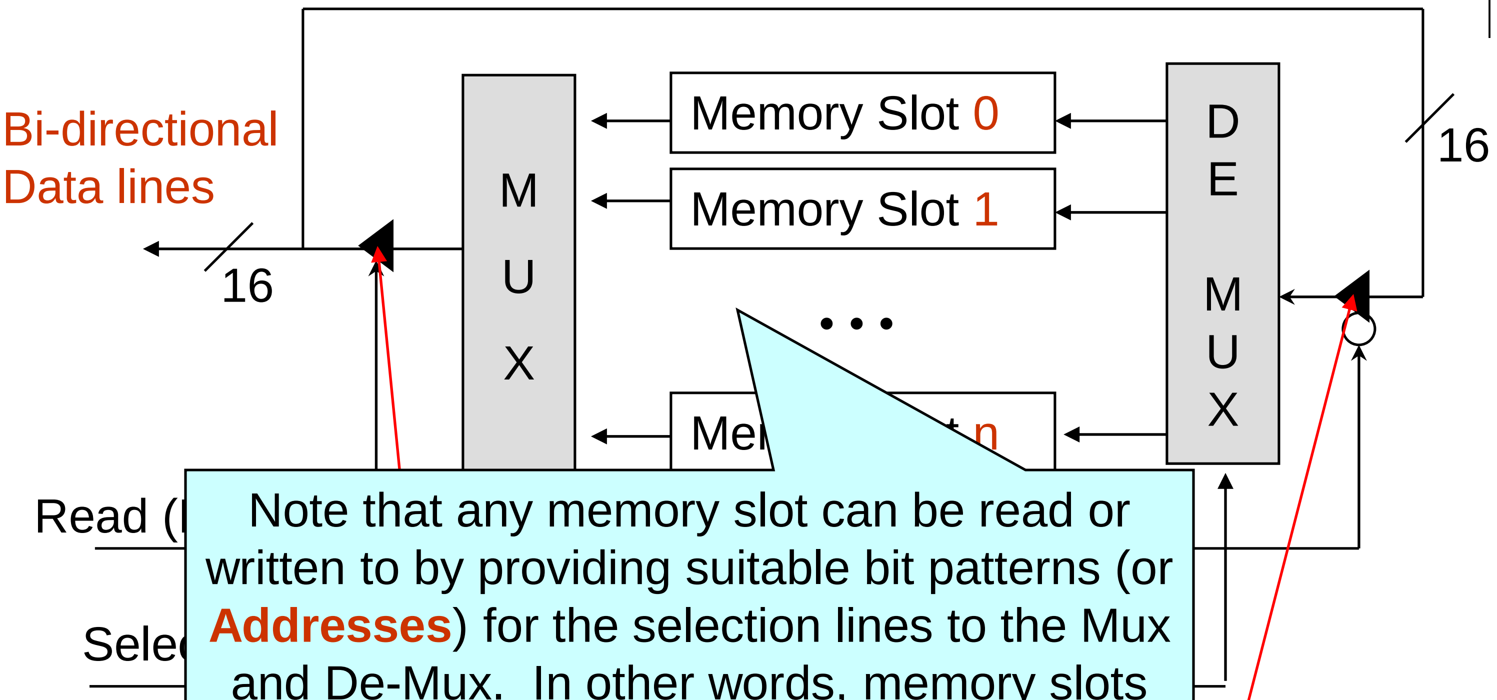
This Mux chooses between data from registers or the 8 bits from the instruction depending on the operation. If the operation is to store a constant then the ALU simply passes the constant bits as its output. The constant bits are then pushed to the DeMux and are finally stored in the register indicated by destination



Where do instructions come from?

- The data path uses a set of bits that constitute an instruction
  - Where do these instruction bits come from?
- Solution: Memory
  - A large collection of words
    - Each word consists of 1 or more bytes (8-bits)
    - Similar in philosophy as a Register File
  - Manufactured using different technology
    - Makes it slower
    - But a whole lot cheaper!

Memory Organization Revisited



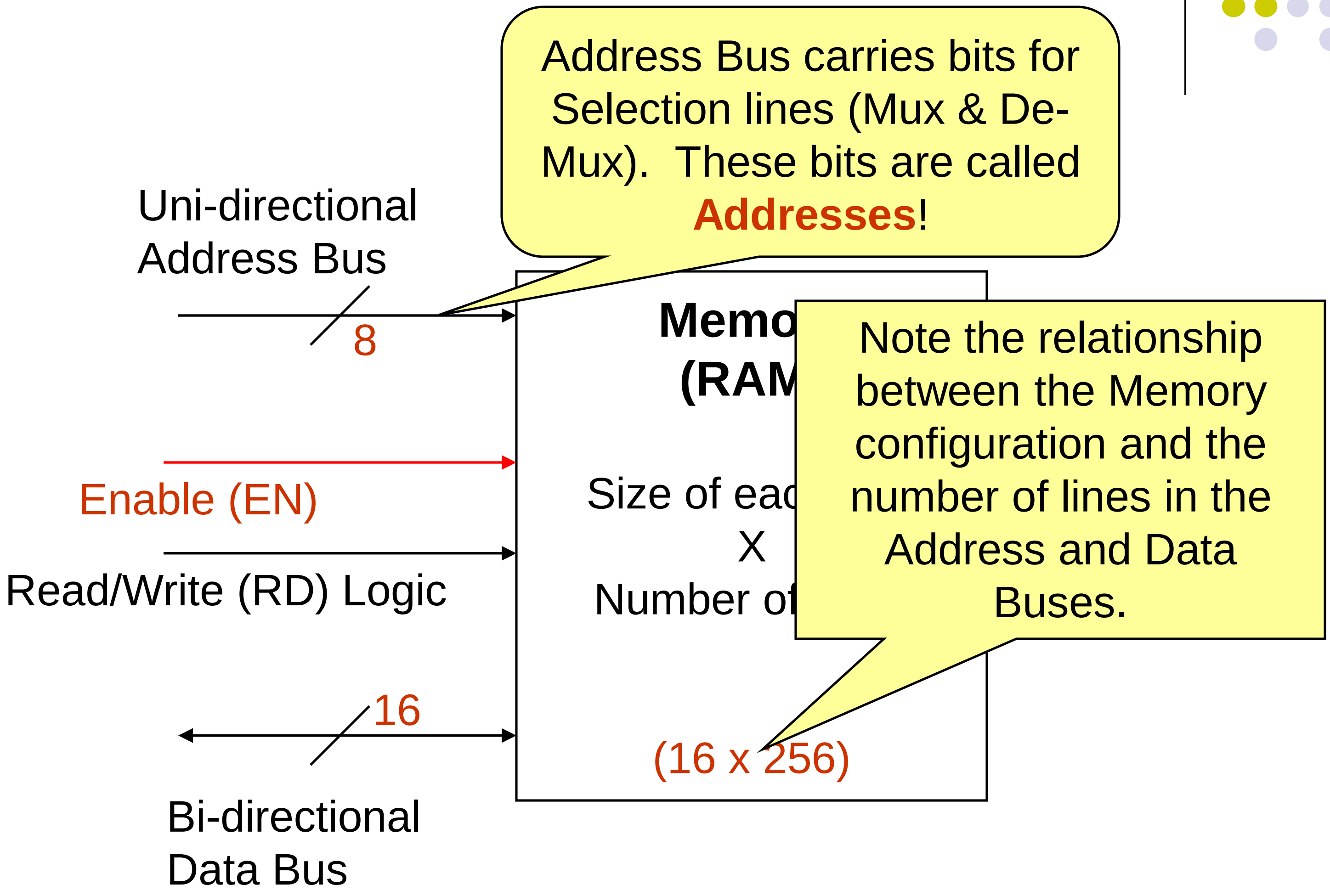
Note that any memory slot can be read or written to by providing suitable bit patterns (or **Addresses**) for the selection lines to the Mux and De-Mux. In other words, memory slots can be Randomly accessed. Consequently, such memory organizations are called Random Access Memory (RAM)

that
me
or
writing (bi-directional)!

Memory-ALU Interconnection

- Typically memory is large in size
  - Gigabytes in size these days
- Cannot be packed into the CPU
  - Cost prohibitive
- Memory is manufactured separately
- Interconnected with the CPU using **Buses**
  - Buses are long wires interconnecting devices
    - Particularly ALU Data Path & Memory
  - Buses for memory
    - Address Bus: Selection lines for Mux and De-Mux
    - Data Bus: Bits to be written into memory locations.

Abstract Notation for Memory



Address Bus carries bits for Selection lines (Mux & De-Mux). These bits are called **Addresses**!

Uni-directional Address Bus

8

Memo
(RAM

Note the relationship between the Memory configuration and the number of lines in the Address and Data Buses.

Enable (EN)

Size of eac
X
Number of

Read/Write (RD) Logic

16

(16 x 256)

Bi-directional Data Bus

Using Memory

- Memory has 3 primary inputs
  - Address Bus carrying address of memory slot
    - Indicates which memory slot to read or write
  - Data Bus (The actual data bits to be stored or read)
  - Control signals (Read/Write & Enable)
- How to wire these inputs & outputs to the ALU data path?
  - Design requirements
    - Need to store output from ALU
    - Need to load data from memory into ALU
    - Need to load instructions from memory into ALU

We need go provide Addresses to the Memory unit in order to do these operations!

Tackling Addresses

- Address is used to select a memory slot
  - For fetching instructions
    - In an repetitive manner
    - Typically from consecutive locations
      - Think of it as an Arra
    - Need to some
      - Typically don
        intermediate
  - For reading/w
    - Address depends on the instruction at hand
      - The instruction typically needs to identify the address to read or write.

```
// Assume each instruction is 16-bits long
short memory[256];
for (int address = 0; (address < 256); address++)  {
            instruction = memory[address];
            process(instruction);
}
```

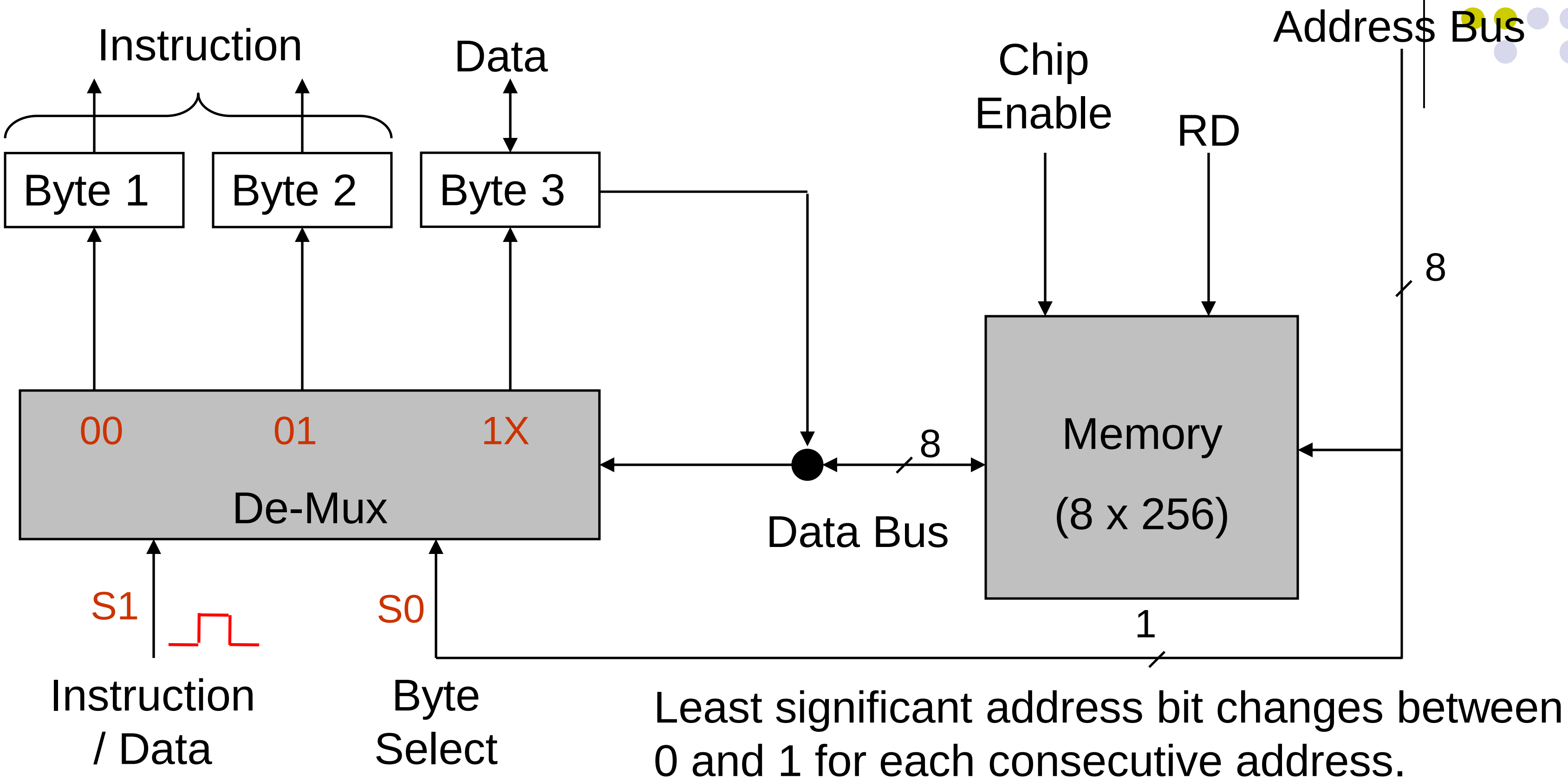
Address Generation Logic Circuit

- Issues to consider & solutions
  - Address needs to be generated by adding 1
    - Use a ripple carry adder to add
  - Addresses need to be stored before/after add
    - Use a register
  - Need alternate between address for instructions and address for storing/reading data
    - Use a multiplexer for choosing
    - Use a Clock to drive the selection lines of this multiplexer
      - Select address for instruction first (Clock = 0).
      - Select address for reading/writing data next (Clock = 1).

Catch!

- Our data path is not symmetric
  - Instructions are 16-bits wide
  - Data or Registers are 8-bits wide
  - How do we design a memory module that can
    - Provide 16-bits first
    - Provide 8-bits next
- It is going to take some work
  - Use the lowest denominator memory module
    - One than can provide 1 byte at a time
    - Assemble bytes together to make instructions

Working with Bytes…



Least significant address bit changes between 0 and 1 for each consecutive address.

When the clock is low (S1=0) the De-Mux places the bytes read from Memory into `Byte1` *and* `Byte2` *depending on S0 (so S0, the least significant bit from the address bus) which switches between 0 and 1 when clock is low to fetch two bytes from memory.. When the clock is high (S1=1), the De-Mux ignores s0 and places the data read from memory into* `Byte 3`*.*

Quiz?

- Any questions?
- How the CPU can communicate with other devices, such as a keyboard and a modem?