Chapter 3

Arithmetic for Computers

Arithmetic for Computers

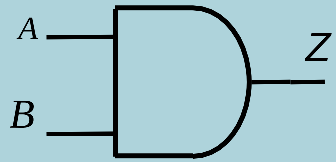
- Introduction
- Gates, Truth Tables, and Logic Equations
- Combinational Logic
- Constructing a Basic Arithmetic Logic Unit
- Addition and Subtraction
- Multiplication
- Division
- Floating Point
- Fallacies and Pitfalls

Boolean Algebra

- Boolean algebra is the basic math used in digital circuits and computers.
- A Boolean variable takes on only 2 values: {0,1} , {T,F}, {Yes, No}, etc.
- There are 3 fundamental Boolean operations:
  - AND, OR, NOT

Fundamental Boolean Operations

AND

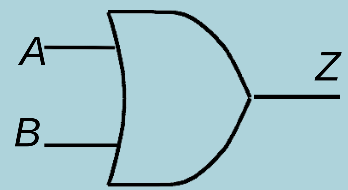


Z=A*B (AB)

Truth Table

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR

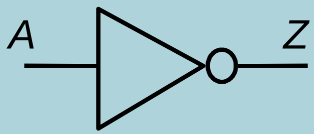


Z=A+B

Truth Table

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

NOT



Z=Ā

Truth Table

A	Z
0	1
1	0

Boolean Algebra

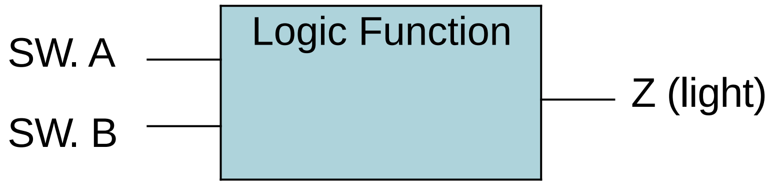
- A *truth table* specifies output signal logic values for every possible combination of input signal logic values
- In evaluating Boolean expressions, the *Operation Hierarchy* is:
  1) NOT 2) AND 3) OR.
  Order can be superseded using (…)
- *Example:* $A=T, B=F, C=T, D=T$
  - What is the value of $Z=(\overline{A}+B)\cdot(C+\overline{B}\cdot D)$?

$$Z=(\overline{T}+F)\cdot(C+\overline{B}\cdot D)=(F+F)\cdot(C+\overline{B}\cdot D)$$

$$=F\cdot(C+\overline{B}\cdot D)=F$$

Deriving Logic Expressions From Truth Tables

Light must be ON when both switches A and B are OFF, or when both of them are ON.



Truth Table:

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

- *What is the Boolean expression for Z?*

$$Z = \bar{A}.\bar{B} + A.B$$

Minterms and Maxterms

- Minterms
  - AND term of all input variables
  - For variables with value 0, apply complements
- Maxterms
  - OR factor with all input variables
  - For variables with value 1, apply complements

A	B	Z	Minterms	Maxterms
0	0	1	$\bar{A}.\bar{B}$	$A+B$
0	1	0	$\bar{A}.B$	$A+\bar{B}$
1	0	0	$A.\bar{B}$	$\bar{A}+B$
1	1	1	AB	$\bar{A}+\bar{B}$

Minterms and Maxterms

- A function with n variables has 2^n minterms (and Maxterms) – exactly equal to the number of rows in truth table
- Each minterm is true for exactly one combination of inputs
- Each Maxterm is false for exactly one combination of inputs

A	B	Z	Minterms	Maxterms
0	0	1	$\bar{A}.\bar{B}$	$A+B$
0	1	0	$\bar{A}.B$	$A+\bar{B}$
1	0	0	$A.\bar{B}$	$\bar{A}+B$
1	1	1	AB	$\bar{A}+\bar{B}$

Equivalent Logic Expressions

- Two <u>equivalent</u> logic expressions can be derived from Truth Tables:

1) *Sum-of-Products* (SOP) expression:
   - Several AND terms OR'd together, e.g.

$$AB\overline{C} + \overline{A}B\overline{C} + ABC$$

2) *Product-of-Sums* (POS) expression:
   - Several OR terms AND'd together, e.g.

$$(\overline{A} + \overline{B} + C)(A + \overline{B} + C)$$

Rules for Deriving SOP Expressions

1) Find each row in TT for which output is 1 (rows 1 & 4)
2) For those rows write a minterm of all input variables.
3) OR together all minterms found in (2)

Such an expression is called a *Canonical* SOP

A	B	Z	Minterms	Maxterms
0	0	1	$\bar{A}.\bar{B}$	$A+B$
0	1	0	$\bar{A}.B$	$A+\bar{B}$
1	0	0	$A.\bar{B}$	$\bar{A}+B$
1	1	1	AB	$\bar{A}+\bar{B}$

$$Z = \overline{A}\,\overline{B} + AB$$

Rules for Deriving POS Expressions

1) Find each row in TT for which output is 0 (rows 2 & 3)
2) For those rows write a maxterm
3) AND together all maxterm found in (2)

Such an expression is called a *Canonical* POS.

0	0	1	$\bar{A}.\bar{B}$	$A+B$
0	1	0	$\bar{A}.B$	$A+\bar{B}$
1	0	0	$A.\bar{B}$	$\bar{A}+B$
1	1	1	AB	$\bar{A}+\bar{B}$

$$Z = (A+\bar{B})(\bar{A}+B)$$

CSOP and CPOS

- Canonical SOP: $Z = \overline{A}\,\overline{B} + AB$
- Canonical POS: $Z = (A + \overline{B})(\overline{A} + B)$
- Since they represent the same truth table, they should be identical

Verify that $Z = \overline{A}\,\overline{B} + AB \equiv (A + \overline{B})(\overline{A} + B)$

- *CPOS and CSOP expressions for the same TT are logically equivalent. Both represent the same information.*

Activity 1

Derive SOP and POS expressions for the following TT.

A	B	Carry
0	0	0
0	1	0
1	0	0
1	1	1

Boolean Identities

- Useful for simplifying logic equations.

	(a)	(b)
1	$\overline{\overline{A}} = A$	$\overline{\overline{A}} = A$
2	$A + \text{false} = A \quad (A + 0 = A)$	$A \cdot \text{true} = A \quad (A \cdot 1 = A)$
3	$A + \text{true} = \text{true} \quad (A + 1 = 1)$	$A \cdot \text{false} = \text{false} \quad (A \cdot 0 = 0)$
4	$A + A = A$	$A \cdot A = A$
5	$A + \overline{A} = \text{true} \quad (A + \overline{A} = 1)$	$A \cdot \overline{A} = \text{false} \quad (A \cdot \overline{A} = 0)$
6	$A + B = B + A$	$A \cdot B = B \cdot A$
7	$A + B + C = (A + B) + C = A + (B + C)$	$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C$
8	$A \cdot (B + C) = A \cdot B + A \cdot C$	$A + B \cdot C = (A + B)(A + C)$
9	$\overline{A + B} = \overline{A} \cdot \overline{B}$	$\overline{A \cdot B} = \overline{A} + \overline{B}$
10	$A \cdot B + A \cdot \overline{B} = A$	$(A + B)(A + \overline{B}) = A$
11	$A + A \cdot B = A$	$A(A + B) = A$
12	$A(\overline{A} + B) = A \cdot B$	$A + \overline{A} \cdot B = A + B$
13	$A \cdot B + \overline{A} \cdot C + B \cdot C = A \cdot B + \overline{A} \cdot C$	$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C)$

Duals

Boolean Identities

- The right side is the dual of the left side
  1. Duals formed by replacing

AND	→	OR
OR	→	AND
0	→	1
1	→	0

  2. The dual of any true statement in Boolean algebra is also a true statement.

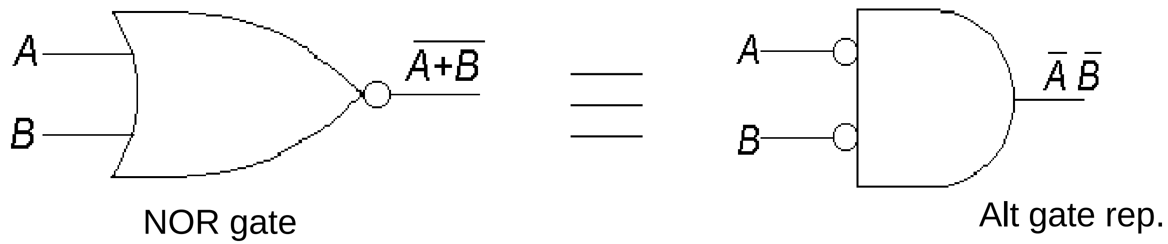
     1 . 1 = 1: "true and true evaluates to true"

     0 + 0 = 0: "false or false evaluates to false"

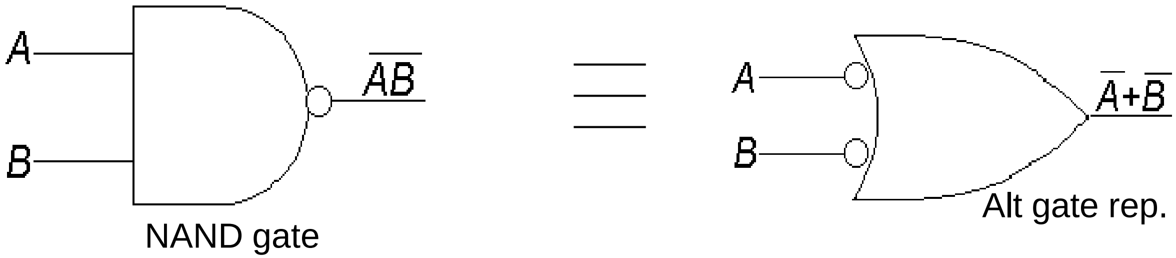
Boolean Identities

- DeMorgan's laws very useful: 9a and 9b

$$\overline{A+B} = \overline{A} \cdot \overline{B}$$



$$\overline{AB} = \overline{A} + \overline{B}$$



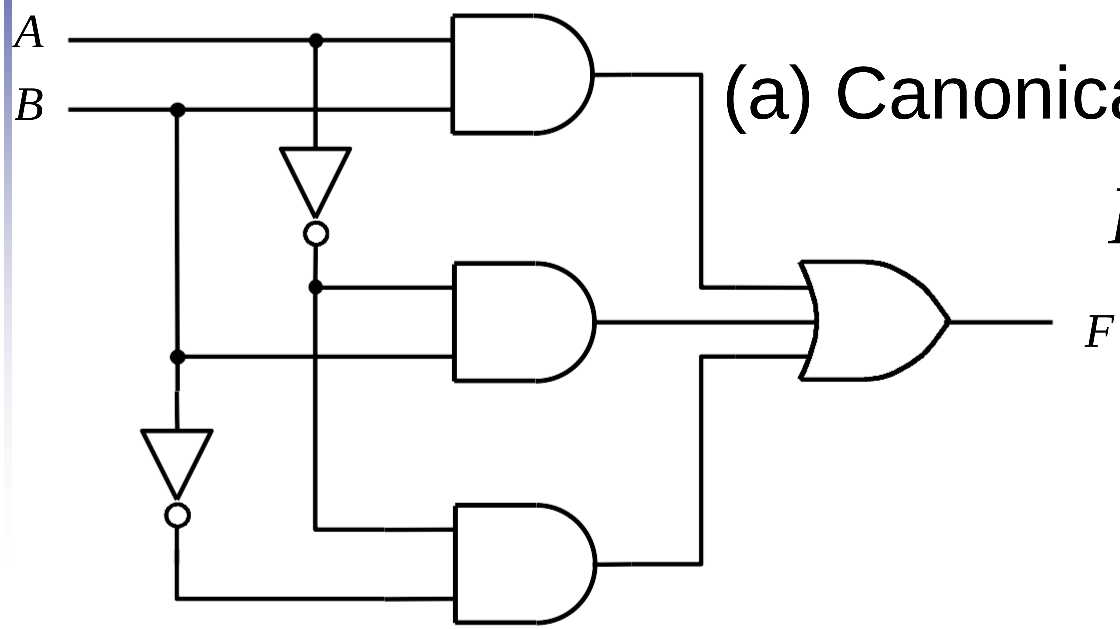
Activity 2

Proofs of some Identities:

12b: $A + \overline{A}B = A + B$

13a: $AB + \overline{A}C + BC = AB + \overline{A}C$

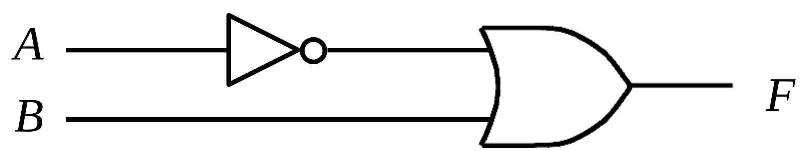
Simplifying Logic Equations – Why?



(a) Canonical sum-of-products

$$F = AB + \overline{A}B + \overline{A}\,\overline{B}$$

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



(b) Minimal-cost realization

Simplifying Logic Equations

- Simplifying logic expressions can lead to using smaller number of gates (parts) to implement the logic expression
- Can be done using
  - Boolean Identities (algebraic)
  - Karnaugh Maps (graphical)
- A *minimum SOP* (MSOP) expression is one that has no more AND terms or variables than any other equivalent SOP expression.
- A *minimum POS* (MPOS) expression is one that has no more OR factors or variables than any other equivalent POS expression.
- There may be several MSOPs of an expression

Example of Using Boolean Identities

- Find an MSOP for

$$F = \overline{X}W + Y + \overline{Z}(Y + \overline{X}W)$$

$$= \overline{X}W + Y + \overline{Z}Y + \overline{Z}\,\overline{X}W$$

$$= \overline{X}W(1+\overline{Z}) + Y(1+\overline{Z})$$

$$= \overline{X}W + Y$$

Activity 3

- Find an MSOP for

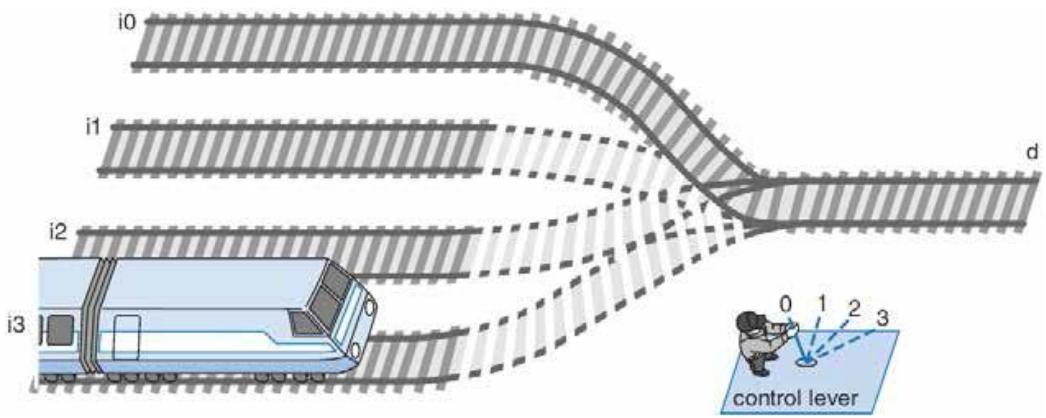
$$
\begin{aligned}
F &= \overline{W}XYZ + WXY\overline{Z} + WXYZ \\
&= XYZ(\overline{W} + W) + WXY(\overline{Z} + Z) \\
&= XYZ(1) + WXY(1) \\
&= XYZ + WXY \\
&= XY(Z + W)
\end{aligned}
$$

Digital Circuit Classification

- Combinational circuits
  - Output depends only on the current combination of circuit inputs
  - Same set of inputs will always produce the same outputs
  - Consists of AND, OR, NOR, NAND, and NOT gates
- Sequential circuits
  - Output depends on the current inputs and state of the circuit (or past sequence of inputs)
  - Memory elements such as flip-flops and registers are required to store the "state"
  - Same set of input can produce completely different outputs

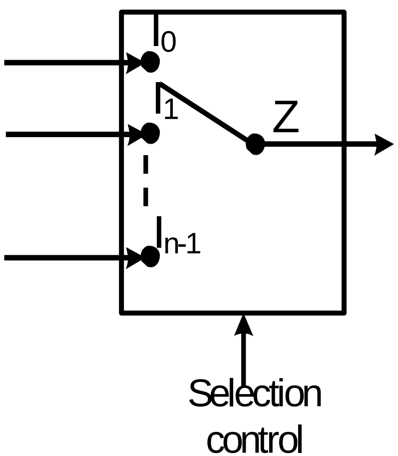
Multiplexor

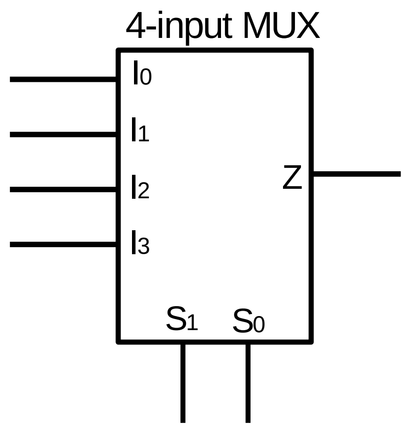
- A multiplexor (MUX) selects data from one of *N* inputs and directs it to a single output, just like a railyard switch
  - 4-input Mux needs 2 select lines to indicate which input to route through
  - N-input Mux needs $\log_2(N)$ selection lines



Multiplexor

- An example of 4-input Mux





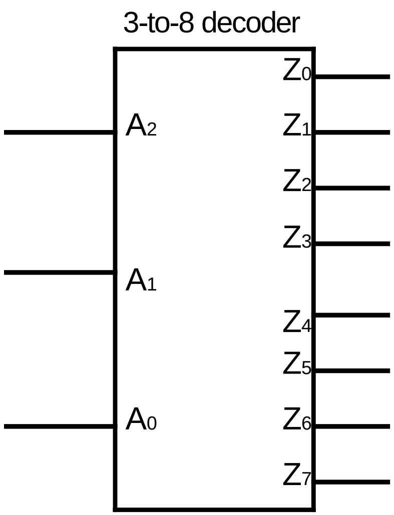
Functional block diagram

S_1	S_0	Z
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Truth Table

Decoder

- A decoder is a circuit that decodes an *N*-bit code.
- It activates an appropriate output line as a function of the applied *N*-bit input code



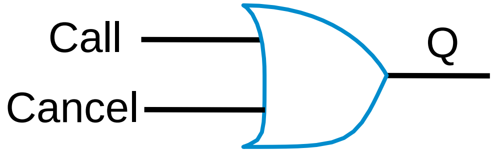
Functional block diagram

Truth Table

A_2	A_1	A_0	Z_0	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

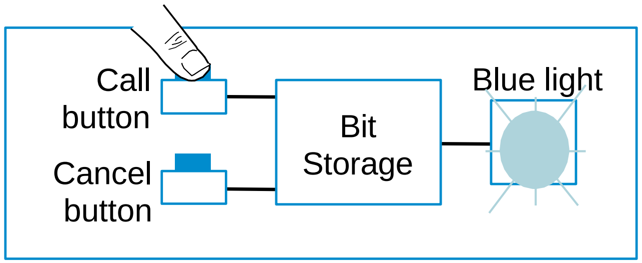
Why Bit Storage ?

- Flight attendant call button
  - Press call: light turns on
    - ***Stays on*** after button released
  - Press cancel: light turns off
  - Logic gate circuit to implement this?

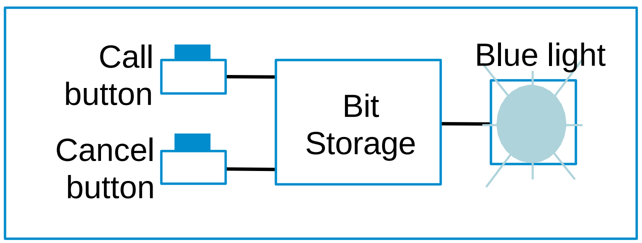


Doesn't work. Q=1 when Call=1, but doesn't stay 1 when Call returns to 0

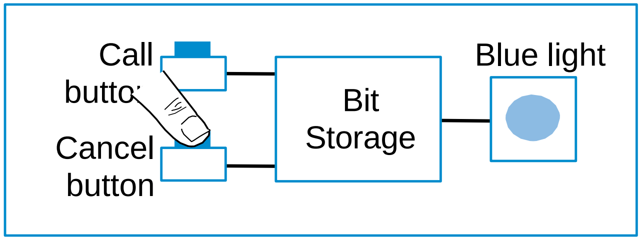
Need some form of "memory" in the circuit



1. Call button pressed – light turns on



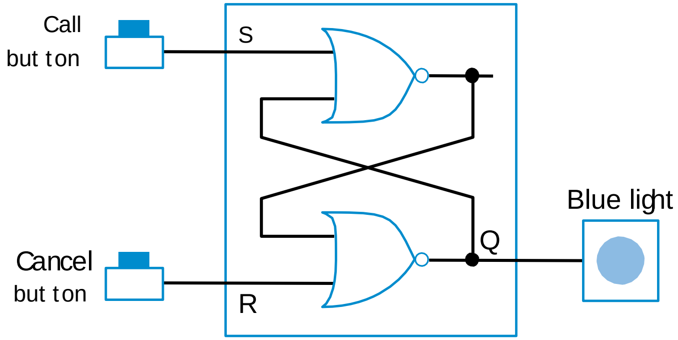
*2. Call button released – light **stays on***



3. Cancel button pressed – light turns off

Bit Storage Using SR Latch

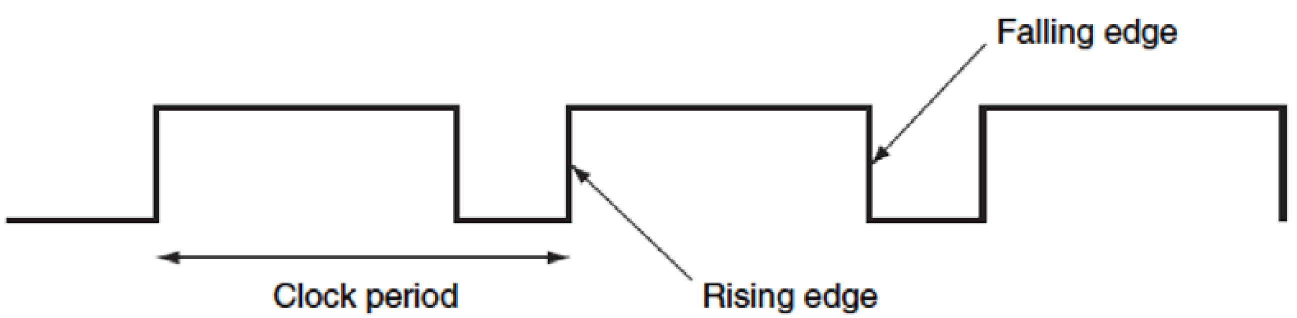
- Simplest memory elements are Latch and Flip-Flops
- SR (set-reset) latch is an ***un-clocked*** latch
  - Output Q=1 when S=1, R=0 (set condition)
  - Output Q=0 when S=0, R=1 (reset condition)
  - Problem - Q is undefined if S=1 and R=1



Clocks

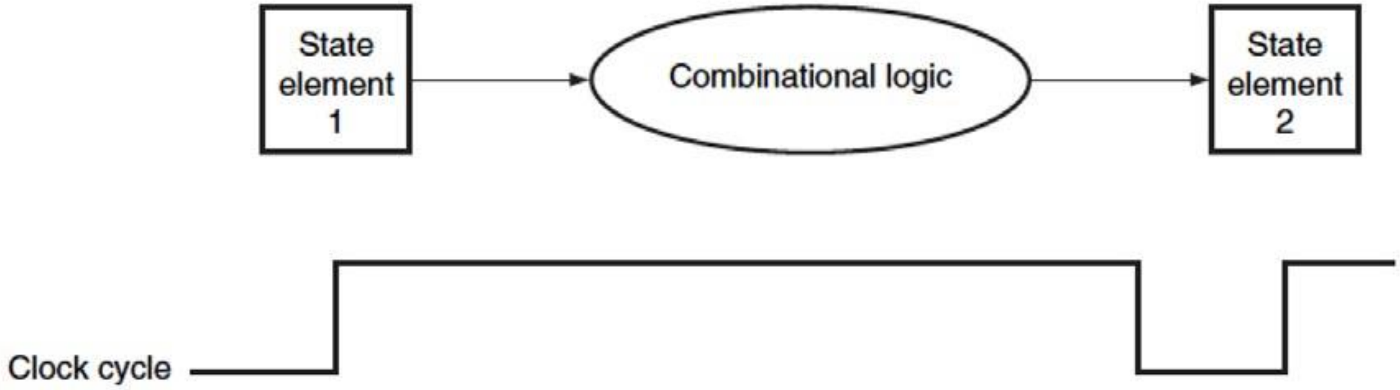
- ***Clock period***: time interval between pulses
  - example: period = 20 ns
- ***Clock frequency***: 1/period
  - example: frequency = 1 / 20 ns = 50 MHz
- ***Edge-triggered clocking***: all state changes occur on a clock edge.

Freq	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	1 ns
100 MHz	10 ns
10 MHz	100 ns



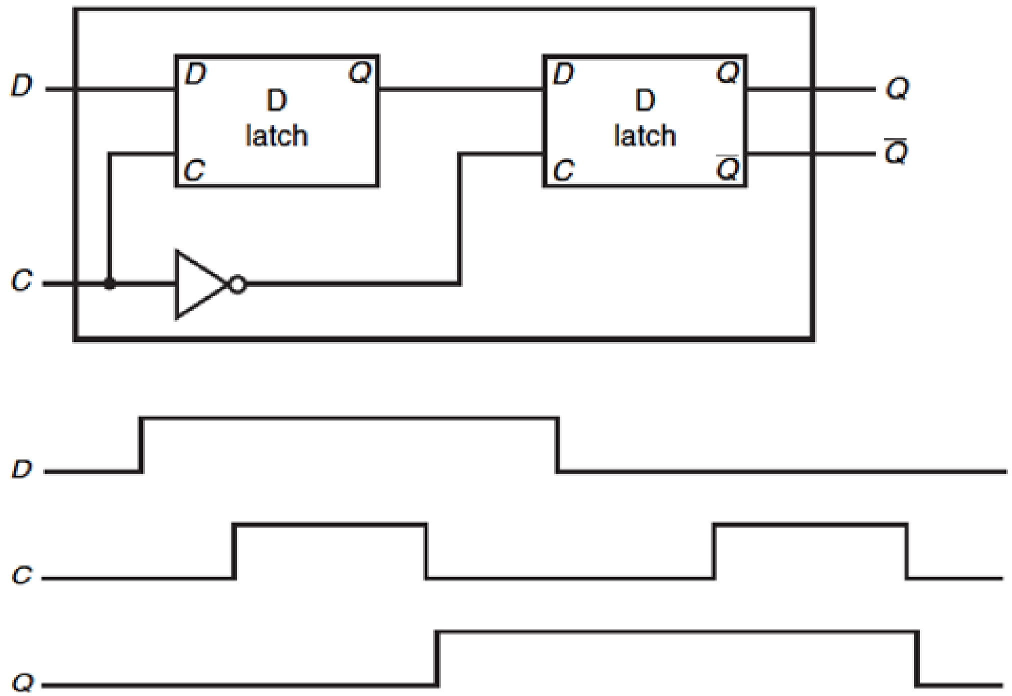
Clock and Change of State

- Clock controls when the state of a memory element changes
- To ensure that the values written into the state elements on the active clock edge are valid, the clock must have a long enough period



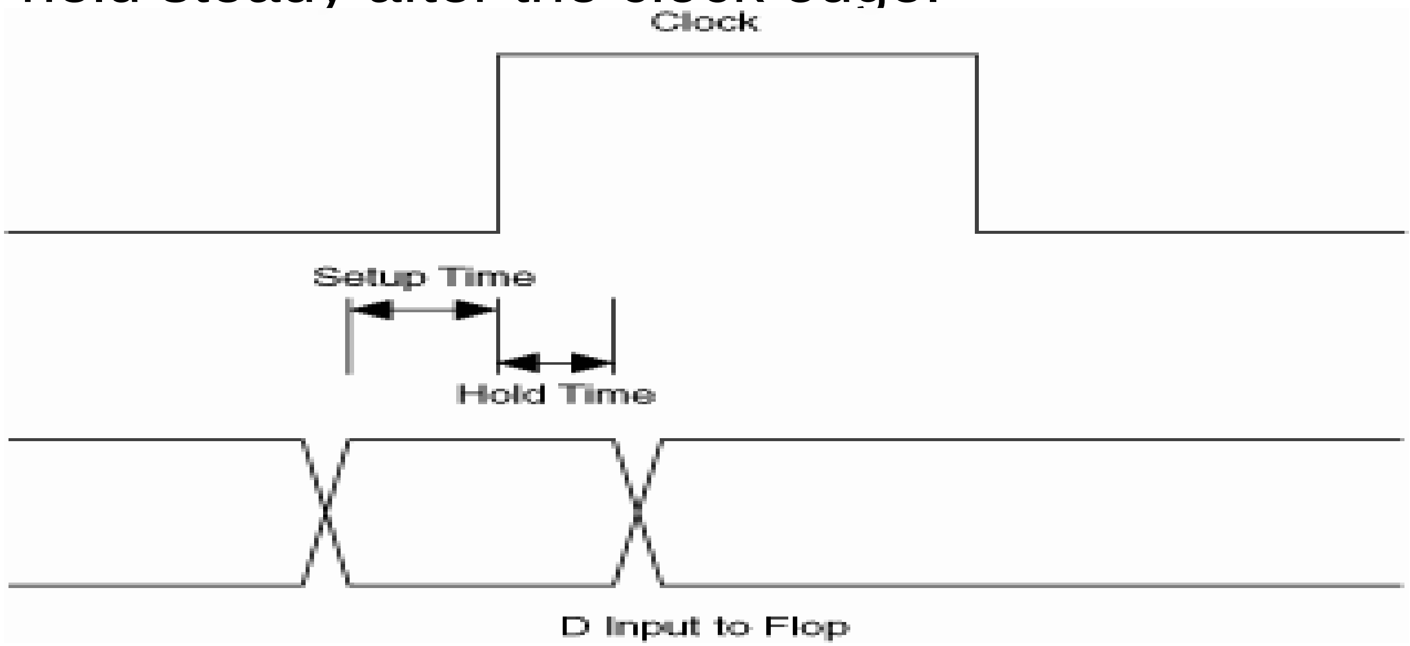
Clock Edge Triggered Bit Storage

- ***Flip-flop*** - Bit storage that stores on clock edge, not level
- D Flip-flop
  - Two latches, master and slave latches.
  - Output of the first goes to input of second, slave latch has inverted clock signal (falling-edge trigger)



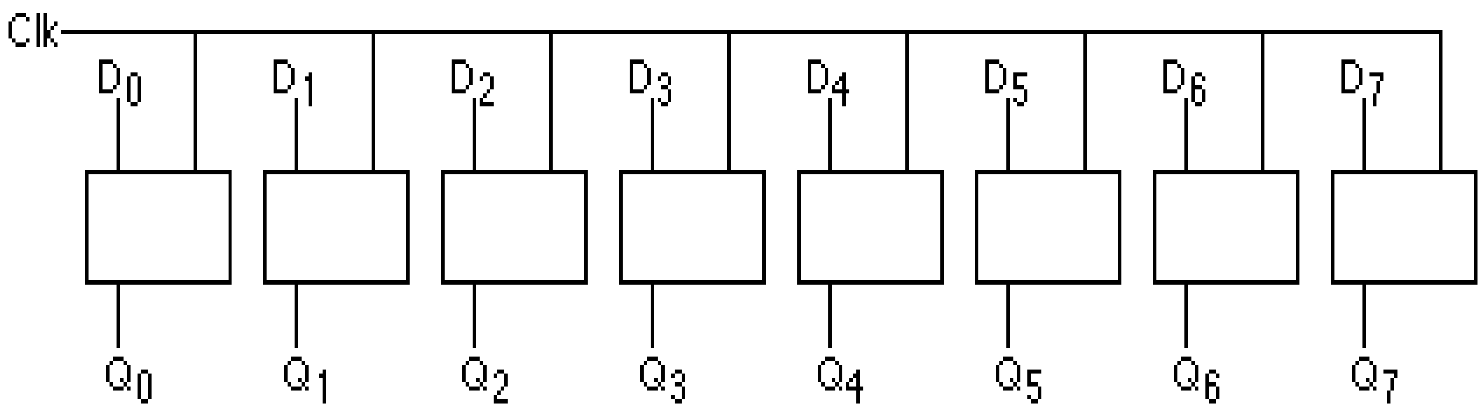
Setup and Hold Time

- Setup time
  - The minimum amount of time the data signal should be held steady before the clock edge arrives.
- Hold time
  - The minimum amount of time the data signal should be held steady after the clock edge.



N-Bit Register

- Cascade *N* number of D flip-flops to form an *N*-bit register
- An example of 8-bit register formed by 8 edge-triggered D flip-flops



Half Adders

- Need to add *bits* {0,1} of A_i and B_i
- Associate
  - binary bit 0 ↔ logic value F (0)
  - binary bit 1 ↔ logic value T (1)

$$
\begin{array}{l}
\qquad\qquad\quad C_{i+1} \\
A : A_n \ldots A_{i+1} A_i \ldots A_0 \\
B : B_n \ldots B_{i+1} B_i \ldots B_0 \\
\qquad\qquad\quad\; S_i
\end{array}
$$

- This leads to the following truth table

A_i	B_i	Sum_i	$Carry_{i+1}$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

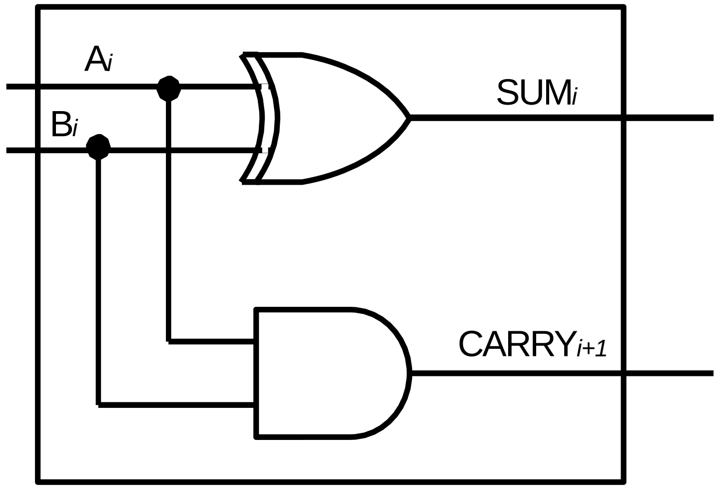
$$SUM_i = \overline{A_i} B_i + A_i \overline{B_i} = A_i \oplus B_i$$

$$CARRY_{i+1} = A_i B_i$$

Half Adder Circuit

$$SUM_i = \overline{A_i}B_i + A_i\overline{B_i} = A_i \oplus B_i$$

$$CARRY_{i+1} = A_i B_i$$



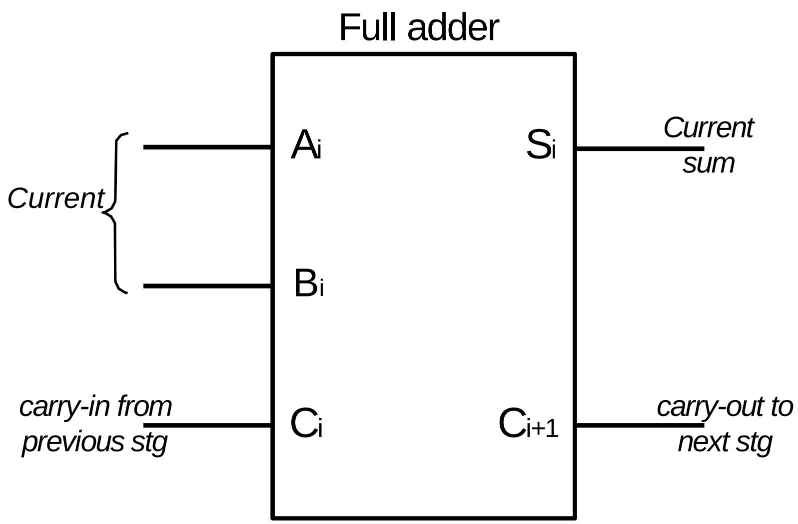
Half Adder Limitations

- Half adder circuits do not suffice for general addition because they do not include the carry bit from the previous stage of addition, e.g.

Carry		0 1 1 0
A		0 1 1 0
B	+	0 0 1 1
SUM		1 0 0 1

Full Adders (1-Bit ALU)

- Full adders can use the carry bit from the previous stage of addition



A_i	B_i	C_i	S_i	C_{i+1}
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

Full Adder Logic Expressions

Sum

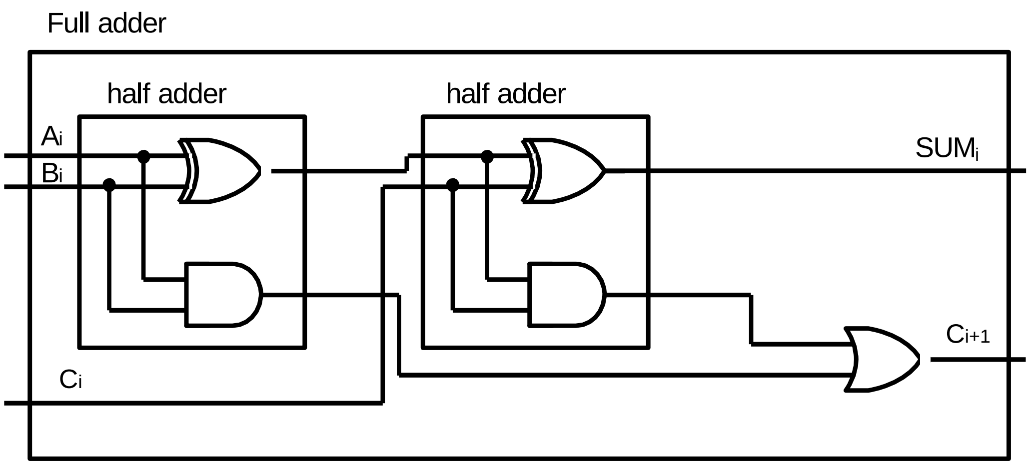
$$
\begin{aligned}
SUM_i &= \overline{A_i}\,\overline{B_i}C_i + \overline{A_i}B_i\overline{C_i} + A_i\overline{B_i}\,\overline{C_i} + A_iB_iC_i \\
&= \overline{A_i}(\overline{B_i}C_i + B_i\overline{C_i}) + A_i(\overline{B_i}\,\overline{C_i} + B_iC_i) \\
&= \overline{A_i}(B_i \oplus C_i) + A_i(\overline{B_i \oplus C_i}) \\
&= A_i \oplus B_i \oplus C_i
\end{aligned}
$$

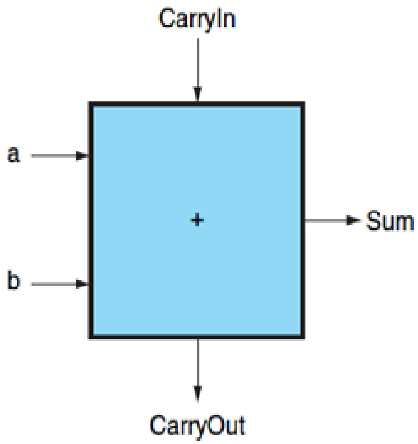
Carry

$$
\begin{aligned}
C_{i+1} &= A_iB_i + \overline{A_i}B_iC_i + A_i\overline{B_i}C_i \\
&= A_iB_i + C_i(\overline{A_i}B_i + A_i\overline{B_i}) \\
&= A_iB_i + C_i(A_i \oplus B_i)
\end{aligned}
$$

Full Adder Circuit

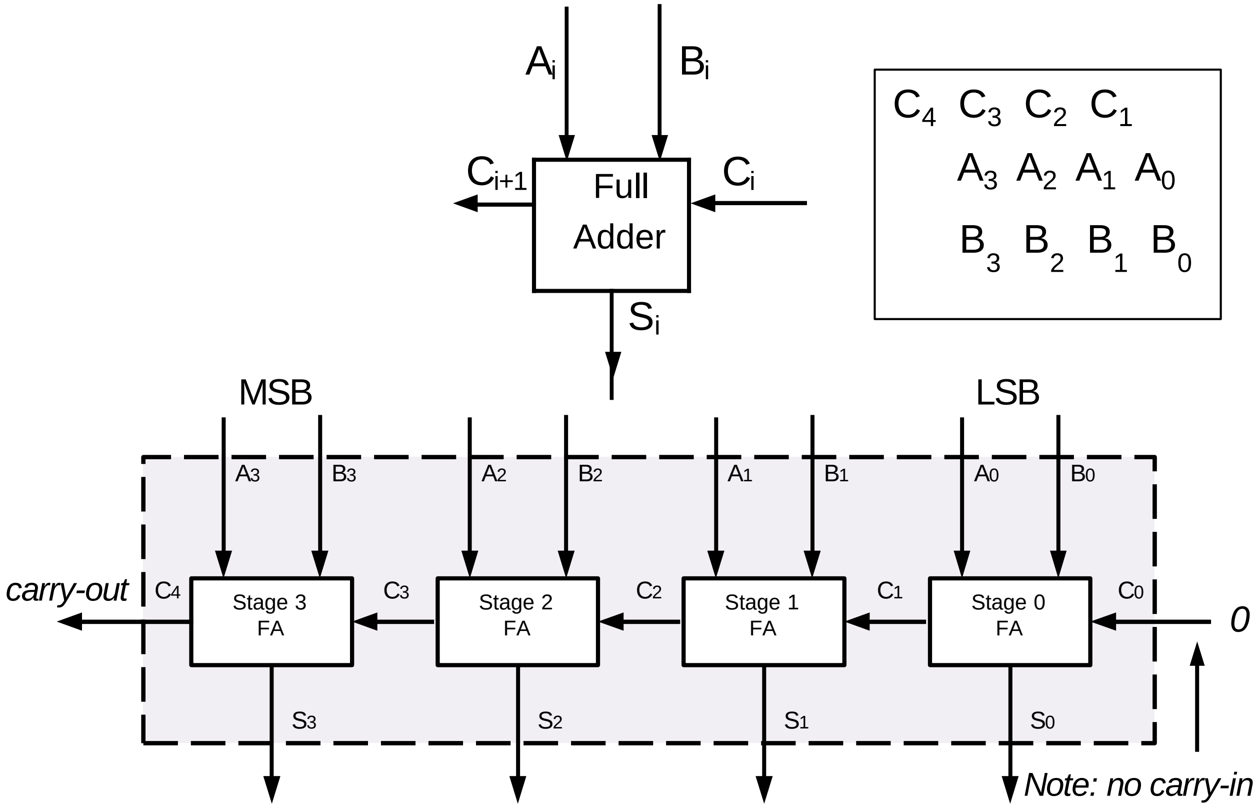
$$SUM = (A_i \oplus B_i) \oplus C_i \qquad C_{i+1} = A_i B_i + C_i(A_i \oplus B_i)$$





Note: A full adder adds 3 bits. Can also consider as first adding first two and then the result with the carry

N-Bit Adders (Ripple Carry)

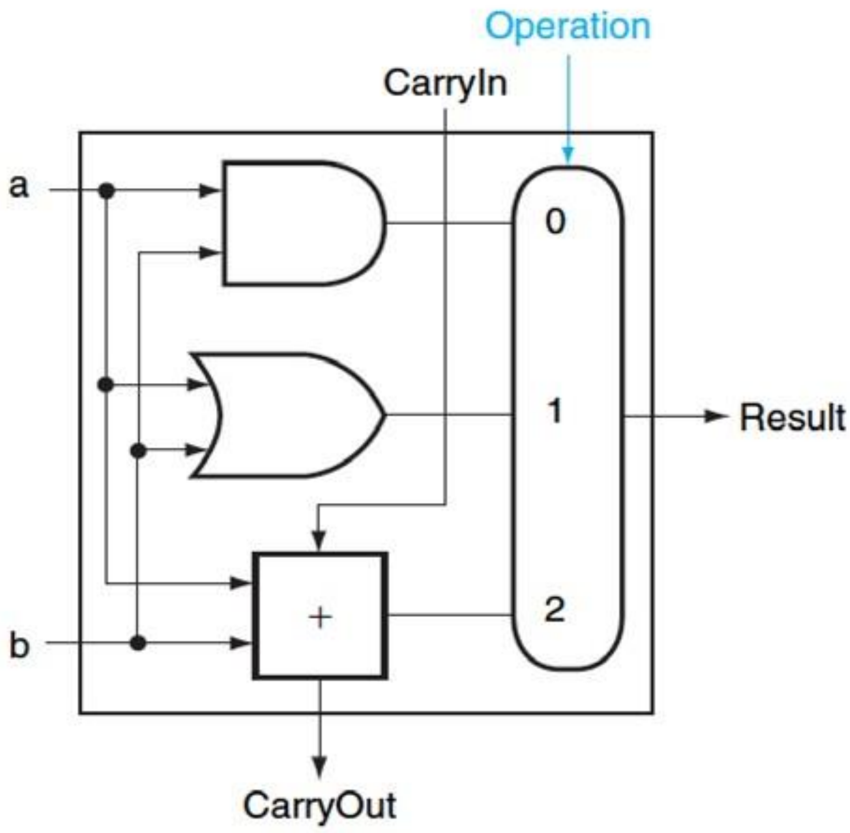


Ripple Carry Adders

- 4 FA's cascaded to form a 4-bit adder
- In general, *N*-FA's can be used to form an *N*-bit adder
- Carry bits have to propagate from one stage to the next. Inherent propagation delays associated with this
- Output of each FA is therefore not stable until the carry-in from the previous stage is calculated

Enhancement to 1-bit Adder

- 1-bit ALU with AND, OR, and addition
  - Supplemented with AND and OR gates
  - A multiplexor controls which gate is connected to the output

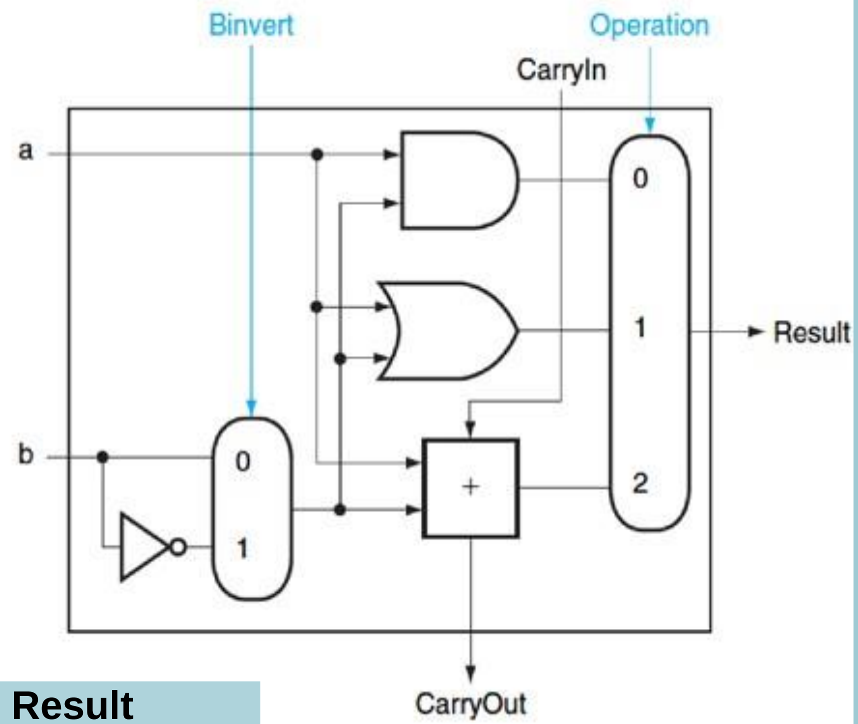


Operation	Result
00	AND
01	OR
10	Addition

Enhancement to 1-bit Adder

- 1-bit ALU for subtraction
  - Subtraction is performed using 2's complement, i.e.

$$a - b = a + \bar{b} + 1$$

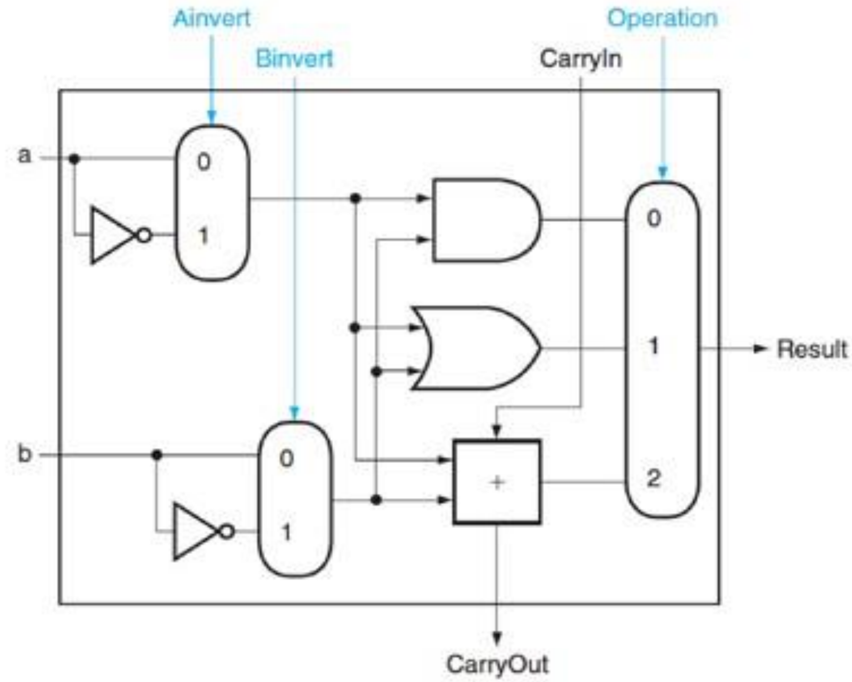


Binvert	CarryIn	Operation	Result
0	0	00	AND
0	0	01	OR
0	0	10	Addition
1	1	10	Subtraction

Enhancement to 1-bit Adder

- 1-bit ALU for NOR operation
- A MIPS ALU also needs a NOR function

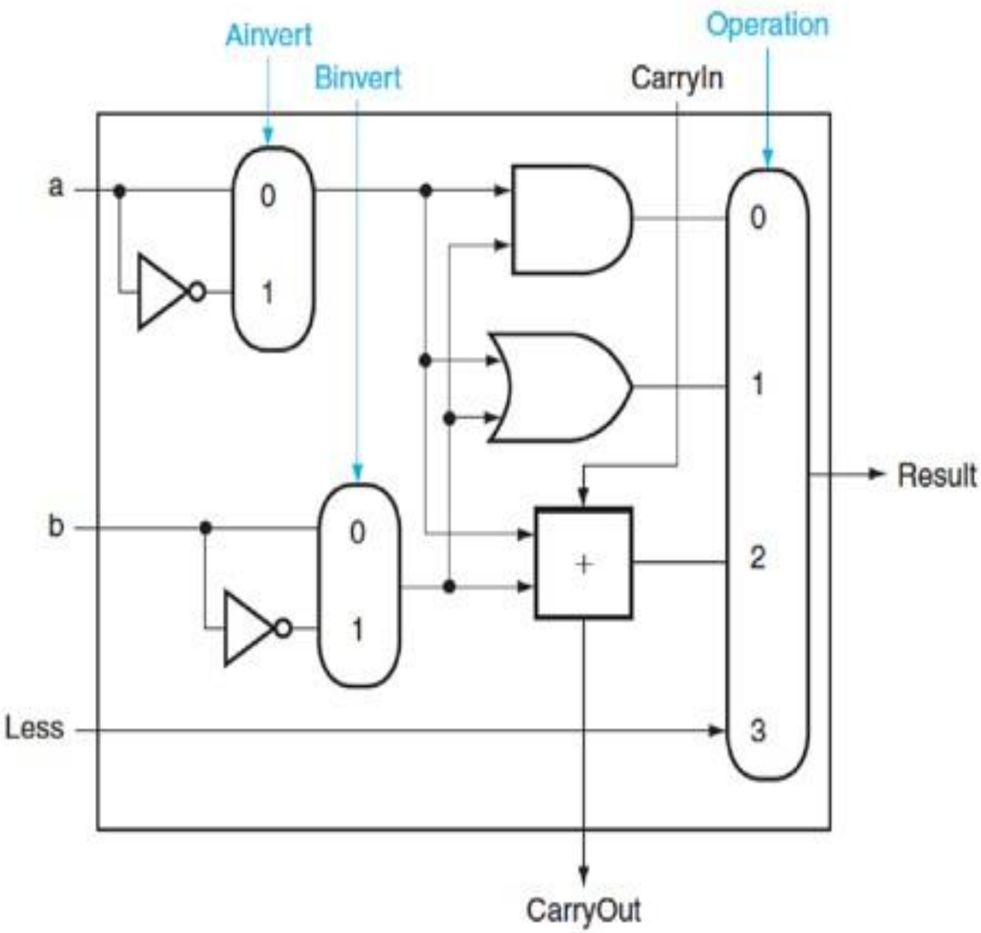
$$\overline{(a + b)} = \overline{a} \cdot \overline{b}$$



Ainvert	Binvert	CarryIn	Operation	Result
0	0	0	00	AND
1	1	0	00	NOR
0	0	0	01	OR
0	0	0	10	Addition
0	1	1	10	Subtraction

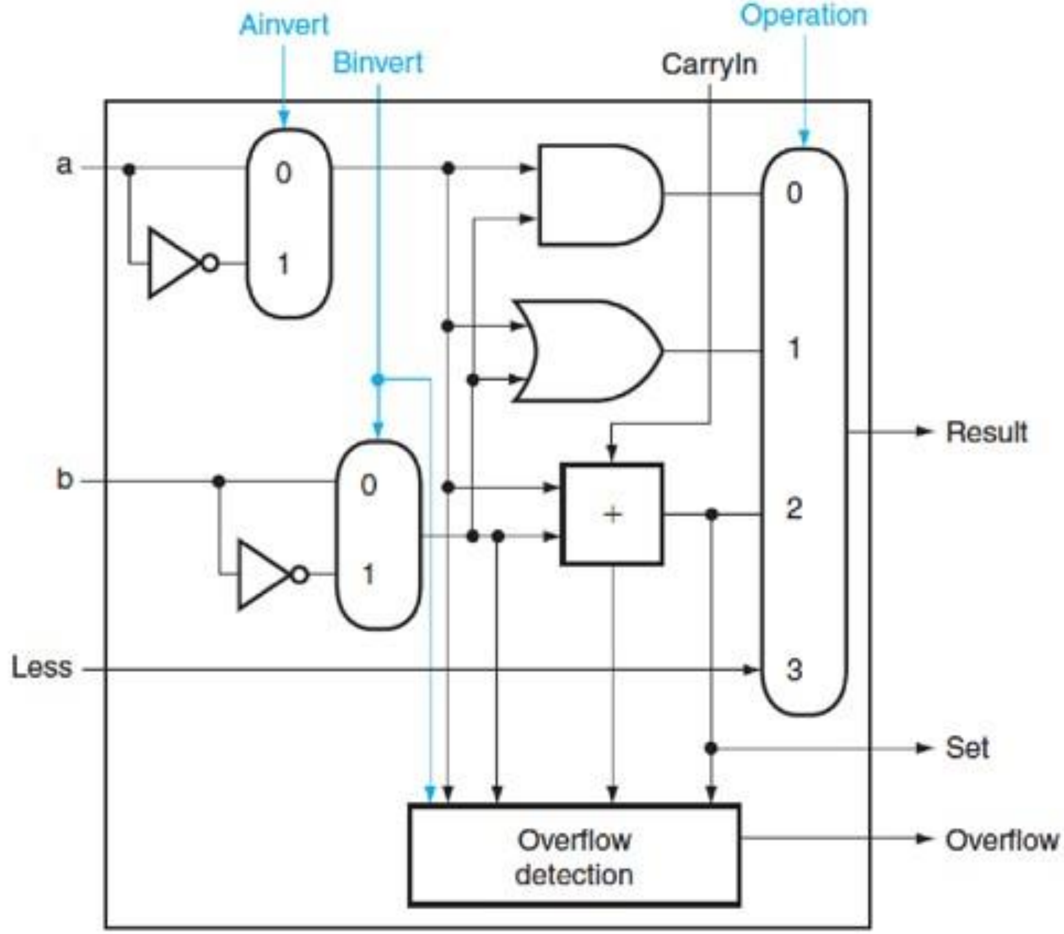
Enhancement to 1-bit Adder

- 1-bit ALU for SLT operations
- slt $s1, $s2, $s3
  - If ($s2<$s3), $s1=1, else $s1=0
- adding one input "*less*"
  - if (a<b), set *less* to 1 or if (a-b)<0, set *less* to 1
  - If the result of subtraction is negative, set *less* to 1



Enhancement to 1-bit Adder

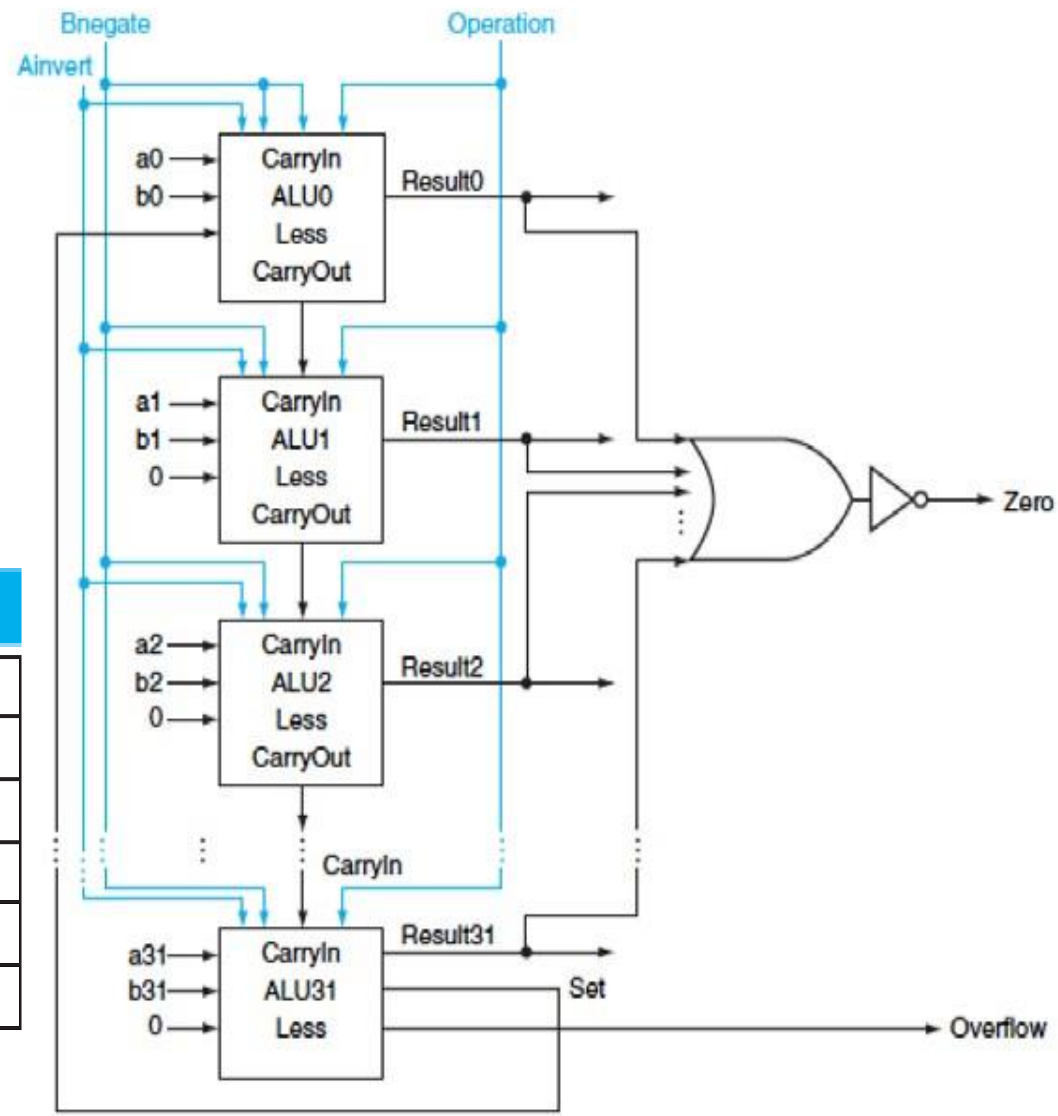
- How to determine if the result is negative?
  - Negative →→ Sign bit value=1
- Create a new output "Set" direct output from the adder and use only for slt
- An overflow detection is included for the most significant bit ALU



32-Bit ALU

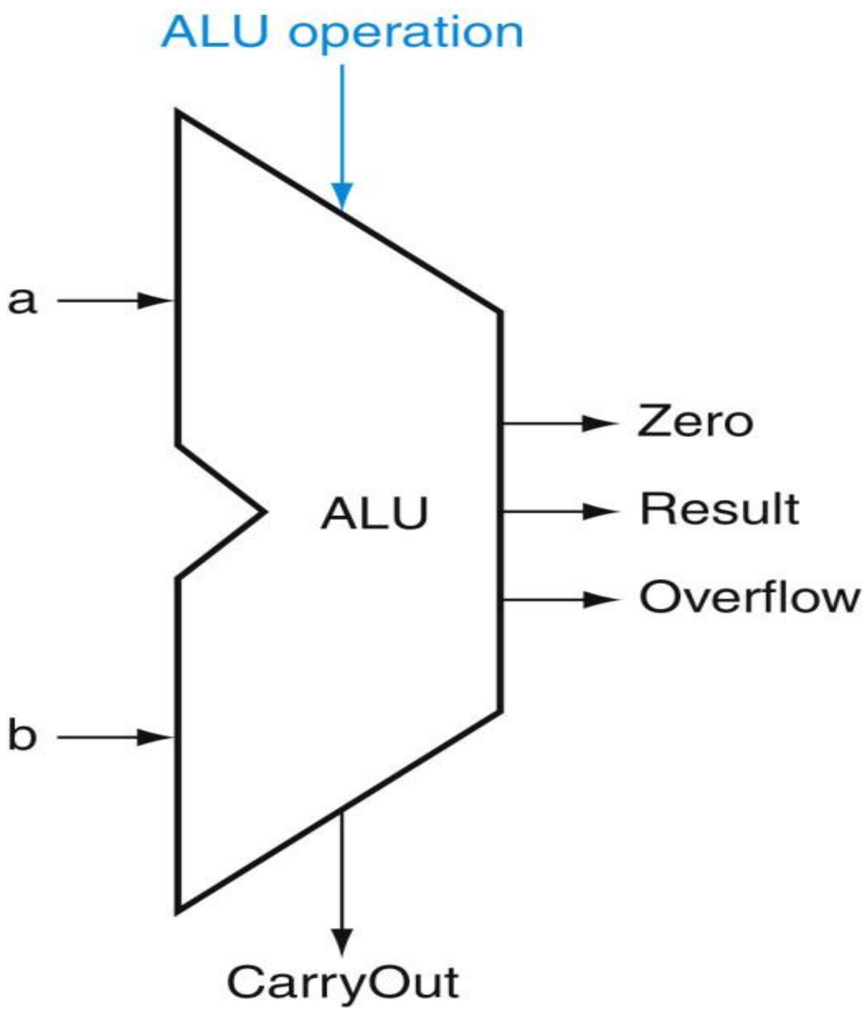
- OR and NOT gates are added to support conditional branch instruction, i.e. test the result of a-b if the result is 0.

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



32-Bit ALU

- The symbol commonly used to represent an ALU
- This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder

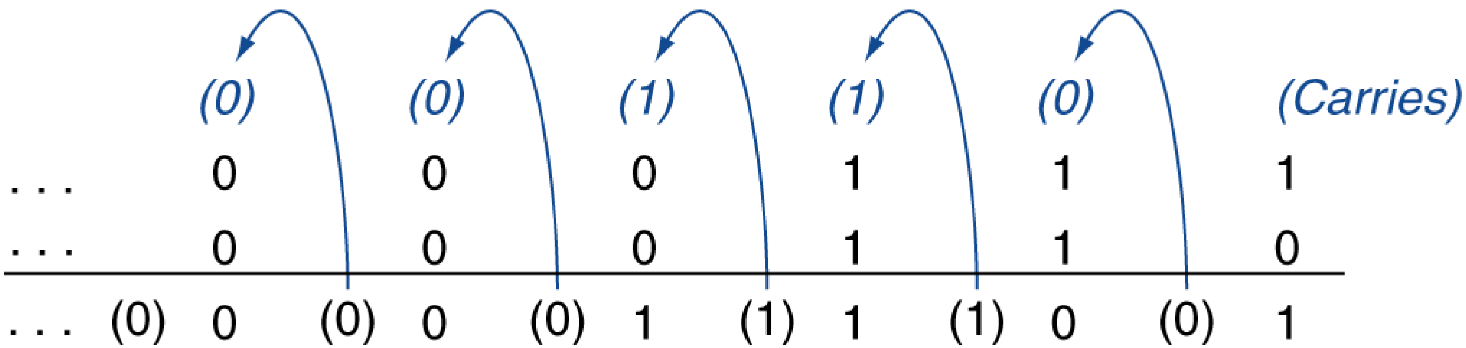


Arithmetic for Computers

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

Integer Addition

- Example: 7 + 6



- Overflow if result 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

- 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 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

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/interrupt
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception/interrupt 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

Multiplication

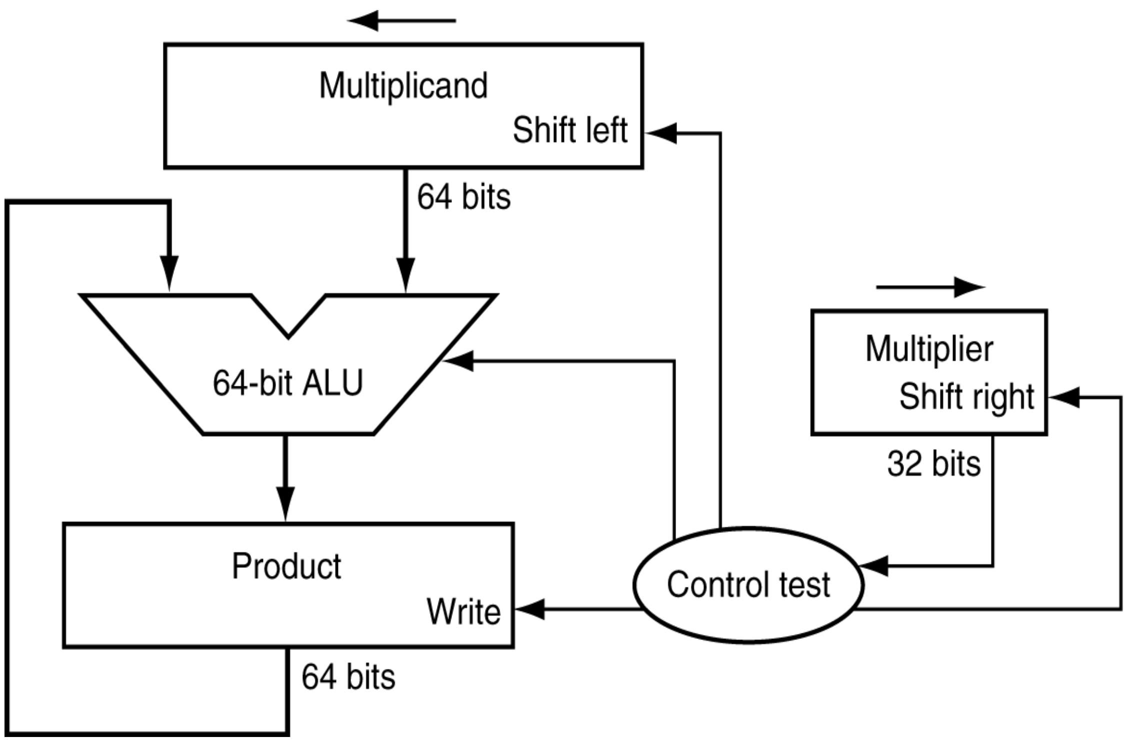
- Start with long-multiplication approach

multiplicand → 1000
multiplier → × 1001

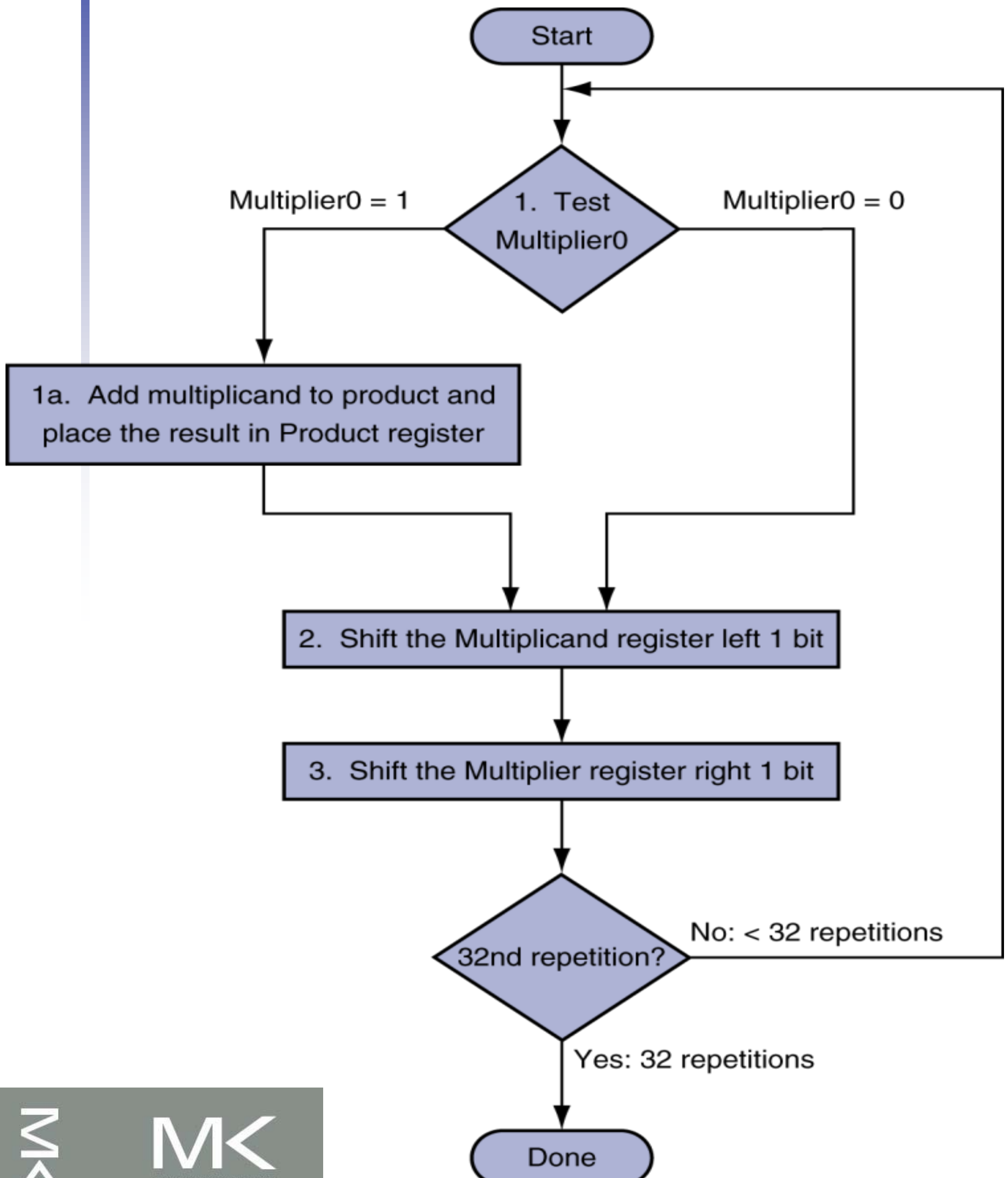
```
   1000
× 1001
-------
   1000
  0000
 0000
1000
-------
1001000
```

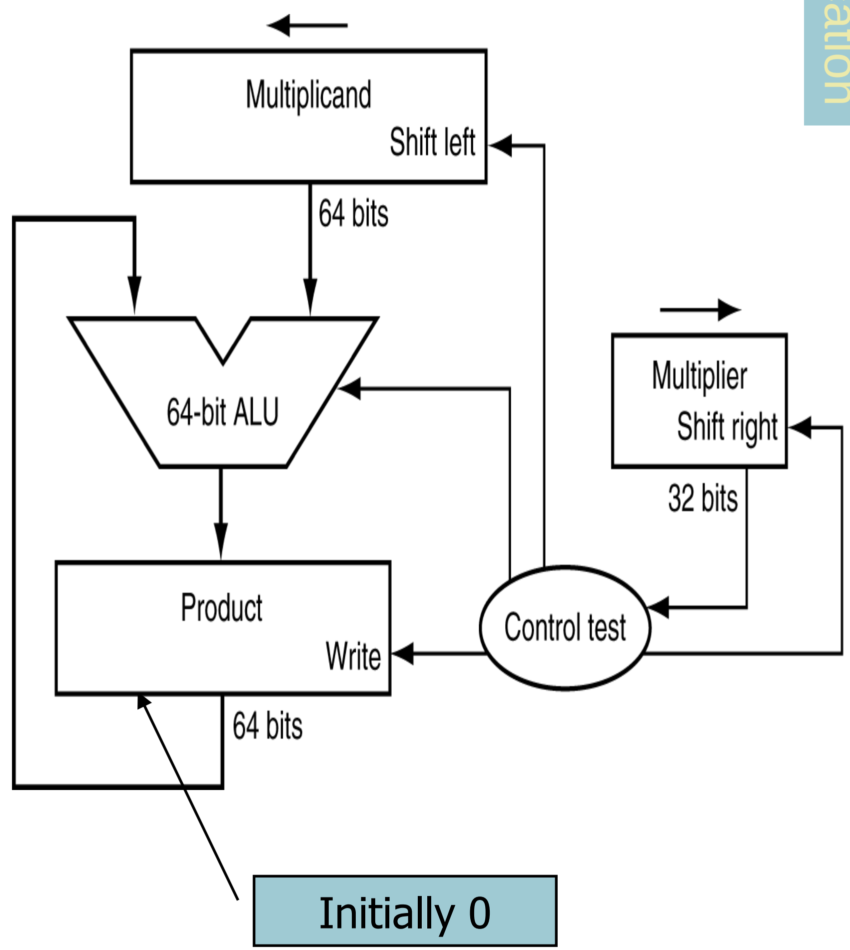
product → 1001000

Length of product is the sum of operand lengths



Multiplication Hardware





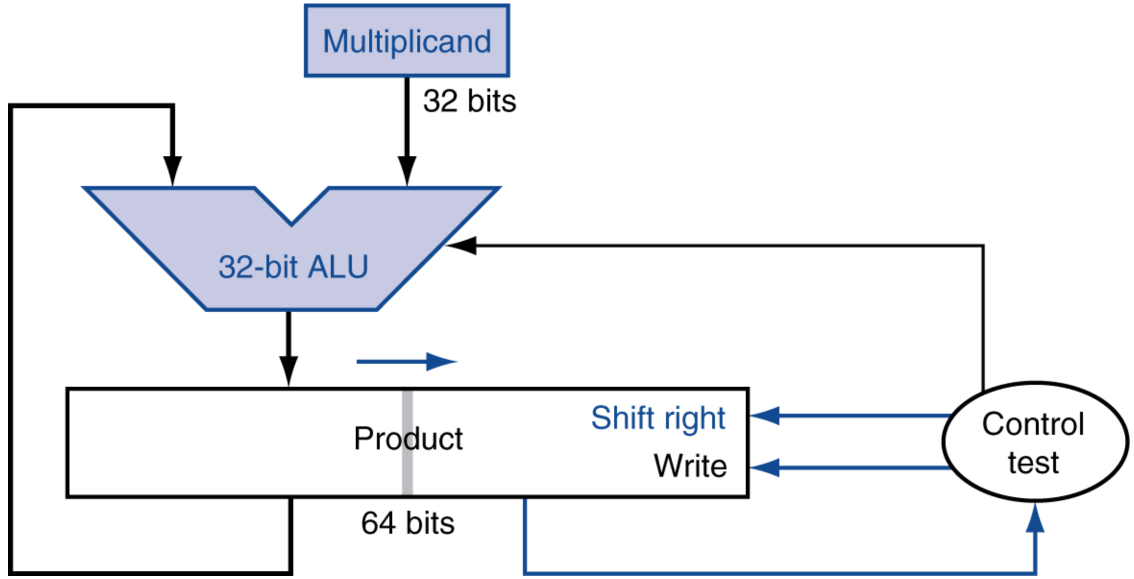
Multiplication Hardware

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

- Multiply example using flow chart algorithm
- The bit examined to determine the next step is circled in color

Optimized Multiplier

- Perform steps in parallel: add/shift
- Read/Write/Shift



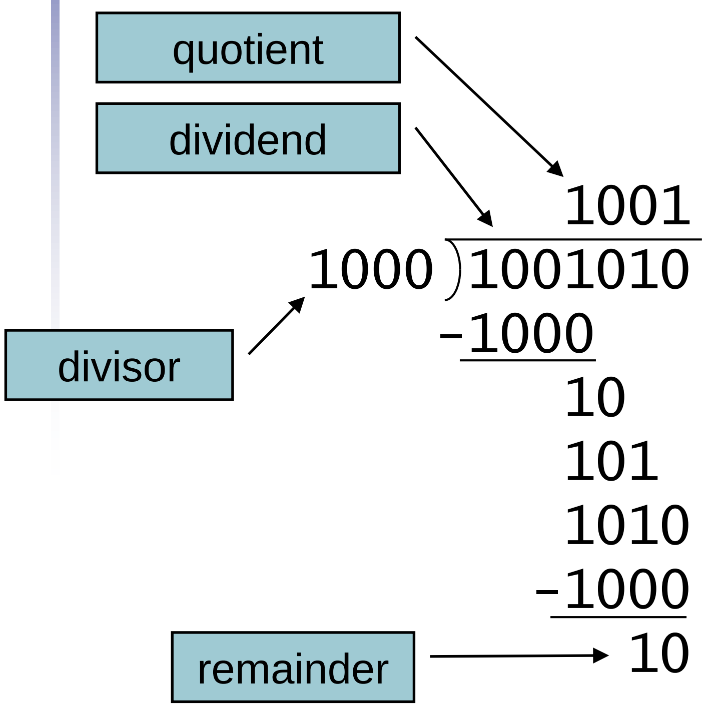
- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

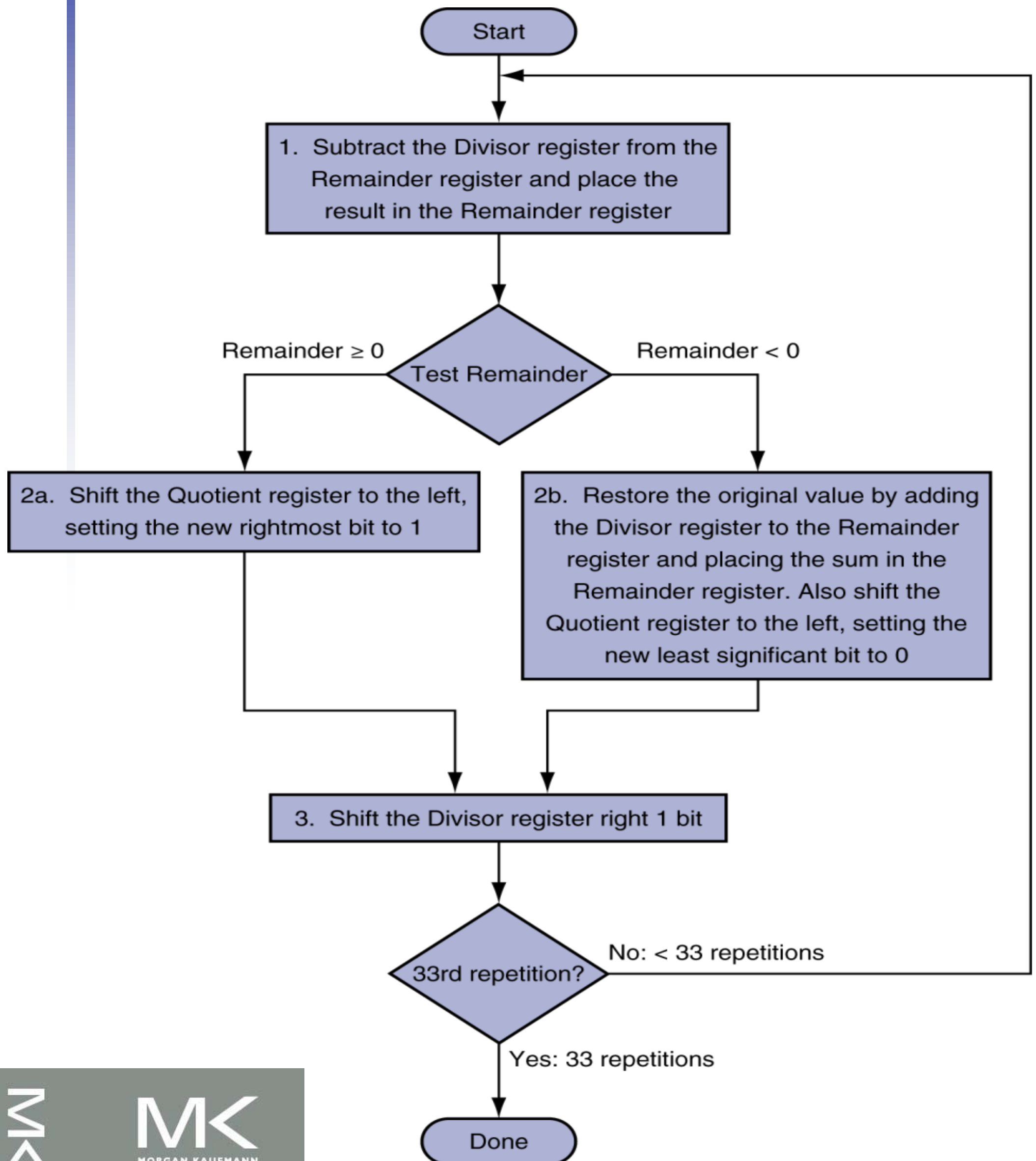
Division

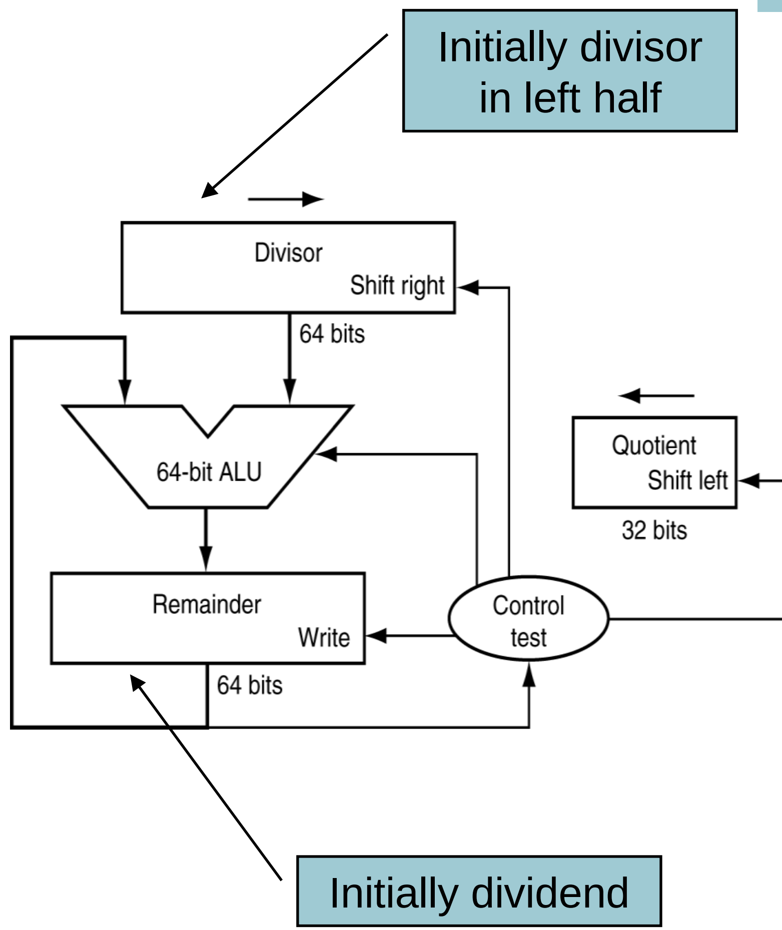
- Check for 0 divisor
- Long division
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute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



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

Division Hardware





Division Example

Using a 4-bit version of the algorithm divide 7_{10} by 2_{10}, or $0000\ 0111_2$ by 0010_2.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt` / `divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← 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 significand: 1.0 ≤ |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
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ 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 = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129-127)}$

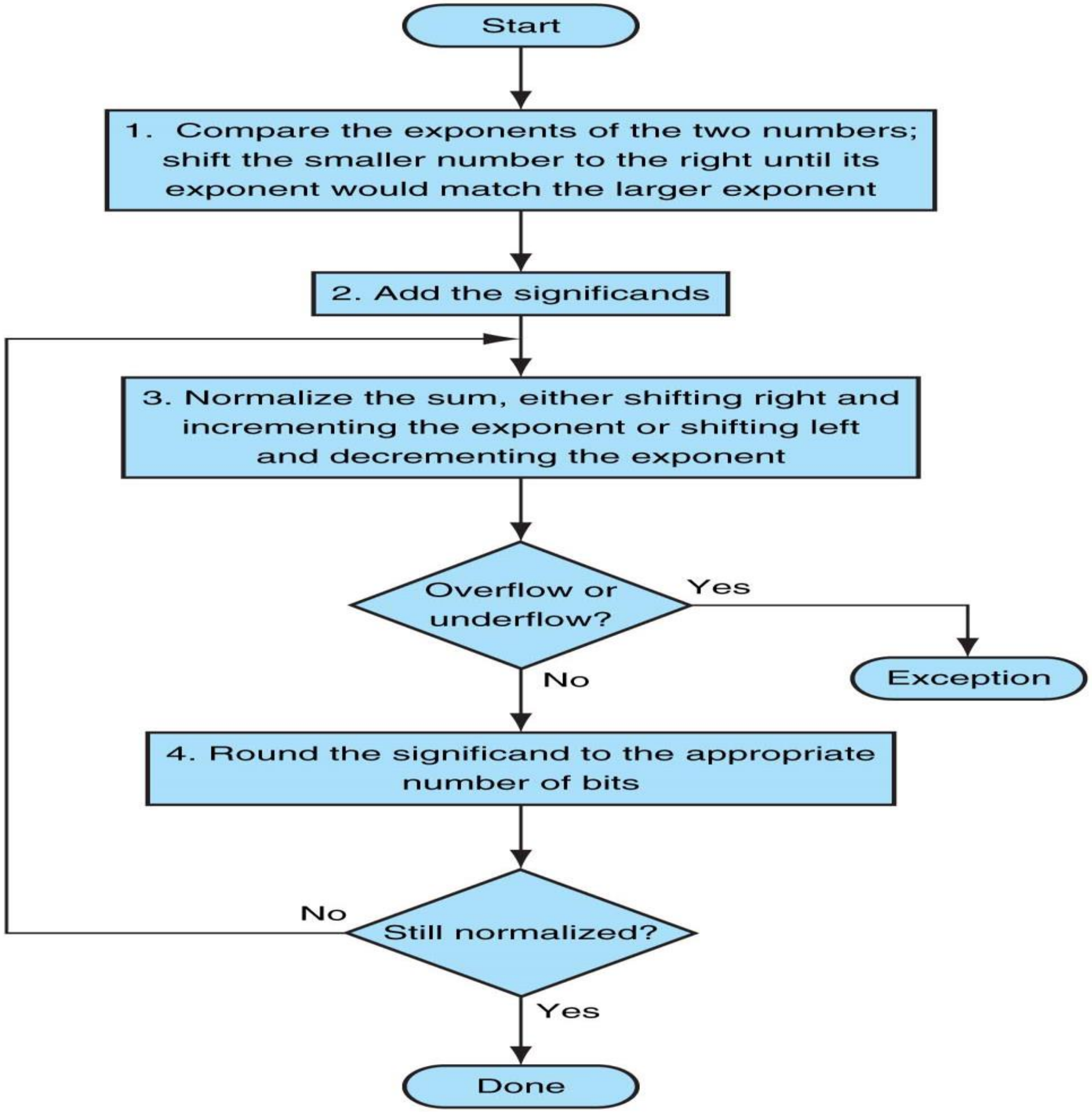
  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

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



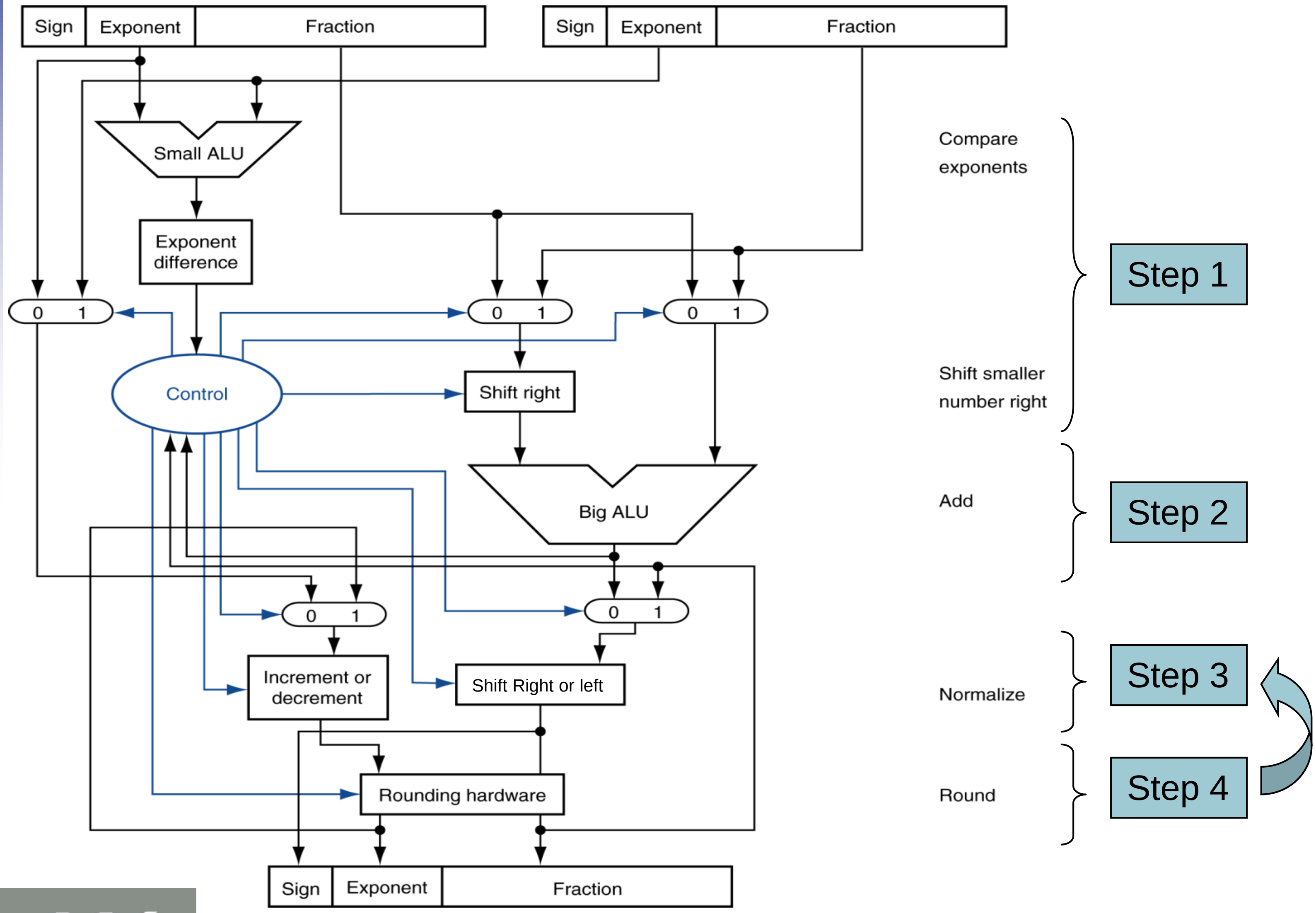
Floating-Point Addition

- 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

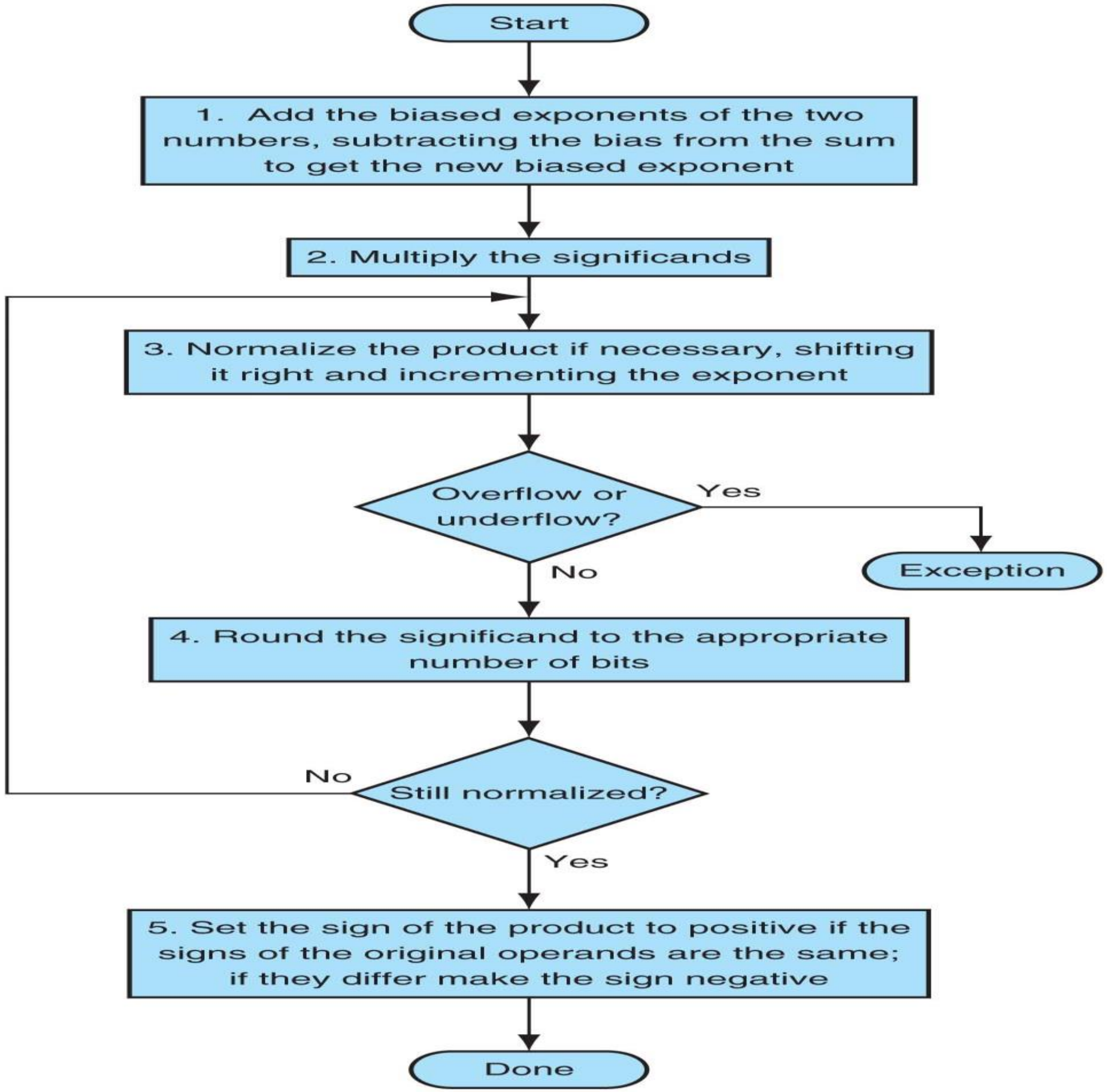
FP Adder Hardware



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



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
  - $-1 + -2 = -3$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 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 $\leftrightarrow$ integer conversion
- Operations usually take several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is eq, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1   $f16, const5($gp)
     lwc1   $f18, const9($gp)
     div.s  $f16, $f16, $f18
     lwc1   $f18, const32($gp)
     sub.s  $f18, $f12, $f18
     mul.s  $f0,  $f16, $f18
     jr     $ra
```

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

Acknowledgement

The slides are adopted from Computer Organization and Design, 5th Edition

by David A. Patterson and John L. Hennessy
2014, published by MK (Elsevier)