CESE4130: Computer Engineering

2024-2025, lecture 2

Binary Computer Arithmetic

Computer Engineering Lab

Faculty of Electrical Engineering, Mathematics & Computer Science

2024-2025

Anteneh Gebregiorgis

TU Delft

Delft University of Technology

Course objectives

- Describe number representation system and inter-conversion.
- Perform binary arithmetic operation such as addition and multiplication.
- Explain basic concepts of computer architecture.
- Use logic gates to implement simple combinational circuits.
- Explain system software and operating systems fundamentals, task management, synchronization, compilation, and interpretation.
- Use design and automation tools to perform synthesis and optimization.

Lecture objectives

- Perform binary addition and counting
- Basic Multiplication and Division Schemes

Lecture objectives

- Perform binary addition and counting
- Basic Multiplication and Division Schemes
- Understand floating point representation and operation

Lecture objectives

- Perform binary addition and counting
- Basic Multiplication and Division Schemes
- Understand floating point representation and operation
- Familiarize with accurate and approximate operations

Overview

- Basic addition and counting
- Carry-Lookahead adders

Overview

- Basic addition and counting
- Carry-Lookahead adders
- Basic multiplication operation
- Basic division operation

Overview

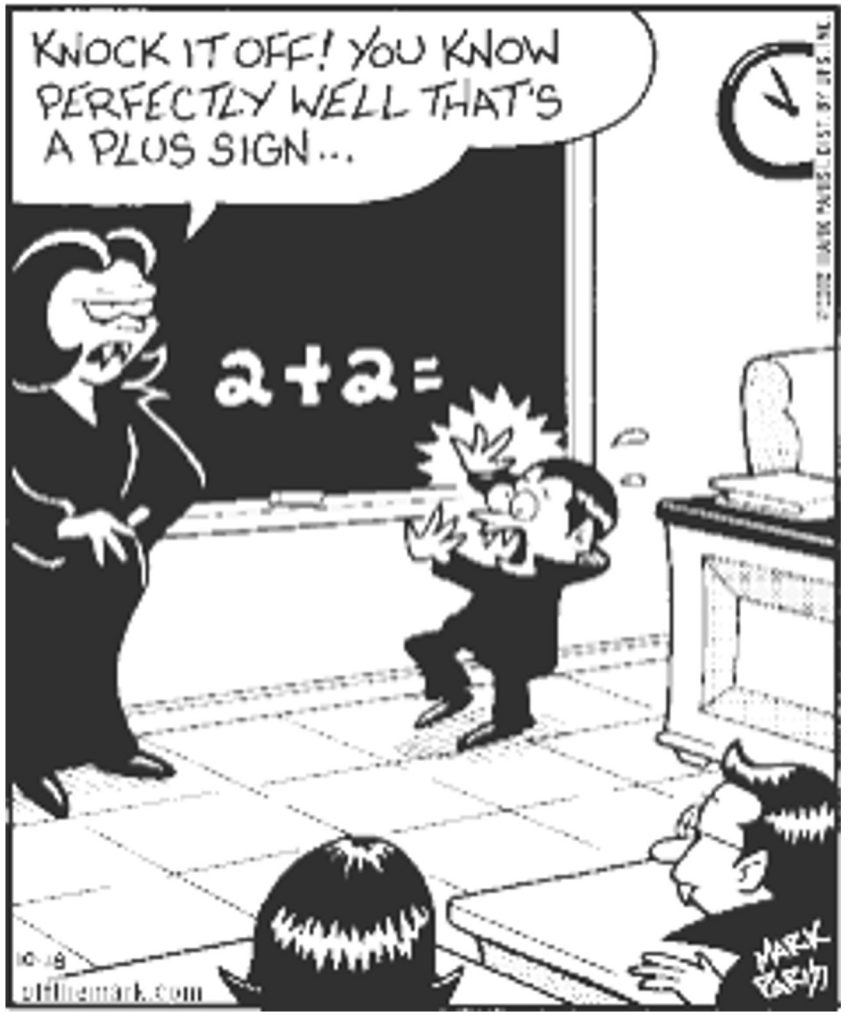
- Basic addition and counting
- Carry-Lookahead adders
- Basic multiplication operation
- Basic division operation
- Floating point representation
- Floating point operation

Overview

- Basic addition and counting
- Carry-Lookahead adders
- Basic multiplication operation
- Basic division operation
- Floating point representation
- Floating point operation
- Advanced cases

Basic addition and counting

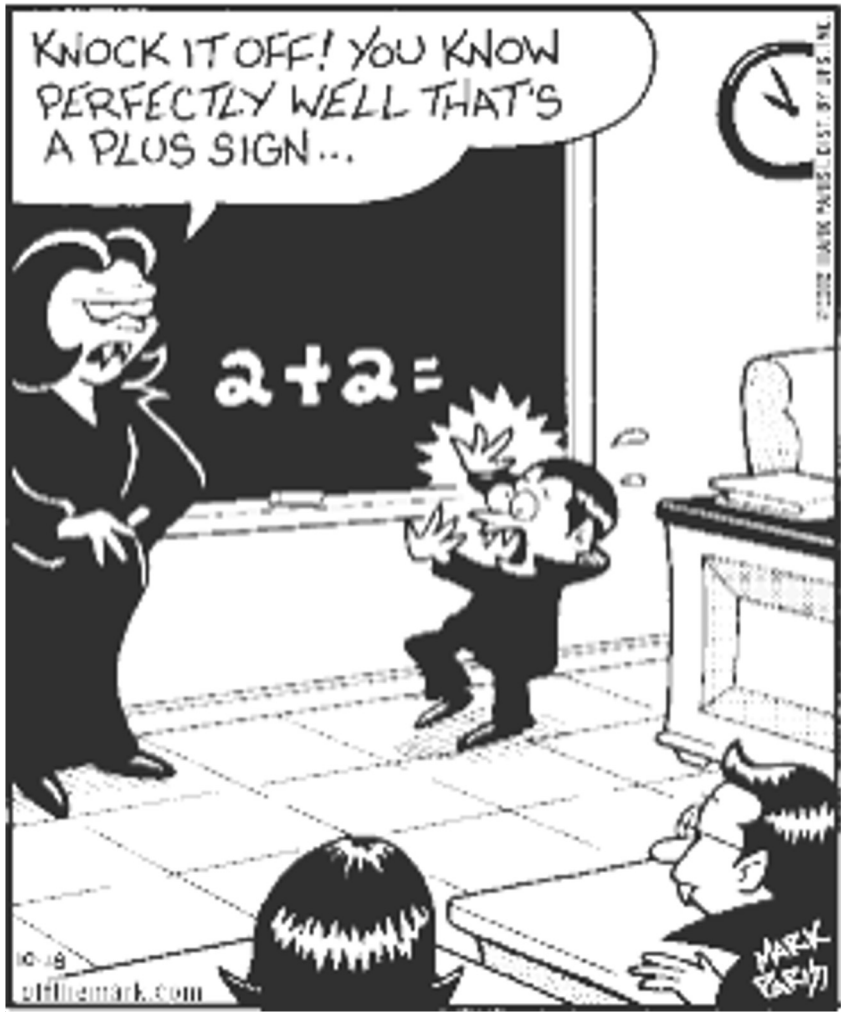
Computers know only 1’s and 0’s



Basic addition and counting

Computers know only 1's and 0's

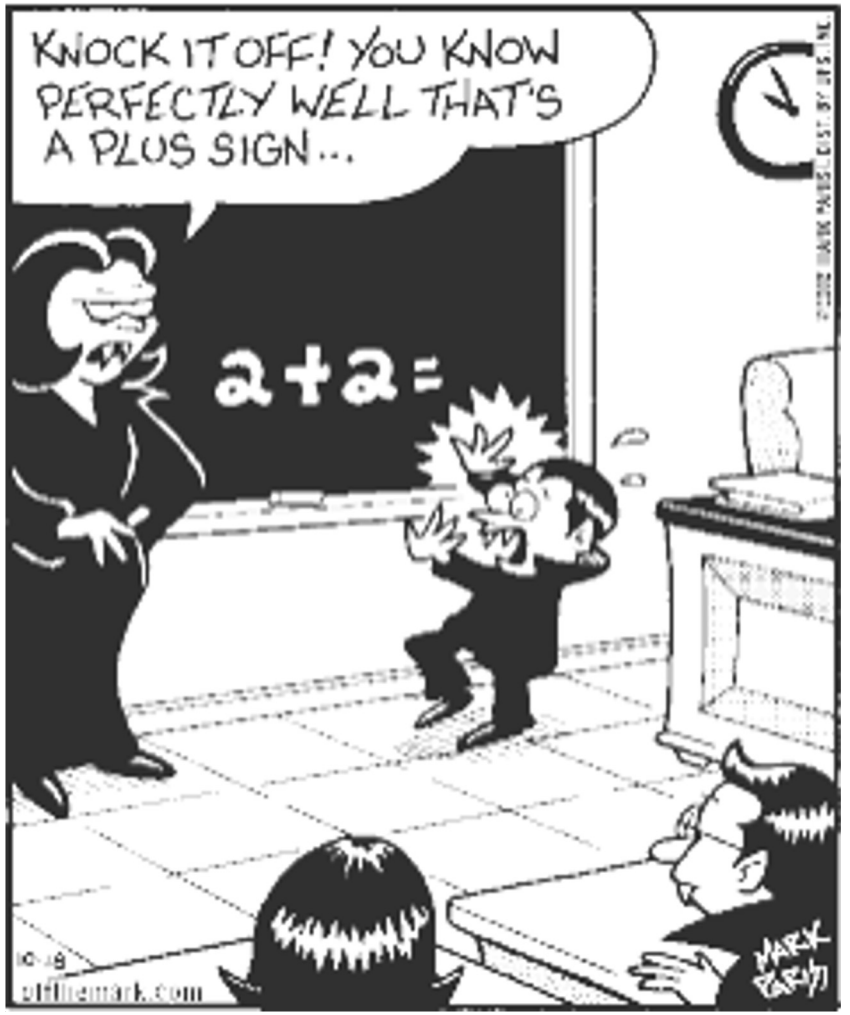
- To add binary numbers is relatively simple



Basic addition and counting

Computers know only 1's and 0's

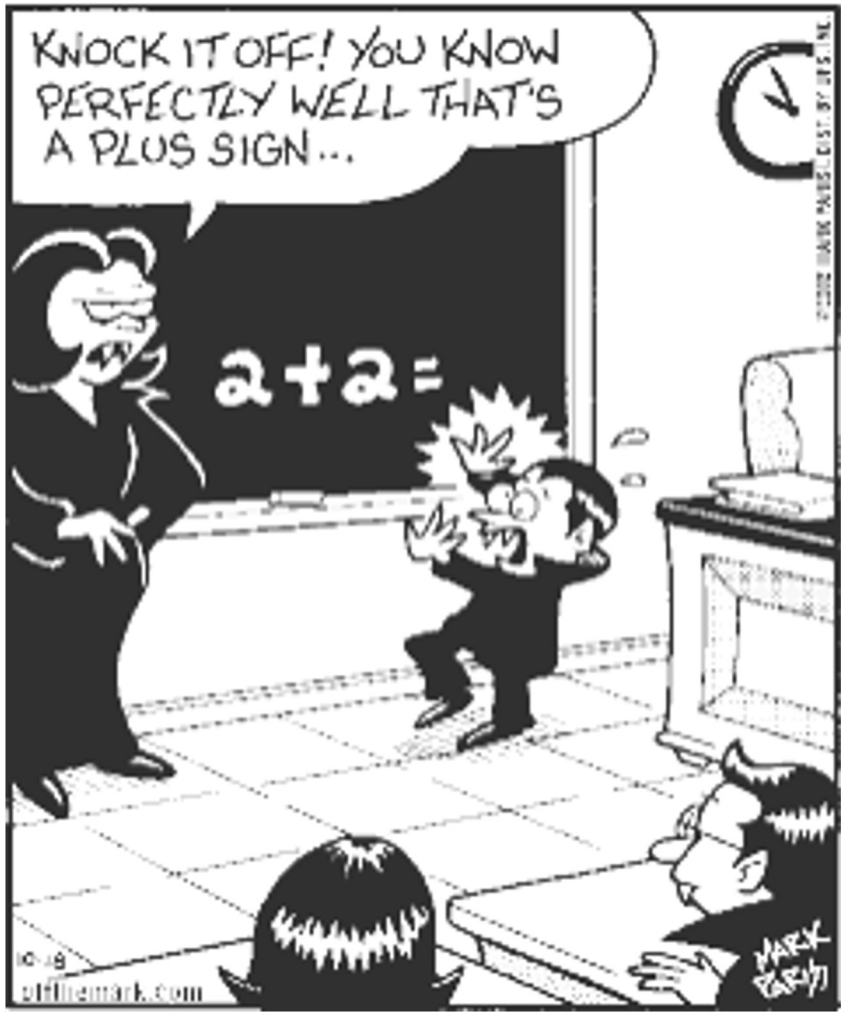
- To add binary numbers is relatively simple
- Same principles of adding normal decimals



Basic addition and counting

Computers know only 1's and 0's

- To add binary numbers is relatively simple
- Same principles of adding normal decimals
- Result needs to fit in with the rules of binary



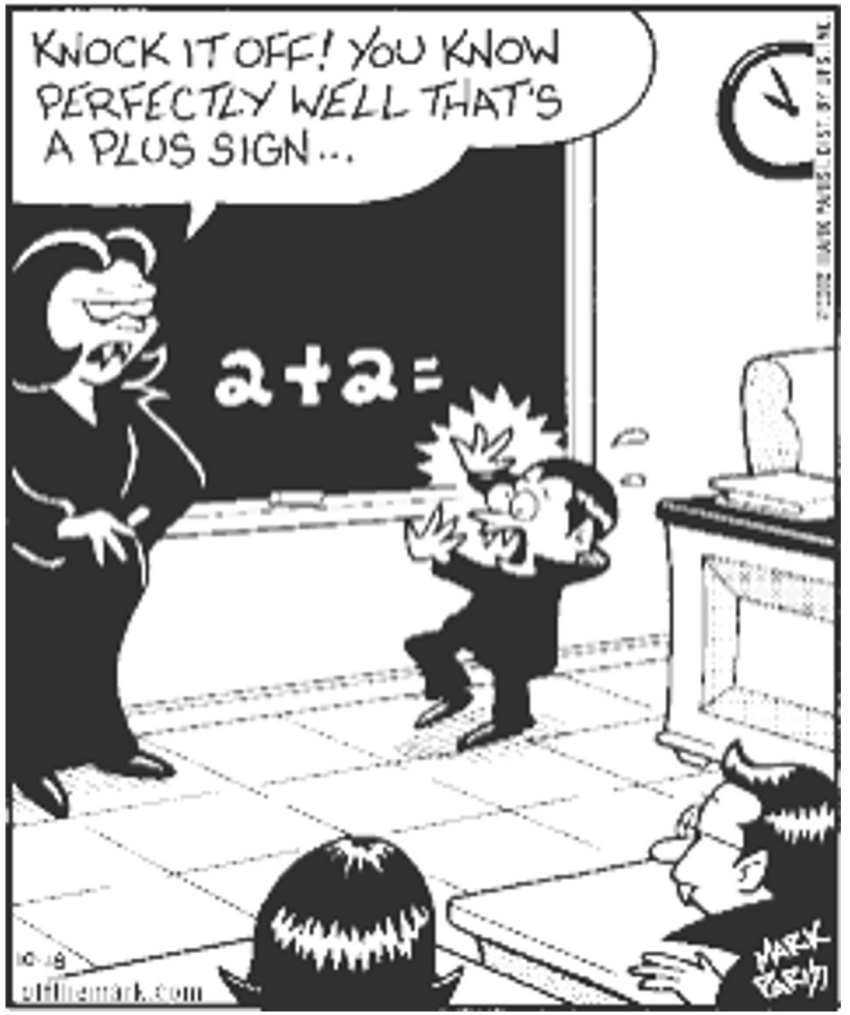
Basic addition and counting

Computers know only 1's and 0's

- To add binary numbers is relatively simple
- Same principles of adding normal decimals
- Result needs to fit in with the rules of binary

Fundamental rules of binary addition

- 0 + 0 = 0



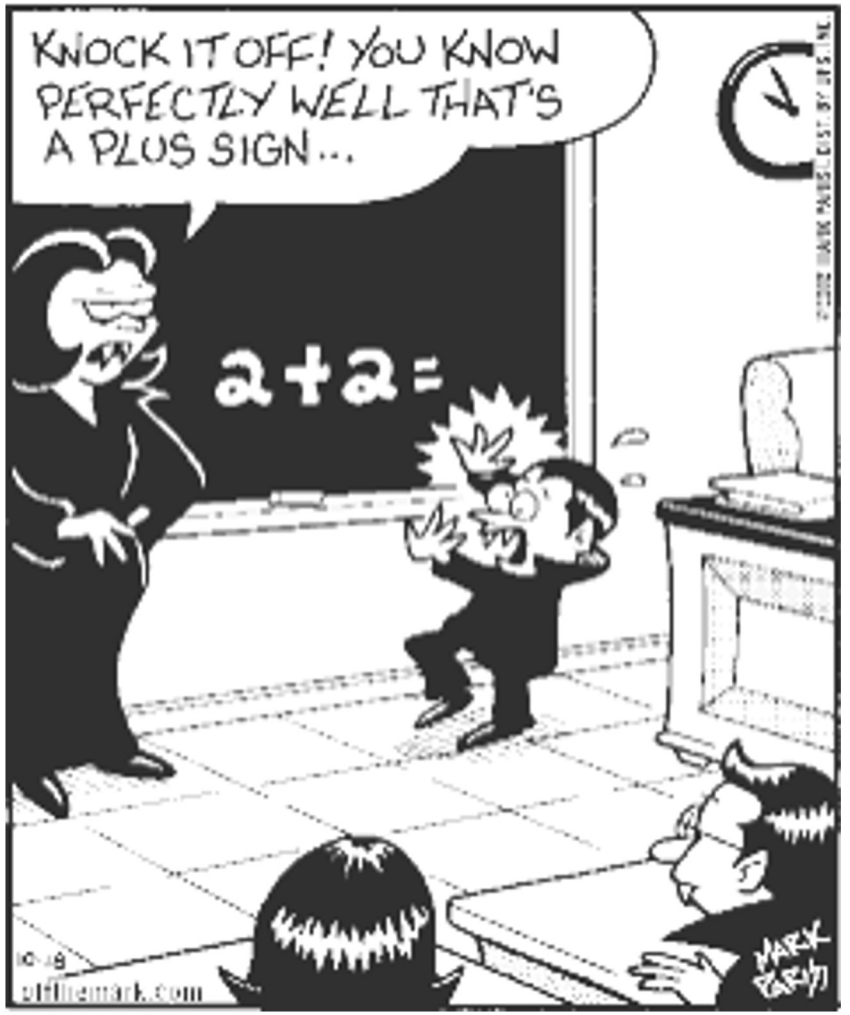
Basic addition and counting

Computers know only 1's and 0's

- To add binary numbers is relatively simple
- Same principles of adding normal decimals
- Result needs to fit in with the rules of binary

Fundamental rules of binary addition

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1



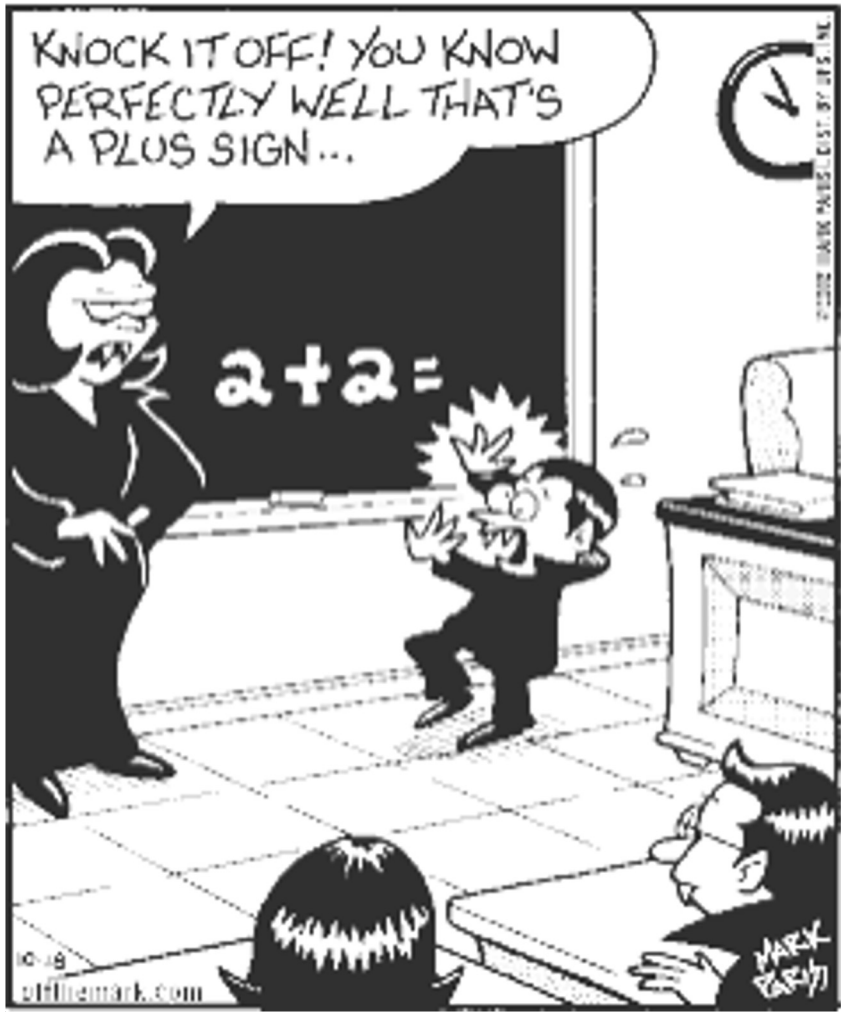
Basic addition and counting

Computers know only 1's and 0's

- To add binary numbers is relatively simple
- Same principles of adding normal decimals
- Result needs to fit in with the rules of binary

Fundamental rules of binary addition

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10



Basic addition and counting

Computers know only 1's and 0's

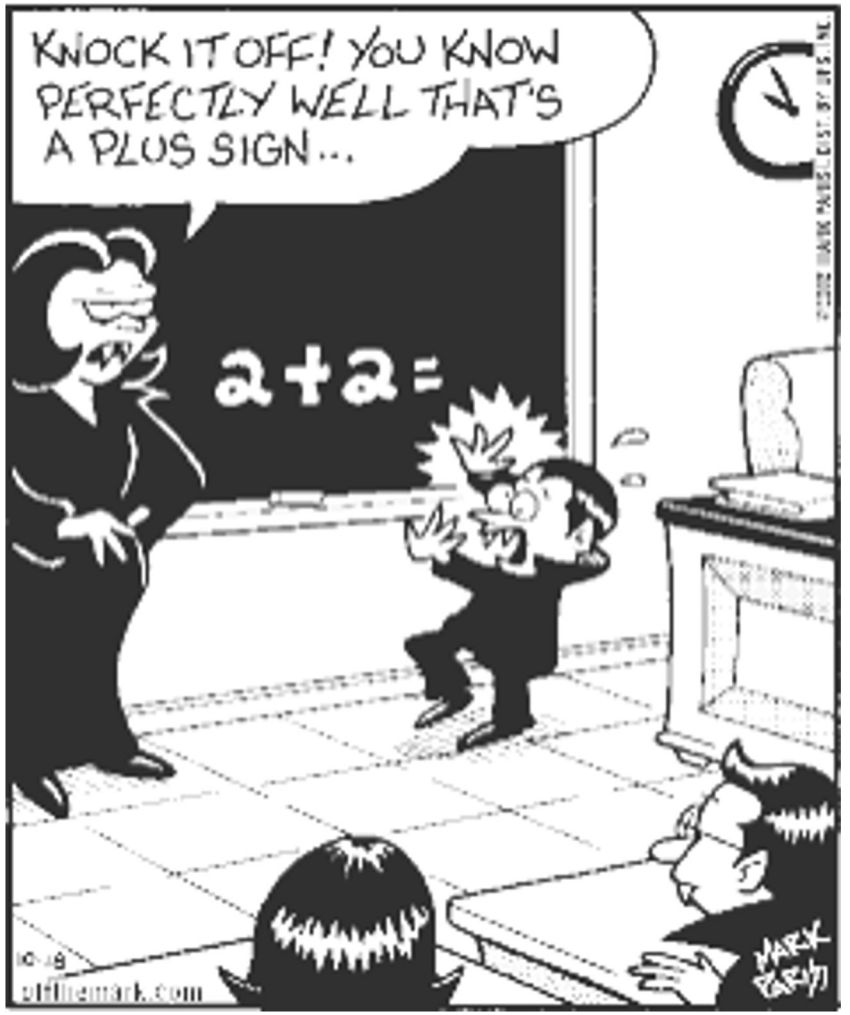
- To add binary numbers is relatively simple
- Same principles of adding normal decimals
- Result needs to fit in with the rules of binary

Fundamental rules of binary addition

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10

Consider

Why do we use the "carry" method?



Binary addition

- Decimal addition is simple:

Binary addition

- Decimal addition is simple:

```
  8
+7
=
```

Binary addition

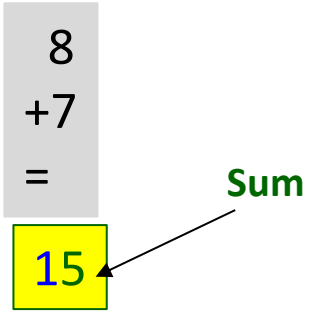
- Decimal addition is simple:

```
  8
+7
=
```

15

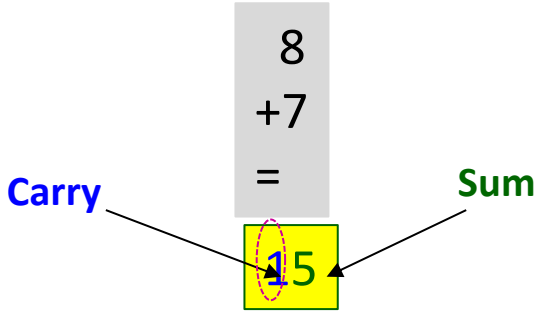
Binary addition

- Decimal addition is simple:



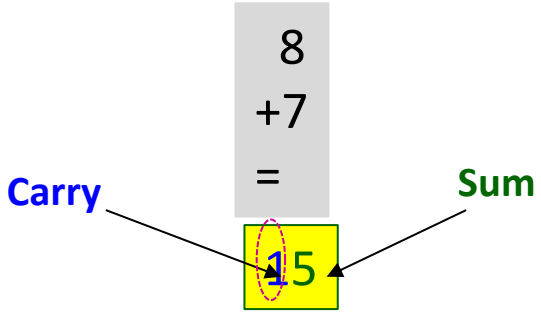
Binary addition

- Decimal addition is simple:



Binary addition

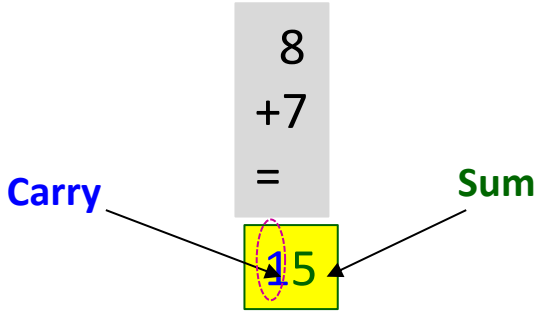
- Decimal addition is simple:



- Binary addition ➔ same concept

Binary addition

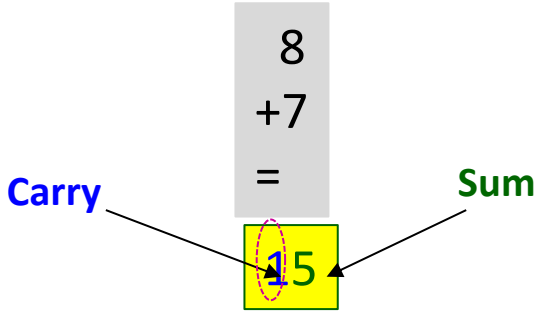
- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0

Binary addition

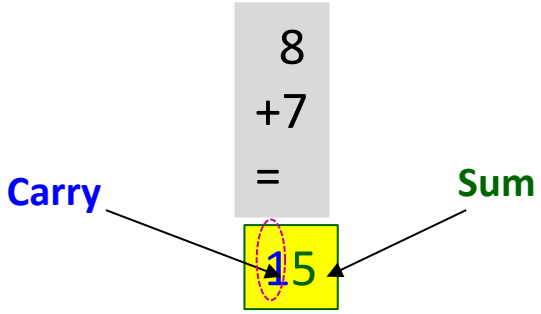
- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1

Binary addition

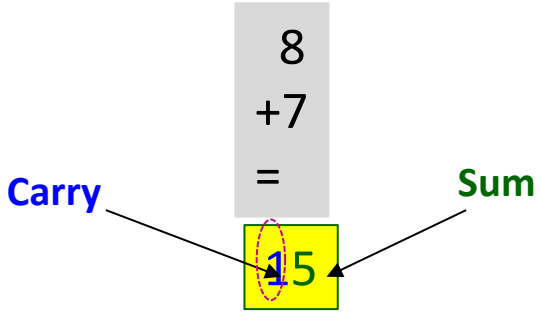
- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Binary addition

- Decimal addition is simple:



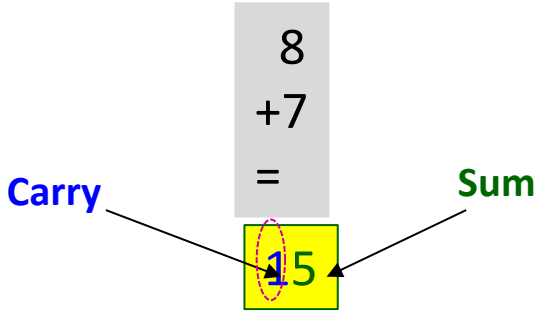
- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

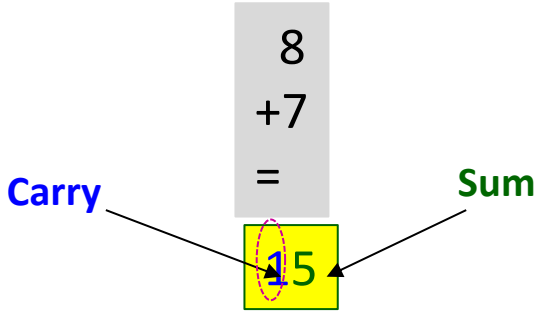
Sum

Carry

- It requires 2 outputs to represent sum and carry

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

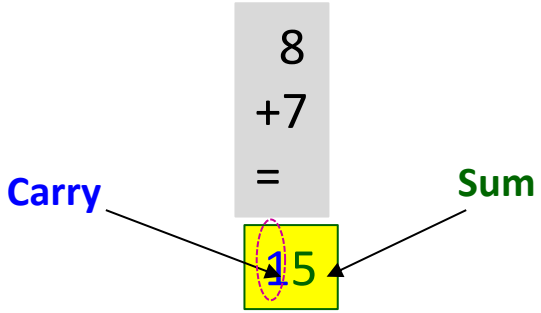
Carry

- It requires 2 outputs to represent sum and carry

- Multibit binary addition
  - Applies the same concept

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

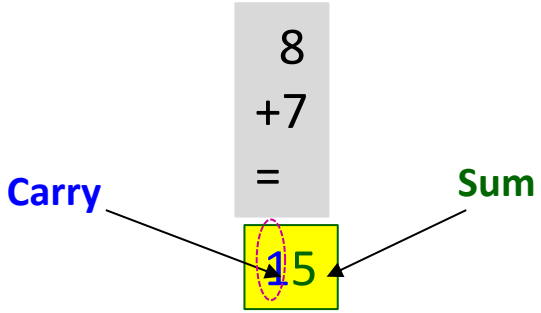
- It requires 2 outputs to represent sum and carry

- Multibit binary addition
  - Applies the same concept

e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

- It requires 2 outputs to represent sum and carry

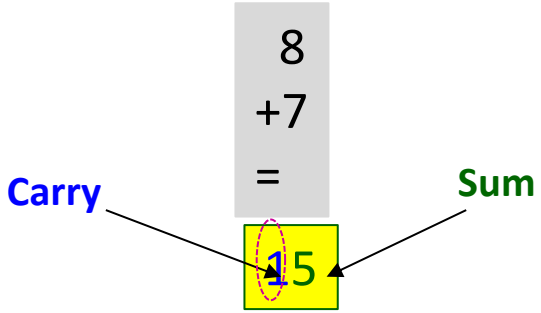
- Multibit binary addition
  - Applies the same concept

e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

```
  0110
+ 1011
------
```

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

- It requires 2 outputs to represent sum and carry

- Multibit binary addition
  - Applies the same concept

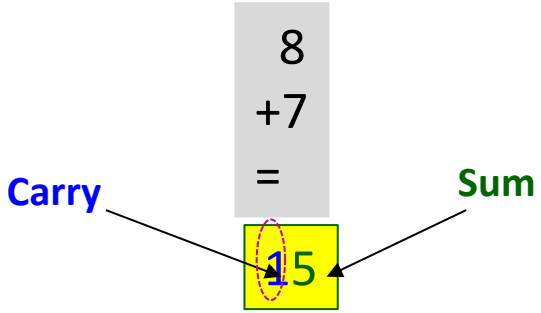
e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

```
   0110
+  1011
------
1
```

Sum

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

- It requires 2 outputs to represent sum and carry

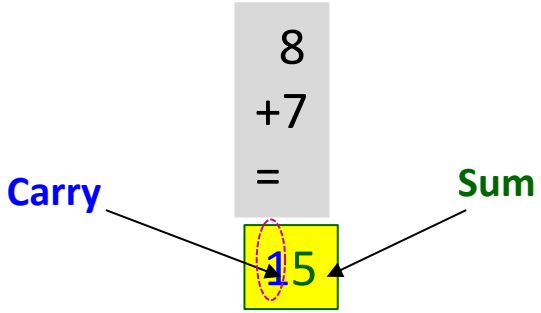
- Multibit binary addition
  - Applies the same concept

e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

```
Carry      0
           0110
         + 1011
-----------------
Sum        1
```

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

- It requires 2 outputs to represent sum and carry

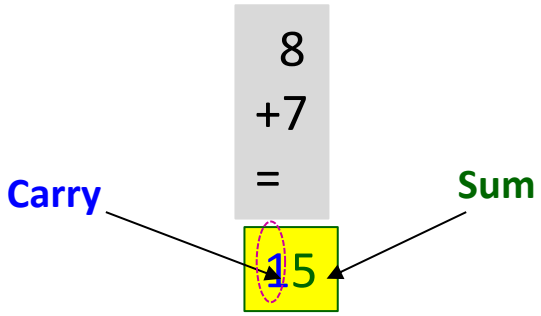
- Multibit binary addition
  - Applies the same concept

e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

```
        1 0
Carry   ^^
          0110
        + 1011
        ------
Sum       01
```

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

- It requires 2 outputs to represent sum and carry

- Multibit binary addition
  - Applies the same concept

e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

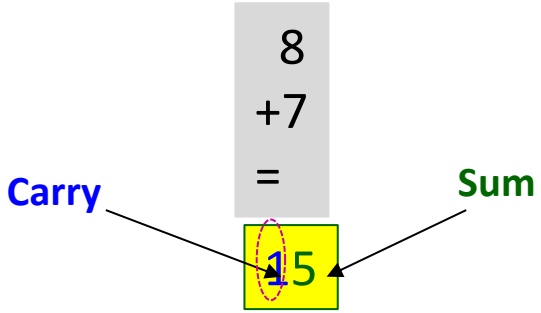
Carry 1 1 0

```
  0110
+ 1011
------
  0001
```

Sum

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

- It requires 2 outputs to represent sum and carry

- Multibit binary addition
  - Applies the same concept

e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

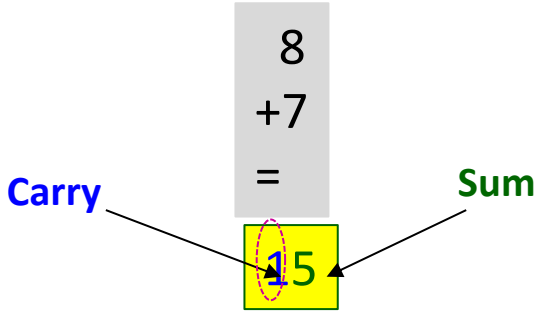
Carry 1 1 0

```
  0110
+ 1011
-------
 10001
```

Sum

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

- It requires 2 outputs to represent sum and carry

- Multibit binary addition
  - Applies the same concept

e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

Carry 1 1 0

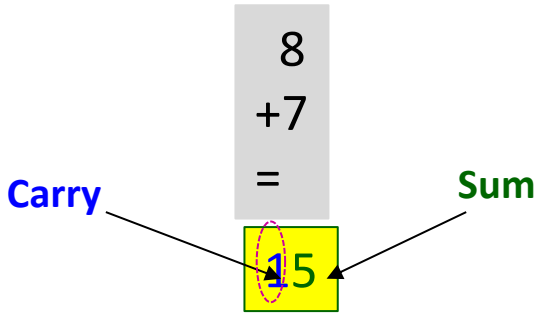
```
  0110
+ 1011
-------
 10001
```

Sum

Addition result = 10001 (17)

Binary addition

- Decimal addition is simple:



- Binary addition ➔ same concept
  - 0 + 0 = 0 0
  - 0 + 1 = 0 1
  - 1 + 0 = 0 1
  - 1 + 1 = 1 0

Sum

Carry

- It requires 2 outputs to represent sum and carry

- Multibit binary addition
  - Applies the same concept

e.g. add two 4-bit numbers
A= 0110 (6) & B=1011 (11)

Carry 1 1 0

```
  0110
+ 1011
------
 10001
```

Sum

Addition result = 10001 (17)

- How can it be implemented?

Basic addition and counting

How are the fundamental addition rules implemented in hardware?

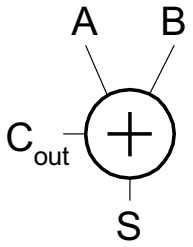
Half Adder

Basic addition and counting

How are the fundamental addition rules implemented in hardware?

Half Adder

- Two 1-bit inputs
- Two 1-bit outputs

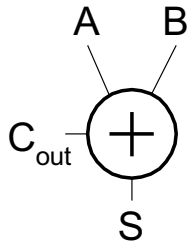


Basic addition and counting

How are the fundamental addition rules implemented in hardware?

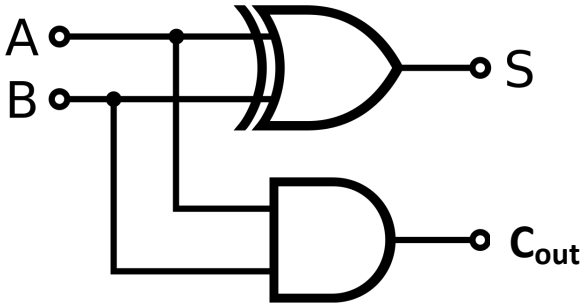
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B$$

$$C_{out} = A \bullet B$$

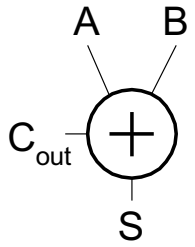


Basic addition and counting

How are the fundamental addition rules implemented in hardware?

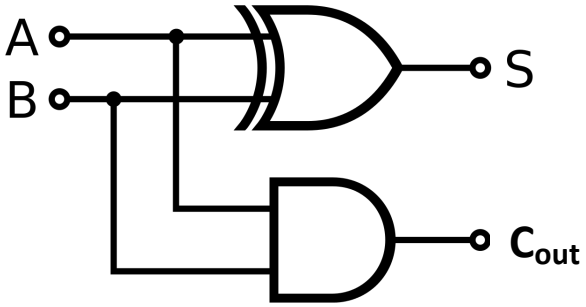
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B$$

$$C_{out} = A \bullet B$$



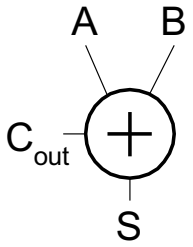
What happens when there is a third/carry input?

Basic addition and counting

How are the fundamental addition rules implemented in hardware?

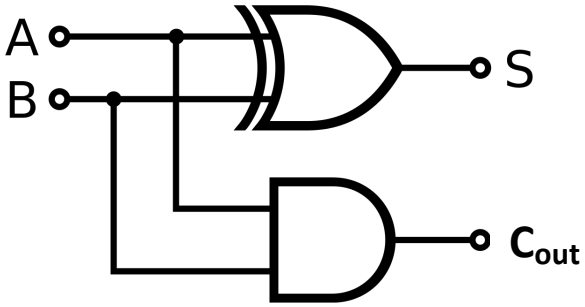
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



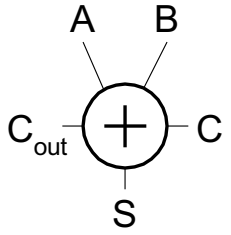
$$S = A \oplus B$$

$$C_{out} = A \bullet B$$



Full Adder

- Three 1-bit inputs
- Two 1-bit outputs



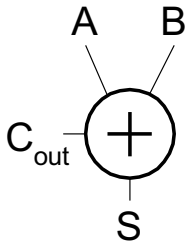
What happens when there is a third/carry input?

Basic addition and counting

How are the fundamental addition rules implemented in hardware?

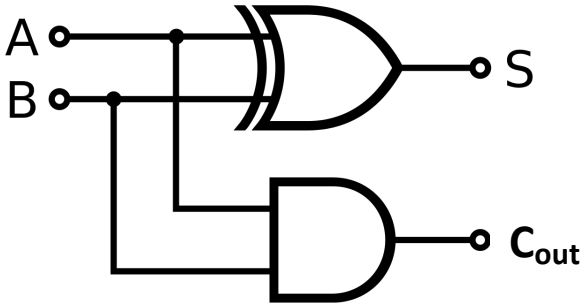
Half Adder

- Two 1-bit inputs
- Two 1-bit outputs



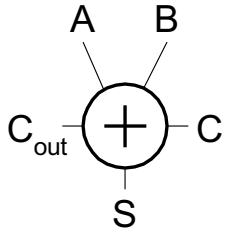
$$S = A \oplus B$$

$$C_{out} = A \bullet B$$



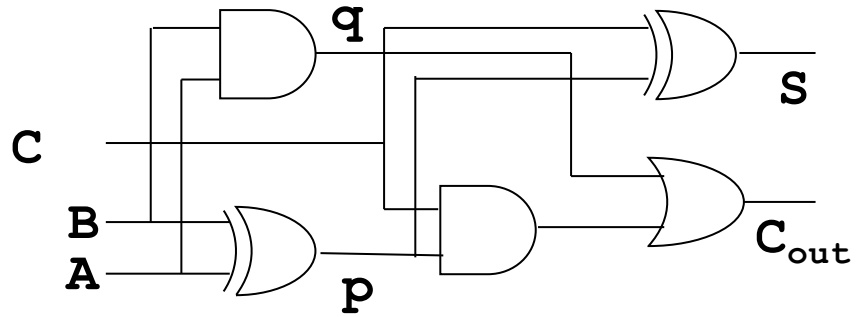
Full Adder

- Three 1-bit inputs
- Two 1-bit outputs



$$S = A \oplus B \oplus C$$

$$C_{out} = MAJ(A, B, C)$$



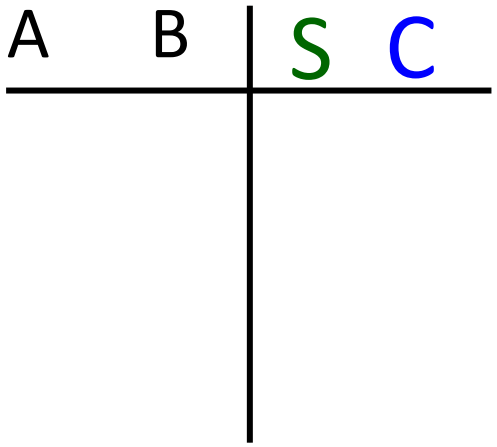
What happens when there is a third/carry input?

Simple adder design: half adder

- Performs 1-bit addition

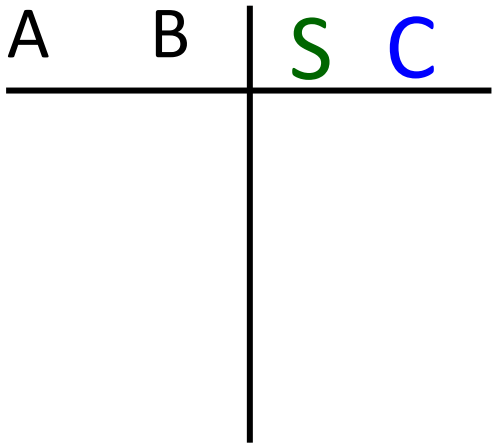
Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C



Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table



Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table

A	B	S	C
0	0	0	0
0	1	1	0

Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression
  - Using K-map

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression
  - Using K-map

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression
  - Using K-map

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

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

Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression
  - Using K-map

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

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

$S = A \oplus B$

Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression
  - Using K-map

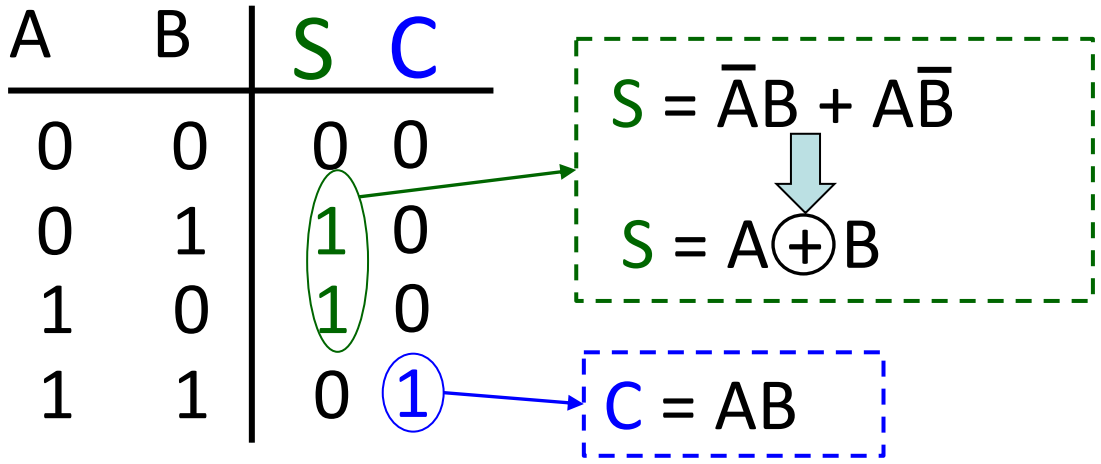
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

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

$S = A \oplus B$

Simple adder design: half adder

- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression
  - Using K-map



Simple adder design: half adder

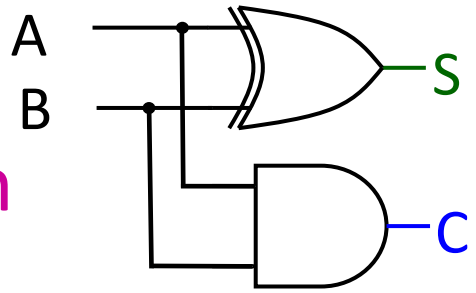
- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression
  - Using K-map
- Implement the Boolean expression with logic gates

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

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

$S = A \oplus B$

$C = AB$



Simple adder design: half adder

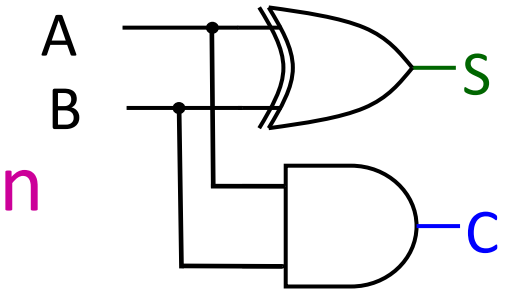
- Performs 1-bit addition
  - Inputs: A, B, Outputs: S, C
  - Generate truth table
- Extract Boolean expression
  - Using K-map
- Implement the Boolean expression with logic gates
- Half adder block diagram

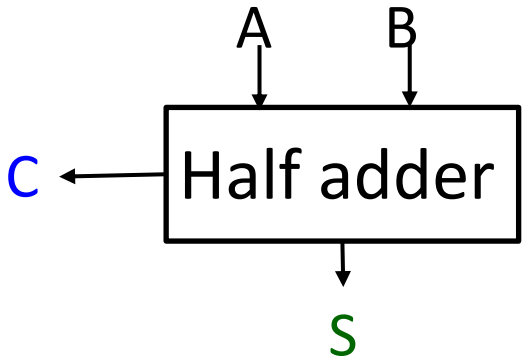
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

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

$S = A \oplus B$

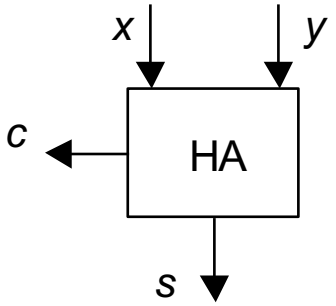
$C = AB$





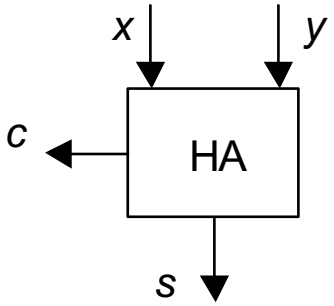
Half adder implementations

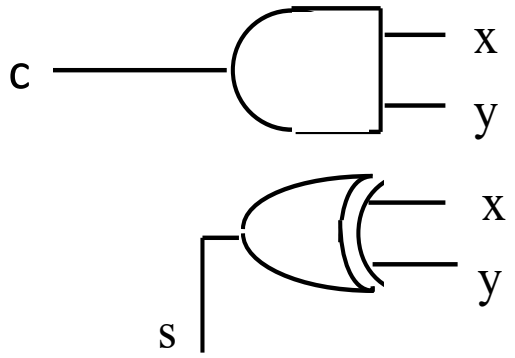
Inputs		Outputs	
x	*y*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



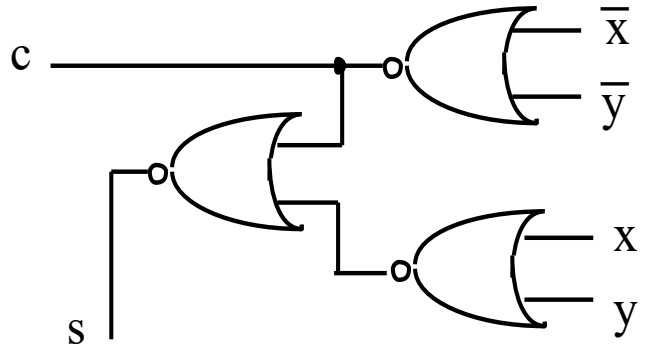
Half adder implementations

Inputs		Outputs	
x	*y*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

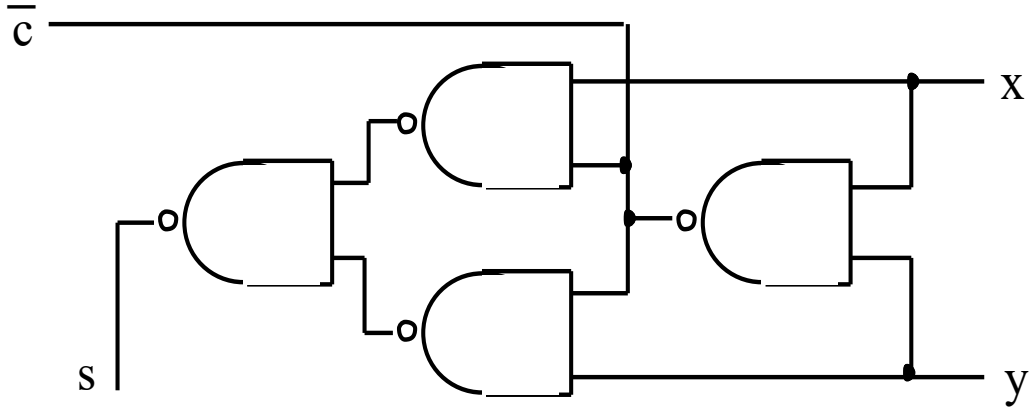




(a) AND/XOR half-adder.



(b) NOR-gate half-adder.



(c) NAND-gate half-adder with complemented carry.

Simple adder design: full adder

- Adds three 1-bit inputs

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o

A_i B_i C_i	S_o C_o

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table

A_i	B_i	C_i	S_o	C_o
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table

A_i	B_i	C_i	S_o	C_o
0	0	0	0	0
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table

A_i	B_i	C_i	S_o	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1		
1	0	0	1	0
1	0	1		
1	1	0		
1	1	1		

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table

A_i	B_i	C_i	S_o	C_o
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		

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table

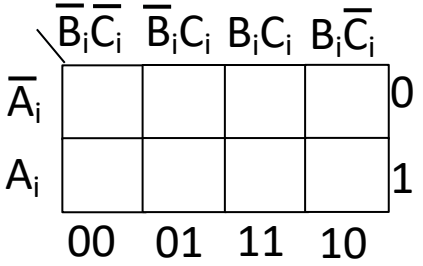
A_i	B_i	C_i	S_o	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table
- Extract Boolean expressions
  - Using K-map

A_i	B_i	C_i	S_o	C_o
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

K-map for S_o



Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table
- Extract Boolean expressions
  - Using K-map

A_i	B_i	C_i	S_o	C_o
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

K-map for S_o

A_i \ B_iC_i	$\overline{B}_i\overline{C}_i$	$\overline{B}_iC_i$	B_iC_i	$B_i\overline{C}_i$
$\overline{A}_i$		1		1
A_i	1		1	

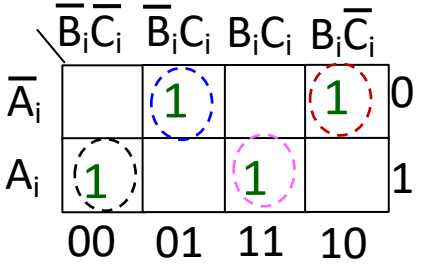
00 01 11 10

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table
- Extract Boolean expressions
  - Using K-map

A_i	B_i	C_i	S_o	C_o
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

K-map for S_o

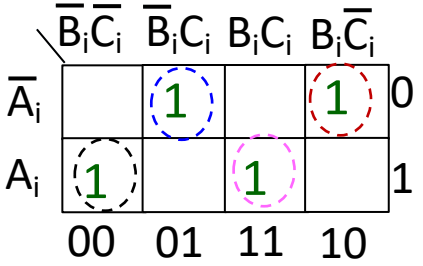


Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table
- Extract Boolean expressions
  - Using K-map

A_i	B_i	C_i	S_o	C_o
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

K-map for S_o



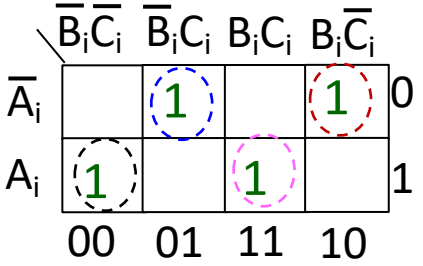
$S_o = \overline{A}_i\overline{B}_iC_i + \overline{A}_iB_i\overline{C}_i + A_i\overline{B}_i\overline{C}_i + A_iB_iC_i$

Simple adder design: full adder

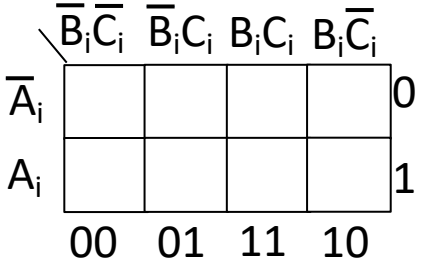
- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table
- Extract Boolean expressions
  - Using K-map

A_i	B_i	C_i	S_o	C_o
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

K-map for S_o



K-map for C_o



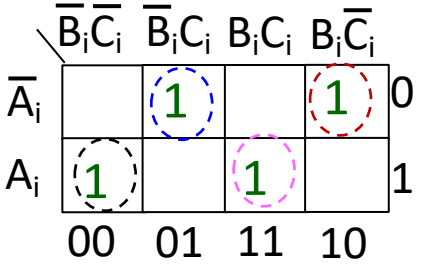
$S_o = \overline{A}_i\overline{B}_iC_i + \overline{A}_iB_i\overline{C}_i + A_i\overline{B}_i\overline{C}_i + A_iB_iC_i$

Simple adder design: full adder

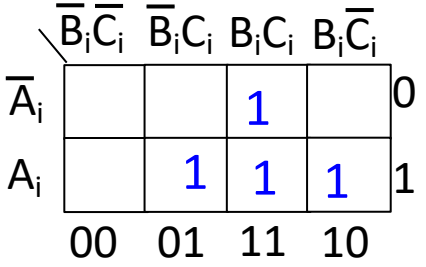
- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table
- Extract Boolean expressions
  - Using K-map

A_i	B_i	C_i	S_o	C_o
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

K-map for S_o



K-map for C_o



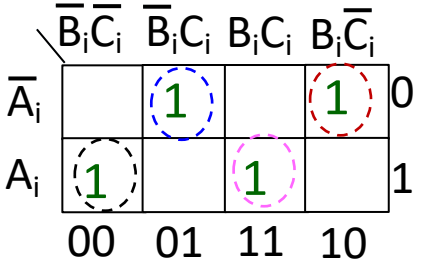
$S_o = \overline{A}_i\overline{B}_iC_i + \overline{A}_iB_i\overline{C}_i + A_i\overline{B}_i\overline{C}_i + A_iB_iC_i$

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table
- Extract Boolean expressions
  - Using K-map

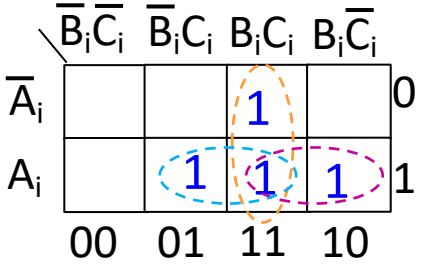
A_i	B_i	C_i	S_o	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

K-map for S_o



$S_o = \overline{A_i}\overline{B_i}C_i + \overline{A_i}B_i\overline{C_i} + A_i\overline{B_i}\overline{C_i} + A_iB_iC_i$

K-map for C_o



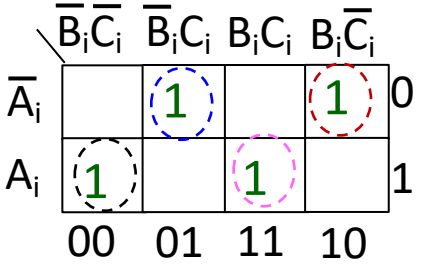
$C_o = A_iB_i + A_iC_i + B_iC_i$

Simple adder design: full adder

- Adds three 1-bit inputs
  - Inputs: A_i, B_i, C_i
  - Outputs: S_o, C_o
- Generate truth table
- Extract Boolean expressions
  - Using K-map
  - Simplify the expressions

A_i	B_i	C_i	S_o	C_o
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

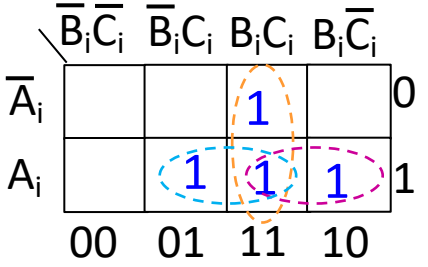
K-map for S_o



$S_o = \bar{A}_i\bar{B}_iC_i + \bar{A}_iB_i\bar{C}_i + A_i\bar{B}_i\bar{C}_i + A_iB_iC_i$

$S_o = A_i \oplus B_i \oplus C_i$

K-map for C_o



$C_o = A_iB_i + A_iC_i + B_iC_i$

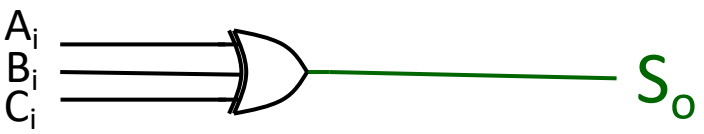
Full adder circuit implementation

- Combinational implementation

Full adder circuit implementation

- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

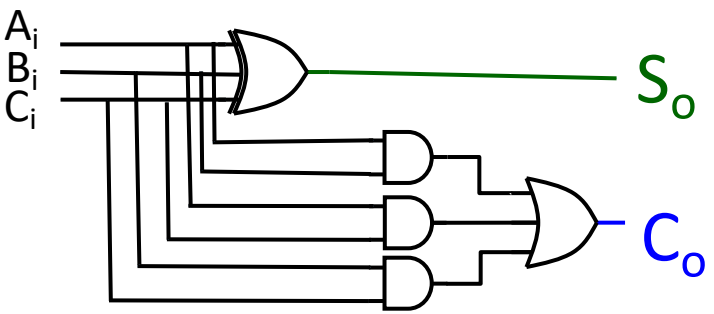


Full adder circuit implementation

- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

  $C_o = A_iB_i + A_iC_i + B_iC_i$

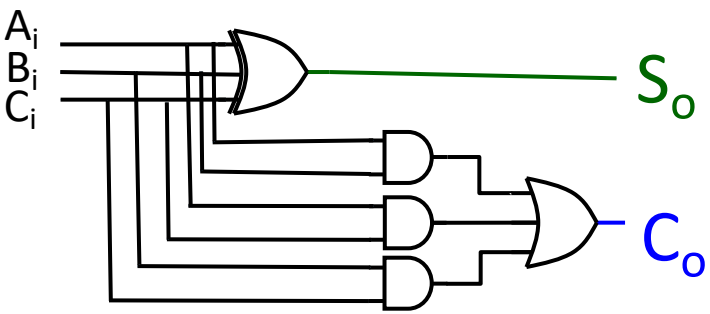


Full adder circuit implementation

- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

  $C_o = A_iB_i + A_iC_i + B_iC_i$



Is there any drawback?

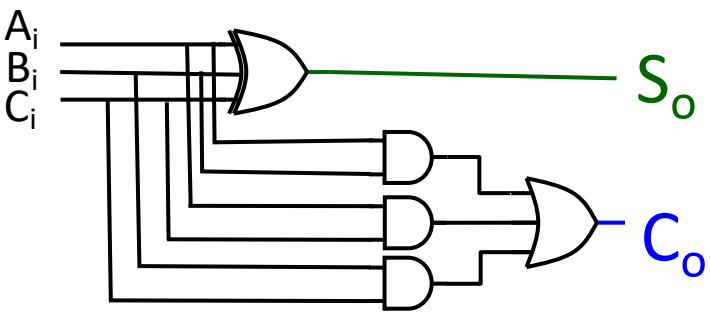
Full adder circuit implementation

- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

  $C_o = A_iB_i + A_iC_i + B_iC_i$

- Using two half adders



Is there any drawback?

Full adder circuit implementation

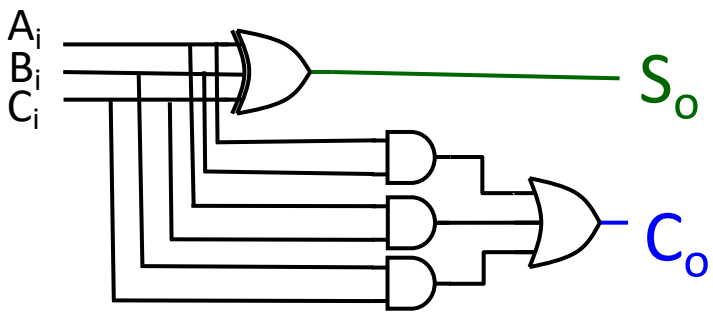
- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

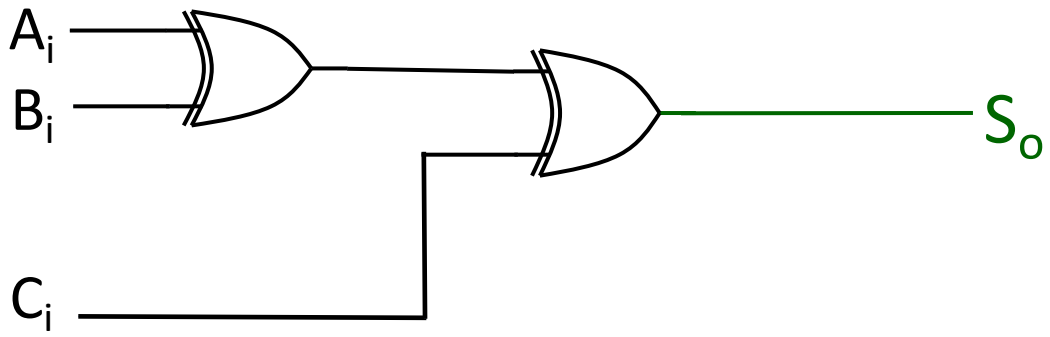
  $C_o = A_iB_i + A_iC_i + B_iC_i$

- Using two half adders

  $S_o = (A_i \oplus B_i) \oplus C_i$



Is there any drawback?

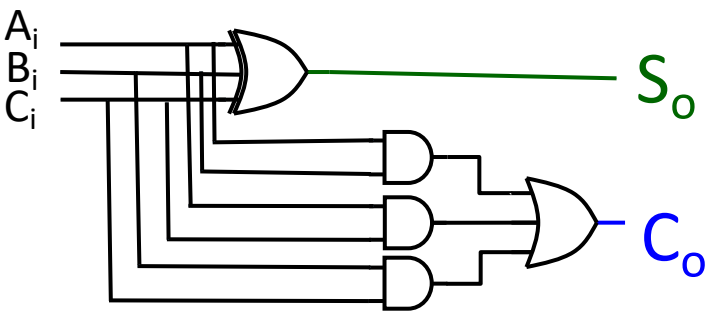


Full adder circuit implementation

- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

  $C_o = A_iB_i + A_iC_i + B_iC_i$

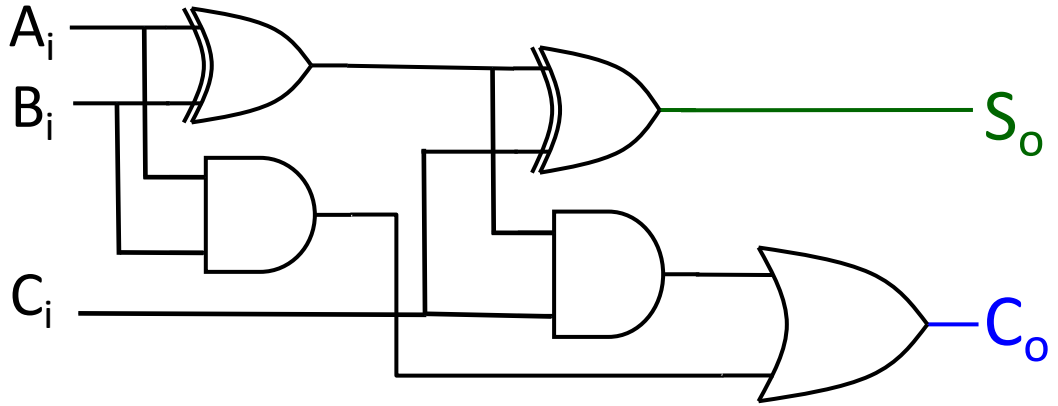


Is there any drawback?

- Using two half adders

  $S_o = (A_i \oplus B_i) \oplus C_i$

  $C_o = A_iB_i + C_i\,(A_i \oplus B_i)$

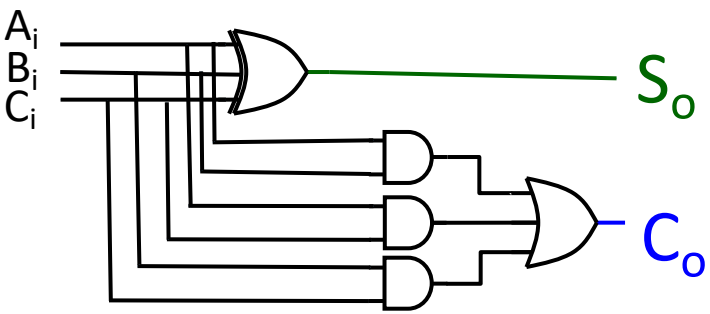


Full adder circuit implementation

- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

  $C_o = A_iB_i + A_iC_i + B_iC_i$

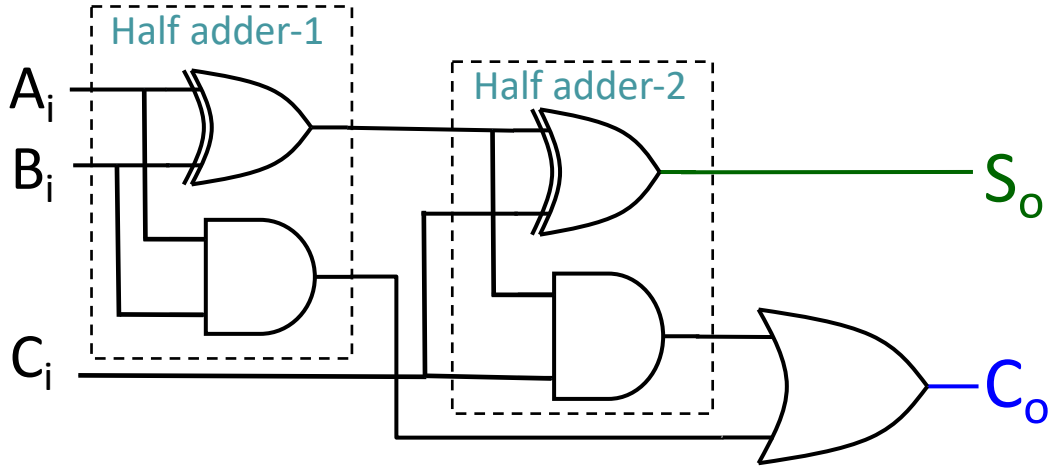


Is there any drawback?

- Using two half adders

  $S_o = (A_i \oplus B_i) \oplus C_i$

  $C_o = A_iB_i + C_i\,(A_i \oplus B_i)$

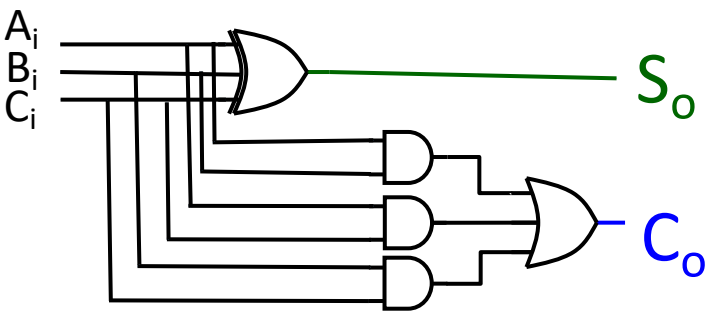


Full adder circuit implementation

- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

  $C_o = A_iB_i + A_iC_i + B_iC_i$

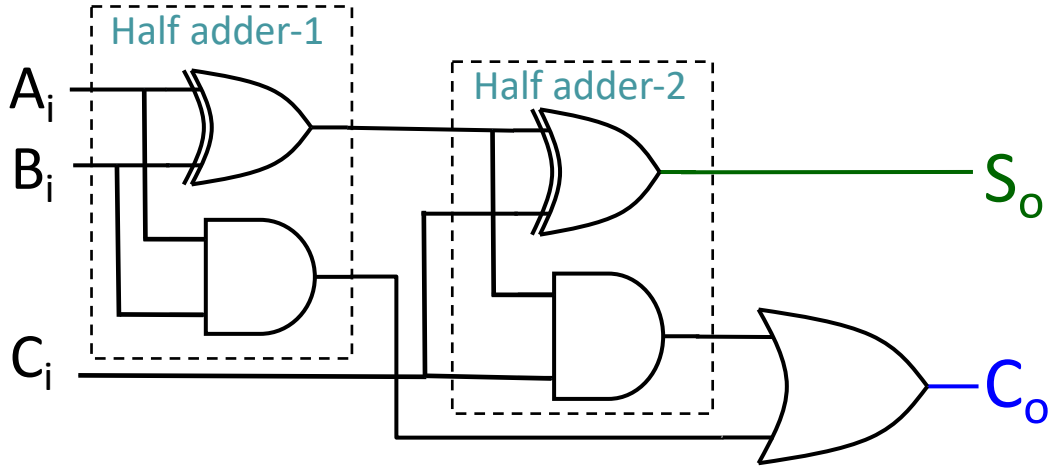


Is there any drawback?

- Using two half adders

  $S_o = (A_i \oplus B_i) \oplus C_i$

  $C_o = A_iB_i + C_i\,(A_i \oplus B_i)$



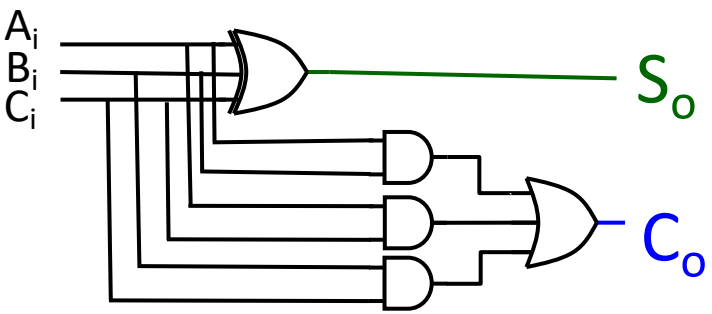
- Full Adder (FA) block diagram

Full adder circuit implementation

- Combinational implementation

  $S_o = A_i \oplus B_i \oplus C_i$

  $C_o = A_iB_i + A_iC_i + B_iC_i$

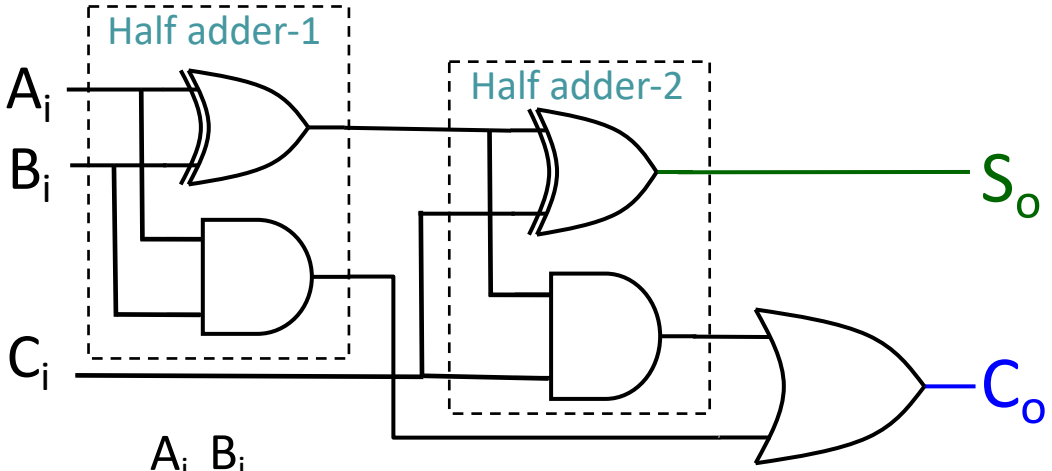


Is there any drawback?

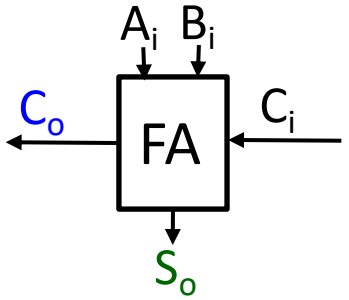
- Using two half adders

  $S_o = (A_i \oplus B_i) \oplus C_i$

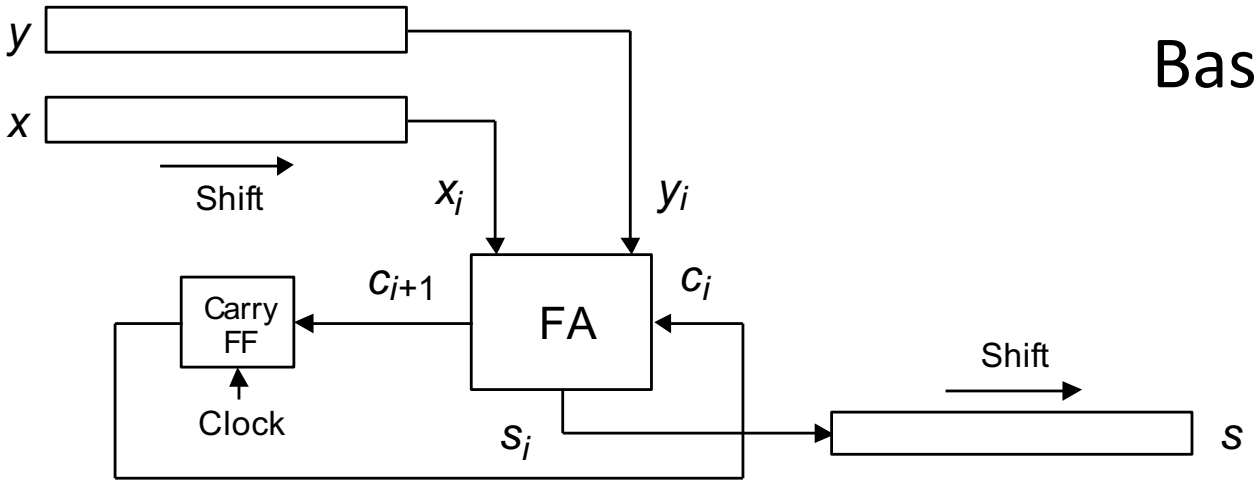
  $C_o = A_iB_i + C_i\,(A_i \oplus B_i)$



- Full Adder (FA) block diagram



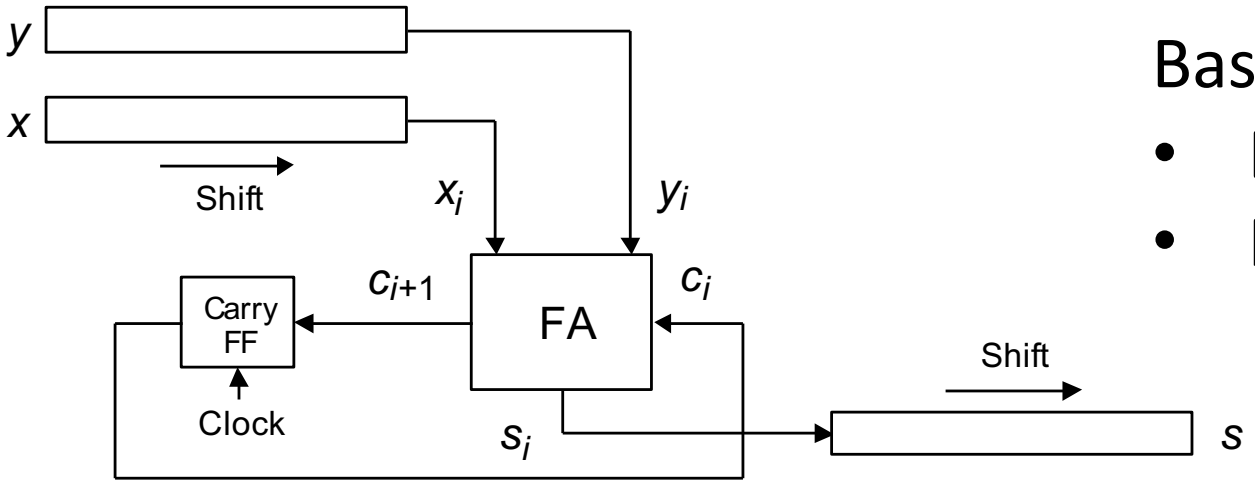
Building simple adders using full adder



(a) Bit-serial adder.

Basics of bit-serial adder

Building simple adders using full adder

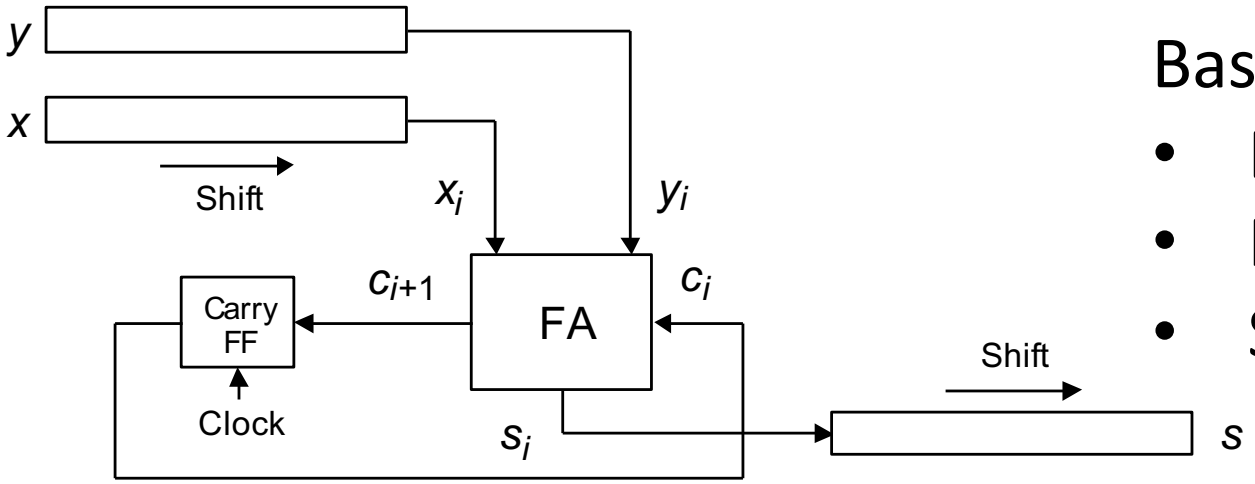


(a) Bit-serial adder.

Basics of bit-serial adder

- Bulit using one full adder, carry FF
- Result is accumulated in a shift register

Building simple adders using full adder

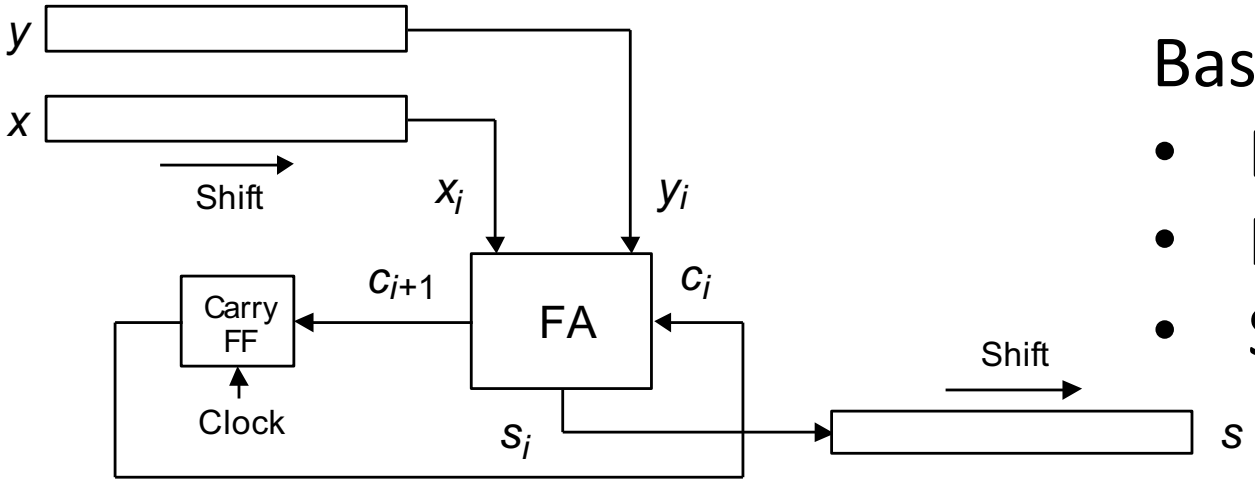


(a) Bit-serial adder.

Basics of bit-serial adder

- Bulit using one full adder, carry FF
- Result is accumulated in a shift register
- Slow but area efficient

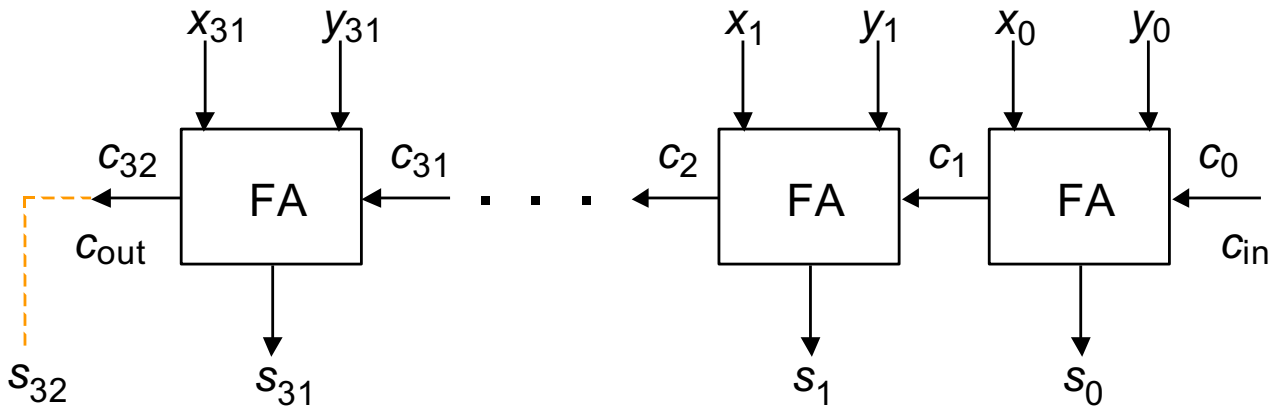
Building simple adders using full adder



(a) Bit-serial adder.

Basics of bit-serial adder

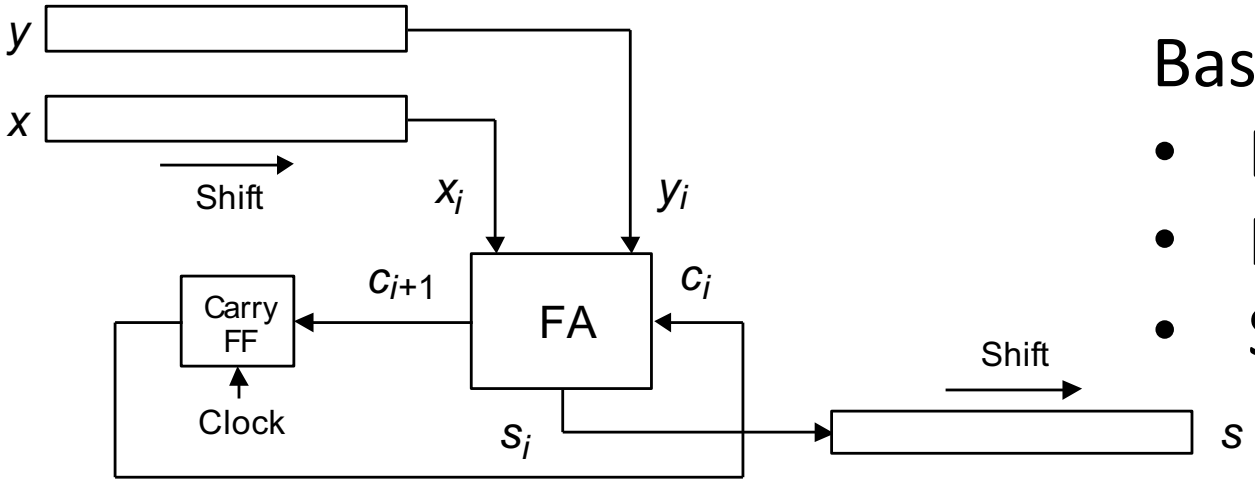
- Bulit using one full adder, carry FF
- Result is accumulated in a shift register
- Slow but area efficient



(b) Ripple-carry adder.

Basics of Ripple-carry adder

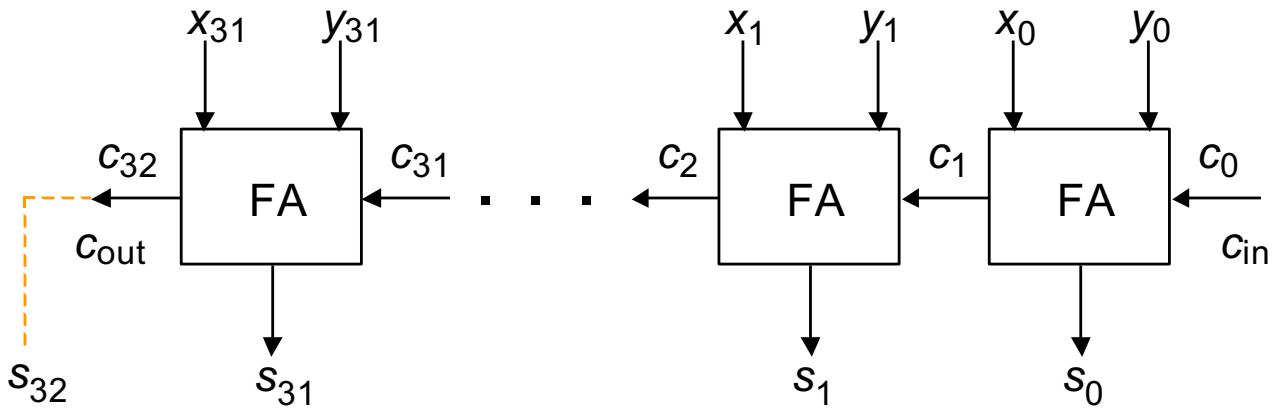
Building simple adders using full adder



(a) Bit-serial adder.

Basics of bit-serial adder

- Bulit using one full adder, carry FF
- Result is accumulated in a shift register
- Slow but area efficient

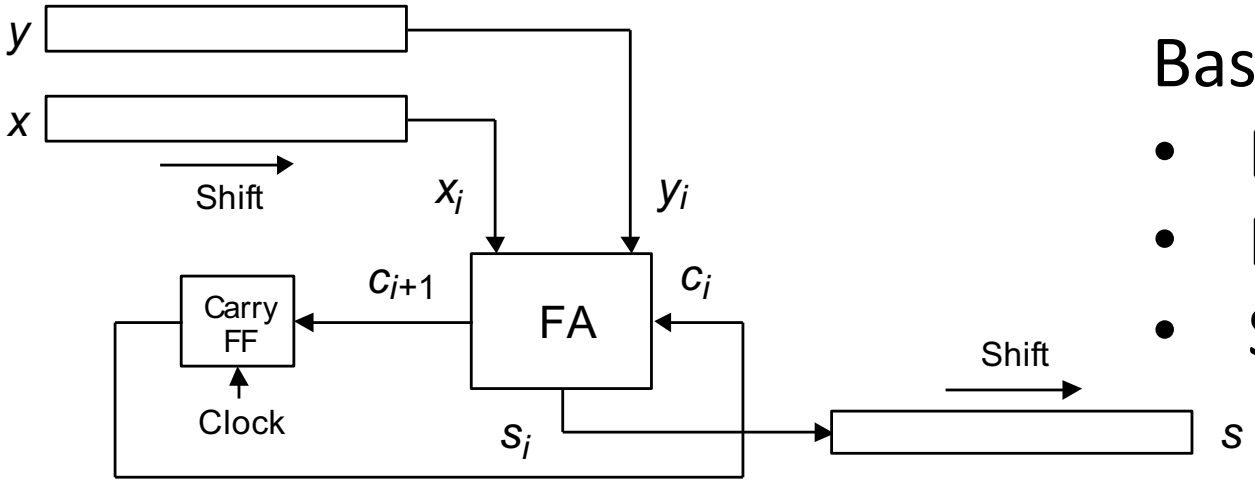


(b) Ripple-carry adder.

Basics of Ripple-carry adder

- Bulit using ‘n’ full adder
- Carry ripples in the carry chain
- Faster than bit serial adder

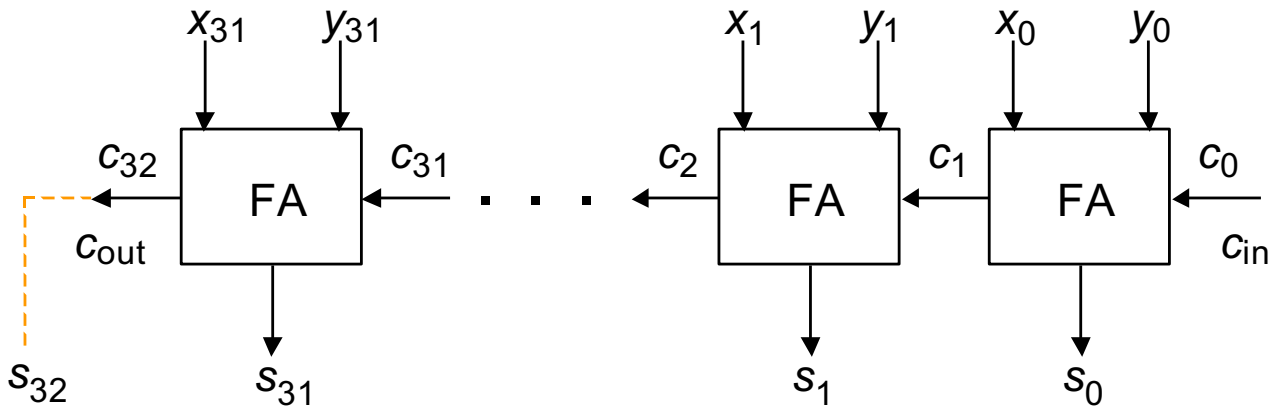
Building simple adders using full adder



(a) Bit-serial adder.

Basics of bit-serial adder

- Bulit using one full adder, carry FF
- Result is accumulated in a shift register
- Slow but area efficient



(b) Ripple-carry adder.

Basics of Ripple-carry adder

- Bulit using ‘n’ full adder
- Carry ripples in the carry chain
- Faster than bit serial adder
- Less area efficient than bit serial

Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA

Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA

Previous e.g., A= 0110, B=1011 & $C_0 = 0$

Carry $0 \Leftarrow C_0$

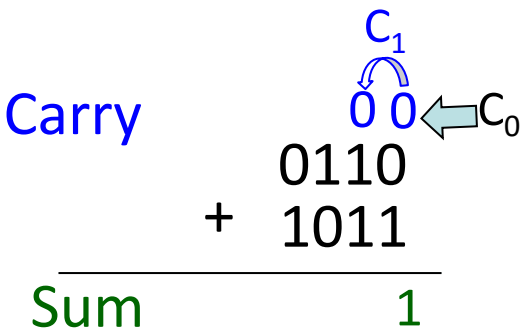
$$\begin{array}{r} 0110 \\ +\ 1011 \\ \hline \end{array}$$

Sum

Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA

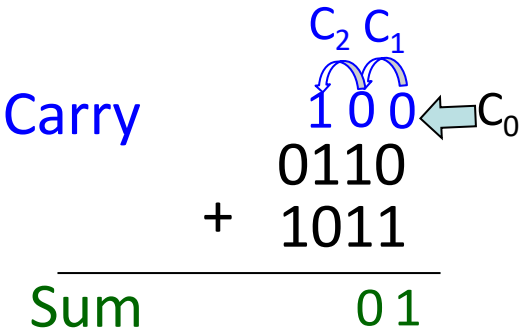
Previous e.g., A= 0110, B=1011 & $C_0 = 0$



Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA

Previous e.g., A= 0110, B=1011 & $C_0 = 0$



Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA

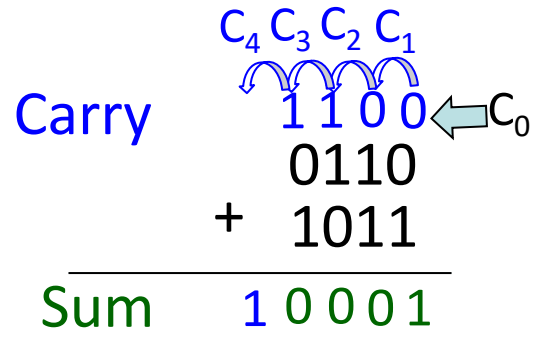
Previous e.g., A= 0110, B=1011 & $C_0 = 0$

		C_3	C_2	C_1	
Carry		1	1	0	0 ⇦ C_0
		0	1	1	0
	+	1	0	1	1
Sum		0	0	1	

Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA

Previous e.g., A= 0110, B=1011 & $C_0 = 0$



Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA

Previous e.g., A= 0110, B=1011 & $C_0 = 0$

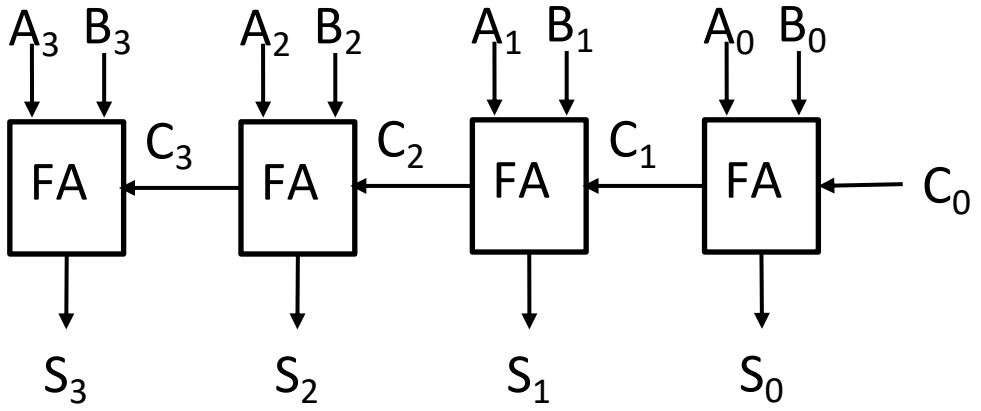
Carry $C_4\ C_3\ C_2\ C_1$: 1 1 0 0 ⇦ C_0

0110

\+ 1011

Sum 1 0 0 0 1

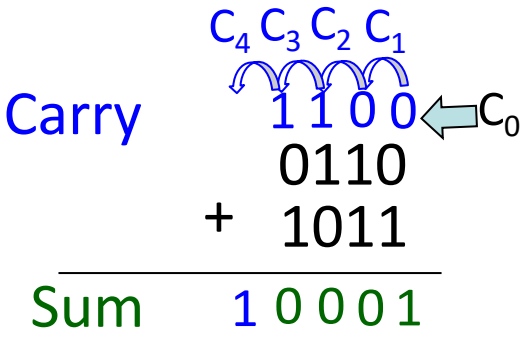
- RCA uses n full adder blocks in parallel (4 in this case)

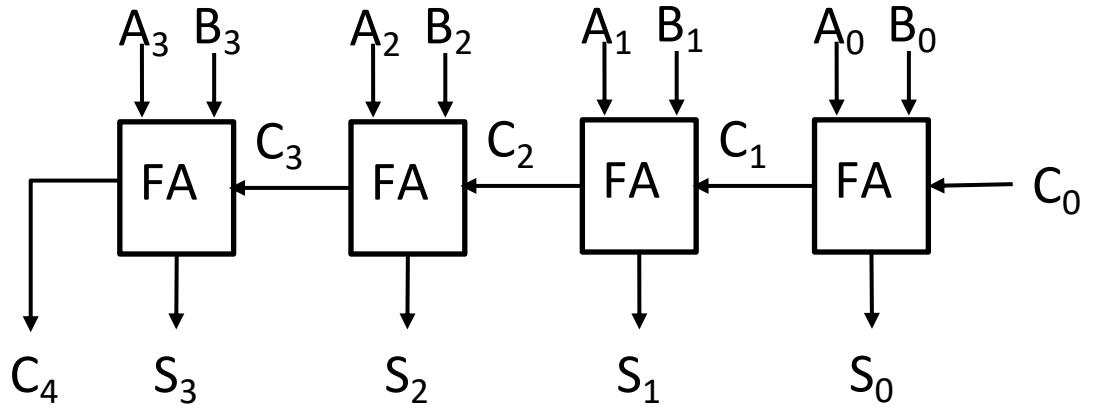


Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA
- RCA uses n full adder blocks in parallel (4 in this case)
- Cascade full adders & establish carry chain

Previous e.g., A= 0110, B=1011 & $C_0 = 0$

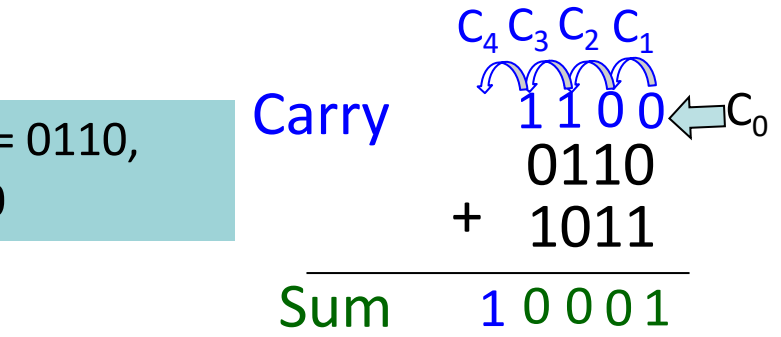


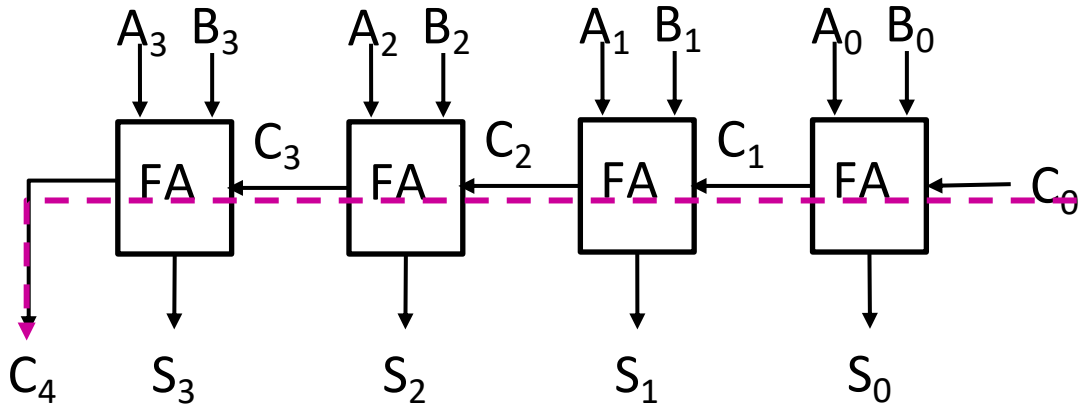


Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA
- RCA uses n full adder blocks in parallel (4 in this case)
- Cascade full adders & establish carry chain

Previous e.g., A= 0110, B=1011 & $C_0 = 0$

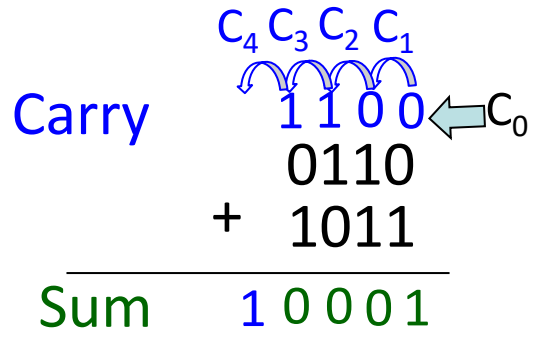




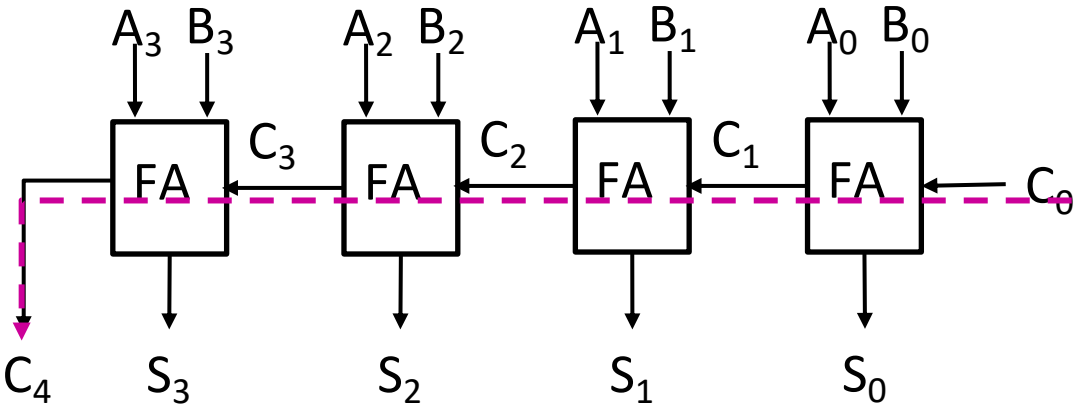
Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA

Previous e.g., A= 0110, B=1011 & $C_0 = 0$



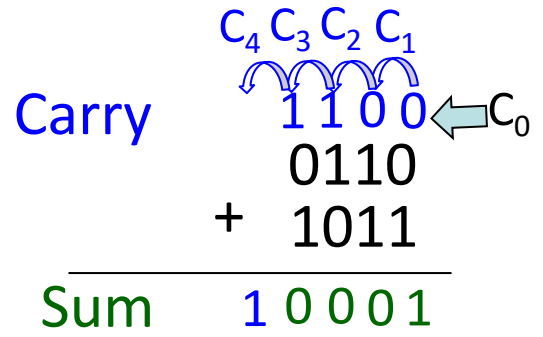
- RCA uses n full adder blocks in parallel (4 in this case)
- Cascade full adders & establish carry chain
- If the delay of a FA is Δ ➔the delay of n-bit RCA will be nΔ

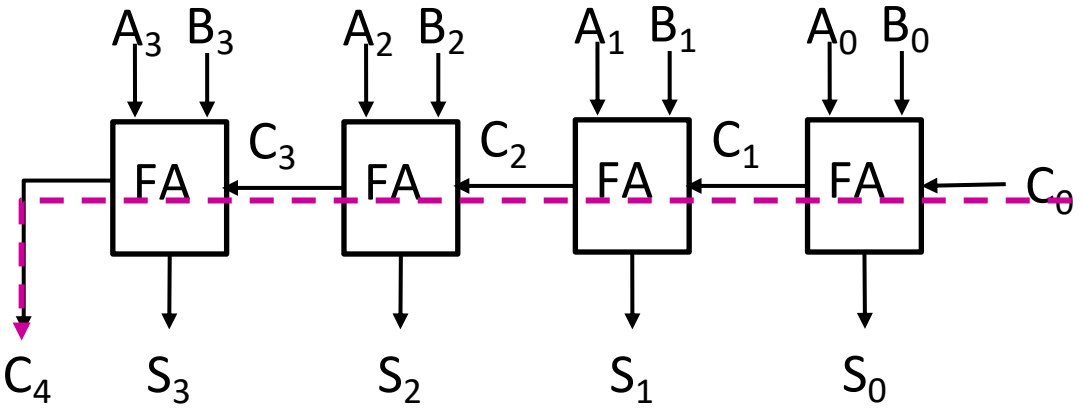


Multi-bit adder: Ripple-Carry Adder (RCA)

- Multi-bit addition can be realized with RCA
- RCA uses n full adder blocks in parallel (4 in this case)
- Cascade full adders & establish carry chain
- If the delay of a FA is Δ ➔the delay of n-bit RCA will be nΔ

Previous e.g., A= 0110, B=1011 & $C_0 = 0$

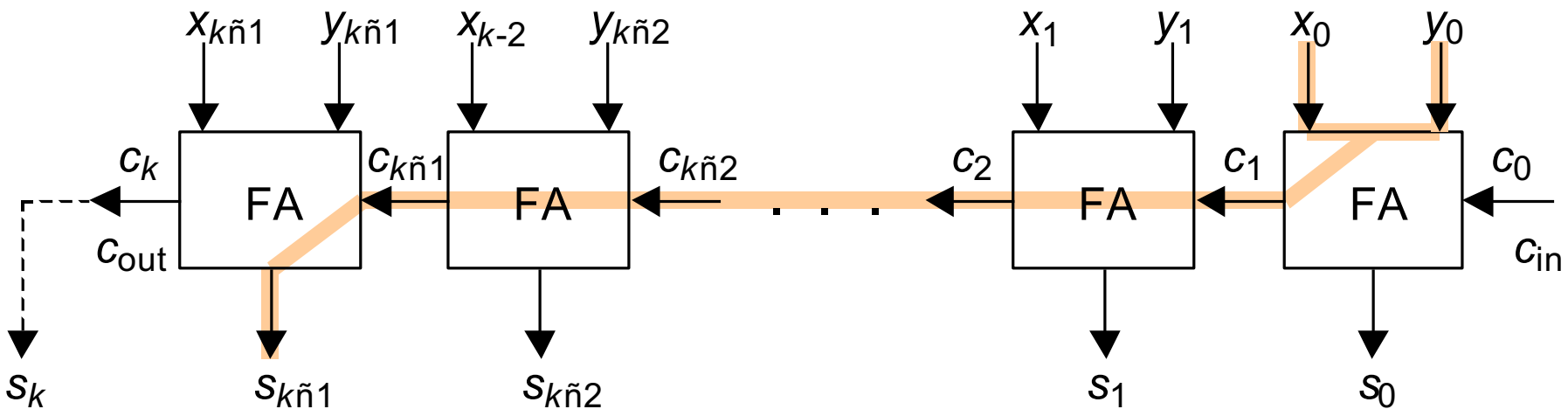




Can we do it faster?

Carry path of ripple-carry adder

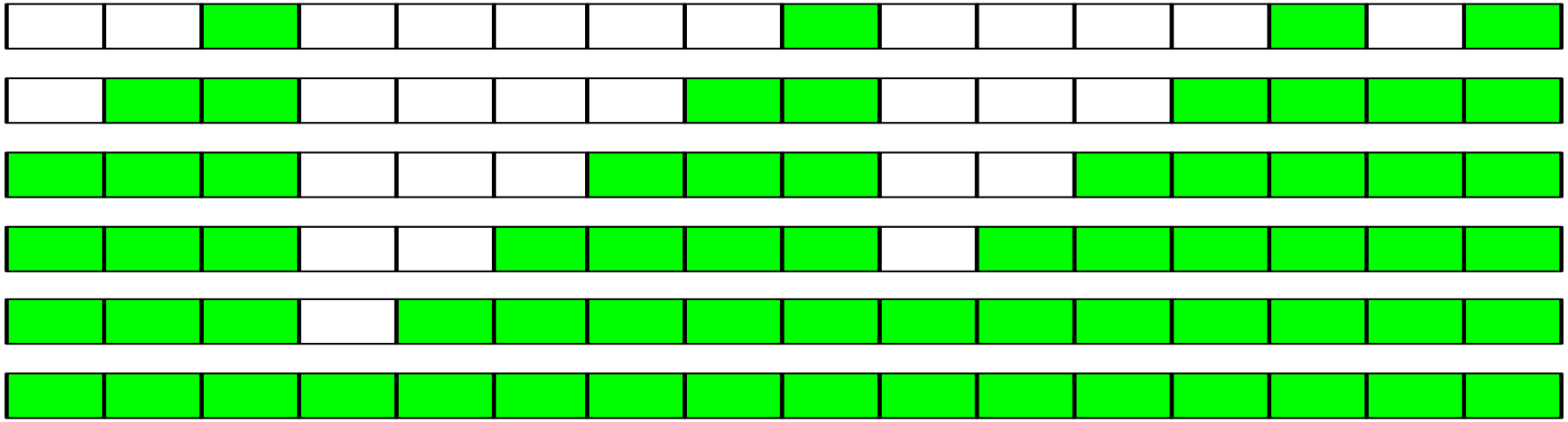
$$T_{\text{ripple-add}} = T_{FA}(x,y \rightarrow c_{out}) + (k-2) \times T_{FA}(c_{in} \rightarrow cout) + T_{FA}(c_{in} \rightarrow s)$$



Critical path in a *k*-bit ripple-carry adder.

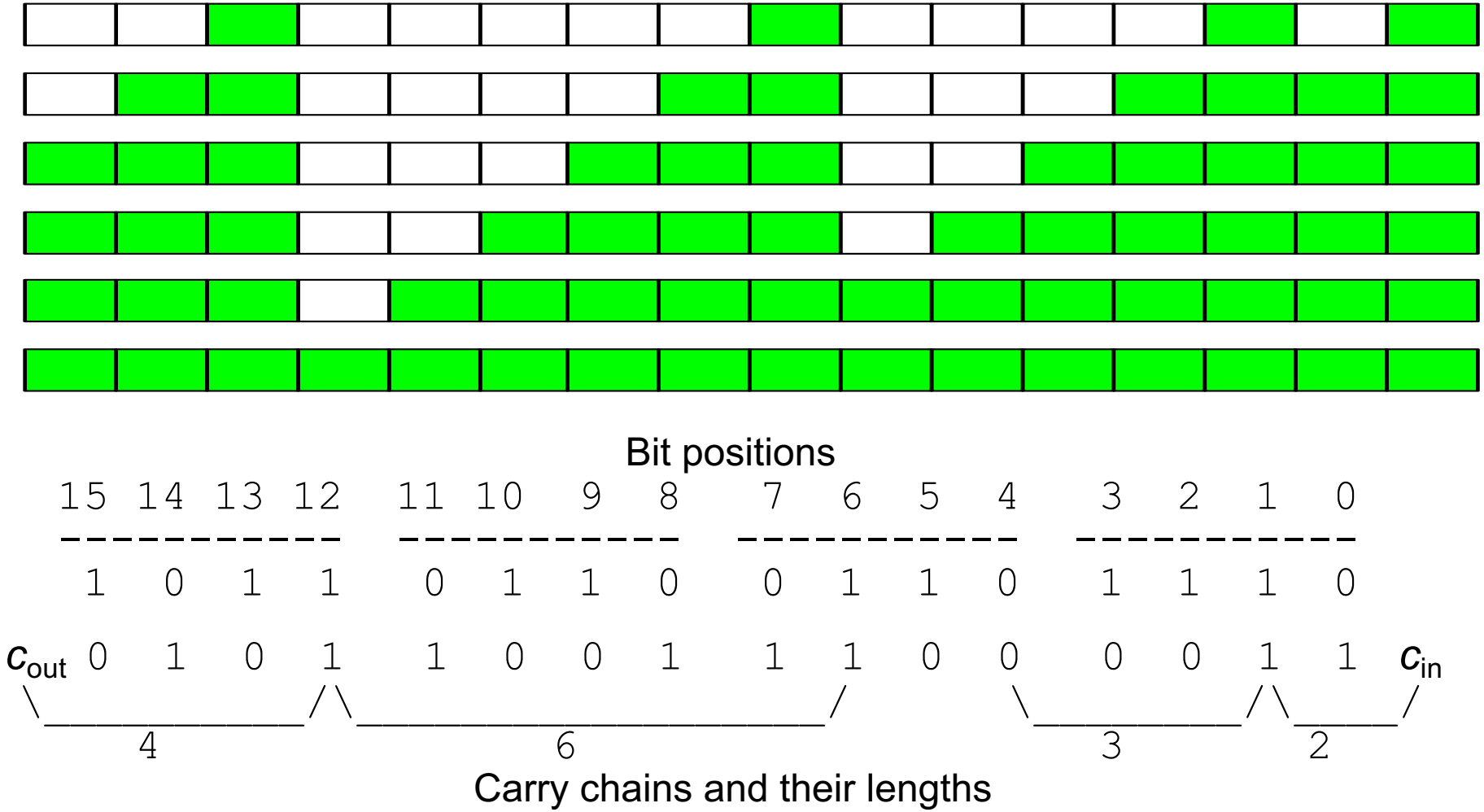
Carry path of ripple-carry adder

Critical path and carry propagation example in a *16*-bit ripple-carry addition



Carry path of ripple-carry adder

Critical path and carry propagation example in a *16*-bit ripple-carry addition



Carry propagation analysis

Given binary numbers with random bits, for each position *i* we have

- Average length of the longest carry chain in *k*-bit addition is strictly less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

Analogy: Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows

Carry propagation analysis

Given binary numbers with random bits, for each position *i* we have

Probability of carry generation = ¼ (both 1s)

- Average length of the longest carry chain in *k*-bit addition is strictly less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

Analogy: Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows

Carry propagation analysis

Given binary numbers with random bits, for each position *i* we have

Probability of carry generation = ¼ (both 1s)

Probability of carry annihilation = ¼ (both 0s)

- Average length of the longest carry chain in *k*-bit addition is strictly less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

Analogy: Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows

Carry propagation analysis

Given binary numbers with random bits, for each position *i* we have

Probability of carry generation = ¼ (both 1s)
Probability of carry annihilation = ¼ (both 0s)
Probability of carry propagation = ½ (different)

- Average length of the longest carry chain in *k*-bit addition is strictly less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

Analogy: Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows

Carry propagation analysis

Given binary numbers with random bits, for each position *i* we have

Probability of carry generation = ¼ (both 1s)
Probability of carry annihilation = ¼ (both 0s)
Probability of carry propagation = ½ (different)

- Probability that carry generated at position *i* propagates through position $j - 1$ and stops at position *j* ($j > i$)

$$2^{-(j-1-i)} \times 1/2 = 2^{-(j-i)}$$

- Average length of the longest carry chain in *k*-bit addition is strictly less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

Analogy: Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows

Carry propagation analysis

Given binary numbers with random bits, for each position *i* we have

Probability of carry generation = ¼ (both 1s)
Probability of carry annihilation = ¼ (both 0s)
Probability of carry propagation = ½ (different)

- Probability that carry generated at position *i* propagates through position $j - 1$ and stops at position *j* ($j > i$)

  $2^{-(j-1-i)} \times 1/2 = 2^{-(j-i)}$

- Expected length of the carry chain that starts at position I

  $2 - 2^{-(k-i-1)}$

- Average length of the longest carry chain in *k*-bit addition is strictly less than $\log_2 k$; it is $\log_2(1.25k)$ per experimental results

Analogy: Expected number when rolling one die is 3.5; if one rolls many dice, the expected value of the largest number shown grows

Counters

Counters are specialized adders:

Counters

Counters are specialized adders:

- One operand is stored in a register
- Start with fixed value/usually zero

Counters

Counters are specialized adders:

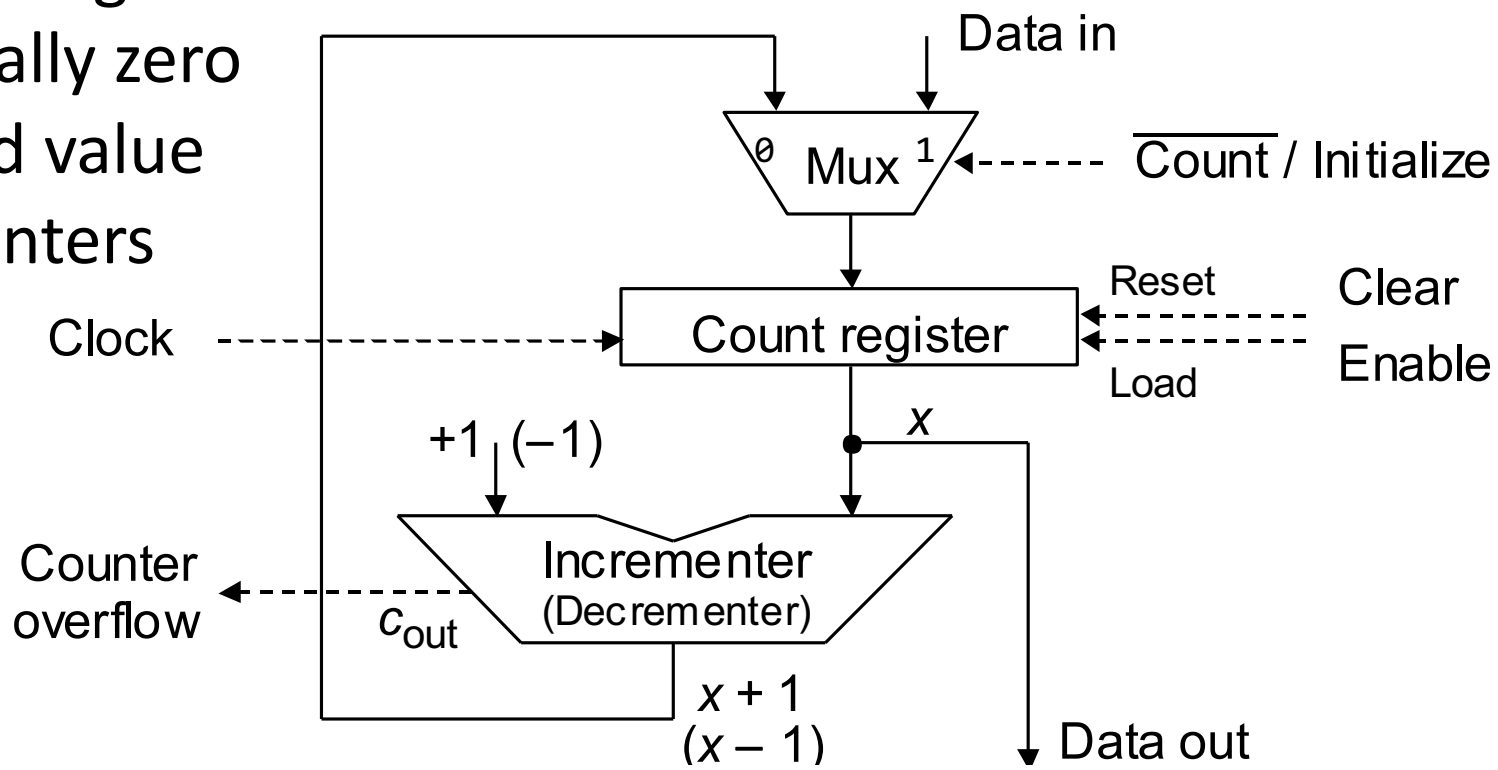
- One operand is stored in a register
- Start with fixed value/usually zero
- Add constant to the stored value
- They can be up/down counters

Counters

Counters are specialized adders:

- One operand is stored in a register
- Start with fixed value/usually zero
- Add constant to the stored value
- They can be up/down counters

An up (down) counter built of a register, an adder and a multiplexer.

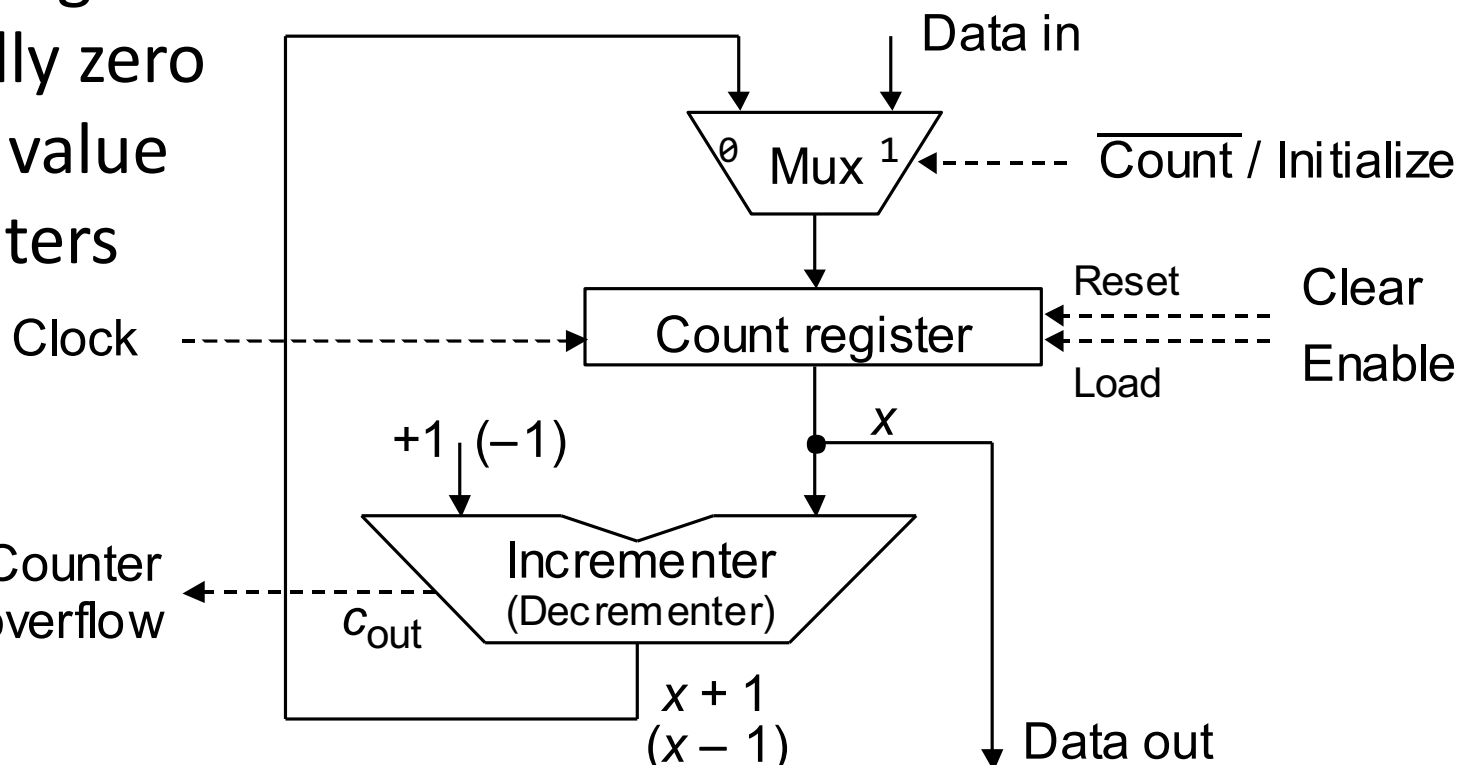


Counters

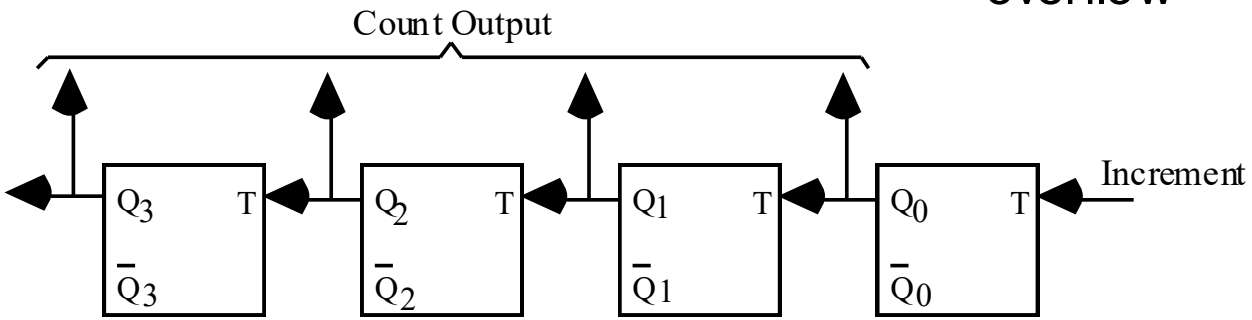
Counters are specialized adders:

- One operand is stored in a register
- Start with fixed value/usually zero
- Add constant to the stored value
- They can be up/down counters

An up (down) counter built of a register, an adder and a multiplexer.



Four-bit asynchronous up counter built using T flip-flops



Fast Addition

Carry network is the essence of fast adders

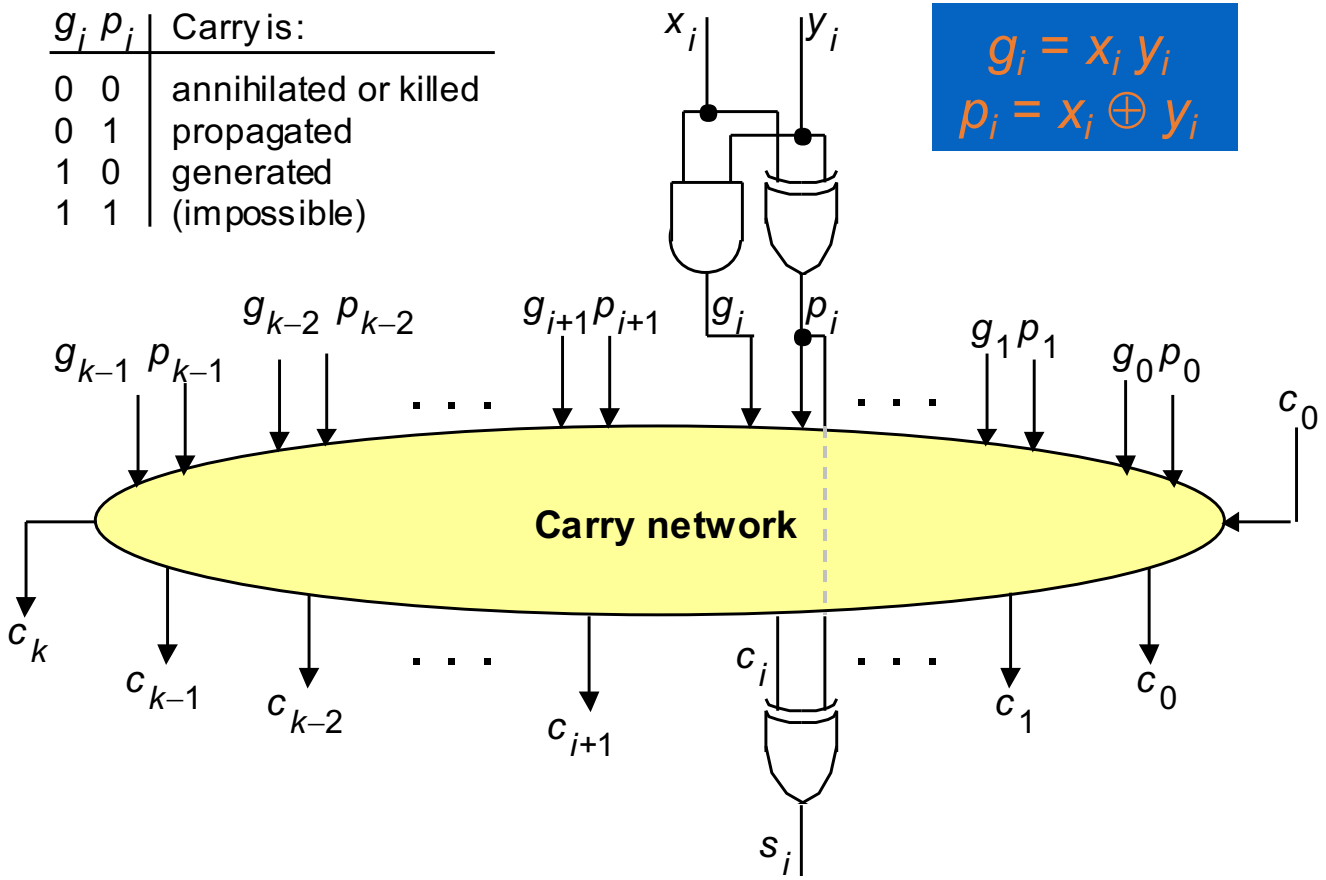
Fast Addition

Carry network is the essence of fast adders

g_i	p_i	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)

$g_i = x_i\, y_i$

$p_i = x_i \oplus y_i$



Generic structure of a binary adder, highlighting its carry network.

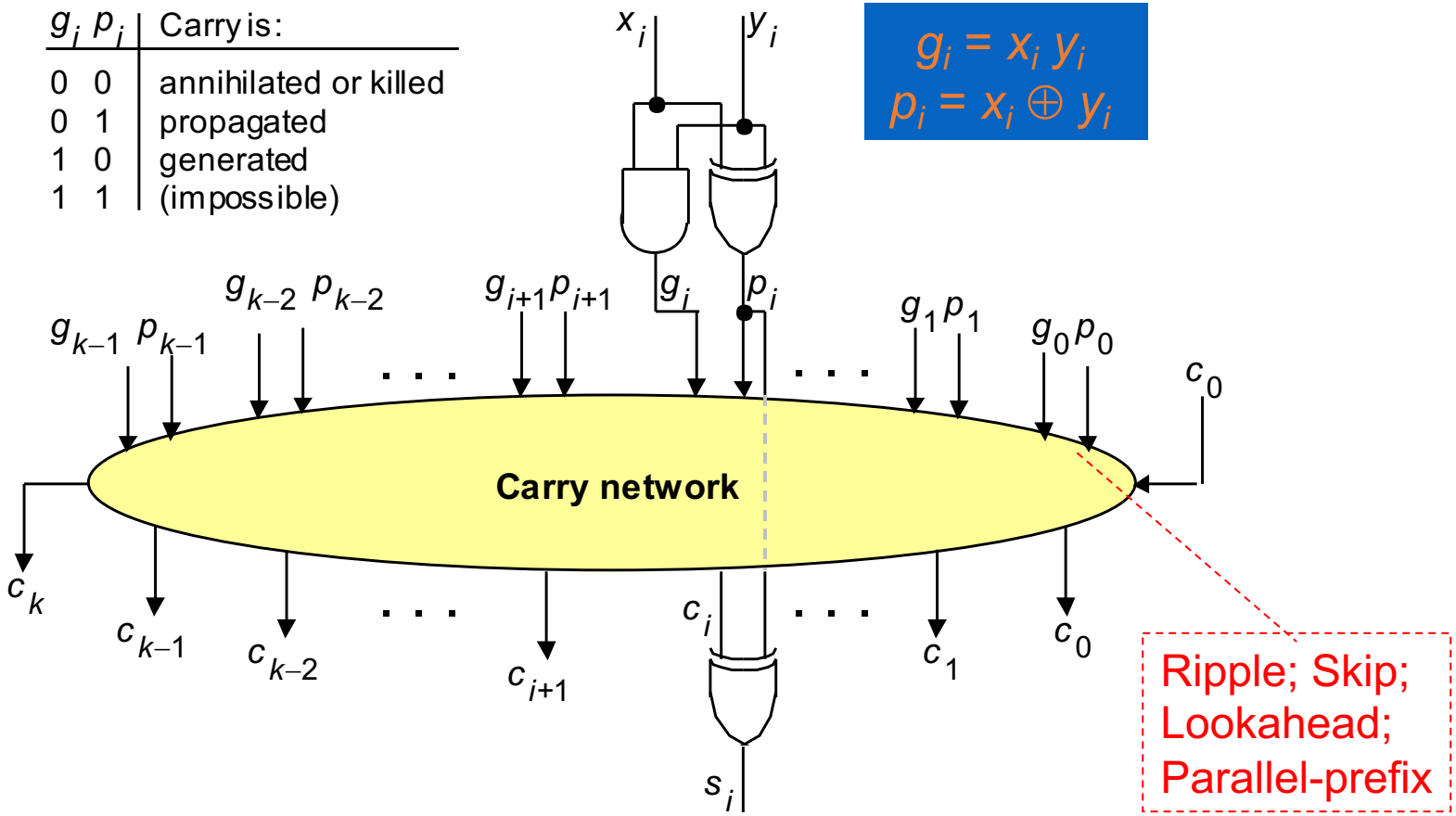
Fast Addition

Carry network is the essence of fast adders

Ripple-Carry Adder Revisited

g_i	p_i	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)

$g_i = x_i\, y_i$
$p_i = x_i \oplus y_i$



Ripple; Skip;
Lookahead;
Parallel-prefix

Generic structure of a binary adder, highlighting its carry network.

Fast Addition

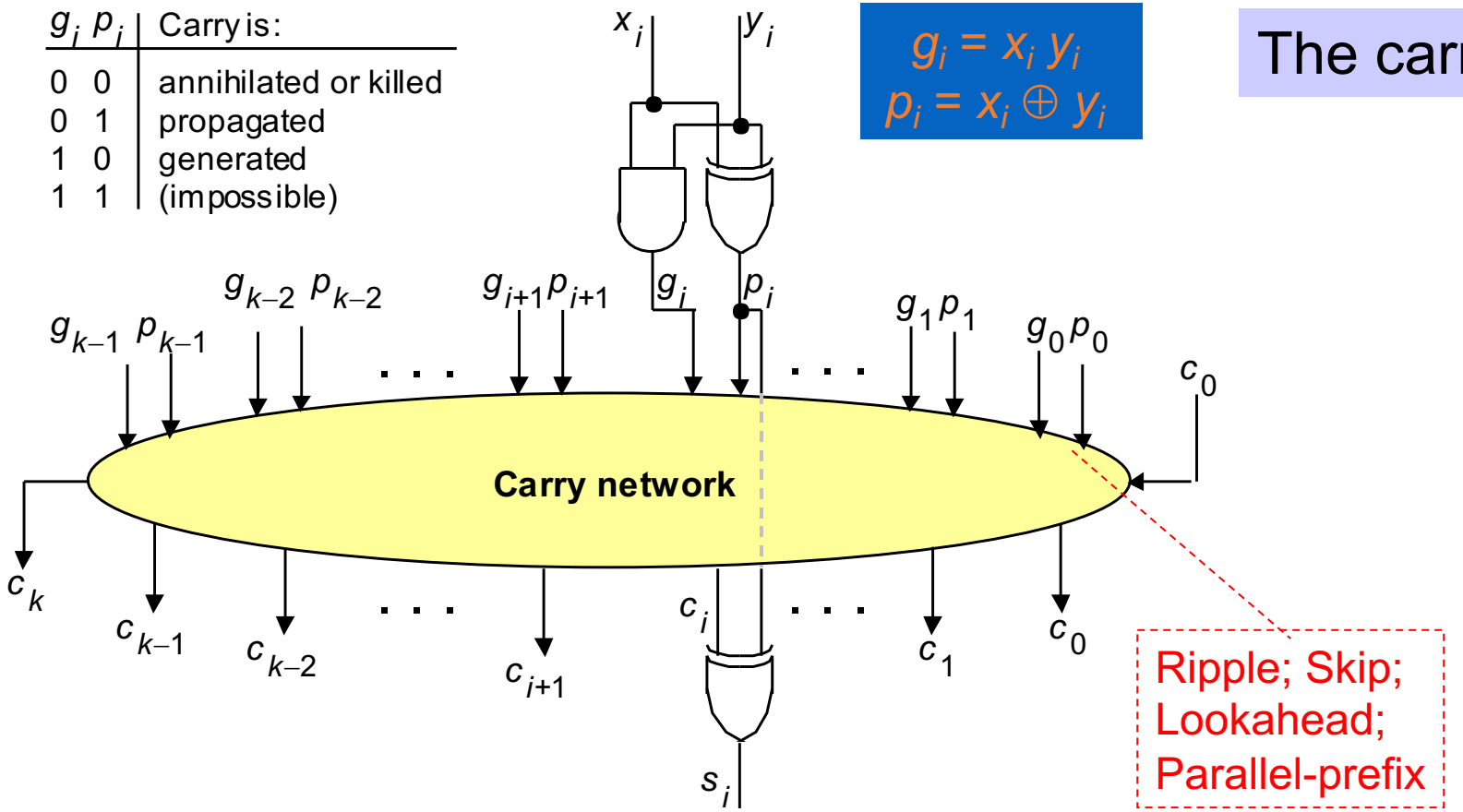
Carry network is the essence of fast adders

Ripple-Carry Adder Revisited

The carry recurrence: $c_{i+1} = g_i \vee p_i\, c_i$

g_i	p_i	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)

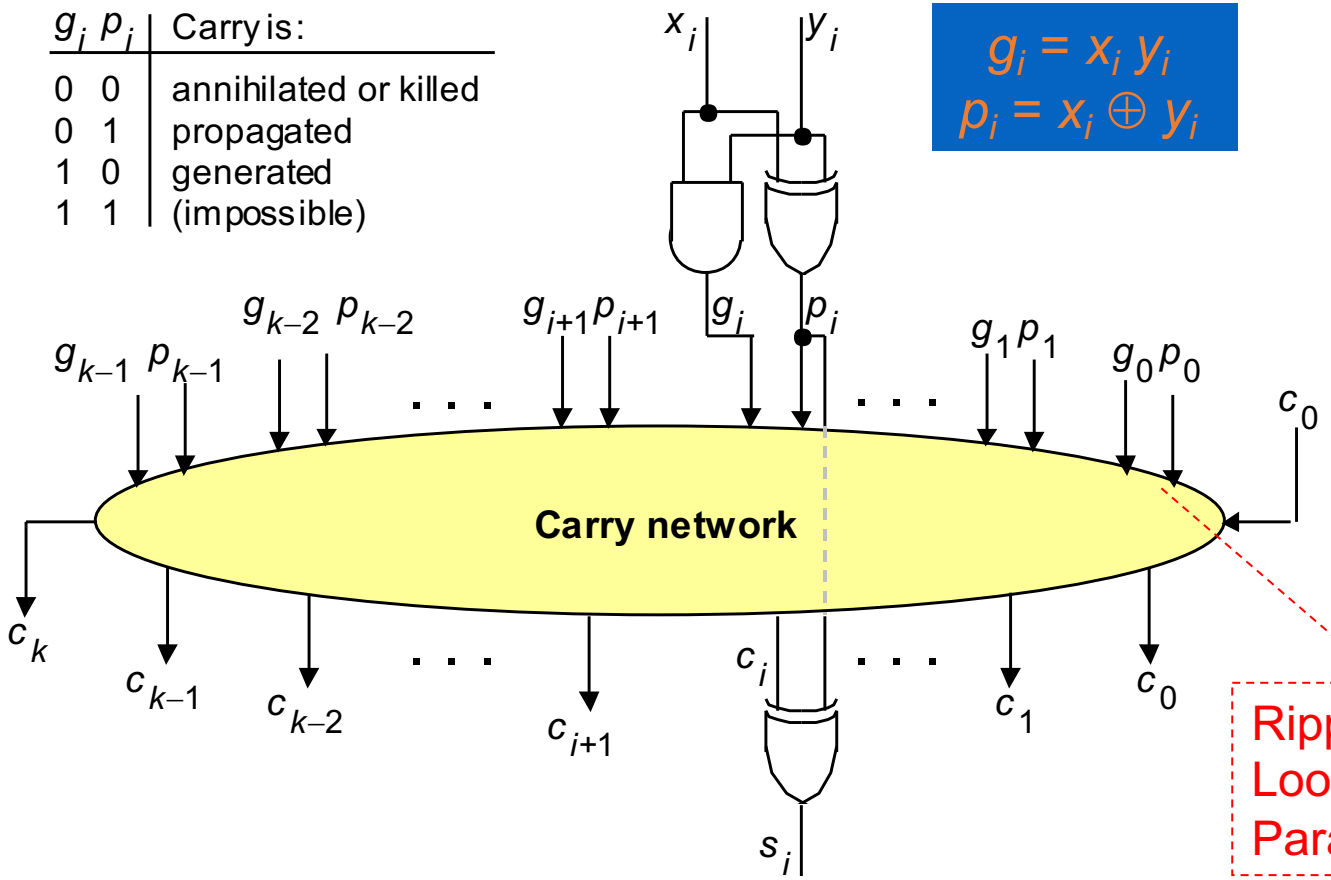
$g_i = x_i\, y_i$
$p_i = x_i \oplus y_i$



Generic structure of a binary adder, highlighting its carry network.

Fast Addition

Carry network is the essence of fast adders



Ripple-Carry Adder Revisited

The carry recurrence: $c_{i+1} = g_i \vee p_i\, c_i$

Latency of k-bit adder is roughly $2k$ gate delays:

- 1 gate delay for production of p and g signals, plus
- $2(k-1)$ gate delays for carry propagation, plus
- 1 XOR gate delay for generation of the sum bits

Ripple; Skip; Lookahead; Parallel-prefix

Generic structure of a binary adder, highlighting its carry network.

Carry-Lookahead Adders

Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

Carry-Lookahead Adders

Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

Signal	Radix r	Binary
g_i	is 1 iff $x_i + y_i \geq r$	$x_i\, y_i$
p_i	is 1 iff $x_i + y_i = r - 1$	$x_i \oplus y_i$
a_i	is 1 iff $x_i + y_i < r - 1$	$x_i' y_i' = (x_i \vee y_i)'$
t_i	is 1 iff $x_i + y_i \geq r - 1$	$x_i \vee y_i$
s_i	$(x_i + y_i + c_i) \bmod r$	$x_i \oplus y_i \oplus c_i$

Carry-Lookahead Adders

Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

Signal	Radix r	Binary
g_i	is 1 iff $x_i + y_i \geq r$	$x_i\, y_i$
p_i	is 1 iff $x_i + y_i = r - 1$	$x_i \oplus y_i$
a_i	is 1 iff $x_i + y_i < r - 1$	$x_i' y_i' = (x_i \vee y_i)'$
t_i	is 1 iff $x_i + y_i \geq r - 1$	$x_i \vee y_i$
s_i	$(x_i + y_i + c_i) \bmod r$	$x_i \oplus y_i \oplus c_i$

The carry recurrence can be unrolled to obtain each carry signal directly from inputs, rather than through propagation

Carry-Lookahead Adders

Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

Signal	Radix r	Binary
g_i	is 1 iff $x_i + y_i \geq r$	$x_i\, y_i$
p_i	is 1 iff $x_i + y_i = r - 1$	$x_i \oplus y_i$
a_i	is 1 iff $x_i + y_i < r - 1$	$x_i' y_i' = (x_i \vee y_i)'$
t_i	is 1 iff $x_i + y_i \geq r - 1$	$x_i \vee y_i$
s_i	$(x_i + y_i + c_i) \bmod r$	$x_i \oplus y_i \oplus c_i$

The carry recurrence can be unrolled to obtain each carry signal directly from inputs, rather than through propagation

$$
\begin{aligned}
c_i &= g_{i-1} \vee c_{i-1} p_{i-1} \\
&= g_{i-1} \vee (g_{i-2} \vee c_{i-2} p_{i-2}) p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee c_{i-2} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee c_{i-3} p_{i-3} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee g_{i-4} p_{i-3} p_{i-2} p_{i-1} \vee c_{i-4} p_{i-4} p_{i-3} p_{i-2} p_{i-1} \\
&= \ldots
\end{aligned}
$$

Carry-Lookahead Adders

Recall the *generate*, *propagate*, *annihilate* (*absorb*), and *transfer* signals:

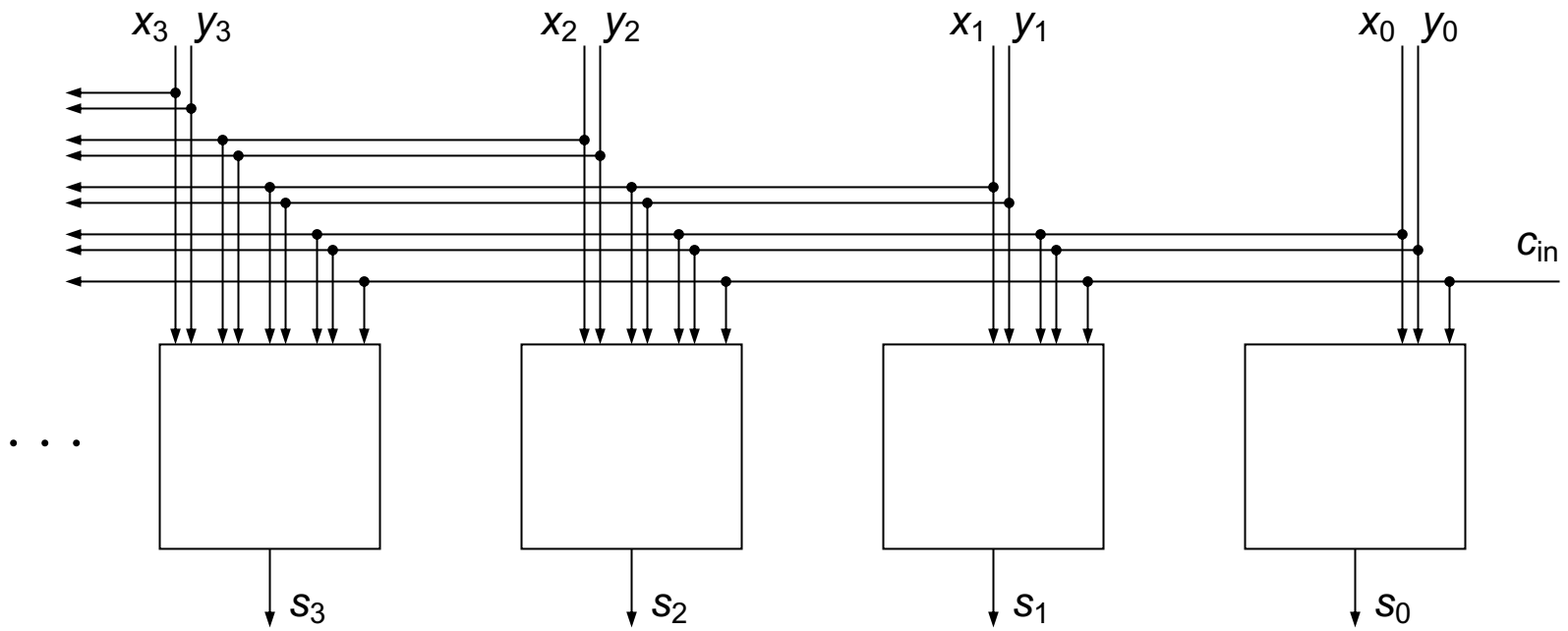
Signal	Radix r	Binary
g_i	is 1 iff $x_i + y_i \geq r$	$x_i\, y_i$
p_i	is 1 iff $x_i + y_i = r - 1$	$x_i \oplus y_i$
a_i	is 1 iff $x_i + y_i < r - 1$	$x_i' y_i' = (x_i \vee y_i)'$
t_i	is 1 iff $x_i + y_i \geq r - 1$	$x_i \vee y_i$
s_i	$(x_i + y_i + c_i) \bmod r$	$x_i \oplus y_i \oplus c_i$

The carry recurrence can be unrolled to obtain each carry signal directly from inputs, rather than through propagation

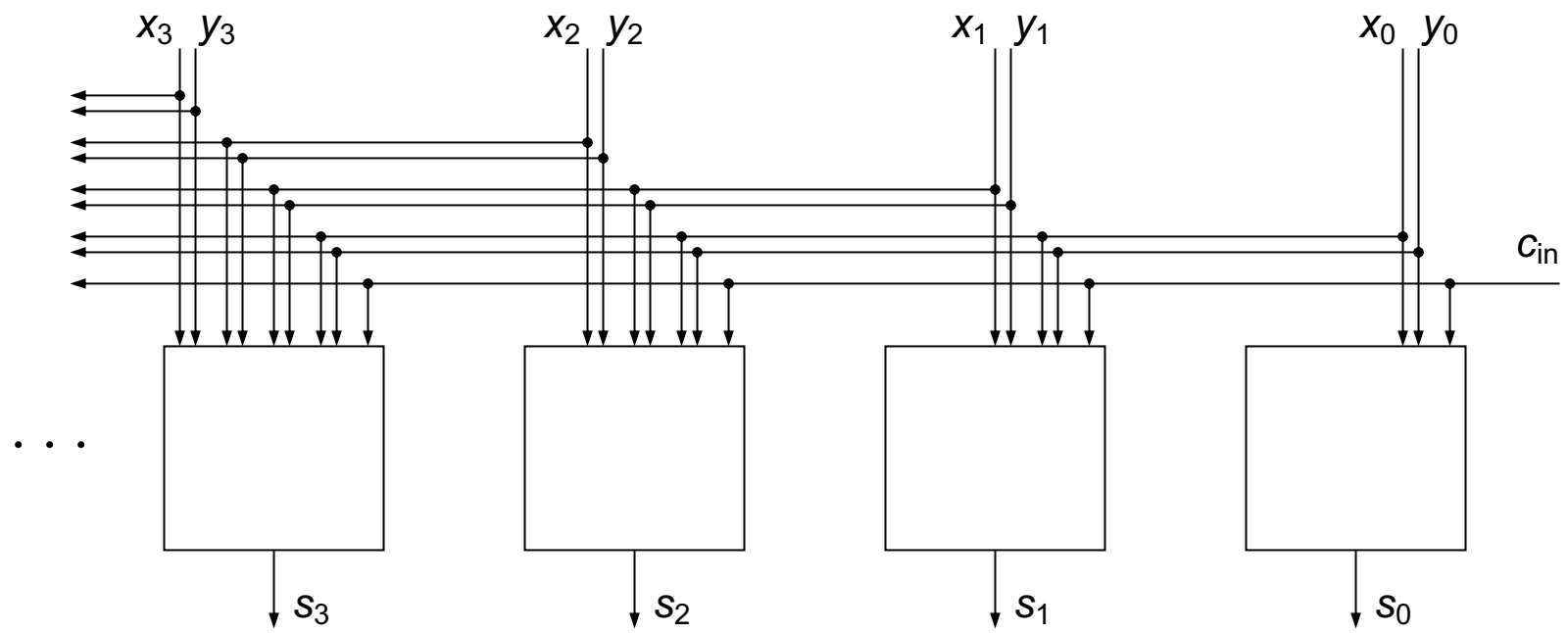
$$
\begin{aligned}
c_i &= g_{i-1} \vee c_{i-1} p_{i-1} \\
&= g_{i-1} \vee (g_{i-2} \vee c_{i-2} p_{i-2}) p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee c_{i-2} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee c_{i-3} p_{i-3} p_{i-2} p_{i-1} \\
&= g_{i-1} \vee g_{i-2} p_{i-1} \vee g_{i-3} p_{i-2} p_{i-1} \vee g_{i-4} p_{i-3} p_{i-2} p_{i-1} \vee c_{i-4} p_{i-4} p_{i-3} p_{i-2} p_{i-1} \\
&= \ldots
\end{aligned}
$$

Note:
Addition symbol vs logical OR

Carry-lookahead adders



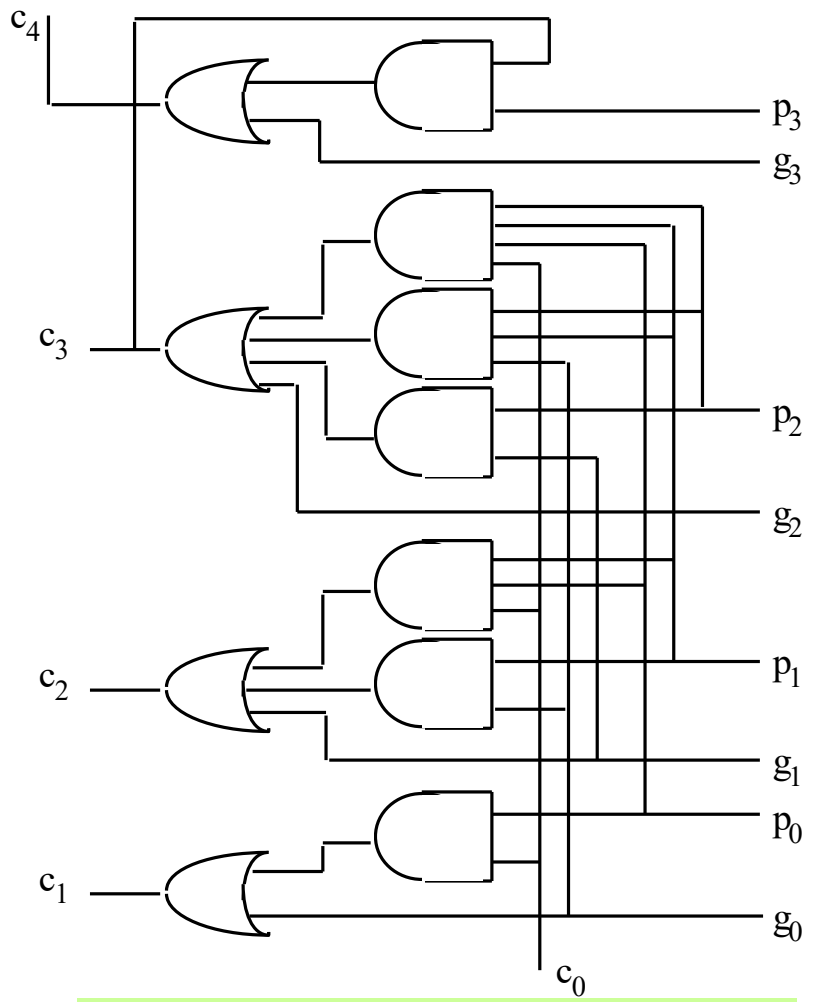
Carry-lookahead adders



Theoretically, it is possible to derive each sum digit directly from the inputs that affect it

Carry-lookahead adder design is simply a way of reducing the complexity of this ideal, but impractical, arrangement by hardware sharing among the various lookahead circuits

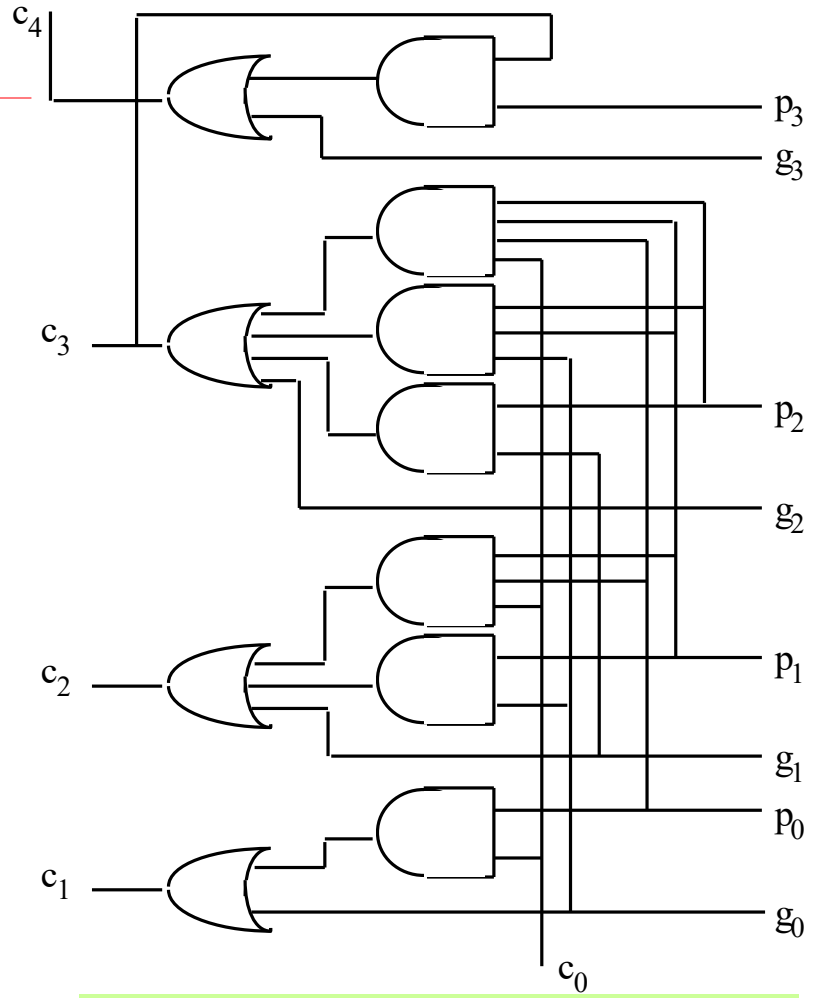
Carry-lookahead adders: 4-bit example



Four-bit carry network with full lookahead

Carry-lookahead adders: 4-bit example

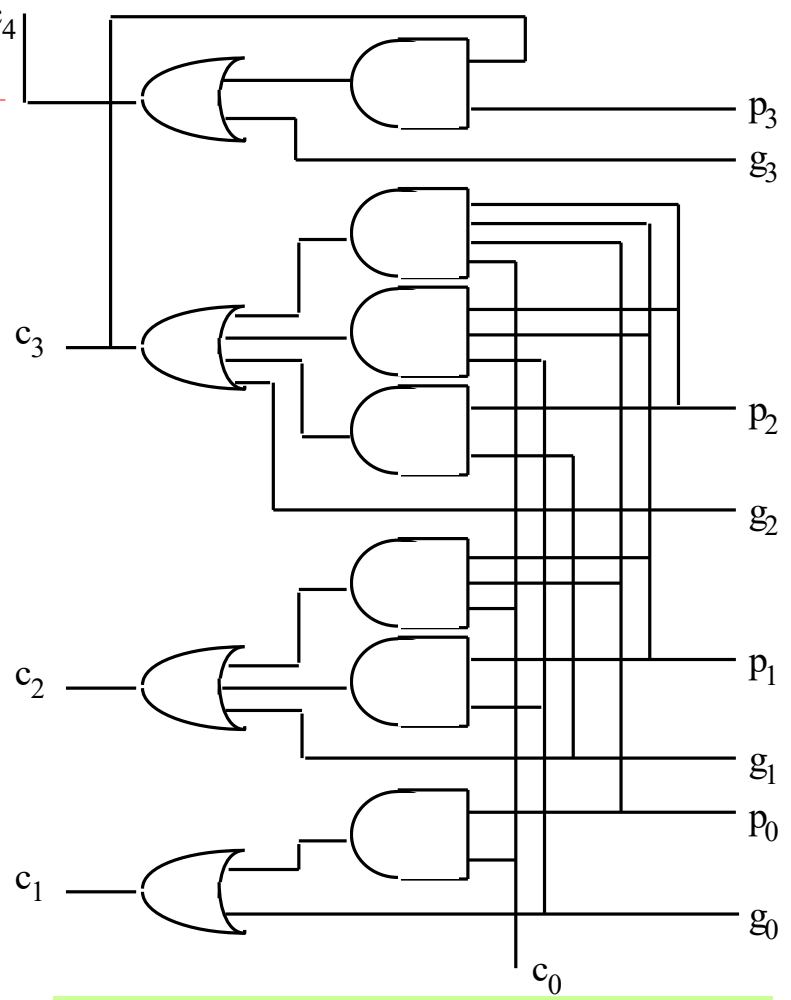
Complexity reduced by deriving the carry-out indirectly



Four-bit carry network with full lookahead

Carry-lookahead adders: 4-bit example

Complexity reduced by deriving the carry-out indirectly



Four-bit carry network with full lookahead

Full carry lookahead is quite practical for a 4-bit adder

$$c_1 = g_0 \vee c_0 p_0$$
$$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$$
$$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$$
$$c_4 = g_3 \vee g_2 p_3 \vee g_1 p_2 p_3 \vee g_0 p_1 p_2 p_3 \vee c_0 p_0 p_1 p_2 p_3$$

Carry-lookahead adders: beyond 4-bits

Consider a 32-bit adder

$c_1 = g_0 \vee c_0 p_0$

$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$

$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$

.

.

.

$c_{31} = g_{30} \vee g_{29} p_{30} \vee g_{28} p_{29} p_{30} \vee g_{27} p_{28} p_{29} p_{30} \vee \ . \ . \ . \ \vee c_0 p_0 p_1 p_2$

$p_3 \ldots p_{29} p_{30}$

By using tree of smaller (e.g., 4-bit) carry-lookahead adders speed is traded-off for area and power

Carry-lookahead adders: beyond 4-bits

Consider a 32-bit adder

$c_1 = g_0 \vee c_0 p_0$

$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$

$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$

.

.

.

32-input AND

$c_{31} = g_{30} \vee g_{29} p_{30} \vee g_{28} p_{29} p_{30} \vee g_{27} p_{28} p_{29} p_{30} \vee \ldots \vee c_0 p_0 p_1 p_2$

$p_3 \ldots p_{29} p_{30}$

By using tree of smaller (e.g., 4-bit) carry-lookahead adders speed is traded-off for area and power

Carry-lookahead adders: beyond 4-bits

Consider a 32-bit adder

$c_1 = g_0 \vee c_0 p_0$

$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$

$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$

.

.

.

$c_{31} = g_{30} \vee g_{29} p_{30} \vee g_{28} p_{29} p_{30} \vee g_{27} p_{28} p_{29} p_{30} \vee \ldots \vee c_0 p_0 p_1 p_2$

$p_3 \ldots p_{29} p_{30}$

32-input AND

32-input OR

By using tree of smaller (e.g., 4-bit) carry-lookahead adders speed is traded-off for area and power

Carry-lookahead adders: beyond 4-bits

Consider a 32-bit adder

$c_1 = g_0 \vee c_0 p_0$

$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$

$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$

. . .

$c_{31} = g_{30} \vee g_{29} p_{30} \vee g_{28} p_{29} p_{30} \vee g_{27} p_{28} p_{29} p_{30} \vee \ldots \vee c_0 p_0 p_1 p_2$

$p_3 \ldots p_{29} p_{30}$

32-input AND

32-input OR

High fan-ins necessitate tree-structured circuits

By using tree of smaller (e.g., 4-bit) carry-lookahead adders speed is traded-off for area and power

Carry-lookahead adders: beyond 4-bits

Consider a 32-bit adder

$c_1 = g_0 \vee c_0 p_0$

$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_0 p_1$

$c_3 = g_2 \vee g_1 p_2 \vee g_0 p_1 p_2 \vee c_0 p_0 p_1 p_2$

. . .

$c_{31} = g_{30} \vee g_{29} p_{30} \vee g_{28} p_{29} p_{30} \vee g_{27} p_{28} p_{29} p_{30} \vee \ldots \vee c_0 p_0 p_1 p_2$

$p_3 \ldots p_{29} p_{30}$

No circuit sharing: Repeated computations

32-input AND

32-input OR

High fan-ins necessitate tree-structured circuits

By using tree of smaller (e.g., 4-bit) carry-lookahead adders speed is traded-off for area and power

Basic multiplication operation

Shift/Add Multiplication Algorithms

Basic multiplication operation

Shift/Add Multiplication Algorithms

Notation for our discussion of multiplication algorithms:

a	Multiplicand	$a_{k-1}a_{k-2} \ldots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2} \ldots x_1x_0$
p	Product ($a \times x$)	$p_{2k-1}p_{2k-2} \ldots p_3p_2p_1p_0$

Initially, we assume unsigned operands

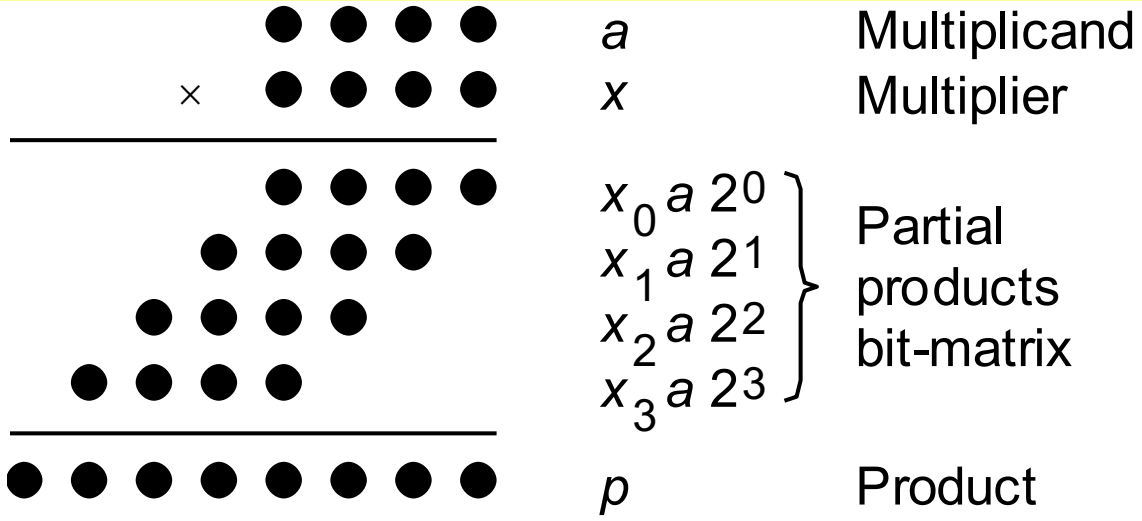
Basic multiplication operation

Shift/Add Multiplication Algorithms

Notation for our discussion of multiplication algorithms:

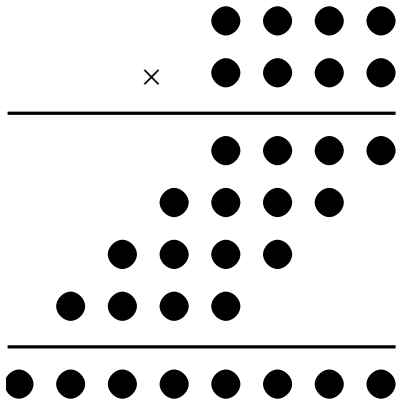
a	Multiplicand	$a_{k-1}a_{k-2} \ldots a_1a_0$
x	Multiplier	$x_{k-1}x_{k-2} \ldots x_1x_0$
p	Product ($a \times x$)	$p_{2k-1}p_{2k-2} \ldots p_3p_2p_1p_0$

Initially, we assume unsigned operands



Multiplication of two 4-bit unsigned binary numbers in dot notation

Multiplication recurrence



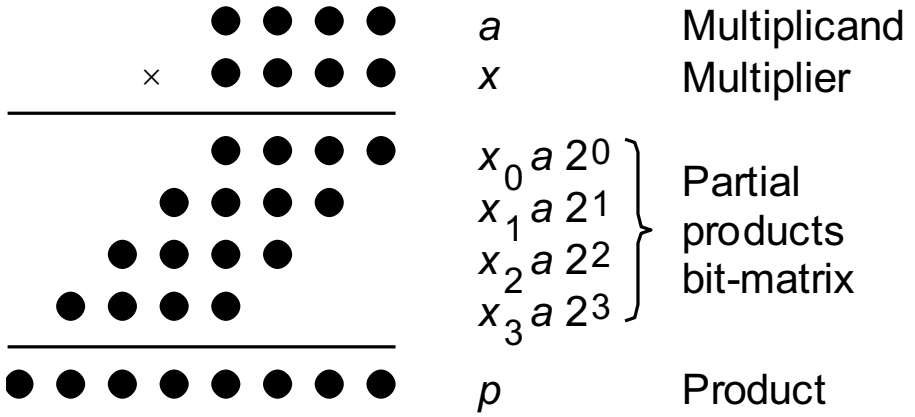
a Multiplicand

x Multiplier

$x_0\, a\, 2^0$
$x_1\, a\, 2^1$
$x_2\, a\, 2^2$
$x_3\, a\, 2^3$
} Partial products bit-matrix

p Product

Multiplication recurrence



Multiplication with left shifts: bottom-to-top accumulation

$$p^{(j+1)} = 2\,p^{(j)} + x_{k-j-1}a$$

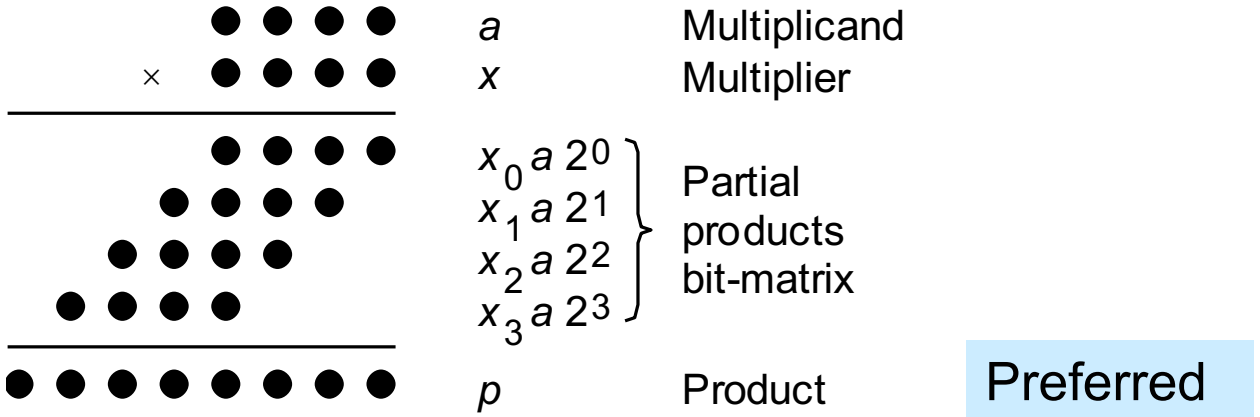
|shift|

|——add——|

with $p^{(0)} = 0$ and

$p^{(k)} = p = ax + p^{(0)}2^k$

Multiplication recurrence



Multiplication with left shifts: bottom-to-top accumulation

$$p^{(j+1)} = 2\,p^{(j)} + x_{k-j-1}a$$

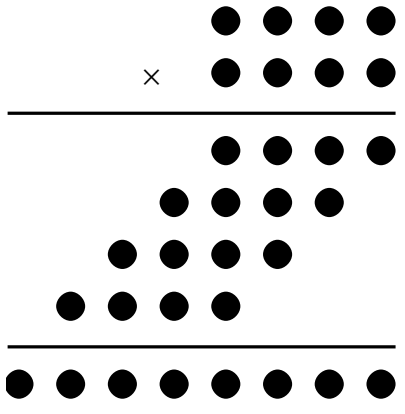
|shift|

|——add——|

with $p^{(0)} = 0$ and

$p^{(k)} = p = ax + p^{(0)}2^k$

Multiplication recurrence



a Multiplicand
x Multiplier
$x_0\, a\, 2^0$
$x_1\, a\, 2^1$
$x_2\, a\, 2^2$
$x_3\, a\, 2^3$
Partial products bit-matrix
p Product

Preferred

Multiplication with right shifts: top-to-bottom accumulation

$$p^{(j+1)} = (p^{(j)} + x_j\, a\, 2^k)\, 2^{-1}$$

|——add——|
|—shift right—|

with $p^{(0)} = 0$ and
$p^{(k)} = p = ax + p^{(0)}2^{-k}$

Multiplication with left shifts: bottom-to-top accumulation

$$p^{(j+1)} = 2\,p^{(j)} + x_{k-j-1}a$$

|shift|
|——add——|

with $p^{(0)} = 0$ and
$p^{(k)} = p = ax + p^{(0)}2^{k}$

Basic multiplication: example

Right-shift algorithm

a				1	0	1	0	
x				1	0	1	1	
$p^{(0)}$		0	0	0	0			
$+x_0a$		1	0	1	0			
$2p^{(1)}$	0	1	0	1	0			
$p^{(1)}$		0	1	0	1	0		
$+x_1a$		1	0	1	0			
$2p^{(2)}$	0	1	1	1	1	0		
$p^{(2)}$		0	1	1	1	1	0	
$+x_2a$		0	0	0	0			
$2p^{(3)}$	0	0	1	1	1	1	0	
$p^{(3)}$		0	0	1	1	1	1	0
$+x_3a$		1	0	1	0			
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

Left-shift algorithm

a				1	0	1	0	
x				1	0	1	1	
$p^{(0)}$				0	0	0	0	
$2p^{(0)}$			0	0	0	0	0	
$+x_3a$				1	0	1	0	
$p^{(1)}$			0	1	0	1	0	
$2p^{(1)}$		0	1	0	1	0	0	
$+x_2a$				0	0	0	0	
$p^{(2)}$			0	1	0	1	0	0
$2p^{(2)}$		0	1	0	1	0	0	0
$+x_1a$				1	0	1	0	
$p^{(3)}$		0	1	1	0	0	1	0
$2p^{(3)}$	0	1	1	0	0	1	0	0
$+x_0a$				1	0	1	0	
$p^{(4)}$	0	1	1	0	1	1	1	0

Examples of sequential multiplication with right and left shifts.

Basic multiplication: example

Right-shift algorithm

a				1	0	1	0	
x				1	0	1	1	
$p^{(0)}$		0	0	0	0			
$+x_0 a$		1	0	1	0			
$2p^{(1)}$	0	1	0	1	0			
$p^{(1)}$		0	1	0	1	0		
$+x_1 a$		1	0	1	0			
$2p^{(2)}$	0	1	1	1	1	0		
$p^{(2)}$		0	1	1	1	1	0	
$+x_2 a$		0	0	0	0			
$2p^{(3)}$	0	0	1	1	1	1	0	
$p^{(3)}$		0	0	1	1	1	1	0
$+x_3 a$		1	0	1	0			
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

Left-shift algorithm

a				1	0	1	0	
x				1	0	1	1	
$p^{(0)}$				0	0	0	0	
$2p^{(0)}$			0	0	0	0	0	
$+x_3 a$				1	0	1	0	
$p^{(1)}$			0	1	0	1	0	
$2p^{(1)}$		0	1	0	1	0	0	
$+x_2 a$				0	0	0	0	
$p^{(2)}$		0	1	0	1	0	0	
$p^{(4)}$	0	1	1	0	1	1	1	0

$$p^{(j+1)} = (p^{(j)} + x_j\, a\, 2^k)\, 2^{-1}$$

|——add——|

|—shift right—|

Examples of sequential multiplication with right and left shifts.

Basic multiplication: example

Right-shift algorithm

a					1	0	1	0
x	1	0	1	0	1	0	1	1
$p^{(0)}$				0	0	0	0	
$+x_0 a$				1	0	1	0	
$2p^{(1)}$			0	1	0	1	0	
$p^{(1)}$				0	1	0	1	0
$+x_1 a$				1	0	1	0	
$2p^{(2)}$		0	1	1	1	1	0	
$p^{(2)}$			0	1	1	1	1	0
$+x_2 a$				0	0	0	0	
$2p^{(3)}$	0	0	1	1	1	1	0	
$p^{(3)}$		0	0	1	1	1	1	0
$+x_3 a$				1	0	1	0	
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

Left-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$					0	0	0	0
$2p^{(0)}$				0	0	0	0	0
$+x_3 a$					1	0	1	0
$p^{(1)}$				0	1	0	1	0
$2p^{(1)}$			0	1	0	1	0	0
$+x_2 a$					0	0	0	0
$p^{(2)}$			0	1	0	1	0	0
$p^{(4)}$		0	1	1	0	1	1	1	0

$$p^{(j+1)} = (p^{(j)} + x_j\, a\, 2^k)\, 2^{-1}$$

add

shift right

Examples of sequential multiplication with right and left shifts.

Basic multiplication: example

Right-shift algorithm

a					1	0	1	0
x	1	0	1	0	1	0	1	1
$p^{(0)}$				0	0	0	0	
$+x_0 a$				1	0	1	0	
$2p^{(1)}$			0	1	0	1	0	
$p^{(1)}$				0	1	0	1	0
$+x_1 a$				1	0	1	0	
$2p^{(2)}$		0	1	1	1	1	0	
$p^{(2)}$			0	1	1	1	1	0
$+x_2 a$				0	0	0	0	
$2p^{(3)}$	0	0	1	1	1	1	0	
$p^{(3)}$		0	0	1	1	1	1	0
$+x_3 a$				1	0	1	0	
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

Left-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$					0	0	0	0
$2p^{(0)}$				0	0	0	0	0
$+x_3 a$					1	0	1	0
$p^{(1)}$				0	1	0	1	0
$2p^{(1)}$			0	1	0	1	0	0
$+x_2 a$					0	0	0	0
$p^{(2)}$			0	1	0	1	0	0
$p^{(4)}$		0	1	1	0	1	1	1	0

$$p^{(j+1)} = (p^{(j)} + x_j\, a\, 2^k)\, 2^{-1}$$

|——add——|

|—shift right—|

Examples of sequential multiplication with right and left shifts.

Check:
10×11
$= 110$
$= 64 + 32 + 8 + 4 + 2$

Basic multiplication: example

Right-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$		0	0	0	0			
$+x_0a$		1	0	1	0			
$2p^{(1)}$	0	1	0	1	0			
$p^{(1)}$		0	1	0	1	0		
$+x_1a$		1	0	1	0			
$2p^{(2)}$	0	1	1	1	1	0		
$p^{(2)}$		0	1	1	1	1	0	
$+x_2a$		0	0	0	0			
$2p^{(3)}$	0	0	1	1	1	1	0	
$p^{(3)}$		0	0	1	1	1	1	0
$+x_3a$		1	0	1	0			
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

Left-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$					0	0	0	0
$2p^{(0)}$			0	0	0	0	0	
$+x_3a$					1	0	1	0
$p^{(1)}$			0	1	0	1	0	
$2p^{(1)}$		0	1	0	1	0	0	
$+x_2a$					0	0	0	0
$p^{(2)}$			0	1	0	1	0	0
$2p^{(2)}$		0	1	0	1	0	0	
$+x_1a$					1	0	1	0
$p^{(3)}$		0	1	1	0	0	1	0
$2p^{(3)}$	0	1	1	0	0	1	0	0
$+x_0a$					1	0	1	0
$p^{(4)}$	0	1	1	0	1	1	1	0

Examples of sequential multiplication with right and left shifts.

Basic multiplication: example

Right-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$		0	0	0	0			
$+x_0 a$		1	0	1	0			
$2p^{(1)}$	0	1	0	1	0			
$p^{(1)}$		0	1	0	1	0		
$+x_1 a$		1	0	1	0			
$2p^{(2)}$	0	1	1	1	1	0		
$p^{(2)}$		0	1	1	1	1	0	
$+x_2 a$		0	0	0	0			
$2p^{(3)}$	0	1	1	1	0	1	0	
$p^{(3)}$		0	1	1	1	0	1	0
$+x_3 a$		1	0	1	0			
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

$p^{(j+1)} = 2\,p^{(j)} + x_{k-j-1}\,a$

|shift|

|——add——|

Left-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$					0	0	0	0
$2p^{(0)}$			0	0	0	0	0	
$+x_3 a$					1	0	1	0
$p^{(1)}$			0	1	0	1	0	
$2p^{(1)}$		0	1	0	1	0	0	
$+x_2 a$					0	0	0	0
$p^{(2)}$		0	1	0	1	0	0	
$2p^{(2)}$	0	1	0	1	0	0	0	
$+x_1 a$					1	0	1	0
$p^{(3)}$	0	1	1	0	0	1	0	
$2p^{(3)}$	0	1	1	0	0	1	0	0
$+x_0 a$					1	0	1	0
$p^{(4)}$	0	1	1	0	1	1	1	0

Examples of sequential multiplication with right and left shifts.

Basic multiplication: example

Right-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$			0	0	0	0		
$+x_0 a$			1	0	1	0		
$2p^{(1)}$	0	1	0	1	0			
$p^{(1)}$		0	1	0	1	0		
$+x_1 a$		1	0	1	0			
$2p^{(2)}$	0	1	1	1	1	0		
$p^{(2)}$		0	1	1	1	1	0	
$+x_2 a$		0	0	0	0			
$2p^{(3)}$	0	1	1	1	0	1	0	
$p^{(3)}$		0	1	1	0	1	0	
$+x_3 a$		1	0	1	0			
$2p^{(4)}$	0	1	1	0	1	1	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

$$p^{(j+1)} = 2\,p^{(j)} + x_{k-j-1}\,a$$

|shift|

|——add——|

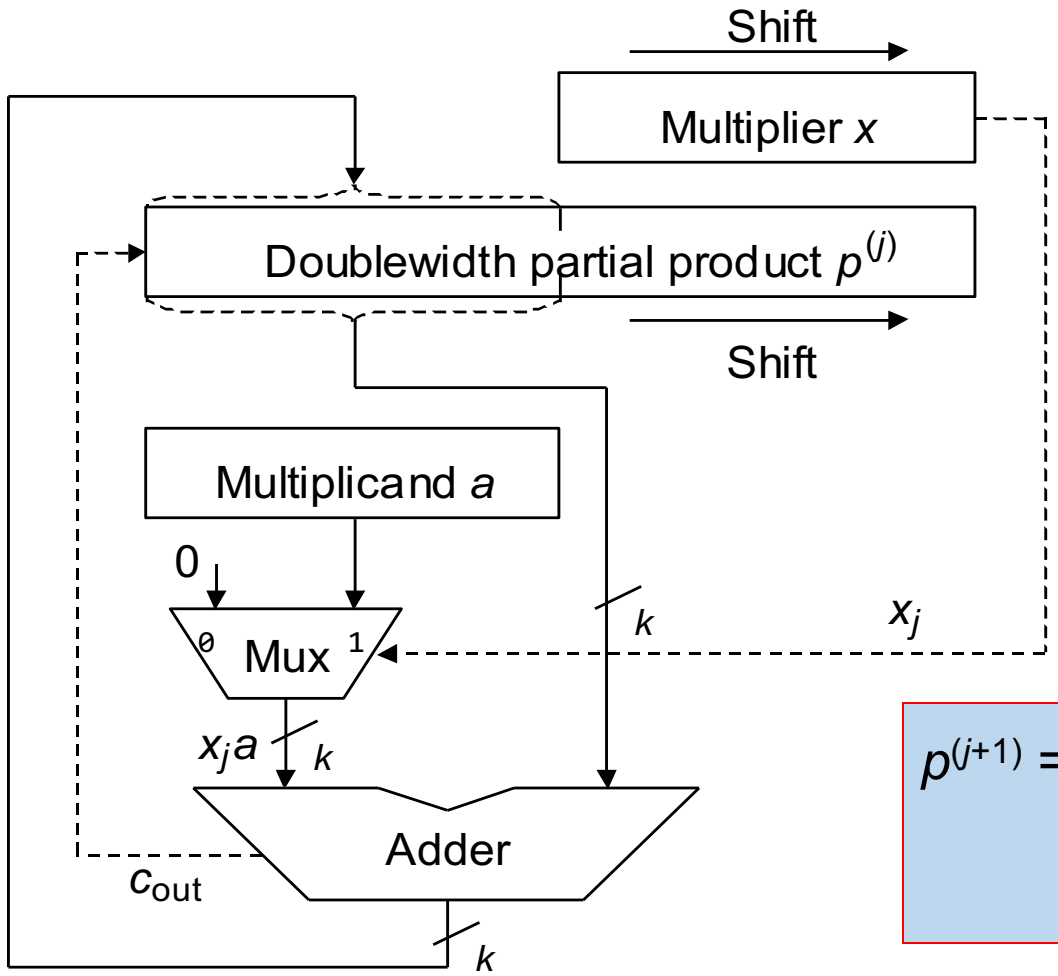
Left-shift algorithm

a					1	0	1	0
x					1	0	1	1
$p^{(0)}$					0	0	0	0
$2p^{(0)}$			0	0	0	0	0	
$+x_3 a$					1	0	1	0
$p^{(1)}$			0	1	0	1	0	
$2p^{(1)}$		0	1	0	1	0	0	
$+x_2 a$					0	0	0	0
$p^{(2)}$		0	1	0	1	0	0	
$2p^{(2)}$	0	1	0	1	0	0	0	
$+x_1 a$					1	0	1	0
$p^{(3)}$		0	1	1	0	0	1	0
$2p^{(3)}$	0	1	1	0	0	1	0	0
$+x_0 a$					1	0	1	0
$p^{(4)}$		0	1	1	0	1	1	1	0

Examples of sequential multiplication with right and left shifts.

Check:
10×11
$= 110$
$= 64 + 32 + 8 + 4 + 2$

Basic multiplier: hardware implementation



$$p^{(j+1)} = (p^{(j)} + x_j\, a\, 2^k)\, 2^{-1}$$
$$\text{add}$$
$$\text{shift right}$$

Hardware realization of the sequential multiplication algorithm with additions and right shifts

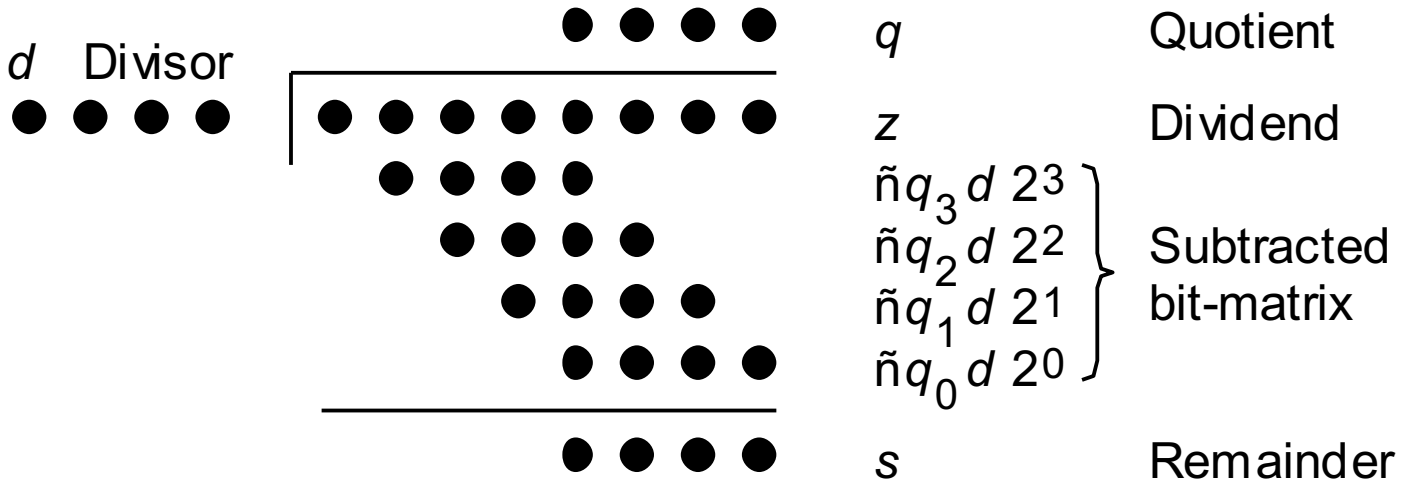
Basic division operation

Division algorithm based on the pen and paper approach

Notation for our discussion of division algorithms:

z	Dividend	$z_{2k-1}z_{2k-2} \ . \ . \ . \ z_3z_2z_1z_0$
d	Divisor	$d_{k-1}d_{k-2} \ldots d_1d_0$
q	Quotient	$q_{k-1}q_{k-2} \ldots q_1q_0$
s	Remainder, $z - (d \times q)$	$s_{k-1}s_{k-2} \ldots s_1s_0$

Initially, we assume unsigned operands

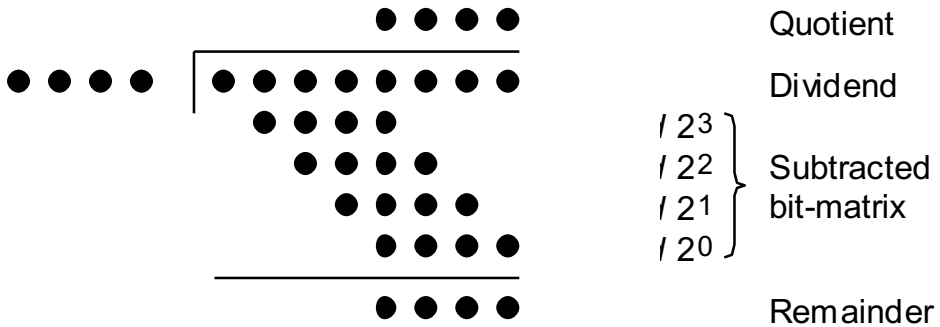


Division of an 8-bit number by a 4-bit number in dot notation

Division versus multiplication

Division is more complex than multiplication:
- Need for quotient digit selection or estimation

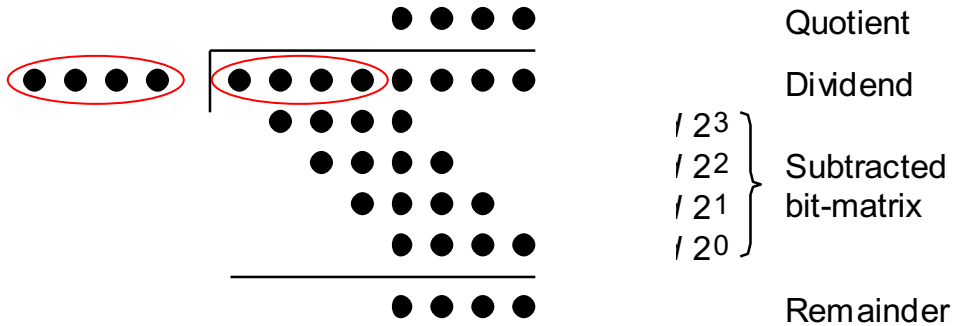
Overflow possibility: the high-order *k* bits of *z* must be strictly less than *d*; this overflow check also detects the divide-by-zero condition.



Division versus multiplication

Division is more complex than multiplication:
- Need for quotient digit selection or estimation

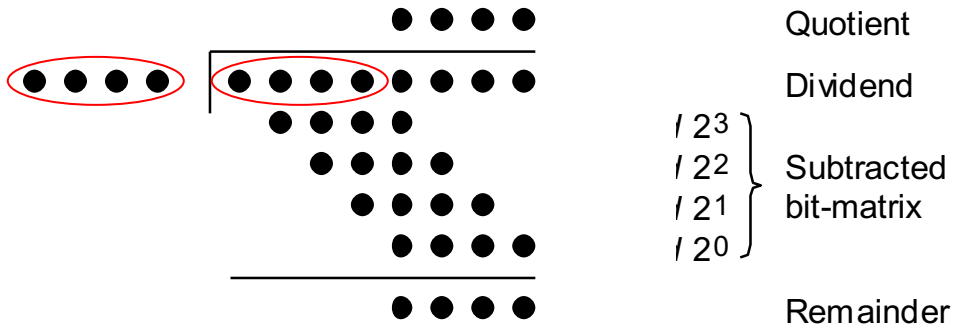
Overflow possibility: the high-order *k* bits of *z* must be strictly less than *d*; this overflow check also detects the divide-by-zero condition.



Division versus multiplication

Division is more complex than multiplication:
Need for quotient digit selection or estimation

Overflow possibility: the high-order *k* bits of *z* must be strictly less than *d*; this overflow check also detects the divide-by-zero condition.



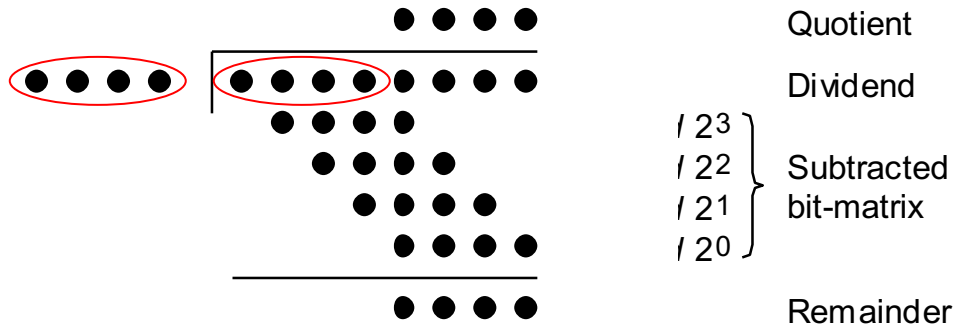
Pentium III latencies

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

Division versus multiplication

Division is more complex than multiplication:
- Need for quotient digit selection or estimation

Overflow possibility: the high-order *k* bits of *z*
- must be strictly less than *d*; this overflow check also detects the divide-by-zero condition.



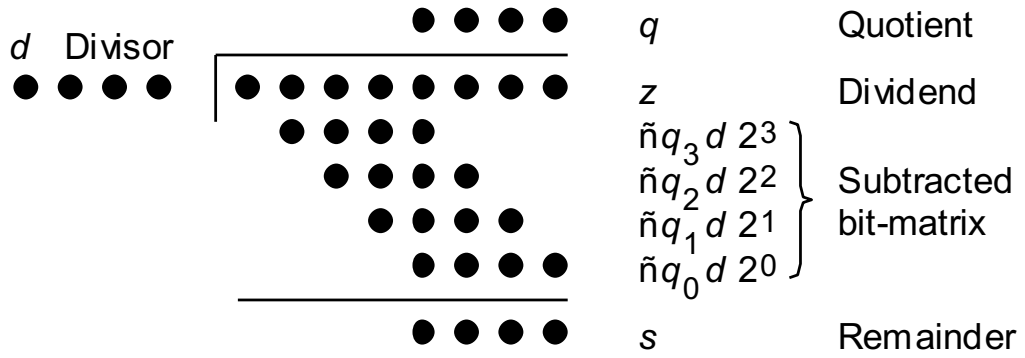
Pentium III latencies

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

The ratios haven't changed much in later Pentiums, Atom, or AMD products*

*Source: T. Granlund, "Instruction Latencies and Throughput for AMD and Intel x86 Processors," Feb. 2012

Division by recurrence: slow division



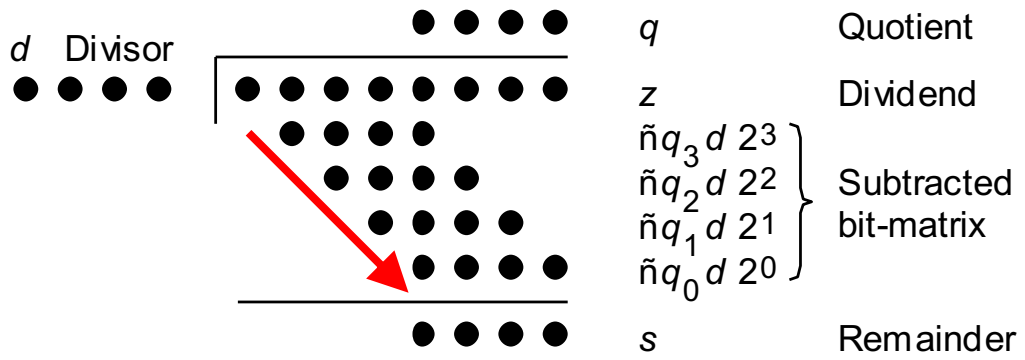
Division with left shifts

$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d)$$

|–shift–|
|——subtract——|

with $s^{(0)} = z$ and $s^{(k)} = 2^k s$

Division by recurrence: slow division



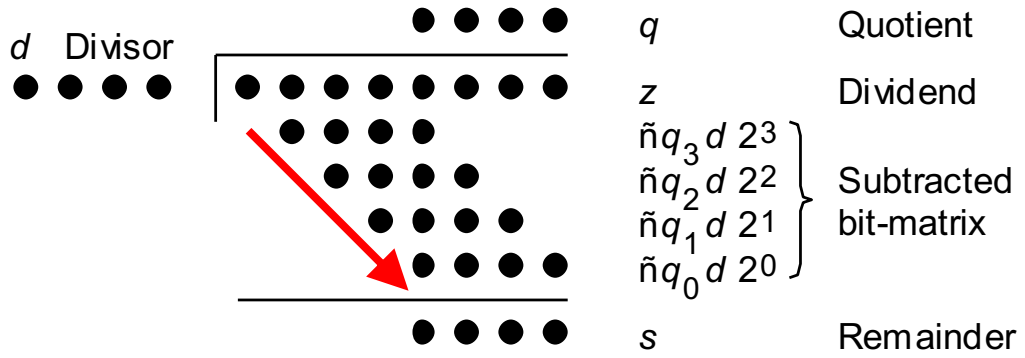
Division with left shifts

$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d)$$

|–shift–|
|——subtract——|

with $s^{(0)} = z$ and $s^{(k)} = 2^k s$

Division by recurrence: slow division



Division with left shifts (There is no corresponding right-shift algorithm)

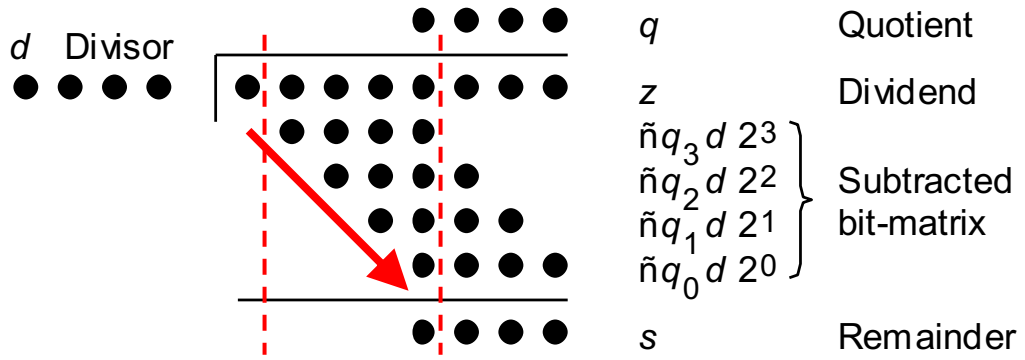
$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d)$$

|–shift–|

|——subtract——|

with $s^{(0)} = z$ and $s^{(k)} = 2^k s$

Division by recurrence: slow division



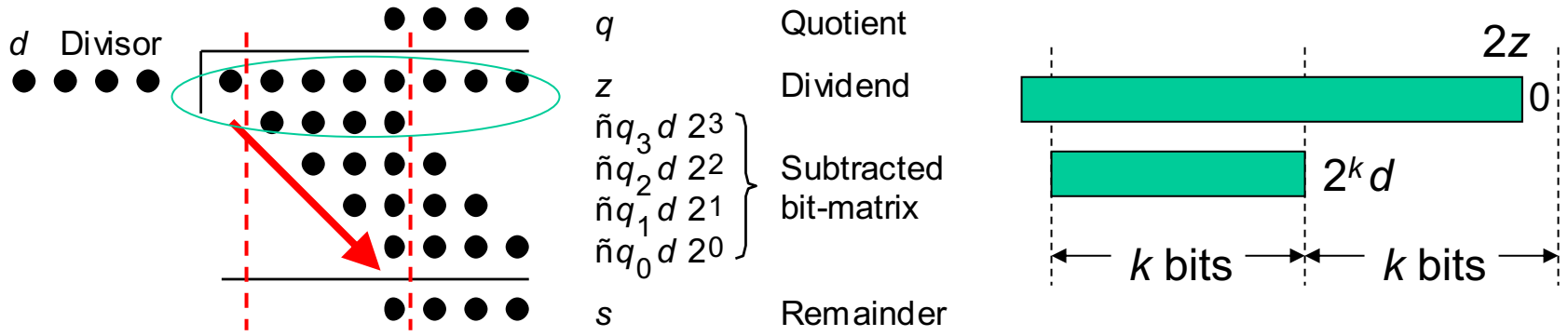
Division with left shifts (There is no corresponding right-shift algorithm)

$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d)$$

|–shift–|
|——subtract——|

with $s^{(0)} = z$ and $s^{(k)} = 2^k s$

Division by recurrence: slow division



Division with left shifts (There is no corresponding right-shift algorithm)

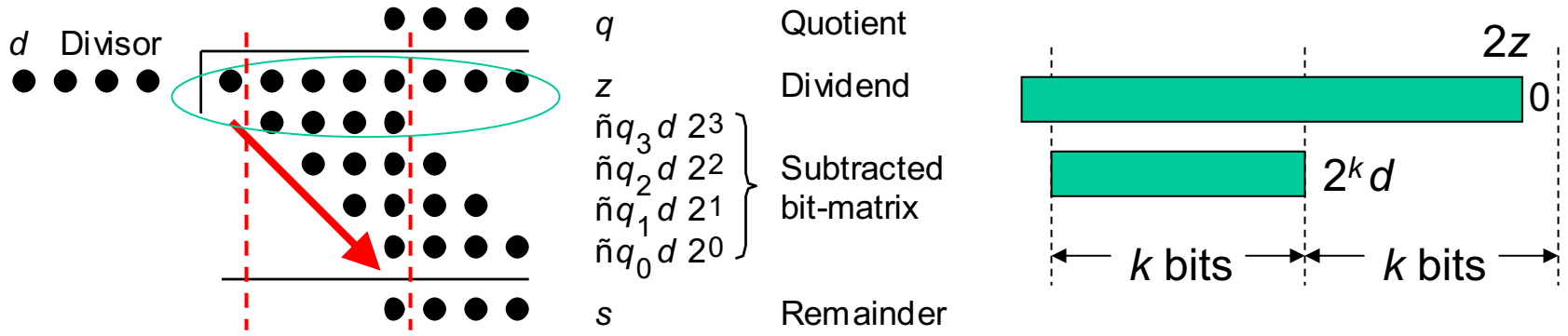
$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d)$$

|–shift–|

|——subtract——|

with $s^{(0)} = z$ and $s^{(k)} = 2^k s$

Division by recurrence: slow division



Division with left shifts (There is no corresponding right-shift algorithm)

$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d)$$

|–shift–|

|——subtract——|

with $s^{(0)} = z$ and $s^{(k)} = 2^k s$

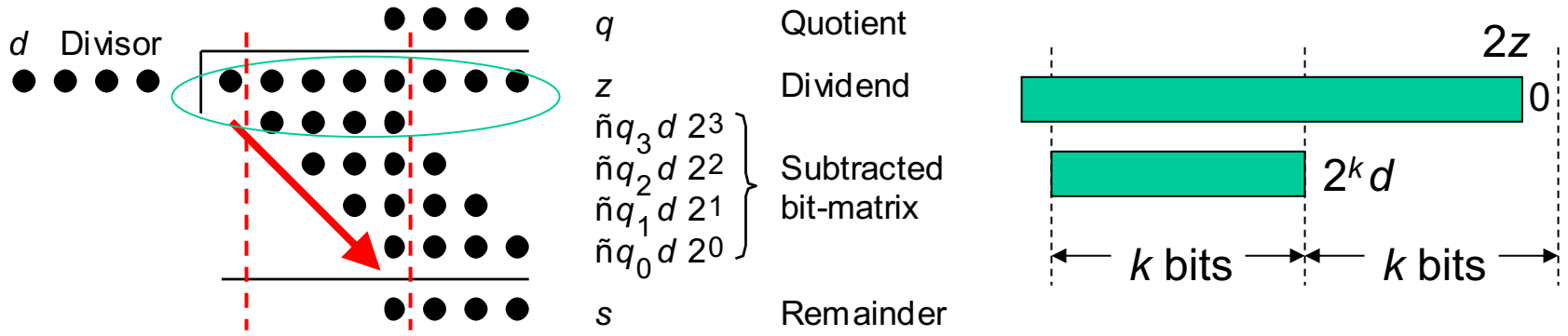
Integer division is characterized by $z = d \times q + s$

$$2^{-2k}z = (2^{-k}d) \times (2^{-k}q) + 2^{-2k}s$$

$$z_{frac} = d_{frac} \times q_{frac} + 2^{-k}s_{frac}$$

Divide fractions like integers; adjust the remainder

Division by recurrence: slow division



Division with left shifts (There is no corresponding right-shift algorithm)

$$s^{(j)} = 2s^{(j-1)} - q_{k-j}(2^k d)$$

|–shift–|

|——subtract——|

with $s^{(0)} = z$ and $s^{(k)} = 2^k s$

Integer division is characterized by $z = d \times q + s$

$$2^{-2k}z = (2^{-k}d) \times (2^{-k}q) + 2^{-2k}s$$

$$z_{frac} = d_{frac} \times q_{frac} + 2^{-k}s_{frac}$$

Divide fractions like integers; adjust the remainder

No-overflow condition for fractions is:

$z_{frac} < d_{frac}$

Example of basic division

Integer division

z		0	1	1	1	0	1	0 1
$2^4 d$		1	0	1	0			
$s^{(0)}$		0	1	1	1	0	1	0 1
$2s^{(0)}$	0	1	1	1	0	1	0	1
$-q_3 2^4 d$		1	0	1	0			$\{q_3 = 1\}$

Fractional division

z_{frac}		.	0	1	1	1	0	1 0 1
d_{frac}		.	1	0	1	0		
$s^{(0)}$		.	0	1	1	1	0	1 0 1
$2s^{(0)}$	0	.	1	1	1	0	1	0 1
$-q_{-1} d$		.	1	0	1	0		$\{q_{-1}=1\}$

Examples of sequential division with integer and fractional operands.

Example of basic division

Decimal

Integer division

z	117	0 1 1 1 0 1 0 1
$2^4 d$		1 0 1 0
$s^{(0)}$		0 1 1 1 0 1 0 1
$2s^{(0)}$		0 1 1 1 0 1 0 1
$-q_3 2^4 d$		1 0 1 0 {$q_3 = 1$}

Fractional division

z_{frac}	. 0 1 1 1 0 1 0 1
d_{frac}	. 1 0 1 0
$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_{-1} d$	. 1 0 1 0 {$q_{-1}=1$}

Examples of sequential division with integer and fractional operands.

Example of basic division

Decimal

Integer division

z	117	0 1 1 1 0 1 0 1
$2^4 d$	10	1 0 1 0
$s^{(0)}$		0 1 1 1 0 1 0 1
$2s^{(0)}$		0 1 1 1 0 1 0 1
$-q_3 2^4 d$		1 0 1 0 {$q_3 = 1$}

Fractional division

z_{frac}	. 0 1 1 1 0 1 0 1
d_{frac}	. 1 0 1 0
$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_{-1} d$	. 1 0 1 0 {$q_{-1}=1$}

Examples of sequential division with integer and fractional operands.

Example of basic division

Decimal

Integer division

z	117	0 1 1 1 0 1 0 1
$2^4 d$	10	1 0 1 0
$s^{(0)}$		0 1 1 1 0 1 0 1
$2s^{(0)}$		0 1 1 1 0 1 0 1
$-q_3 2^4 d$		1 0 1 0 $\{q_3 = 1\}$
$s^{(1)}$		0 1 0 0 1 0 1
$2s^{(1)}$		0 1 0 0 1 0 1
$-q_2 2^4 d$		0 0 0 0 $\{q_2 = 0\}$

Fractional division

z_{frac}	. 0 1 1 1 0 1 0 1
d_{frac}	. 1 0 1 0
$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_{-1} d$	. 1 0 1 0 $\{q_{-1}=1\}$
$s^{(1)}$	. 0 1 0 0 1 0 1
$2s^{(1)}$	0 . 1 0 0 1 0 1
$-q_{-2} d$	. 0 0 0 0 $\{q_{-2}=0\}$

Examples of sequential division with integer and fractional operands.

Example of basic division

Decimal

Integer division

z	117	0 1 1 1 0 1 0 1
$2^4 d$	10	1 0 1 0
$s^{(0)}$		0 1 1 1 0 1 0 1
$2s^{(0)}$		0 1 1 1 0 1 0 1
$-q_3 2^4 d$		1 0 1 0 $\{q_3 = 1\}$
$s^{(1)}$		0 1 0 0 1 0 1
$2s^{(1)}$		0 1 0 0 1 0 1
$-q_2 2^4 d$		0 0 0 0 $\{q_2 = 0\}$
$s^{(2)}$		1 0 0 1 0 1
$2s^{(2)}$		1 0 0 1 0 1
$-q_1 2^4 d$		1 0 1 0 $\{q_1 = 1\}$

Fractional division

z_{frac}	. 0 1 1 1 0 1 0 1
d_{frac}	. 1 0 1 0
$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_{-1} d$	. 1 0 1 0 $\{q_{-1}=1\}$
$s^{(1)}$	. 0 1 0 0 1 0 1
$2s^{(1)}$	0 . 1 0 0 1 0 1
$-q_{-2} d$	. 0 0 0 0 $\{q_{-2}=0\}$
$s^{(2)}$	. 1 0 0 1 0 1
$2s^{(2)}$	1 . 0 0 1 0 1
$-q_{-3} d$	. 1 0 1 0 $\{q_{-3}=1\}$

Examples of sequential division with integer and fractional operands.

Example of basic division

Decimal

Integer division

z	117	0 1 1 1 0 1 0 1
$2^4 d$	10	1 0 1 0
$s^{(0)}$		0 1 1 1 0 1 0 1
$2s^{(0)}$		0 1 1 1 0 1 0 1
$-q_3 2^4 d$		1 0 1 0 $\{q_3 = 1\}$
$s^{(1)}$		0 1 0 0 1 0 1
$2s^{(1)}$		0 1 0 0 1 0 1
$-q_2 2^4 d$		0 0 0 0 $\{q_2 = 0\}$
$s^{(2)}$		1 0 0 1 0 1
$2s^{(2)}$		1 0 0 1 0 1
$-q_1 2^4 d$		1 0 1 0 $\{q_1 = 1\}$
$s^{(3)}$		1 0 0 0 1
$2s^{(3)}$		1 0 0 0 1
$-q_0 2^4 d$		1 0 1 0 $\{q_0 = 1\}$

Fractional division

z_{frac}	. 0 1 1 1 0 1 0 1
d_{frac}	. 1 0 1 0
$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_{-1} d$	. 1 0 1 0 $\{q_{-1}=1\}$
$s^{(1)}$	. 0 1 0 0 1 0 1
$2s^{(1)}$	0 . 1 0 0 1 0 1
$-q_{-2} d$	. 0 0 0 0 $\{q_{-2}=0\}$
$s^{(2)}$	. 1 0 0 1 0 1
$2s^{(2)}$	1 . 0 0 1 0 1
$-q_{-3} d$	. 1 0 1 0 $\{q_{-3}=1\}$
$s^{(3)}$	. 1 0 0 0 1
$2s^{(3)}$	1 . 0 0 0 1
$-q_{-4} d$	. 1 0 1 0 $\{q_{-4}=1\}$

Examples of sequential division with integer and fractional operands.

Example of basic division

Decimal

Integer division

z	117	0 1 1 1 0 1 0 1
$2^4 d$	10	1 0 1 0
$s^{(0)}$		0 1 1 1 0 1 0 1
$2s^{(0)}$		0 1 1 1 0 1 0 1
$-q_3 2^4 d$		1 0 1 0 $\{q_3 = 1\}$
$s^{(1)}$		0 1 0 0 1 0 1
$2s^{(1)}$		0 1 0 0 1 0 1
$-q_2 2^4 d$		0 0 0 0 $\{q_2 = 0\}$
$s^{(2)}$		1 0 0 1 0 1
$2s^{(2)}$		1 0 0 1 0 1
$-q_1 2^4 d$		1 0 1 0 $\{q_1 = 1\}$
$s^{(3)}$		1 0 0 0 1
$2s^{(3)}$		1 0 0 0 1
$-q_0 2^4 d$		1 0 1 0 $\{q_0 = 1\}$
$s^{(4)}$		0 1 1 1
s		0 1 1 1
q		1 0 1 1

Fractional division

z_{frac}	. 0 1 1 1 0 1 0 1
d_{frac}	. 1 0 1 0
$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_{-1} d$	. 1 0 1 0 $\{q_{-1}=1\}$
$s^{(1)}$	. 0 1 0 0 1 0 1
$2s^{(1)}$	0 . 1 0 0 1 0 1
$-q_{-2} d$	. 0 0 0 0 $\{q_{-2}=0\}$
$s^{(2)}$	. 1 0 0 1 0 1
$2s^{(2)}$	1 . 0 0 1 0 1
$-q_{-3} d$	. 1 0 1 0 $\{q_{-3}=1\}$
$s^{(3)}$	. 1 0 0 0 1
$2s^{(3)}$	1 . 0 0 0 1
$-q_{-4} d$	. 1 0 1 0 $\{q_{-4}=1\}$
$s^{(4)}$	. 0 1 1 1
s_{frac}	0 . 0 0 0 0 0 1 1 1
q_{frac}	. 1 0 1 1

Examples of sequential division with integer and fractional operands.

Example of basic division

Decimal

Integer division

z	117	0 1 1 1 0 1 0 1
$2^4 d$	10	1 0 1 0
$s^{(0)}$		0 1 1 1 0 1 0 1
$2s^{(0)}$		0 1 1 1 0 1 0 1
$-q_3 2^4 d$		1 0 1 0 $\{q_3 = 1\}$
$s^{(1)}$		0 1 0 0 1 0 1
$2s^{(1)}$		0 1 0 0 1 0 1
$-q_2 2^4 d$		0 0 0 0 $\{q_2 = 0\}$
$s^{(2)}$		1 0 0 1 0 1
$2s^{(2)}$		1 0 0 1 0 1
$-q_1 2^4 d$		1 0 1 0 $\{q_1 = 1\}$
$s^{(3)}$		1 0 0 0 1
$2s^{(3)}$		1 0 0 0 1
$-q_0 2^4 d$		1 0 1 0 $\{q_0 = 1\}$
$s^{(4)}$		0 1 1 1
s		0 1 1 1
q	11	1 0 1 1

Fractional division

z_{frac}	. 0 1 1 1 0 1 0 1
d_{frac}	. 1 0 1 0
$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_{-1} d$	. 1 0 1 0 $\{q_{-1}=1\}$
$s^{(1)}$	. 0 1 0 0 1 0 1
$2s^{(1)}$	0 . 1 0 0 1 0 1
$-q_{-2} d$	. 0 0 0 0 $\{q_{-2}=0\}$
$s^{(2)}$	. 1 0 0 1 0 1
$2s^{(2)}$	1 . 0 0 1 0 1
$-q_{-3} d$	. 1 0 1 0 $\{q_{-3}=1\}$
$s^{(3)}$	. 1 0 0 0 1
$2s^{(3)}$	1 . 0 0 0 1
$-q_{-4} d$	. 1 0 1 0 $\{q_{-4}=1\}$
$s^{(4)}$	. 0 1 1 1
s_{frac}	0 . 0 0 0 0 0 1 1 1
q_{frac}	. 1 0 1 1

Examples of sequential division with integer and fractional operands.

Example of basic division

Decimal

Integer division

z	117	0 1 1 1 0 1 0 1
$2^4 d$	10	1 0 1 0
$s^{(0)}$		0 1 1 1 0 1 0 1
$2s^{(0)}$		0 1 1 1 0 1 0 1
$-q_3 2^4 d$		1 0 1 0 $\{q_3 = 1\}$
$s^{(1)}$		0 1 0 0 1 0 1
$2s^{(1)}$		0 1 0 0 1 0 1
$-q_2 2^4 d$		0 0 0 0 $\{q_2 = 0\}$
$s^{(2)}$		1 0 0 1 0 1
$2s^{(2)}$		1 0 0 1 0 1
$-q_1 2^4 d$		1 0 1 0 $\{q_1 = 1\}$
$s^{(3)}$		1 0 0 0 1
$2s^{(3)}$		1 0 0 0 1
$-q_0 2^4 d$		1 0 1 0 $\{q_0 = 1\}$
$s^{(4)}$		0 1 1 1
s	7	0 1 1 1
q	11	1 0 1 1

Fractional division

z_{frac}	. 0 1 1 1 0 1 0 1
d_{frac}	. 1 0 1 0
$s^{(0)}$	. 0 1 1 1 0 1 0 1
$2s^{(0)}$	0 . 1 1 1 0 1 0 1
$-q_{-1} d$	. 1 0 1 0 $\{q_{-1}=1\}$
$s^{(1)}$	. 0 1 0 0 1 0 1
$2s^{(1)}$	0 . 1 0 0 1 0 1
$-q_{-2} d$	. 0 0 0 0 $\{q_{-2}=0\}$
$s^{(2)}$	. 1 0 0 1 0 1
$2s^{(2)}$	1 . 0 0 1 0 1
$-q_{-3} d$	. 1 0 1 0 $\{q_{-3}=1\}$
$s^{(3)}$	. 1 0 0 0 1
$2s^{(3)}$	1 . 0 0 0 1
$-q_{-4} d$	. 1 0 1 0 $\{q_{-4}=1\}$
$s^{(4)}$	. 0 1 1 1
s_{frac}	0 . 0 0 0 0 0 1 1 1
q_{frac}	. 1 0 1 1

Examples of sequential division with integer and fractional operands.

Floating point representation

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$
Rational	$\pm p/q$
Floating-point	$\pm s \times b^e$
Logarithmic	$\pm \log_b x$

Floating point representation

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$
Rational	$\pm p/q$
Floating-point	$\pm s \times b^e$
Logarithmic	$\pm \log_b x$

Low precision and/or range
Difficult arithmetic
Most common scheme
Limiting case of floating-point

Floating point representation

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$
Rational	$\pm p/q$
Floating-point	$\pm s \times b^e$
Logarithmic	$\pm \log_b x$

Fixed-point numbers

$x = (0000\ 0000.0000\ 1001)_{two}$	Small number
$y = (1001\ 0000.0000\ 0000)_{two}$	Large number

Low precision and/or range
Difficult arithmetic
Most common scheme
Limiting case of floating-point

Floating point representation

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$
Rational	$\pm p/q$
Floating-point	$\pm s \times b^e$
Logarithmic	$\pm \log_b x$

Fixed-point numbers

$x = (0000\ 0000.0000\ 1001)_{two}$ Small number
$y = (1001\ 0000.0000\ 0000)_{two}$ Large number

Low precision and/or range
Difficult arithmetic
Most common scheme
Limiting case of floating-point

Square of neither number representable

Floating point representation

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$
Rational	$\pm p/q$
Floating-point	$\pm s \times b^e$
Logarithmic	$\pm \log_b x$

Fixed-point numbers

$x = (0000\ 0000.0000\ 1001)_{two}$ Small number
$y = (1001\ 0000.0000\ 0000)_{two}$ Large number

Floating-point numbers

$x = \pm s \times b^e$ or $\pm \text{significand} \times \text{base}^{\text{exponent}}$

Low precision and/or range
Difficult arithmetic
Most common scheme
Limiting case of floating-point

Square of neither number representable

Floating point representation

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$
Rational	$\pm p/q$
Floating-point	$\pm s \times b^e$
Logarithmic	$\pm \log_b x$

Fixed-point numbers

$x = (0000\ 0000.0000\ 1001)_{two}$ Small number
$y = (1001\ 0000.0000\ 0000)_{two}$ Large number

Floating-point numbers

$x = \pm s \times b^e$ or $\pm \text{significand} \times \text{base}^{\text{exponent}}$

Low precision and/or range
Difficult arithmetic
Most common scheme
Limiting case of floating-point

Square of neither number representable

$x = 1.001 \times 2^{-5}$
$y = 1.001 \times 2^{+7}$

Floating point representation

No finite number system can represent all real numbers
Various systems can be used for a subset of real numbers

Fixed-point	$\pm w.f$
Rational	$\pm p/q$
Floating-point	$\pm s \times b^e$
Logarithmic	$\pm \log_b x$

Fixed-point numbers

$x = (0000\ 0000.0000\ 1001)_{two}$ Small number
$y = (1001\ 0000.0000\ 0000)_{two}$ Large number

Floating-point numbers

$x = \pm s \times b^e$ or $\pm \text{significand} \times \text{base}^{\text{exponent}}$

A floating-point number comes with two signs:

Number sign, usually appears as a separate bit
Exponent sign, usually embedded in the biased exponent

Low precision and/or range
Difficult arithmetic
Most common scheme
Limiting case of floating-point

Square of neither number representable

$x = 1.001 \times 2^{-5}$
$y = 1.001 \times 2^{+7}$

Floating point prior to IEEE standard

- Computer manufacturers tended to have their own hardware-level formats
- This created many problems, as floating-point computations could produce vastly different results (not just differing in the last few significant bits)

Floating point prior to IEEE standard

- Computer manufacturers tended to have their own hardware-level formats
- This created many problems, as floating-point computations could produce vastly different results (not just differing in the last few significant bits)

- To get a sense for the wide variations in floating-point formats, visit:

http://www.mrob.com/pub/math/floatformats.html

Floating point prior to IEEE standard

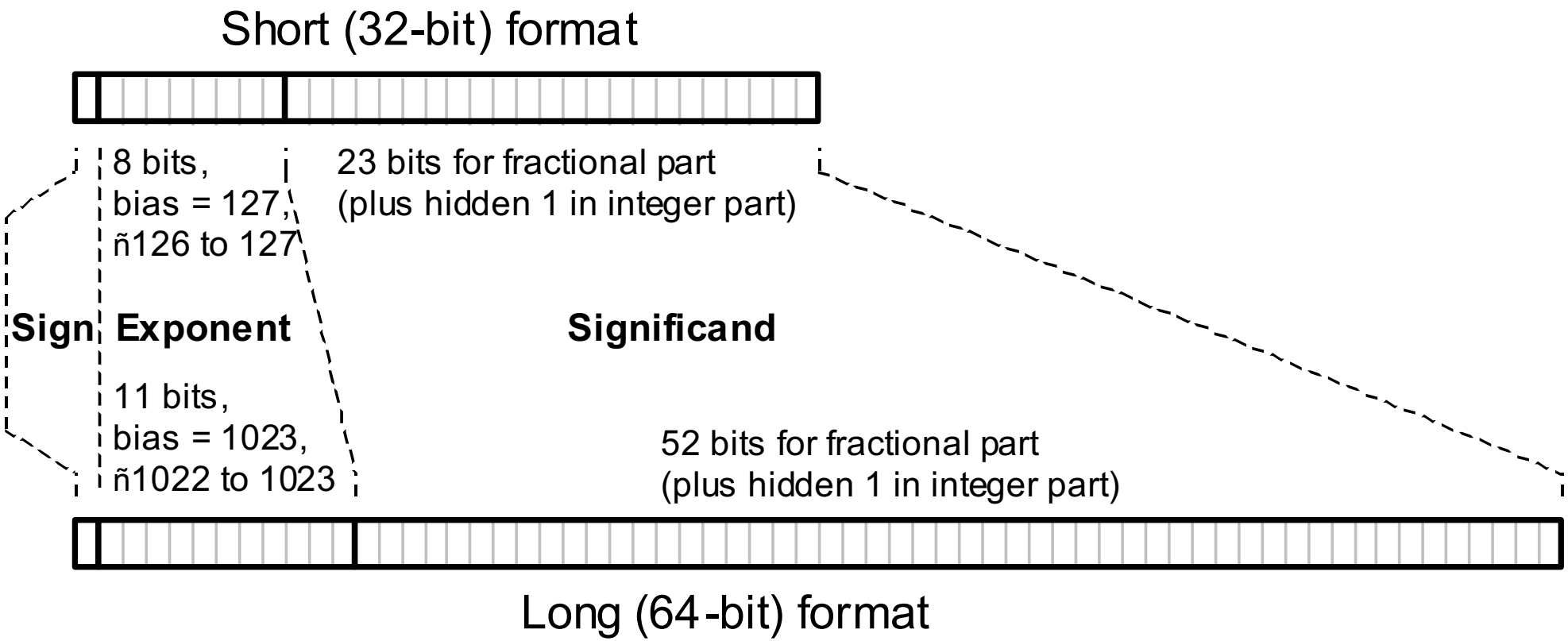
- Computer manufacturers tended to have their own hardware-level formats
- This created many problems, as floating-point computations could produce vastly different results (not just differing in the last few significant bits)

- To get a sense for the wide variations in floating-point formats, visit:

http://www.mrob.com/pub/math/floatformats.html

- First IEEE standard for binary floating-point arithmetic was adopted in 1985 after years of discussion
- The 1985 standard was continuously discussed, criticized, and clarified for a couple of decades
- In 2008, after several years of discussion, a revised standard was issued

The IEEE 754 floating point standard representation



The IEEE 754 standard floating-point number representation formats

Overview of IEEE 754 standard

Feature	Single / Short	Double / Long
Word width (bits)	32	64
Significand bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (± 0)	$e + bias = 0$, $f = 0$	$e + bias = 0$, $f = 0$
Denormal	$e + bias = 0$, $f \neq 0$	$e + bias = 0$, $f \neq 0$
	represents $\pm 0.f \times 2^{-126}$	represents $\pm 0.f \times 2^{-1022}$
Infinity ($\pm\infty$)	$e + bias = 255$, $f = 0$	$e + bias = 2047$, $f = 0$
Not-a-number (NaN).	$e + bias = 255$, $f \neq 0$	$e + bias = 2047$, $f \neq 0$
Ordinary number	$e + bias \in [1, 254]$	$e + bias \in [1, 2046]$
	$e \in [-126, 127]$	$e \in [-1022, 1023]$
	represents $1.f \times 2^e$	represents $1.f \times 2^e$
min	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
max	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

Floating point representation

- Representation format

Floating point representation

- Representation format
- S = sign bit 0 = positive 1 = negative

Floating point representation

- Representation format

- S = sign bit 0 = positive 1 = negative
- Biased exponent 8-bits for single and 11-bits for double precision
  - It should represent both positive and negative exponents
  - Bias is added to actual exponent 127 for single precision and

Floating point representation

- Representation format

- S = sign bit 0 = positive 1 = negative
- Biased exponent 8-bits for single and 11-bits for double precision
  - It should represent both positive and negative exponents
  - Bias is added to actual exponent 127 for single precision and
- Normalised mantissa ➔ only one ‘1’ to the left of the decimal point

Floating point representation

- Representation format

- S = sign bit 0 = positive 1 = negative
- Biased exponent 8-bits for single and 11-bits for double precision
  - It should represent both positive and negative exponents
  - Bias is added to actual exponent 127 for single precision and
- Normalised mantissa ➔ only one ‘1’ to the left of the decimal point
- Special values
  - Zero ➔ all exponent and mantissa of 0 values-0 and +0 are equal
  - Denormalised ➔ exponent = 0 and mantissa ≠ 0, i.e., the number does not have leading 1 before the binary point
  - Infinity ➔ all exponent ‘1’ and all mantissa ‘0’. Sign bit 1 = -∞ and 0 = ∞
  - NAN ➔represents error value exponent all ‘1’ mantissa ≠ 0

Floating point representation: conversion

- Conversion example: Convert 100_{10} to single precision

Floating point representation: conversion

- Conversion example: Convert 100_{10} to single precision
  - Step1: convert it to binary ➔ 0110 0100

Floating point representation: conversion

- Conversion example: Convert 100_{10} to single precision
  - Step1: convert it to binary ➔ 0110 0100
  - Step2: represent the binary in the form of 1.xxx
    - 0110 0100 = 1.100100×2^6

Floating point representation: conversion

- Conversion example: Convert 100_{10} to single precision
  - Step1: convert it to binary ➔ 0110 0100
  - Step2: represent the binary in the form of 1.xxx
    - 0110 0100 = 1.100100×2^6
  - Thus, the three elements are:
    - Exponent = 6, biased exponent will be 6+127 = 133 = 1000 0101
    - Sign will be 0 for positive number
    - Stored fraction will be: 1001

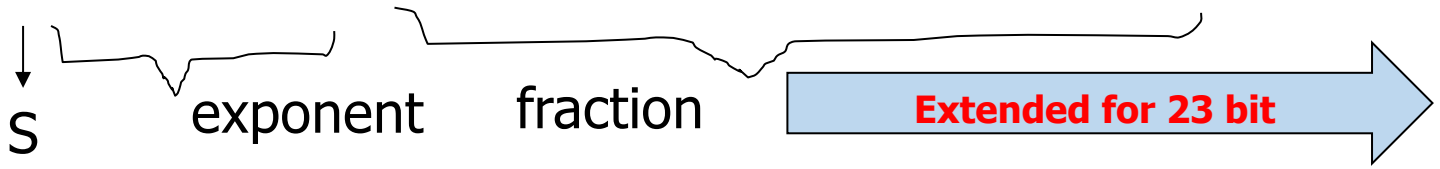
Floating point representation: conversion

- Conversion example: Convert 100_{10} to single precision
  - Step1: convert it to binary ➔ 0110 0100
  - Step2: represent the binary in the form of 1.xxx
    - 0110 0100 = 1.100100×2^6
  - Thus, the three elements are:
    - Exponent = 6, biased exponent will be 6+127 = 133 = 1000 0101
    - Sign will be 0 for positive number
    - Stored fraction will be: 1001
  - The floating point equivalent representation of the number is:

    - 0 10000101 10010000000000000000000

Floating point representation: conversion

- Conversion example: Convert 100_{10} to single precision
  - Step1: convert it to binary ➔ 0110 0100
  - Step2: represent the binary in the form of 1.xxx
    - 0110 0100 = 1.100100×2^6
  - Thus, the three elements are:
    - Exponent = 6, biased exponent will be 6+127 = 133 = 1000 0101
    - Sign will be 0 for positive number
    - Stored fraction will be: 1001
  - The floating point equivalent representation of the number is:
    - 0 10000101 10010000000000000000000



Floating point arithmetic

Results of the 4 basic arithmetic operations (+, −, ×, ÷) as well as square-rooting must match those obtained if all intermediate computations were infinitely precise

Floating point arithmetic

Results of the 4 basic arithmetic operations (+, −, ×, ÷) as well as square-rooting must match those obtained if all intermediate computations were infinitely precise

That is, a floating point arithmetic operation should introduce no more imprecision than the error attributable to the final rounding of a result that has no exact representation (this is the best possible)

Floating point arithmetic

Results of the 4 basic arithmetic operations (+, −, ×, ÷) as well as square-rooting must match those obtained if all intermediate computations were infinitely precise

That is, a floating point arithmetic operation should introduce no more imprecision than the error attributable to the final rounding of a result that has no exact representation (this is the best possible)

Example:

$$(1 + 2^{-1}) \quad \times \quad (1 + 2^{-23})$$

Floating point arithmetic

Results of the 4 basic arithmetic operations (+, −, ×, ÷) as well as square-rooting must match those obtained if all intermediate computations were infinitely precise

That is, a floating point arithmetic operation should introduce no more imprecision than the error attributable to the final rounding of a result that has no exact representation (this is the best possible)

Example:

$$(1 + 2^{-1}) \times (1 + 2^{-23})$$

Exact result $1 + 2^{-1} + 2^{-23} + 2^{-24}$

Floating point arithmetic

Results of the 4 basic arithmetic operations (+, –, ×, ÷) as well as square-rooting must match those obtained if all intermediate computations were infinitely precise

That is, a floating point arithmetic operation should introduce no more imprecision than the error attributable to the final rounding of a result that has no exact representation (this is the best possible)

Example:

$$(1 + 2^{-1}) \times (1 + 2^{-23})$$

Exact result $1 + 2^{-1} + 2^{-23} + 2^{-24}$

Chopped result $1 + 2^{-1} + 2^{-23}$ Error = –½ *ulp*

Floating point arithmetic

Results of the 4 basic arithmetic operations (+, –, ×, ÷) as well as square-rooting must match those obtained if all intermediate computations were infinitely precise

That is, a floating point arithmetic operation should introduce no more imprecision than the error attributable to the final rounding of a result that has no exact representation (this is the best possible)

Example:

$$(1 + 2^{-1}) \times (1 + 2^{-23})$$

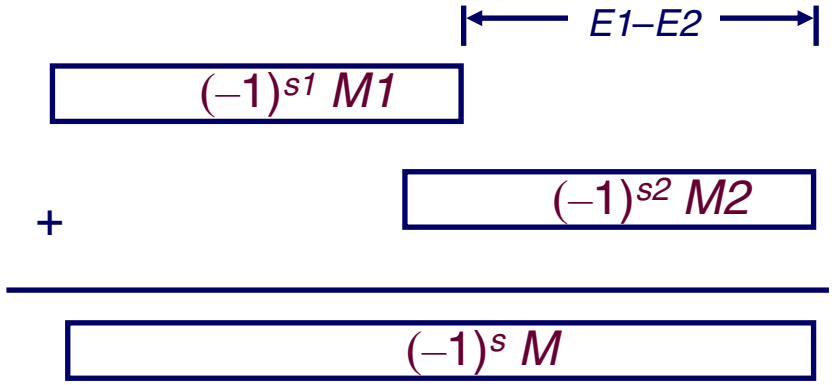
Exact result $1 + 2^{-1} + 2^{-23} + 2^{-24}$

Chopped result $1 + 2^{-1} + 2^{-23}$ Error = –½ *ulp*

Rounded result $1 + 2^{-1} + 2^{-22}$ Error = +½ *ulp*

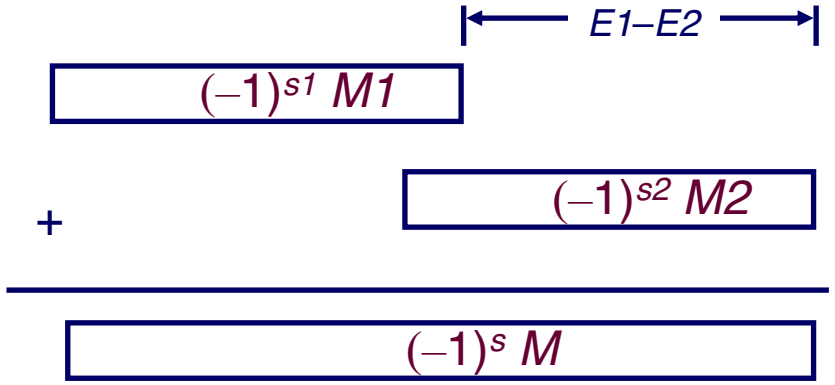
Floating point arithmetic

- Floating point addition example:
  - **Operands**
    - $(-1)^{s1}\ M1\ \ 2^{E1}$
    - $(-1)^{s2}\ M2\ \ 2^{E2}$
    - **Assume** $E1 > E2$



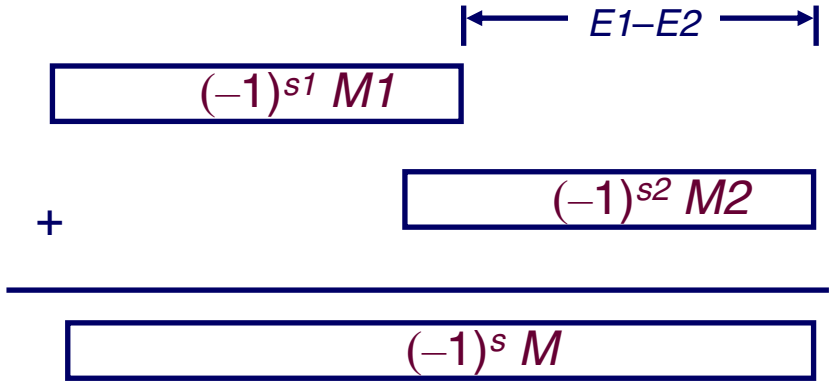
Floating point arithmetic

- Floating point addition example:
  - **Operands**
    - $(-1)^{s1}\ M1\ \ 2^{E1}$
    - $(-1)^{s2}\ M2\ \ 2^{E2}$
    - **Assume** $E1 > E2$
  - **Exact Result**
    - $(-1)^{s}\ M\ \ 2^{E}$
    - **Sign** s**, significand** M**:**
      - **Result of signed align & add**
    - **Exponent** E**:** $E1$



Floating point arithmetic

- Floating point addition example:
  - **Operands**
    - $(-1)^{s1}\ M1\ \ 2^{E1}$
    - $(-1)^{s2}\ M2\ \ 2^{E2}$
    - **Assume** $E1 > E2$
  - **Exact Result**
    - $(-1)^{s}\ M\ \ 2^{E}$
    - **Sign** s**, significand** M**:**
      - **Result of signed align & add**
    - **Exponent** E**:** $E1$



Fixing

- **If** $M \geq 2$**, shift** M **right 'k' positions, increment** E **by k**
- **if** $M < 1$**, shift** M **left '**k**' positions, decrement** E **by** k
- **Overflow if** E **out of range**
- **Round** M **to fit** `frac` **precision**

Floating point addition: example

- Complex operation when compared to integer

  Consider a 4-digit binary example

  $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

Floating point addition: example

- Complex operation when compared to integer

  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

     – Right Shift number with smaller exponent

     $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

Floating point addition: example

- Complex operation when compared to integer

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
   - Right Shift number with smaller exponent

   $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
2. Add significands

   $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$

Floating point addition: example

- Complex operation when compared to integer

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
   - Right 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

Floating point addition: example

- Complex operation when compared to integer

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
   - Right 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

Floating point addition: example

- Complex operation when compared to integer

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

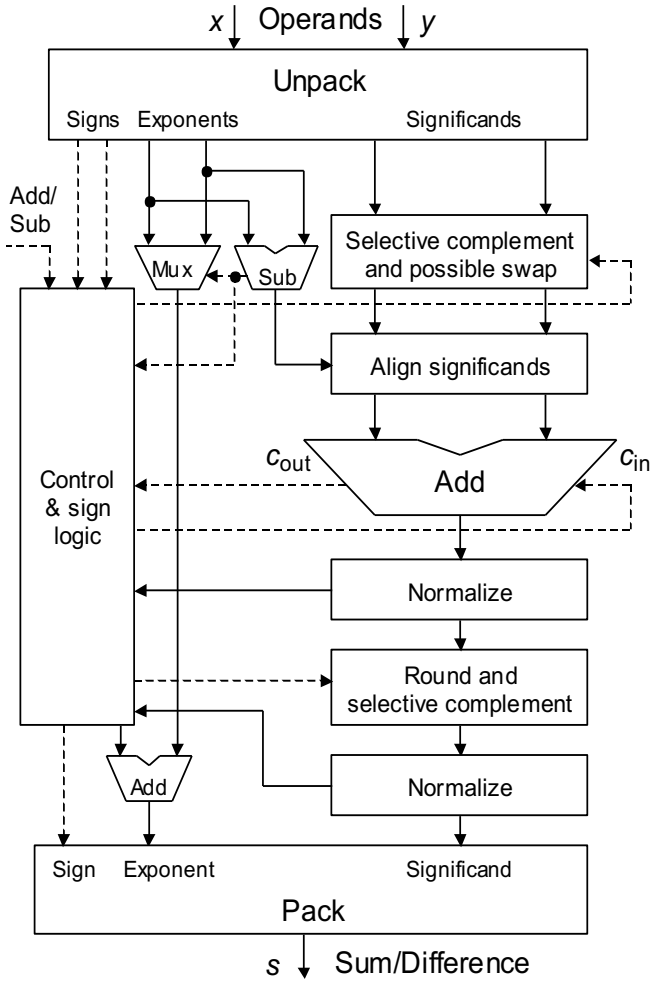
1. Align binary points
   - Right Shift number with smaller exponent

   $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
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

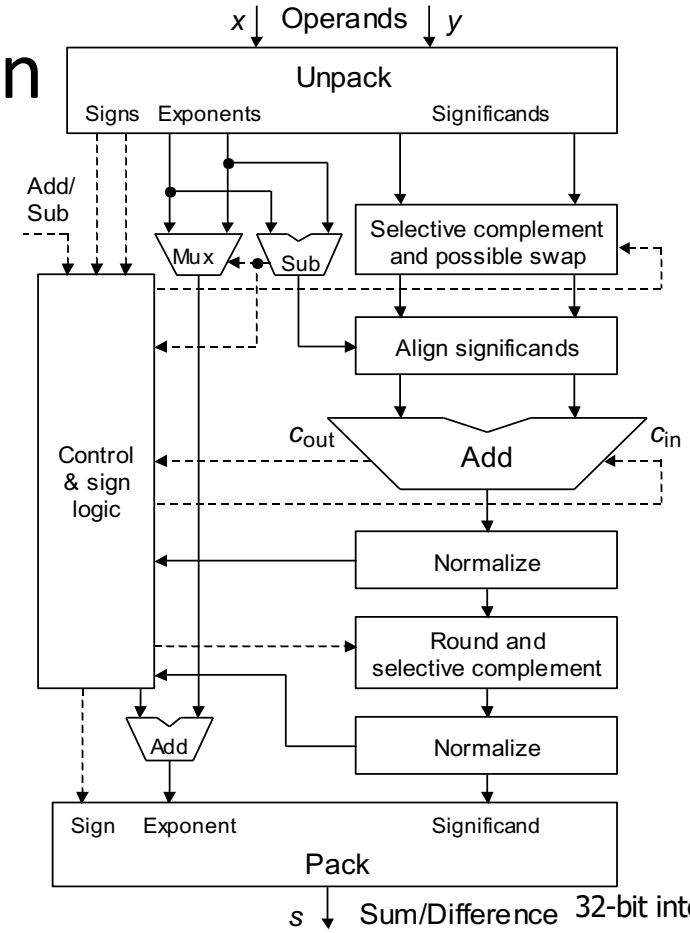
   $1.000_2 \times 2^{-4}$ (no change) = 0.0625



Floating point vs integer addition

Integer addition simple operation

- Handle carry
- Handle overflow



32-bit integer adder with carry skip adder blocks

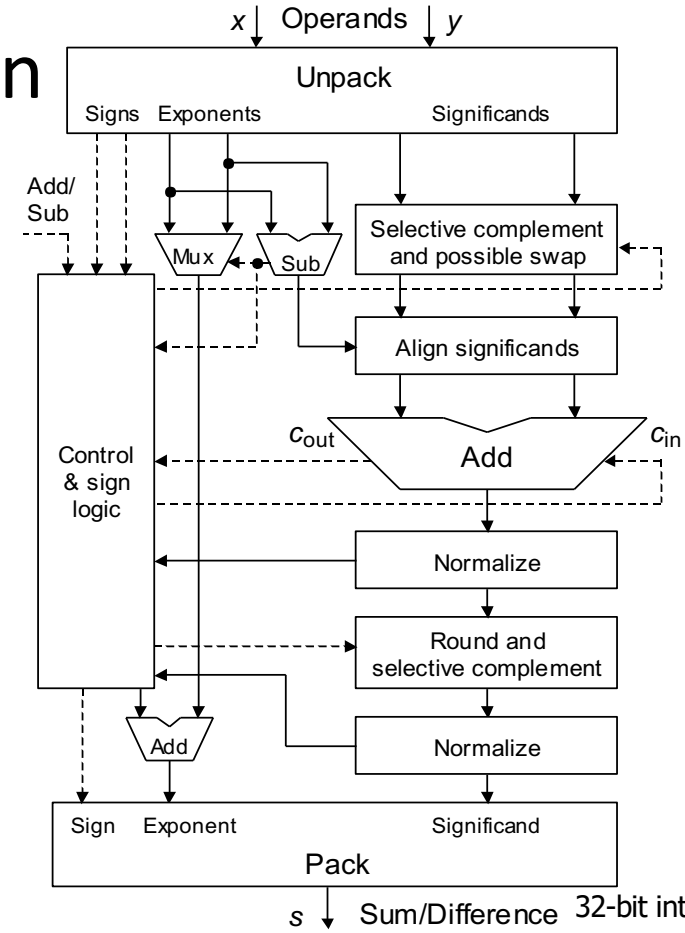
Floating point vs integer addition

Integer addition simple operation

- Handle carry
- Handle overflow

Floating point addition

- Operand pre-processing
- Integer addition for mantissas
- Exponent adjustment
- Rounding operation
- Result post-processing



32-bit integer adder with carry skip adder blocks

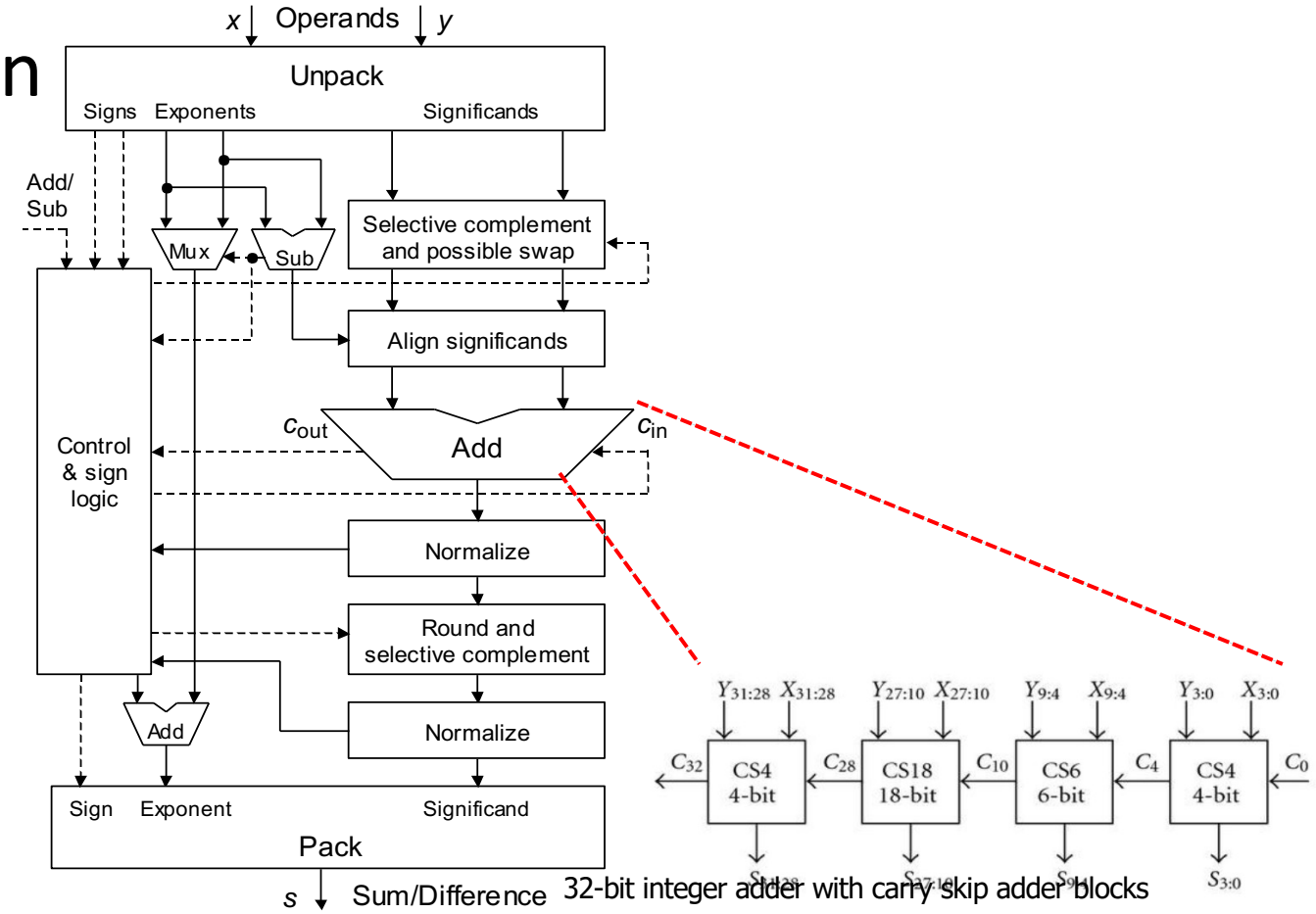
Floating point vs integer addition

Integer addition simple operation

- Handle carry
- Handle overflow

Floating point addition

- Operand pre-processing
- Integer addition for mantissas
- Exponent adjustment
- Rounding operation
- Result post-processing



32-bit integer adder with carry skip adder blocks

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1* ^ *s2*
  - **Significand** *M*: *M1* * *M2*
  - **Exponent** *E*: *E1* + *E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1* ^ *s2*
  - **Significand** *M*: *M1* * *M2*
  - **Exponent** *E*: *E1* + *E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**

Running example

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1* ^ *s2*
  - **Significand** *M*: *M1* * *M2*
  - **Exponent** *E*: *E1* + *E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**

Running example

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

1. Determine result sign S1 $\oplus$ S2 (0 $\oplus$ 1 = 1)

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1 ^ s2*
  - **Significand** *M*: *M1 * M2*
  - **Exponent** *E*: *E1 + E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**

Running example

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

1. Determine result sign S1 ⊕ S2 (0 ⊕ 1 = 1)
2. Multiply significands

$1.000_2 \times 1.110_2 = 1.1102$ ⍰ $1.110_2 \times 2^{-3}$

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1 ^ s2*
  - **Significand** *M*: *M1 * M2*
  - **Exponent** *E*: *E1 + E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**

Running example

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

1. Determine result sign S1 ⊕ S2 (0 ⊕ 1 = 1)
2. Multiply significands

$1.000_2 \times 1.110_2 = 1.1102$? $1.110_2 \times 2^{-3}$

3. Add exponents
   - Unbiased: –1 + –2 = –3
   - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1 ^ s2*
  - **Significand** *M*: *M1 * M2*
  - **Exponent** *E*: *E1 + E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**

Running example

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

1. Determine result sign S1 ⊕ S2 (0 ⊕ 1 = 1)
2. Multiply significands

$1.000_2 \times 1.110_2 = 1.1102$ ⍰ $1.110_2 \times 2^{-3}$

3. Add exponents
   - Unbiased: –1 + –2 = –3
   - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
4. Normalize result & check for over/underflow
   - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1 ^ s2*
  - **Significand** *M*: *M1 * M2*
  - **Exponent** *E*: *E1 + E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**

Running example

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

1. Determine result sign S1 ⊕ S2 (0 ⊕ 1 = 1)
2. Multiply significands

$1.000_2 \times 1.110_2 = 1.1102$? $1.110_2 \times 2^{-3}$

3. Add exponents
   - Unbiased: –1 + –2 = –3
   - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
4. Normalize result & check for over/underflow
   - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
5. Round and renormalize if necessary
   - $1.110_2 \times 2^{-3}$ (no change)

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1 ^ s2*
  - **Significand** *M*: *M1 * M2*
  - **Exponent** *E*: *E1 + E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**

Running example

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

1. Determine result sign S1 ⊕ S2 (0 ⊕ 1 = 1)
2. Multiply significands

   $1.000_2 \times 1.110_2 = 1.1102$? $1.110_2 \times 2^{-3}$
3. Add exponents
   - Unbiased: –1 + –2 = –3
   - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
4. Normalize result & check for over/underflow
   - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
5. Round and renormalize if necessary
   - $1.110_2 \times 2^{-3}$ (no change)
6. Determine final result

   $-1.110_2 \times 2^{-3} = -0.21875$

Floating point multiplication

- **Operands**
  - $(-1)^{s1}\ M1\ \ 2^{E1}$ * $(-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result**
  - $(-1)^{s}\ M\ \ 2^{E}$
  - **Sign** *s*: *s1 ^ s2*
  - **Significand** *M*: *M1 * M2*
  - **Exponent** *E*: *E1 + E2*
- **Fixing**
  - **If** $M \geq 2$, **shift** *M* **right, increment** *E*
  - **If** *E* **out of range, overflow**
  - **Round** *M* **to fit frac precision**
- **Implementation**
  - **Complex part multiplying significands**

Running example

Consider a 4-digit binary example

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)

1. Determine result sign S1 ⊕ S2 (0 ⊕ 1 = 1)
2. Multiply significands

$1.000_2 \times 1.110_2 = 1.1102$ ⍰ $1.110_2 \times 2^{-3}$

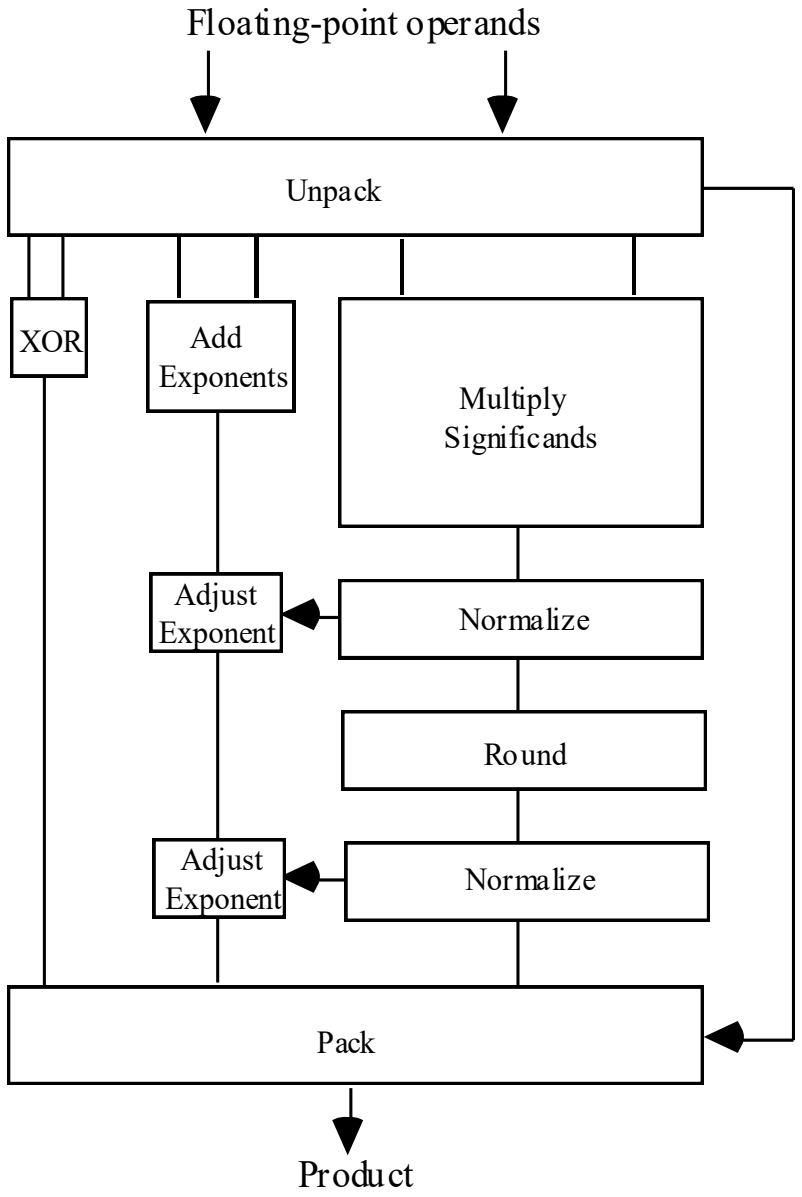
3. Add exponents
   - Unbiased: –1 + –2 = –3
   - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
4. Normalize result & check for over/underflow
   - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
5. Round and renormalize if necessary
   - $1.110_2 \times 2^{-3}$ (no change)
6. Determine final result

$-1.110_2 \times 2^{-3} = -0.21875$

Floating point mulitipier hardware

$$(\pm s1 \times b^{e1}) \times (\pm s2 \times b^{e2}) = (\pm s1 \times s2) \times b^{e1+e2}$$

$s1 \times s2 \in [1, 4)$: may need post-shifting

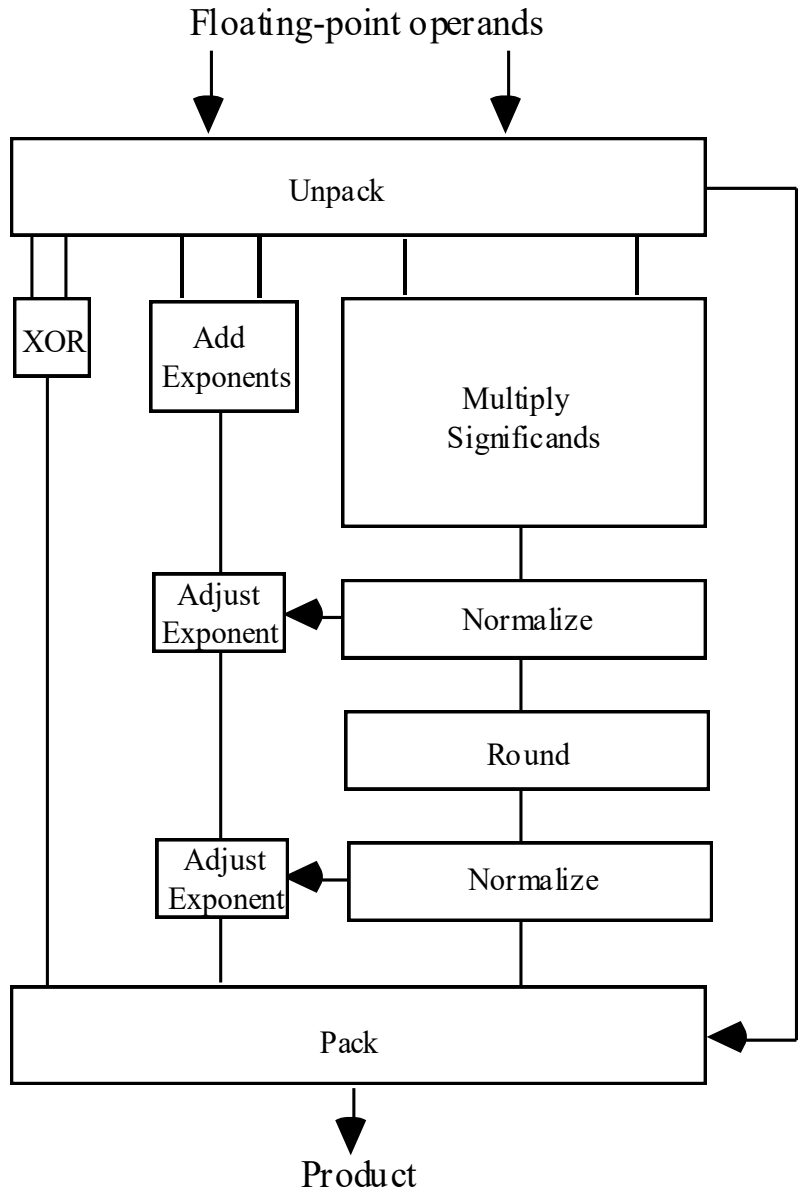


Floating point mulitipier hardware

$$(\pm s1 \times b^{e1}) \times (\pm s2 \times b^{e2}) = (\pm s1 \times s2) \times b^{e1+e2}$$

$s1 \times s2 \in [1, 4)$: may need post-shifting

Overflow or underflow can occur during multiplication or normalization



Floating point mulitipier hardware

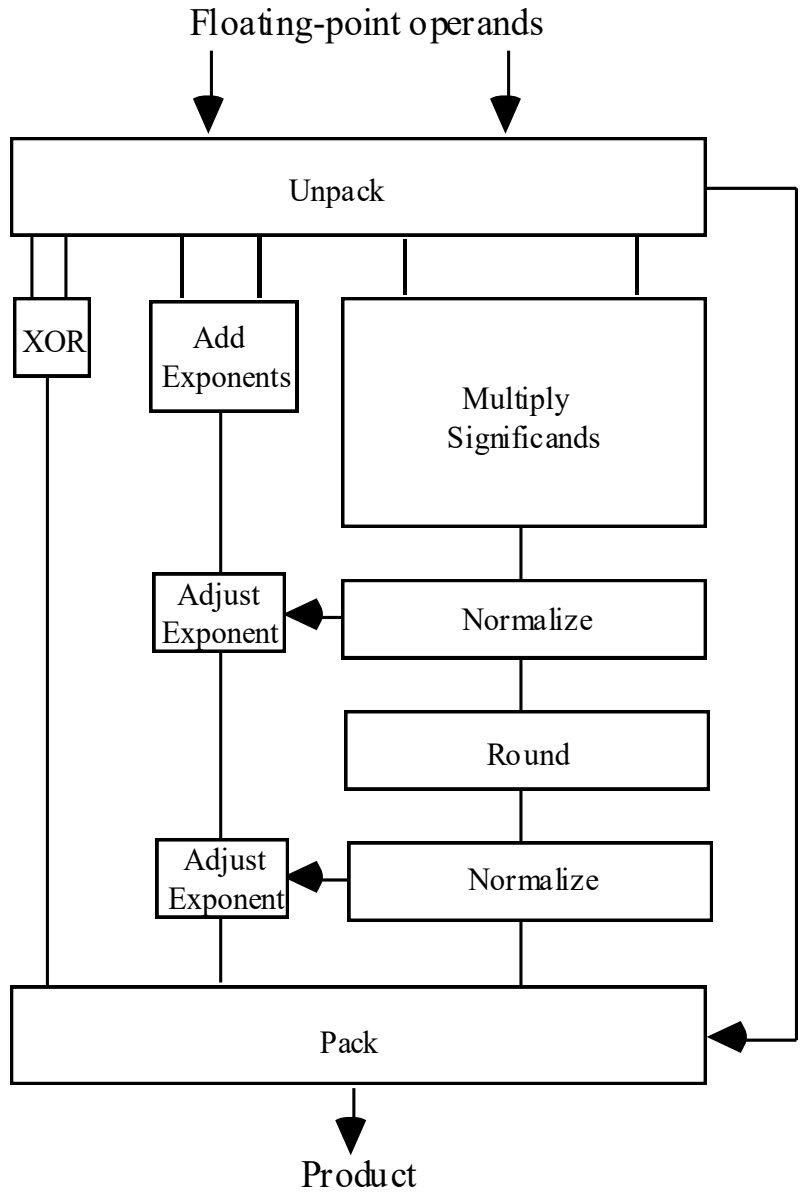
$$(\pm s1 \times b^{e1}) \times (\pm s2 \times b^{e2}) = (\pm s1 \times s2) \times b^{e1+e2}$$

$s1 \times s2 \in [1, 4)$: may need post-shifting

Overflow or underflow can occur during multiplication or normalization

Considerations:

- Rounding and truncation
- Result normalization



Exceptions in floating point operation

Divide by zero

Overflow

Underflow

Exceptions in floating point operation

Divide by zero

Overflow

Underflow

Inexact result: Rounded value not the same as original

Exceptions in floating point operation

Divide by zero

Overflow

Underflow

Inexact result: Rounded value not the same as original

Invalid operation: examples include

Addition	$(+\infty) + (-\infty)$
Multiplication	$0 \times \infty$
Division	$0/0$ or ∞/∞
Square-rooting	operand < 0

Exceptions in floating point operation

Divide by zero

Overflow

Underflow

Inexact result: Rounded value not the same as original

Invalid operation: examples include

Addition	$(+\infty) + (-\infty)$
Multiplication	$0 \times \infty$
Division	$0/0$ or ∞/∞
Square-rooting	operand < 0

Produce NaN as their results

Advanced cases: big problems in today's computing

- Too much energy and power needed per calculation
- Not enough bandwidth (the "memory wall")
- Rounding errors prevent use of parallel methods
- IEEE floats give different answers on different platforms

Advanced cases: big problems in today's computing

The ones vendors
care most about

- Too much energy and power needed per calculation
- Not enough bandwidth (the "memory wall")
- Rounding errors prevent use of parallel methods
- IEEE floats give different answers on different platforms

Advanced cases: big problems in today's computing

The ones vendors care most about

- Too much energy and power needed per calculation
- More hardware parallelism than we know how to use
- Not enough bandwidth (the "memory wall")
- Rounding errors more treacherous than people realize
- Rounding errors prevent use of parallel methods
- Sampling errors turn physics simulations into guesswork
- Numerical methods are hard to use, require experts
- IEEE floats give different answers on different platforms

Advanced cases: big problems in today's computing

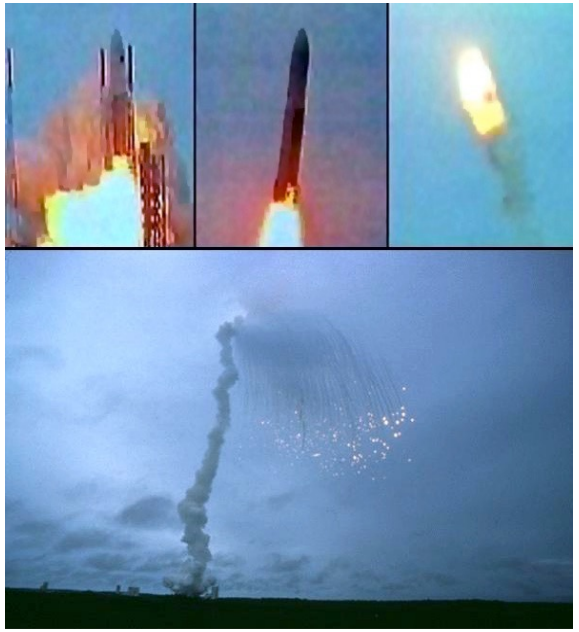
The ones vendors care most about

The ones *designers* care most about

- Too much energy and power needed per calculation
- More hardware parallelism than we know how to use
- Not enough bandwidth (the "memory wall")
- Rounding errors more treacherous than people realize
- Rounding errors prevent use of parallel methods
- Sampling errors turn physics simulations into guesswork
- Numerical methods are hard to use, require experts
- IEEE floats give different answers on different platforms

Advanced cases: exactness and float disaster

"The computer cannot give you the exact value, sorry. Use *this* value instead. It's close."

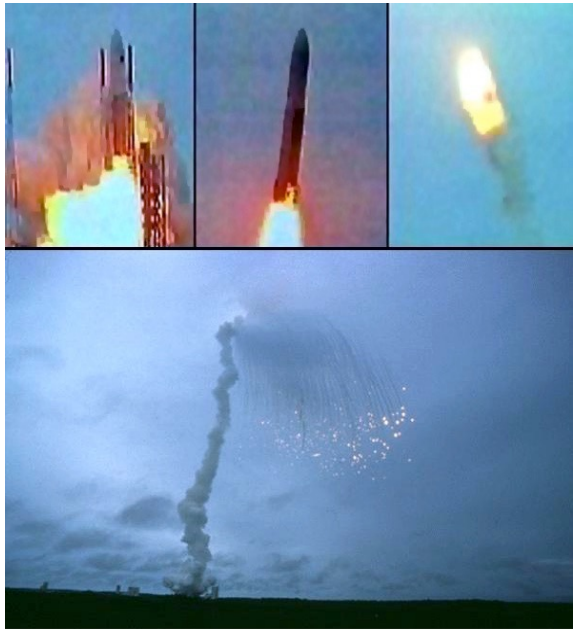


- 64-bit float measured speed
- 16-bit guidance system; *oops*
- $0.7 billion gone in seconds

Floats *prevent use of parallelism*

Advanced cases: exactness and float disaster

"The computer cannot give you the exact value, sorry. Use *this* value instead. It's close."



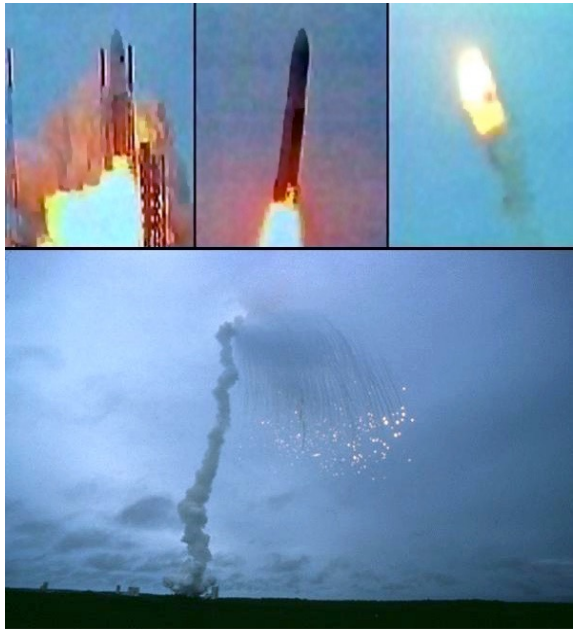
- 64-bit float measured speed
- 16-bit guidance system; *oops*
- $0.7 billion gone in seconds

Floats *prevent use of parallelism*

- No associative property for floats

Advanced cases: exactness and float disaster

"The computer cannot give you the exact value, sorry. Use *this* value instead. It's close."



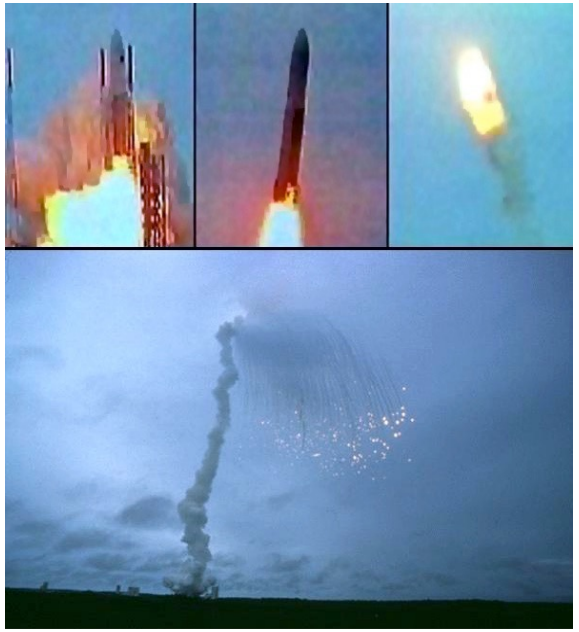
- 64-bit float measured speed
- 16-bit guidance system; *oops*
- $0.7 billion gone in seconds

Floats *prevent use of parallelism*

- No associative property for floats
- $(a + b) + (c + d)$ (parallel) ≠ $((a + b) + c) + d$ (serial)

Advanced cases: exactness and float disaster

"The computer cannot give you the exact value, sorry. Use *this* value instead. It's close."



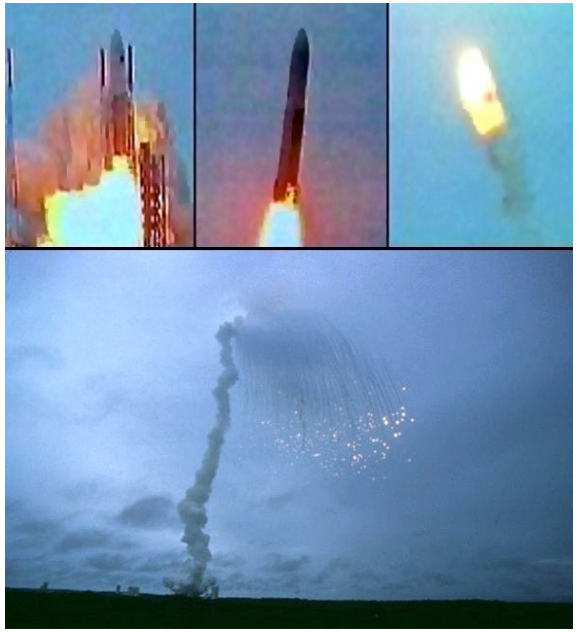
- 64-bit float measured speed
- 16-bit guidance system; *oops*
- $0.7 billion gone in seconds

Floats *prevent use of parallelism*

- No associative property for floats
- $(a + b) + (c + d)$ (parallel) $\neq ((a + b) + c) + d$ (serial)
- Looks like a "wrong answer"

Advanced cases: exactness and float disaster

"The computer cannot give you the exact value, sorry. Use *this* value instead. It's close."



- 64-bit float measured speed
- 16-bit guidance system; *oops*
- $0.7 billion gone in seconds

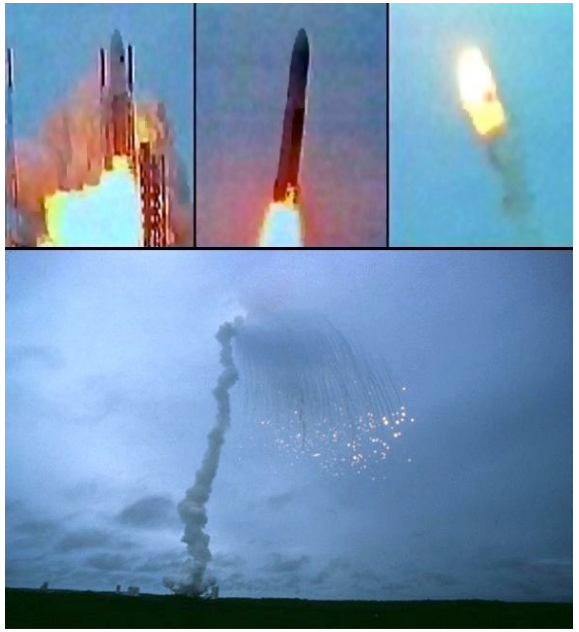
Floats *prevent use of parallelism*

- No associative property for floats
- $(a + b) + (c + d)$ (parallel) ≠ $((a + b) + c) + d$ (serial)
- Looks like a "wrong answer"
- Programmers trust serial, reject parallel

©Gustafson

Advanced cases: exactness and float disaster

"The computer cannot give you the exact value, sorry. Use *this* value instead. It's close."



- 64-bit float measured speed
- 16-bit guidance system; *oops*
- $0.7 billion gone in seconds

Floats *prevent use of parallelism*

- No associative property for floats
- $(a + b) + (c + d)$ (parallel) ≠ $((a + b) + c) + d$ (serial)
- Looks like a "wrong answer"
- Programmers trust serial, reject parallel
- IEEE floats report rounding, overflow, underflow in *processor register bits that no one ever sees*.

©Gustafson

What can we do? A new number format?

A New Number Format: The Unum

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788

©Gustafson

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums

©Gustafson

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums
- No rounding, no overflow to ∞, no underflow to zero

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums
- No rounding, no overflow to ∞, no underflow to zero
- They obey algebraic laws!

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums
- No rounding, no overflow to ∞, no underflow to zero
- They obey algebraic laws!
- Safe to parallelize

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums
- No rounding, no overflow to ∞, no underflow to zero
- They obey algebraic laws!
- Safe to parallelize
- *Fewer bits* than floats

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums
- No rounding, no overflow to ∞, no underflow to zero
- They obey algebraic laws!
- Safe to parallelize
- *Fewer bits* than floats
- But… they're *new*

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums
- No rounding, no overflow to ∞, no underflow to zero
- They obey algebraic laws!
- Safe to parallelize
- *Fewer bits* than floats
- But… they're *new*
- Some people don't like *new*

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums
- No rounding, no overflow to ∞, no underflow to zero
- They obey algebraic laws!
- Safe to parallelize
- *Fewer bits* than floats
- But… they're *new*
- Some people don't like *new*
- *Are they simple/less complex?*

What can we do? A new number format?

A New Number Format: The Unum

- Universal numbers
- **Superset** of IEEE types, both 754 and 1788
- Integers ➔floats ➔ unums
- No rounding, no overflow to ∞, no underflow to zero
- They obey algebraic laws!
- Safe to parallelize
- *Fewer bits* than floats
- But… they're *new*
- Some people don't like *new*
- *Are they simple/less complex?*

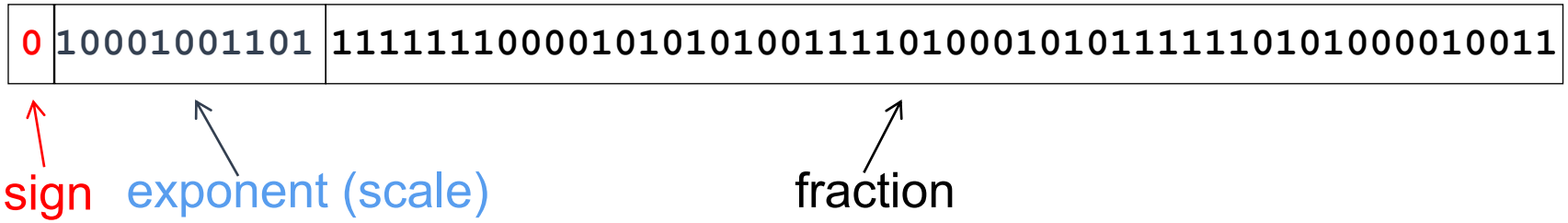
"You can't boil the ocean."

—Former Intel exec, when shown the unum idea

©Gustafson

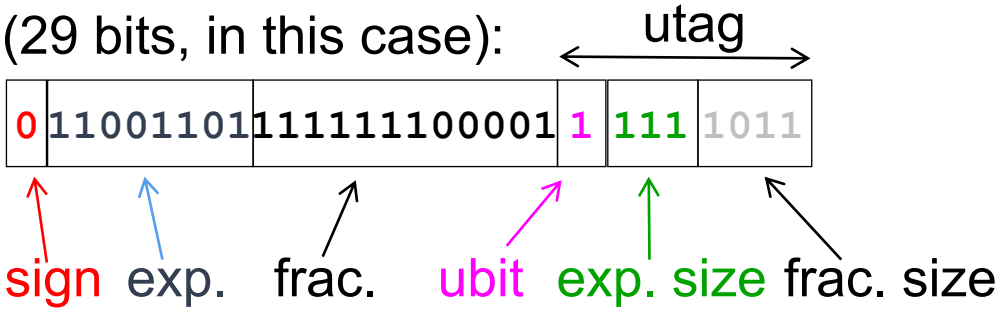
Unum basics: avoid the price of "one size fits all"

IEEE Standard Float (64 bits):



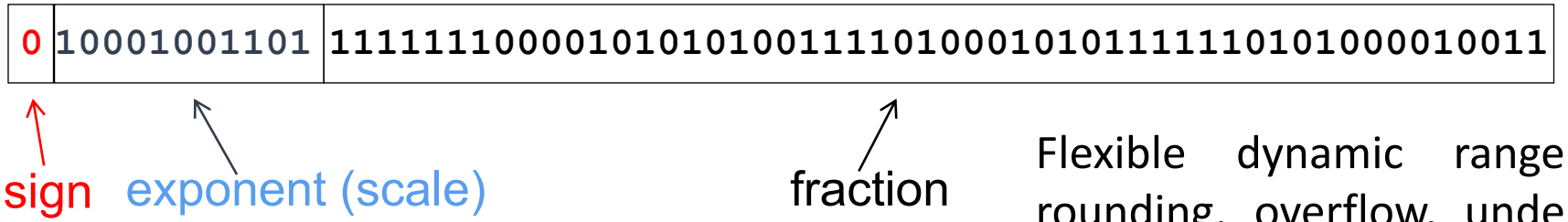
Self-descriptive "utag" bits track and manage uncertainty, exponent size, and fraction size

Unum
(29 bits, in this case):



Unum basics: avoid the price of "one size fits all"

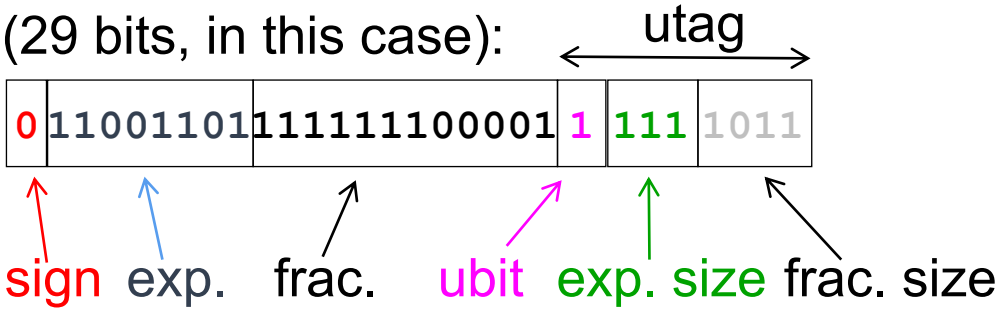
IEEE Standard Float (64 bits):



Flexible dynamic range Flexible precision No rounding, overflow, underflow No "negative zero" Fixes wasted NaN values Makes results bit-identical

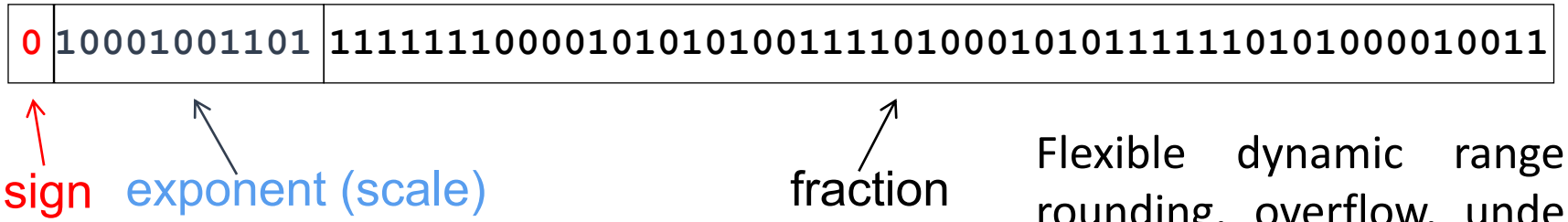
Self-descriptive "utag" bits track and manage uncertainty, exponent size, and fraction size

Unum
(29 bits, in this case):



Unum basics: avoid the price of "one size fits all"

IEEE Standard Float (64 bits):



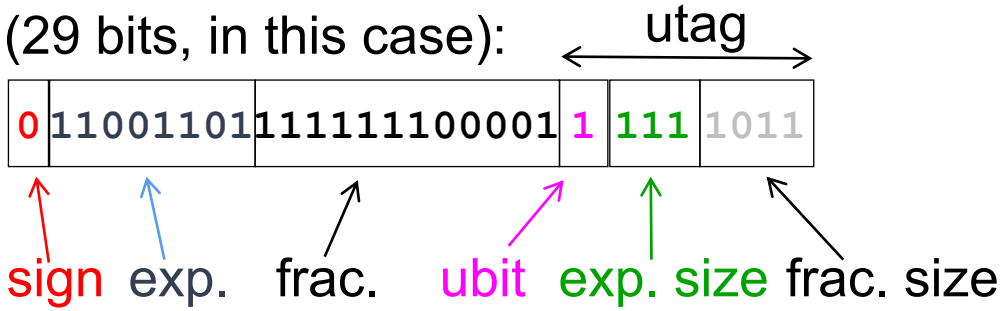
Flexible dynamic range Flexible precision No rounding, overflow, underflow No "negative zero" Fixes wasted NaN values Makes results bit-identical

Self-descriptive "utag" bits track and manage uncertainty, exponent size, and fraction size

BUT:

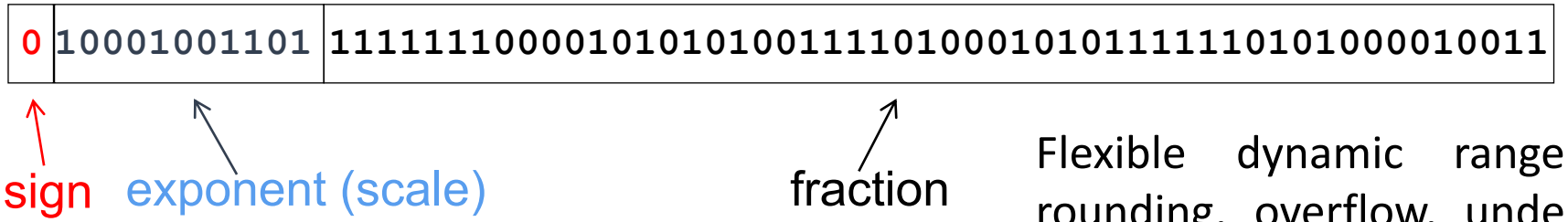
Variable storage size Adds indirection Many conditional tests

Unum
(29 bits, in this case):



Unum basics: avoid the price of "one size fits all"

IEEE Standard Float (64 bits):



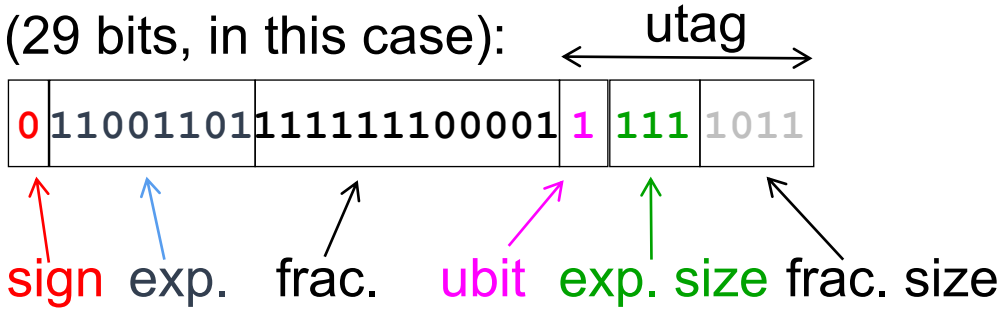
Flexible dynamic range Flexible precision No rounding, overflow, underflow No "negative zero" Fixes wasted NaN values Makes results bit-identical

Self-descriptive "utag" bits track and manage uncertainty, exponent size, and fraction size

BUT:

Variable storage size Adds indirection Many conditional tests

Unum
(29 bits, in this case):

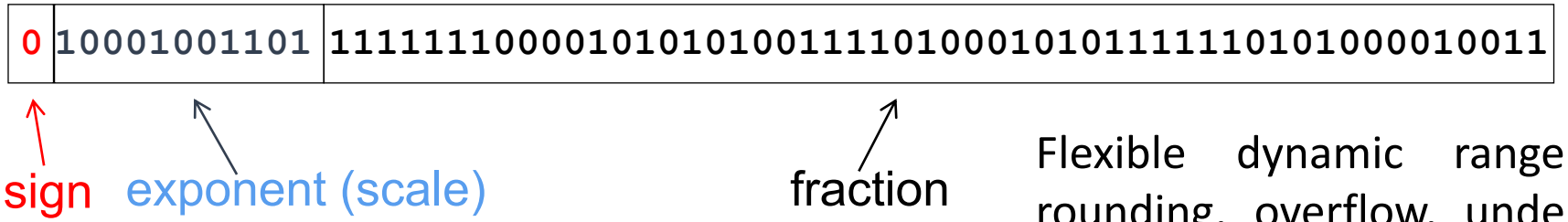



```
The biggest
objection to unums
is likely to come
from the fact that
they are variable in
size, at least when
they are stored in
packed form.
```

18-point Courier
(fixed width font)

Unum basics: avoid the price of "one size fits all"

IEEE Standard Float (64 bits):



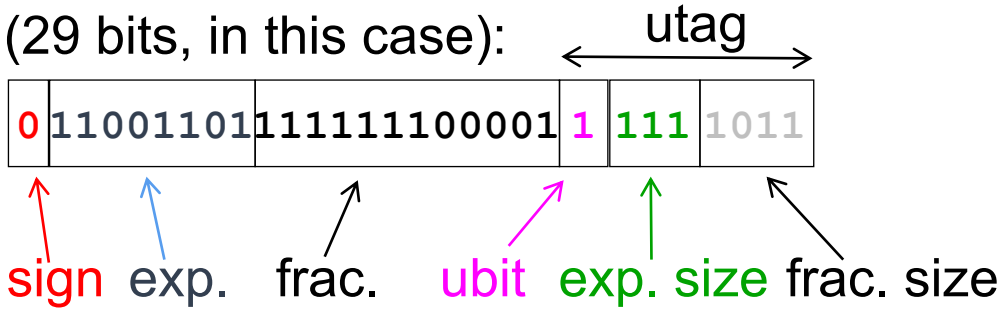
Flexible dynamic range Flexible precision No rounding, overflow, underflow No "negative zero" Fixes wasted NaN values Makes results bit-identical

Self-descriptive "utag" bits track and manage uncertainty, exponent size, and fraction size

BUT:

Variable storage size Adds indirection Many conditional tests

Unum
(29 bits, in this case):




```
The biggest
objection to unums
is likely to come
from the fact that
they are variable in
size, at least when
they are stored in
packed form.
```

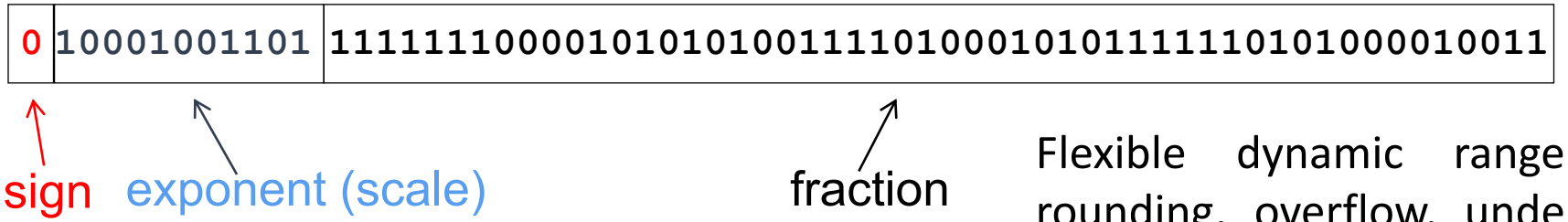
18-point Courier
(fixed width font)

The biggest objection to unums is likely to come from the fact that they are variable in size, at least when they are stored in packed form.

18-point Times
(variable width font)

Unum basics: avoid the price of "one size fits all"

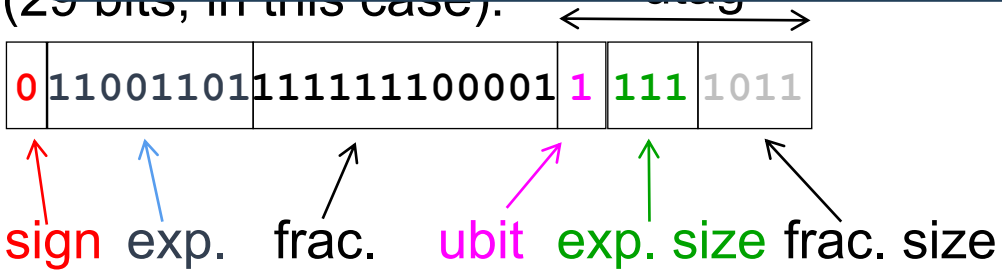
IEEE Standard Float (64 bits):



Flexible dynamic range Flexible precision No rounding, overflow, underflow No "negative zero"

Can unum ultimately become the future of computing?
We don't know yet…
Some known issues with unum:
- expensive in terms of time and power consumption
- Each computation in unum space is likely to change the bit length of the structure
- …

(29 bits, in this case):




```
from the fact that
they are variable in
size, at least when
they are stored in
packed form.
```

18-point Courier
(fixed width font)

in size, at least when they are stored in packed form.

18-point Times
(variable width font)

Summary

- Basic addition and counting
  - Challenges of fast addition
  - Carry-Lookahead adders

Summary

- Basic addition and counting
  - Challenges of fast addition
  - Carry-Lookahead adders
- Basic multiplication and division operation

Summary

- Basic addition and counting
  - Challenges of fast addition
  - Carry-Lookahead adders
- Basic multiplication and division operation
- Floating point representation
- Floating point operation
  - Floating point addition and multiplication
  - Hardware resource requirements

Summary

- Basic addition and counting
  - Challenges of fast addition
  - Carry-Lookahead adders
- Basic multiplication and division operation
- Floating point representation
- Floating point operation
  - Floating point addition and multiplication
  - Hardware resource requirements
- Advanced cases