Homework

- Reading: Chapter 8
- Exercises: Chapter 8, all
- Due Friday, 9/28

Computer Science Foundations

COS 140: Foundations of Computer Science

Adders

Fall 2018

Computer
Science
Foundations

What is an adder?

- An adder is a logic circuit that adds binary numbers
- Could add two 1-digit numbers or two n-bit numbers

Computer Science Foundations

Why Study Adders?

- Interesting example of a combinational circuit.
  - Circuit whose output relies solely on its inputs.
  - Perform an important function for the computer.
    - Addition is also basis for other arithmetic functions in the computer (subtraction, multiplication, etc.)
    - Would like the function done in hardware so it is done quickly.

Computer Science Foundations

How Do We Do Addition?

1. Write down numbers that will be added using symbols from 0 to 9.
2. Use arithmetic facts to add numbers in a column. If more than 9, carry the most significant digit to the next column.

How Do We Do Addition?

1. Write down numbers that will be added using symbols from 0 to 9.
2. Use arithmetic facts to add numbers in a column. If more than 9, carry the most significant digit to the next column.

$$
\begin{array}{cccc}
 & & {\scriptstyle 1} & \\
 & 3 & 5 & 6 \\
+ & 2 & 3 & 5 \\
\hline
 & 5 & 9 & 1
\end{array}
$$

Computer Science Foundations

Numbers and Digital Logic

- Symbols will correspond to the 0 or 1 that is the input or output of the circuit. So, have 2 symbols to work with, not 10.
- Create a binary system that is like our digital system.

Computer Science Foundations

A Closer Look at Our Digital System

- Have 10 digits: 0–9
- Have “places” for 1’s, 10’s, 100’s, 1000’s, 10,000’s, etc. that correspond to powers of 10.

A Closer Look at Our Digital System

- Have 10 digits: 0–9
- Have “places” for 1’s, 10’s, 100’s, 1000’s, 10,000’s, etc. that correspond to powers of 10.
  - $10^0 = 1; 10^1 = 10; 10^2 = 100; 10^3 = 1000; 10^4 = 10,000$
- To find the value of a number, add all the digits times their place values.

Computer Science Foundations

A Closer Look at Our Digital System

- Have 10 digits: 0–9
- Have “places” for 1’s, 10’s, 100’s, 1000’s, 10,000’s, etc. that correspond to powers of 10.
  - $10^0 = 1; 10^1 = 10; 10^2 = 100; 10^3 = 1000; 10^4 = 10,000$
- To find the value of a number, add all the digits times their place values.
  - $359 = 9 \times 1 + 5 \times 10 + 3 \times 100$

Computer Science Foundations

In a Binary System...

- Have 2 digits: 0 and 1
- Places correspond to powers of 2

Computer Science Foundations

In a Binary System...

- Have 2 digits: 0 and 1
- Places correspond to powers of 2:

2^0	1	2^4	16	2^8	256
2^1	2	2^5	32	2^9	512
2^2	4	2^6	64	2^{10}	1024
2^3	8	2^7	128	2^{11}	2048

Computer Science Foundations

In a Binary System...

- Have 2 digits: 0 and 1
- Places correspond to powers of 2:

2^0	1	2^4	16	2^8	256
2^1	2	2^5	32	2^9	512
2^2	4	2^6	64	2^{10}	1024
2^3	8	2^7	128	2^{11}	2048

- To find the value, add all the 1's and 0's times their place values.
  - $10110 = 0 \times 1 + 1 \times 2 + 1 \times 4 + 0 \times 8 + 1 \times 16 = 22$

Computer Science Foundations

From decimal to binary

Given: n, a decimal number

Computer Science Foundations

From decimal to binary

Given: n, a decimal number

1. First find the largest power of two less than n; let i be the exponent

From decimal to binary

Given: n, a decimal number

1. First find the largest power of two less than n; let i be the exponent
2. Write down a 1, and $n = n$ minus that power of two

Computer Science Foundations

From decimal to binary

Given: n, a decimal number

1. First find the largest power of two less than n; let i be the exponent
2. Write down a 1, and $n = n$ minus that power of two
3. Decrement i to work on next-lower binary digit; if $i = 0$, we're done

From decimal to binary

Given: n, a decimal number

1. First find the largest power of two less than n; let i be the exponent
2. Write down a 1, and $n = n$ minus that power of two
3. Decrement i to work on next-lower binary digit; if $i = 0$, we're done
4. If $2^i > n$, then there should be a 0 for that power of two; write that down, and go to 3

Computer Science Foundations

From decimal to binary

Given: n, a decimal number

1. First find the largest power of two less than n; let i be the exponent
2. Write down a 1, and $n = n$ minus that power of two
3. Decrement i to work on next-lower binary digit; if $i = 0$, we're done
4. If $2^i > n$, then there should be a 0 for that power of two; write that down, and go to 3
5. Else, if $2^i = n$, then write 0s for all the rest of the digits, and you're done

Computer Science Foundations

From decimal to binary

Given: n, a decimal number

1. First find the largest power of two less than n; let i be the exponent
2. Write down a 1, and $n = n$ minus that power of two
3. Decrement i to work on next-lower binary digit; if $i = 0$, we're done
4. If $2^i > n$, then there should be a 0 for that power of two; write that down, and go to 3
5. Else, if $2^i = n$, then write 0s for all the rest of the digits, and you're done
6. Otherwise ($2^i < n$), write a 1, since this power of 2 "fits" in n; $n = n - 2^i$, and go to 3

The algorithm

Algorithm Convert(d)
    Input: d, a decimal number
    Output: the binary version of d
    Let n be largest whole number such that $2^n \leq d$
    while $n \geq 0$ **do**
        if $d = 2^n$ **then**
            Output 1 followed by $n - 1$ 0s
            return
        else if $d < 2^n$ **then**
            Output 0
            $n = n - 1$
        else
            Output 1
            $d = d - 2^n$
            $n = n - 1$
        end if
    end while
End.

Computer Science Foundations

Example

359

Computer Science Foundations

Example

359 $\quad 2^8 < d < 2^9, \therefore n = 8 \Rightarrow$ 1

Computer Science Foundations

Example

	359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$ 1
-	256	Subtract $2^8 = 256, \; n = 7$
	103	

Computer Science Foundations

Example

	359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
-	256	Subtract $2^8 = 256, \; n = 7$	
	103	$2^7 > d; \; n = 6 \Rightarrow$	0

Computer Science Foundations

Example

	359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
-	256	Subtract $2^8 = 256,\ n = 7$	
	103	$2^7 > d;\ n = 6 \Rightarrow$	0
		$2^6 < d \Rightarrow$	1
-	64	Subtract $2^6 = 64,\ n = 5$	
	39		

Computer Science Foundations

Example

	359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
-	256	Subtract $2^8 = 256,\ n = 7$	
	103	$2^7 > d;\ n = 6 \Rightarrow$	0
		$2^6 < d \Rightarrow$	1
-	64	Subtract $2^6 = 64,\ n = 5$	
	39	$2^5 < d \Rightarrow$	1
-	32	Subtract $2^5 = 32,\ n = 4$	
	7		

Example

	359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
-	256	Subtract $2^8 = 256,\ n = 7$	
	103	$2^7 > d;\ n = 6 \Rightarrow$	0
		$2^6 < d \Rightarrow$	1
-	64	Subtract $2^6 = 64,\ n = 5$	
	39	$2^5 < d \Rightarrow$	1
-	32	Subtract $2^5 = 32,\ n = 4$	
	7	$2^4 > d;\ n = 3 \Rightarrow$	0

Computer Science Foundations

Example

	359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
-	256	Subtract $2^8 = 256, \; n = 7$	
	103	$2^7 > d; \; n = 6 \Rightarrow$	0
		$2^6 < d \Rightarrow$	1
-	64	Subtract $2^6 = 64, \; n = 5$	
	39	$2^5 < d \Rightarrow$	1
-	32	Subtract $2^5 = 32, \; n = 4$	
	7	$2^4 > d; \; n = 3 \Rightarrow$	0
		$2^3 > d; \; n = 2 \Rightarrow$	0

Computer Science Foundations

Adders

Binary Numbers

- A Closer Look at Our Digital System
- In a Binary System...
- From decimal to binary
- Algorithm

Adders

Multi-bit Adders

Example

359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
- 256	Subtract $2^8 = 256,\ n = 7$	
103	$2^7 > d;\ n = 6 \Rightarrow$	0
	$2^6 < d \Rightarrow$	1
- 64	Subtract $2^6 = 64,\ n = 5$	
39	$2^5 < d \Rightarrow$	1
- 32	Subtract $2^5 = 32,\ n = 4$	
7	$2^4 > d;\ n = 3 \Rightarrow$	0
	$2^3 > d;\ n = 2 \Rightarrow$	0
	$2^2 < d \Rightarrow$	1
- 4	Subtract $2^2 = 4,\ n = 1$	
3		

Computer Science Foundations

Example

359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
- 256	Subtract $2^8 = 256,\ n = 7$	
103	$2^7 > d;\ n = 6 \Rightarrow$	0
	$2^6 < d \Rightarrow$	1
- 64	Subtract $2^6 = 64,\ n = 5$	
39	$2^5 < d \Rightarrow$	1
- 32	Subtract $2^5 = 32,\ n = 4$	
7	$2^4 > d;\ n = 3 \Rightarrow$	0
	$2^3 > d;\ n = 2 \Rightarrow$	0
	$2^2 < d \Rightarrow$	1
- 4	Subtract $2^2 = 4,\ n = 1$	
3	$2^1 < d \Rightarrow$	1
- 2	Subtract $2^1 = 2,\ n = 0$	
1		

Computer Science Foundations

Example

359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
- 256	Subtract $2^8 = 256, \; n = 7$	
103	$2^7 > d; \; n = 6 \Rightarrow$	0
	$2^6 < d \Rightarrow$	1
- 64	Subtract $2^6 = 64, \; n = 5$	
39	$2^5 < d \Rightarrow$	1
- 32	Subtract $2^5 = 32, \; n = 4$	
7	$2^4 > d; \; n = 3 \Rightarrow$	0
	$2^3 > d; \; n = 2 \Rightarrow$	0
	$2^2 < d \Rightarrow$	1
- 4	Subtract $2^2 = 4, \; n = 1$	
3	$2^1 < d \Rightarrow$	1
- 2	Subtract $2^1 = 2, \; n = 0$	
1	$2^0 = d, \Rightarrow$	1

Example

	359	$2^8 < d < 2^9, \therefore n = 8 \Rightarrow$	1
-	256	Subtract $2^8 = 256,\ n = 7$	
	103	$2^7 > d;\ n = 6 \Rightarrow$	0
		$2^6 < d \Rightarrow$	1
-	64	Subtract $2^6 = 64,\ n = 5$	
	39	$2^5 < d \Rightarrow$	1
-	32	Subtract $2^5 = 32,\ n = 4$	
	7	$2^4 > d;\ n = 3 \Rightarrow$	0
		$2^3 > d;\ n = 2 \Rightarrow$	0
		$2^2 < d \Rightarrow$	1
-	4	Subtract $2^2 = 4,\ n = 1$	
	3	$2^1 < d \Rightarrow$	1
-	2	Subtract $2^1 = 2,\ n = 0$	
	1	$2^0 = d, \Rightarrow$	1

So $359_{10} = 101100111_2$

Computer Science Foundations

There are only 10 kinds of people in this world. Those that understand binary and those that don't.

– graduate student's T-shirt

Use Arithmetic Facts to Add Numbers

- Addition results from applying facts about arithmetic to numbers
- For the computer to use arithmetic facts, we need to construct a circuit.
- So: start with a truth table.
- Construct a truth table for all of the inputs, including the possible carry.

Truth Table for Addition

Carry-in	A	B	Carry-out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Computer Science Foundations

Circuit from truth table

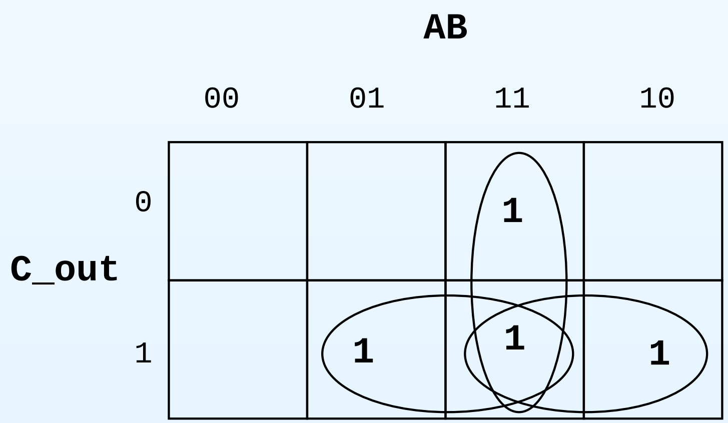
- Can we find a circuit for this? A minimal circuit?

Computer Science Foundations

Circuit from truth table

- Can we find a circuit for this? A minimal circuit?
- Karnaugh map for carry out:

C_in	A	B	C_out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



AB + BC + AC

Circuit from truth table

- Can we find a circuit for this? A minimal circuit?
- Karnaugh map for sum out:

C_in	A	B	C_out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

AB

Sum	00	01	11	10
0		**1**		**1**
1	**1**		**1**	

~A~BC + ~AB~C +
ABC + A~BC

Computer
Science
Foundations

A better idea

- So minimization using Karnaugh maps, algebraic substitution – not so good!
- Can we do better?

A better idea

- So minimization using Karnaugh maps, algebraic substitution – not so good!
- Can we do better?
- Maybe – inspect the truth table

Computer Science Foundations

A better idea

- So minimization using Karnaugh maps, algebraic substitution – not so good!
- Can we do better?
- Maybe – inspect the truth table
- Things are simplified when we look at just A and B as inputs:

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

Computer Science Foundations

A better idea

- So minimization using Karnaugh maps, algebraic substitution – not so good!
- Can we do better?
- Maybe – inspect the truth table
- Things are simplified when we look at just A and B as inputs:

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

- Sum and carry – both correspond to basic operations/gates

Computer Science Foundations

A better idea

- So minimization using Karnaugh maps, algebraic substitution – not so good!
- Can we do better?
- Maybe – inspect the truth table
- Things are simplified when we look at just A and B as inputs:

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

- Sum and carry – both correspond to basic operations/gates
- Sum = $A \oplus B$

Computer Science Foundations

A better idea

- So minimization using Karnaugh maps, algebraic substitution – not so good!
- Can we do better?
- Maybe – inspect the truth table
- Things are simplified when we look at just A and B as inputs:

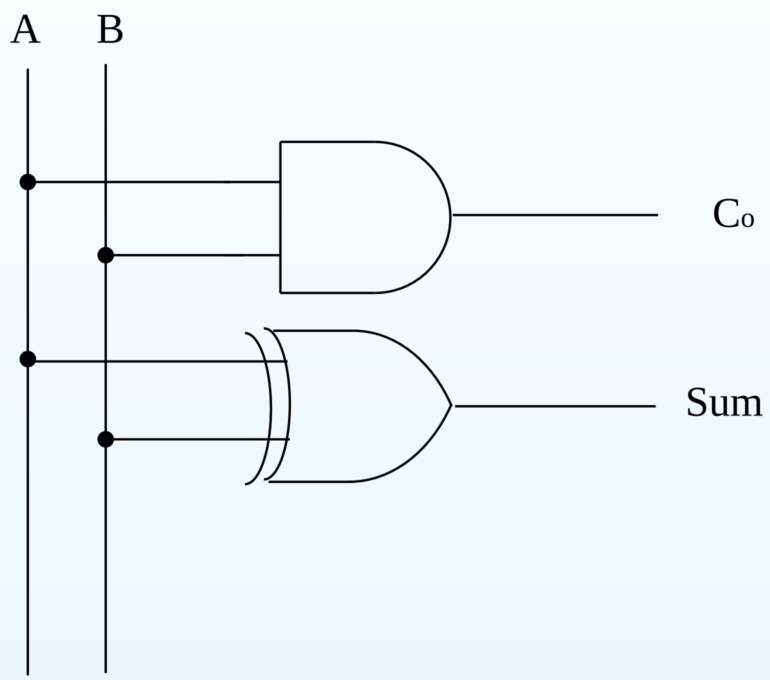
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Sum and carry – both correspond to basic operations/gates
- Sum = $A \oplus B$
- Carry = AB

Computer Science Foundations

Half-Adder

We can create a very simple circuit to add A and B.

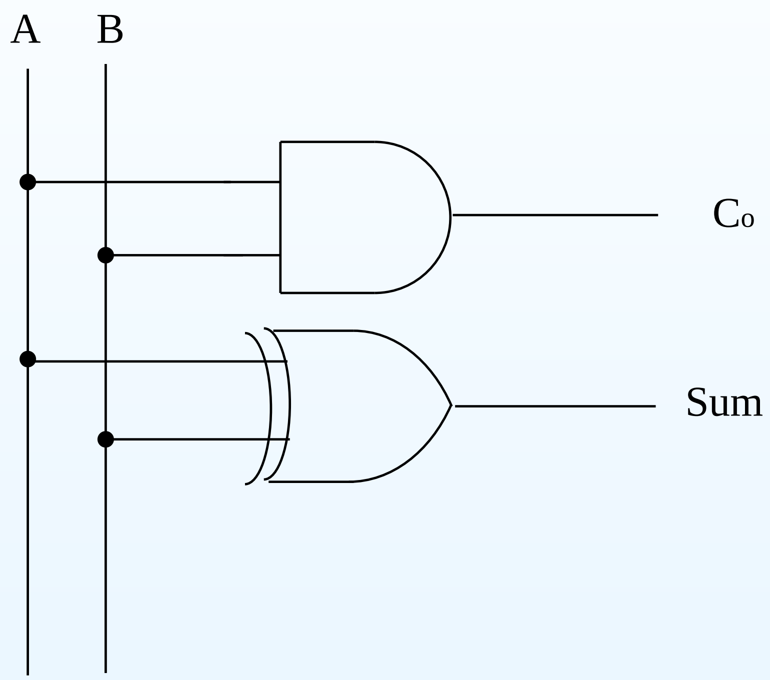


Half-adder because only does half the job.

Computer Science Foundations

Half-Adder

We can create a very simple circuit to add A and B.



Half-adder because only does half the job.

We need a *full adder* that adds $A + B + C_{in}$

Full Adder

- A, B, $C_{in} \longrightarrow S$ (sum), C (carry out)
- Can we use a half-adder + additional logic get outputs?

Computer Science Foundations

Full Adder

- $A, B, C_{in} \longrightarrow S$ (sum), C (carry out)
- Can we use a half-adder + additional logic get outputs?
- Half adder: $A, B \longrightarrow S_h, C_h$

Full Adder

- $A, B, C_{in} \longrightarrow S$ (sum), C (carry out)
- Can we use a half-adder + additional logic get outputs?
- Half adder: $A, B \longrightarrow S_h, C_h$
- Generating S (sum):

Computer Science Foundations

Full Adder

- $A, B, C_{in} \longrightarrow S$ (sum), C (carry out)
- Can we use a half-adder + additional logic get outputs?
- Half adder: $A, B \longrightarrow S_h, C_h$
- Generating S (sum):
  - $S = A + B + C_{in}$

Computer Science Foundations

Full Adder

- $A, B, C_{in} \longrightarrow S$ (sum), C (carry out)
- Can we use a half-adder + additional logic get outputs?
- Half adder: $A, B \longrightarrow S_h, C_h$
- Generating S (sum):
  - $S = A + B + C_{in} = (A + B) + C_{in}$

Computer Science Foundations

Full Adder

- $A, B, C_{in} \longrightarrow S$ (sum), C (carry out)
- Can we use a half-adder + additional logic get outputs?
- Half adder: $A, B \longrightarrow S_h, C_h$
- Generating S (sum):
  - $S = A + B + C_{in} = (A + B) + C_{in} = S_h + C_{in}$

Computer Science Foundations

Full Adder

- $A, B, C_{in} \longrightarrow S$ (sum), C (carry out)
- Can we use a half-adder + additional logic get outputs?
- Half adder: $A, B \longrightarrow S_h, C_h$
- Generating S (sum):
  - $S = A + B + C_{in} = (A + B) + C_{in} = S_h + C_{in}$
  - Use another half-adder: $S_h, C_{in} \longrightarrow S_{h2} = S$

Full Adder

- $A, B, C_{in} \longrightarrow S$ (sum), C (carry out)
- Can we use a half-adder + additional logic get outputs?
- Half adder: $A, B \longrightarrow S_h, C_h$
- Generating S (sum):
  - $S = A + B + C_{in} = (A + B) + C_{in} = S_h + C_{in}$
  - Use another half-adder: $S_h, C_{in} \longrightarrow S_{h2} = S$

Computer Science Foundations

Full Adder

- When is C (carry out) $= 1$?

Computer Science Foundations

Full Adder

- When is C (carry out) $= 1$? When $A + B + C_{in} \geq 10_2$

Computer Science Foundations

Full Adder

- When is C (carry out) $= 1$? When $A + B + C_{in} \geq 10_2$
  - Case 1: $A + B = 10_2$

Computer Science Foundations

Full Adder

- When is C (carry out) $= 1$? When $A + B + C_{in} \geq 10_2$
  - Case 1: $A + B = 10_2$
    - Doesn't matter what C_{in} is: $C = 1$
    - In this case: $C_h = 1$

Computer Science Foundations

Full Adder

- When is C (carry out) $= 1$? When $A + B + C_{in} \geq 10_2$
  - Case 1: $A + B = 10_2$
    - Doesn't matter what C_{in} is: $C = 1$
    - In this case: $C_h = 1$
  - Case 2: $A + B = 1$ and $C_{in} = 1$

Full Adder

- When is C (carry out) $= 1$? When $A + B + C_{in} \geq 10_2$
  - Case 1: $A + B = 10_2$
    - Doesn't matter what C_{in} is: $C = 1$
    - In this case: $C_h = 1$
  - Case 2: $A + B = 1$ and $C_{in} = 1$
    - This means that $S_h = 1$, $C_{in} = 1$
    - In this case, carry out of second half-adder $C_{h2} = 1$

Computer Science Foundations

Full Adder

- When is C (carry out) $= 1$? When $A + B + C_{in} \geq 10_2$
  - Case 1: $A + B = 10_2$
    - Doesn't matter what C_{in} is: $C = 1$
    - In this case: $C_h = 1$
  - Case 2: $A + B = 1$ and $C_{in} = 1$
    - This means that $S_h = 1$, $C_{in} = 1$
    - In this case, carry out of second half-adder $C_{h2} = 1$
  - So $C = 1$ when either either or both half-adder carries is 1

Computer Science Foundations

Full Adder

- When is C (carry out) $= 1$? When $A + B + C_{\mathit{in}} \geq 10_2$
  - Case 1: $A + B = 10_2$
    - Doesn't matter what C_{in} is: $C = 1$
    - In this case: $C_h = 1$
  - Case 2: $A + B = 1$ and $C_{\mathit{in}} = 1$
    - This means that $S_h = 1$, $C_{\mathit{in}} = 1$
    - In this case, carry out of second half-adder $C_{h2} = 1$
  - So $C = 1$ when either either or both half-adder carries is 1
  - $\therefore C = C_h \vee C_{h2}$

Computer Science Foundations

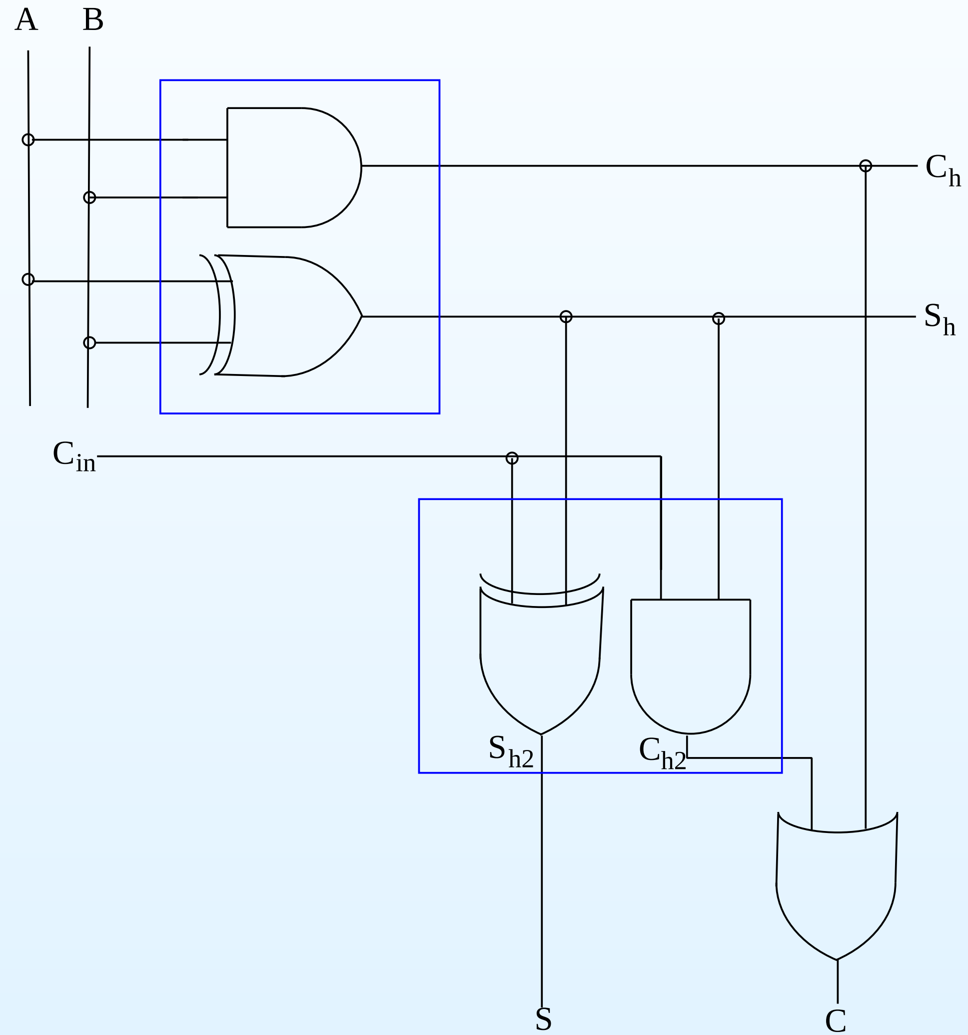
Full Adder

We can create a full adder by putting two half adders together as described above.

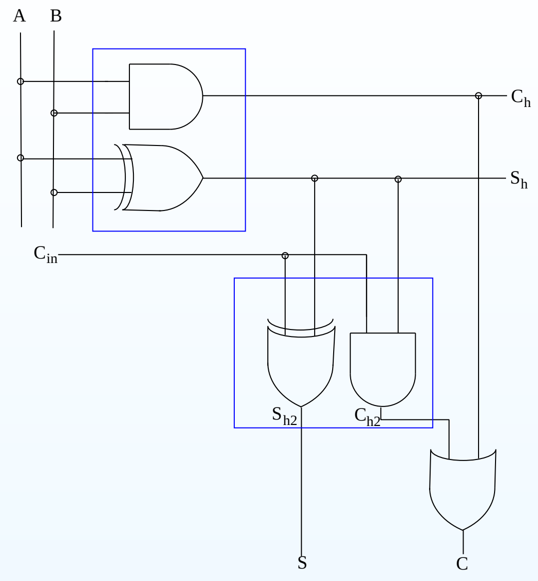
Computer Science Foundations

Full Adder

We can create a full adder by putting two half adders together as described above.

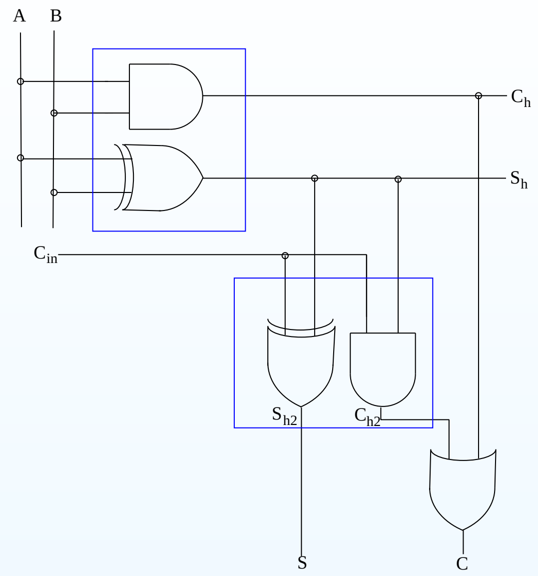


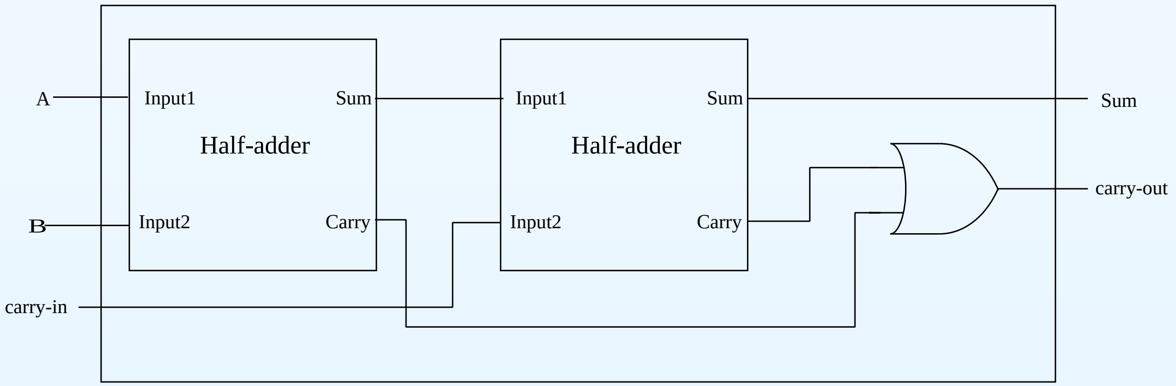
Full Adder



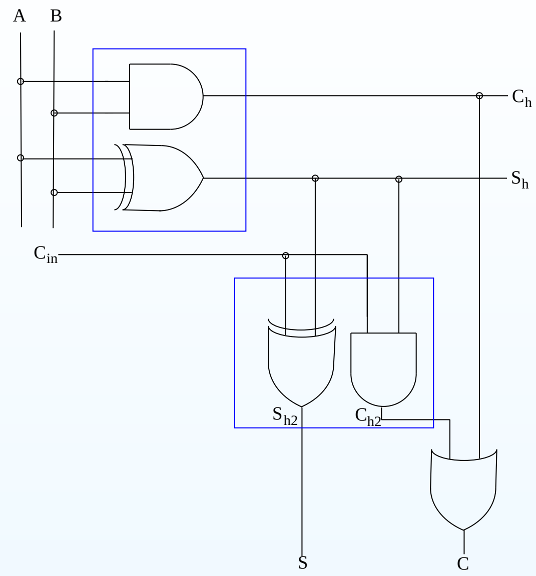
Computer Science Foundations

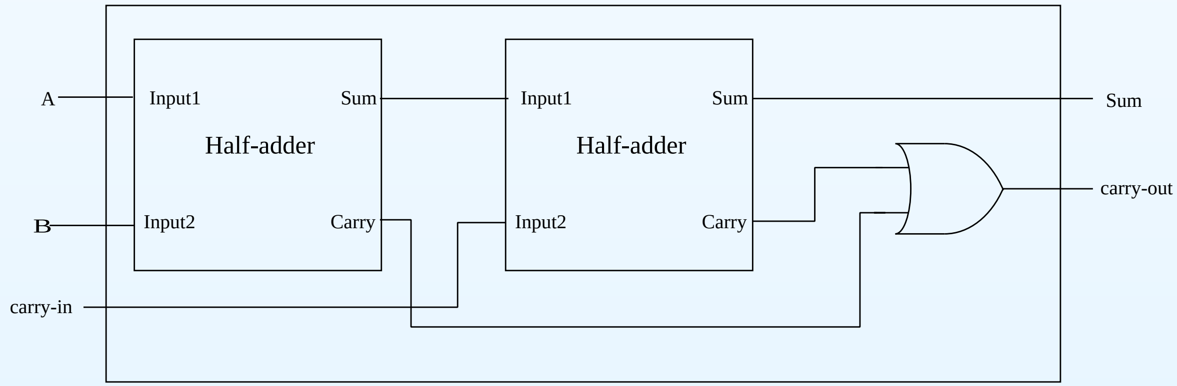
Full Adder

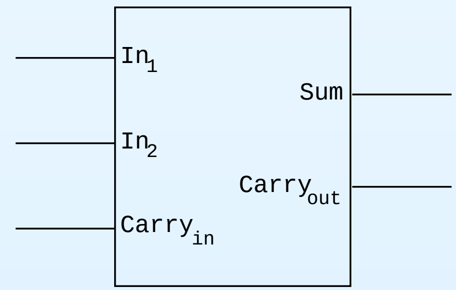

A
B
C_h
S_h
C_{in}
S_{h2}
C_{h2}
S
C


A
Input1
Sum
Half-adder
B
Input2
Carry
Input1
Sum
Half-adder
Input2
Carry
Sum
carry-out
carry-in

Full Adder







Computer Science Foundations

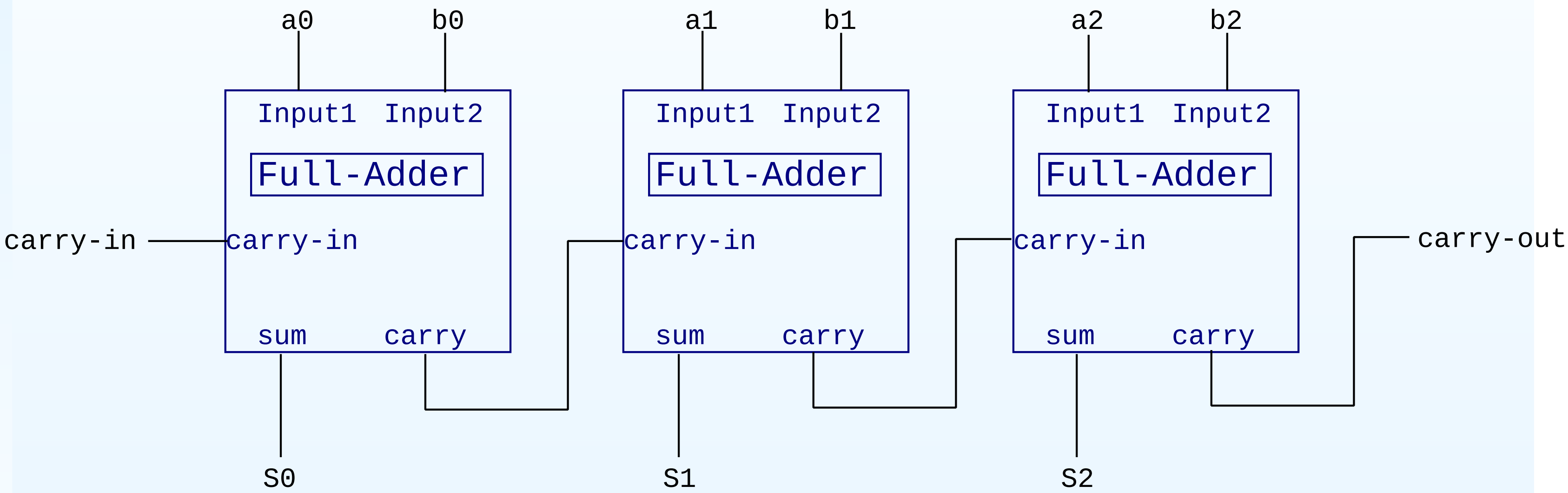
Creating Multi-Bit Adders

- Add multi-digit binary numbers using a full-adder for each bit.
- Problem: How to compute the carry-in for adder n?

Computer Science Foundations

Computing Carries: Ripple Carry

- Hook up required number of full adders.
- As carry is calculated, passed to next bit.



Computing Carries: Carry Lookahead

- First calculate what carry bits would be, based on previous bits
- Another way to specify carry-out is: carry-out of any full-adder (C_n) is true if:
  - carry-out of the first half-adder ($A_n B_n$) is true, or
  - either one of the inputs and the carry-in is true: $(A_n + B_n)C_{n-1}$
- So: $C_n = A_n B_n + (A_n + B_n)C_{n-1}$ – a *recurrence relation*

Computer Science Foundations

Computing Carries: Carry Lookahead

- We can calculate any carry using the recurrence relation:

$$C_n = A_n B_n + (A_n + B_n) C_{n-1}$$

  - $C_0 = A_0 B_0$, assuming no carry-in to low-order bit

Computing Carries: Carry Lookahead

- We can calculate any carry using the recurrence relation:

$$C_n = A_n B_n + (A_n + B_n) C_{n-1}$$

  - $C_0 = A_0 B_0$, assuming no carry-in to low-order bit
  - $C_1 = A_1 B_1 + (A_1 + B_1) C_0$

Computer Science Foundations

Computing Carries: Carry Lookahead

- We can calculate any carry using the recurrence relation:

$$C_n = A_n B_n + (A_n + B_n) C_{n-1}$$

  - $C_0 = A_0 B_0$, assuming no carry-in to low-order bit
  - $C_1 = A_1 B_1 + (A_1 + B_1) C_0 \Rightarrow$
    $C_1 = A_1 B_1 + (A_1 + B_1) A_0 B_0$

Computer Science Foundations

Computing Carries: Carry Lookahead

- We can calculate any carry using the recurrence relation:

$$C_n = A_nB_n + (A_n + B_n)C_{n-1}$$

  - $C_0 = A_0B_0$, assuming no carry-in to low-order bit
  - $C_1 = A_1B_1 + (A_1 + B_1)C_0 \Rightarrow$
    $C_1 = A_1B_1 + (A_1 + B_1)A_0B_0$
  - $C_2 = A_2B_2 + (A_2 + B_2)C_1$

Computer Science Foundations

Computing Carries: Carry Lookahead

- We can calculate any carry using the recurrence relation:

$$C_n = A_n B_n + (A_n + B_n) C_{n-1}$$

  - $C_0 = A_0 B_0$, assuming no carry-in to low-order bit
  - $C_1 = A_1 B_1 + (A_1 + B_1) C_0 \Rightarrow$
    $C_1 = A_1 B_1 + (A_1 + B_1) A_0 B_0$
  - $C_2 = A_2 B_2 + (A_2 + B_2) C_1 \Rightarrow$
    $C_2 = A_2 B_2 + (A_2 + B_2)(A_1 B_1 + (A_1 + B_1) A_0 B_0)$
  - ...

Computer Science Foundations

Computing Carries: Carry Lookahead

- Alternative could be done using $\oplus$:
  - For carry-out of the first full-adder (C_0), it's the carry-out of the full-adder's first half-adder OR'd with the carry-out of the second:
    $C_0 = C_{0_h} \vee C_{0_f} \Rightarrow$
    $C_0 = A_0 B_0 \vee S_0 C_{in} \Rightarrow$
    $C_0 = A_0 B_0 \vee (A_0 \oplus B_0) C_{in}$
  - C_1 is computable based on C_0 in the same way:
    $C_1 = A_1 B_1 \vee (A_1 \oplus B_1) C_0 \Rightarrow$
    $C_1 = A_1 B_1 \vee (A_1 \oplus B_1)(A_0 B_0 \vee (A_0 \oplus B_0) C_{in})$
  - Can generalize to n bits
- But better to keep with ANDs and ORs

Computer Science Foundations

Trade-offs for Types of Carry Propagation

- Ripple carry has larger delay.
  - Ripple carry has delay as more significant binary digits wait for results from less significant digits.
  - Carry lookahead uses a sum of products to get result for each carry, so only a two gate delay (have an AND layer and an OR layer).
- Complexity of circuit.
  - Ripple carry requires only connecting carry-out to next carry-in.
  - Number of AND gates and number of inputs to OR gate is on the order of the number of digits for carry lookahead (i.e., $O(n)$, where n is the number of digits)

Computer Science Foundations

Combining Ripple Carry and Carry Lookahead

- Minimize complexity of carry lookahead by only using it on a small number of bits in a group.

Computer Science Foundations

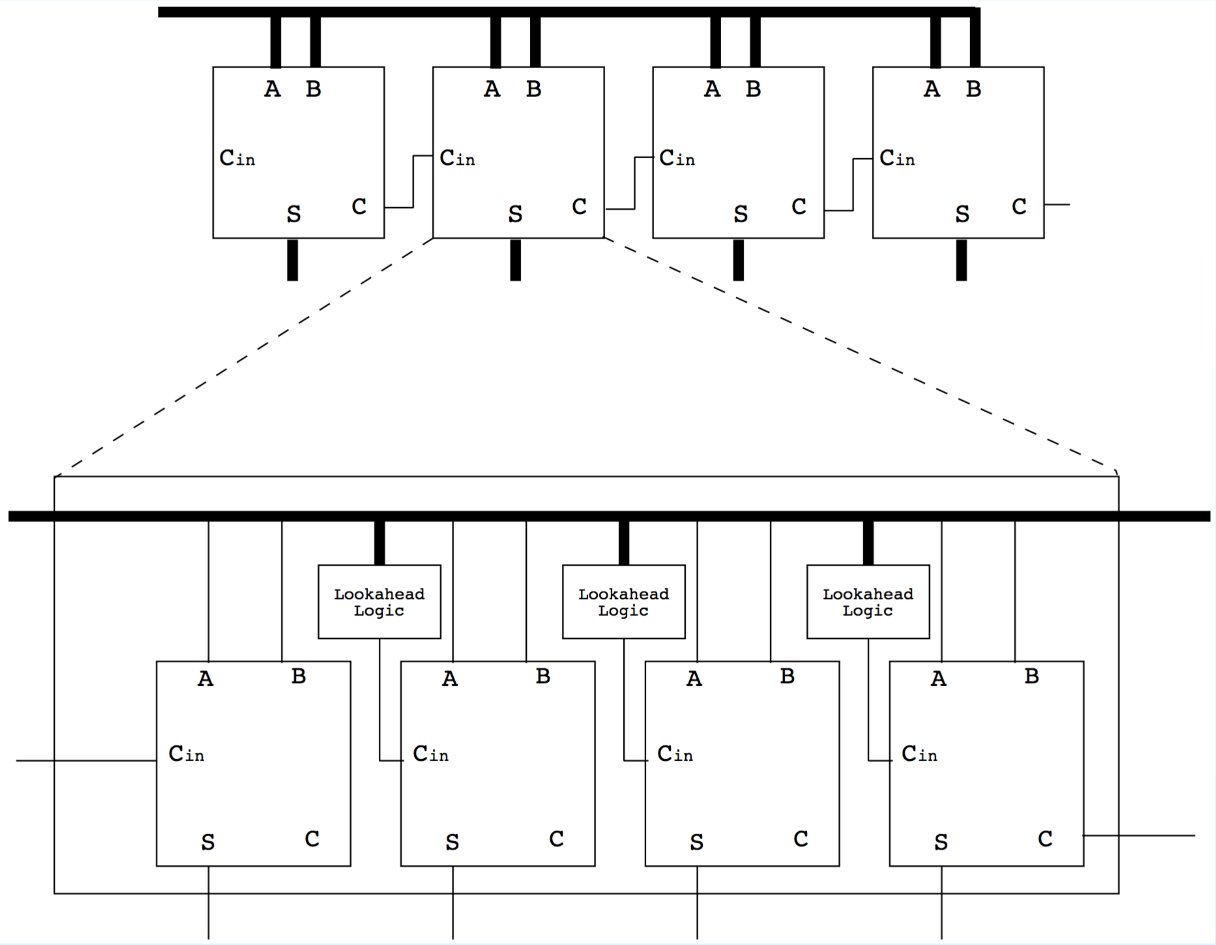
Combining Ripple Carry and Carry Lookahead

- Minimize complexity of carry lookahead by only using it on a small number of bits in a group.
- Put groups together with a ripple carry.

Computer Science Foundations

Combining Ripple Carry and Carry Lookahead

- Minimize complexity of carry lookahead by only using it on a small number of bits in a group.
- Put groups together with a ripple carry.



Computer Science Foundations

What have we learned?

- Combinational circuits are important for computers

Computer Science Foundations

What have we learned?

- Combinational circuits are important for computers
- Sometimes direct minimization of circuits via SOP may not be best. . .

Computer Science Foundations

What have we learned?

- Combinational circuits are important for computers
- Sometimes direct minimization of circuits via SOP may not be best. . .
- . . . need to think outside the mechanistic box!

What have we learned?

- Combinational circuits are important for computers
- Sometimes direct minimization of circuits via SOP may not be best. . .
- . . . need to think outside the mechanistic box!
- Easy way of combining circuits may not be best way

What have we learned?

- Combinational circuits are important for computers
- Sometimes direct minimization of circuits via SOP may not be best. . .
- . . . need to think outside the mechanistic box!
- Easy way of combining circuits may not be best way
  - Sometimes best way requires a lot of work to find

What have we learned?

- Combinational circuits are important for computers
- Sometimes direct minimization of circuits via SOP may not be best. . .
- . . . need to think outside the mechanistic box!
- Easy way of combining circuits may not be best way
  - Sometimes best way requires a lot of work to find
  - Sometimes “best” may not have a single meaning. . .

What have we learned?

- Combinational circuits are important for computers
- Sometimes direct minimization of circuits via SOP may not be best. . .
- . . . need to think outside the mechanistic box!
- Easy way of combining circuits may not be best way
  - Sometimes best way requires a lot of work to find
  - Sometimes “best” may not have a single meaning. . .
  - . . . may have to trade off (e.g.) time for circuit complexity

What have we learned?

- Combinational circuits are important for computers
- Sometimes direct minimization of circuits via SOP may not be best. . .
- . . . need to think outside the mechanistic box!
- Easy way of combining circuits may not be best way
  - Sometimes best way requires a lot of work to find
  - Sometimes "best" may not have a single meaning. . .
  - . . . may have to trade off (e.g.) time for circuit complexity
- Sometimes what looks hard to implement (carry lookahead) may not be (2 layers of gates)

Computer Science Foundations