COM1006 Devices and Networks (Autumn)
COM1090 Computer Architectures

Lecture #7

Implementing Arithmetic

Dr Dirk Sudholt
Department of Computer Science
University of Sheffield

d.sudholt@sheffield.ac.uk
http://staffwww.dcs.shef.ac.uk/~dirk/campus_only/com1006/

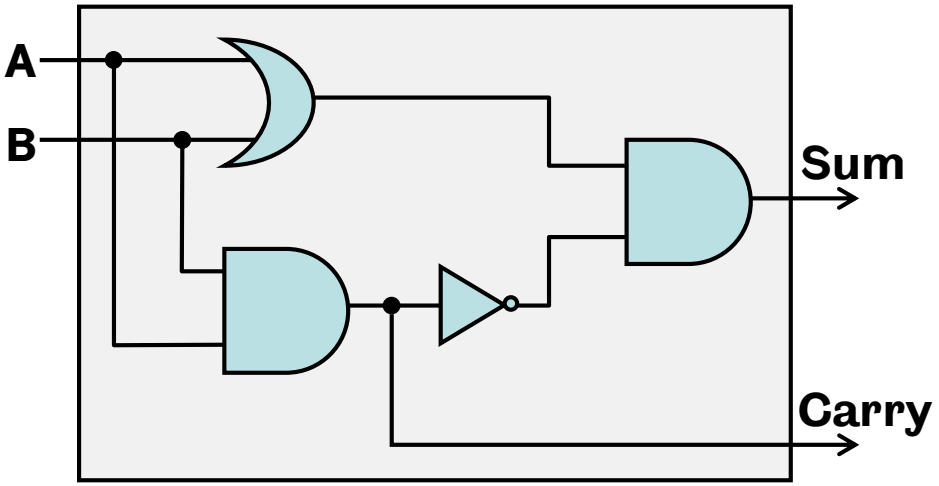
Aims of this lecture

- How arithmetic operations on *n*-bit words are implemented in logic.
- To explain how binary numbers are added in logic circuits.
- To show how unsigned binary numbers are multiplied.
- To describe how these circuits can be optimized with more clever designs.

The half adder

- The most primitive circuit for adding binary numbers is the **half adder**, which adds two bits to give a sum and carry.

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



- Note that the sum is the EOR of the inputs, and the carry is the AND of the inputs.
- EOR can be written as $(A+B)\cdot\overline{AB}$. This implementation uses fewer gates, and we get AB for free!

The full adder

- The half adder is of limited use because we have to take account of carry bits when adding *m*-bit numbers.
- We use a series of full adders, which add two bits A and B plus a carry-in C_{in} from the previous stage to give a sum S and carry out C_{out}.
- This is essentially a 3-bit adder.

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

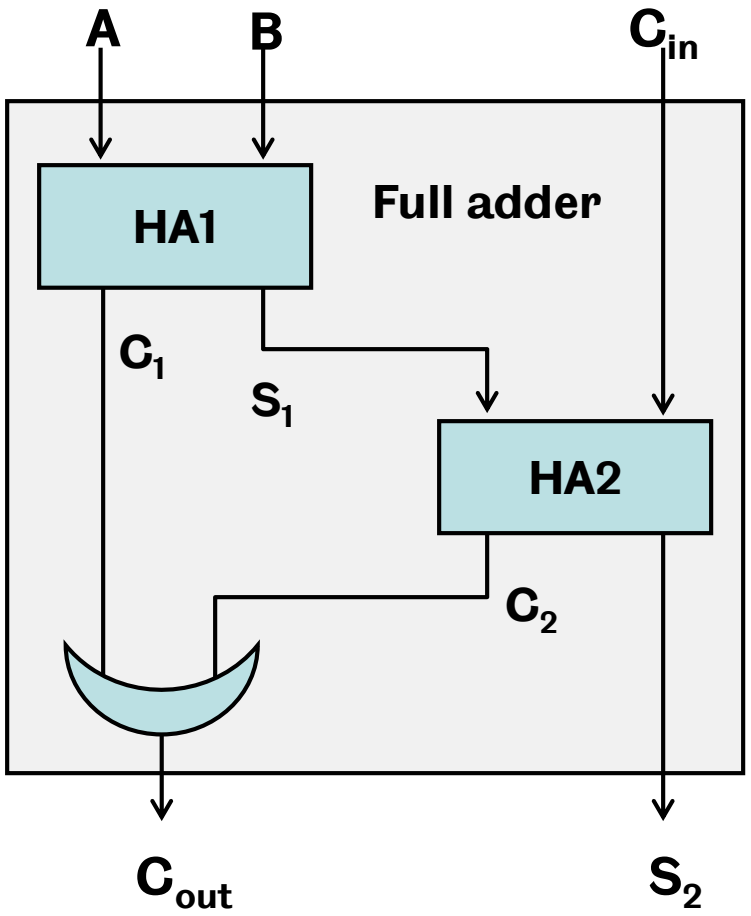
Truth table for full adder

Implementing a full adder

- Implement a full adder from two half adders.
- Conceptually, add the two bits of A and B in HA1 and then add the carry-in to this result in HA2:

  $(A+B)+C_{in}$

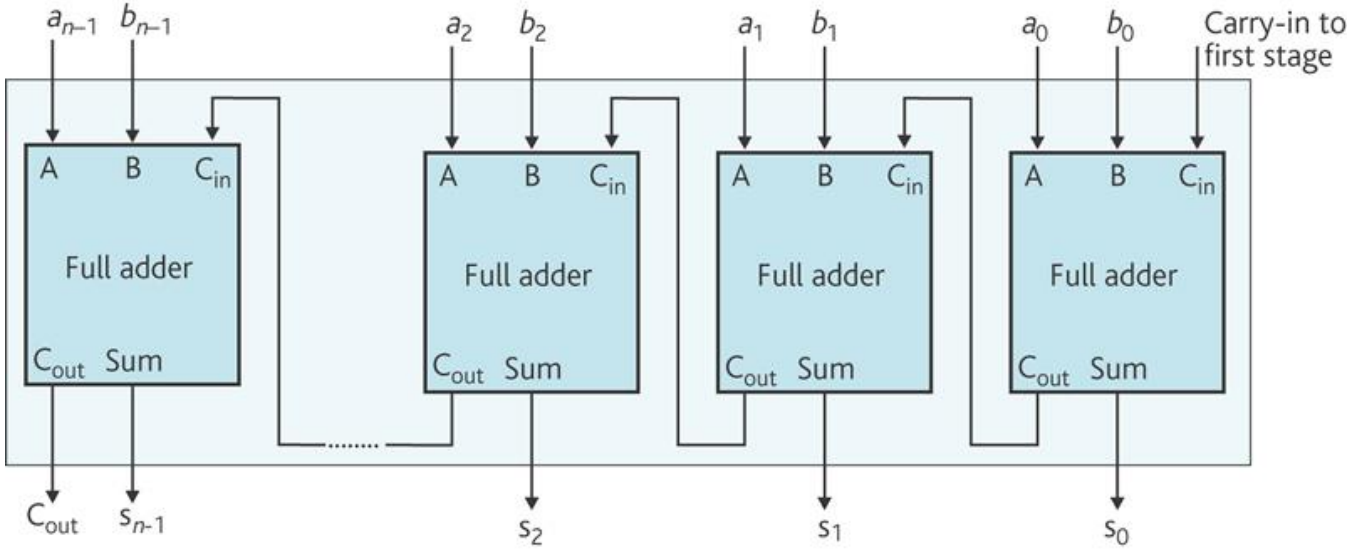
- Carry out obtained by ORing the carries from both half adders.



Adding words

- The full adder still isn't sufficient, because normally we need to add *n*-bit numbers rather than single bits.
- This is achieved using an array of *n* full adders, organised so that the carry-out from each full adder provides the carry-in to the next stage on the left.
- Just like adding words with pen and paper – bit by bit.
- The carry ripples through circuit – a so-called **ripple-carry adder**.

Ripple-carry adder

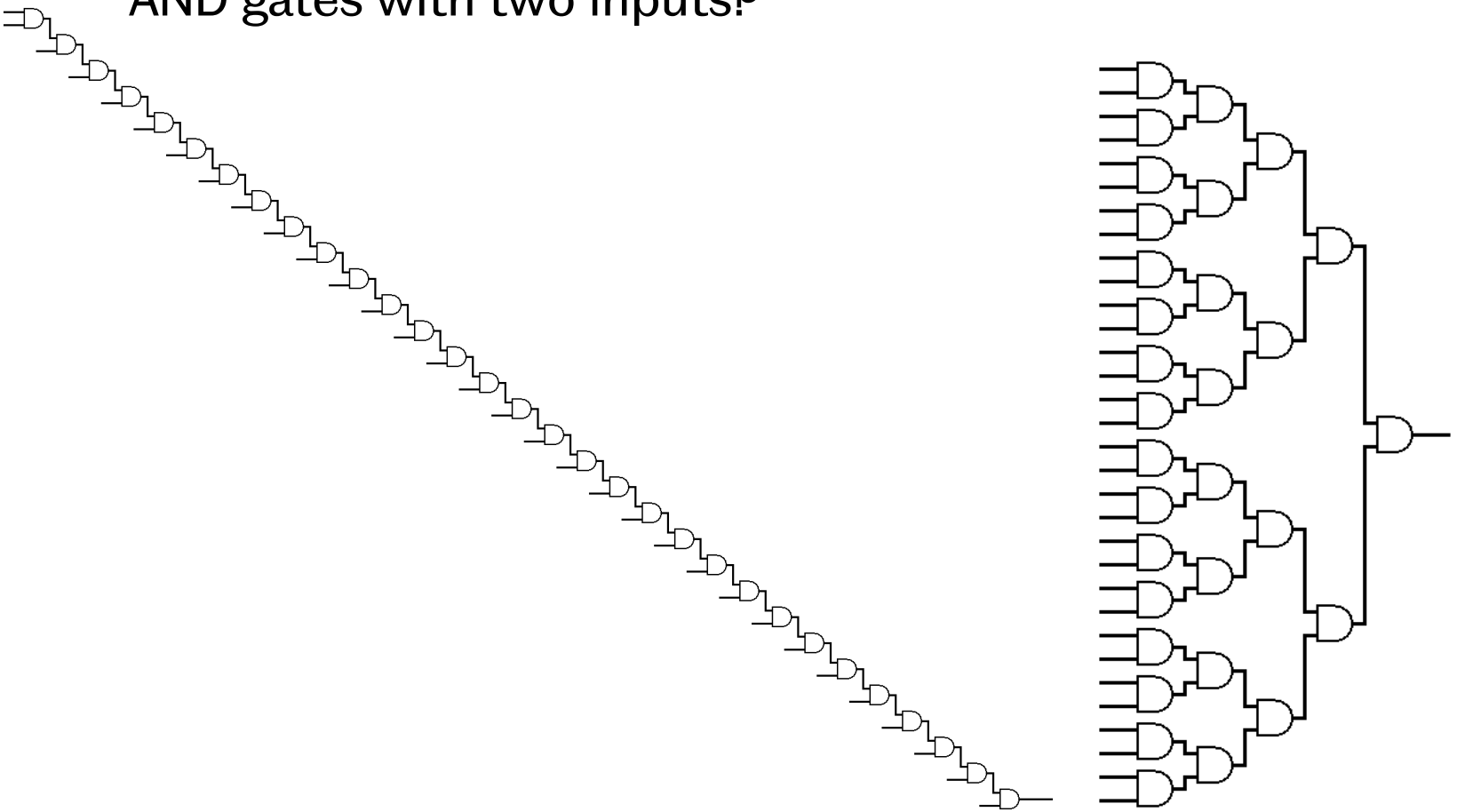


- If there is a carry out from the last stage then the result has gone out of range. Typically this carry-out is stored as part of the computer's condition code register.

❓ **What use is the carry-in to the first stage? Couldn't we replace this stage with a half adder?**

Brain teaser

- How would you compute the AND of 32 bits if you only had AND gates with two inputs?

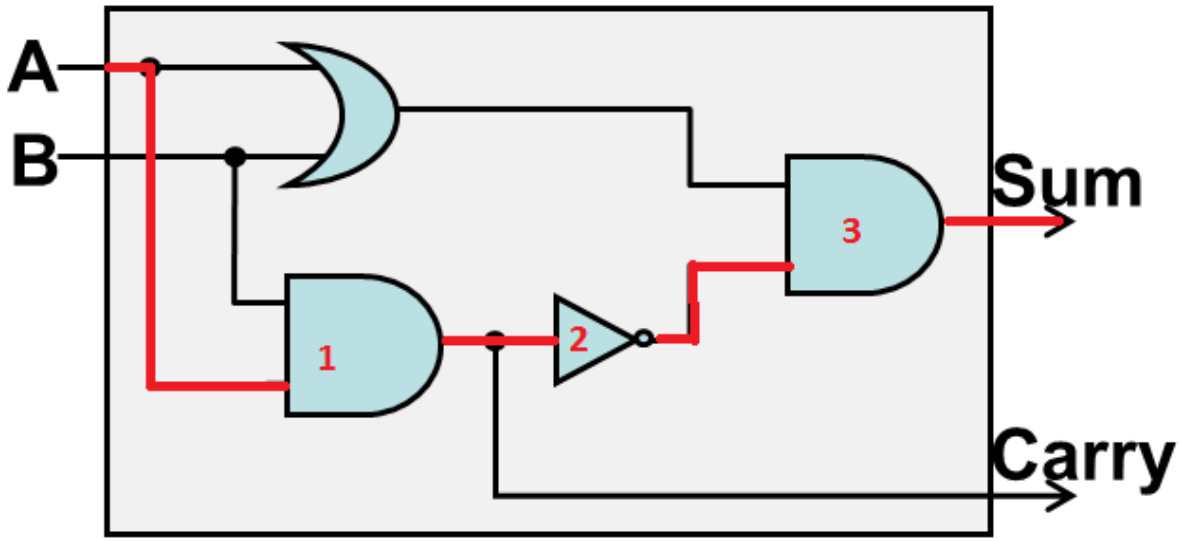


Speed does matter

- Each gate needs ~1ns to adapt to changing input signals.
- **Linear design**:
  signals pass through 31 gates → 31ns
- **Parallel design**:
  signals pass through 5 gates → 5ns (6x faster!)
- Do this 1 million times: nanoseconds become milliseconds.
- Doing things **in parallel** in circuits is a good idea!

Performance of circuits

- Performance can be measured by the **circuit depth**: the largest number of gates on any path from an input to an output.
- Example: single half-adder has depth 3.



Improving on the ripple-carry adder

- Drawback of the ripple-carry adder: high circuit depth (input needs to traverse many gates to compute the output).

❓ Can we do things in parallel instead?

- In order to compute the result of $a_i + b_i$, we need to know the carry-in for that position, c_i.
- The ripple-carry adder computes c_i while computing **all sums** $a_0 + b_0$, $a_1 + b_1$, ..., $a_{i-1} + b_{i-1}$.
- Is there a faster circuit that specialises in **just computing the carry c_i** without these sums?

Guess the carry bit!

A

a_3	0	1	1
b_3	0	1	0
p			

B

a_3	1	0	1
b_3	1	0	0
p			

C

a_3	1	1	0
b_3	0	1	0
p			

D

a_3	1	1	1
b_3	0	0	0
p			

E

a_3	0	1	1
b_3	1	0	1
p			

Carry look-ahead adder

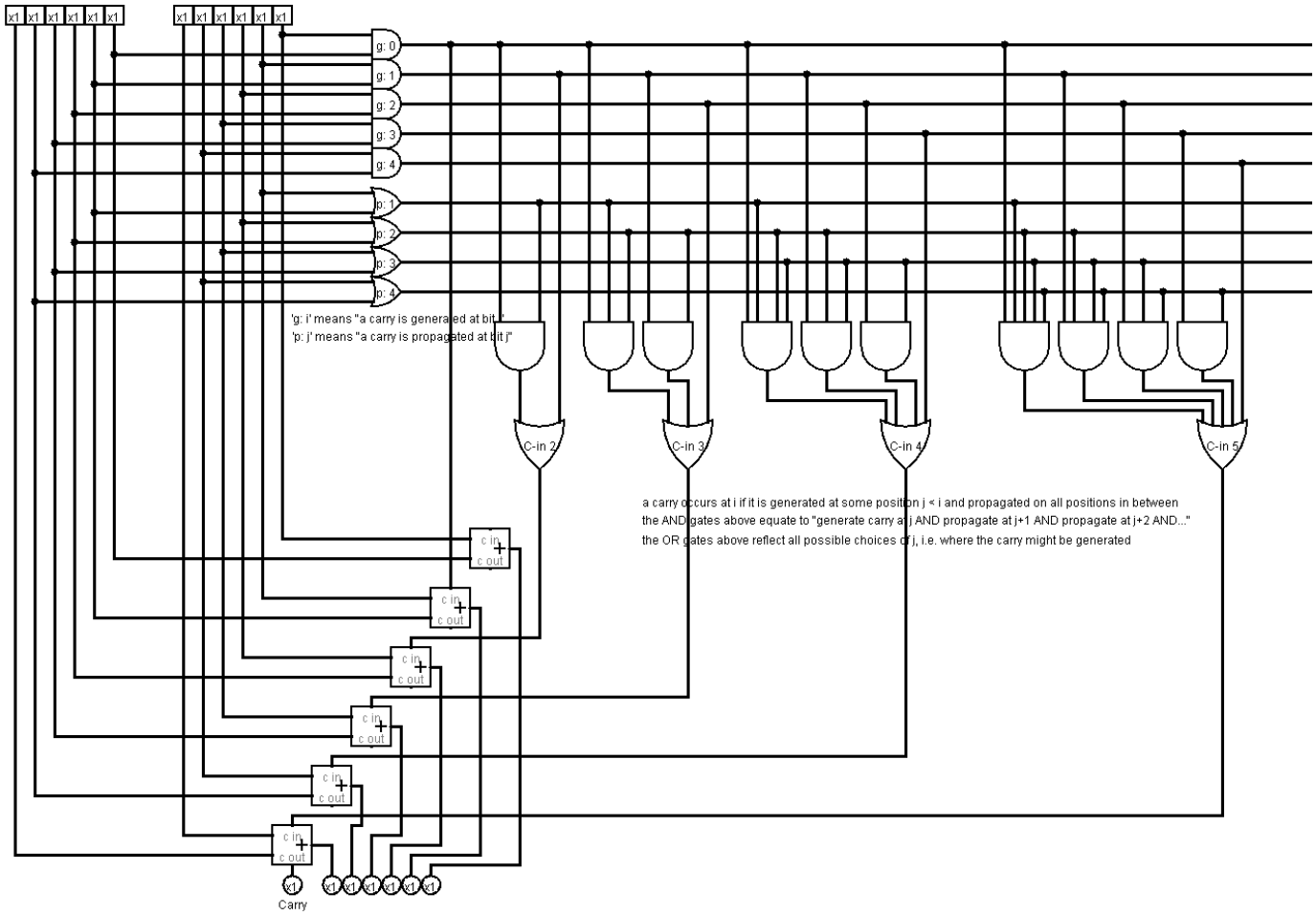
- **Idea:** each bit position computes its carry-in **in parallel**, in a specialised circuit.
- A carry-in occurs at position *i* if and only if a carry is **generated** at some lower position *j* < *i* and **propagated** on all positions between *j* and *i* (if there are any).
- All possibilities for generating a carry-in c_3:

bits in a	a_3	1	x	x	a_3	0	1	x	a_3	0	1	1
bits in b	b_3	1	x	x	b_3	1	1	x	b_3	1	0	1
carries	1	x	x		1	1	x		1	1	1	
		g				p	g			p	p	g

- Formula for carry-in c_3:

$$c_3 = a_2\, b_2 \;+\; (a_2 + b_2)\, a_1\, b_1 \;+\; (a_2 + b_2)\, (a_1 + b_1)\, a_0\, b_0$$

Here's one I made earlier...



Carry look-ahead adder

- Logisim circuits of 6-bit ripple-carry & carry look-ahead adders:
  https://staffwww.dcs.shef.ac.uk/people/D.Sudholt/campus_only/misc/ripple-carry-adder-6.circ
  https://staffwww.dcs.shef.ac.uk/people/D.Sudholt/campus_only/misc/carry-look-ahead-adder-6.circ
  (you might have to remove the file ending ".xml“ after saving the file)
- Look-ahead circuits can be implemented with circuit depth 3!
  (if we had unlimited numbers of inputs, otherwise use parallel design from brain teaser)
- One clever idea, huge effect!
- Smaller depth comes at the expense of size.
- More sophisticated designs provide a better trade-off between depth and size.

Multiplication

- The multiplication of two bits is equivalent to logical AND.
- Multiplication of unsigned binary numbers involves the multiplication of an *n*-bit number by a **single digit**, followed by **adding all partial products**.
- In binary, digits are just 0 and 1 → multiply by 0 or 1 using AND.
- **Example**: 10 x 13, multiplier = 1101_2, multiplicand = 1010_2

1010	multiplicand $a_3a_2a_1a_0$
x 1101	multiplier $b_3b_2b_1b_0$
0000$a_3a_2a_1a_0$	0000$a_3a_2a_1a_0$ AND $b_0b_0b_0b_0b_0b_0b_0b_0$
000$a_3a_2a_1a_0$0	000$a_3a_2a_1a_0$0 AND $b_1b_1b_1b_1b_1b_1b_1b_1$
00$a_3a_2a_1a_0$00	00$a_3a_2a_1a_0$00 AND $b_2b_2b_2b_2b_2b_2b_2b_2$
0$a_3a_2a_1a_0$000	0$a_3a_2a_1a_0$000 AND $b_3b_3b_3b_3b_3b_3b_3b_3$
10000010	add together *n* partial products, gives *2n*-bit number

Implementing multiplication

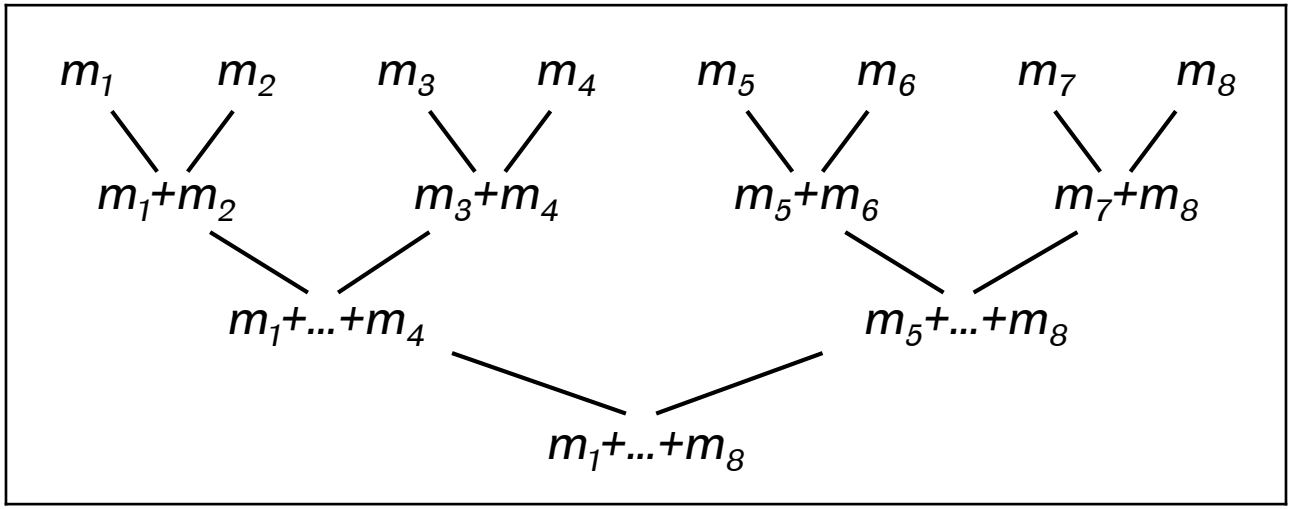
- The multiplication of two n-bit numbers gives a $2n$-bit product.
- Partial products like "$00a_3a_2a_1a_0 00$ AND $b_2b_2b_2b_2b_2b_2b_2b_2$" can be formed **wiring** the inputs $a_3a_2a_1a_0$ appropriately and using a single layer of AND gates (depth 1).
- Left to do: add n words simultaneously with adders operating on $2n$-bit words:

$$m_1+m_2+m_3+m_4+m_5+m_6+m_7+m_8$$

Adding *n* numbers

- Slow: $((((((m_1+m_2)+m_3)+m_4)+m_5)+m_6)+m_7)+m_8$ (deep!)
- Faster: compute sums **in parallel**:

$$(((m_1+m_2) + (m_3+m_4)) + ((m_5+m_6) + (m_7+m_8)))$$



- In circuits it's always good to do things **in parallel**!

Summary

- Seen how addition and multiplication can be implemented in circuits.
- The speed of a circuit can be described by its **circuit depth**.
- Performing operations in parallel in circuits can significantly reduce circuit depth.
- The carry look-ahead adder computes carry-ins in parallel, leading to better depth at expense of size.
- Multiplication in binary boils down to adding n words, each of length $2n$.