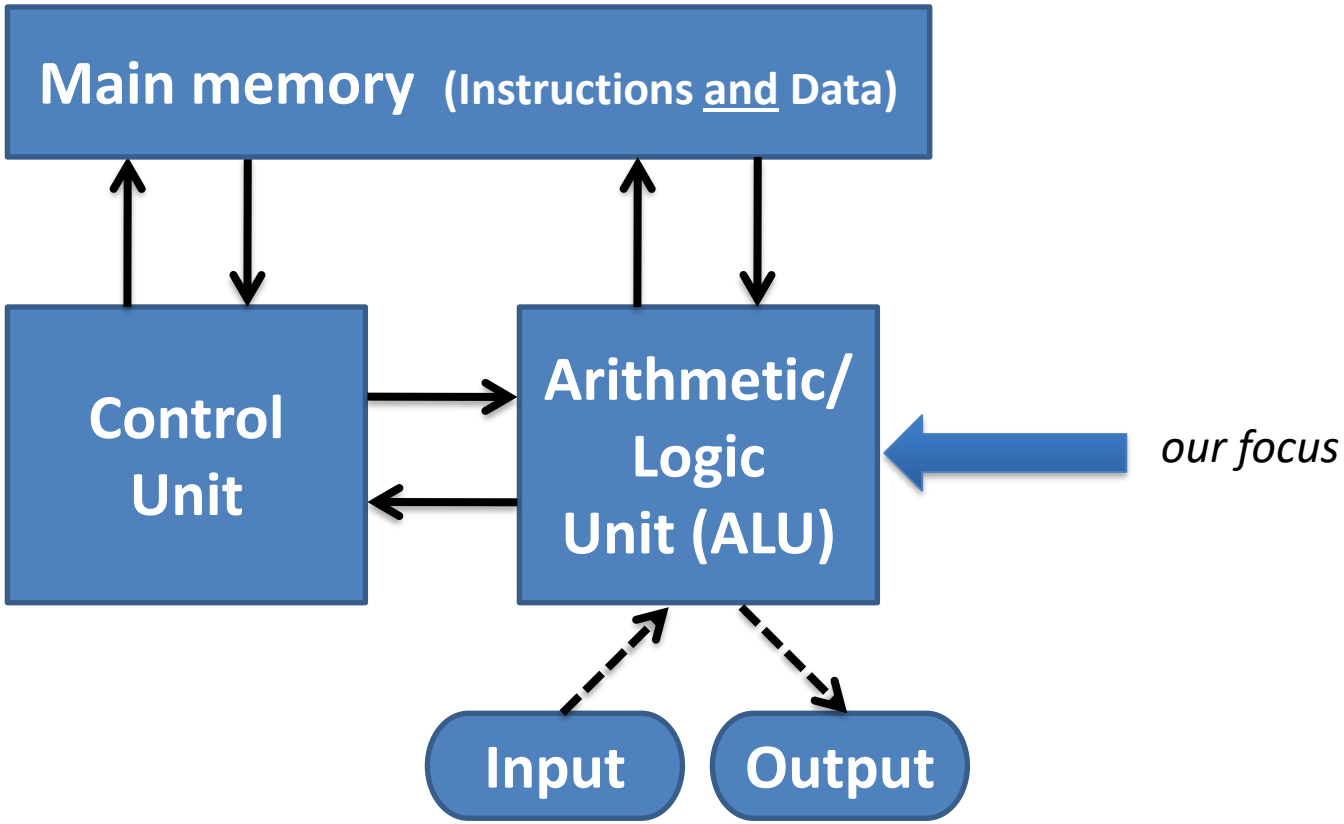
Lancaster University

SCC131: Digital Systems

Topic 6: Building the ALU

Reminder of the von Neumann architecture



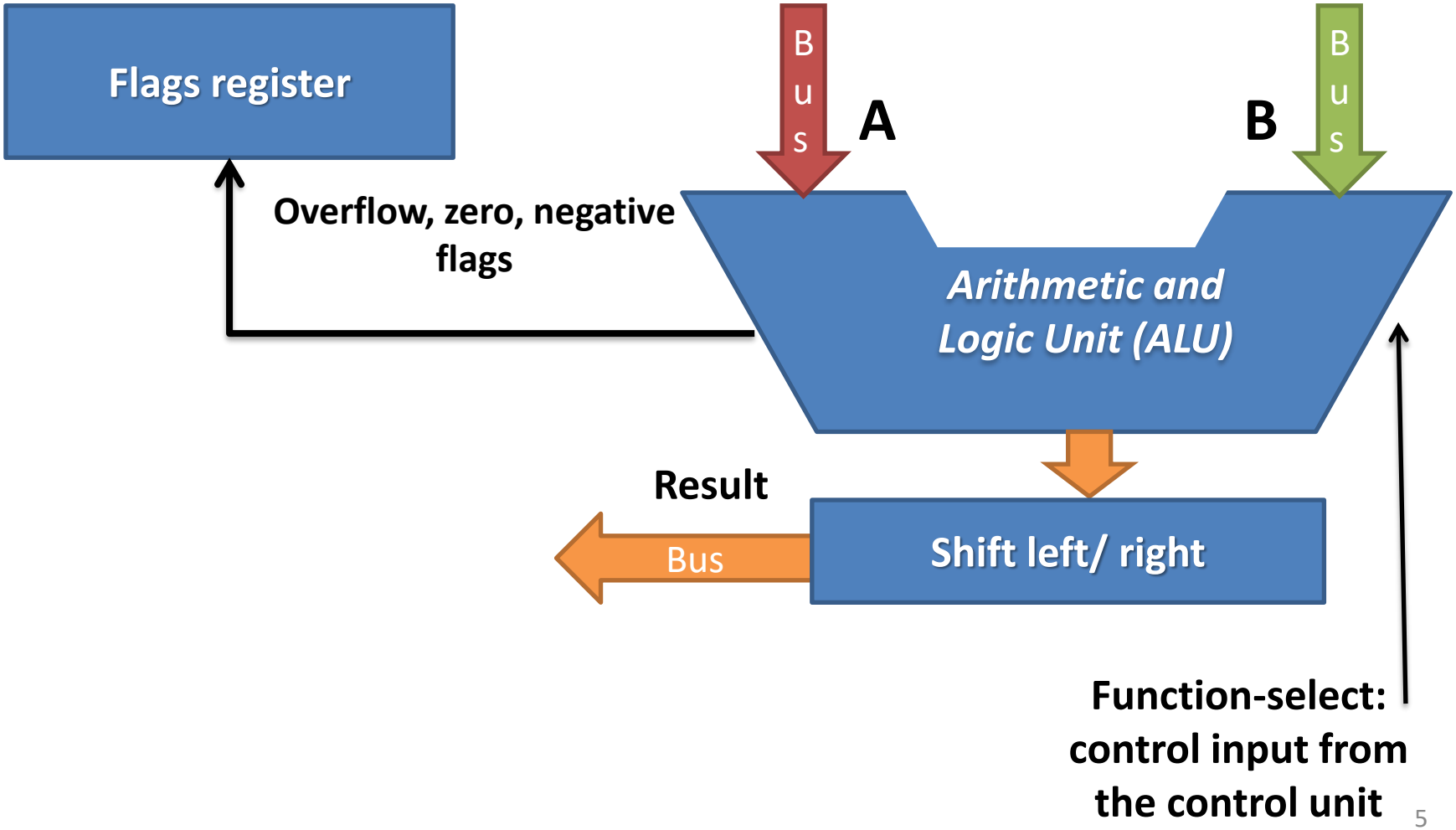
What is the role of the ALU? (1)

- Implements arithmetic and logic operations:
  - add, subtract, multiply, divide, shift of integers, …
    - (perhaps also floating point operations; or may use a separate floating point unit (FPU))
  - and, or, not, xor, …
  - comparisons: $<$, $\leq$, $=$, $\neq$, $\geq$, $>$

What is the role of the ALU? (2)

- Provides the following facilities:
  - registers: working storage for operands and results
  - status flags: typically, **overflow, zero** and **negative**
    - Overflow flag tells us if the previous arithmetic instruction resulted in overflow
    - Zero and negative are also set by arithmetic operations, and are typically used to determine the outcome of branch decisions

The ALU in outline



Adding two binary bits: the “half adder”

HALF ADDER

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

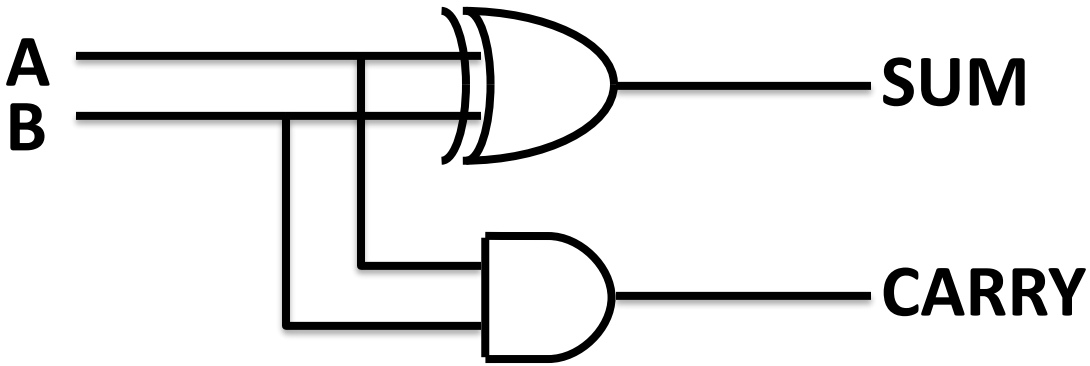
SUM

A	B	S
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

CARRY

A	B	C
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

Implementing a half adder



SUM = A ⊕ B

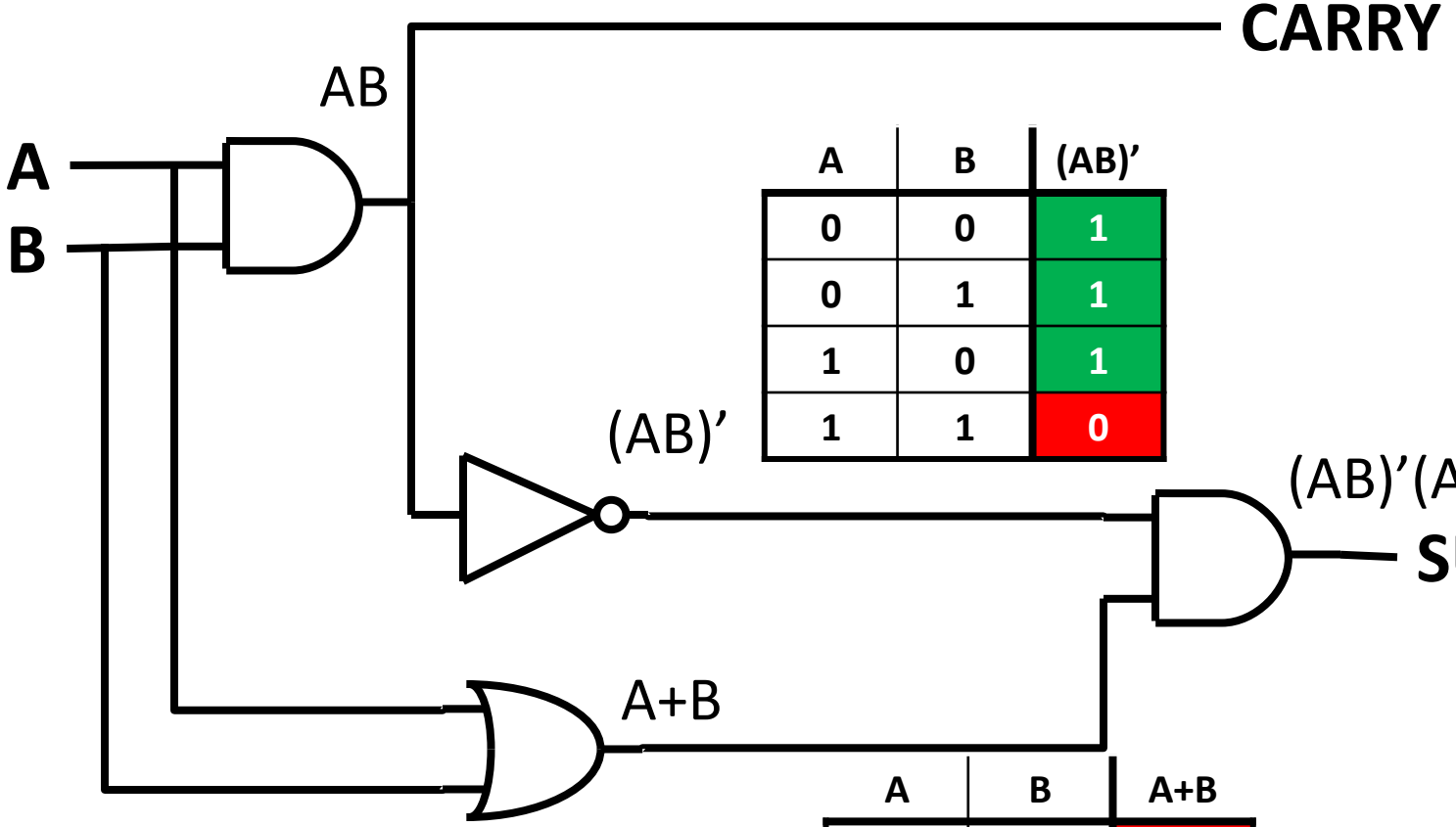
A	B	Q
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

CARRY = AB

A	B	Q
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

We can break the ⊕ down into something easier to build...

Half adder with expanded ⊕



A	B	(AB)'
0	0	1
0	1	1
1	0	1
1	1	0

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

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

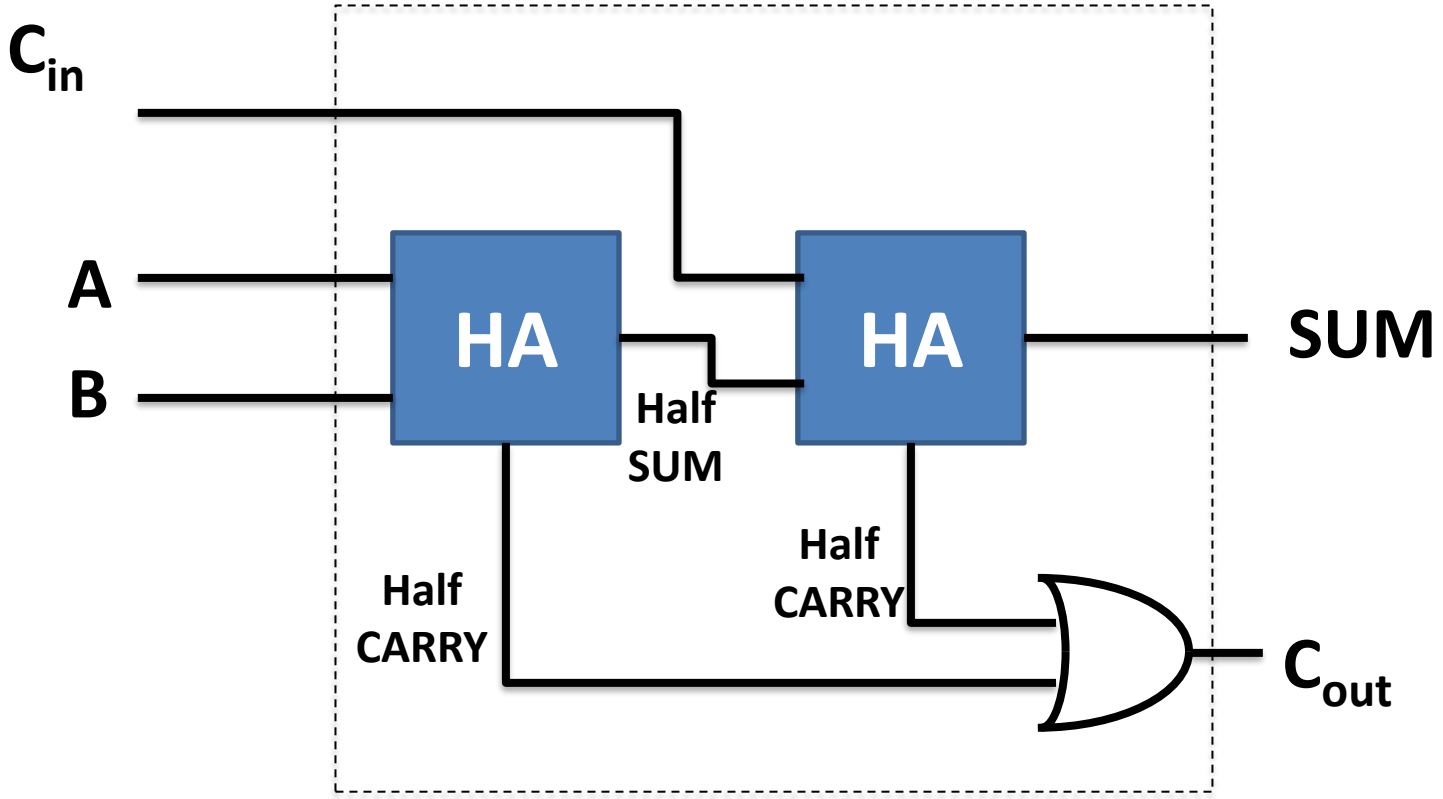
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

(*Can you redesign this to use just NAND gates*?)

Adding the 3rd bit

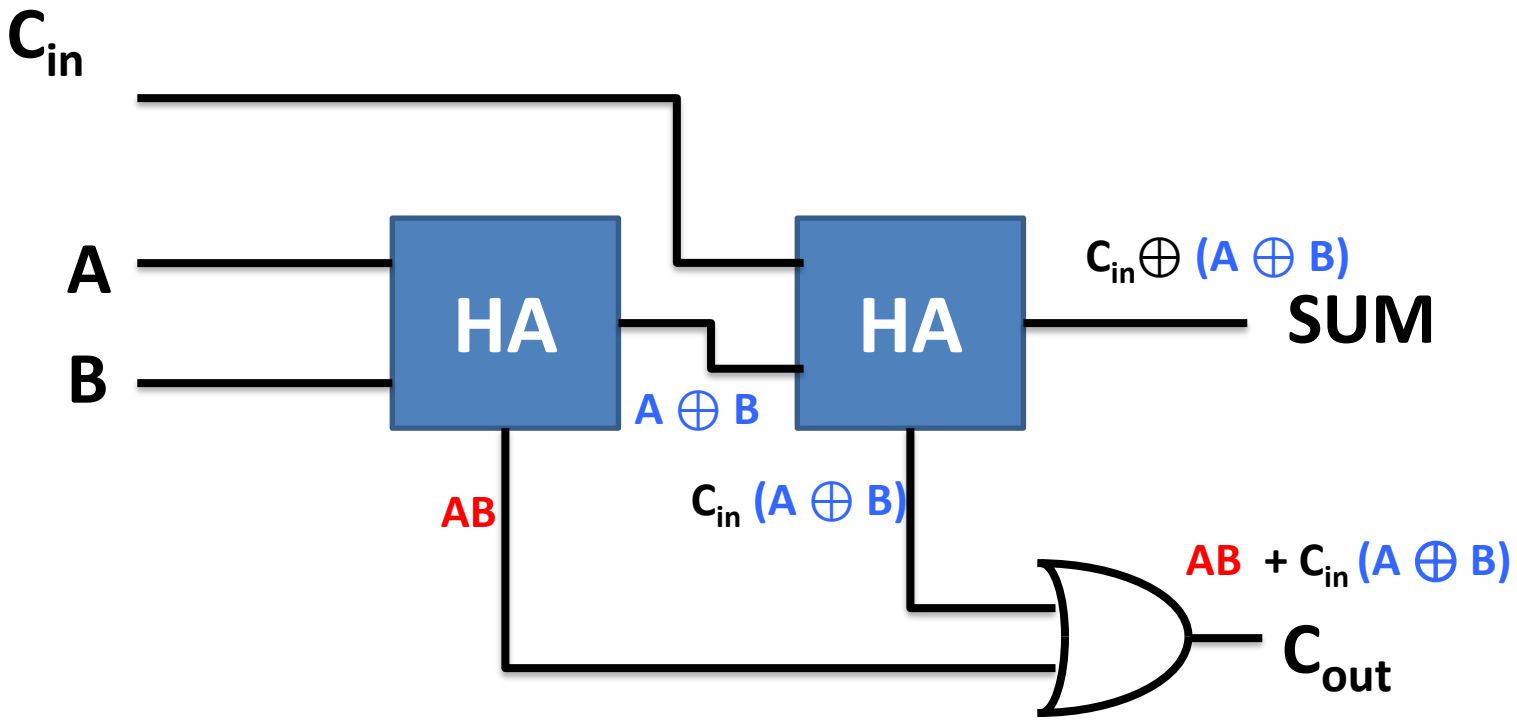
- A half adder adds only *two* bits
- But when adding mult-digit numbers consisting of multiple bits, we need also to consider CARRYs
  - When CARRYs are considered, we need to be able to add **3** bits: two input bits plus a possible CARRY from the previous stage
  - Where the CARRY and input bits are all 1s, this results in a maximum output of 3 (i.e., 1 + 1 + 1 = 11)
- To add these 3 bits, we can use *two* half adders; this is called a **full adder**...

Full adder



By combining these we can build adders that can handle any length of numbers…

Why does this work?



$SUM = C_{in} \oplus (A \oplus B)$ $\quad C_{out} = AB + C_{in}\ (A \oplus B)$

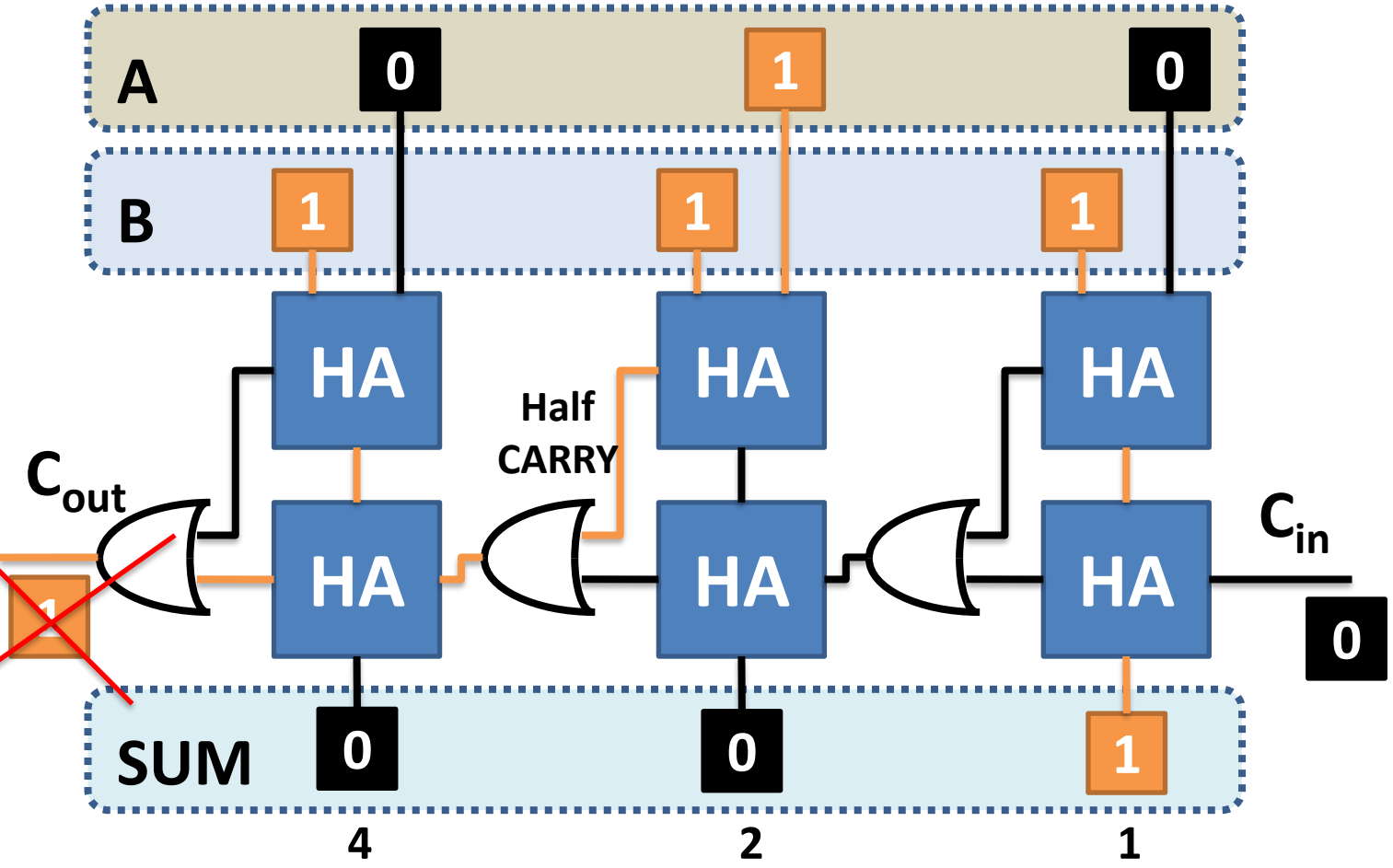
Proof by perfect induction

C_{in}	A	B	SUM	C_{out}	$A \oplus B$	$C_{in} \oplus (A \oplus B)$	AB	$C_{in}(A \oplus B)$	$AB + C_{in}(A \oplus B)$
0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	0	0
0	1	0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1	0	1
1	0	0	1	0	0	1	0	0	0
1	0	1	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0	1	1
1	1	1	1	1	0	1	1	0	1

$C_{out} = AB + C_{in}(A \oplus B)$

SUM = $C_{in} \oplus (A \oplus B)$

Multi-stage full adder with ripple carry



2

-1

= 1

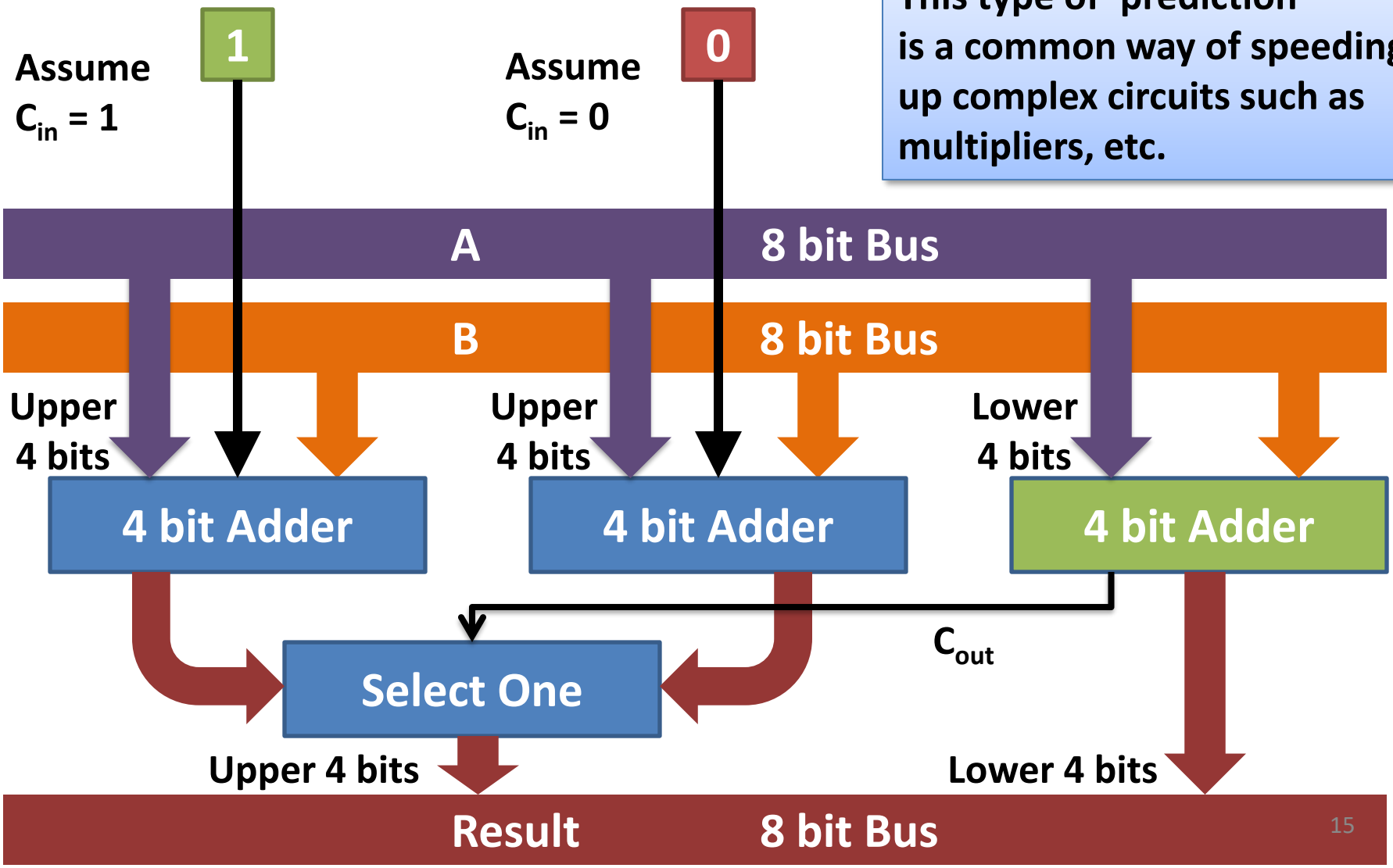
…Problem: *timing issues* – we would need to add "slow-down mechanisms"!

Carry-select adders

- Ripple-carry adders are *slow* because each stage must wait for the carry bit from the previous stage

- Solution: Split the problem: add the "lower" $n/2$ bits and the "upper" $n/2$ bits independently
  - Add the lower $n/2$ bits as normal
  - For the upper $n/2$ bits, use ***two*** sets of full adders: one set assumes $C_{in} = 0$, ...the other assumes $C_{in} = 1$
    - When we've added the first n bits, we'll know which set to use
  - This effectively **doubles the speed**
    - And we can keep splitting as long as cost/ space allows

Carry-select adder



Checkpoint: how far have we got with our ALU? (1)

- We can add whole numbers
  - And, of course, **using two's complement gives us subtraction**, too

    ...and **subtraction allows us to do comparisons**:
    <, >, =
    (do "subtraction", then check for +ve, -ve, or zero result)

Checkpoint: how far have we got with our ALU? (2)

- Multiplication and division??
  - We could do these in software, using addition and subtraction
  - Or we could do a hardware design based on long multiplication and long division methods
    - As with our adder, prediction can speed these up too

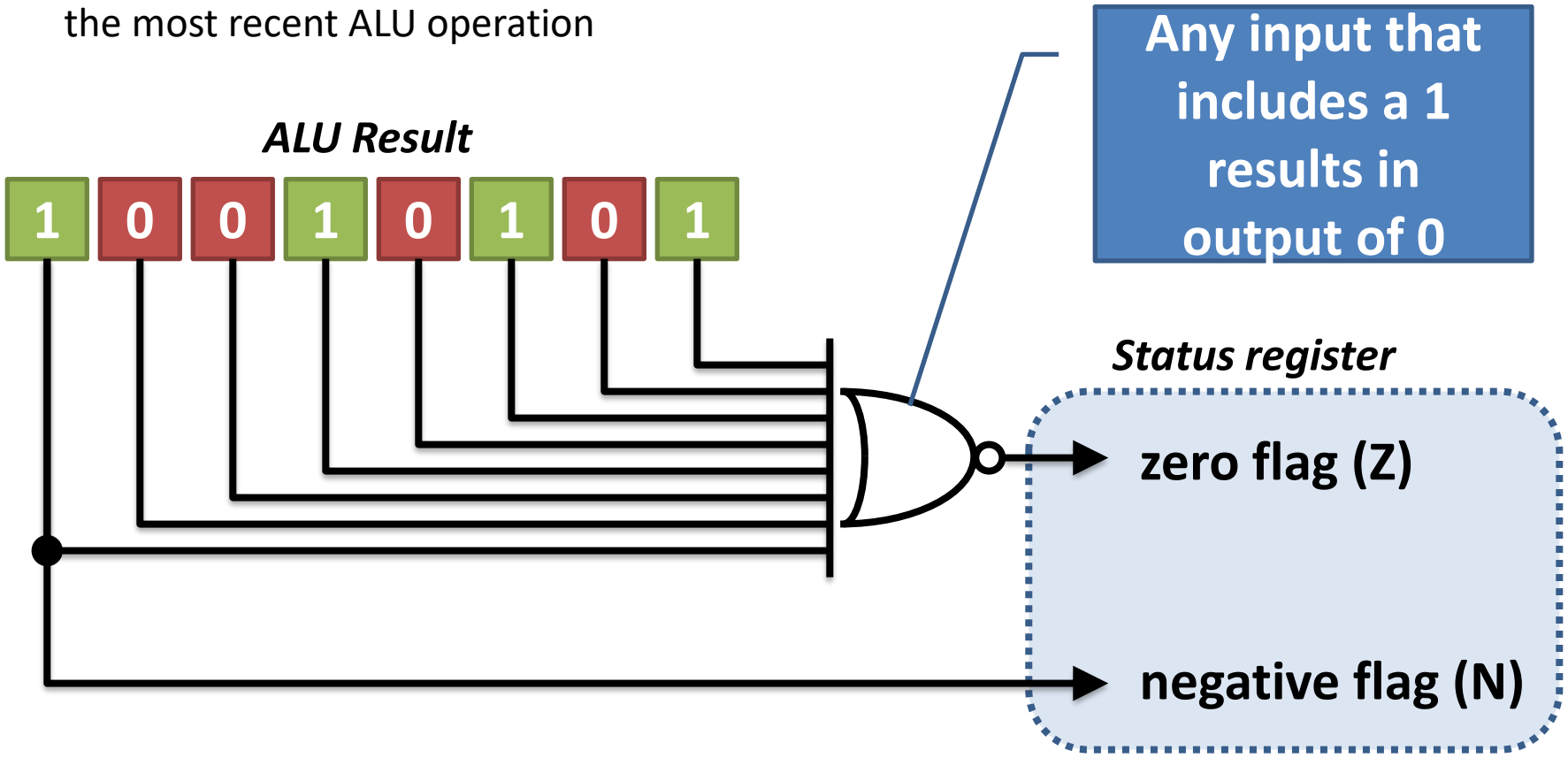
Checkpoint: how far have we got with our ALU? (3)

- Logic operations can easily be built directly from logic gates
  - AND, OR, NOT, ...

So, just the status flags to go...

The status flags

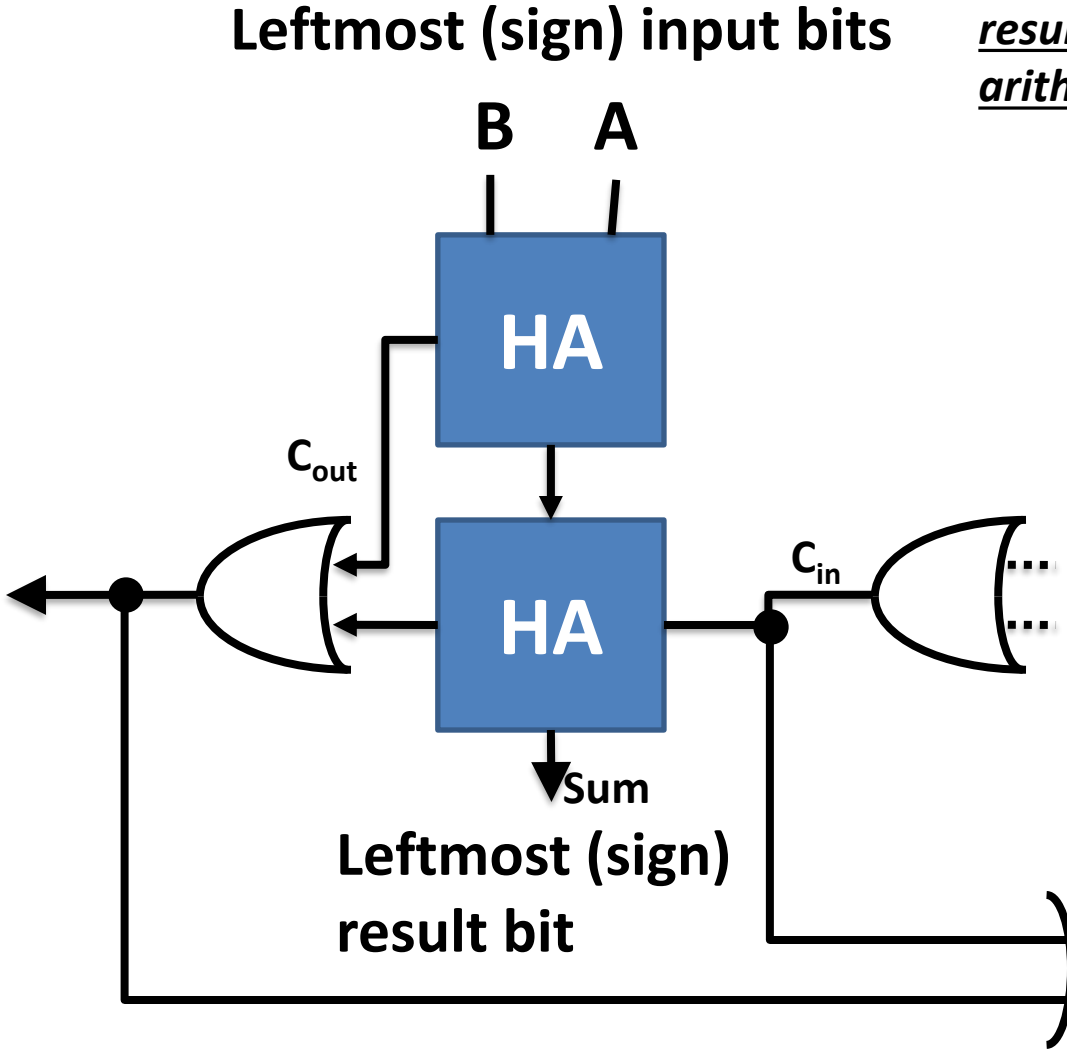
- The status flags are bits organised into a special register called the ***flags register***
- As we've already noted, each flag reflects an aspect of the outcome of the most recent ALU operation



The overflow flag is a little more challenging...

Overflow flag

"If the sign bits are the same but the result has a different sign, we have arithmetic overflow error"



C_{in}	C_{out}	V
0	0	0
0	1	1
1	0	1
1	1	0

exclusive OR again!

Why does this work?

- If C_{in} is 0, A is 1, B is 1, SUM is 0
  - In these circumstances, *we are adding two negative numbers and getting a positively-signed result*
- If C_{in} is 1, A is 0, B is 0, SUM is 1
  - In these circumstances, *we are adding two positive numbers and getting a negatively-signed result*

In Both situations, Overflow bit:
$C_{in} \oplus C_{out} = 1$ indicating <u>arithmetic overflow error</u>

1 bit addition

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

"***<u>If the sign bits are the same but the result has a different sign, we have arithmetic overflow error</u>***"

Bit shifting (1)

- As we've already seen, bit shifting enables simple multiplication/ division by powers of two, and can also speed up addition-based methods of multiplication

- Movement of a bit pattern left or right is also widely used in control systems and pattern matching

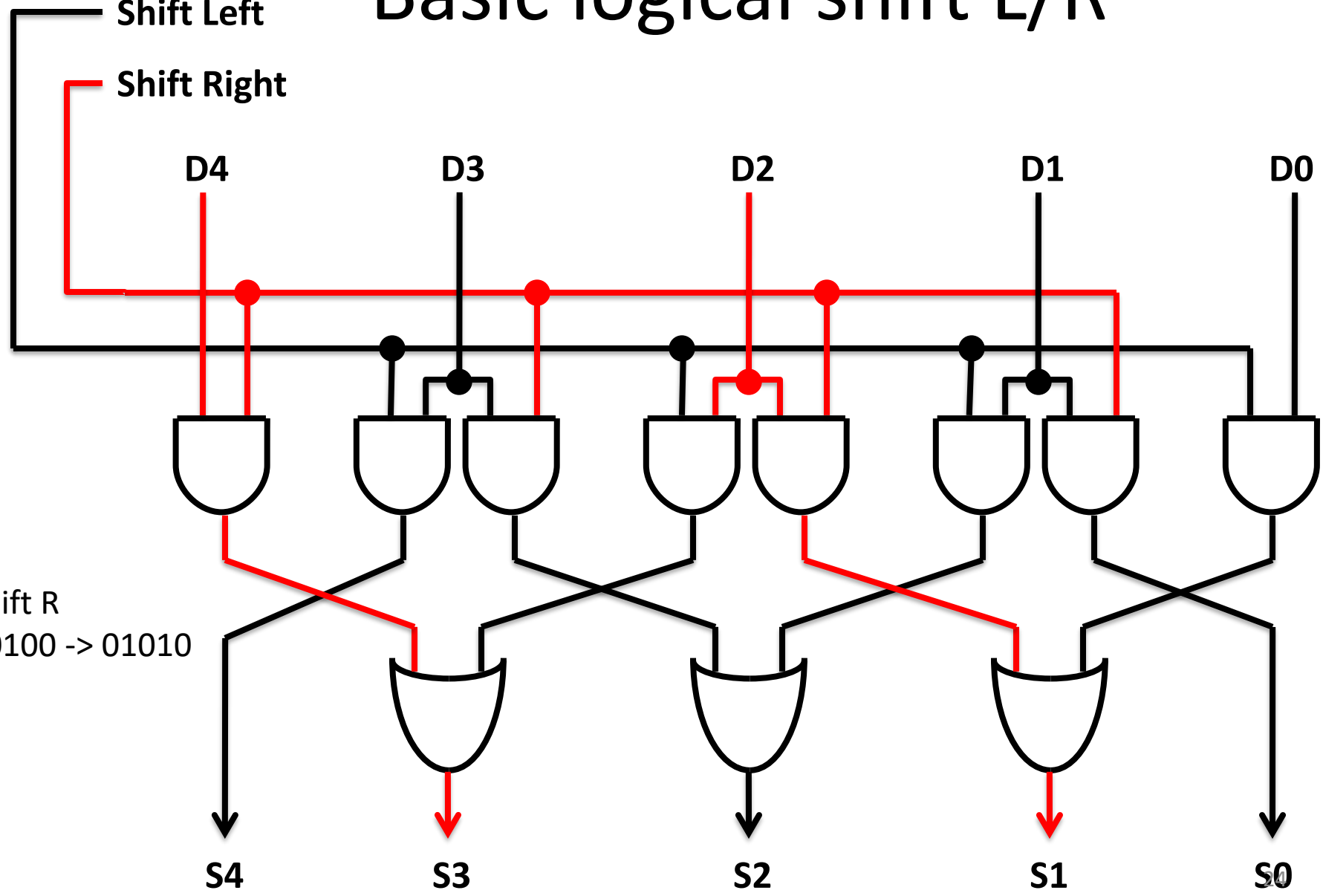
Bit shifting (2)

- Mainly 4 types of shifts: ***arithmetic shift (considers 2's complement), logical shift***, ***rotate, and rotate through carry***

- A simple "shifter" is shown on the next slide
  - More complex shifters can shift by multiple bits in a single operation (e.g. they employ "barrel shifters")
  - (Can you design this circuit from scratch, maybe a 3-bit version, using our now-familiar approach of:

    truth table → "sum of products" logic expression → logic circuit??)

Basic logical shift L/R



Shift R
10100 -> 01010

Summary

- We know (minimal forms of) all the fundamental facilities that an ALU needs to provide
  - We know how to do addition and, given 2's complement, subtraction
    - We can do this using full adders combined into either *ripple-carry* or *carry-select* multi-digit adders
  - We know how to implement the zero, negative and overflow flags
  - We know how to do bit shifting
- Again, all this is a *minimal* basis for an ALU – a real ALU will likely offer more!