CS2002 Logic Lecture 6

Digital Logic: Combinational Circuits

Ian Gent

Last Time

- DPLL Algorithm
  - Conjunctive Normal Form (CNF)
  - The algorithm
- Solving the whodunnit!
  - Figuring out the murderer
  - Proving they were the only one who could have done it
- Digital Logic Basics
  - Gates and simple circuits
  - Combinational Circuits

This time

- Combinational Circuits
- Circuits for Arithmetic
  - half adders and full adders
  - n-bit ripple adders
- Circuits for other activities in computers
  - Decoder
  - Multiplexer
  - Universal Logic Circuit

Adding Binary Numbers in Hardware

Half adder

- adding two binary digits A and B
- Their sum is going to be TWO bits
  - call these the “sum bit” and “carry bit”
- e.g. 1+1 = 10 (binary)
  - sum bit 0 and carry bit 1
- e.g. 0+1 = 01 (binary)
  - sum bit 1 and carry bit 0

Half adder

- Step 1
- Form truth table to get functionality we want
- Step 2
- Figure out how to do this with logic gates we have
- In this case ...
- Notice sum = XOR, carry = AND

inputs		outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Half adder

- Step 3
- Build circuit with right logic
- In this case
- XOR, AND and wire splits
- A/B inputs
- S = sum output
- C = carry output

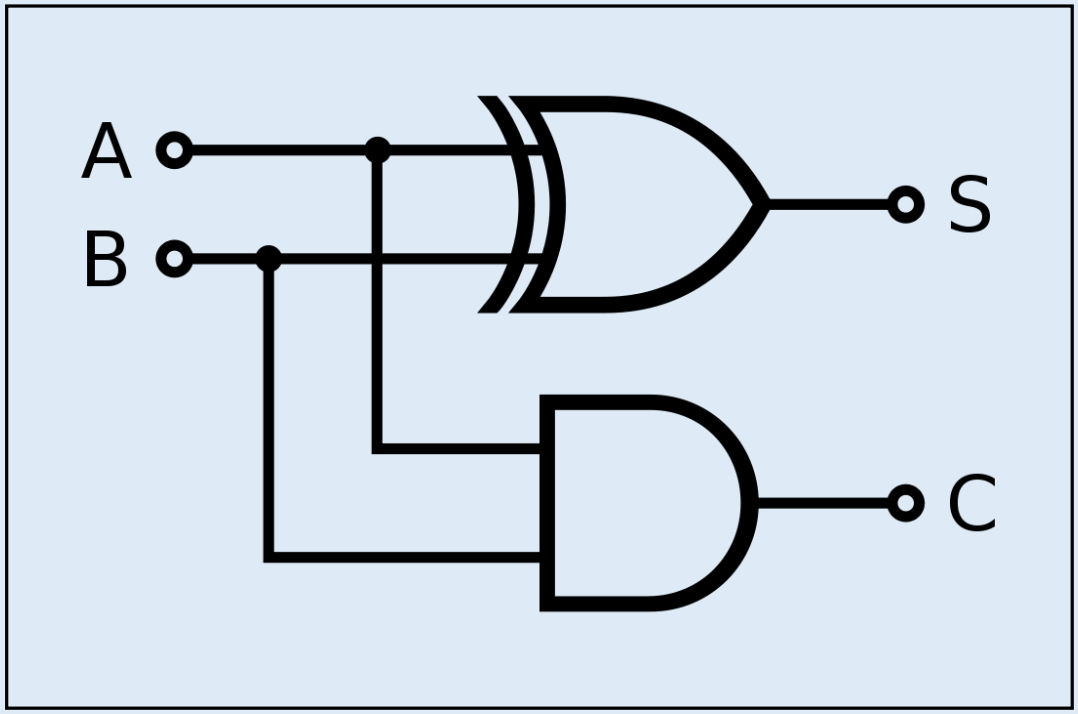


image: wikipedia

From Half to Full Adder

- All you can do with the half adder is add two bits together
- However, you can extend this adder to a circuit that allows the addition of larger binary numbers
- Key change is that we have THREE inputs
- Two bits to add plus “Carry In”
- Just as in normal long addition in base ten
  - have to record carry from one column to next

Full adder - Truth Table

Inputs			Outputs	
A	B	Carry-in	Sum	Carry-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

Full adder

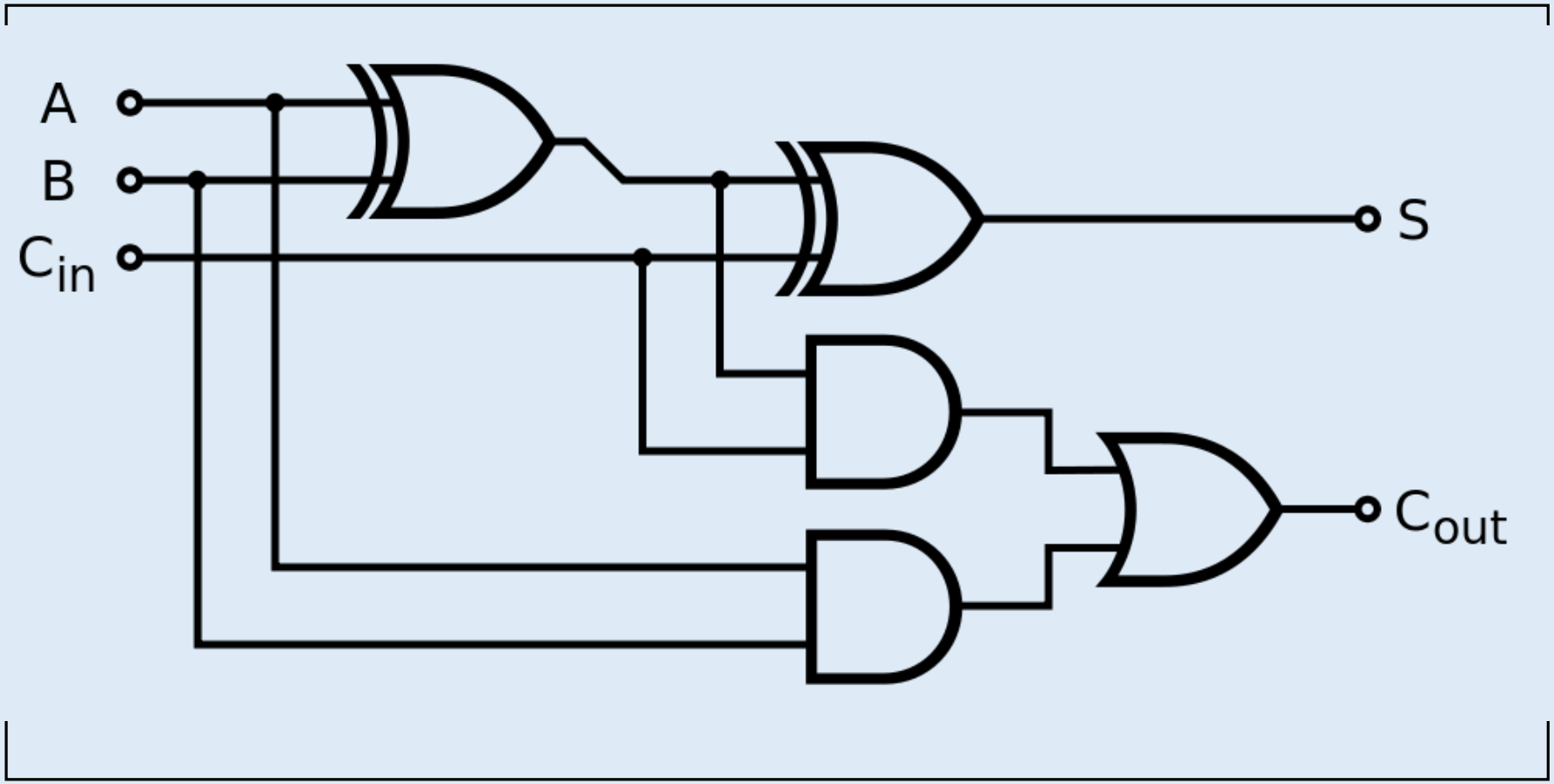


image: wikipedia

Full adder as half adders

You can just check the truth table for full adder
But you can also look at a full adder this way:

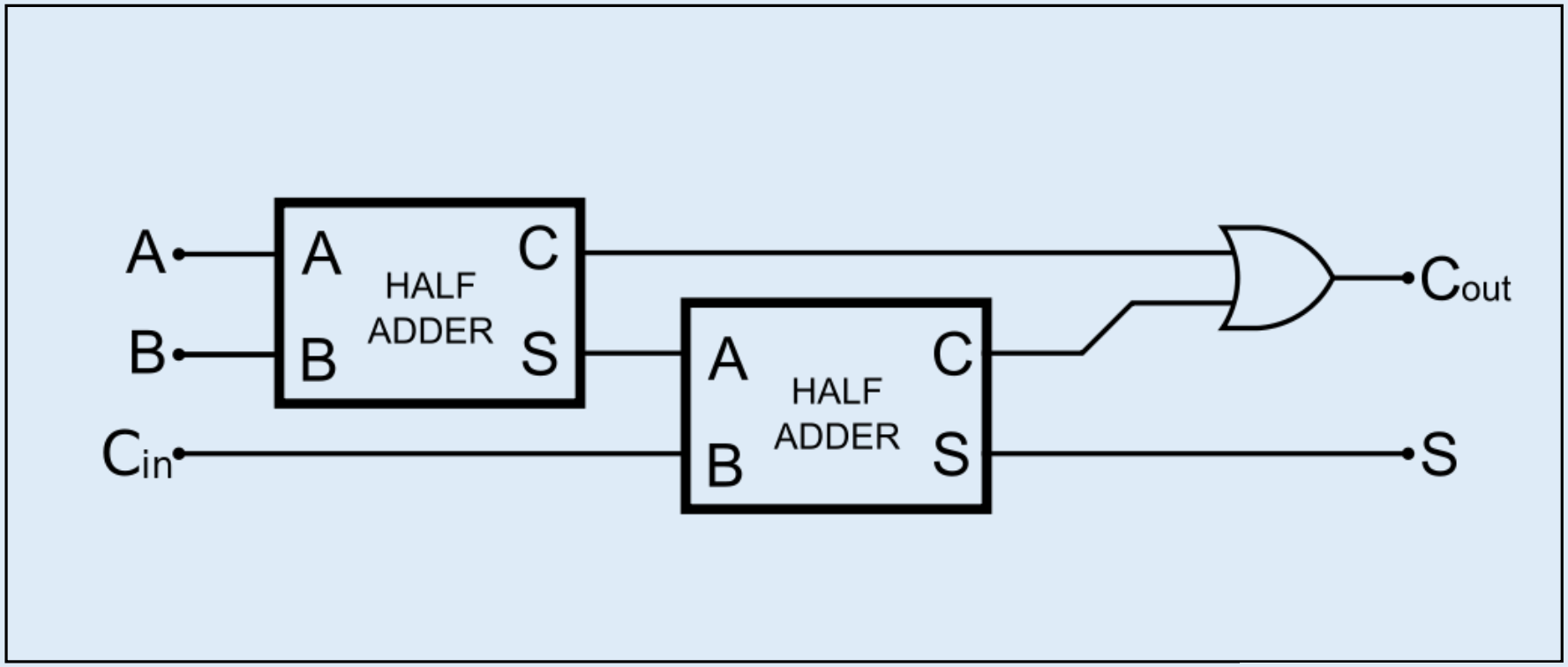


image: wikimedia commons

Can we add binary numbers now?

- We still can't: a full adder can only add 3 bits
- But we can build an adder capable of adding two 16-bit words, for example, by replicating the above circuit 16 times!
- The carry-out of one circuit becomes the carry-in of the circuit to its left, etc.

4-bit Ripple Adder

s = a+b

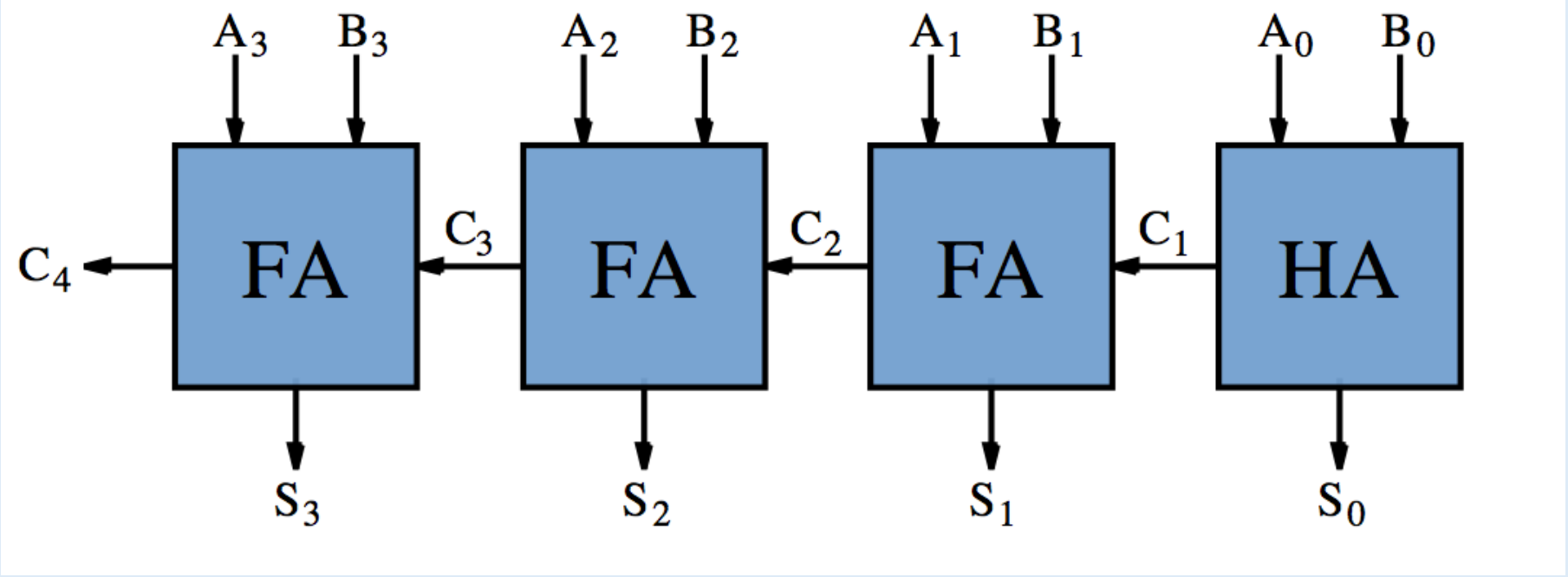


image: wikipedia, edited by Ian Gent

Right to left because numbers usually little-endian

Outputs are s3 to s0

Plus c4 if we want an overflow/carry bit

4-bit Ripple Adder

s = a+b

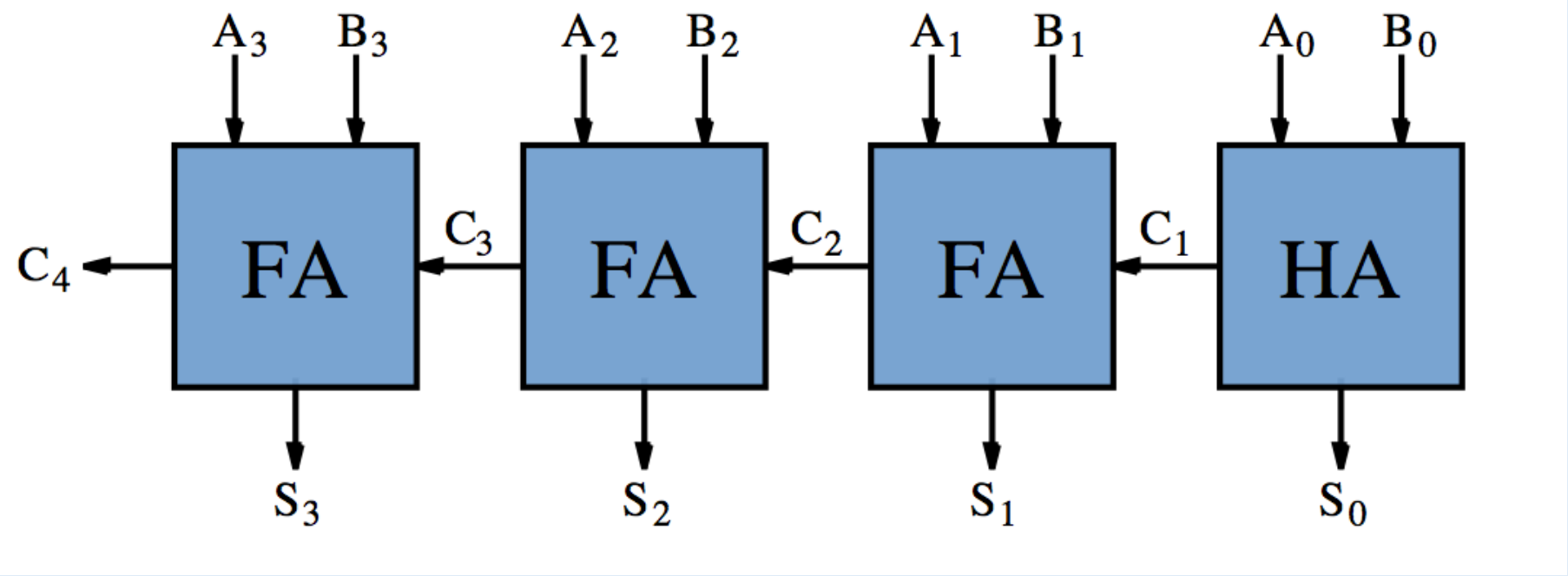


image: wikipedia, edited by Ian Gent

17 gates, 10 Gate delays,

157 and 94 for 32 bits,

317 and 190 for 64

Ripple-Carry Adder

- It is called ripple-carry adder because of the sequential generation of carries that “ripple” through the adder stages
- This adder is very slow, and is normally not implemented (but it gives a good idea of how addition can be achieved)
- Modifications have resulted in the carry-look-ahead adder, the carry-select adder, the carry-save adder -- 40% to 90% faster

Slow?

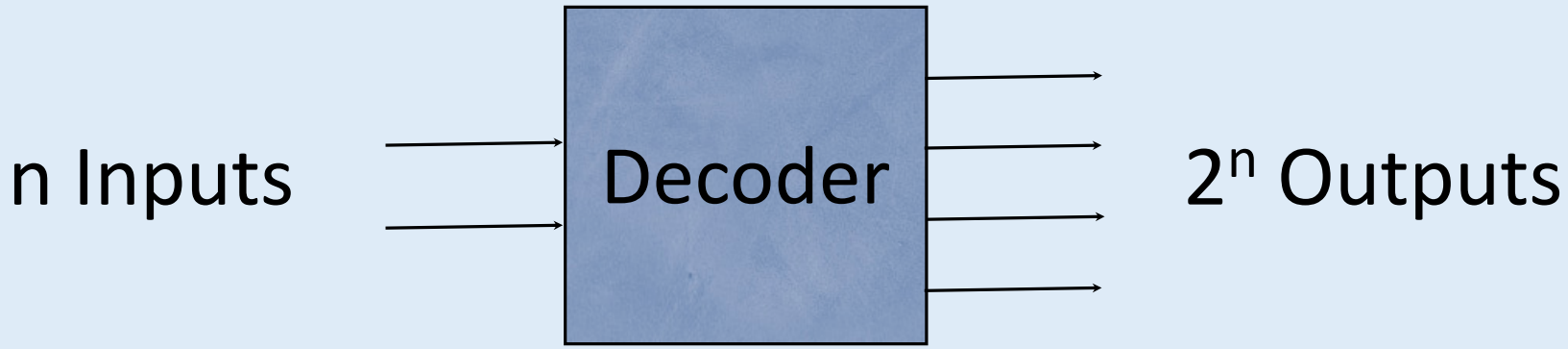
- “This adder is very slow”
- Why isn’t it instantaneous?

Slow?

- Two main reasons
- The speed of light is finite
  - on a 2GHz clock cycle light travels 15cm per tick
- Transistors are not instantaneous
  - “Gate delay” or “propagation delay”
  - Time from when input becomes stable to when output becomes stable
  - Total gate delay is maximum number of gates in any path from input to output
    - Larger gate delay = slower circuit
- So we want to minimise number of gate delays

Non-Arithmetical Functionality in Hardware

Decoder



Decoder

- All computers frequently need to decode binary information from a set of n inputs to a maximum of 2^n outputs
- A decoder uses the inputs and their respective values to select one specific output line (i.e., only one output line is set to 1, all others are 0)
- Example: Memory addresses are specified as binary numbers. When a memory address is referenced, the computer needs to determine the actual address.

2-to-4 Decoder

- Let's assume memory consisting of 4 chips, each containing 16Ki bytes
- Chip 0 memory addresses 0-16,383
- Chip 1 memory addresses 16,384-32,767
- and so on ...
- We have a total of 16Ki x 4 = 64Ki (65,536) addresses available

2-to-4 Decoder

- Given $64=2^6$ and $1Ki=2^{10}$ then $64Ki=2^{16}$ which means we need 16 bits to store an address
- Addresses in chip 0: 00xxxxxxxxxxxxx
- Addresses in chip 1: 01xxxxxxxxxxxxx
- Addresses in chip 2: 10xxxxxxxxxxxxx
- Addresses in chip 3: 11xxxxxxxxxxxxx
- We have to decode the first two bits
- Then send the xxx... to the right chip so it can retrieve that address

2-to-4 Decoder

- So far doesn't sound logical at all, does it?
- But the truth table is simple:

Input		Output			
bit1	bit2	chip0	chip1	chip2	chip3
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

2-to-4 Decoder

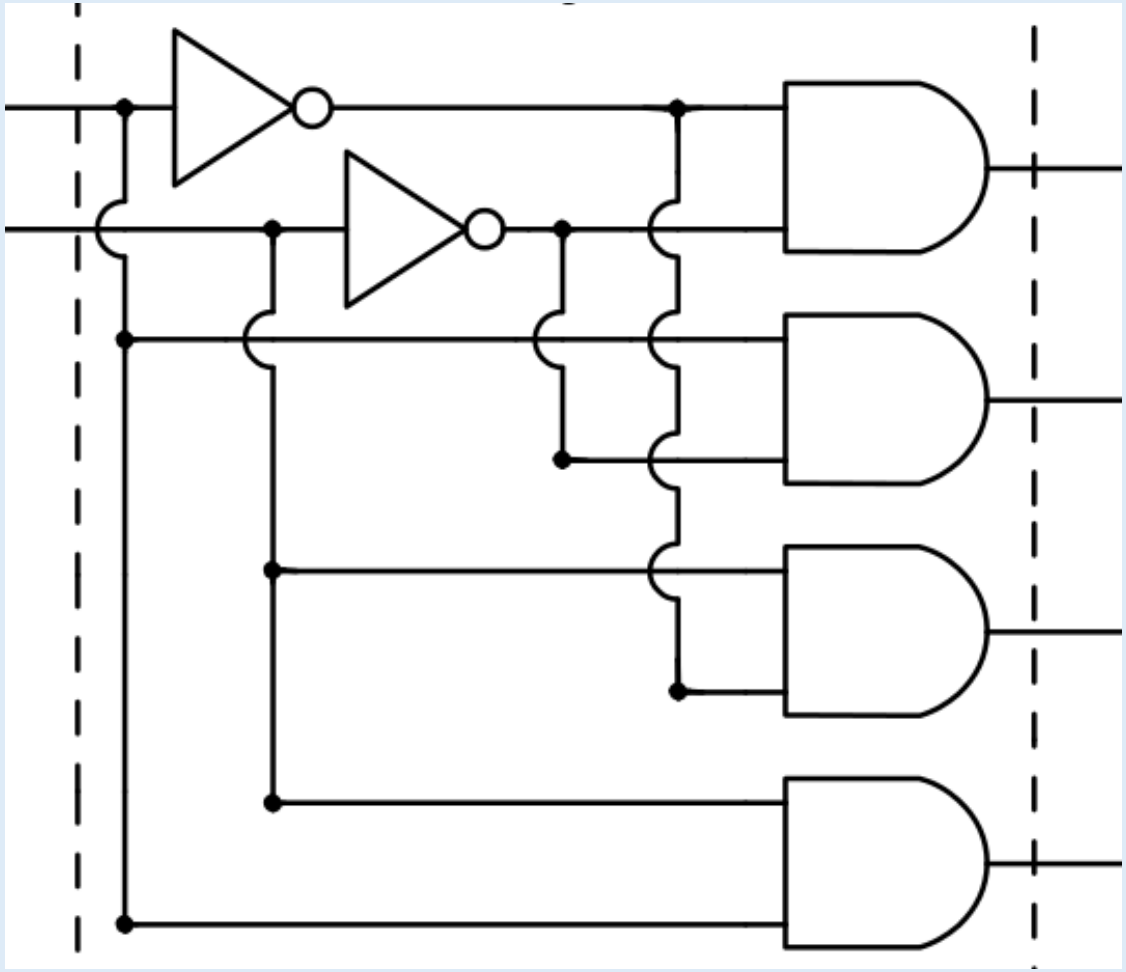


image: wikipedia

2-to-4 Decoder

We have successfully decoded a Binary number and put 1 on the correct output

But what if we want to do more
E.g. copy data from correct input to the output?

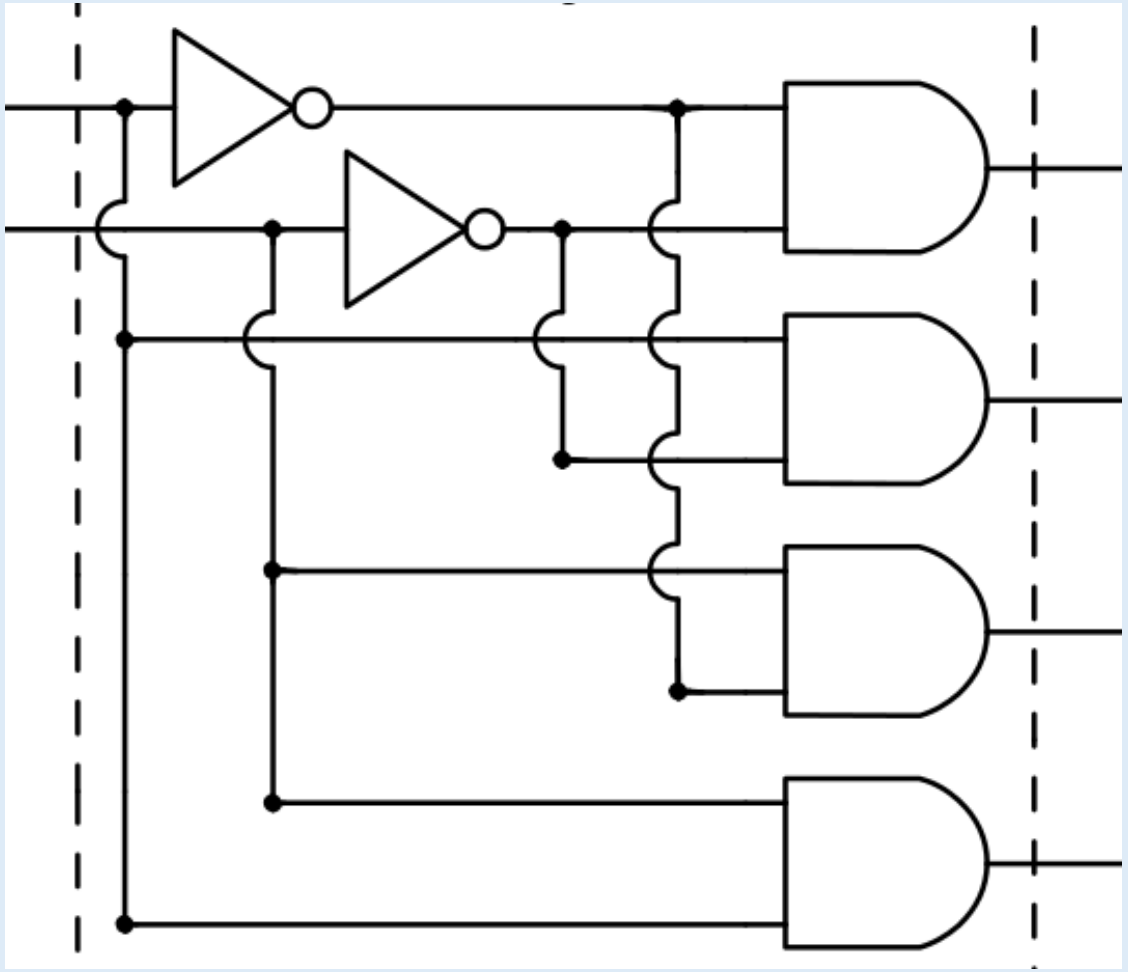


image: wikipedia

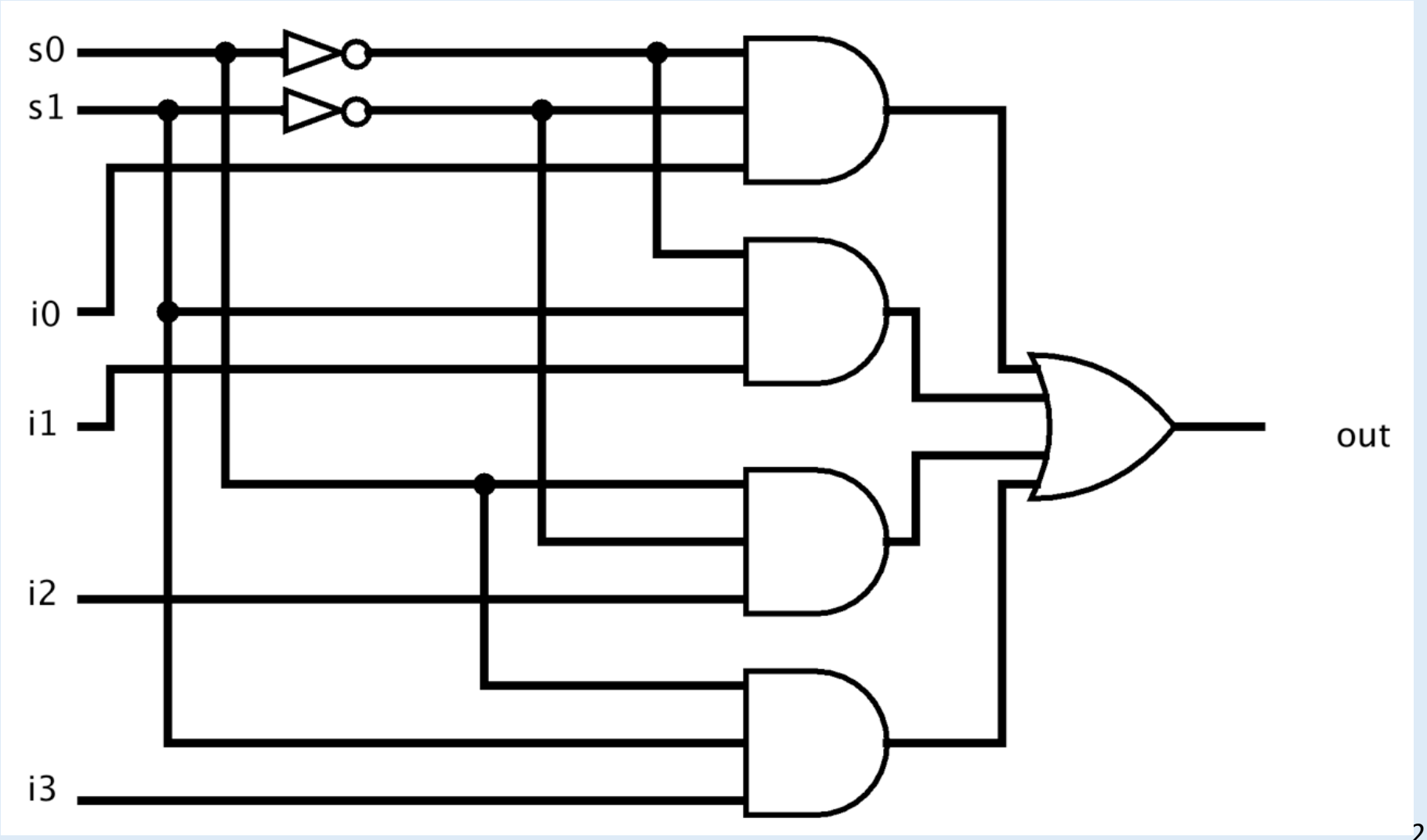
Multiplexer

- Another common combinational circuit
- Selects binary information from one of many input lines and directs it to a single output line
- Selection is controlled by selection variables
- E.g. select one of four inputs i0 to i3
- Based on binary values of two selectors s0 s1
- Will shorten truth table with don’t care “?” values

Multiplexer

Selectors		Input Lines				Out
s0	s1	i0	i1	i2	i3	out
0	0	0	?	?	?	0
0	0	1	?	?	?	1
0	1	?	0	?	?	0
0	1	?	1	?	?	1
1	0	?	?	0	?	0
1	0	?	?	1	?	1
1	1	?	?	?	0	0
1	1	?	?	?	1	1

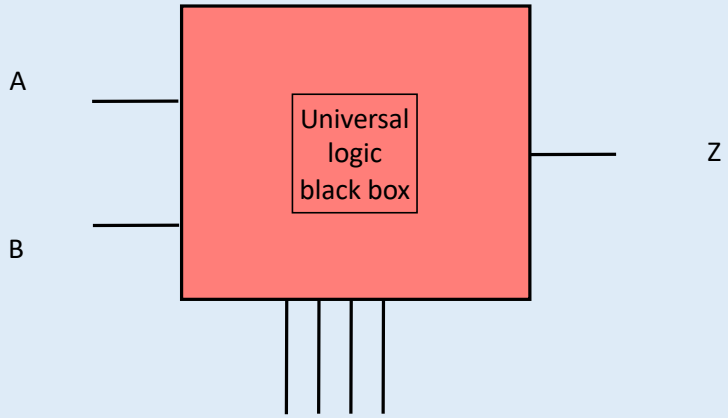
A Multiplexer



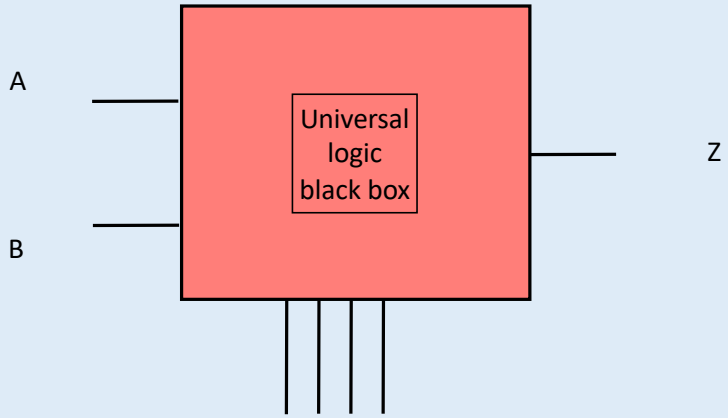
A Universal Logic Circuit

- We have gates for implementing specific logic operations
  - AND, OR, NOT, and so on
- Useful when we know at design time what logic we need to implement in a circuit
- However, in many applications we must perform various logic operations on a set of inputs based on command information which is not available when we are designing

Universal Logic Circuit (ULC)



Why four control inputs for two bits?



Control Inputs i0 i1 i2 i3

Why four control inputs for two bits A & B?

- There are 2^2=4 possible values of A & B corresponding to the 4 lines in truth table
- So there are 2^4=16 possible truth functions applied to A & B corresponding to the 2x2x2x2=16 ways we can assign the truth table

16 Logic Functions Z0 to Z15

A B	Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15
0 0 0 1 1 0 1 1	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
Function	0	∧	A ∧ ¬B	A	¬A ∧ B	B	XOR	v	NOR	↔	¬B	B → A	¬A	A → B	NAND	1

What is going on here?

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
Function	0	∧	A ∧ ¬B	A	¬A ∧ B	B	XOR	v	NOR	↔	¬B	B → A	¬A	A → B	NAND	1

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1	0 0 0 0															
Function	0															

Can

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1		0 0 0 1														
Function		∧														

Can you

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1			0 0 1 0													
Function			A ∧ ¬B													

Can you see

A B	Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15
0 0 0 1 1 0 1 1				0 0 1 1												
Function				A												

Can you see how

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1					0 1 0 0											
Function					¬A ∧ B											

Can you see how the

A B	Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15
0 0 0 1 1 0 1 1						0 1 0 1										
Function						B										

Can you see how the binary

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1							0 1 1 0									
Function							XOR									

Can you see how the binary representation

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1								0 1 1 1								
Function								v								

Can you see how the binary representation of

A B	Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15
0 0 0 1 1 0 1 1									1 0 0 0							
Function									NOR							

Can you see how the binary representation of the

A B	Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15
0 0 0 1 1 0 1 1										1 0 0 1						
Function										↔						

Can you see how the binary representation of the function

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1											1 0 1 0					
Function											¬B					

Can you see how the binary representation of the function number

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1												1 0 1 1				
Function												B → A				

Can you see how the binary representation of the function number actually

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1													1 1 0 0			
Function													¬A			

Can you see how the binary representation of the function number actually defines

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1														1 1 0 1		
Function														A → B		

Can you see how the binary representation of the function number actually defines the

A B	Z0	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10	Z11	Z12	Z13	Z14	Z15
0 0 0 1 1 0 1 1															1 1 1 0	
Function															NAND	

Can you see how the binary representation of the function number actually defines the function?

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1																1 1 1 1
Function																1

Can you see how the binary representation of the function number actually defines the function?

A B	Z 0	Z 1	Z 2	Z 3	Z 4	Z 5	Z 6	Z 7	Z 8	Z 9	Z 10	Z 11	Z 12	Z 13	Z 14	Z 15
0 0 0 1 1 0 1 1	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
Function	0	∧	A ∧ ¬B	A	¬A ∧ B	B	XOR	v	NOR	↔	¬B	B → A	¬A	A → B	NAND	1

A universal logical circuit

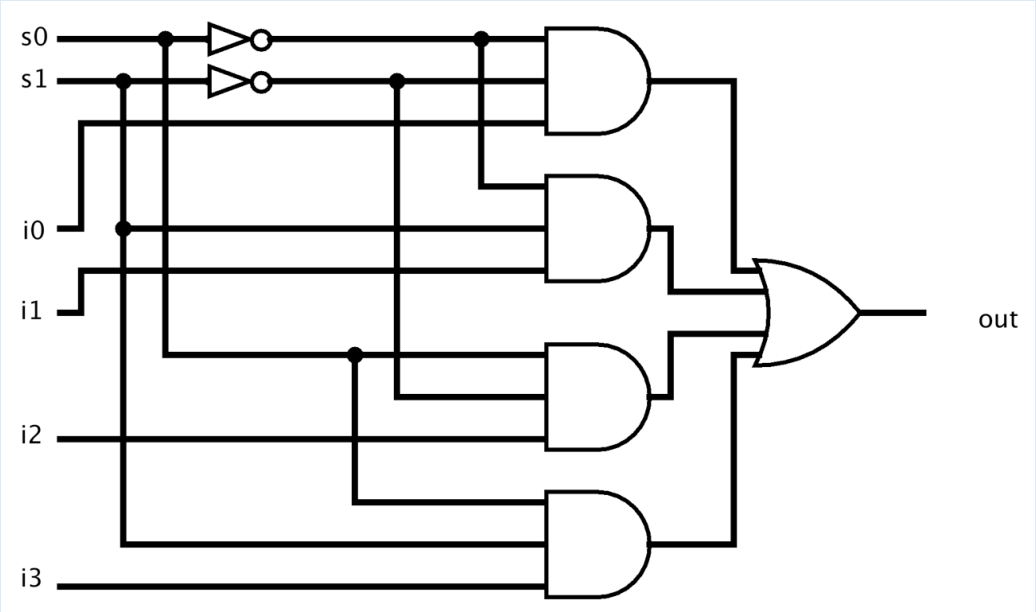
- A circuit that implements this table can:
  - Ignore A and B and produce a fixed 0 or 1
  - Pass input A or input B unchanged
  - Perform our usual logic functions, NOT, AND, OR, XOR, Implies (either direction)
  - Perform additional functions of two variables (e.g. NAND and NOR)

Universal Logic Circuit

- How could we implement this ULC?
  - 4 input bits i0, i1, i2, i3
  - controls which function Z0 to Z15
  - 2 more input bits A, B
- There are many circuit designs supporting all 16 Boolean operations
- As before, we could construct a large circuit from the truth table
- There is a more concise way ...

ULC via Multiplexer

- Normal view of MUX
  - 4 input bits
  - two selection bits
  - selects one of 4 inputs
- ULC view of MUX
  - two input bits A, B (here s0 s1)
  - four selection bits
  - selects one of 16 truth functions
  - E.g. A=B=1, value of MUX is value of i3
  - You can check on earlier slide that value of i3 is correct



Do you see what we did?

- In the Universal Logic Circuit
- We have 4 selection bits
- Those decide which function the ULC implements
- Those 4 bits are really control bits
- In a simple 4 bit instruction set
- This is (of course) the idea behind CPUs
  - Not the ULC itself but control bits making things happen

Next Time

- Sequential circuits