We acknowledge and pay our respects to the Kaurna people,
the traditional custodians whose ancestral lands we gather on.

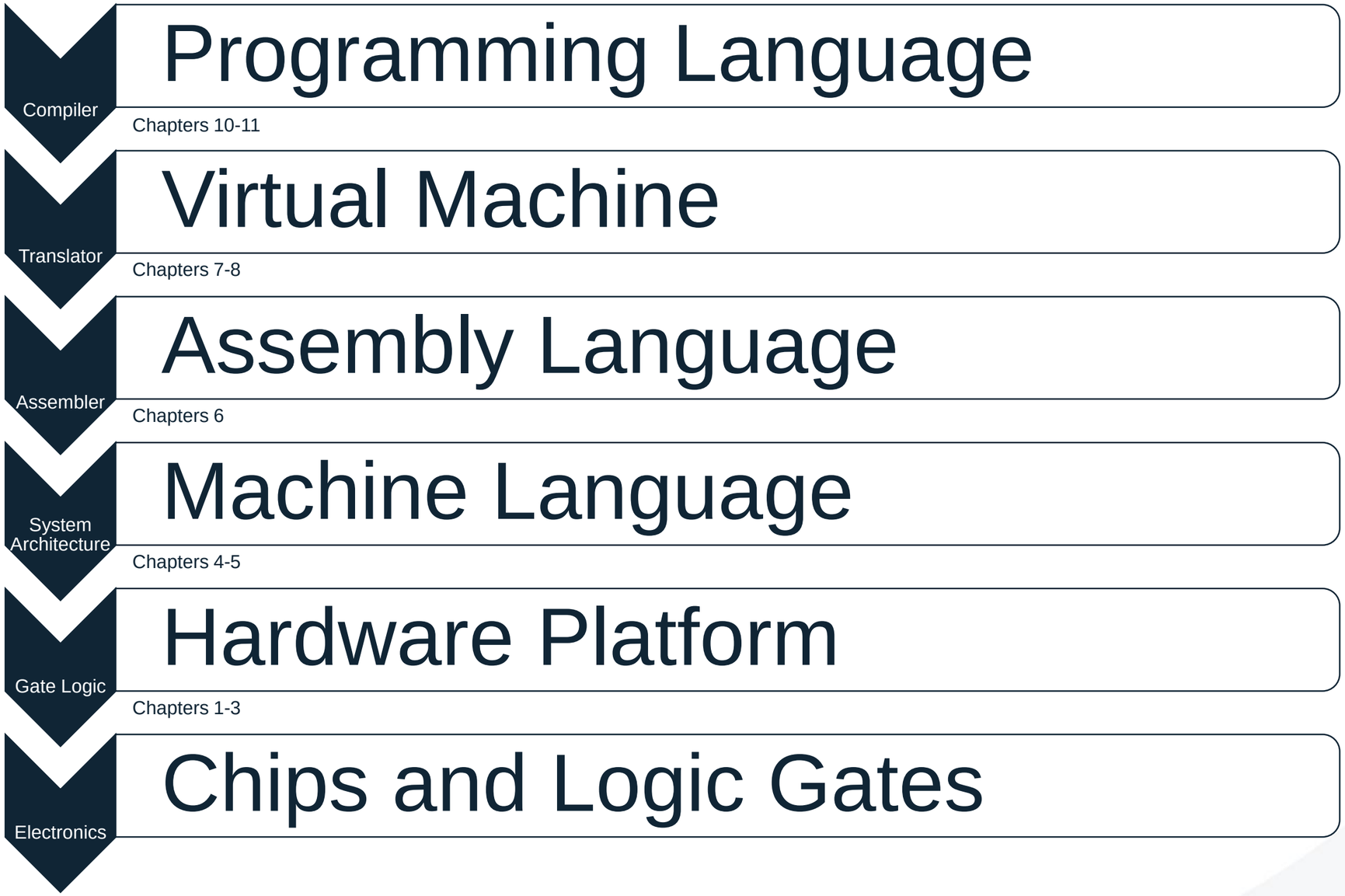
We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

THE UNIVERSITY of ADELAIDE

Computer Systems

Lecture 02: Boolean Arithmetic & Sequential Logic

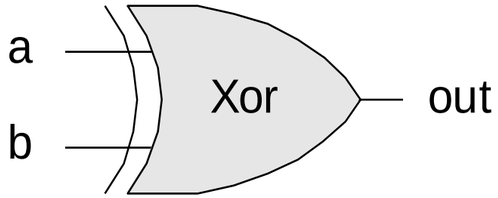
Review: The whole system



THE UNIVERSITY of ADELAIDE

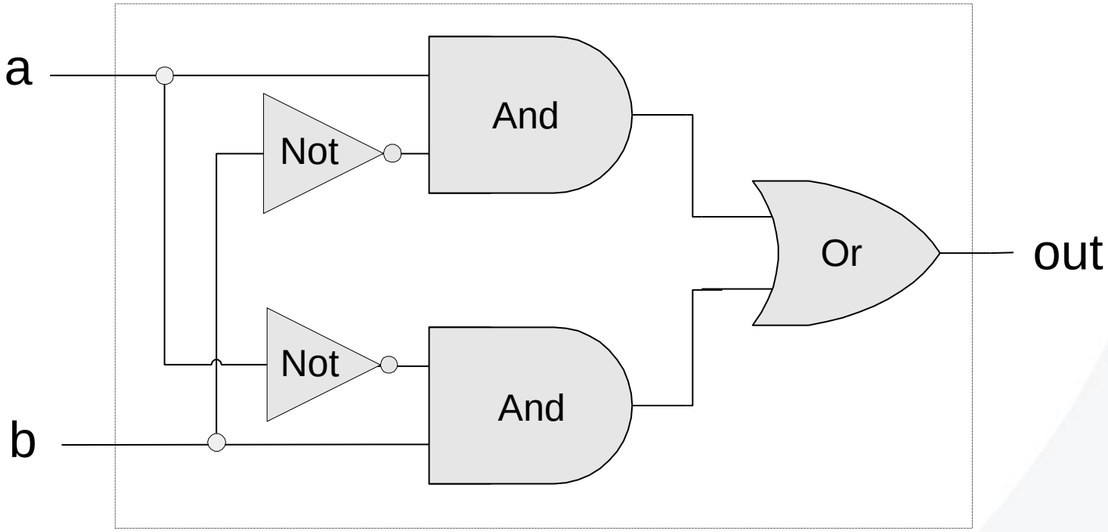
Review: Example

Interface



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Implementation



Review: Canonical Form

We can construct a canonical representation of any boolean function

For each row that gives a 1 in its truth table

- **and** together all terms after applying **not** to any 0 to make it a 1
- if applied to any other row, the equation will evaluate to 0

Then

- **or** together the equations for every row that gives a 1

XOR

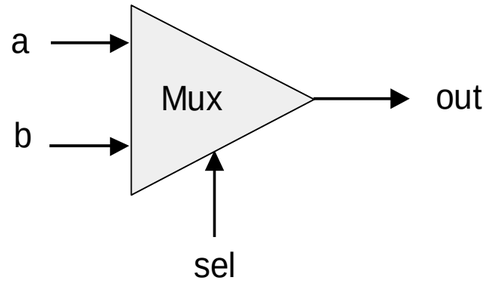
$\bar{x}.y + x.\bar{y}$

x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

So you only need and, or and not gates

Canonical Form – Mux

a	b	sel	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

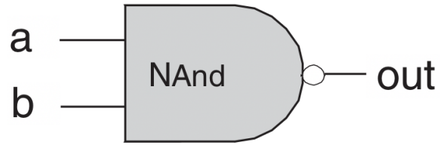



```
out = if sel == 0
then a else b
```

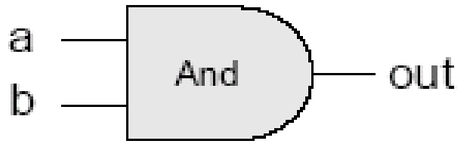
Review: Gate logic

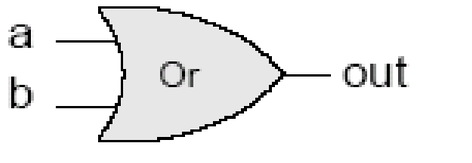
Gate logic – a gate architecture designed to implement a Boolean function

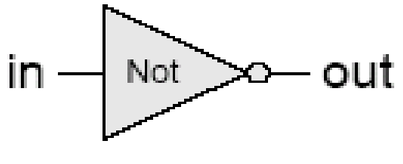
- Elementary gates:

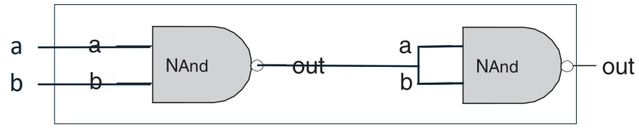


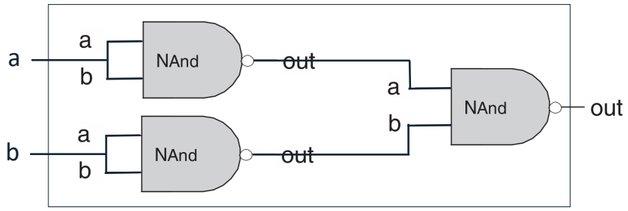
- Composite gates:

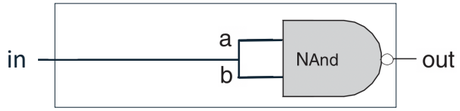












- <u>Important distinction:</u> Interface (*what*) VS implementation (*how*).

THE UNIVERSITY of ADELAIDE

Binary Arithmetic

Number Representation

How do we represent numbers in the Computer?

- Our logic gates can only handle 2 possible values; 0 or 1
- How do we represent negative numbers when we don’t have a +/- sign?

Number Representation

Decimal	4-bit 2's Complement	Decimal
-8	1000	8
-7	1001	9
-6	1010	10
-5	1011	11
-4	1100	12
-3	1101	13
-2	1110	14
-1	1111	15
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Binary Addition

Assuming a 4-bit system:

```
  0 0 0 1          1 1 1 1
    0 0 0 1          1 0 1 1
+   0 1 0 1      +   0 1 1 1
-------------    -------------
  0 0 1 1 0        1 0 0 1 0
```

No overflow 1 + 5 = 6

Overflow -5 + 7 = 2

- Algorithm: exactly the same as in decimal addition
- Overflow (MSB carry) may need to be dealt with – we usually ignore it.

Binary Addition

- How do we know if a 2's complement number is negative?
  - The Most Significant Bit is 1
- There is only one representation of 0
- To negate a number, flip all the bits and add 1
- If you flip all the bits in a number x, you get -x – 1
- Sometimes the result of an add operation is wrong!
  - Using subtract to compare numbers needs to account for this effect

```
 1  0  1  1
    1  0  1  1
+   1  0  1  1
-------------
 1  0  1  1  0
```

Bad overflow -5 + -5 = 6

```
 0  1  0  0
    0  1  0  1
+   0  1  0  0
-------------
 0  1  0  0  1
```

Bad overflow 5 + 4 = -7

THE UNIVERSITY of ADELAIDE

Building an Adder chip

Adder: a chip designed to add two integers

Proposed implementation:

Half adder: designed to add 2 bits (we only need one)

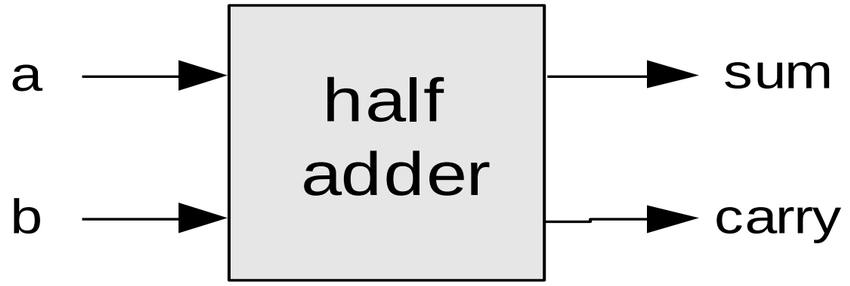
Full adder: designed to add 3 bits (we need n-1 of these)

Adder: designed to add two *n*-bit numbers.

THE UNIVERSITY of ADELAIDE

Half adder (designed to add 2 bits)

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



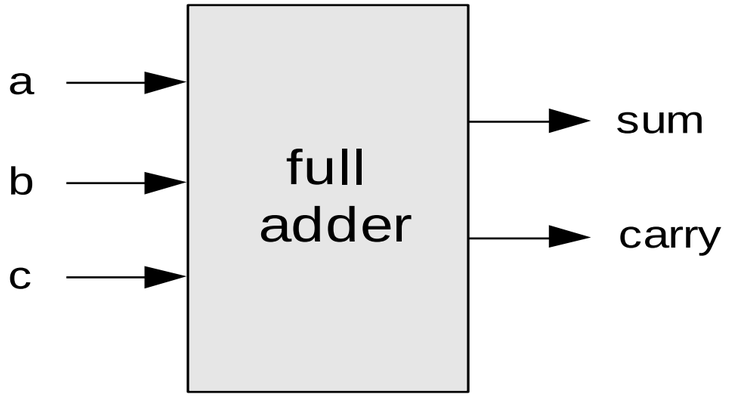
A half adder can be built from an xor and an and

the sum column matches xor

the carry columns matches and

Full adder (designed to add 3 bits)

a	b	c	sum	carry
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



Implementation: can be based on half-adder gates.

Perspective

- Combinational logic
  - The canonical representation would be too big to use
- Our adder design is very basic: no parallelism
- What about chips for more advanced mathematical operations?
  - A typical hardware/software tradeoff.

Sequential Logic

Sequential logic

This operates on data and a clock signal;
as such, can be made to be *state*-aware and provide storage and synchronization services

What does *state-aware* mean?

Sequential devices are sometimes called “clocked devices”

DFFs

All sequential chips can be based on one low-level sequential gate, called “data flip flop”, or DFF (DFFs can be made from NAND gates).

The clock-dependency details can be encapsulated at the low-level DFF level

Higher-level sequential chips can be built on top of DFF gates using combinational logic only

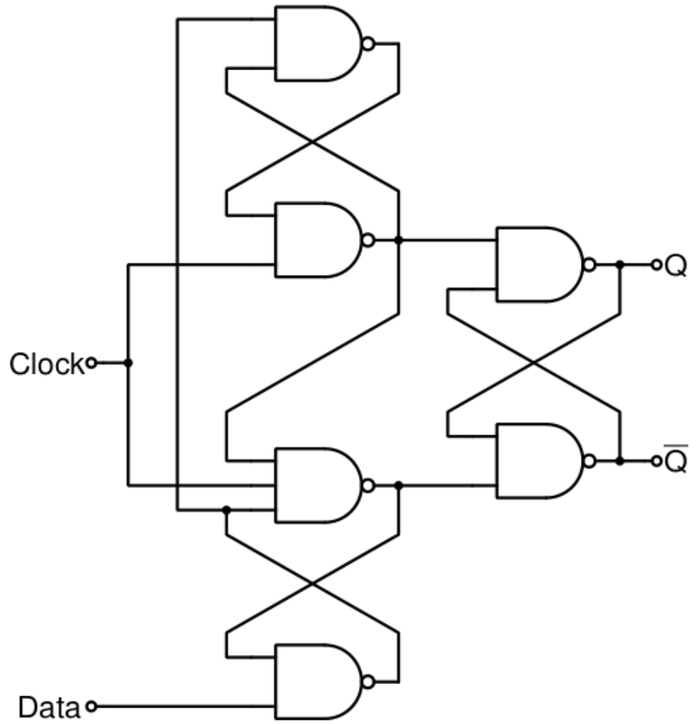
You’re probably getting used to the idea that, if you understand some of the low-level stuff, you can build more and more complex machines.

THE UNIVERSITY of ADELAIDE

Inside a DFF

Just so you can see it can be done with NANDs

We won’t worry about the details inside a DFF in this course!

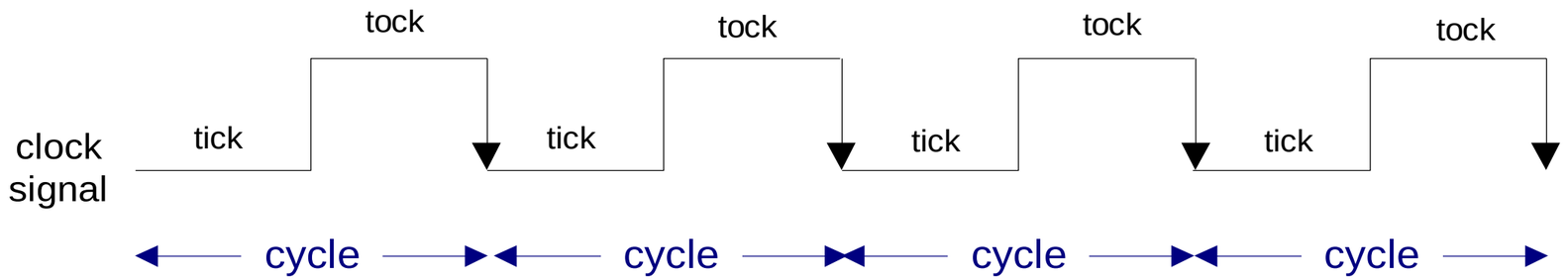


What's a clock in this context?

We use clocks to measure time.

In this case, the clock *signal* is used to allow us to reflect what happens as the state of hardware changes over time.

The hardware clock signal *oscillates* in low/high tick/tock phase.



Clocks

- The speed of the clock is going to affect the speed of everything in the computer as a clock cycle is the smallest unit of time in which anything can change.
- In actual hardware, we have specific components and circuits to do this for us.
- In the simulator, the user can feed the clock signal in manually or we can use a test script.

State

The state of an element is the condition it is in at a given time.

A binary digit (bit) can be 0 or 1 but the value of that bit, at a certain time, is its state.

What does this look like as a concept?

out will give us a 0 or 1 depending on what is in there.

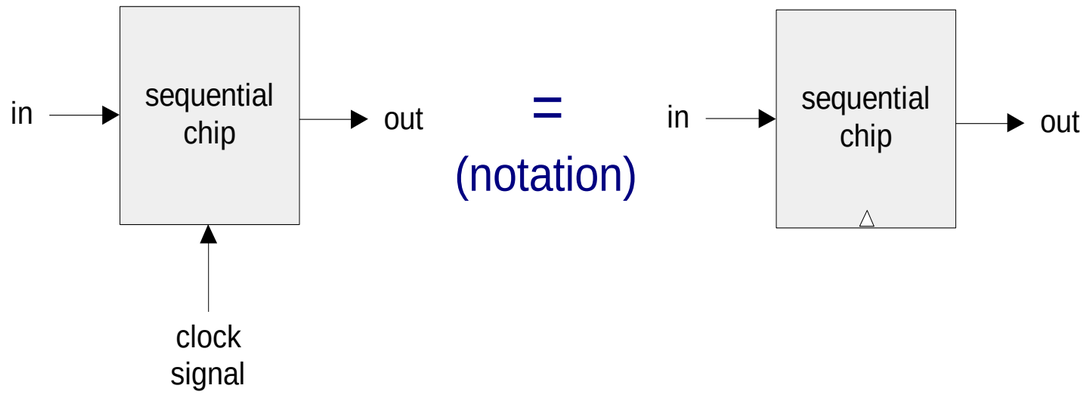
out(t) = in(t-1)

Using the clock

Memory is made up of lots of these cells, all running on the same master clock signal.

(We're not worrying about the implementation at the moment.)

When do we change the contents? The *in* signal will always be set to something!

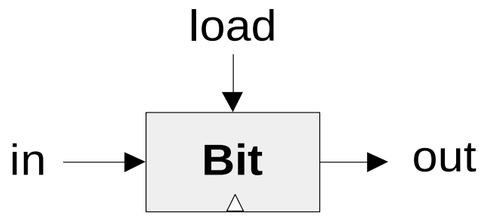


Let’s build a bit! (1-bit register)

We want to be able to:

Change the state of the bit

Retain that state until we want to change it.



if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

Now we can tell the bit when to change.

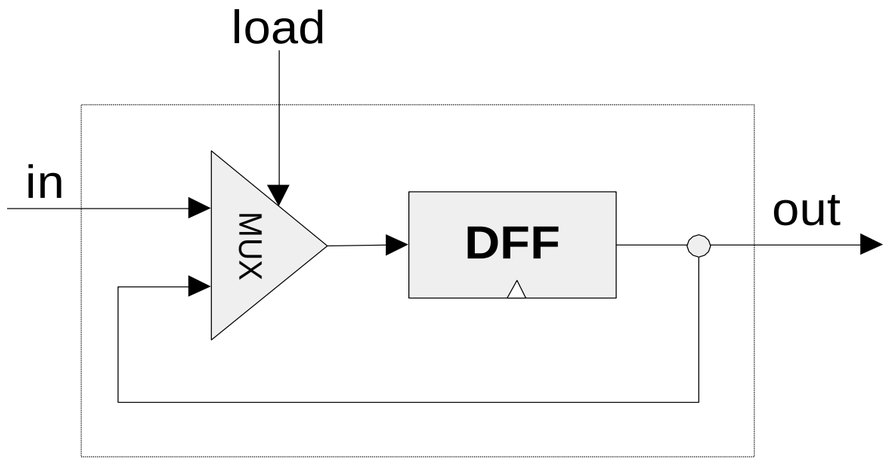
Can we build this from the DFF and gates we already know?

Bit register

Notice how we use the load signal.

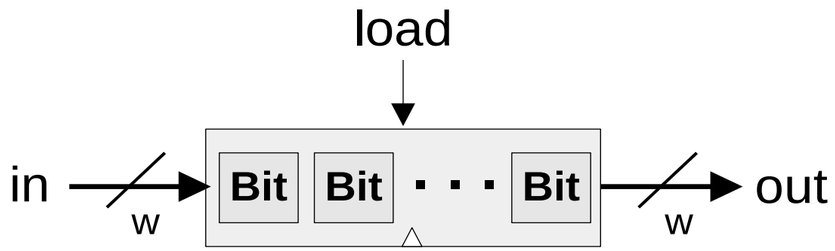
Now we have read logic and write logic.

Implementation



Bigger registers

We know we can work with more than one bit.

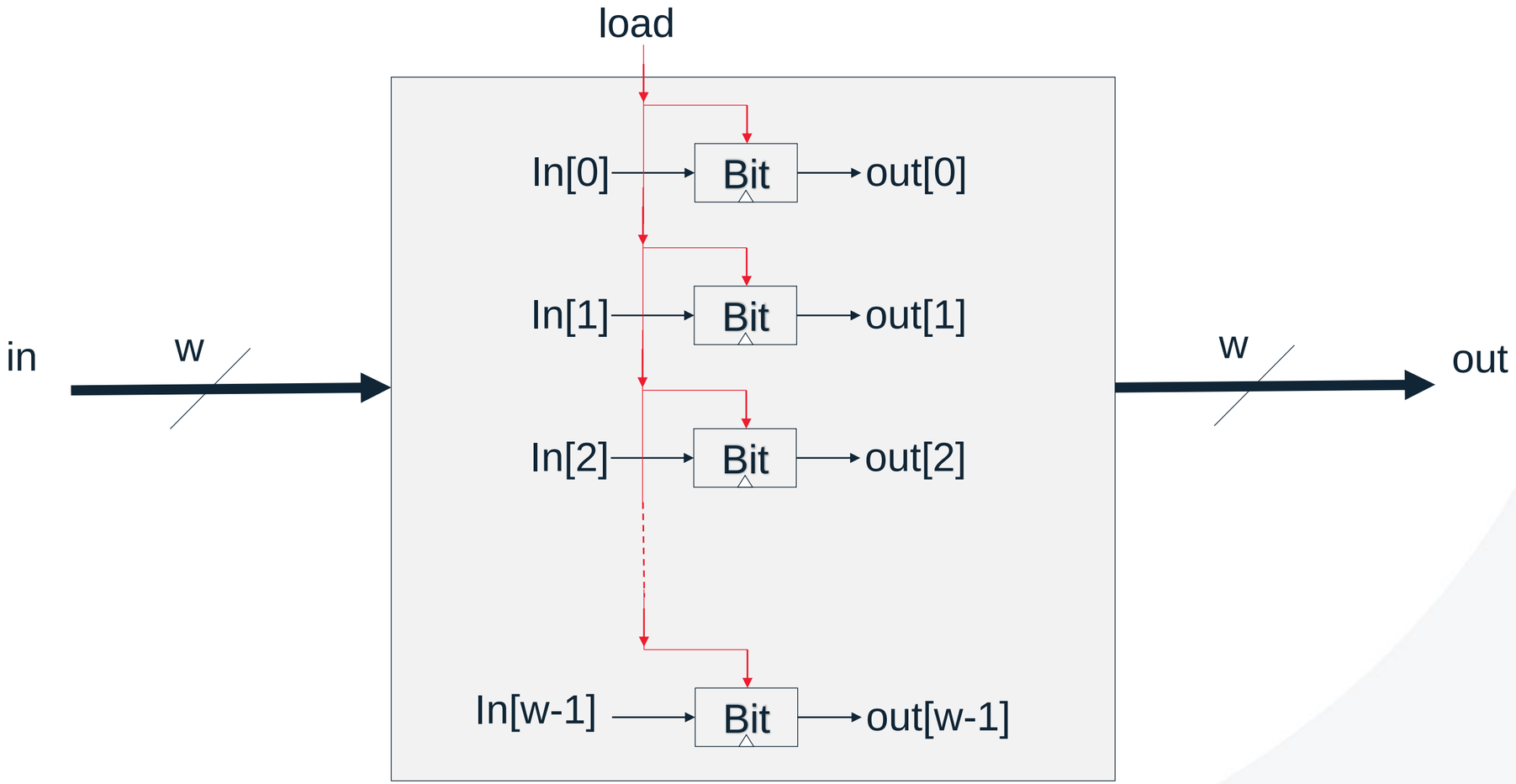


if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

w-bit register

Bigger registers

We know we can work with more than one bit.



Summary

- You can construct many gates from NAND – this is just one example of how gates are built up.
- By understanding arithmetic, we can combine gates to add two numbers, then combine full-adders to add larger numbers.
- DFFs allow us to store state by delaying output. This allows us to build registers.

This Week

- Workshops start this week (for real this time)
- Review Chapters 2 & 3 of the Text Book (if you haven’t already)
- Start Assignment 1
- Review Chapter 4 of the Text Book before next week.