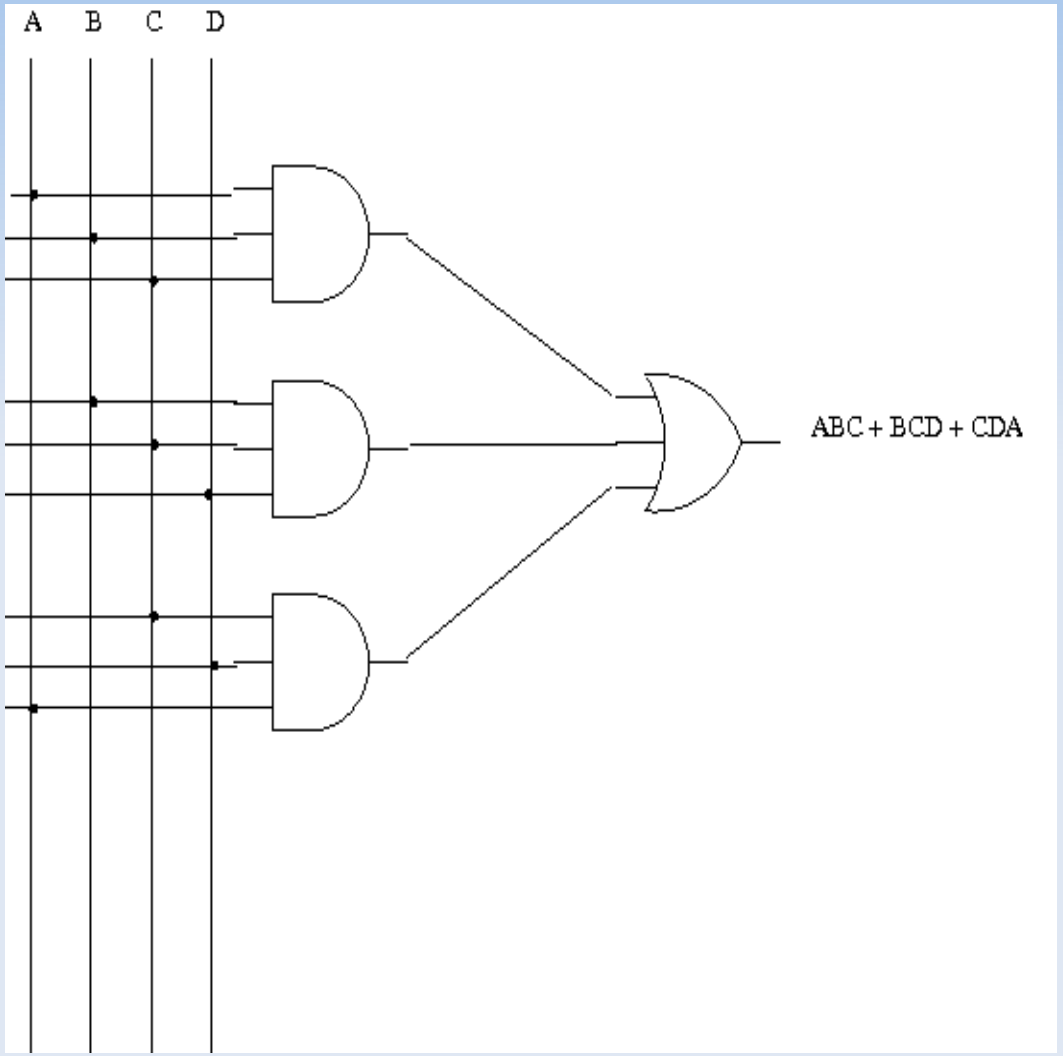
Digital Logic

Logic And Verilog

Gates



The most obvious gates are AND and OR

We can combine them to implement any logic function

Conventions

Zero volts is logic-0

5 volts is logic-1 (that’s old fashioned TTL logic)

Unless we use negative logic

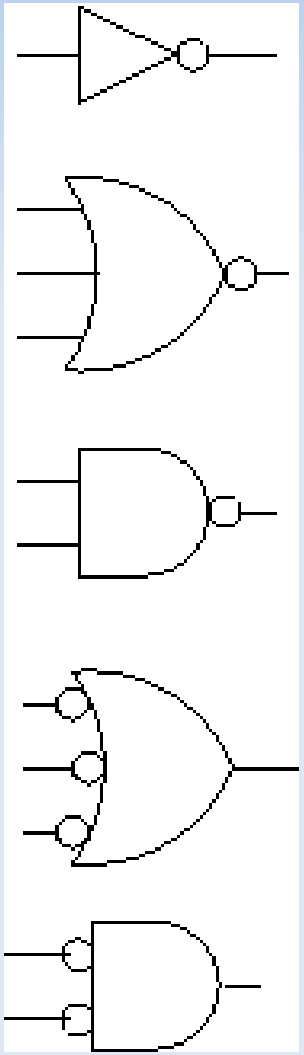
Most computers use smaller voltages now

1.5 volt is used by DDR3 memories

In this case 1.5 volt is logic-1

Due to electrical noise the logic levels are defined by a range.

Other gates



The little circle means ”not”

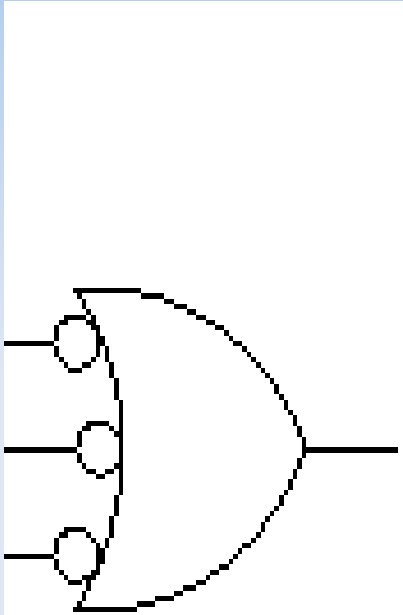
NOR gate (not-OR)

NAND gate

WHAT???

WHAT???

Truth Tables

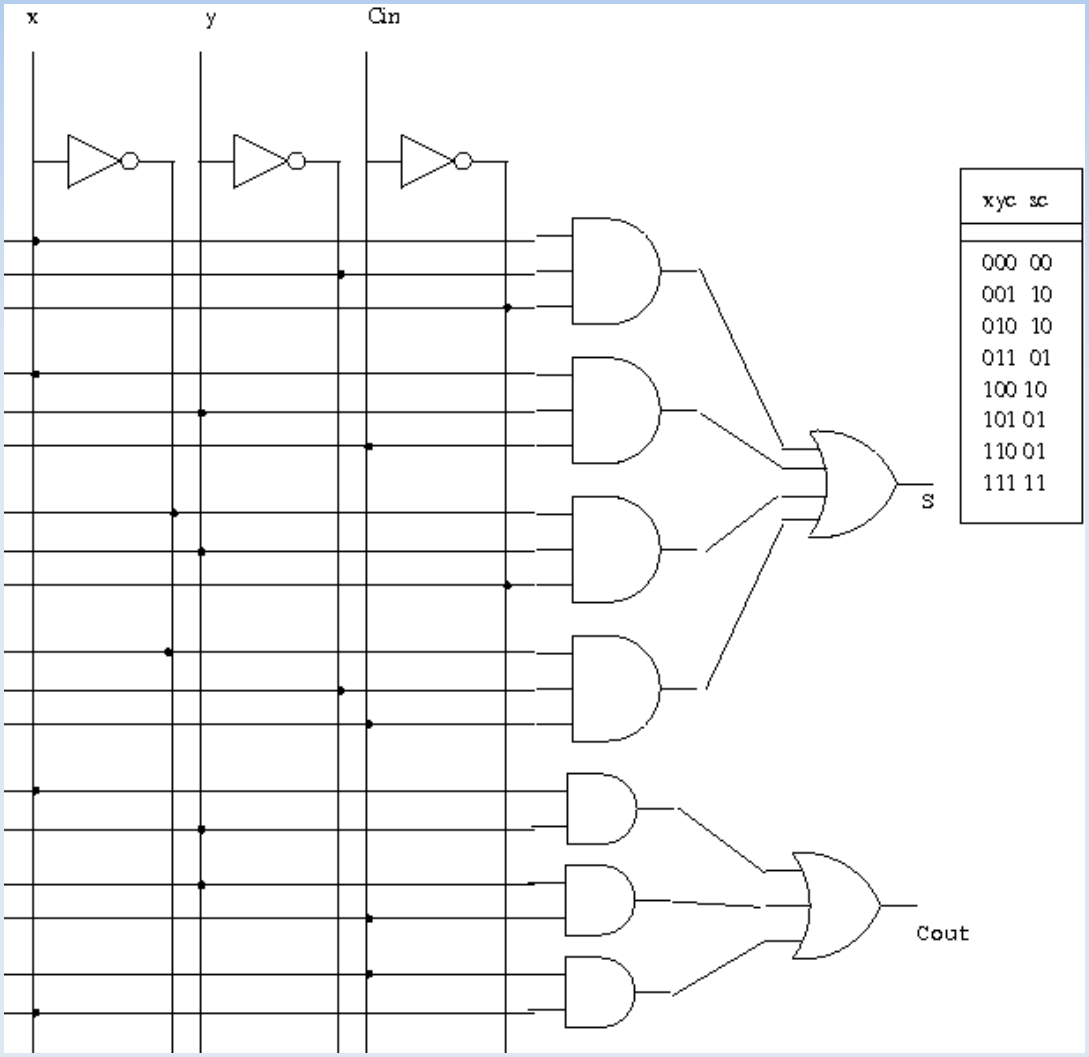


INP	OUT
000	1
001	1
010	1
011	1
100	1
101	1
110	1
111	0

It is the opposite of an AND gate

It is a NAND gate

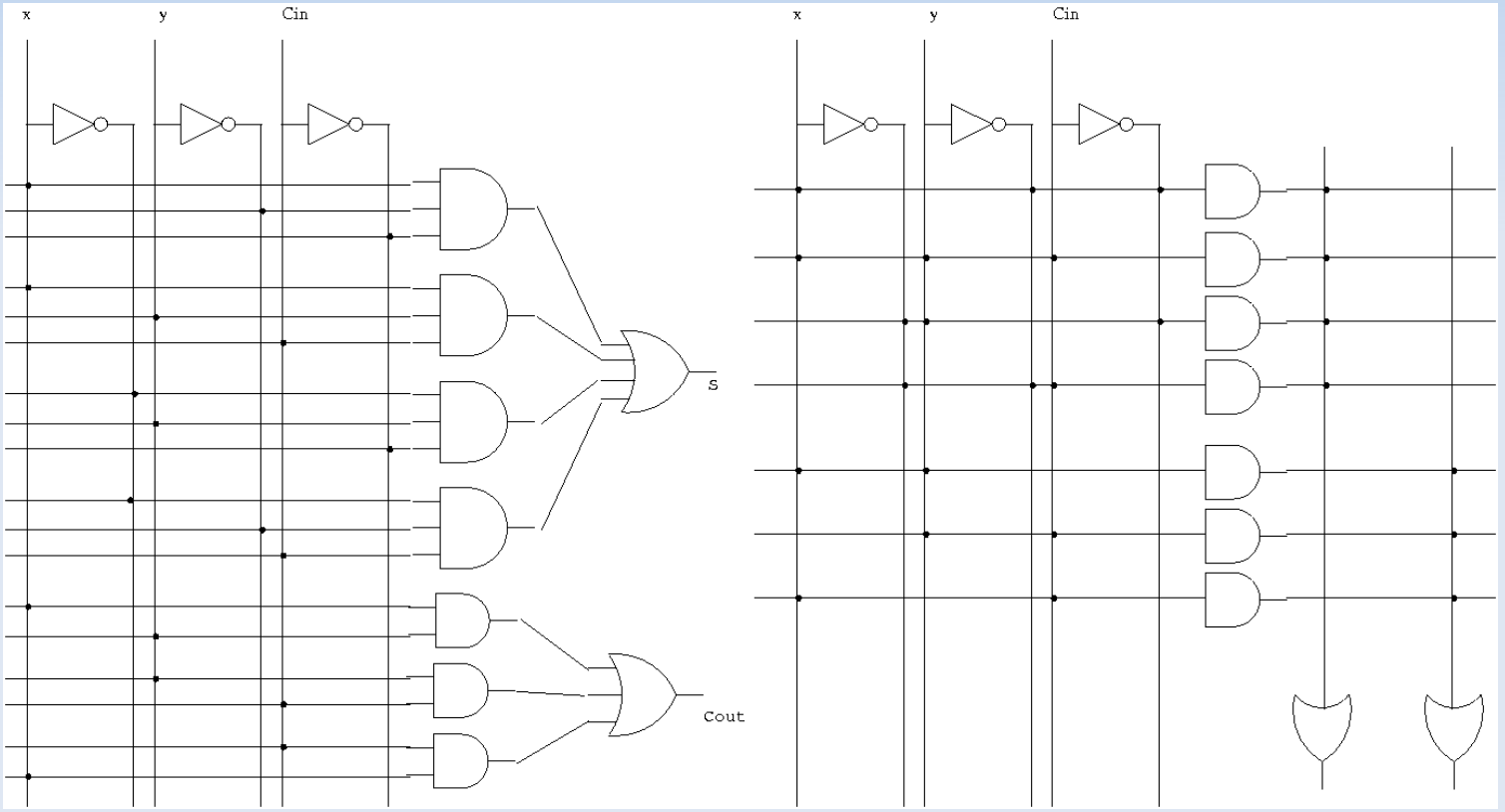
Example



Try to figure out what this does

It is a one bit adder with carry in.

Simpler Drawing



Programmable Logic Arrays

PLA for short

The dots are really fuses inside a chip

Fuses can be programmed once

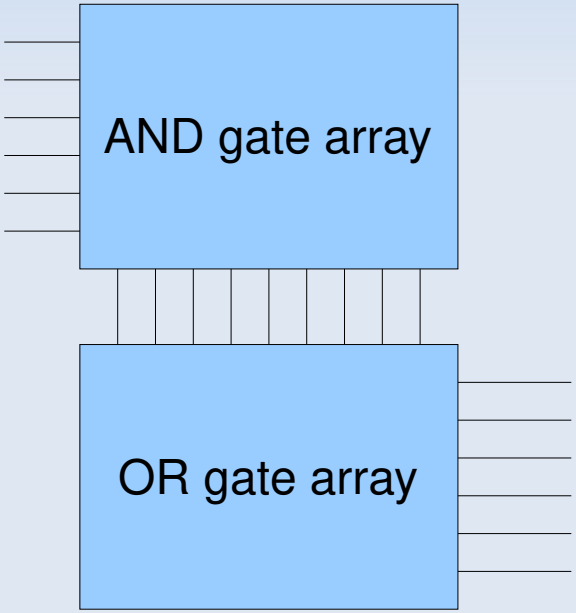
Can implement any logic function

Modern “fuses” can be programmed many times

PLAs on hormones are called

- Field Programmable Gate Arrays (FPGA)

PLAs



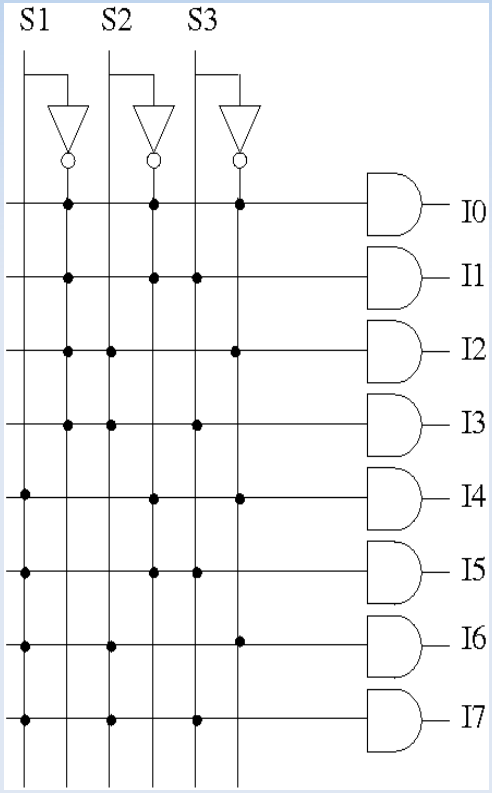
Standard Components

Decoders

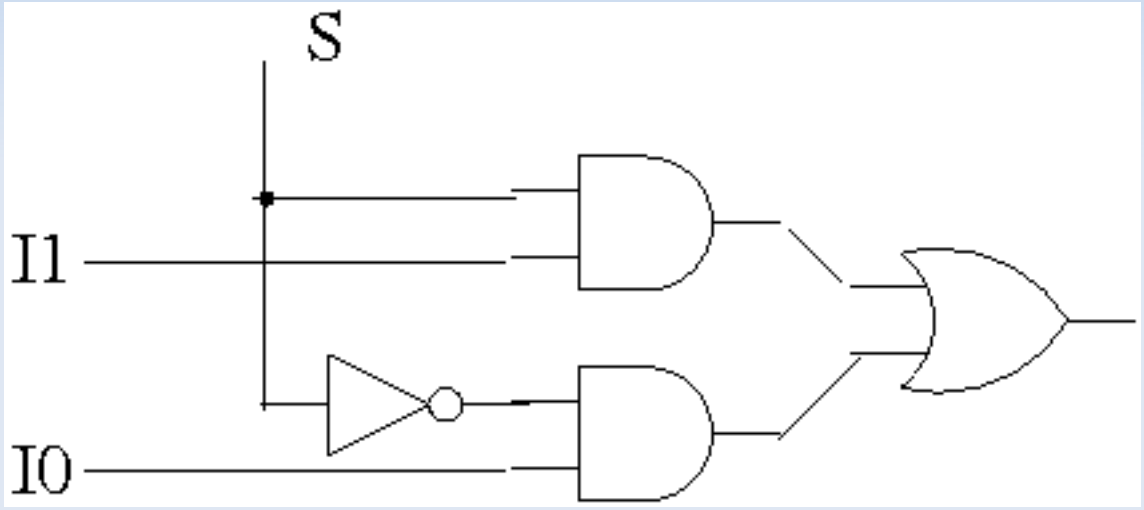
Multiplexers

ROM

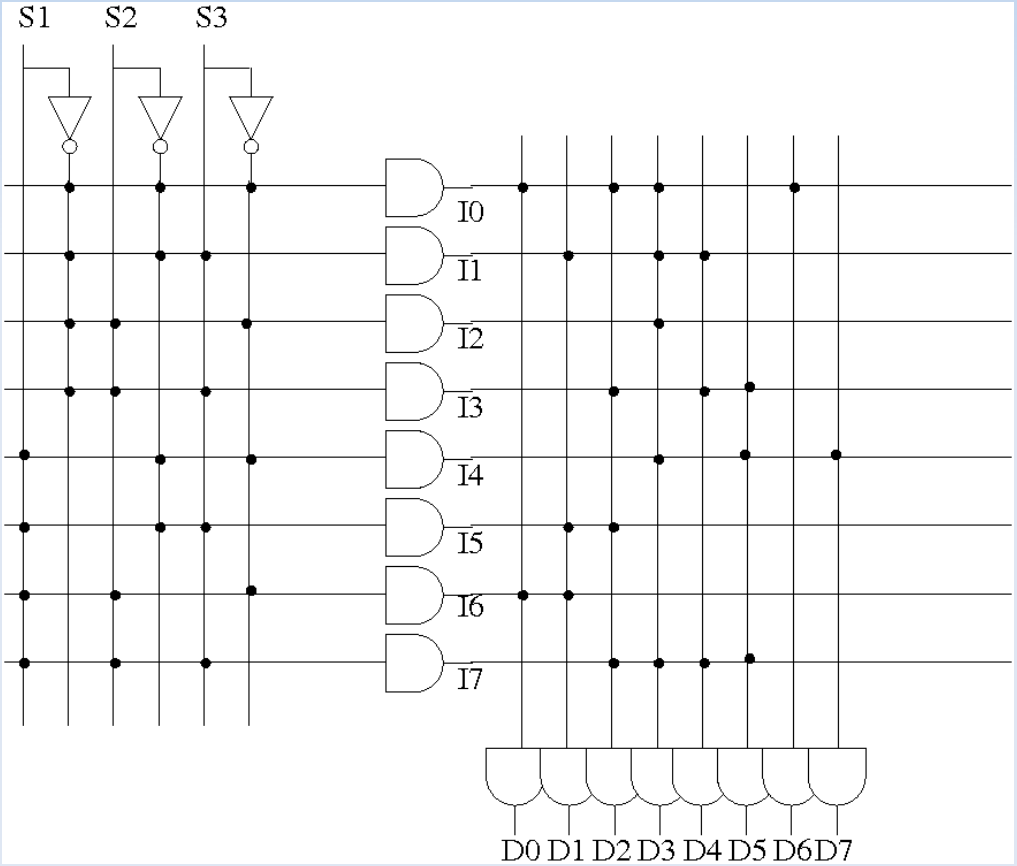
Decoder



Multiplexer



ROM



Boolean Algebra Laws

Identity Law: A+0=A, A*1=A

Zero & One Law: A+1=1, A*0=0

Existence of complement: A+A' = 1, A*A' = 0

Commutative Law: A+B=B+A, A*B=B*A

Associative Law: A+(B+C)=(A+B)+C

A*(B*C)=(A*B)*C

Distributive Law: A*(B+C)=A*B+A*C

A+(B*C)=(A+B)*(A+C) (surprising!)

De Morgan's Law

(A+B)' = A' * B'

(A*B)' = A' + B'

Principle of Duality

AND and OR are symmetric

So is 0 and 1

Optimization

Two different logic expressions can have exactly the same behavior.

Two different expressions with identical behavior may have different cost of implementation

Choosing the cheapest is optimization

May have to satisfy other criteria

- Propagation delay, no glitches, etc

Optimization

AB + AB'

=A(B+B')

=A*1=A

A'B'C + ABC

= (A'B' + AB)C

=((A'B' + A)(A'B' + B))C

(B' + A)(A'+B)C

Half Adder

S = A'B + AB'

C = AB

AB	SC
00	00
01	10
10	10
11	01

Full Adder

$S = A'B'C + AB'C' + A'BC' + ABC$

$Cout = ABC + A'BC + ABC' + AB'C$

$C_{out} = AB + BC + CA$ (optimized)

ABC	SC
000	00
001	10
010	10
011	01
100	10
101	01
110	01
111	11

Don’t Cares

- We use don’t cares when we do not care if for a certain combination of input, the output is true or false.
  - When this input is illegal and does not appear
  - When this input appears only when the output is disabled
- They are very useful when optimizing
  - Try it with both outputs and see which one is cheaper

Verilog

A hardware description language

Can be used to design, optimize and simulate hardware

Started in the mid-80's as a hardware simulation system

Hardware synthesis was added later

Its main competitor is VHDL

What can Verilog do?

Describe a circuit for simulation purposes

Many of the Verilog constructs can be synthesizeable.

Allows the designer to specify

Behavior and/or

Structure

Structure of a Verilog Module

- Contains “initial” constructs
- Parallel blocks called “always” constructs
- Continuous assignments to specify combinational circuits (gates w/o memory)
- Instances of modules

Elements of Verilog

Wire: mathematical abstraction of a real wire

Can have 4 possible values!!

True or 1

False or 0

X: unknown (not yet defined, unconnected etc)

Z: high impedance

Electrically disconnected. A smart trick electronics engineers have invented.

Elements of Verilog

Registers (reg): Are memory elements

The Verilog compiler may map them to actual memory elements (flip-flops)

Same set of possible values

Elements of Verilog

Constants

Can be specified as plain constants like 3, 15, 20...

Often we want to specify the bit-width of a constant

4'b0011 is 4-bit representation of 3

5'b00011 is a 5-bit representation of 3

-4'b0011 is 4-bit representation of -3 (2's compl.)

4'hF is 4-bit representation of 15

Operators in Verilog

+,-,*,/ like C

&, |, ~, ^ again like C

==, !=, <, >, <=, >= like C

<<, >> like C

con?expr1:expr2 like C

Operators in Verilog

But adds to C

Unary &, |, ^

Apply the operator on all bits of the operand

{A,B} the bits of A followed by the bits of B

{x{const}} is {const,const... x times}

Combinational Circuits

A network of gates

Directed graph

There should be no cycles (if there are it is not really combinational)

Output determined exclusively by inputs

Implements logic functions

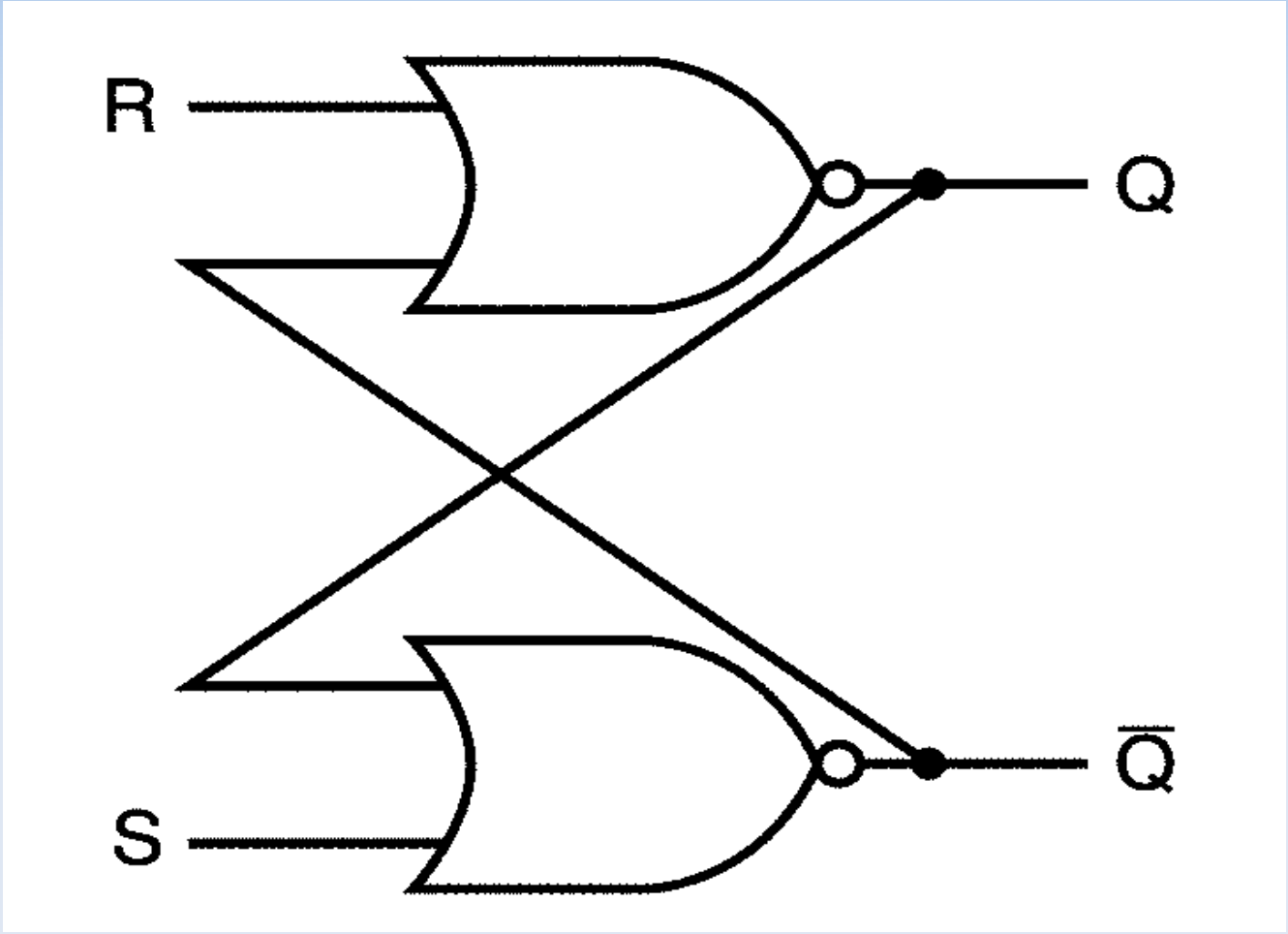
Combinational Circuits

```
module h_adder(A, B, S, Cout)
   input A, B;
   output S, Cout;
   assign S = A^B;
   assign Cout = A&B;
endmodule
```

Combinational Circuits

```
module h_adder(A, B, S, Cout)
   input A, B;
   output S, Cout;
   always @(A, B)
   begin
      S <= A^B;
      Cout <= A&B;
   end
endmodule
```

Memory elements



Memory Elements

We can think of memory elements as combinational circuits with feedback

We would rather think of them as little black boxes

Sometimes memory is implemented using other technologies (capacitors for DRAM)

Combinational Circuits

```
Module half_adder(A,B,Sum,Carry);
  input A,B;
  output Sum, Carry;
  assign Sum = A^B;
  assign Carry = A & &;
endmodule
```

Combinational Circuits

- Use the `assign` keyword
- They represent permanent connections
- The `assign` keyword can specify only combinational circuits
- Combinational circuits can be specified with the `always` construct as well
- The `always` construct can also specify sequential circuits

The always construct

```
Module half_adder(A,B,Sum,Carry)
  input A,B;
  output reg S, C
  always @(A,B) begin
    case ({A,B})
      2'b00: begin S=0; C=0; end;
      2'b01: begin S=1; C=0; end;
      2'b10: begin S=1; C=0; end;
      2'b11: begin S=0; C=1; end;
  end
endmodule
```

Combinational with always

Previous example used always to implement a half-adder

Uses blocking assignments

Pretty much the same as C

If properly defined, most compilers will not use flip-flops to implement it

If all input signals are on sensitivity list

Sequential Circuits

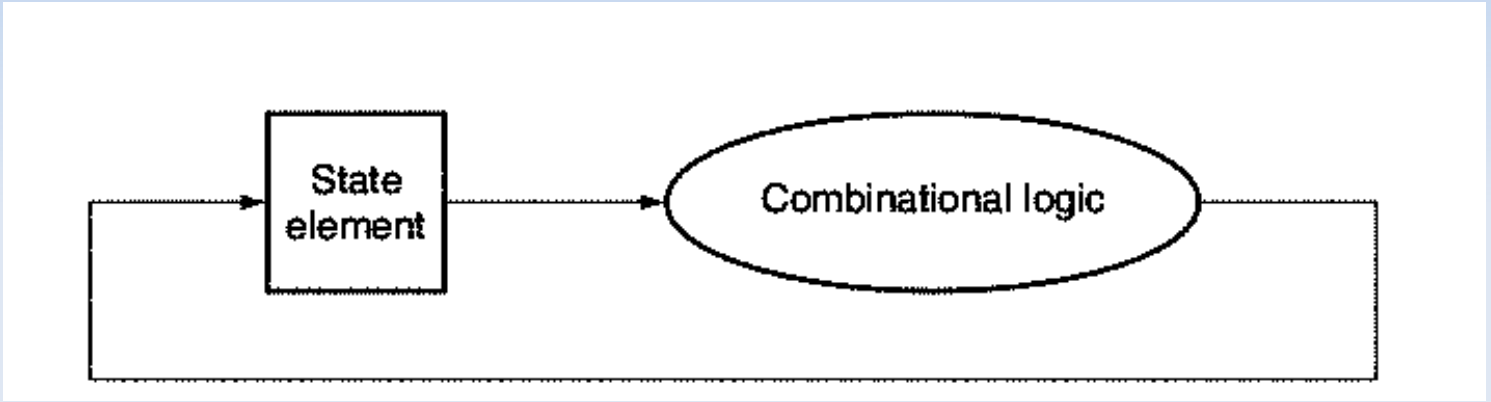
Any circuit that contains memory

If it contains memory then it has “state”

If it has state then the state changes, so it goes through a sequence of states

Hence the name sequential.

Sequential Circuits



Sequential Circuits

How come signals don't rush around the loop uncontrollably?

This is where the “clock” comes in

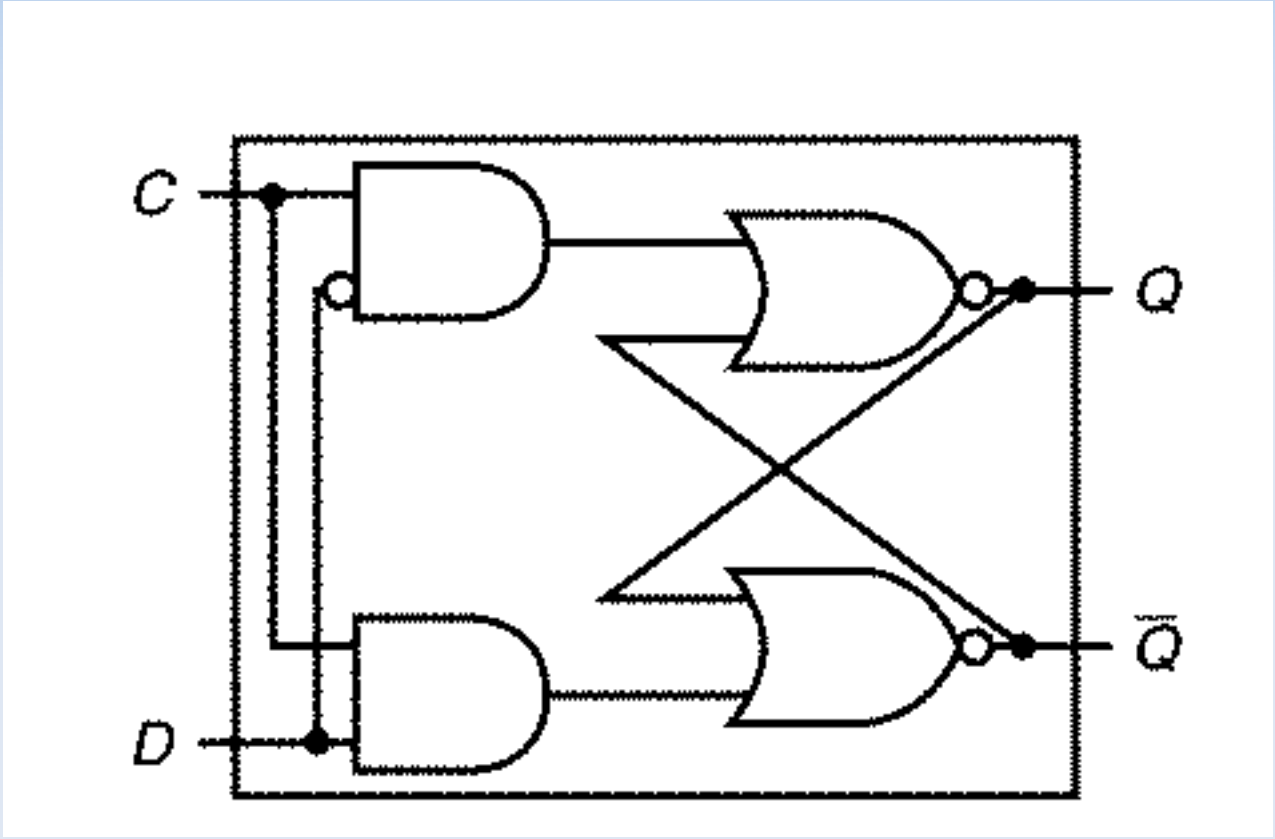
It is the same clock you see on the specs of your CPU

With every clock pulse the signal goes around once

These are called synchronous sequential circuits

There are also asynchronous (we do not deal with them)

Typical Latch



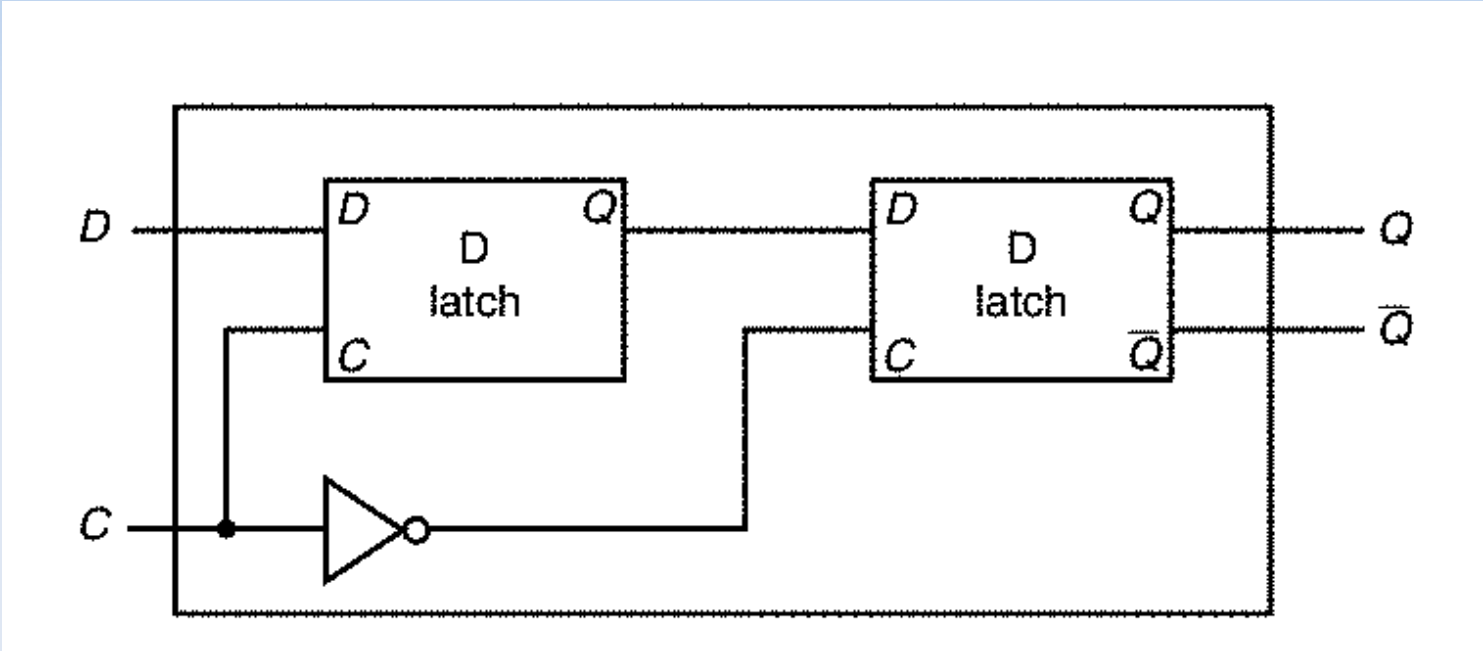
Still....

Unless the width of the clock pulse is wisely selected...

- The signal will travel around more than once

These latches are useful in some cases, but not good enough for our current task

Falling edge trigger FF



Edge triggered D-Flip-Flop

```
Module DFF(clock,D,Q,Qb)
  input clock, D;
  output reg Q;
  output Qb;
  assign Qb = ~Q;
  always @(posedge clock)
    Q <= D;
endmodule
```

Timings

Timing is complex

We use a simplified model

Setup time: time the input to the FF has to be stable before the clock edge

Hold time: time the input has to be stable after the clock edge

Multibit Wires and Registers

```
reg [31:0] regA;
```

`regA[0]` is the LSB;

```
wire [31:0] ALUout;
```

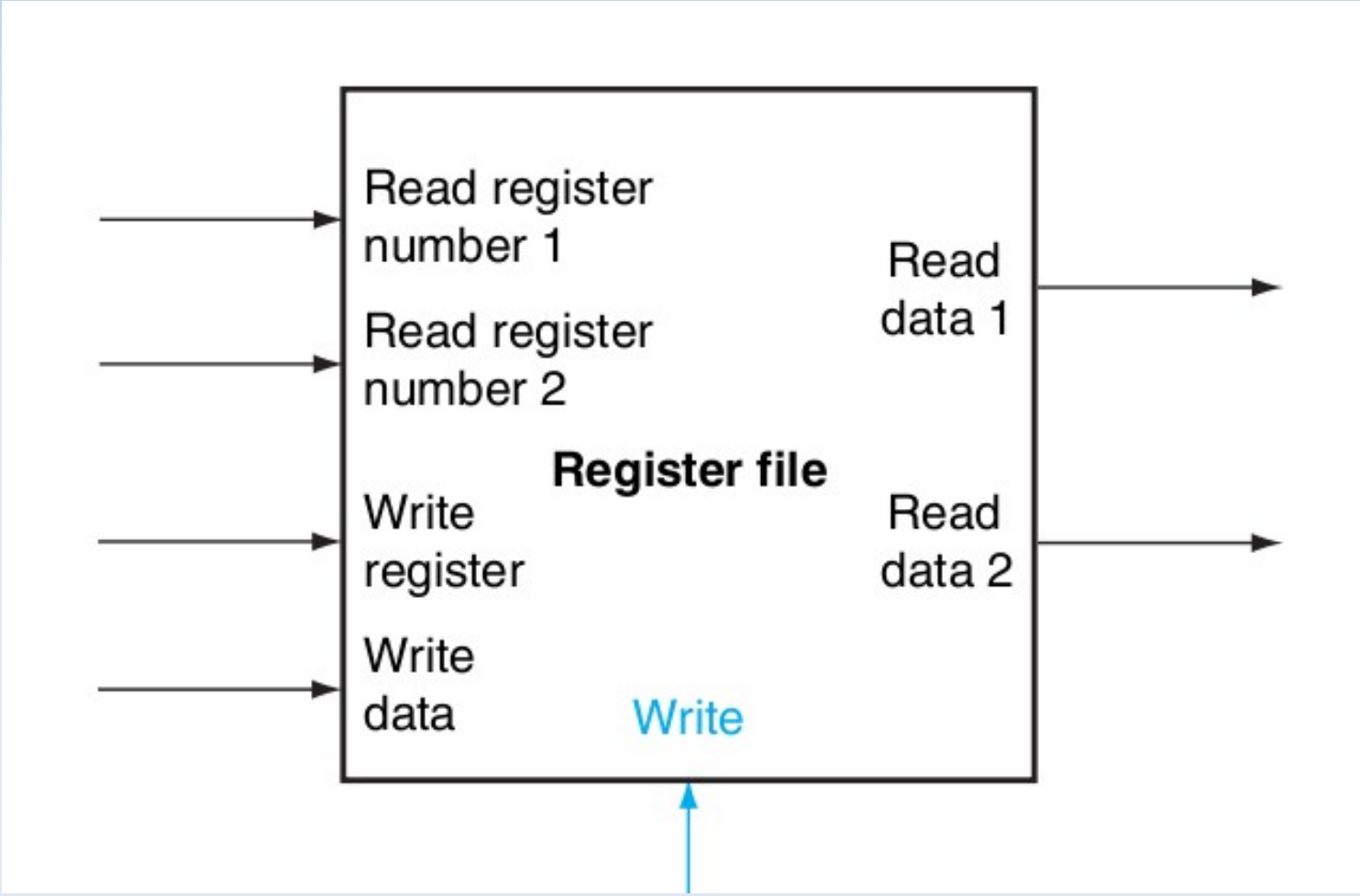
```
reg [31:0] regfile[0:31];
```

`regfile[0]` is the first register in the register file.

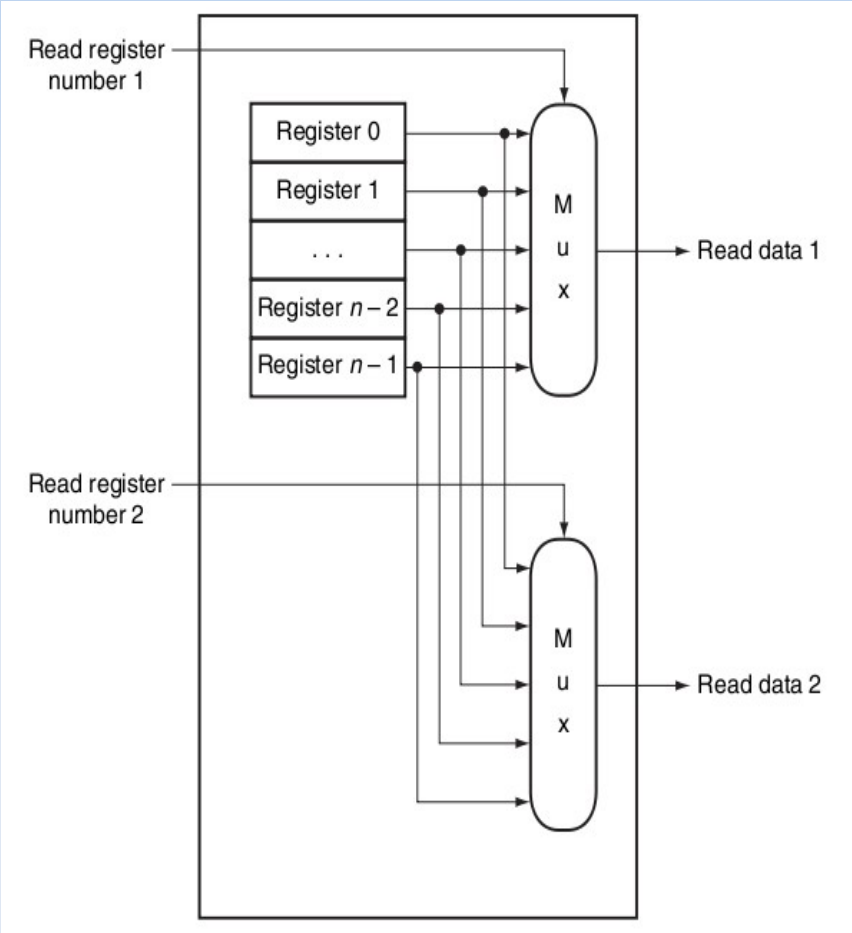
RISC-V ALU

```
module RV_ALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;         input [63:0] A,B;
   output reg [63:0] ALUOut;   output Zero;
   assign Zero = (ALUOut==0);   //Zero is true if ALUOut is 0
   always @(ALUctl, A, B) begin //reevaluate if these change
     case (ALUctl)
       0:  ALUOut <= A & B;
       1:  ALUOut <= A | B;
       2:  ALUOut <= A + B;
       6:  ALUOut <= A - B;
       7:  ALUOut <= A < B ? 1 : 0;
       12: ALUOut <= ~(A | B); // result is nor
       default: ALUOut <= 0;
     endcase
   end
endmodule
```

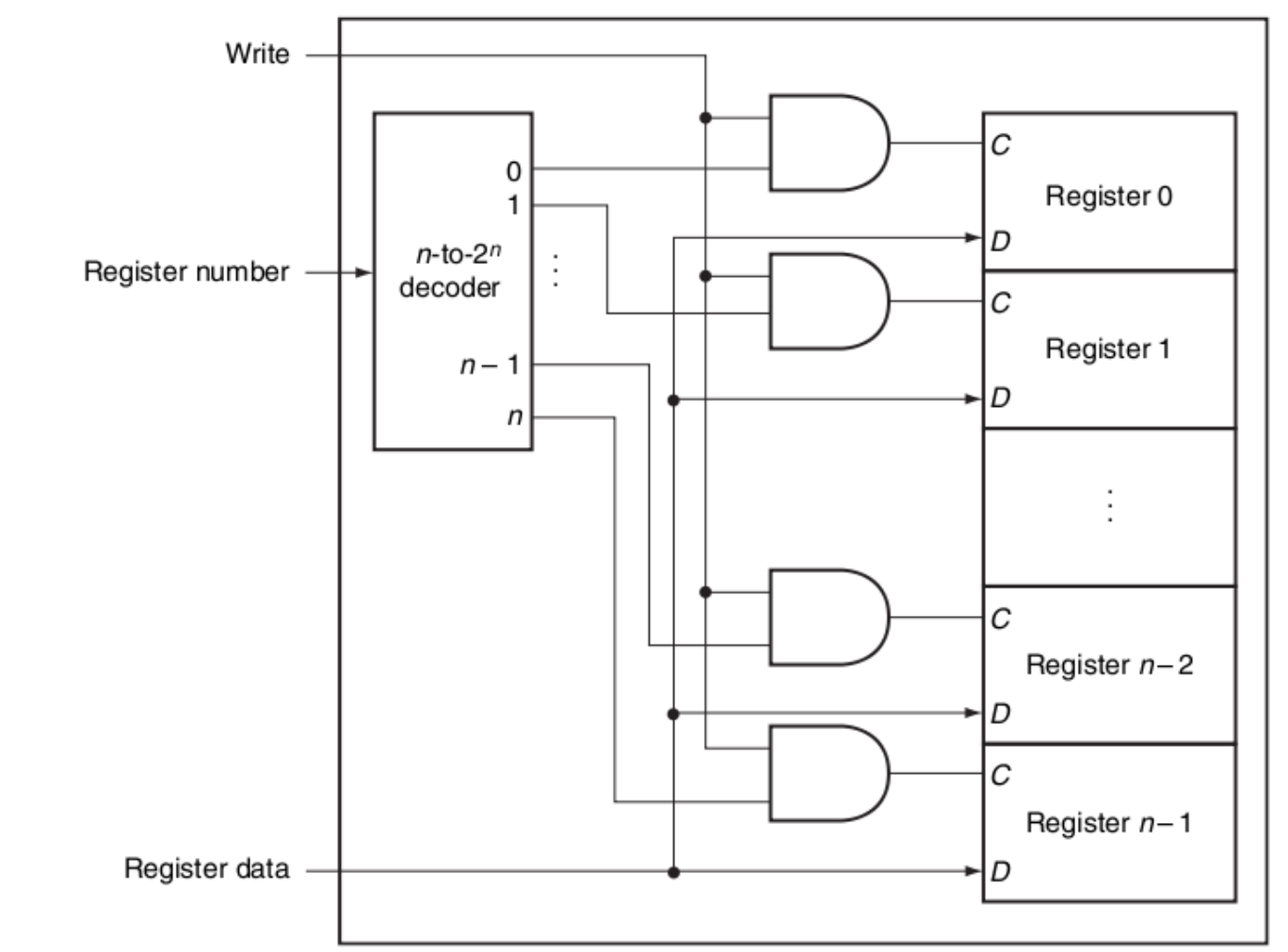
Register File



Register File: read



Register File: write



Register File: Verilog

```
module rfile(R1,R2,W,WD,Wctl,RD1,RD2,clock)
  input [4:0] R1,R2,W;   // Select what to read/write
  input [63:0] WD;
  input Wctl, clock;
  output [63:0] RD1,RD2;
  reg [63:0] RF[31:0];
  assign RD1 = RF[R1];
  assign RD2 = RF[R2];
  always @(posedge clock)
    if (Wctl) RF[W] <= WD;
endmodule
```

Specifying Gates

Verilog allows the designer to specify individual gates

Can be bulky

Similar syntax can be used for user defined modules

Half Adder

```
module HA(A,B,S,C)
  input A, B;
  output S, C;
  wire Bn, An, Abn, AnB;
  not N1(An,A);
  not N2(Bn,B);
  and (Abn,A,Bn);
  and (AnB,An,B);
  or  (S,ABn,AnB);
  and (C,A,B);
endmodule
```

Speeding Up Addition

- Carry propagation is what slows down addition
- Sometimes the LSB of input will affect the MSB or the carry out
- We design for the worst case scenario
- The simpler adders are called ripple adders

Carry LookAhead

a0, a1, a2, etc; b0, b1, b2, etc are the inputs

c0, c1, c2 are the carries.

$c1 = b0\ c0 + a0\ c0 + a0\ b0$

$c1 = a0\ b0 + c0\ (a0 + b0)$

$c1 = g0 + c0\ p0$

$g0 = a0\ b0;\ p0 = a0 + b0;$

Carry LookAhead

Define

$$g_i = a_i \, b_i$$

$$p_i = a_i + b_i$$

Then

$$c_{i+1} = g_i + p_i \, c_i$$

Carry LookAhead

$c1 = g0 + p0\ c0$

$c2 = g1 + p1\ g0 + p1\ p0\ c0$

$c3 = g2 + p2\ g1 + p2\ p1\ g0 + p2\ p1\ p0\ c0$

And…

$c4 = g3 + p3\ g2 + p3\ p2\ g1 + p3\ p2\ p1\ g0 +$

$p3\ p2\ p1\ p0\ c0$

$c4 = G + P\ c0$

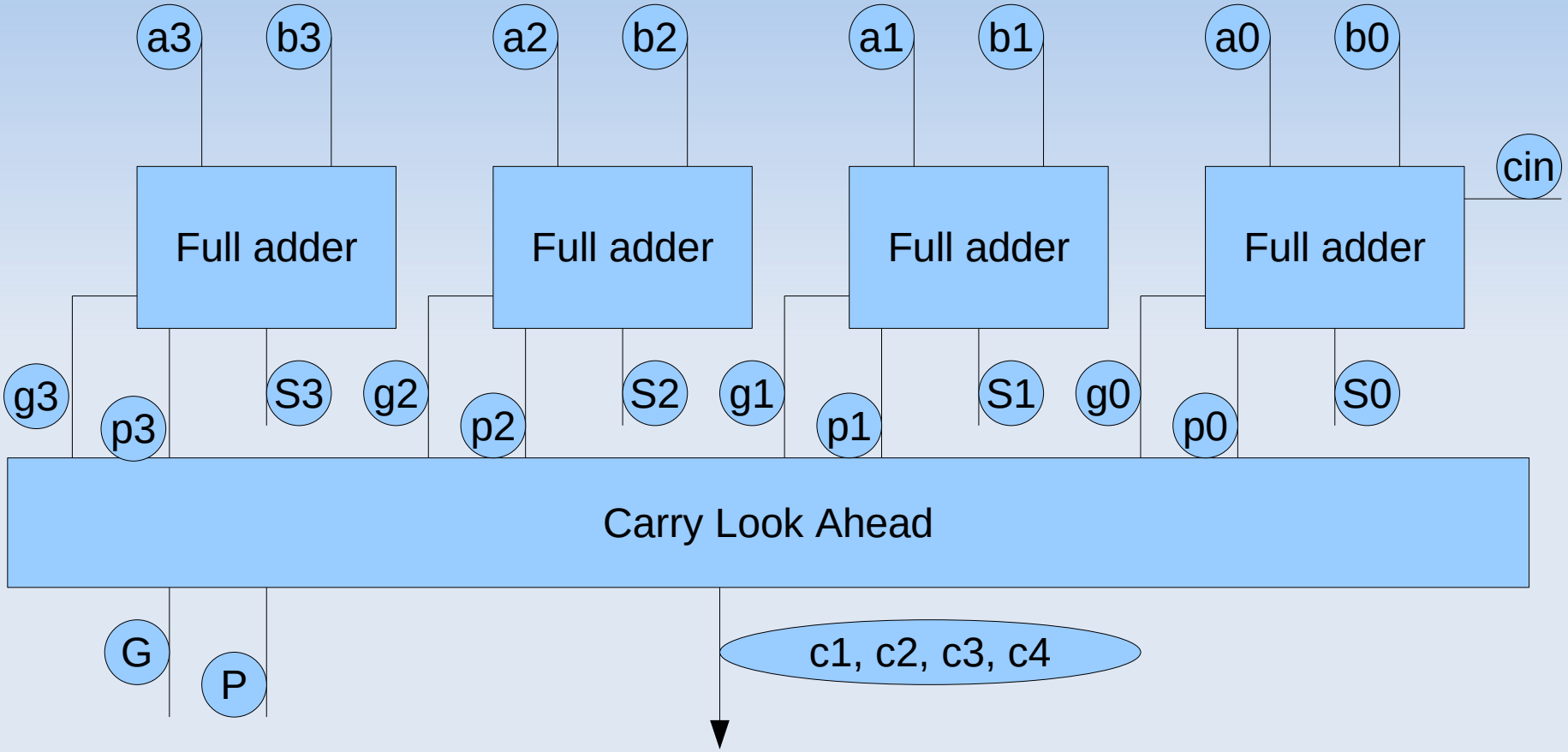
Where $G = g3 + p3\ g2 + p3\ p2\ g1 + p3\ p2\ p1\ g0$

And $P = p3\ p2\ p1\ p0$

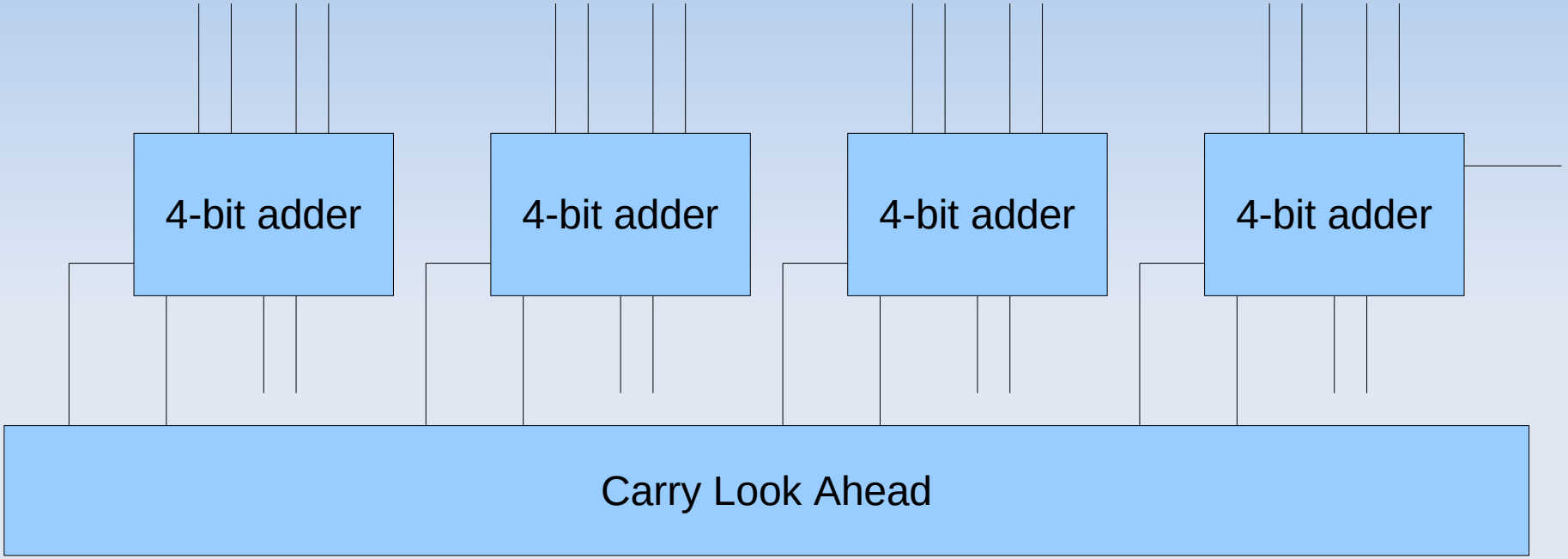
Delays

- For p0, p1, p2, p3 …….. 1
- For g0, g1, g2, g3 …….. 1
- For c1, c2, c3 …….. 3
- For c4 …….. 3 or 4
- For P …….. 2
- For G …….. 3

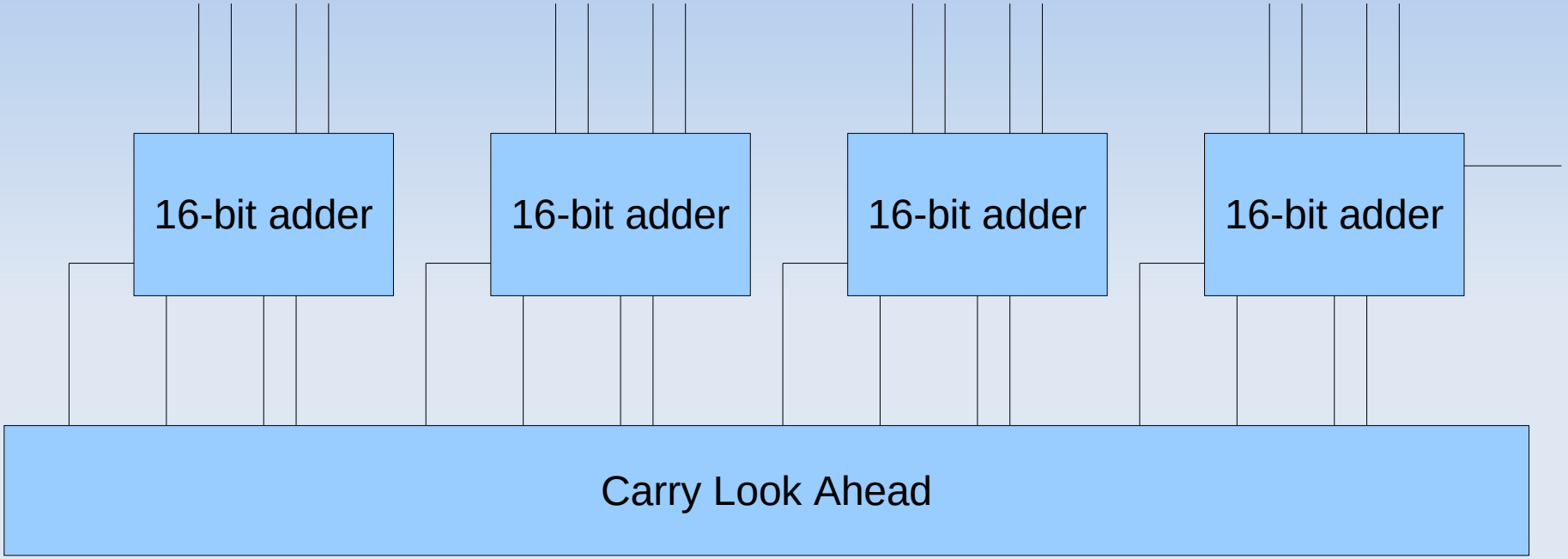
4-bit Adder


a3
b3
a2
b2
a1
b1
a0
b0
cin
Full adder
Full adder
Full adder
Full adder
g3
p3
S3
g2
p2
S2
g1
p1
S1
g0
p0
S0
Carry Look Ahead
G
P
c1, c2, c3, c4

16-bit Adder


4-bit adder
4-bit adder
4-bit adder
4-bit adder
Carry Look Ahead

64-bit Adder


16-bit adder
16-bit adder
16-bit adder
16-bit adder
Carry Look Ahead