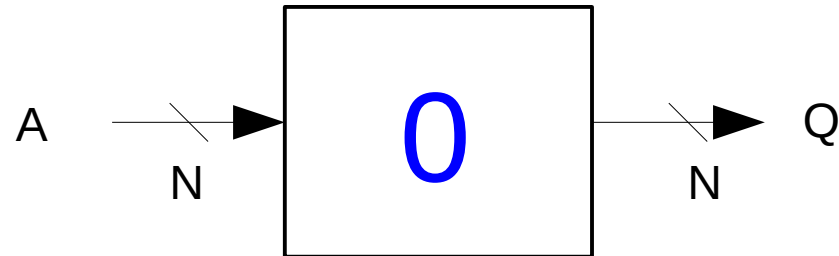


# Sequential circuits

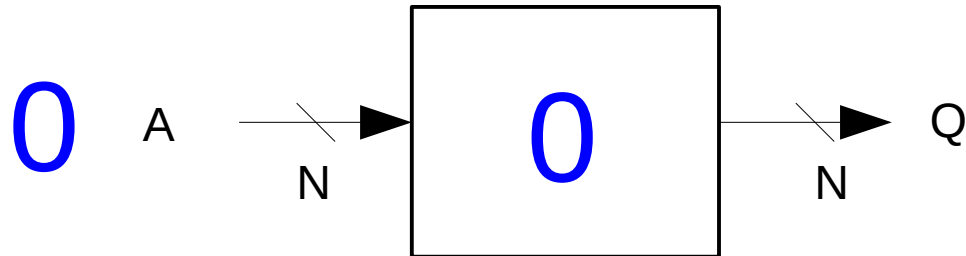
A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1		
2		
3		
4		
5		



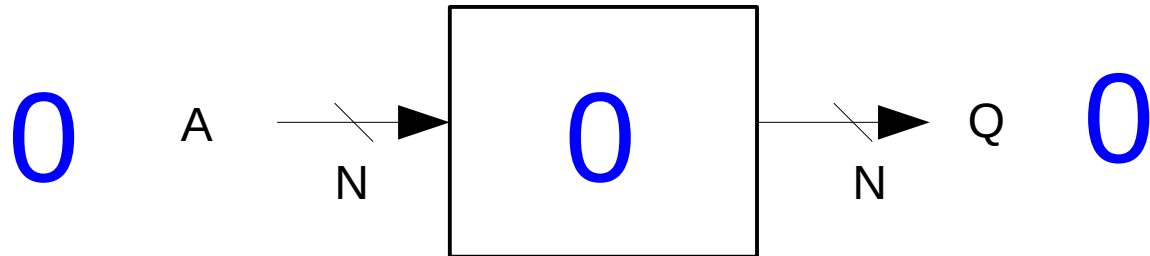
A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1	0	
2		
3		
4		
5		



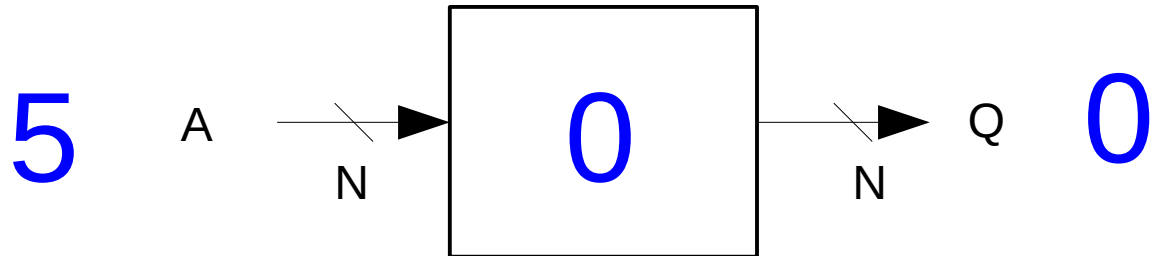
A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1	0	0
2		
3		
4		
5		



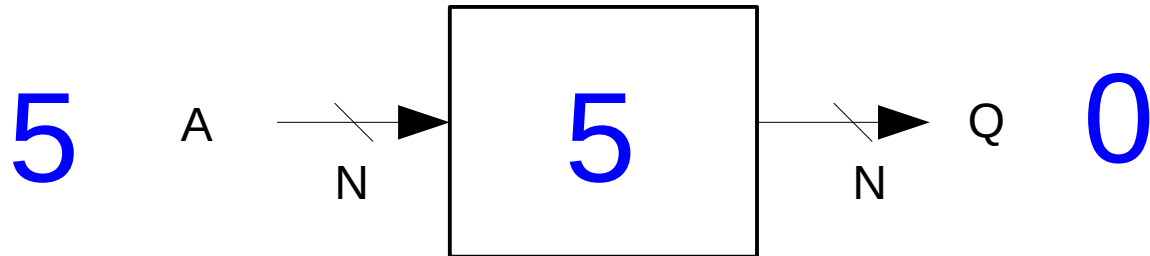
A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1	0	0
2	5	
3		
4		
5		



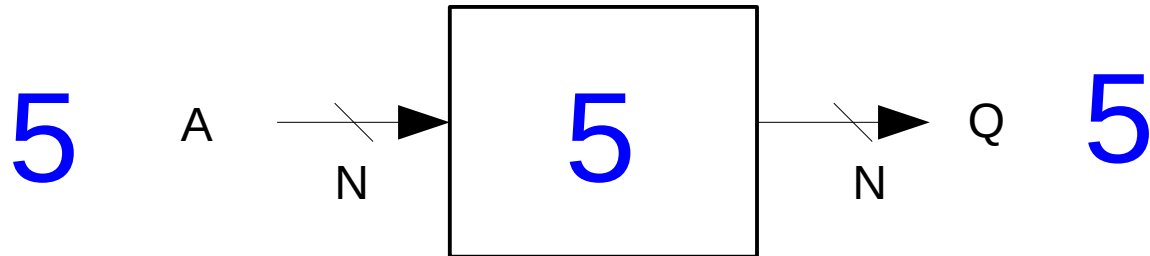
A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1	0	0
2	5	
3		
4		
5		



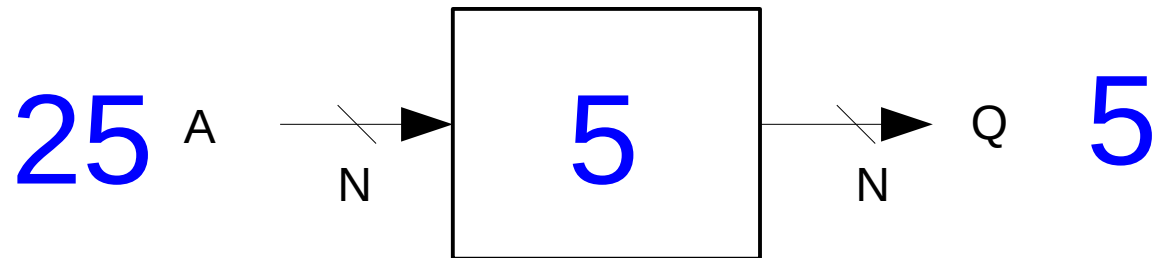
A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1	0	0
2	5	5
3		
4		
5		



A machine to sum a sequence of  $n$ -bit values

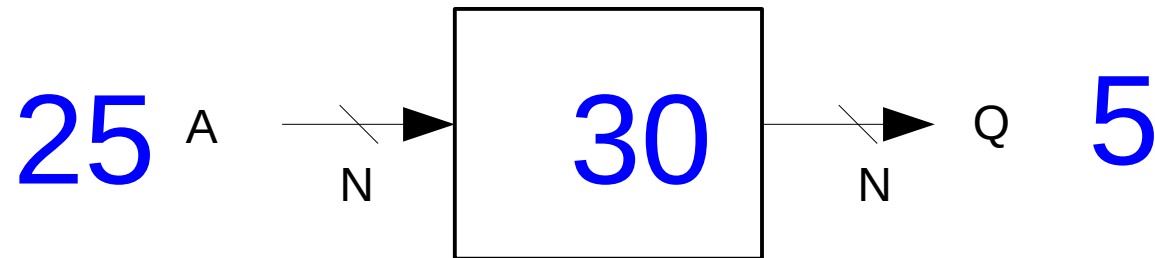
Time	A	Q
1	0	0
2	5	5
3	25	
4		
5		





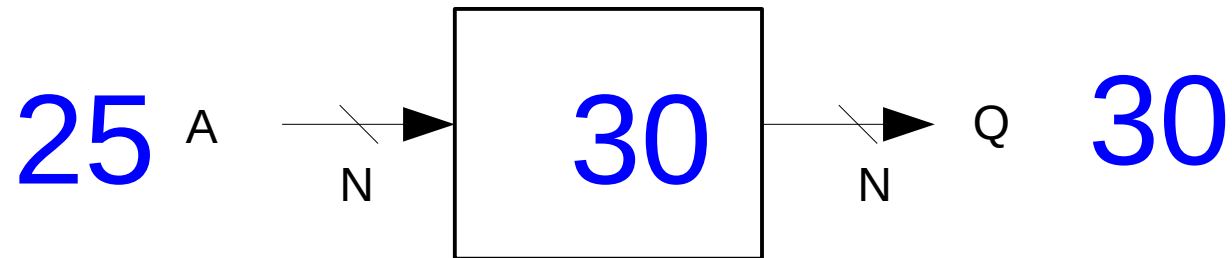
A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1	0	0
2	5	5
3	25	
4		
5		



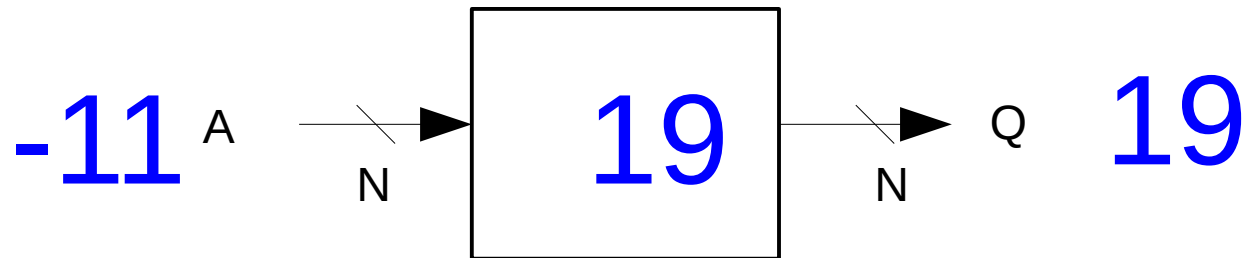
A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1	0	0
2	5	5
3	25	30
4		
5		



A machine to sum a sequence of  $n$ -bit values

Time	A	Q
1	0	0
2	5	5
3	25	30
4	-11	19
5		



## A machine to sum a sequence of $n$ -bit values

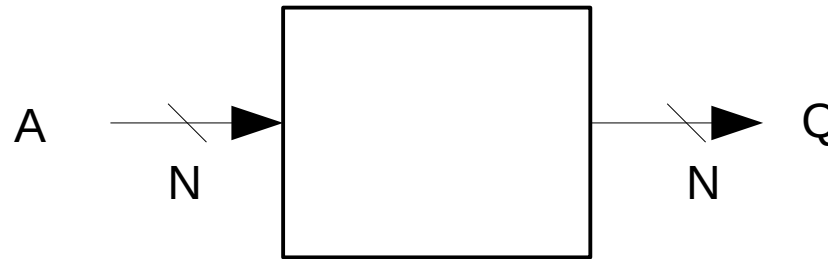
Time	A	Q
1	0	0
2	5	5
3	25	30
4	-11	19
5	0	19



$A$ :  $n$ -bit input  
 $Q$ :  $n$ -bit output

## A machine to sum a sequence of $n$ -bit values

Time	A	Q
1	0	0
2	5	5
3	25	30
4	-11	19
5	0	19



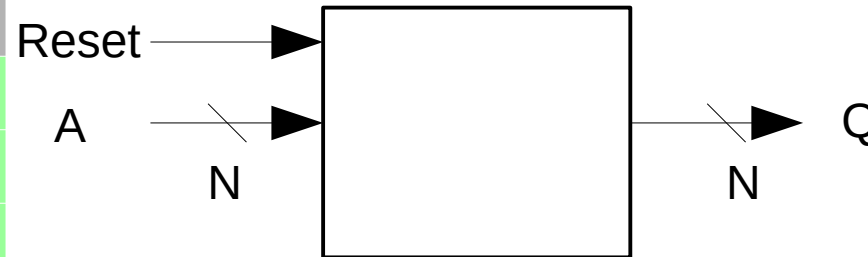
A:  $n$ -bit input  
Q:  $n$ -bit output

Some questions:

- How is the initial value of the accumulator set?
- How does the circuit know when the next input is ready?
- How does the circuit store the running (accumulated) sum?
- How can we express this circuit in Verilog?

## A machine to sum a sequence of $n$ -bit values

Time	A	Reset	Q
0	0	0	?
1	0	1	0
2	5	0	5
3	25	0	30
4	-11	0	19
5	0	0	19



A:  $n$ -bit input  
R: 1-bit input  
Q:  $n$ -bit output

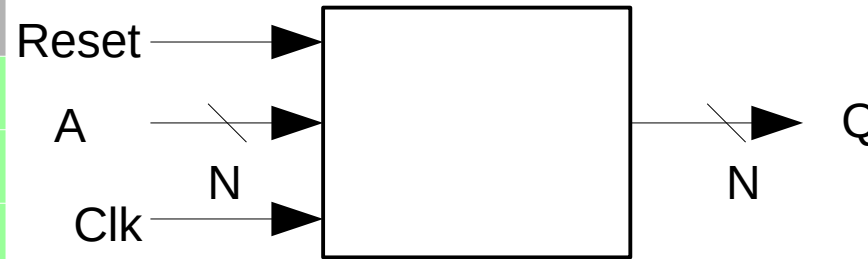
Some questions:

- **How is the initial value of the accumulator set?**
- How does the circuit know when the next input is ready?
- How does the circuit store the running (accumulated) sum?
- How can we express this circuit in Verilog?

We need to add a Reset input. This will usually be 0, but when it's one, the internal counter will be reset. Prior to using Reset, the value of the internal counter is undefined!

## A machine to sum a sequence of $n$ -bit values

Time	A	Reset	Q
0	0	0	?
1	0	1	0
2	5	0	5
3	25	0	30
4	-11	0	19
5	0	0	19



A:  $n$ -bit input  
R: 1-bit input  
Clk: 1-bit input  
Q:  $n$ -bit output

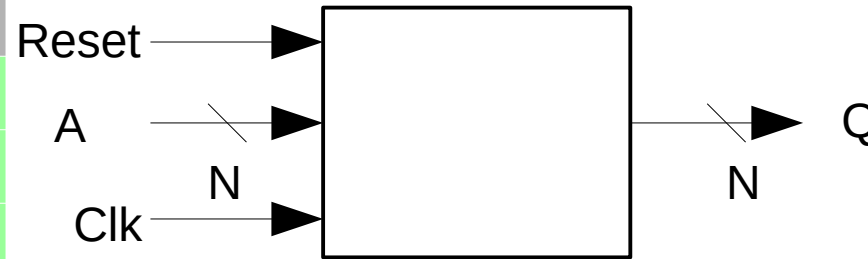
Some questions:

- How is the initial value of the accumulator set?
- **How does the circuit know when the next input is ready?**
- How does the circuit store the running (accumulated) sum?
- How can we express this circuit in Verilog?

We need to add a Clk (clock) input. The input will go to 1 at a regular interval (the *clock frequency*) and at that time, the circuit will read the current value on A.

## A machine to sum a sequence of $n$ -bit values

Time	A	Reset	Q
0	0	0	?
1	0	1	0
2	5	0	5
3	25	0	30
4	-11	0	19
5	0	0	19



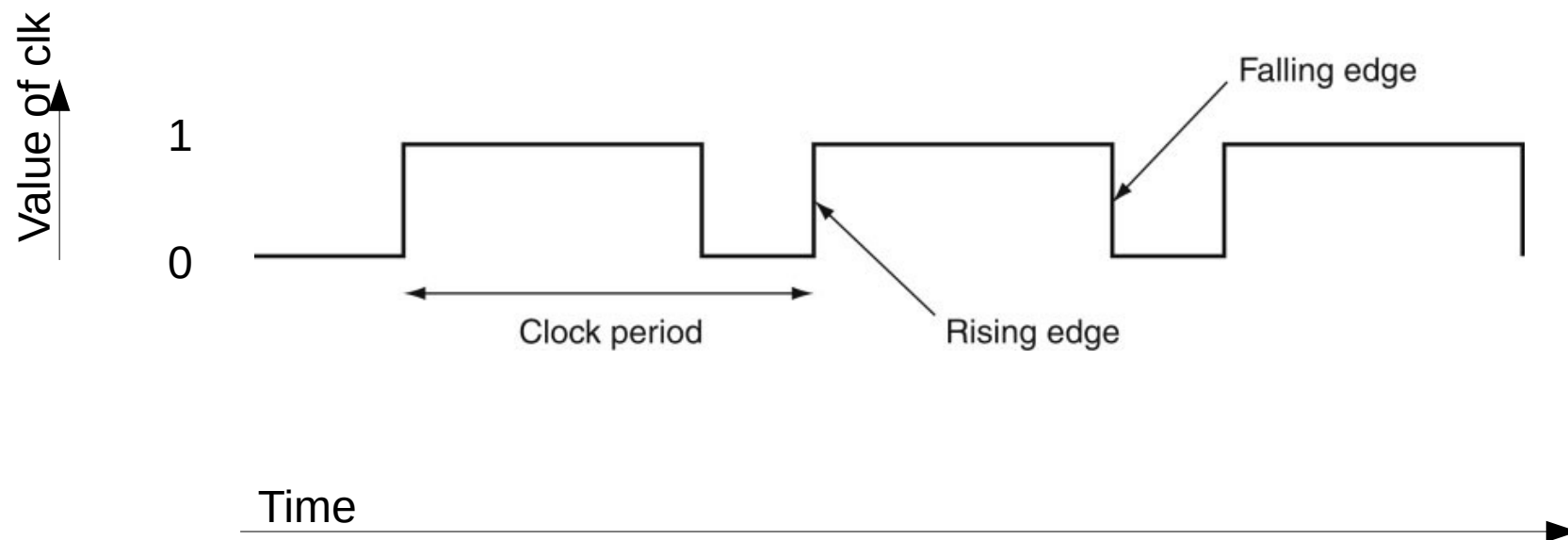
A:  $n$ -bit input  
R: 1-bit input  
Clk: 1-bit input  
Q:  $n$ -bit output

Some questions:

- How is the initial value of the accumulator set?
- How does the circuit know when the next input is ready?
- **How does the circuit store the running (accumulated) sum?**
- How can we express this circuit in Verilog?

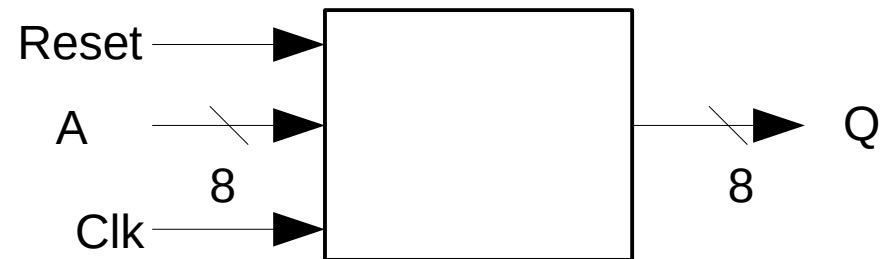
It uses a *register*, a component that can store a bit value. An array of  $n$  registers can store an  $n$ -bit number: this is an  $n$ -bit register. You can write a value into a register, and subsequently read a stored value.





## A machine to sum a sequence of $n$ -bit values

```
module summer(clk, Reset, A, Q);  
    input clk;  
    input Reset;  
    input [7:0] A;  
    output [7:0] Q;  
    reg [7:0] Value;  
    assign Q = Value;  
  
    always @(posedge clk)  
        if (Reset)  
            Value <= 8'b0;  
        else  
            Value <= Value + A;  
  
endmodule
```



### Some questions:

- How is the initial value of the accumulator set?
- How does the circuit know when the next input is ready?
- How does the circuit store the running (accumulated) sum?
- **How can we express this circuit in Verilog?**

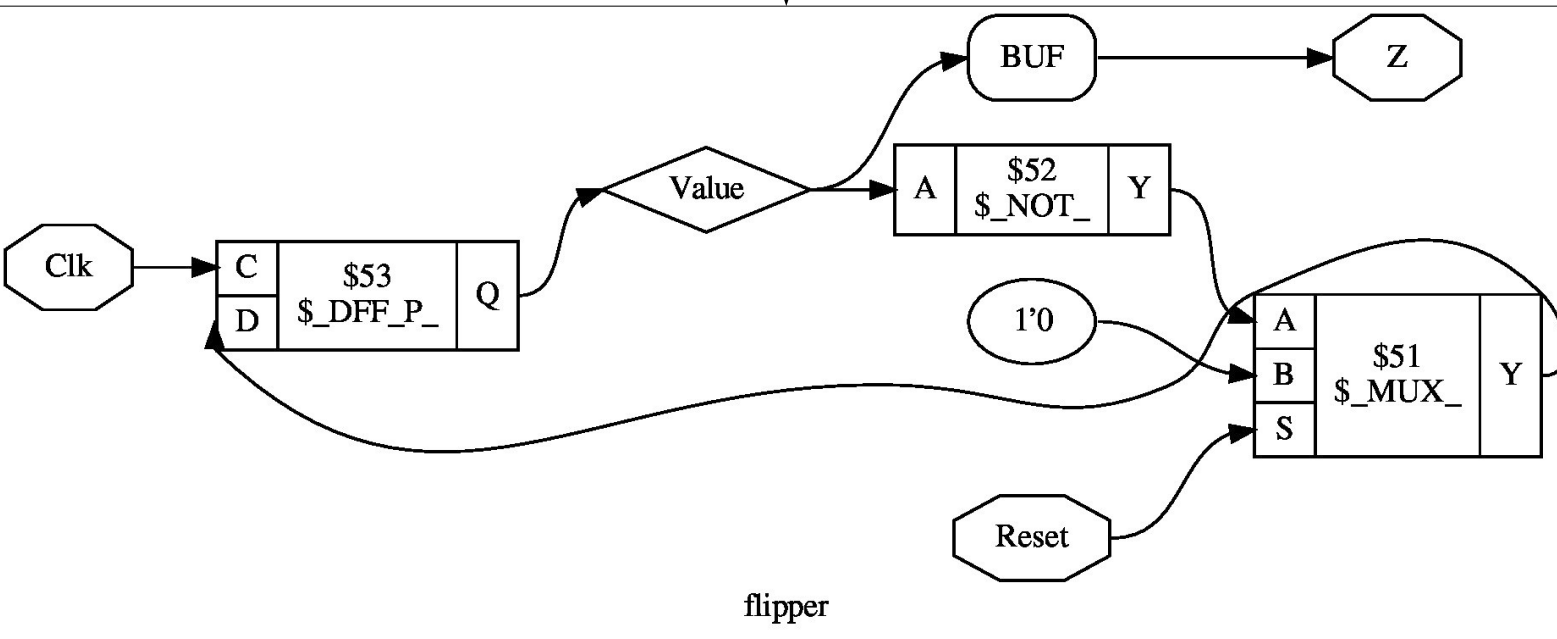
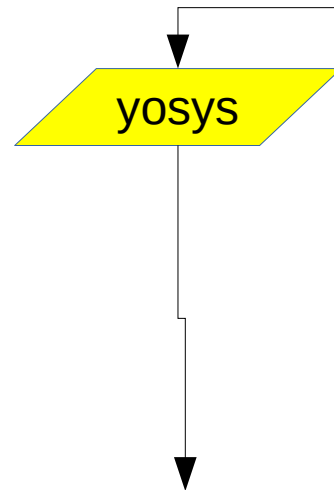
We define registers using the **reg** syntax.  
We store values into registers with the **<=** syntax.  
We define actions to be taken on a clock signal with the **always @(posedge clk)** syntax.

# Synthesis

```
module flipper(Clk, Reset, Z);  
  
    input Clk, Reset;  
    reg Value;  
    output Z;  
  
    always @(posedge Clk)  
    begin  
        if (Reset)  
            Value <= 1'b0;  
        else  
            Value <= ~Value;  
        end  
  
        assign Z = Value;  
    endmodule
```

# Synthesis

```
module flipper(Clk, Reset, Z);  
  
    input Clk, Reset;  
    reg Value;  
    output Z;  
  
    always @(posedge Clk)  
    begin  
        if (Reset)  
            Value <= 1'b0;  
        else  
            Value <= ~Value;  
        end  
  
    assign Z = Value;  
endmodule
```



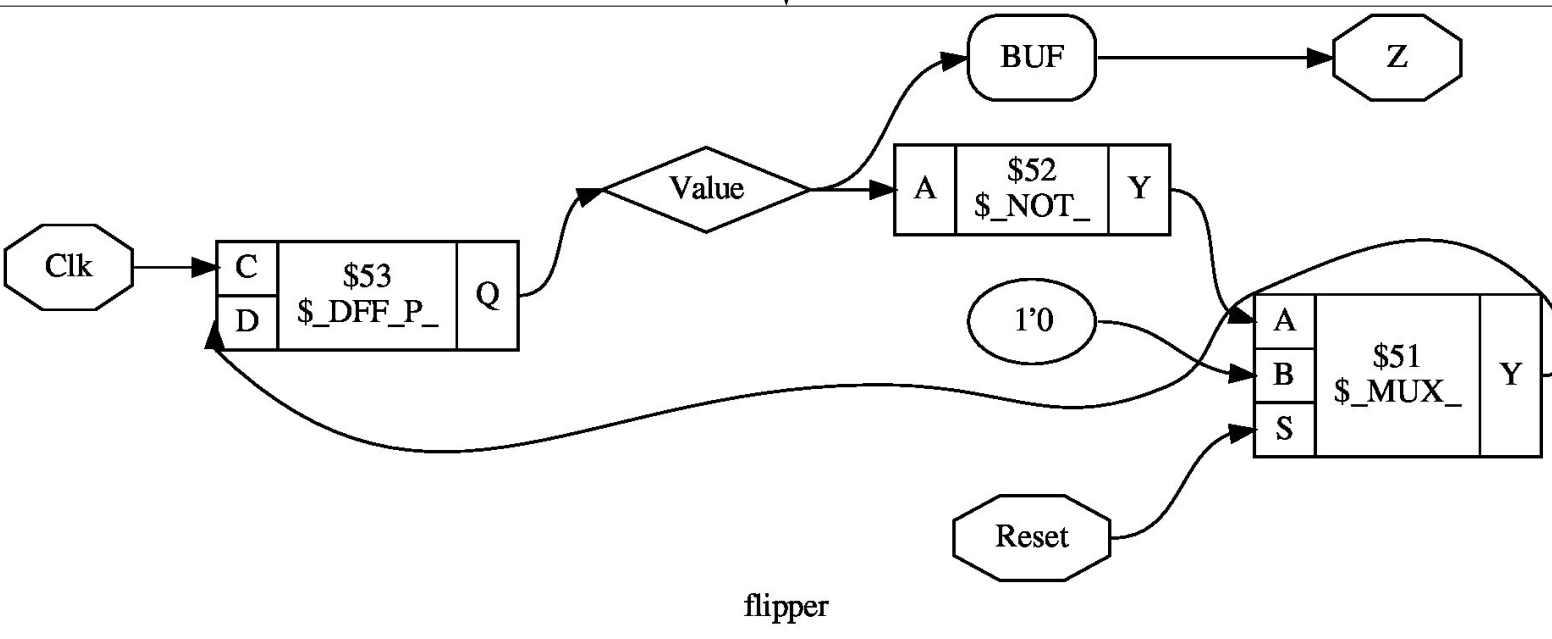
# Synthesis

Sequential Verilog, including registers and register assignment, can be synthesized into gates.

It is **not** important that you understand these low-level components.

yosys

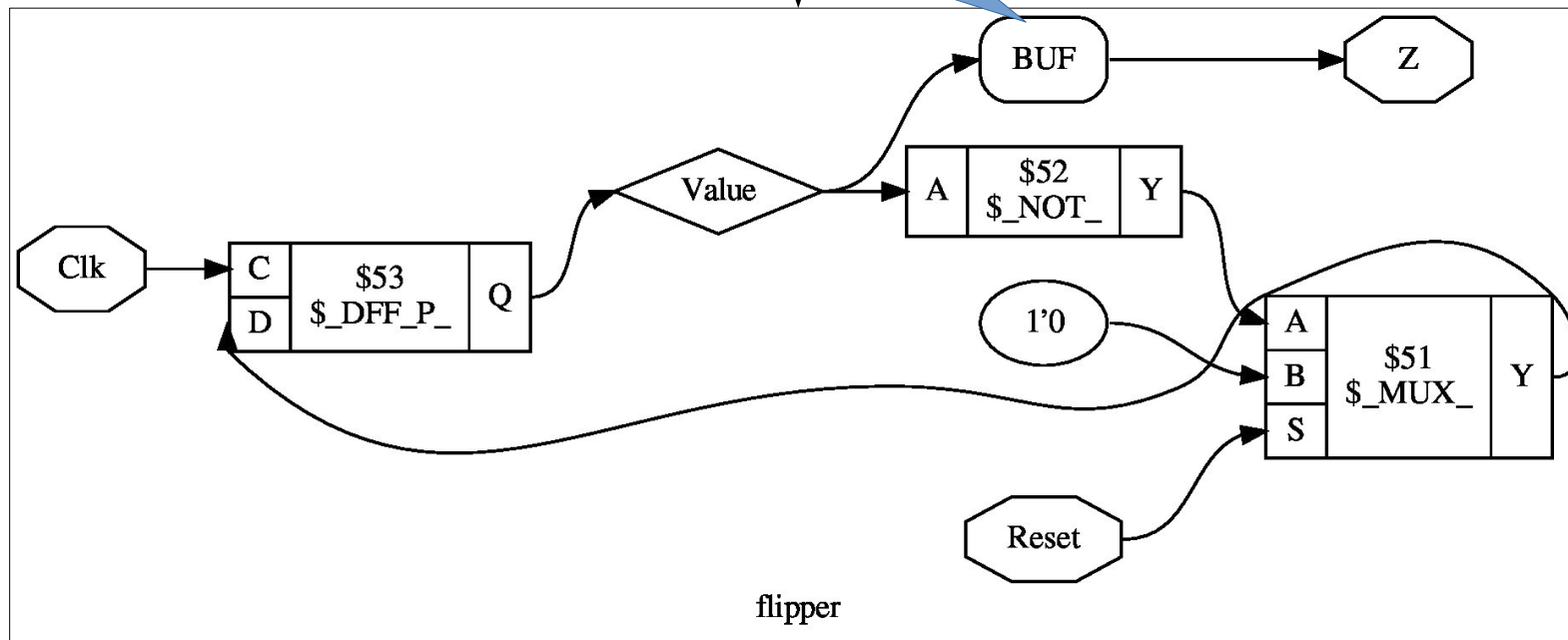
```
module flipper(Clk, Reset, Z);  
  
    input Clk, Reset;  
    reg Value;  
    output Z;  
  
    always @(posedge Clk)  
    begin  
        if (Reset)  
            Value <= 1'b0;  
        else  
            Value <= ~Value;  
        end  
  
    assign Z = Value;  
endmodule
```



# Synthesis

```
module flipper(Clk, Reset, Z);  
  
    input Clk, Reset;  
    reg Value;  
    output Z;  
  
    always @(posedge Clk)  
    begin  
        if (Reset)  
            Value <= 1'b0;  
        else  
            Value <= ~Value;  
        end  
  
    assign Z = Value;  
endmodule
```

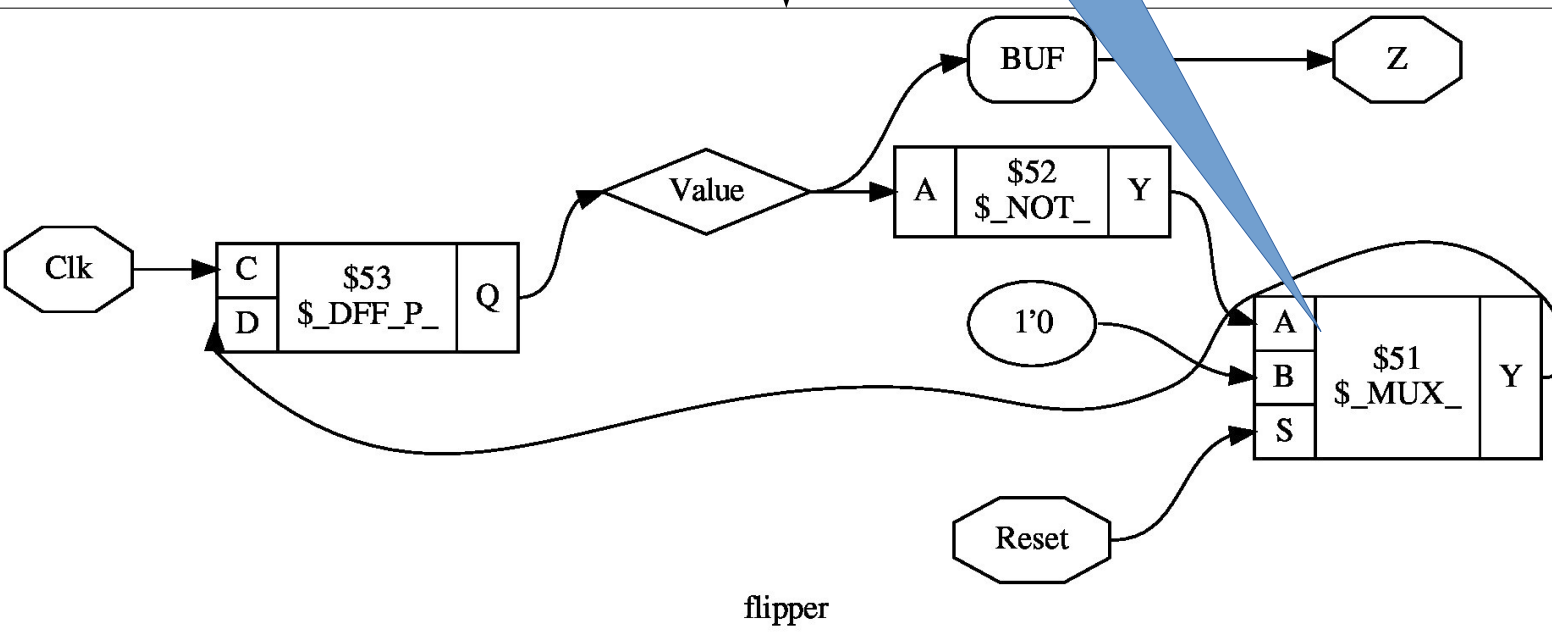
BUF indicate a net-to-net connection:  
Value is passed unchanged  
to output Z.



# Synthesis

This is a multiplexer.  
When its S input is 0,  
it selects input A, otherwise B.

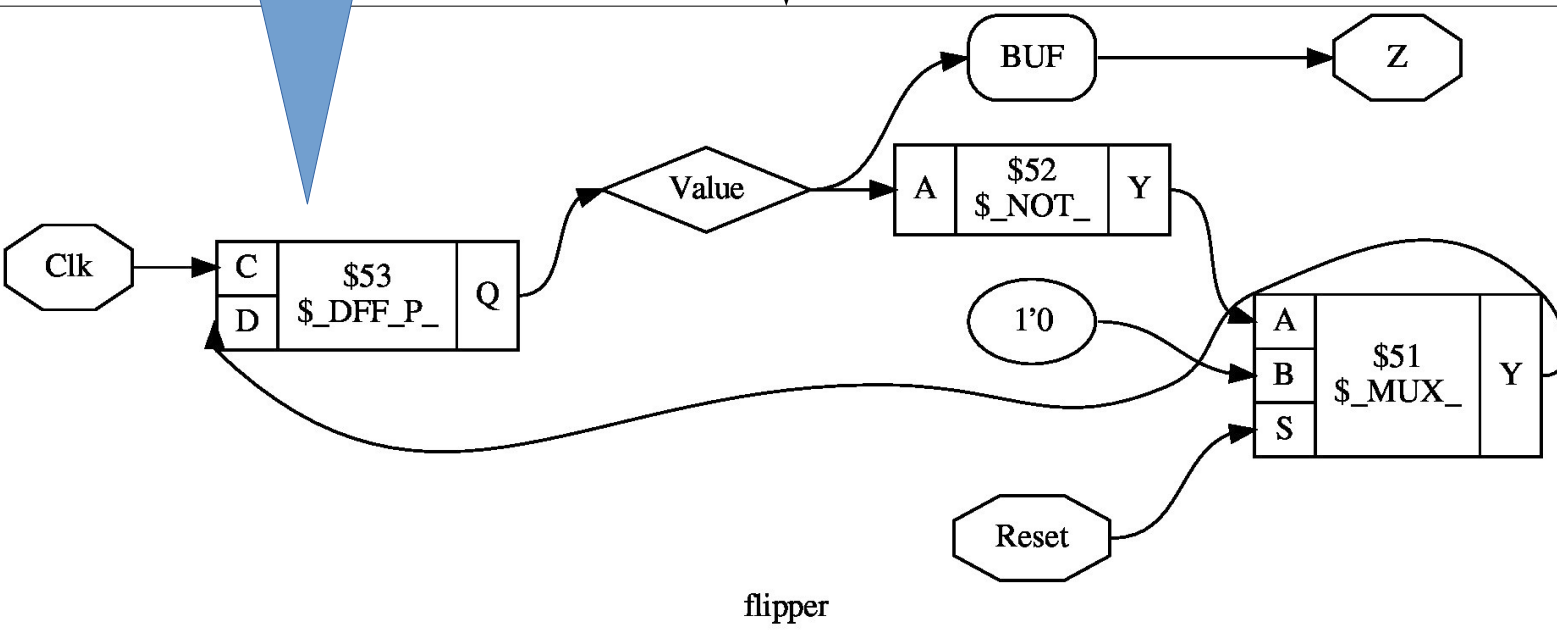
```
module flipper(Clk, Reset, Z);  
  
    input Clk, Reset;  
    reg Value;  
    output Z;  
  
    always @(posedge Clk)  
    begin  
        if (Reset)  
            Value <= 1'b0;  
        else  
            Value <= ~Value;  
        end  
  
        assign Z = Value;  
    endmodule
```



# Synthesis

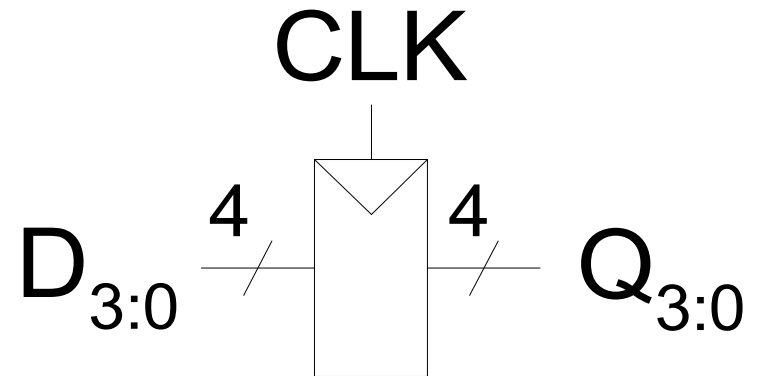
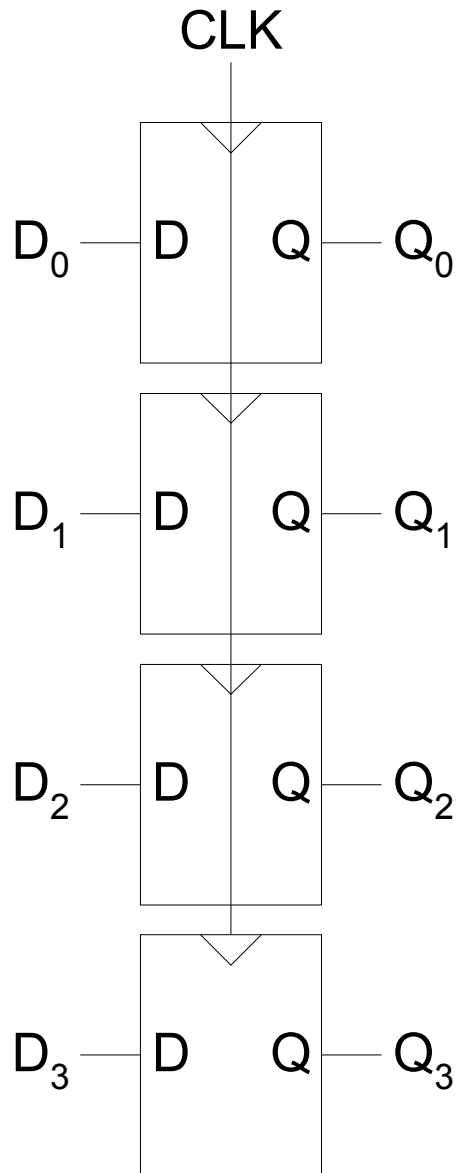
DFF is a flip-flop, the implementation circuit of a register.  
It actually stores the current Value.  
This is covered in Digital Logic.

```
module flipper(Clk, Reset, Z);  
  
    input Clk, Reset;  
    reg Value;  
    output Z;  
  
    always @(posedge Clk)  
    begin  
        if (Reset)  
            Value <= 1'b0;  
        else  
            Value <= ~Value;  
        end  
  
    assign Z = Value;  
endmodule
```





# Registers



# Combinatorial Verilog

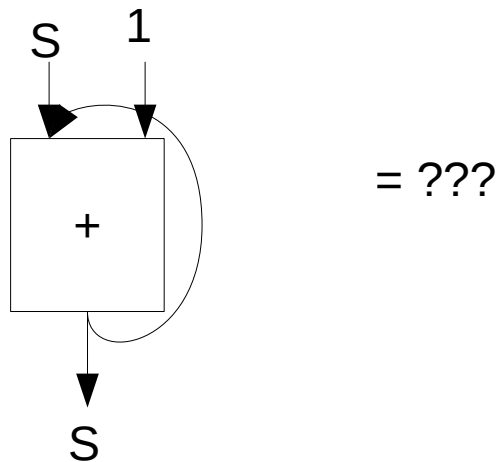
Circuits are defined in terms of gates. Operations are applied *continuously*, i.e. as soon as an input changes, the output reflects that. Calculates result based only on current inputs.

Use `assign` to connect ports.

For example:

```
module example(A, B);  
    input [7:0] A;  
    output [7:0] B;  
    assign B = A + 1;  
endmodule
```

`assign S = S + 1;` // You can't do this!



# Sequential Verilog

Everything from combinatorial Verilog, plus registers. Operations are usually defined in terms of registers. Operations are applied *discretely*, i.e. at the particular point in time when a positive clock edge is received. Coding style is similar to a traditional imperative programming language, like Python or C++. Calculates results based on current inputs *and* stored values.

Use `always` to introduce sequential code. Within sequential code, you can use `if`, `<=`, `case`, `etc.`

For example:

```
module example(Clk, A, B);  
    input [7:0] A;  
    output [7:0] B;  
    input Clk;  
    reg [7:0] state;  
    assign B = state;  
    always@(posedge Clk)  
        state <= state + 1;  
endmodule
```

# Verilog, combinatorial vs. sequential: a guide for the perplexed

## Sequential Verilog

```
reg something;  
reg something_else;  
always@(posedge clk)  
    if (something == whatever)  
        something <= 1;
```

Declare flip-flops (mutable storage locations) with `reg`. Assign to `reg` within `always@` block. This connects given flip-flop to the clock. You can use imperative-style constructs like `if` and `else`.

## Combinatorial Verilog

```
wire whatever;  
assign whatever = (something & something_else) ?  
    something : something_else;
```

Declare immutable connections between gates with `wire`. Connect wires to gates, given as effect-free expressions using `assign =` syntax. Conditions are usually expressed with `?:` syntax.

You can (and should!) use both sequential and combinatorial code in designing FSMs, but don't mix up the syntax: the sequential code updates the state registers, while the combinatorial code can express the calculations for the output and the next state.

Within combinatorial Verilog code (i.e. in `always@` blocks) we can use two kinds of assignment to `regs`:

`=`        Blocking assignment operator  
`<=`       Non-blocking assignment operator

Blocking assignments mean 'assign the value to the variable right away this instant'. Nonblocking assignments mean 'figure out what to assign to this variable, and store it away to assign at some future time'.

In many cases it doesn't make a difference which you use; it *does* make a difference when you have multiple assignments within a single block of code. In that case, the right-hand side of *all* non-blocking assignments will be calculated *before* any assignment takes place.

Don't confuse these sequential assignment syntaxes with the combinatorial `assign =` syntax. The latter produces a connection *continuously*, which is to say not synced to any clock. It shouldn't be used in an `always@` block and can only give values to `wires`, not to `regs`.

# Blocking and Non-Blocking Assignments

- Blocking assignments ( $X=A$ )
  - completes the assignment before continuing on to next statement
- Non-blocking assignments ( $X<=A$ )
  - completes in zero time and doesn't change the value of the target until a blocking point (delay/wait) is encountered
- Example: swap

```
always @(posedge CLK)
begin
    temp = B;
    B = A;
    A = temp;
end
```

```
always @(posedge CLK)
begin
    A <= B;
    B <= A;
end
```

```
always @(posedge CLK)
begin
    A = A ^ B;
    B = A ^ B;
    A = A ^ B;
end
```

# To Block or Not to Block ? cont

Module blocking(a,b,c,x,y);

input a,b,c;

output x,y;

reg x,y;

always @\*

begin

x = a & b;

y = x | c;

end

endmodule

Blocking behavior	a	b	c	x	y
Initial values	1	1	0	1	1
a changes → always block execs	0	1	0	1	1
x = a & b; //make assignment	0	1	0	0	1
y = x   c; //make assignment	0	1	0	0	0

Module nonblocking(a,b,c,x,y);

input a,b,c;

output x,y;

reg x,y;

always @\*

begin

x <= a & b;

y <= x | c;

end

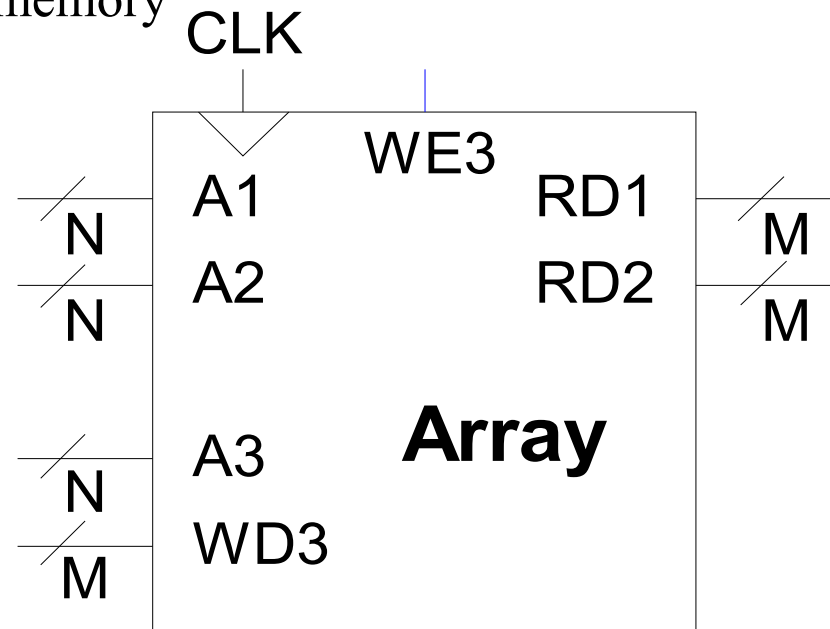
endmodule

Non-blocking behavior	a	b	c	x	y
Initial values	1	1	0	1	1
a changes → always block execs	0	1	0	1	1
x = a & b;	0	1	0	1	1
y = x   c; //x not passed from here	0	1	0	1	1
make x, y assignments	0	1	0	0	1

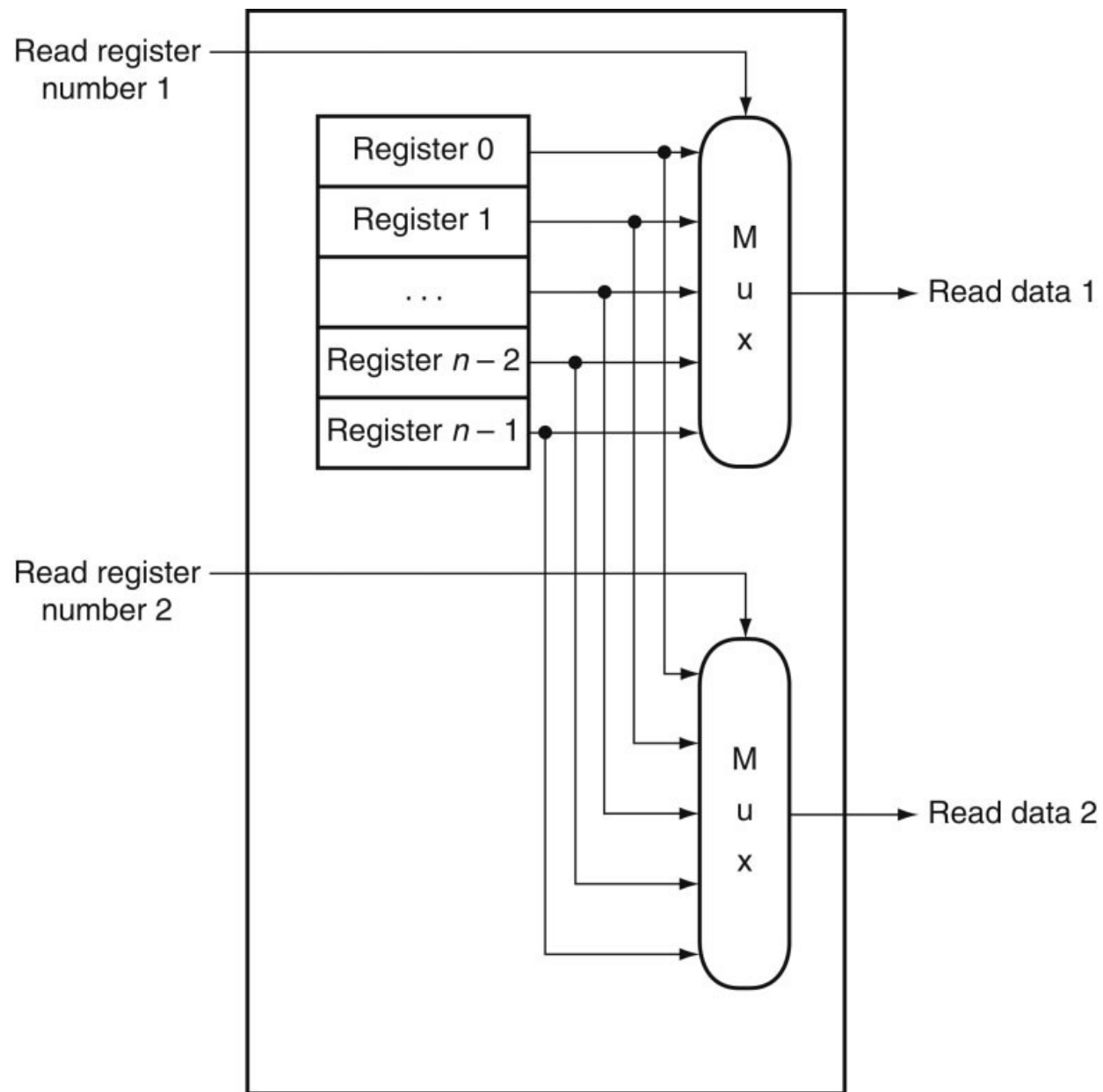
non-blocking behavior does not preserve logic flow!!

# Register Files

- **Port:** address/data pair
- 3-ported memory
  - 2 read ports (A1/RD1, A2/RD2)
  - 1 write port (A3/WD3, WE3 enables writing)
- **Register file:** small multi-ported memory
- $N = 5$  implies  $32 = 2^5$  registers



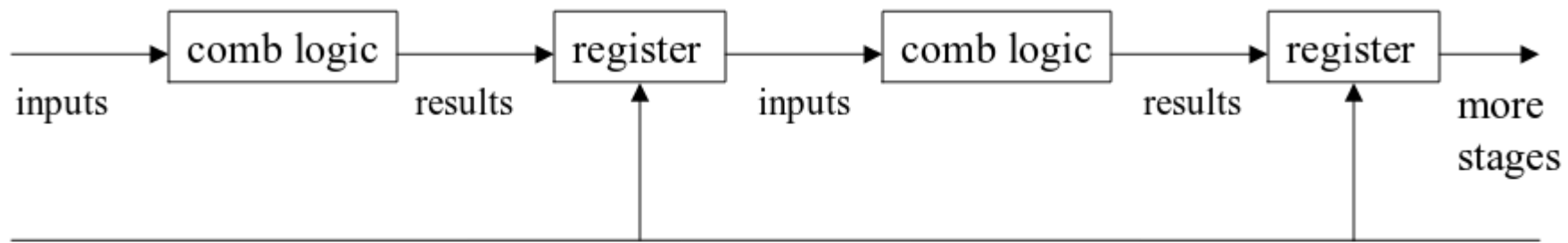




The implementation of two read ports for a register file with  $n$  registers can be done with a pair of  $n$ -to-1 multiplexors, each 32 bits wide. The register read number signal is used as the multiplexor selector signal.

# Sequential Logic

- Can be generalized as a series of combinational blocks with registers to hold results.



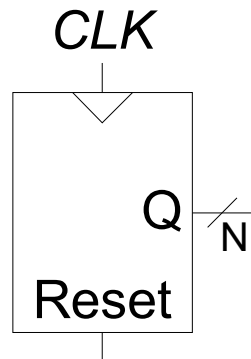
clk

Results of each stage are stored (latched) in a register (D-Flip-Flops) by a common clock

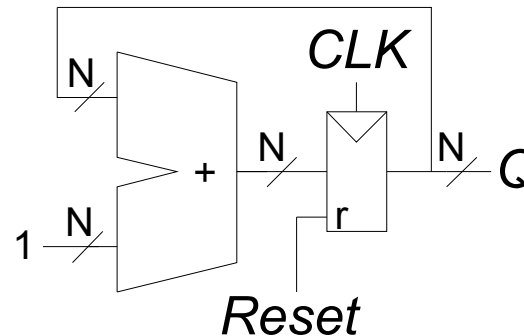
# Counters

- Increments on each clock edge
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
- Example uses:
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

## Symbol



## Implementation

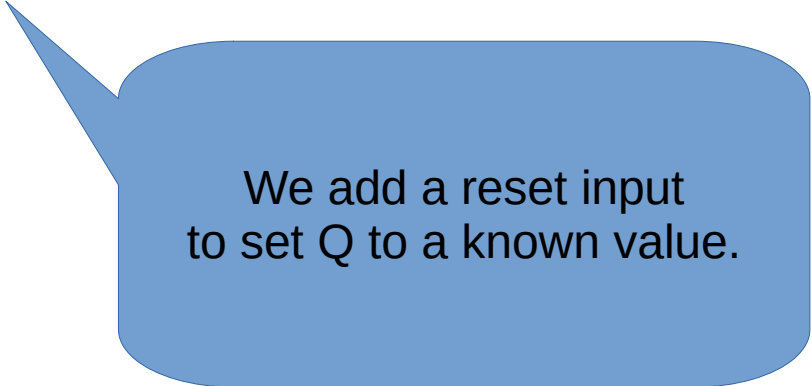


```
module upcount(clk, E, Q);  
    input clk, E;  
    output reg [3:0] Q;  
  
    always @(posedge clk)  
        if (E) begin  
            Q <= Q + 1;  
        end  
endmodule
```



What is the initial value of Q?

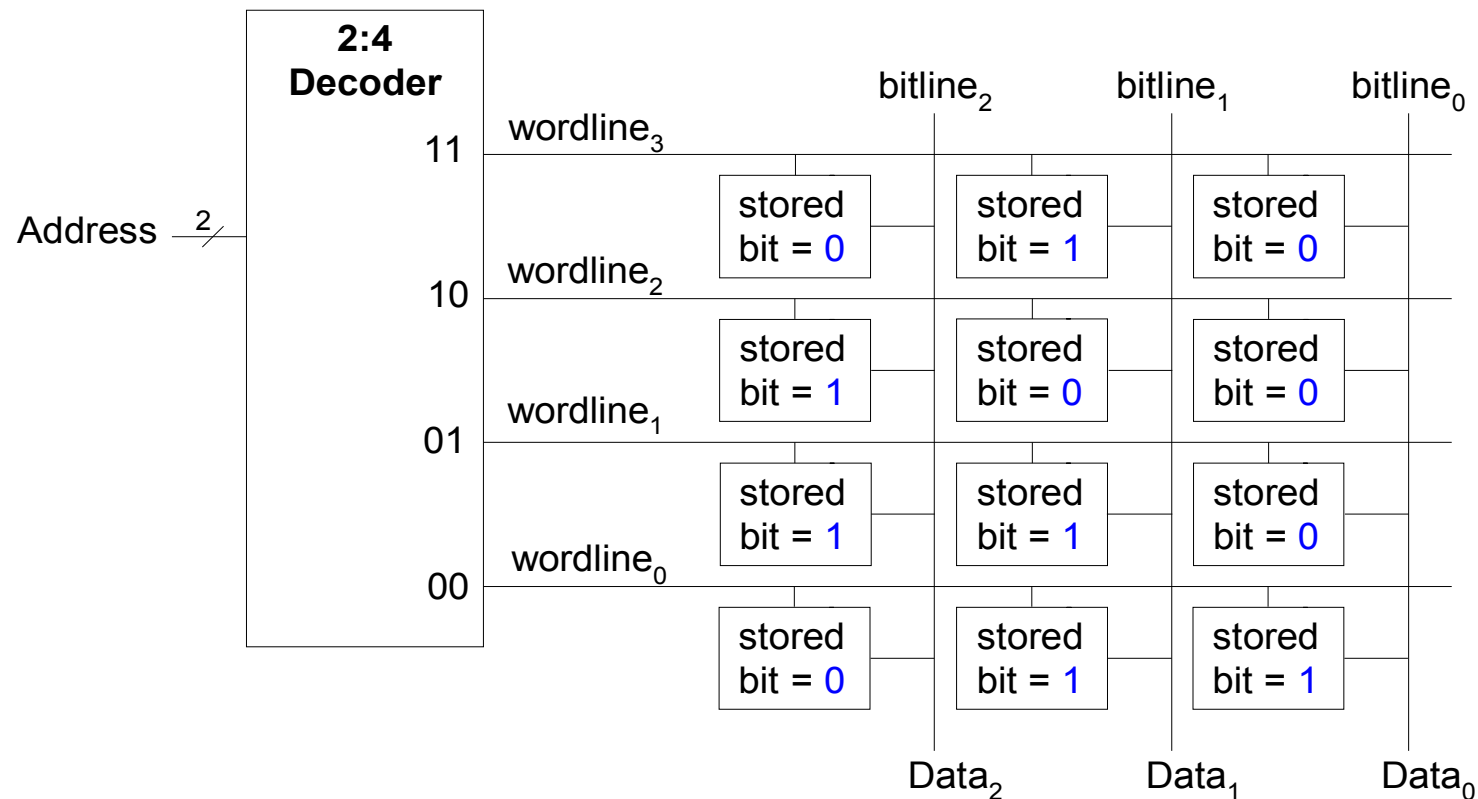
```
module upcount(clk, rst, E, Q);  
    input clk, E, rst;  
    output reg [3:0] Q;  
  
    always @(posedge clk)  
        if (rst) begin  
            Q <= 4'b0;  
        end else if (E) begin  
            Q <= Q + 1;  
        end  
endmodule
```

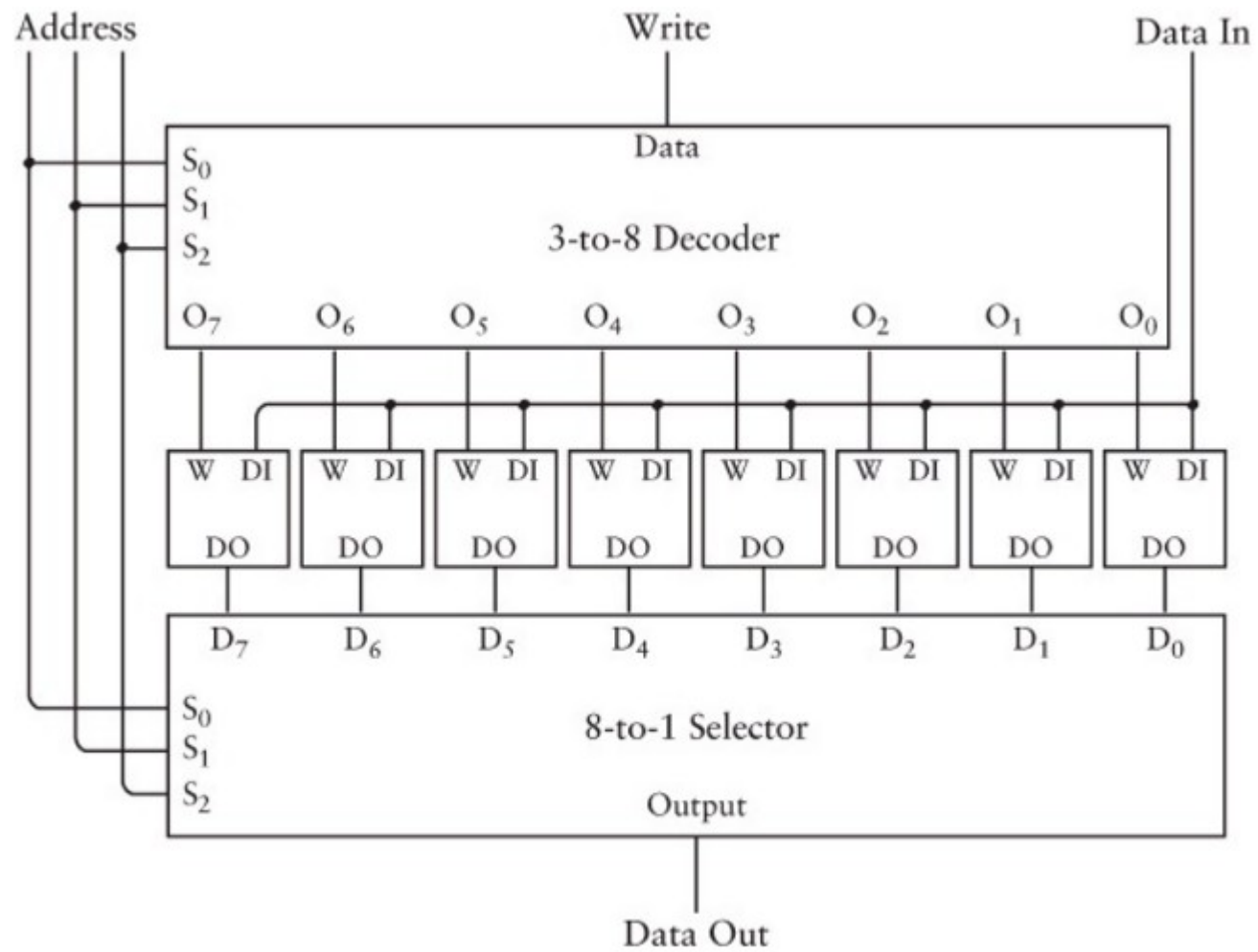


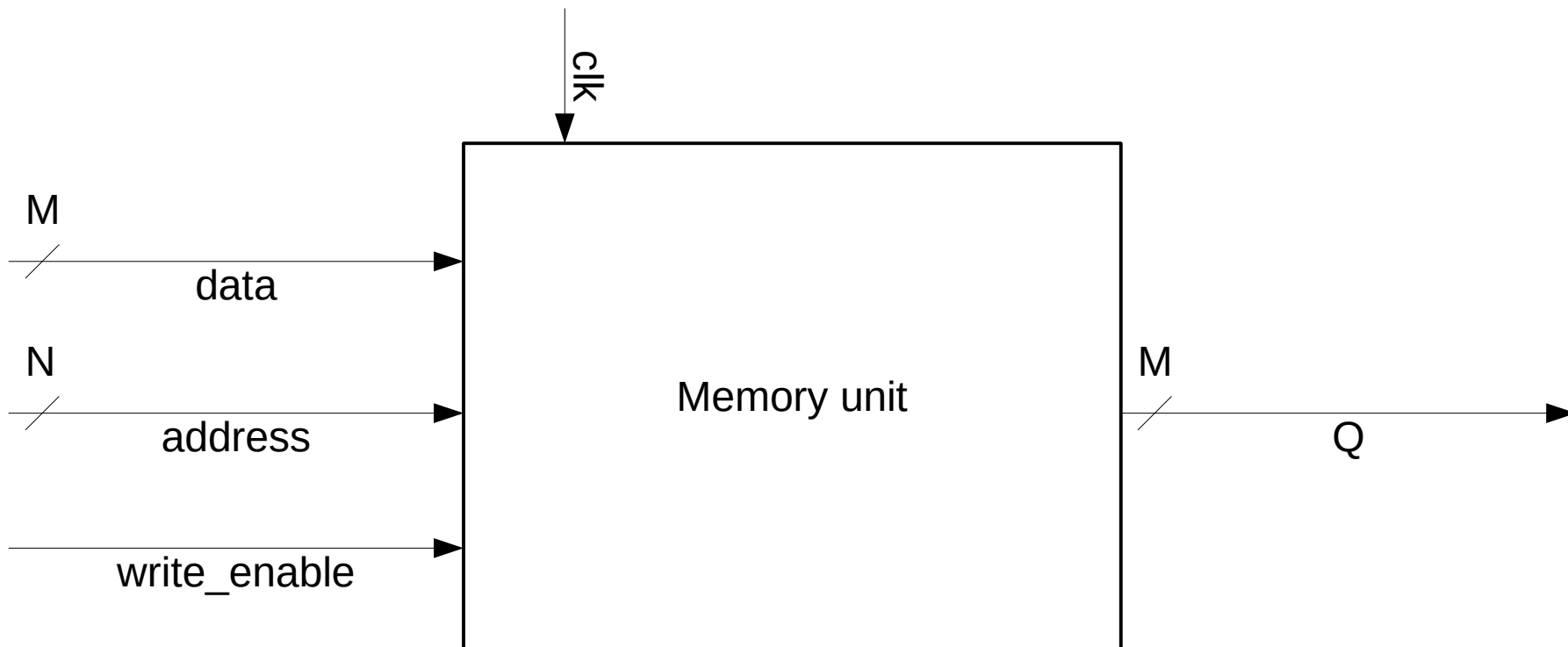
We add a reset input  
to set Q to a known value.

# Memory Array

- Wordline:
  - like an enable
  - single row in memory array read/written
  - corresponds to unique address
  - only one wordline HIGH at once









```
module memory_array(  
    data, addr, we, clk, q  
);  
  
    input [7:0] data;  
    input [5:0] addr;  
    input we, clk;  
    output [7:0] q;  
  
    reg [7:0] ram[63:0];  
  
    reg [5:0] addr_reg;  
  
    always@(posedge clk)  
    begin  
        if (we)  
            ram[addr] <= data;  
  
        addr_reg <= addr;  
  
    end  
  
    assign q = ram[addr_reg];  
  
endmodule
```

```

module memory_array(
    data, addr, we, clk, q
);

    input [7:0] data;
    input [5:0] addr;
    input we, clk;
    output [7:0] q;

    reg [7:0] ram[63:0];

    reg [5:0] addr_reg;

    always@(posedge clk)
    begin
        if (we)
            ram[addr] <= data;

        addr_reg <= addr;

    end

    assign q = ram[addr_reg];

endmodule

```

RAM (often known as *memory*) is a mutable, volatile, byte-indexed array.

Each byte has a unique numeric *address*.  
 Given an address, we can put a particular value in the array (a *store*) or retrieve the current value (a *load*).

Do not confuse with disk storage!

```
module memory_array(  
    data, addr, we, clk, q  
);  
  
    input [7:0] data;  
    input [5:0] addr;  
    input we, clk;  
    output [7:0] q;  
  
    reg [7:0] ram[63:0];  
  
    reg [5:0] addr_reg;  
  
    always@(posedge clk)  
    begin  
        if (we)  
            ram[addr] <= data;  
  
        addr_reg <= addr;  
  
    end  
  
    assign q = ram[addr_reg];  
  
endmodule
```

Cycle	data	addr	we	q
0	42	0	1	42
1	99	1	1	99
2				
3				
4				

In cycle 0, we store value 42

In cycle 1, we store value 99

In cycle 2, we store value 99

In cycle 3, we read the value at cell 0. That cell was previously set to 42

In cycle 4, we read the value at cell 9. That cell was previously set to 30, so we get that value now. The value of input **data** is ignored.