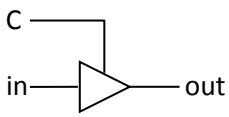
## Last time

- *Introduced tri-state devices.*

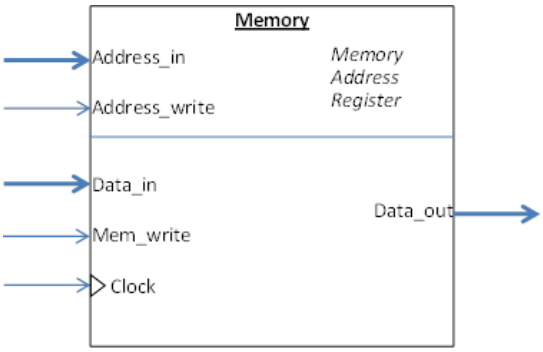



| C | in | out |
|---|---|---|
| 0 | 0 | Not connected |
| 0 | 1 | Not connected |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

- *Introduced registers.* They output their data at all times. You can make Data_out become Data_in if you set "Write=1" and then have a rising edge on clock (0 changing to a 1).

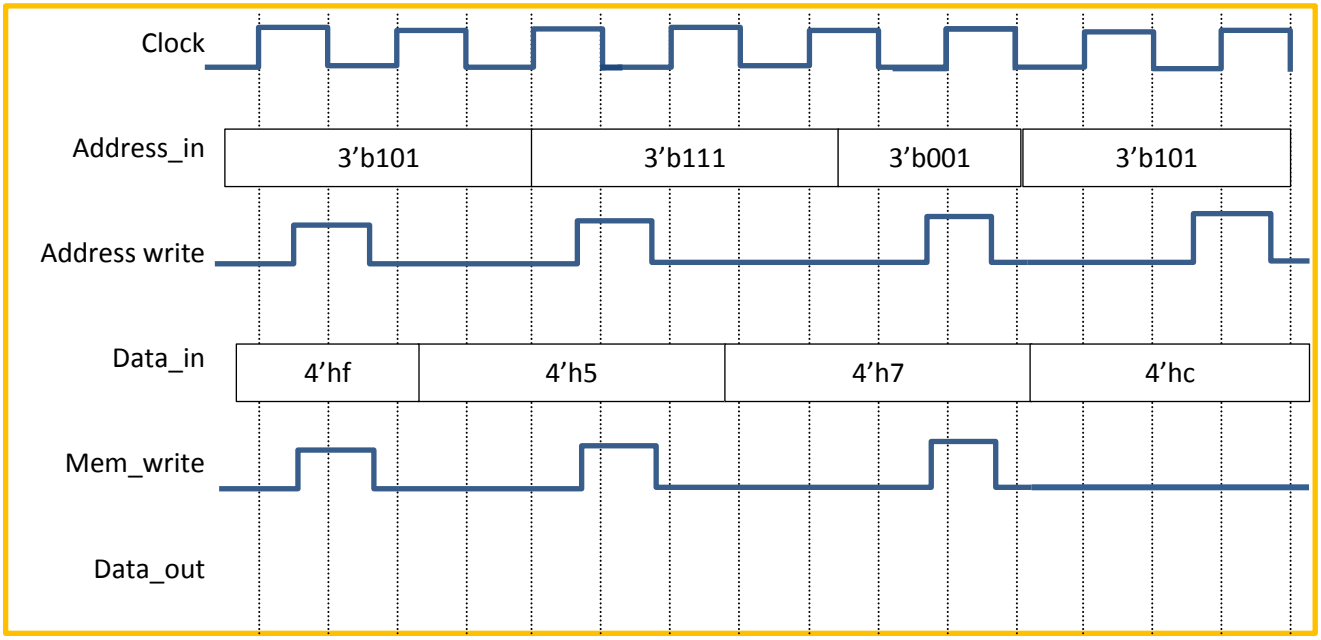

- *Introduced memories.* Basically an array where a register is a variable. You set the Memory Address Register to the index you want to read or write. Writing is otherwise done just like a register.

| Register 0 |
|---|
| Register 1 |
| Register 2 |
| Register 3 |
| Register 4 |
| Register 5 |
| Register 6 |
| Register 7 |




## Questions:

1. Fill in "Data_out" for a 8-entry memory, where each entry is 4 bits. Assume all memory addresses are currently set so memory[x]=2x (e.g., location 4 has initially has a value of 8). Address starts as 2.




2. Can you make an AND gate using only "0s" and "1s", tri-state devices, and inverters? (This is tricky…)

# Finite State Machines

So we have combinational logic and a way of storing state (registers/memory). What we want to do now if find a way to combine these to be able to implement an *algorithm*. Doing a step-by-step process in hardware involves something called a *finite state machine* (FSM).

## Baking a cake

Let's say we want to bake a cake. We could break that down into a few steps:

0. Reset (begin)
1. Pour ingredients into a pan
2. Mix ingredients in oven
3. Bake until done
4. End

If we write a truth table, what should the next step (state) be given the current state?

| state | next_state |
|---|---|
| Reset | Pour |
| Pour | |
| Mix | |
| Bake | |
| End | |

This is just a truth table. We can do it in Verilog.

```
module compute_next_state(
    input wire [2:0] state,
    output reg [2:0] next_state);

        always @* begin
            next_state = state_reset;

            case (state)
                state_reset: begin
                    next_state = _____________;
                end

                state_pour: begin
                    next_state = _____________;
                end

                state_mix: begin
                    next_state = _____________;
                end

                state_bake: begin
                    next_state = _____________;
                end

                state_end: begin
                    next_state = _____________;
                end
            endcase
        end
endmodule
```

So how is it we can use terms like "state_bake"? Simple, we define those terms to be numbers, just like one might use a #define in C/C++.[1]

```
parameter state_reset = 3'h0;
parameter state_pour  = 3'h1;
parameter state_mix   = 3'h2;
parameter state_bake  = 3'h3;
parameter state_end   = 3'h4;
```

Now we can use the terms freely. In general, just like C++ programming, you don't want to use "magic numbers" when you can use a more descriptive term (i.e., it's better to use state_pour than 3'h1 in your code).

Now, we want to be in each state for a fixed amount of time. Let's say one clock tick. All we need to do is run this through a register. So here's a register:

```
module state_register(
        input wire clock,
        input wire reset;
        input wire [2:0] data_in,
        output reg [2:0] data_out);

        always @(posedge clock) begin
                if (reset == 1'b1) begin
                        data_out <= state_reset;
                end else begin
                        data_out <= data_in;
                end
        end
endmodule
```

We can take the two modules and build our finite state machine:

```
module top(
        input wire reset,
        input wire clock);

        wire [2:0] next_state;
        wire [2:0] state;

        state_register u1 (clock, reset, next_state, state);
        compute_next_state u2 (state, next_state);

endmodule
```

[1] The parameter is mighty close to a #define, but Verilog has something that is even closer, "`define". We don't tend to use it because it has some syntax issues that are a bit annoying.

Now let's make it a bit more realistic (at least for how I cook). It is going to take a while to cook the cake. Using the Brehob Patented Cooking Method™, we will simply wait until the smoke detector goes off (yes I've done that more than once).

So we will end up with the following:

```
state    smoke    |   next_state
------------------------------
reset      0      |       pour
reset      1      |       pour
                  |
pour       0      |       mix
pour       1      |       mix
                  |
mix        0      |       bake
mix        1      |       bake
                  |
bake       0      |       bake
bake       1      |       end
                  |
end        0      |       end
end        1      |       end
```

We can reduce that by only having two listings for the case where smoke matters.

```
module compute_next_state(
    input wire [2:0] state,
    input wire smoke,
    output reg [2:0] next_state);

    always @* begin

        next_state = state_reset;   // default.  This is needed,
                                    // so we define next_state even
                                    // when state is not one of
                                    // the defined values.

        case (state)
            state_reset: begin
                next_state = state_pour;
            end

            state_pour: begin
                next_state = state_mix;
            end

            state_mix: begin
                next_state = state_bake;
            end

            state_bake: begin
                if (smoke == 1'b0) begin
                    next_state = state_bake;
                end else begin
                    next_state = state_end;
                end
            end

            state_end: begin
                next_state = state_end;
            end
        endcase
    end
endmodule
```

## High-level Finite State Machine picture:

## Outputs

Having states is all well-and-good, but it doesn't help us *do* anything. What we want are *outputs*. So in our case, let's say we have three things we are controlling: the pourer, mixer and oven. In that case, we can indicate what outputs we want in each state easily:

| state | pourer | mixer | oven |
|---|---|---|---|
| reset | 0 | 0 | 0 |
| pour | 1 | 0 | 0 |
| mix | 0 | 1 | 0 |
| bake | 0 | 0 | 1 |
| end | 0 | 0 | 0 |

With very little effort, we can combine the next_state and output logic. For brevity, only write "1"s where needed and leave out the "0s".

| state | smoke | next_state | pourer | mixer | oven |
|---|---|---|---|---|---|
| reset | | | | | |
| pour | | | | | |
| mix | | | | | |
| bake | 0 | | | | |
| bake | 1 | | | | |
| end | | | | | |

Now, let's do it in Verilog!

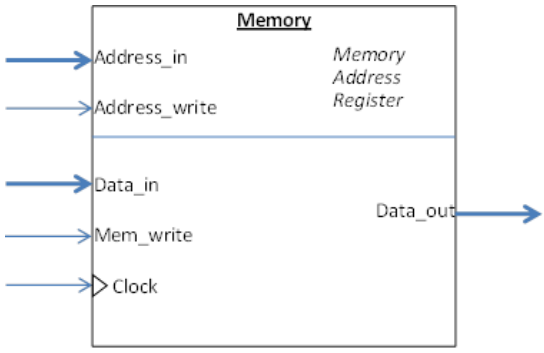
```
always @* begin
    next_state = state_reset;









    case (state)
        state_reset: begin
            pourer = 1'b0;
            mixer = 1'b0;
            oven = 1'b0;
            next_state = state_pour;
        end

        state_pour: begin
            pourer = 1'b1;
            mixer = 1'b0;
            oven = 1'b0;
            next_state = state_mix;
        end

        state_mix: begin
            pourer = 1'b0;
            mixer = 1'b1;
            oven = 1'b0;
            next_state = state_bake;
        end

        state_bake: begin
            pourer = 1'b0;
            mixer = 1'b0;
            oven = 1'b1;
            if (smoke == 1'b0) begin
                next_state = state_bake;
            end else begin
                next_state = state_end;
            end
        end

        state_end: begin
            pourer = 1'b0;
            mixer = 1'b0;
            oven = 1'b0;
            next_state = state_end;
        end
    endcase
end
```

It is generally best to provide default values for everything when in a combinational logic block. So that *anything* that gets assigned anywhere in the block has a default. *Fill that in now*…

You can also take advantage of having the defaults by only assigning things when they aren't the default*. Cross out all the lines you don't need once you've got the default values...*

## A bit more realistic example

Let's take our memory that we did last time (and reviewed at the start of class) and take a shot at building an interface for it so we can just ask for a read or write and it works—we don't need to change the address and then data when writing.




Of course, the memory still *requires* that we change the address and then the data. So what we are going to do is build an FSM that does it for us. Let's have our system have the following inputs:

- Address_in
- Data_in
- Type (read or write, we'll have read be a 0 and write be a 1)
- Start (Indicates we want to do a read or a write)

We wait until start goes high. Once it does we either just want to read (type=0) or write (type=1) the memory at the address specified. We'll assume Address_in and Data_in are tied to the memory's Address_in and Data_in. So all we need to do is drive Address_write and Mem_write at the right times.

```
state    start    type   |  next_state   address_write  mem_write
-------------------------------------------------------------
reset                    |

decide     0             |
decide     1        0    |
decide     1        1    |

read1                    |

write1                   |
write2                   |
```

Now let's do the Verilog!