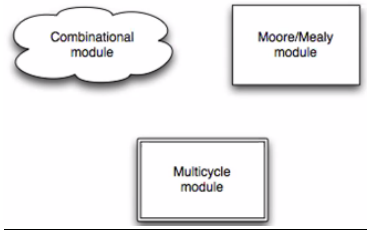
laalcham@ucsc.edu for CSE 125, UCSC

Modules

combinatorial logic cloud: no FFs
- drawn as cloud

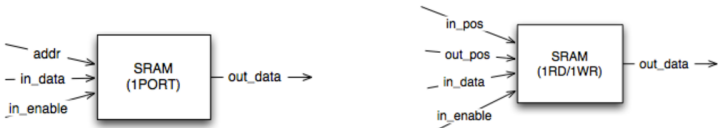
module: uses FFs
- drawn as rectangle
- assumed to be output register modules
- may have combinatorial logic within them
- all outputs are assumed to be registered using FFs

multicycle module:
- contains pipeline (multiple FFs in series)
- it takes multiple cycles for output values to reflect input values at earlier clock cycle



Storage

- storage is drawn as a box marked with the SRAM keyword (static random access memory)
- SRAMs can be addressed via a data input and data output.



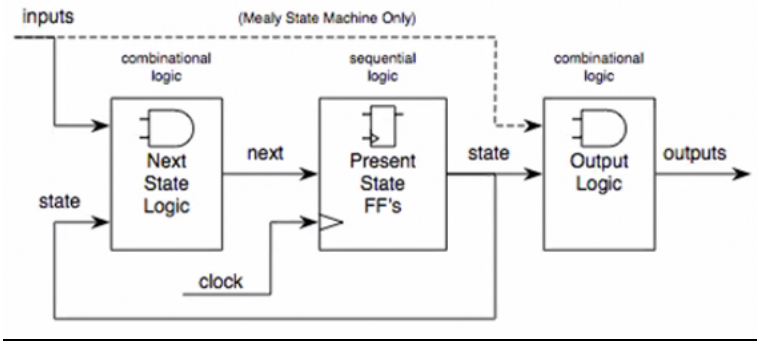
! For this class we will be using **2 types of storage**!
1. 1 port SRAMs
2. 1rd/1wr SRAMs

➔ FPGAs only have up to 1 read and 1 write port.
➔ ASICs do NOT provide SRAMs. They only provide FFs
- ASIC memory compilers generate 1port and 1rd/1wr SRAMs
- more complicated SRAMs require custom implementation

Assume that **reading from SRAM takes 1 full clock cycle**
- SRAM outputs are equivalent to a FF
- avoid logic in SRAM inputs

FSMs

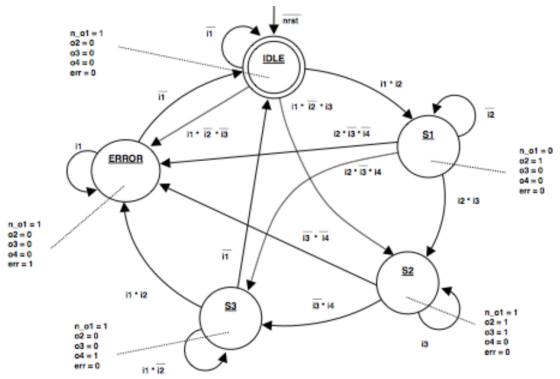
Finite state machines have **state elements** that store the **state** of the machine and combinatorial logic that determines the outputs of the machine.



Mealy Machine: FSM outputs can change at any time because they depend on inputs and state
Moore Machine: FSM outputs can only change when state changes, only depend on states

Heiner Litz ☺ strongly recommends Moore machines because mealy is problematic!

- Mealy is sometimes easier to implement, but can make timing more complicated
- use Moore when developing hardware <3



Encoding the states of a FSM

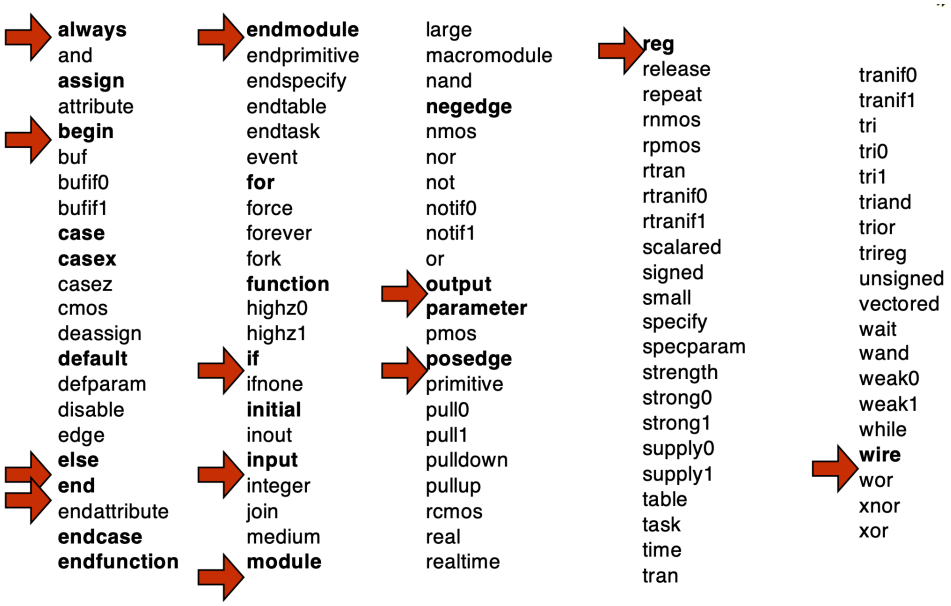
- Each state S_i is indexed by some value i
- Each state is represented by a binary pattern P_i
- A function E maps the state index to its pattern
- There are different coding schemes for encoding states! Pros and cons to each.

[Example of One-hot encoding: 4 states]

1 <= i <= 4
states = {S_1, S_2, S_3, S_4}
E(1) = 00
E(2) = 01
E(3) = 10
E(4) = 11

Verilog Keywords

Keywords cannot be used as variable names ☺



Verilog Identifiers and Values

Identifiers must start with a character and cannot be a keyword.

Examples: my_identifier, adder_8b

Values are specified in the format <binary size>'<base><value>
legal bases:

- b (binary)
  - 0, 1, x, z, ? (where ? == z)
- o(octal)
  - 0-7, x, z, ?
- d (decimal)
  - 0-9
- h(hexadecimal)
  - 0-9, a-f, x, z, ?

[Examples of values]

4'o6 == 4'b0110

8'hc5 == 8'b11000101

6'hf0 == 6'b110000 (truncated the 4'hf)

6'hf == 6'b001111 (zero filled)

6'h? == 6'b?????? (? or z filled)

Verilog Operators

Bitwise operators operate on 2 n-bit binary values and produce n bits

→ bitwise operators are faster because they can be executed in parallel.

Unary operators operate on 1 binary value and produce 1 bit.

→ unary operators have gate-specific runtime

→ logarithmic depth of logic levels

Bitwise Operators

```
~    Bitwise negation
&    Bitwise AND
|    Bitwise OR
^    Bitwise XOR
~&   Bitwise NAND
~|   Bitwise NOR
~^ or ^~ Equivalence   Bitwise NOT XOR
```

Unary Operators

```
&   AND reduction
|   OR reduction
^   XOR reduction
~&  NAND reduction
~|  NOR reduction
~^  XNOR reduction
```

[Examples of Bitwise and unary operations]

```
$display("%b", (4'b1100 & 4'b1001));    1000        $display("%b", (& 4'b1001));      0
$display("%b", (4'b1100 & 4'bx111));    x100        $display("%b", (& 4'bx111));      x
$display("%b", (4'b1100 & 4'bz111));    x100        $display("%b", (& 4'bz111));      x

$display("%b", (4'b1100 | 4'b1001));    1101        $display("%b", (| 4'b1001));      1
```

Recall:

z == unknown output of high impedance/tristate devices

x == unknown/ don't care for all devices.

Verilog Logical Operators

For logical operators, any value that is 0 evaluates to false and anything other than 0 evaluates to true.

Logical Operators

```
&&   Boolean AND
||   Boolean OR
```

```
$display("%b", (4'b0000 && 4'b0000));    0
$display("%b", (4'b1100 && 4'b1001));    1
$display("%b", (4'b1100 && 4'bx111));    1
$display("%b", (4'b1100 && 4'bz111));    1

$display("%b", (4'b1100 || 4'b1001));    1
```

Verilog Relational Operators

```
>  Greater than
>= Greater than or equal
<  Less than
<= Less than or equal
== Logical equality
!= Logical inequality
```

Verilog Shift Operators

```
$display("%b", (4'b1001 << 1));          0010
$display("%b", (4'b10x1 << 1));          0x10
$display("%b", (4'b10z1 << 1));          0z10

$display("%b", (4'b1001 >> 1));          0100
$display("%b", (4'b10x1 >> 1));          010x
$display("%b", (4'b10z1 >> 1));          010z
```

Verilog Concatenation/Replication

Concatenation is used to combine buses into 1

```
d = {d1, d2};

{carry, val} = a+b;

foo = #{val}
```

[Examples of Concatenation]

```
$display("%b" , {4'b1001,4'b10x1});   100110x1
$display("%b" , {4{4'b1001}});        1001100110011001
```

Verilog Comparators

There are 2 comparators, (==) and (===)

== and != are traditional C-like comparators

=== and !== comparators accomodate x and z values

- not synthesizable

```
$display("4'bx == 4'bx %b",(4'bx==4'bx));       x
$display("4'bx === 4'bx %b",(4'bx===4'bx));     1
$display("4'b0 == 4'bx %b",(4'b0==4'bx));       x
$display("4'b0 === 4'bx %b",(4'b0===4'bx));     0
$display("4'bx == 4'bz %b",(4'bx==4'bz));       x
$display("4'bx === 4'bz %b",(4'bx===4'bz));     0
```

Variable Declaration

- wires used to connect gates
- reg used for constant vals
- type chosen determines what operators can be used on them

```
<reg|wire>  <width> <name> <n Elements>;
```

[Examples]

```
reg [3:0] val1;  // 4 bits element
wire [2:0] val2; // 3 bits element

reg [3:0] memory [1:0]; // 2 elements of 4 bits each
```

Detailed net Types

Net Type	
wire	- establishes connection between gates - default value for all signals
tri	- similar to wire - indicates 0, 1, z - cannot be connected to output of logic gates
supply0, supply1	- ground and power connections - alternatively, use 1'b0 and 1'b1
wand, wor, triand, trior, tri0, tri1, trireg	- perform signal resolution - not used in this course

Registers versus Wires

Wire	Reg
- not an actual driver - used to connect components - cannot store values - used with assign	- represents a driver - can store values, doesn't have to - used with always @

- always use wires for inputs
- use reg for output if FF is desired, otherwise use a wire
- use reg for storage

Structural vs. Behavioral Verilog

In **structural Verilog**, module instances are interconnected by wires and **assign** statements are used for outputs.

Behavioral Verilog uses **always** blocks to implicitly generate Flip Flops

- Behavioral Verilog has a more powerful means to describe complex logic elements
- more convenient to develop state elements

Procedural Blocks	

initial Blocks
- execute 1 time
- behavioral and not synthesizable
- used to build testbenches
	Executed only once before time 0
	`reg c;`
	`initial begin`
	`  c = 0;`
	`end`

always blocks
- executed any time the event list is true
- represent an event-based trigger mechanism
- used to describe comb logic w/o FFs
- use reg for anything other than inputs in always blocks
	`always @(a or b) begin`
	`  c = a+b;`
	`end`
	// does not generate a FF
	// only an assignment of c = a + b

always block examples

Executed when a or b change

```
always @(a or b) begin
   c = a+b;
end
```

Executed when a or b change

```
always @(*) begin
   c = a+b;
end
```

Verilog-2001, always use this for combinational logic

Executed when a goes from 0 to 1

```
always @(posedge a) begin
   c <= a+b;
end
```

Synchronous Design

```
always @(posedge clk) begin
   c <= a+b;
end
```

Executed when b changes

```
always @(b) begin
   c = a+b;
end
```

IMPLICIT LATCH, AVOID THIS!

* -> when inputs change
posedge -> positive edge event