laalcham@ucsc.edu for CSE 125, UCSC

Functions and tasks are of limited use, but sometimes come up and/or can be handy.

Functions Verilog

Functions define a single output that is returned. They can be used directly in assign statements.

- they cannot be associated with a timing #delay
- they must return a value
- should be thought of as a fancy macro for combinational logic
-

```
function [1:0] fulladd;
input a,b,c_in;
begin
{c_out,sum}=a+b+c_in;
end
endfunction

assign res = fulladd(a, b, c_in);
```

Tasks in Verilog

Tasks are like procedures. Regard as a type of macro.

- they can be associated with a timing #delay
- they do not return a value
- they have outputs
- they can call other tasks/functions

```
task [1:0] fulladd;
input a,b,c_in;
output sum,c_out;
{c_out,sum}=a+b+c_in;
endtask

fulladd(.sum(res), .a(a), .b(b), .c_in(cin));
```

Calling a Function	Instantiating a Module

`always @(A or B or C_in) begin`
`  {C0,SUM[0]}=fulladd(A[0],B[0],C_in);`
`  {C1,SUM[1]}=fulladd(A[1],B[1],C0);`
`  {C_out,SUM[2]}=fulladd(A[2],B[2],C1);`
`end`
	`always @(A or B or C_in) begin`
	`  fulladd(A[0],B[0],C_in,SUM[0],C0);`
	`  fulladd(A[1],B[1],C0,SUM[1],C1);`
	`  fulladd(A[2],B[2],C1,SUM[2],C2);`
	`end`

Function versus	**Module**
see code below	see code below

Function versus

```
function decoder_3_8(input select);
begin
     case(select)
          0 : decoder_3_8 = 8'd1;
          1 : decoder_3_8 = 8'd2;
          2 : decoder_3_8 = 8'd4;
          3 : decoder_3_8 = 8'd8;
          4 : decoder_3_8 = 8'd16;
          5 : decoder_3_8 = 8'd32;
          6 : decoder_3_8 = 8'd64;
          7 : decoder_3_8 = 8'd128;
          default:
             $display("ERROR: not supported decoder option"
     endcase
endfunction

wire [2:0] select;

always@(posedge clk) begin
     onehot_reg <= decoder_3_8(select);
end
```

Module

```
module decoder_3_8(input [2:0]  select
                  ,output reg [7:0]    out);

always@(*)
        case(select)
         3'd0 : out = 8'd1;
         3'd1 : out = 8'd2;
         3'd2 : out = 8'd4;
         3'd3 : out = 8'd8;
         3'd4 : out = 8'd16;
         3'd5 : out = 8'd32;
         3'd6 : out = 8'd64;
         3'd7 : out = 8'd128;
        endcase

endmodule

wire [2:0]select;
wire [7:0]onehot;
reg [7:0]onehot_reg;

decoder_3_8 dec(select, out);

always@(posedge clk) begin
     onehot_reg <= onehot;
end
```

Generate (Verilog-2001)

generate is a mechanism run by the pre-compiler that can be used with if/else, for, or case statements to generate multiple instances of the same thing

- each generate net will have a unique name
- each generated primitive instance will have a unique instance name
- generate can instantiate modules

Verilog-2001 generate vs.

```
genvar i;

generate
  for (i=0; i < 4; i=i+1) begin : MEM
     memory U (read, write, data_in[(i*8)+7:(i*8)],
                  address,data_out[(i*8)+7:(i*8)]);
  end
endgenerate
```

4 instantiations of the memory module U_0, U_1, U_2, U_3 are generated.

for loop

```
integer i;

for(i=0 ; i <4 ; i++ ) begin
     if(d[i])
          a[i] = b[i] ^ c[i];
     else
          a[i] = 0;
end
```

4 copies generated of a combinatorial logic circuit.

Generate

generate is more flexible but potentially slower to simulate than a for loop.

[Example: generate]
ripple carry versus carry look-ahead adder.

```
generate
if (adder_width < 16) begin
  ripple_carry_adder # (.W(adder_width)) adder_I
(.a(a), .b(b), .sum(sum));
end
else
begin
  carry_look_ahead_adder # (.W(adder_width)) adder_I (.a(a), .b(b), .sum(sum));
end
endgenerate
```

[Example: for loop]
decision based on static well-known parameter param_use_pos

```
if ( param_use_pos == 1) begin : use_pos
  always @(posedge sysclk) begin
  ...
  end
end
else begin : use_neg
  always @(negedge sysclk) begin
  ...
  end
end
```

Hurrah! the notes up to now cover all we need to know to write decent programs in Verilog. Remember to always think about timing and performance. Have fun.

Design Verification: If you didn't verify it, it doesn't work!

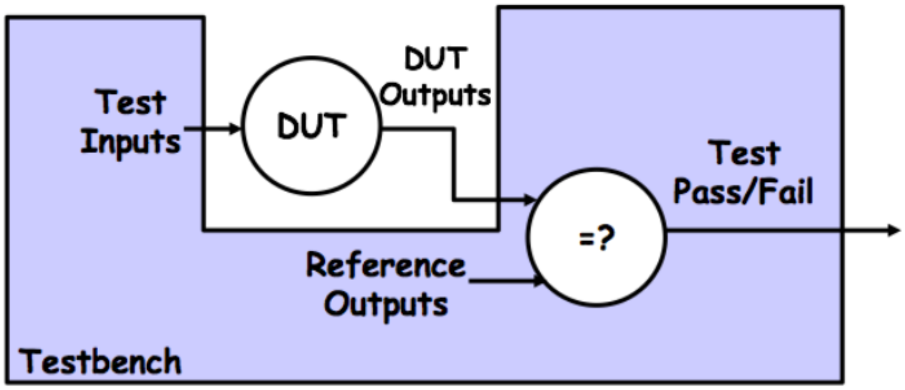
Verification is an intrinsic part of the design process. Verification is as important as design. Verification infrastructure should be available before design and it is good practice to write the tests first.

- use a known or accepted reference results and prove your design's equivalence
- a GUI is not practical for verification

DUT: Design Under Testing

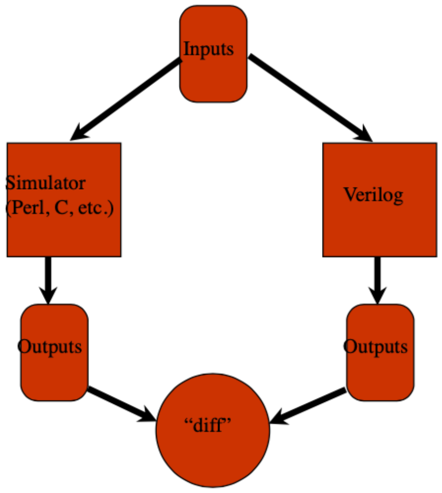
All the testbenches have the same principle: the testbench wraps around Design Under Testing

- inputs are driven by verification environment
- reference from golden design checks that the DUT results match golden results
- not always a trivial task, as the format of outputs can be different



A Simple but Flexible Test Bench

A tried and true method of design verification is **File I/O**.



[Example: A simple and flexible testbench]

$readmemb(filename, memory block): gets file and first row incrementing counter cmd is used to get lines from test data

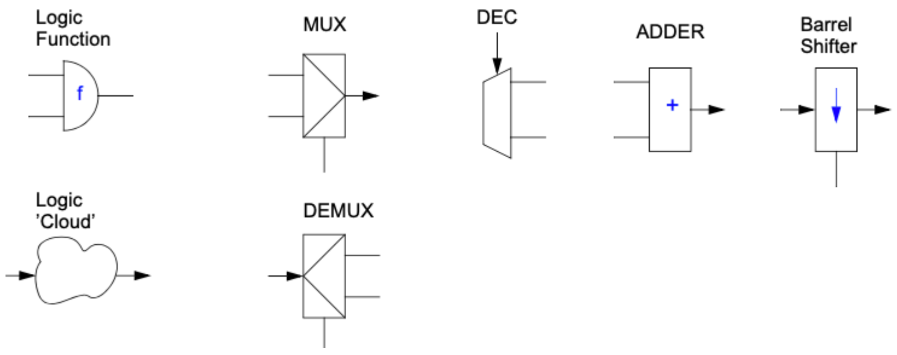
```
integer    cmd;
reg [TEST_BITS-1:0]  testdata[0:CMDNUM];

initial
$readmemb(CMDFILE,testdata);

// decode the test instruction
wire [2:0] A=testdata[cmd][32:30];
wire [2:0] B=testdata[cmd][29:27];
// etc.

always @(posedge clk) begin
      if (!rst_n)
        cmd <= 0;
      else
        cmd <= cmd+1;
      end
integer outfile = fopen(“out.dat”);
$fdisplay(outfile,"%d A=%b B=%b OUT1=%h OUT2=%4h”
                 $realtime,A,B,OUT1,STATE);
```

File I/O is great, but What should my inputs be?

1. **Focused Testing**
    a. write handmade tests for corner cases
    b. focused testing is difficult to have for large tests
2. **Random Testing**
    a. perform random operations at random times, check assertions
    b. random testing can find problems in handshake protocols, timing issues, etc
    c. can easily run for millions of cycles
    d. can be combined with coverage analysis: there are tools that measure what states/components have been visited in the design
3. **Formal verification**
    a. tools that will prove assertions about a design
    b. examples
        i. "We will never enter an invalid state"
        ii. "we will never end up in deadlock"

Design Elements

Basic Design Elements



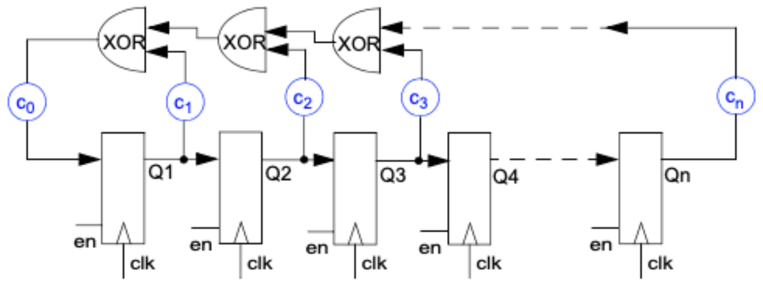
LFSR: Linear Feedback Shift Register

LFSRs generate pseudorandom numbers and have little overhead.

- They consist of a Flip Flop chain with feedback paths.
- LFSR structure is described by a polynomial

$$P(x) = c_0 + c_1x^1 + c_2x^2 + \ldots + c_nx^n$$

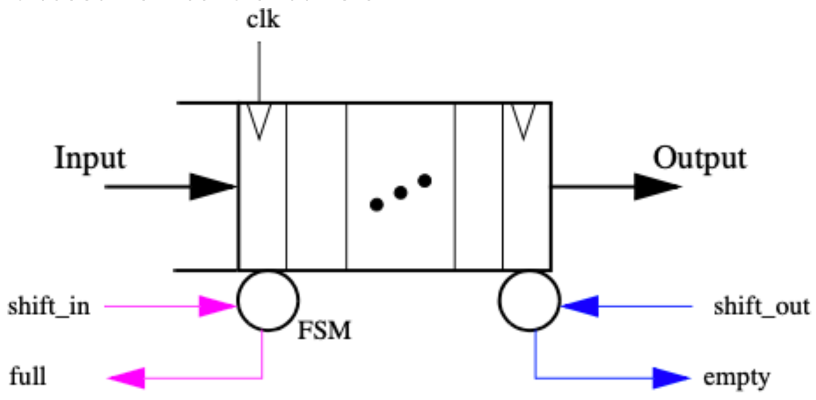
n: number of FFs
c_i: respective FF output XORed into the feedback path.



FIFO Data Structure

FIFOs are an extremely important data structure in HW.

- they can decouple s fixed timing relation between modules
- interfaces slow and fast components, even between different block domains
- spread/smear out bursty traffic
- used for credit-based flow control buffers



FIFO Data Structure Function Table

We want to be able to inject values at a certain place depending on current state. The table below shows how to adjust the count of items in the FIFO.

Special Care: watch out for

- writing to a full FIFO
- reading from an empty FIFO
- who is responsible for these cases? the FIFO or the user?

shift_in	shift_out	function performed
0	0	hold (do nothing)
1	0	shift in a value, cnt++
0	1	shift out a value, cnt--
1	1	shift in a value and shift out a value

Implementations of a FIDO data structure are either

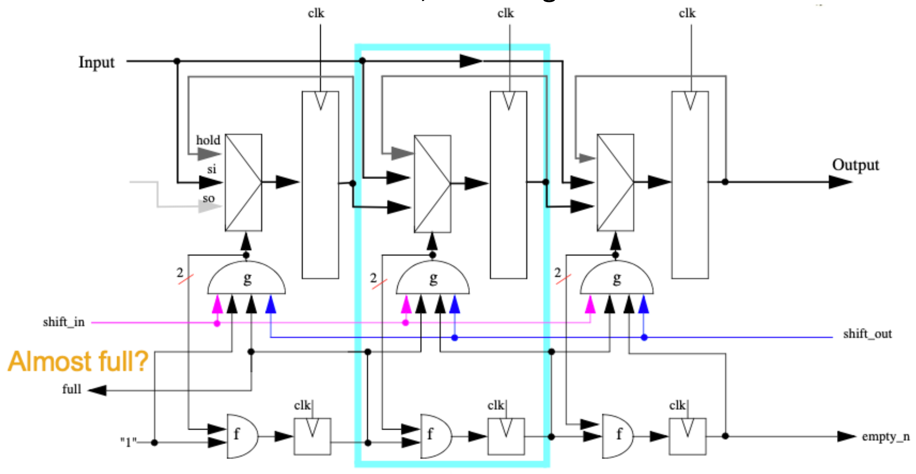
- register based
- RAM based
- IP block based

Register-based FIFO Data Structure

There are input paths to all registers in the FIFO. where input goes depends on the count of items in the FIFO. For each stage of the FIFO you must know:

1. what is the **count** of the FIFO
2. what is the **register position** in the FIFO
3. is there a **shift in/shift out signal**

Put them all in a case statement, each FF gets a case statement on what to do.



Capacitive load on shift signals scales with depth (~8 stages max)