EEE4120F

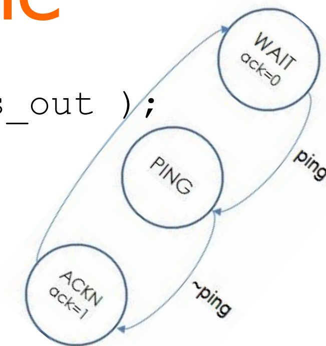
High Performance Embedded Systems

Lecture 16

More Verilog & Statemachine

```
module myveriloglecture ( wishes_in, techniques_out );
   …
     // implementation of today's lecture
   …
endmodule
```

Lecturer:
Simon Winberg

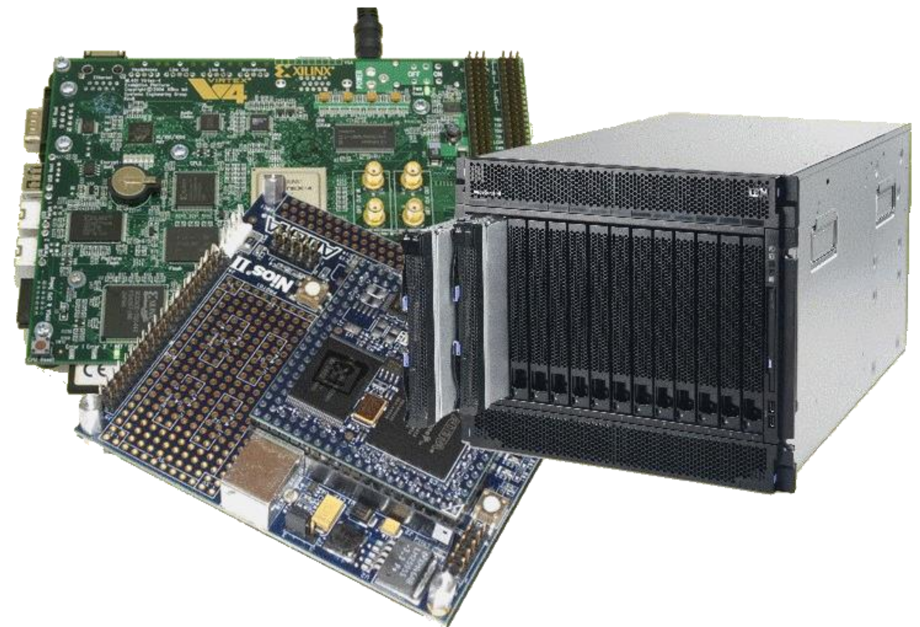


statemachine

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

Lecture Overview

- Brief recap
- Busses and Endianness
- Functions in Verilog
- Implementing state machines



Module declaration (recap)

- Modules are defined in Verilog as follows:

```
module mymodule (a, b, x, y);
 input      a, b;   // these are inputs, usually first
 output     x, y;   // outputs, listed usually after inputs
 x <= or(a,b);     // these all executed at same time:
 y <= and(a,b);
endmodule    // no semicolon needed after endmodule
```

Initial block

- The initial block is like a constructor for a Verilog module in simulation. It is activated the first time the module starts up.
- Used in simulation to set up conditions and to implement test benches.

```
module testbench;   // top level module
 wire  a, b, x;      // set up some signals
 add  myadd(a,b,x); // module to test
 myAnd_tb1 tb(a, b, x); // use this test
endmodule

module myAnd_tb1(a,b,x);
  input a,b;
  output x;
  reg    a, b; // registered inputs
  initial begin
    // log these signals as follows:
    $monitor ($time,
        "a=%b, b=%b, x=%b", a, b, x);
    // exercise the signals
    a = 0; b = 0;
    #10 b = 1;
    #10 a = 1;
    #10 b = 0;
    #10 $finish; // tell simulator to quit
  end // end initial
endmodule
```

Monitor : a standard Verilog simulation operation

- The monitor operation is sensitive to a selection of signals. Whenever one of the signals change, it prints out whatever is requested to be printed, using a printf type formatting

$monitor(text,signals) — give it a string (or multiple strings) followed by a list of variables.
e.g. $monitor(“mysigs: ”, “a=%b b=%b:”, a,b);

Examples:
$monitor($time,”a=%b”,a);   // displays time value followed by a=xx (bin val)
                                              // $time is actually a string value

The always@ (sensitivity_list)

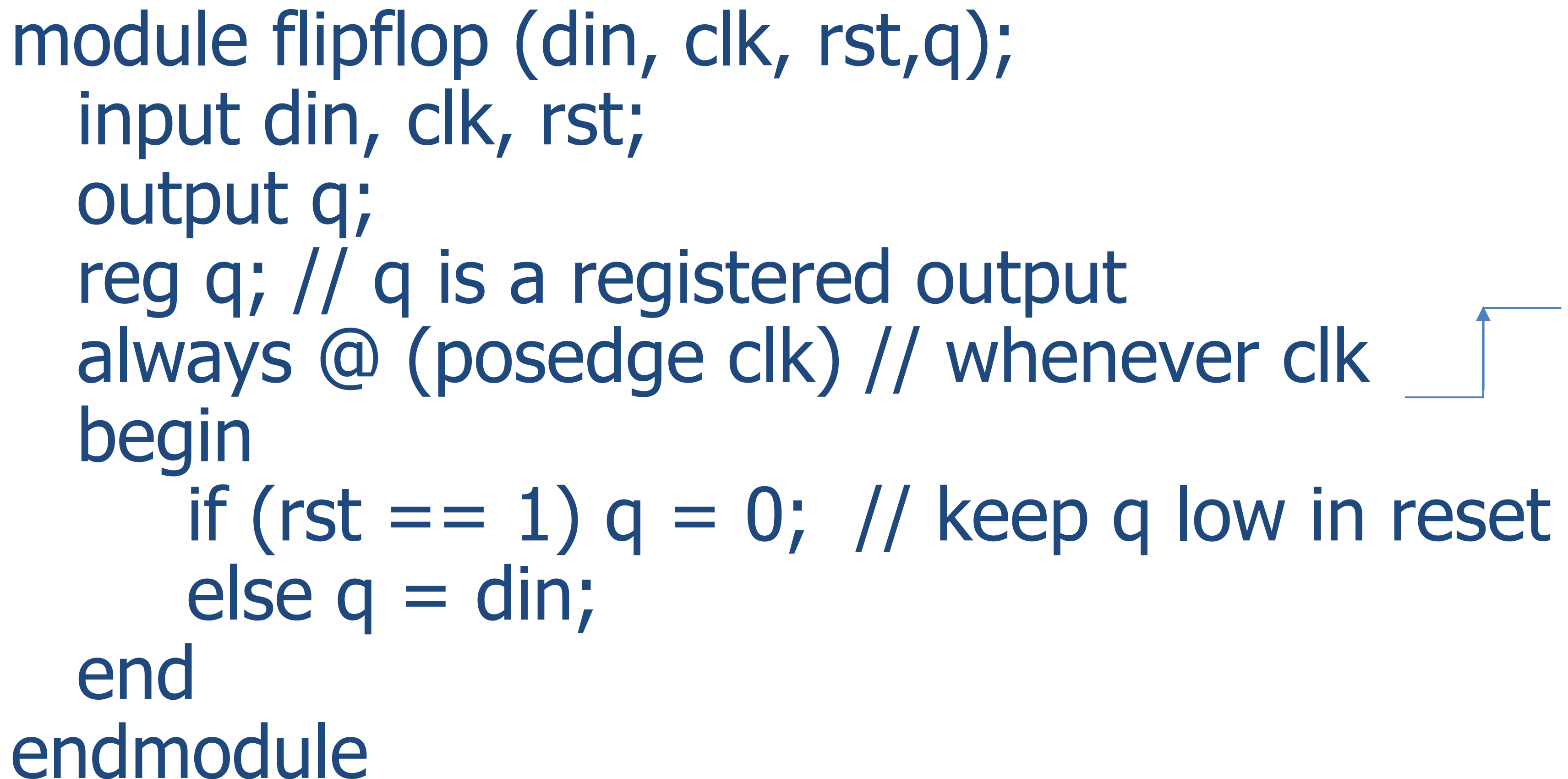
- The always@ expression is used within a Verilog module to group operations that activate whenever the sensitivity list is triggered
- Syntax:

```
always @ (<sensitivity1, sensitivity2, ...>)
  begin
      <actions>
  end
```

Used in behaviour descriptions and statemachines

Example: Implementing a D-type flip-flop …

always@ Example : D-type Flip Flop



Busses & endian

- Busses or bit signal vectors are specified as follows:
  - reg [20:0] dataA; // little endian LSB in bit 0
  - reg [0:20] dataB; // big endian MSB in bit 0

It doesn’t really matter if you are using them just as busses, it is only relevant when applying operations such as add.

Question: Can you say dataA <= dataB without an error?

Functions in Verilog

functions are not modules in Verilog

- Functions can be used as macros within the body of a Verilog module.
  - These can effectively save typing.
  - They work differently to module instantiations.
  - These are defined inside a module.
  - Can only have input parameters
- Example:

```
function [31:0] negate;
  input [31:0] a;
  negate = ~a;
endfunction
```

```
reg [15:0] a;
wire [15:0] x;
assign b = negate (a);
initial begin
  a=10;
  a = add(1, a);
  $display(" a=%b -a=%b”, a, b);
end
```

Example function: Converting endianness

- If need be you can construct a function to convert endianness, e.g.:

```
function [31:0] toBigEndian;
   // transform data from little-endian to big-endian
   input [31:0] x;
   toBigEndian = {x[7:0], x[15:8], x[23:16], x[31:24]};
endfunction
```

Vectors & Signal concatenation { }

Syntax for concatenating wires: **{ x1, x2, … xn }** the collective can be used just the same as any other variable.

```
module adder4 (a, b, cin, sum, cout);
   input [3:0] a, b;     // 2x 4 bit vector inputs
   input cin;            // carry input
   output [3:0] sum; // 4-bit little endian vector
   output cout;          // carry out
   // perform the adder operation
   assign {cout,sum} = a + b + cin;
     // the leftmost is MSB since it is little endian
endmodule
```

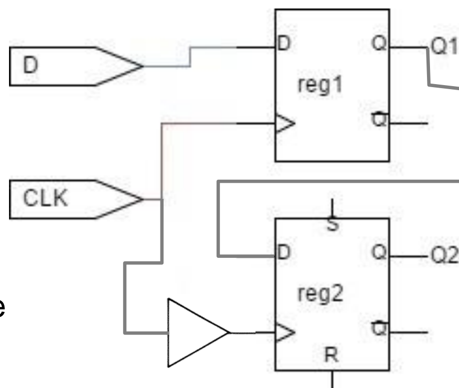
i.e. if the input is little endian so is the output
If you said {sum,cout}=A

Blocking/Non-blocking statemets

Blocking

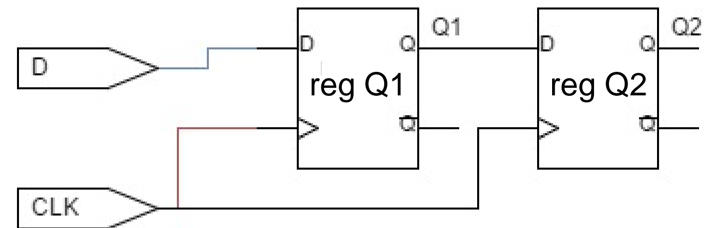
```
module blockFFs (
  input D, input clk,
  output reg Q1, output reg Q2)
always @ (posedge clk)
 begin
   Q1 = D;
   Q2 = Q1;
 end
endmodule
```

This diagram is fairly accurate to what is happening in the blocking case. Reg2 is slightly delayed so that the first operation, Q1 = D; completes first before the Q2 = Q1 is done.



Non-blocking

```
module nonblockFFs (
  input D, input clk,
  output reg Q1, output reg Q2)
always @ (posedge clk)
 begin
   Q1 <= D;
   Q2 <= Q1;
 end
endmodule
```



On a clock, value of D feeds into Q1 and value of Q1 at same point, before Q1 changes to D, feeds into Q2.

See simulation example of these cases at: https://www.edaplayground.com/x/9dxx

Implementing a FSM

Creating finite state machines in Verilog

Statemachines

- A state machine has:
  - Input events
  - Output events
  - Set of states
  - A function that maps
    (state,input) → (state,output)
  - A indication of the initial state
- A Finite State Machine (FSM) has a limited number of states

Implementing a FSM with Verilog

- The state machine needs a register to store its state
- It is sensitive to zero or more inputs, which can change state and/or produce an output

The state register

- States could be numbered in sequence 0 .. 2^n-1 where n is the number of bits for the state.
- Or a different encoding / ordering scheme could be used to make state changes more robust, e.g.:
  - use of grey scale or 'one hot' encoding (where 'one hot' means there is just one pin in the state set at a time)

State	Binary	Gray	One Hot
0	3'b000	3'b000	8'b00000001
1	3'b001	3'b001	8'b00000010
2	3'b010	3'b011	8'b00000100
3	3'b011	3'b010	8'b00001000
4	3'b100	3'b110	8'b00010000
5	3'b101	3'b111	8'b00100000
6	3'b110	3'b101	8'b01000000
7	3'b111	3'b100	8'b10000000

The states and state changes

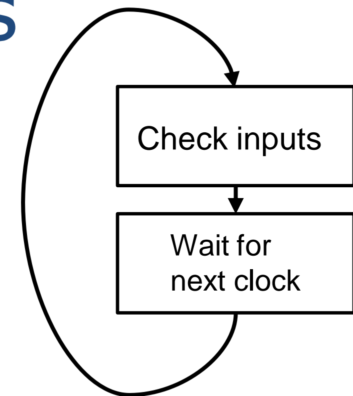
- Need a means to force initial state at startup (i.e. reset state register)
  - Typically a reset handler does this.
- Specifying and changing states
  - Usually a case construct is used to define the states, but could use ifs and elses
- Need to decide if the statemachine
  - is synchronous (clocked) or
  - asynchronous (activates whenever an input changes)
- May need recovery mechanism (e.g. watchdog or recovery default state)

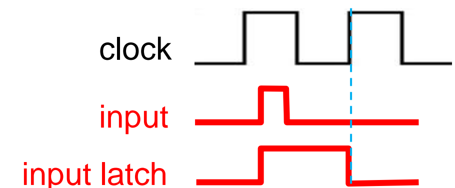
State-machine triggering

- There are three basic approaches to state-machines:
  - Clock-triggered: Those that are trigged only by the clock (a hot-running FMS)
  - Clock-disciplined: Those that are activated only on a clock pulse
  - Input/event triggered: triggered by various events possible including the clock

Clock-triggered state machine

- Also called “polled” or “hot loop”
- May trigger on: posedge, megedge or level of clock
- Generally respond to edge levels of inputs checked each clock
- But beware:
  - Danger if missing inputs that change rapidly or do not say on the required level for more than a clock pulse (i.e. Nyquist sampling problem).
  - May need ‘latch’ or ‘locking’ circuit to grab inputs that may change rapidly.





Posedge triggered FSM responds only here to the input blip

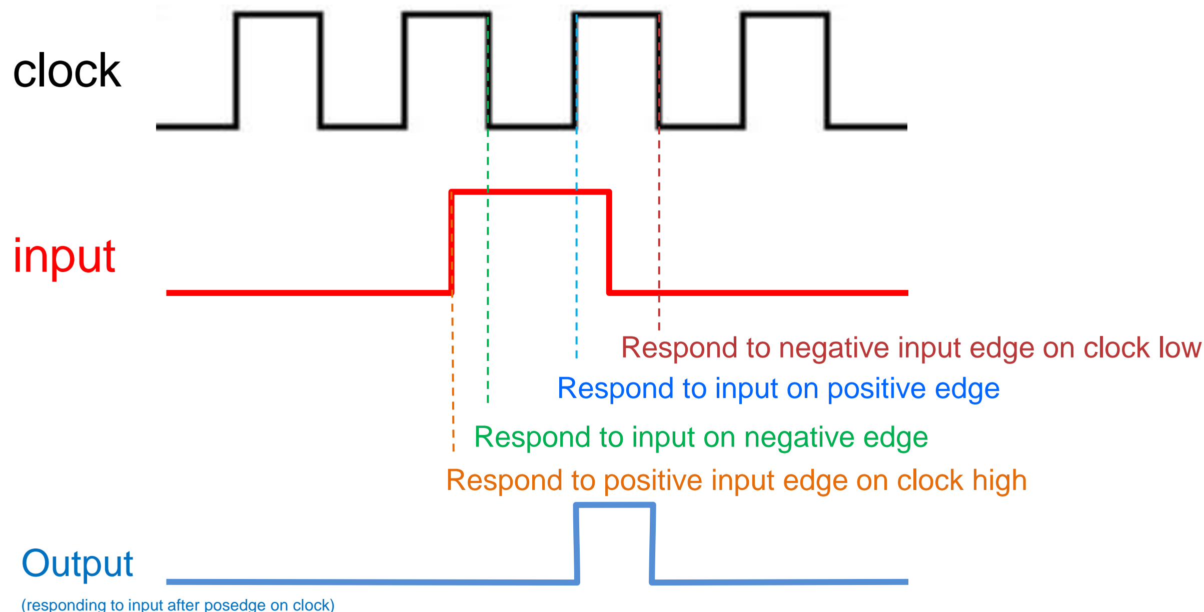
Pros and cons…

Benefit
- Easier to implement and (to some extent) debug.

Drawback
- Might miss input transitions.
- May need latch. Possibly less efficient.

Clock-disciplined state machine

- Or clock-synchronized state machine
- This design is one that responds to inputs but only after at clock pulses (or on suitable levels of a clock pulse)



Pros and cons…

Benefit

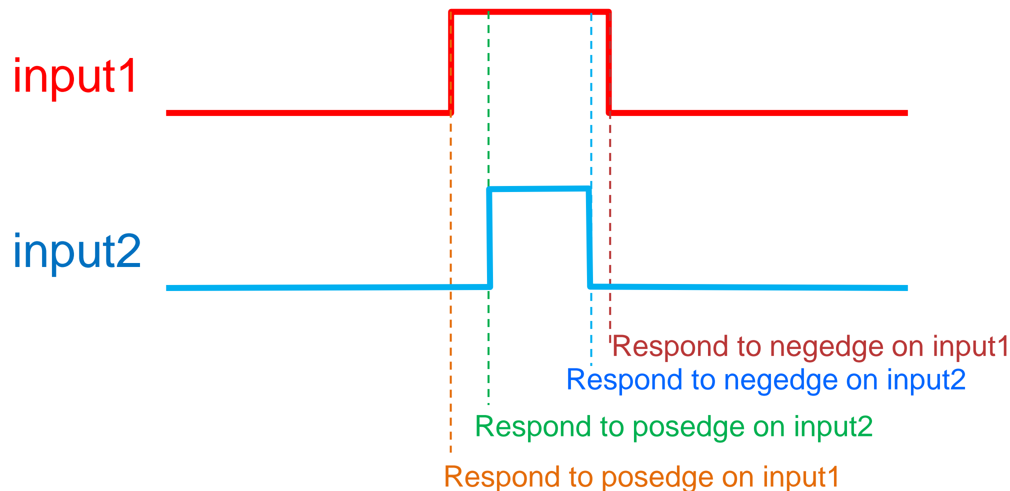
- Generally the ‘preferred’ professional approach
- Robust
- Easier to debug

Drawback

- More effort to code
- Not as responsive as a purely event triggered state machine (e.g. might miss a input changing rapidly between clocks)

Event-triggered state machine

- This type of a state machine triggers on particular inputs, possibly including clock changes. You might see a Verilog sensitivity list using '*' to say trigger on anything that is read in the code block, i.e. whether a positive or negative edge.



Pros and cons…

Benefit

- likelihood of your code working, i.e not neglecting something in the sensitivity list
- Easier, less thinking work needed

Drawback

- is it can be wasteful
- Change of quickly re-entering the statemachine if inputs change very quickly but with a tiny delay

Example Statemachine

```
// Code your design here
module alivefms (input clk, input reset, input ping,
output reg ack,
       output reg[1:0] state);
 parameter [1:0] WAIT  = 2'b11;
 parameter [1:0] PING  = 2'b01;
 parameter [1:0] ACKN  = 2'b10;
 always @(posedge clk or posedge reset or
          posedge ping or negedge ping)
 begin
  if (reset)
  begin
     state <= WAIT;
     ack   <= 0;
  end else
  begin
    case(state)
      WAIT:    begin
               if (ping) state <= PING;
                   else  state <= WAIT;
               ack <= 0;
               end
      PING:    if (ping) state <= PING;
                   else  state <= ACKN;
      ACKN:    begin
                ack <= 1;
                state <= WAIT;
               end
      default: state <= WAIT;
    endcase
  end
end
endmodule
```

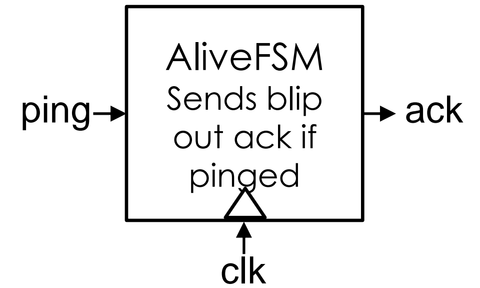
← **Activation**

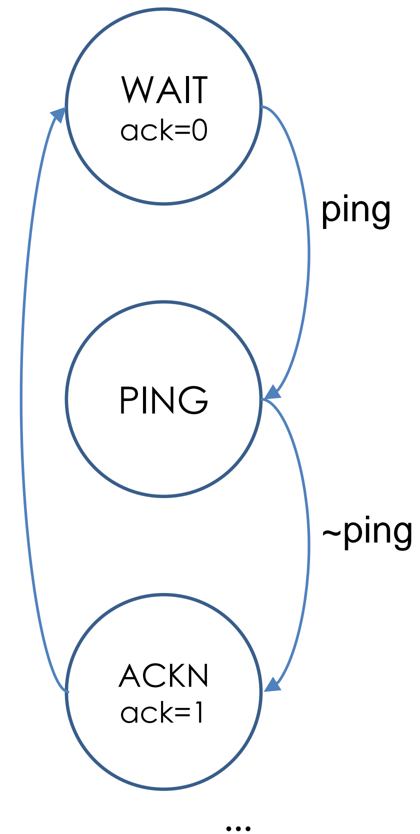
(i.e. activation is clock and event-triggered)

← **Start up**

← **States**

← **Recovery**





Notes: Must only send ack only after ping goes low

Create a testbench

```
// testbench for alivefsm
module alivefms_tb ();
reg  clk, reset, poke;
wire ack;
wire [1:0] state;
integer i;
  // instantiate the FSM
  alivefms alivefsm_tb (clk,reset,poke,ack,state);

initial  // method for testing the FSM:
begin
   // enable monitoring of wires of interest
  $monitor("reset=%d state=%d poke=%d ack=%d\n",
           reset,state,poke,ack);
   clk = 0; reset = 1; #5    // do the reset
   reset = 0; clk=0; #5
   poke = 1;  clk=1; #5   // poke the fsm
   poke = 0; #5           // release poke
   clk  = 0; #5           // needs a clk transition
   clk  = 1; #5
   if (ack == 1)          // check if worked as planned
      begin
        $display("SUCCESS\n");
      end
end
endmodule
```

Avail online at: https://www.edaplayground.com/x/25N5

Run on iverilog (or other sim)

```
$ iverilog '-Wall' design.sv testbench.sv && unbuffer vvp a.out
reset=1 state=3 poke=x ack=0

reset=0 state=3 poke=x ack=0

reset=0 state=1 poke=1 ack=0

reset=0 state=2 poke=0 ack=0

reset=0 state=3 poke=0 ack=1

SUCCESS

Done
```

That's essential it for state machines

But I am giving an impression of over-simplification… state machines can get challenging. But they are considered by many (including me) as recommended, indeed the appetizing *defecto favorito*, for elegant and understandable Verilog.

BUT WAIT… THERE'S MORE!!

Let me throw you a more tasty and real problem to try....

Take-home Activity

A (hypothetical) cosmic ray detector system is able to detect particles passing its sensors[1]. But there is also an occasional backscatter effect or false detection (caused e.g. by reflected energy) which can also be detected. The aim is to have a system that counts the valid particles detections and also tallies false event (which may actually be true detections but is determine later in post-processing)… technically the backscatter could be determined by the trajectory of particles if there are multiple detectors panels, but we won't get into those complications).

The diagram on right illustrates the situation.

A C program was written to explain what this detector needs to do (see next slide…) but even C is hopelessly too slow for this application; the code it is only provided to explain what the system needs to do.

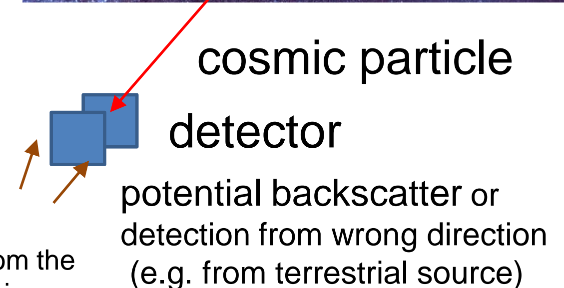
TODO:

Attempt a Verilog state machine for this problem

Implement a Verilog equivalent of the particle detector code that will count the number of true particle detections *tp* (when backscatter is low), as well as the number of potential false detections *fp* (when backscatter is high) for each slow clock (sclk) period.

See further details in the code on the next slide and module interface on slide 28.

Notes: You can make various simplifications or just attempt to implement a part of the task. This exercise is provided to get you practicing your Verilog, as well as seeking the awesome responsive power of Verilog and FPGA systems 😉



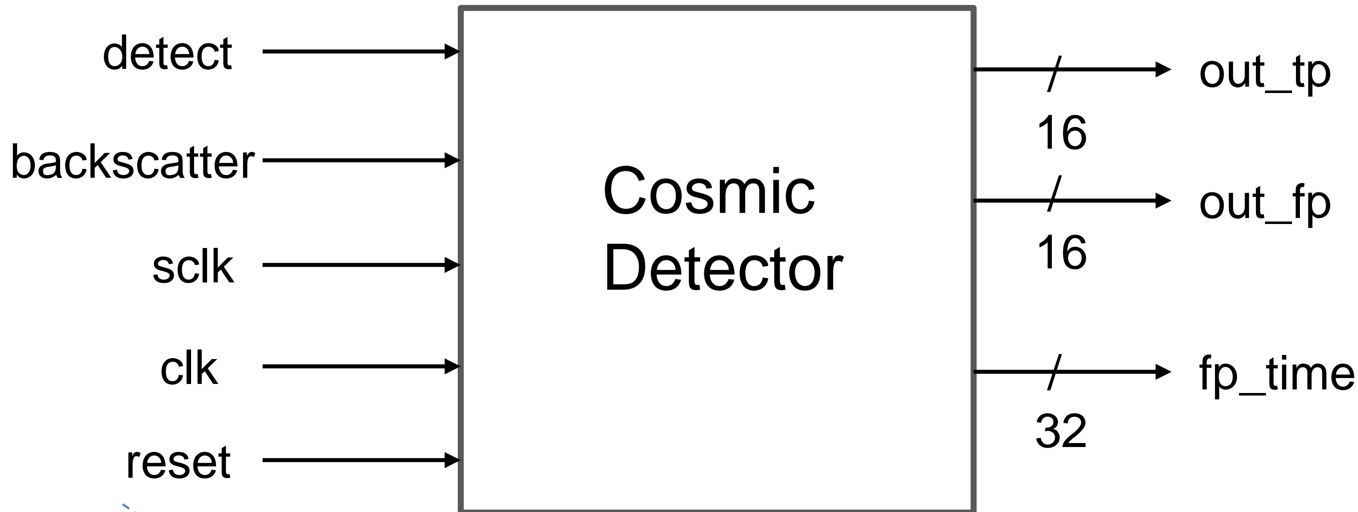
1 Note: technically it is not the cosmic particle itself that is detected but one of many particles from the grammar-ray shower that result from a super high-energy cosmic particle colliding with e.g. an air particle in the atmosphere. Info on this at https://en.wikipedia.org/wiki/Cosmic-ray_observatory

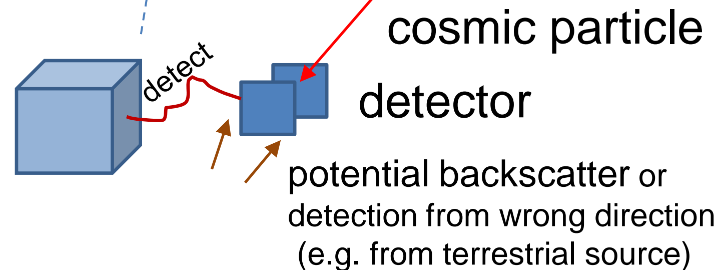
//! FPGA Particle Detector Simulation ('golden' measure)

```
#include <stdio.h>
#include <stdlib.h>

int printlog = 1;  //! Debug various, set to 1 to generate log report
unsigned sclk    = 20;  // slow clock  (i.e. observation interval)
unsigned clk     = 0;   // fast clock

int detect_particle()
{   //! A dummy function to simulate input indicating particle is present

    // detecting particles when sclk%4  and clk is %4
    if (sclk%4==0) if (clk%4==0) return 1;

    return 0;
}
int detect_backscatter()
{   //! A dummy function to simulate reading an input

    // detecting backscatter particles when sclk==8  and clk is 5
    if (sclk==8) if (clk==5) return 1;

    return 0;
}

void out_tp(unsigned tp)
{
    //! simulate outputting value of tp
}

void out_fp(unsigned fp)
{
    //! simulate outputting value of fp
}

void out_fptime(unsigned time_fp)
{
    //! simulate outputting value that fp backscatter triggered
}

….  cont →

int main()
{
    unsigned count   = 0;   // timer counter
    unsigned time_fp = 0;
    unsigned countdown_fp = 0;
    int      backscatter = 0;
    unsigned  tp;  // number true particles
    unsigned  fp;  // number of false particles / backscatter

    if (!printlog) printf("Particle detector simulation!\n");
    if (printlog) printf("sck,clk,tp ,tp ,b,tfp\n"); // print headings out log output

    while (sclk)  // slow clock
    {
        out_tp(tp); out_fp(fp);
        out_fptime(time_fp);
        time_fp = 0;
        tp = fp = 0;
        clk=10; // fast clock
        while (clk) {
            count++;
            if (countdown_fp) {
                countdown_fp--;
                if (countdown_fp==0) backscatter = 0;
            }
            if (detect_backscatter() && (countdown_fp==0)) {
                time_fp = count;
                countdown_fp = 10;
                backscatter  = 1;
            }
            if (detect_particle()) {
                if (backscatter == 0) tp++; else fp++;
            }
            if (printlog)
printf("%03d,%03d,%03d,%03d,%01d,%03d\n",sclk,clk,tp,fp,backscatter,time_fp);
            clk--;
        }
        sclk--;
    }
    return 0;
}
```

main.c

Cosmic Detector Module





Disclaimers and copyright/licensing details

I have tried to follow the correct practices concerning copyright and licensing of material, particularly image sources that have been used in this presentation. I have put much effort into trying to make this material open access so that it can be of benefit to others in their teaching and learning practice. Any mistakes or omissions with regards to these issues I will correct when notified. To the best of my understanding the material in these slides can be shared according to the Creative Commons "Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)" license, and that is why I selected that license to apply to this presentation (it's not because I particularly want my slides referenced but more to acknowledge the sources and generosity of others who have provided free material such as the images I have used).

Image sources:

man working on laptop – flickr

scroll, video reel, big question mark – Pixabay http://pixabay.com/ (public domain)

References: Verilog code adapted from http://www.asic-world.com/examples/verilog