4. Design With Verilog

Dr. Paul D. Franzon

Outline

1. Procedural Examples
2. Continuous Assignment
3. Structural Verilog
4. Common Problems
5. More sophisticated examples

Always Design Before Coding

4.3 Common Problems, Fixes, Functions, Tasks and Arrays

Dr. Paul D. Franzon

Outline

1. Common Problems and fixes
2. Functions, Tasks illustrated via an LFSR
3. Arrays of vectors, illustrated via a Register File
4. Exercise

Objectives and Motivation

Objectives

- Describe the four most common problems that occur in HDL-based design and how to fix them.
- Understand how functions, tasks and arrays can be used in synthesizable code.
- Describe how memories and register files can be modeled.

Motivation

- Useful tools in your repertoire

References

- Ciletti:
  - Inside front cover : Summary
  - Section 5.10: A different LFSR
  - Section 5.13: Functions and tasks
  - Section 5.16: Register file
- Smith and Franzon
  - Section 6.8 : Common Problems and fixes
- Sutherland Reference Guide

Common Problems and Fixes

Unintentional Latches

- How to detect : Found by Synopsys after "read" command
- How to fix : Make sure every variable is assigned for every way code is executed (except for flip-flops)
- What happens if unfixed : Glitches on "irregular clock" to latch cause set up and hold problems in actual hardware (➔ transient failures)

Problem Code :

```
always@(A or B)
  begin
    if (A) C = ~B;
    else D = |B;
  end
```

Possible Fix :

```
always@(A or B)
  begin
    C = x; D = x;
    if (A) C = ~B;
    else D = |B;
  end
```

Common Problems and Fixes

Incomplete Sensitivity List

- How to detect : After "read" command synopsys says "Incomplete timing specification list"
- How to fix : All logic inputs have to appear in sensitivity list OR switch to Verilog 2001 (`always@(*)`).
- What happens if unfixed : Since simulation results won't match what actual hardware will do, bugs can remain undetected

Problem Code

```
always@(A or B)
  begin
    if (A) C = B ^ A;
    else C = D & E;
    F = C | A;
  end
```

Fix

```
always@(A or B or D or E)
  begin
    if (A) C = B ^ A;
    else C = D & E;
    F = C | A;
  end
```

Note : C not in list (internal signal to block, not an input)

Common Problems and Fixes

Unintentional Wired-OR logic

- How to detect : After "read" command synopsys says "variable assigned in more than one block"
- How to fix : Redesign hardware so that every signal is driven by only one piece of logic (or redesign as a tri-state bus if that is the intention)
- What happens if unfixed : Unsynthesizable.

This is a symptom of NOT designing before coding

Problem Code

```
always@(B)
  C = |B;

always@(E)
  C = ^E;
```

Possible Fix

```
always@(A or B or E)
  if (A) C = |B;
  else C = ^E;
```

Common Problems and Fixes

Improper Startup

- How to detect : Can't
- How to fix : Make sure "don't cares" are propagated
- What happens if unfixed : Possible undetected bug in reset logic

Problem Code

```
always@(posedge clock)
  if (A) Q <= D;

always@(Q or E)
  casex (Q)
    0 : F = E;
    default : F = 1;
  endcase
```

```
clock : 00001111
    D : xxxxxxxx
    E : 11110000
    Q : xxxxxxxx
    F : xxxx1111
```

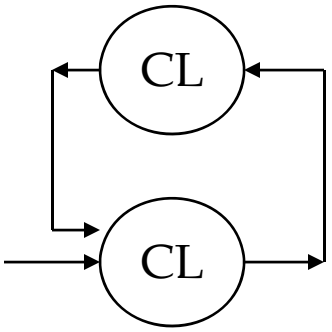
Fix

```
always@(Q or E)
  casex (Q)
    0 : F = E;
    1 : F = 1;
    default : F = 1'bx;
  endcase
```

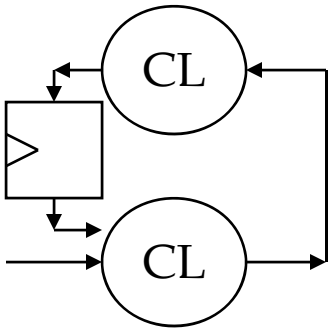
Common Problems and Fixes

Feedback in Combinational Logic

- Either results in:
  - Latches, when the feedback path is short
  - "Timing Arcs", when feedback path is convoluted
- Fix by redesigning logic to remove feedback
  - Feedback can only be through flip-flops



WRONG!



OK

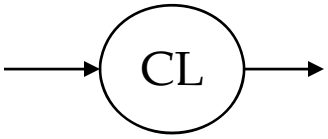
Common Problems and Fixes

Incorrect Use of FOR loops

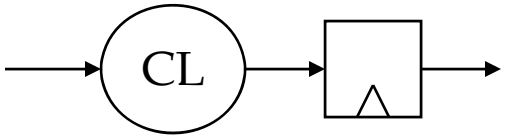
- Only correct use is to iterate through an array of bits
- If in doubt, do NOT use a FOR loop

Unconstrained Timing

- To calculate permitted delay, Synthesis must know where the flip-flops are
- If you have a path from input port to output port that does not path through a flip-flop, Synopsys can not calculate the timing
  - Timing Report presents "Unconstrained Paths"
- Fix: revisit module partitioning (see later) to include flip-flops in all paths



Causes Problems



OK

Common Problems and Fixes

Unconnected Nets and wires

- How to detect: check_design
- Why is this bad:
  - Might indicate other problems that are true faults
  - At best is wasteful
- How to fix:
  - Make sure all wires within and between modules internal to your design are connected at both ends
- Often occurs with unconnected bits of adders, etc.
  - OK if these connections are not needed

Debug

- How to prevent a lot of need to debug:
  - Carefully think through design before coding
  - Simulate "in your head" or on paper
    - Produce a timing diagram before coding

- How to debug:
  - Track bug point back in design and back in time
    - Check if each "feeding" signal makes sense
  - Compare against a hand derived simulation
  - If all else fails, recode using a different technique

Summary

Problem	Detection	Repair
Unintentional Latches	Reported after read	Remove implied memory
Incomplete Sensitivity List	Reported after read	Complete or use *
Wired-or logic	Reported after read	Redesign
Improper startup	Can't	Propagate unknowns
Combinational Logic feedback	"Timing arcs" in synthesis	Redesign
Incorrect use of for loops and other control constructs	Unanticipated behavior	Redesign
Unconstrained timing	"Unconstrained timing" in synthesis	Flip-flop(s) in every path between input and output in each module

→ Quiz and next sub-module

Functions and Tasks

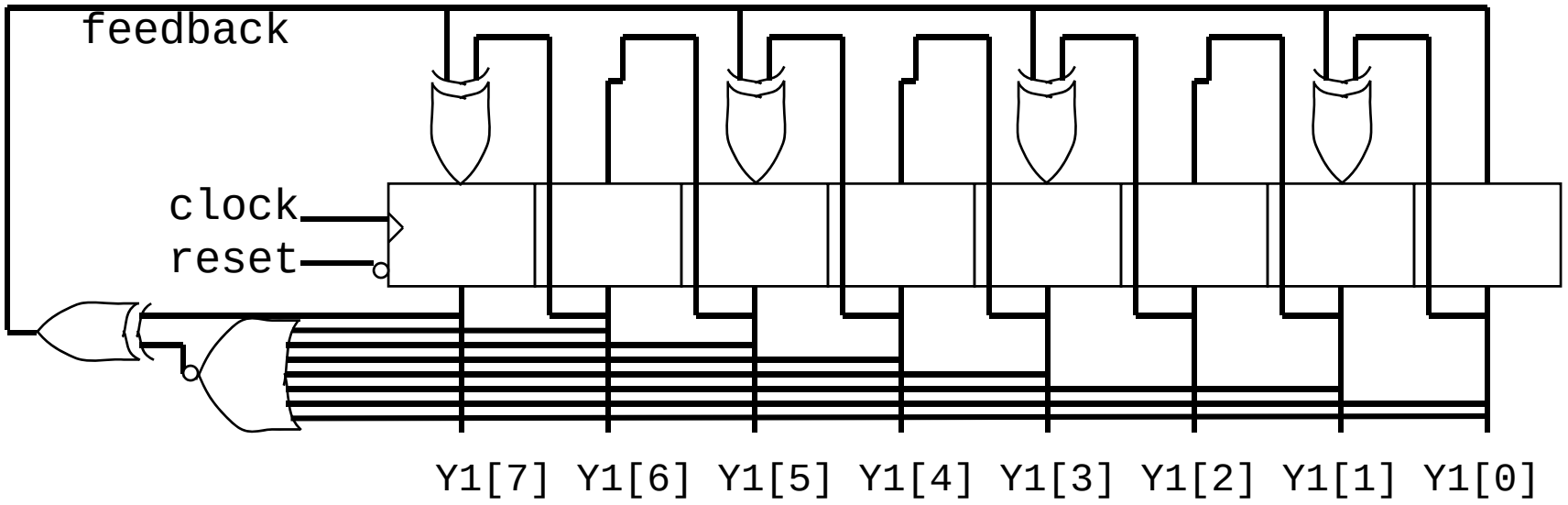
- Function
  - Takes multiple inputs and returns a single output
  - Must be within module its called in
  - Can not contain timing or <=
  - Use for repeated combinational logic or repeated test fixture processes
- Task
  - Can have multiple inputs and multiple outputs
  - Can contain timing (e.g. @) but not if intended for synthesis
  - Must be within module its called in
  - Use for repeated combinational logic or repeated test fixture processes

Larger Examples : Linear Feedback Shift Register (fn)

```
module LFSR_FN (Clock, Reset, Y1, Y2); // produces a pseudorandom sequence
input Clock, Reset;
output [7:0] Y1, Y2; reg [7:0] Y1, Y2;
parameter [7:0] seed1 = 8'b01010101;  // seed defines the start of the
parameter [7:0] seed2 = 8'b01110111;  // pesudorandom sequence

function [7:0] LFSR_TAPS8_FN;
input [7:0] A;
integer N;
parameter [7:0] Taps = 8'b10001110;  // taps defines the sequence
reg Bits0_6_Zero, Feedback;
begin
  Bits0_6_Zero = ~| A[6:0];
  Feedback = A[7] ^ Bits0_6_Zero;
  for (N=7; N>=1; N=N-1)
    if (Taps[N-1] == 1) LFSR_TAPS8_FN[N] = A[N-1] ^ Feedback;
    else LFSR_TAPS8_FN[N] = A[N-1]; LFSR_TAPS8_FN[0] = Feedback;
end
endfunction /* LFSR_TAP8_FN */

/* Build 2 LFSRs using the LFSR_TAPS8_TASK */
always@(posedge Clock or negedge Reset)
  if (!Reset) Y1 <= seed1; else Y1 <=  LFSR_TAPS8_FN (Y1);
always@(posedge Clock or negedge Reset)
  if (!Reset) Y2 <= seed2; else Y2 <= LFSR_TAPS8_FN (Y2);
endmodule
```

LFSR

- Sketch Design



(Note: Taps here is 01010101).

LFSR (Task)

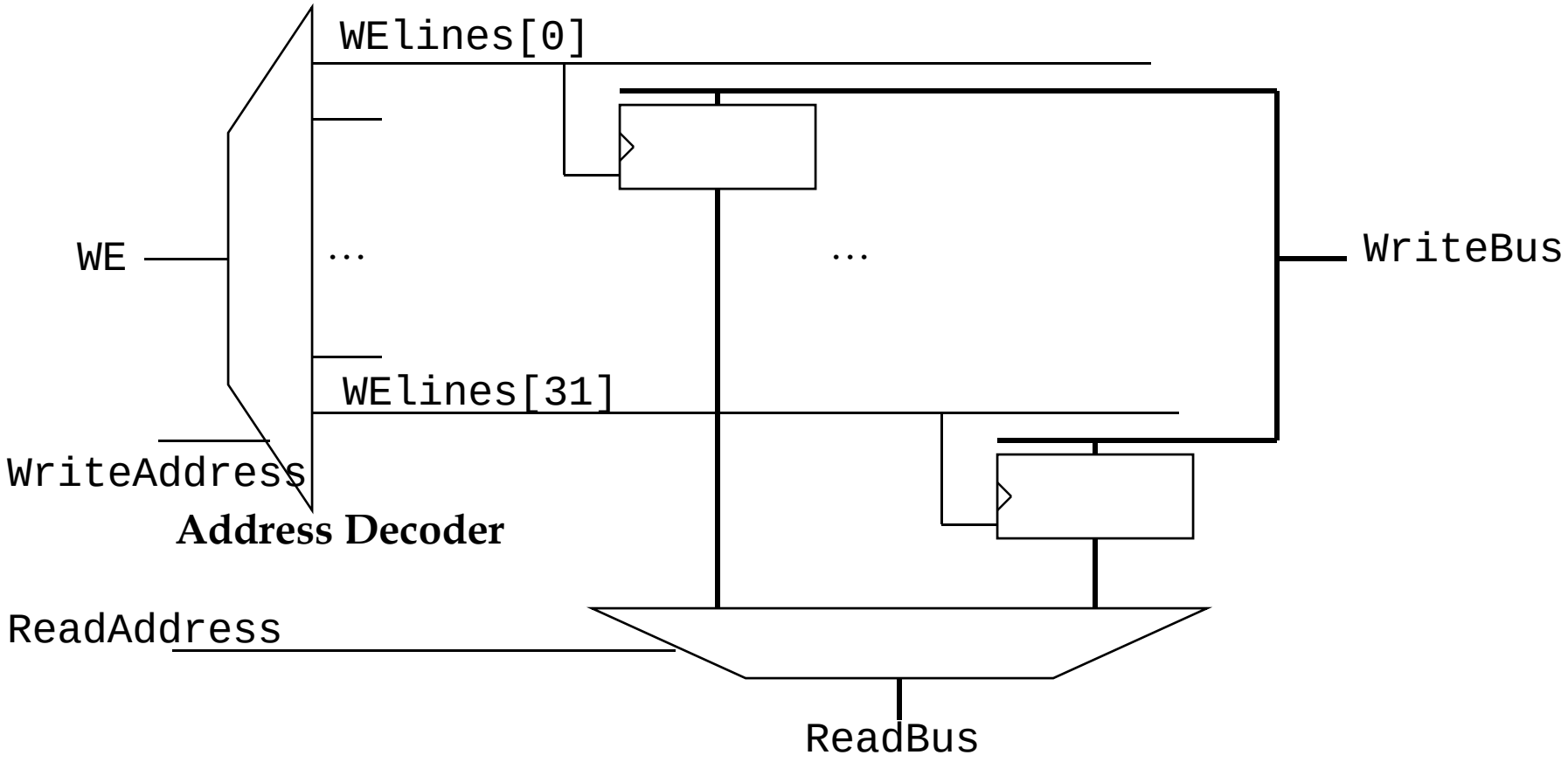
```
module LFSR_TASK (clock, Reset, Y1, Y2);
input clock, Reset;
output [7:0] Y1;
reg [7:0] Y1;
parameter [7:0] seed1 = 8'b01010101;    parameter [7:0] Taps1 = 8'b10001110;

task LFSR_TAPS8_TASK;
input [7:0] A; input [7:0] Taps;       output [7:0] Next_LFSR_Reg;
integer N;     reg Bits0_6_Zero, Feedback; reg [7:0] Next_LFSR_Reg;
begin
  Bits0_6_Zero = ~| A[6:0];    Feedback = A[7] ^ Bits0_6_Zero;
  for (N=7; N>=1; N=N-1)
    if (Taps[N-1] == 1) Next_LFSR_Reg[N] = A[N-1] ^ Feedback;
    else Next_LFSR_Reg[N] = A[N-1];
  Next_LFSR_Reg[0] = Feedback;
end
endtask /* LFSR_TAP8_TASK */

always@(posedge clock or negedge Reset)
  if (!Reset) Y1 = seed1;
  else LFSR_TAPS8_TASK (Y1, Taps1, Y1);
endmodule
```

Register File

```
module RegFile (clock, WE, WriteAddress, ReadAddress, WriteBus, ReadBus);
input  clock, WE;
input  [4:0] WriteAddress, ReadAddress;
input  [15:0] WriteBus;
output [15:0] ReadBus;

reg [15:0]   Register [0:31];   // thirty-two 16-bit registers

// provide one write enable line per register
wire [31:0] WElines;
integer i;

// Write '1' into write enable line for selected register
assign WElines = (WE << WriteAddress);

always@(posedge clock)
    for (i=0; i<=31; i=i+1)
      if (WElines[i]) Register[i] <= WriteBus;

assign ReadBus  =  Register[ReadAddress];
endmodule
```

Register File

- Sketch Design



Notes

- Really, “array of vectors”, not a true 2D array
- Can NOT be passed in port list

Alternative coding:

```
always@(posedge clock)
  if (WE) Register[WriteAddress] = WriteBus;
```

tends to result in more logic, with its implied address decoder.

Submodule Summary

- Task and Function permit complex logic to be built multiple times by calling task and function
  - In synthesis can only be used for combinational logic
- General structure of register file given
  - Can be used for other arrays of registers
  - Explicit address decoder reduces size of design

- Goto sub-module quiz and next sub-module

Sample Problem

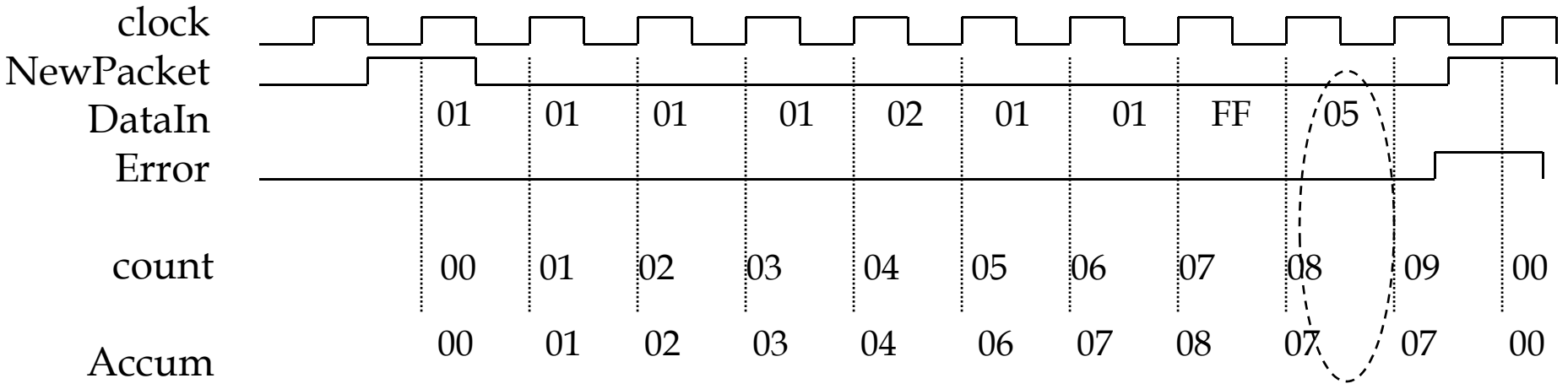
- Checksum Unit:
  - Checksum field used to check that a packet is transmitted correctly
    - Checked against truncated sum of data stored in accumulator
  - e.g. simple 9 byte packet:

    01 01 01 01 01 01 01 FF 06

    Payload Calculated Checksum = 06 (no error in this case)

    01 01 01 01 02 01 01 FF 05

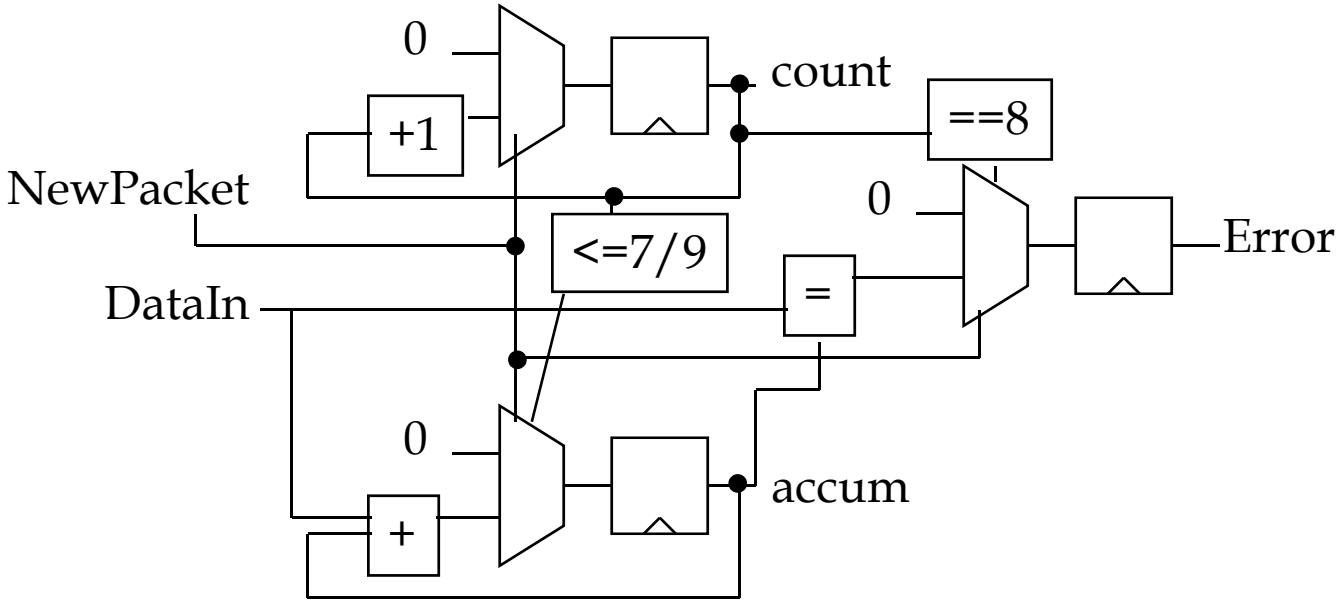
    Payload Calculated Checksum = 07 (ERROR in this case)
  - Design an 8-bit checksum unit with the following timing
    - Inputs : NewPacket (goes high when a new packet starts); DataIn
    - Outputs : Error (hi = error); goes low when a new packet starts

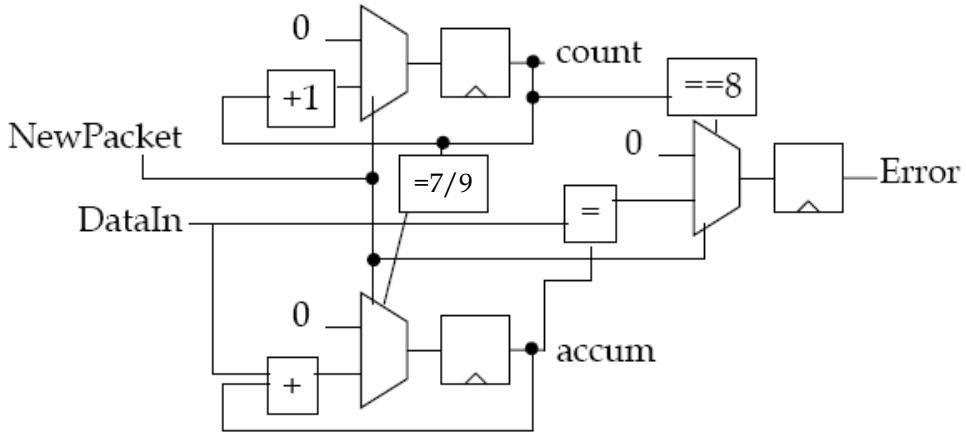


Design

Strategy : Use a counter to identify Byte in packet and event sequence.

1. Draw I/O
2. Identify Registers
3. Describe comb. Logic
4. Sketch timing diagram
4. Controller = counter






```
always@(posedge clock)
  begin
    if (NewPacket)
       begin
         count <= 0;  error <= 0;  accum <= 0;
       end
    else if (count <=7) accum <= accum + DataIn;
    else if (count == 8) error <= ~(DataIn == accum);
    if (count <= 8) count <= count + 1; // will stop at 9
  end
```