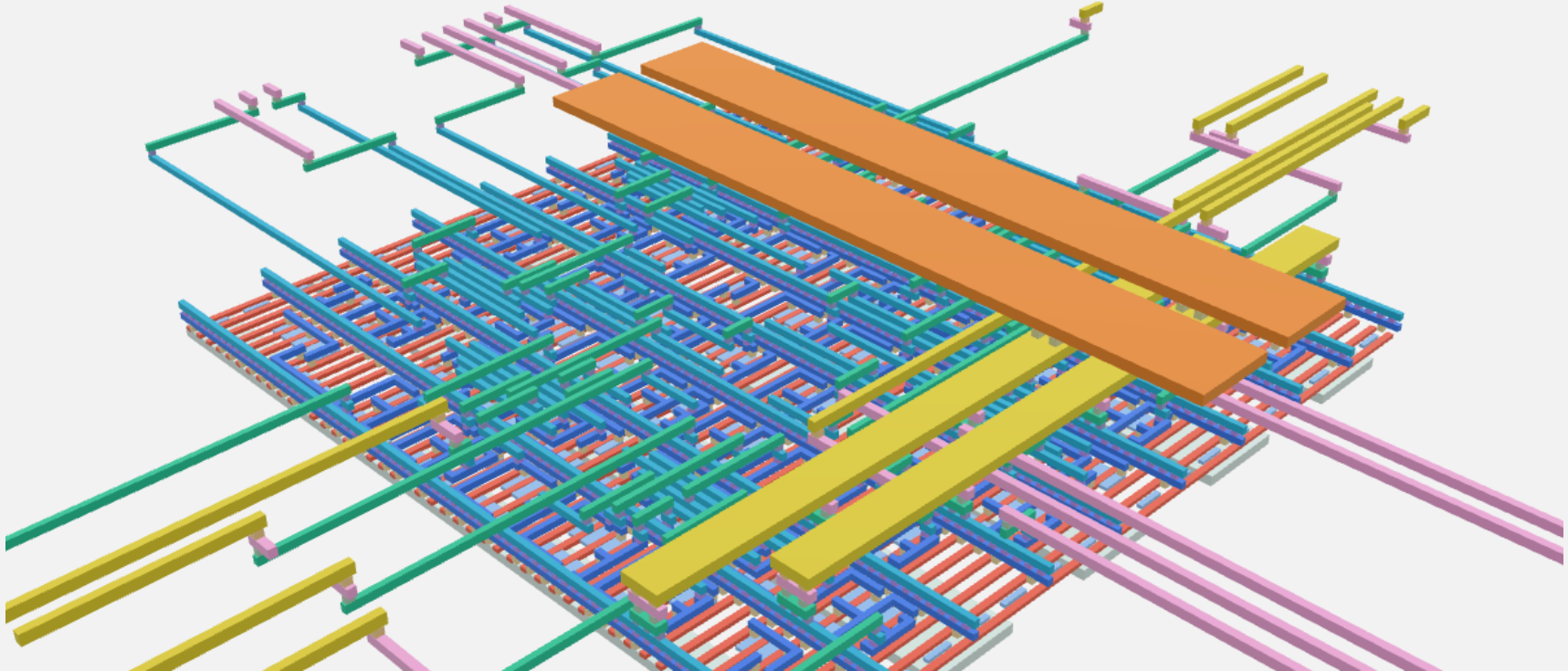
5.1
CSE140 – Digital Design

2026 Summer Session 1

Abarajithan G



Daily reminder:

- Open FeedbackFruits
  Canvas → CSE140 → Assignments → Slides 5.1

- Attendance is mandatory

- Special office hour (QR code): lettucemeet.com/l/E7P94

Final Exam in 3 days

- ~60 MCQs, cumulative
- One letter-size, double-sided cheat sheet.
- Closed-book. No phones/calculators
- Exam is designed for 1.5 hours. We'll give you 3 hours.

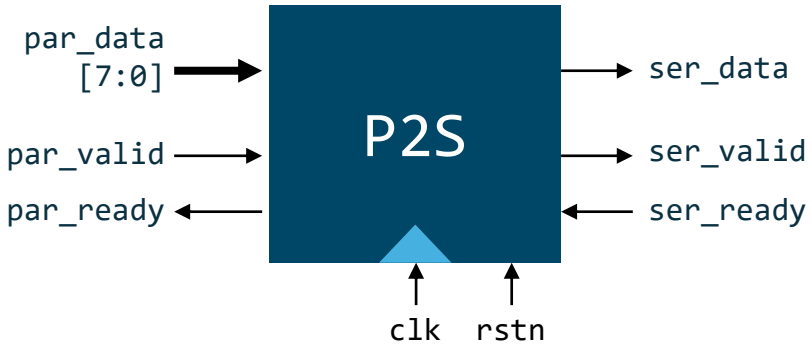
- Content you will NOT be tested on:
  - FPGA
  - SV code of UART RX, TX designs
  - SV code of retimed/transposed FIR Filter
  - Machine Learning
  - Writing assembly for CPU
- You will be tested on:
  - Code from all your assignments
  - Theory questions from all your assignments
  - Wrong MCQ options from the midterm & assignment questions
  - Lecture & discussion slides except the above
  - Tracing design + testbench through clock cycles
  - Reading a waveform and debugging a design

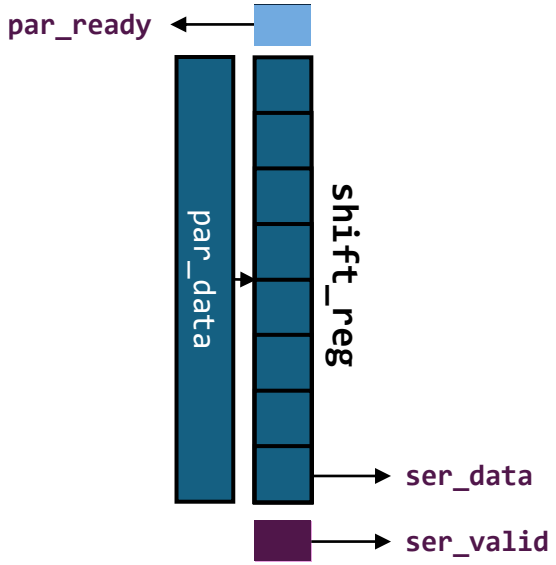
Previous Feedback

- Going through the systemverilog AND testbench wave data was great enough teaching where I understood the AXI stream.
- I really liked how in-depth we went over all of the topics

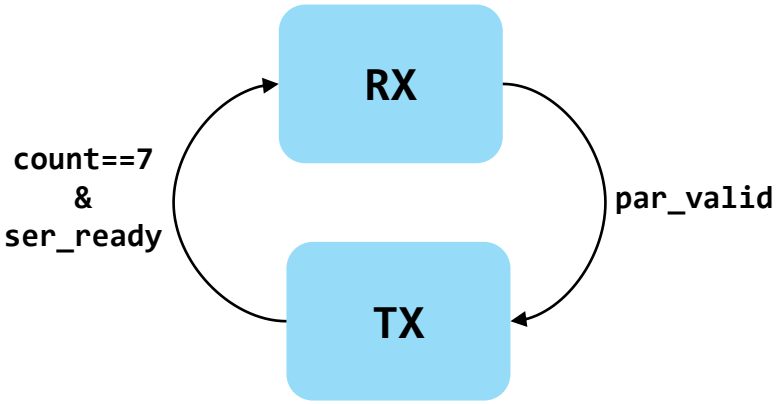
- It is a bit hard to see the presentation on the board and the writing on the blackboard sometimes.
- I think the slides use a lot of abbreviations

Parallel to Serial – with Flow Control





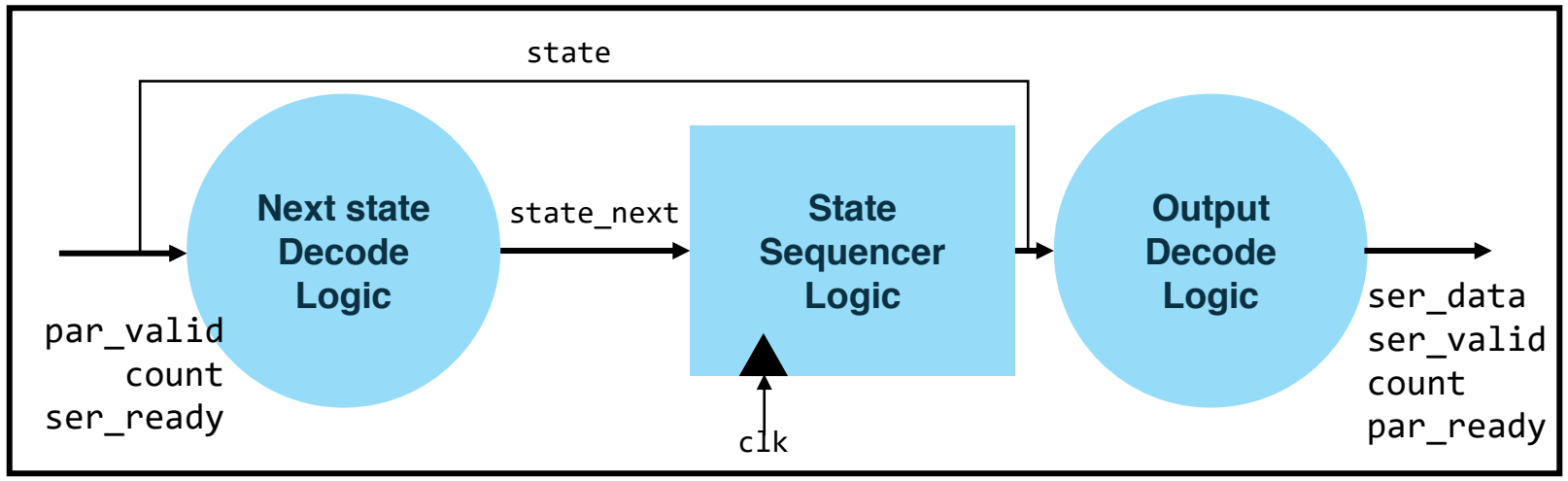

```
shift_reg <= par_data
par_ready    = 1
ser_valid    = 0
```



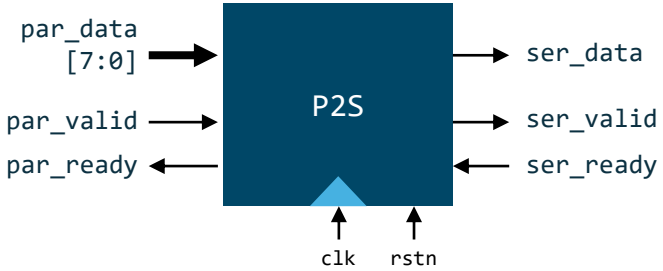

```
ser_valid   = 1
par_ready   = 0

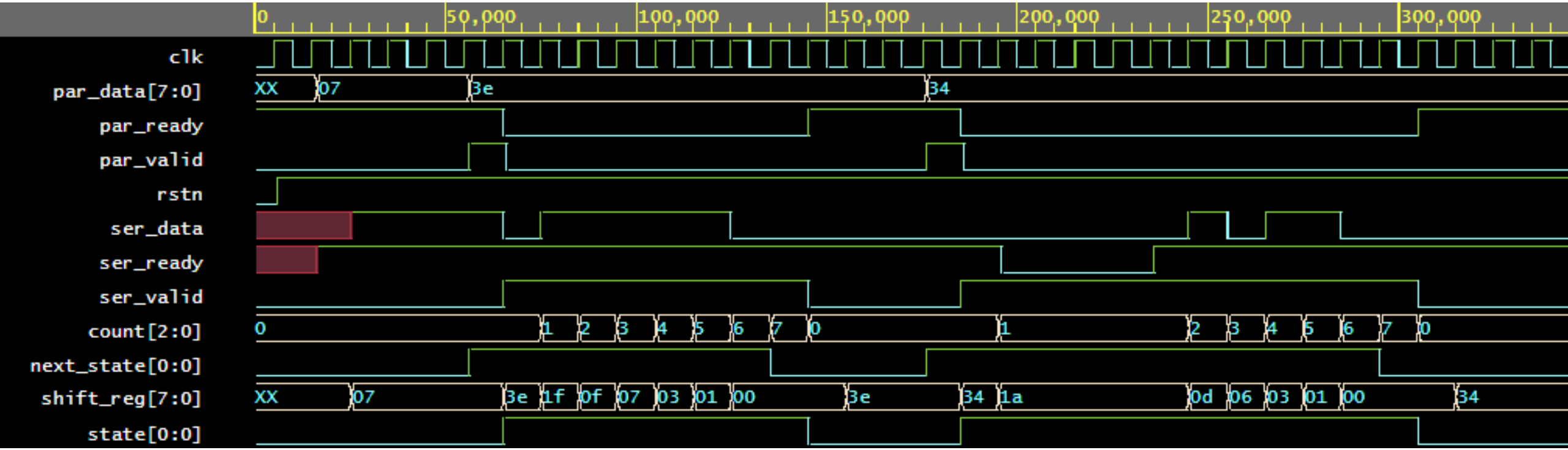
if (ser_ready)
  SHIFT
  count  <= count + 1
else
  DON'T SHIFT
  count  <= count
```

STATE MACHINE: THREE-PROCEDURE CODING STYLE



Parallel to Serial Waveform

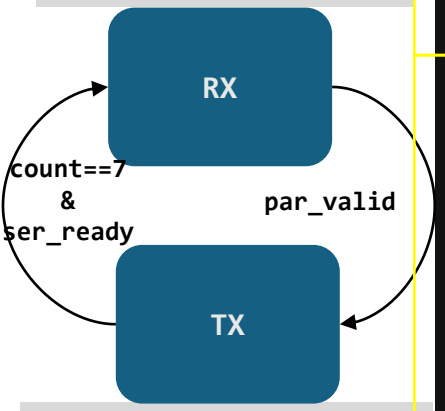




Parallel to Serial

Two coding styles

```
shift_reg <= par_data
par_ready    = 1
ser_valid    = 0
```




```
ser_valid   = 1
par_ready   = 0

if (ser_ready)
  SHIFT
  count  <= count + 1
else
  DON'T SHIFT
  count  <= count
```

More common style. Next value of registers is implicit. Less verbose, but harder to understand, and no access to next signals

```
module parallel_to_serial #(WIDTH = 8)(
  input  logic clk, rstn,
  input  logic ser_ready, par_valid,
  input  logic [WIDTH-1:0] par_data,
  output logic par_ready, ser_data, ser_valid
);
  localparam N_BITS = $clog2(WIDTH);
  enum logic {RX=0, TX=1} next_state, state;
  logic [N_BITS-1:0] count;
  logic [WIDTH-1:0] shift_reg;

  always_comb unique case (state)                    NEXT STATE DECODER
      RX: next_state = par_valid                                ?TX:RX;
      TX: next_state = ser_ready&&count==WIDTH-1                ?RX:TX;
    endcase

  always_ff @(posedge clk or negedge rstn)                      STATE
    if (!rstn) state <= RX;                                     SEQUENCER
    else       state <= next_state;

  always_comb begin                                             OUTPUT DECODER
    ser_data  = shift_reg[0];
    par_ready = (state == RX);
    ser_valid = (state == TX);
  end

  always_ff @(posedge clk or negedge rstn)
    if (!rstn) begin
      count     <= '0;
      shift_reg <= '0;
    end else if (state == RX) begin
      shift_reg <= par_data;
      count     <= '0;
    end else if (state == TX && ser_ready) begin
      shift_reg <= shift_reg >> 1;
      count     <= count + 1'd1;
    end
endmodule
```

Less common, more verbose, Easier to understand Access to next signals

```
module parallel_to_serial #(parameter WIDTH = 8)(
  input  logic clk, rstn, ser_ready, par_valid,
  input  logic [WIDTH-1:0] par_data,
  output logic par_ready, ser_data, ser_valid
);
  enum logic {RX, TX} state, next_state;
  logic [$clog2(WIDTH)-1:0] count, count_next;
  logic [WIDTH-1:0]  shift_reg, shift_reg_next;
```

Combinationally compute the next values of all registers

```
  always_comb begin
    par_ready = (state == RX);
    ser_valid = (state == TX);
    ser_data  = shift_reg[0];

    next_state     = state;
    count_next     = count;
    shift_reg_next = shift_reg;
    unique case (state)
      RX: begin
        count_next = '0;
        if (par_valid) begin
          shift_reg_next = par_data;
          next_state     = TX;
      end end
      TX: begin
        if (ser_ready) begin
          shift_reg_next = shift_reg >> 1;
          if (count == WIDTH - 1) begin
            count_next = '0;
            next_state = RX;
          end else begin
            count_next = count + 1'b1;
      end end end
    endcase end
```

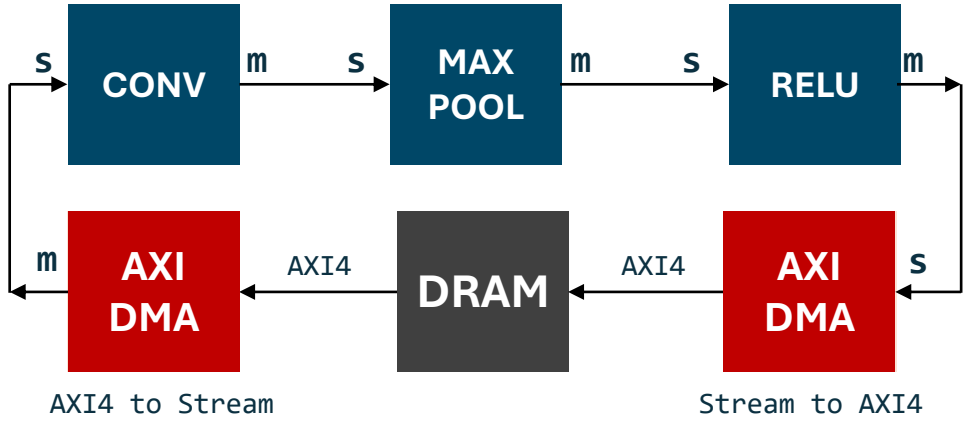
Just assign them to regs every clock

```
  always_ff @(posedge clk or negedge rstn)
    if (!rstn) begin
      state <= RX; {count, shift_reg} <= '0;
    end else begin
      state     <= next_state; count     <= count_next;
      shift_reg <= shift_reg_next;
    end
endmodule
```

AXI Stream Protocol

Bus

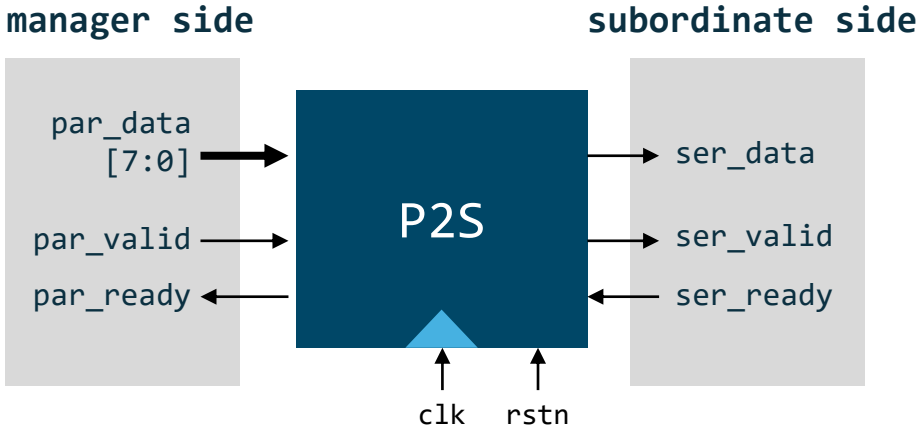
- Standardized set of wires and a “protocol”
- Used to keep modules independent & reusable when designing anything larger than this
- Eg: Bluetooth, Wi-Fi…etc. of mobile system-on-chips are made by different companies. But they follow the AXI (full) bus protocol.



AXI Stream

Wires : data, valid, [ready, last, user, id…]

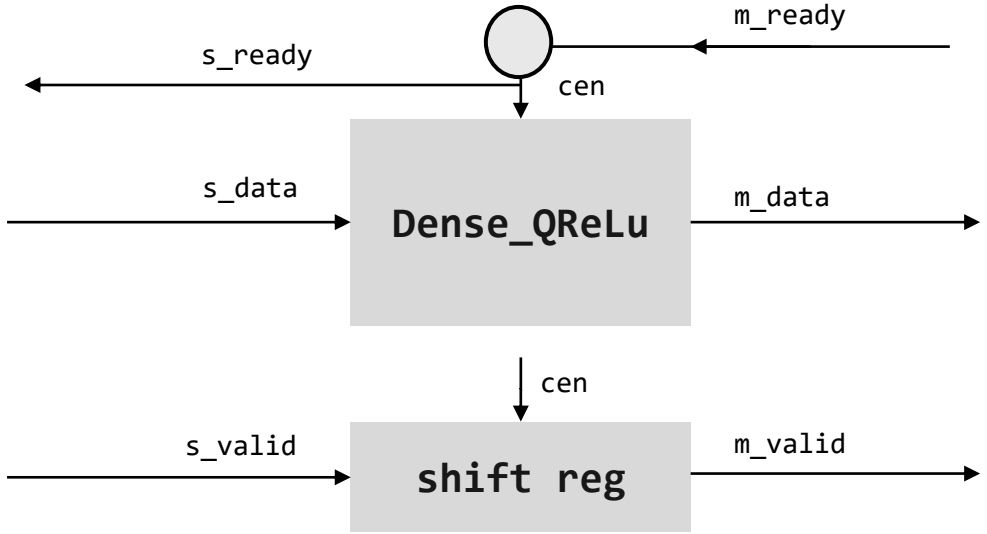
Protocol : If subordinate (downstream) gives ready and manager (upstream) gives valid in the same clock, data is transferred.

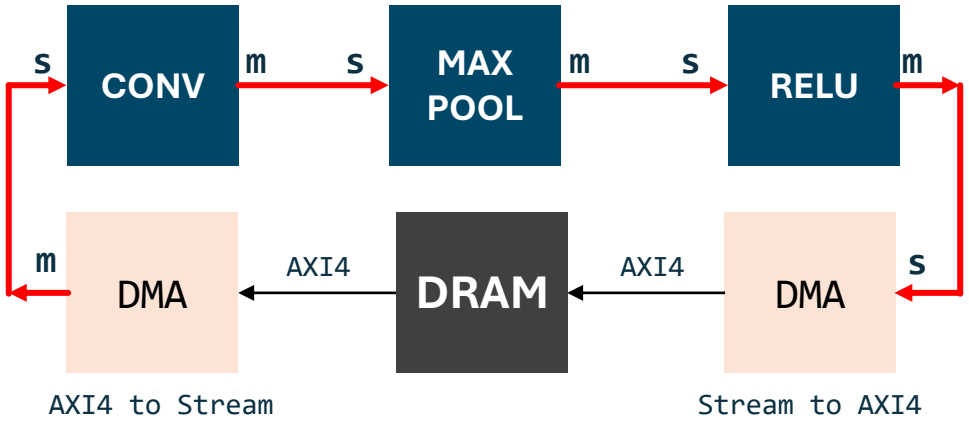


- Simplest & fastest bus I’ve seen. From ARM (AMBA/AXI)
- Useful for data pipelines where modules have independent data rates
- Backpressure: when a subordinate gets busy & pulls down ready
- Can be chained without long paths (unlike ours) using a skid buffer
- Each full AXI bus is made of 5 AXI Stream channels (ready/valid)

Valid/Ready Handshake

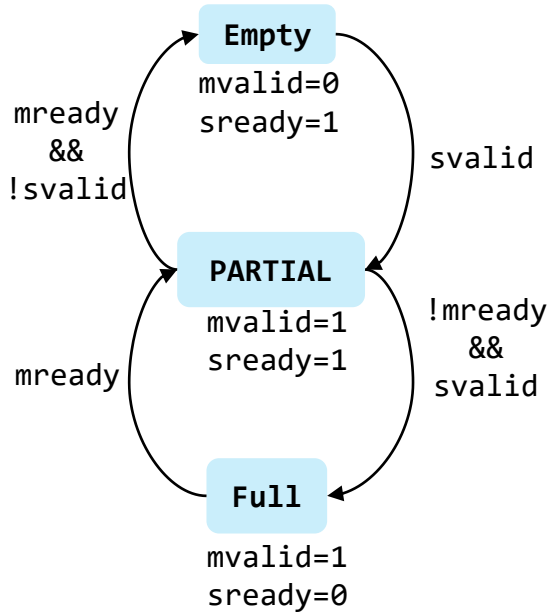
- Most common form of flow control
- Backpressure: ready = 0, upstream must hold valid & data
- Converting pipelined modules to AXI Stream (A4):
  - **`stall = m_valid && !m_ready`**
  - **`cen = !stall`**
  - **`s_ready = cen`**
- Problem: ready is combinational (long paths)

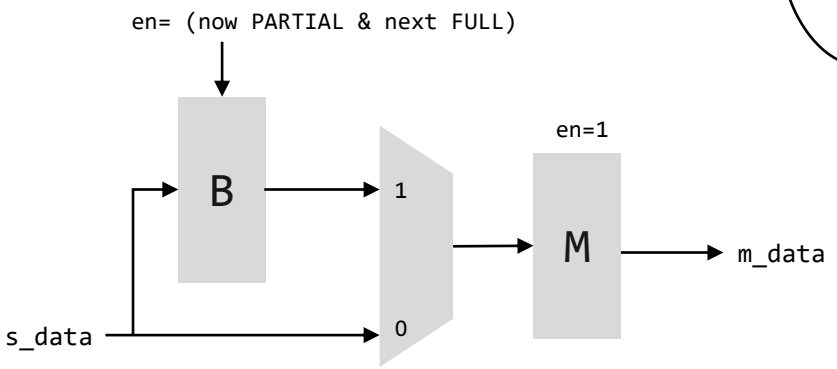




Skid Buffer

- **Problem:** If we register ready, the module cannot freeze immediately. It will freeze one clock later.
- **Solution:** "skid" the data into a buffer
- States:
  - Empty : Both B,M empty
  - Partial : Only M is valid
  - Full : Both B,M valid





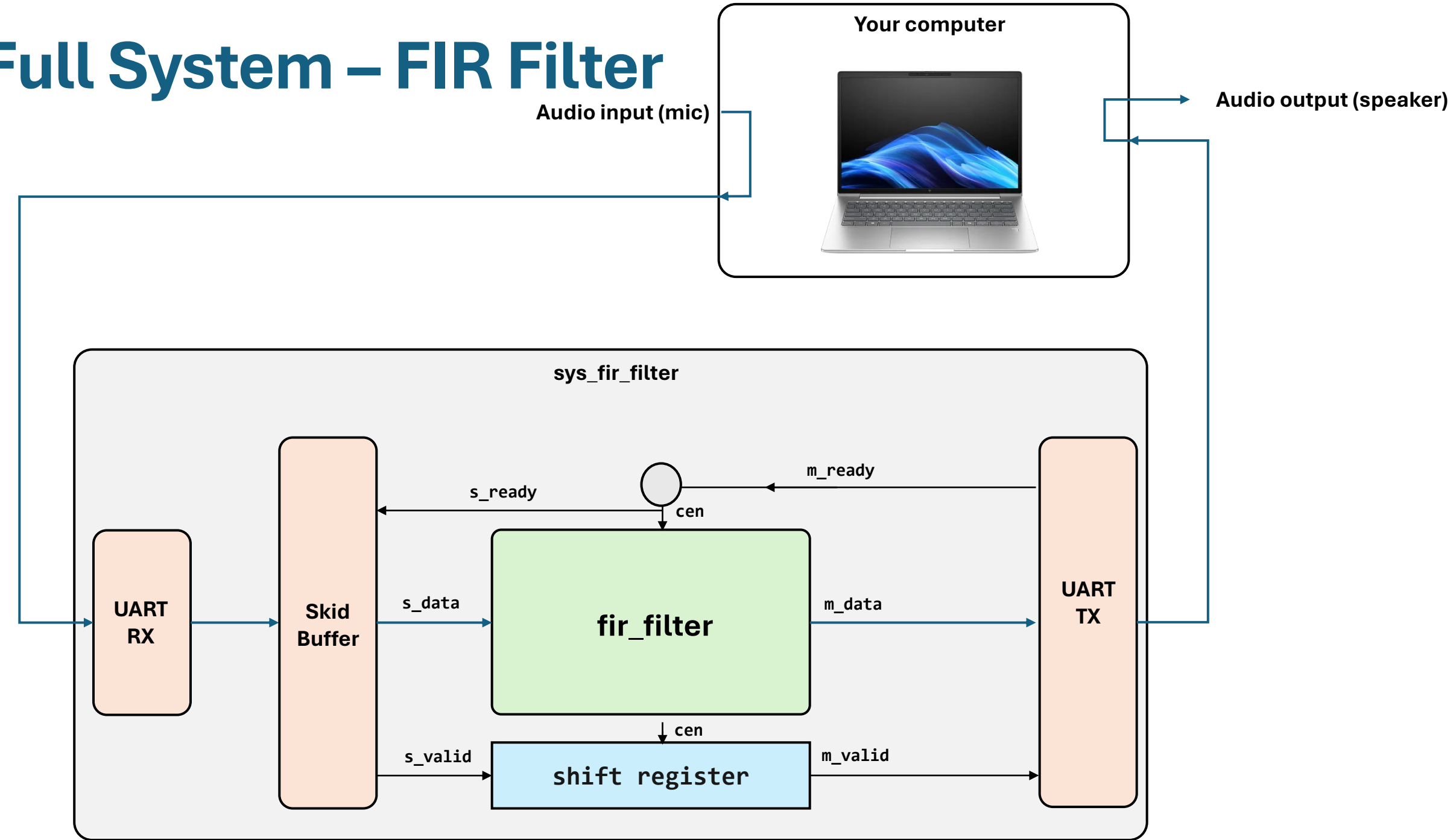
0 if (now EMPTY & next PARTIAL) or (now PARTIAL & mready & svalid)
1 if (now FULL & mready)

```
module skid_buffer #(parameter WIDTH = 8)(
  input  logic clk, rstn, s_valid, m_ready,
  input  logic [WIDTH-1:0] s_data,
  output logic [WIDTH-1:0] m_data,
  output logic m_valid, s_ready
);
  enum {EMPTY, PARTIAL, FULL} state, state_next;     State Machine: 3 procedure
  always_comb begin                                                  Why?
    state_next = state;                          Having state_next as comb gives
    unique case (state)                                    a lot of flexibility
      EMPTY  : if      ( s_valid)             state_next = PARTIAL;
      PARTIAL: if      (!m_ready &&  s_valid) state_next = FULL;
               else if ( m_ready && !s_valid) state_next = EMPTY;
      FULL   : if      ( m_ready)             state_next = PARTIAL;
    endcase
  end
  always_ff @(posedge clk)
    if (!rstn) state <= EMPTY;
    else       state <= state_next;

  logic [WIDTH-1:0] buffer;
  always_ff @(posedge clk)
    if (!rstn) begin
      {m_valid, buffer, m_data} <= 0;
      s_ready <= 1;
    end else begin
      m_valid <= state_next != EMPTY;
      s_ready <= state_next != FULL;     s_ready registered, breaking the path

      if (state == PARTIAL && state_next == FULL)
        buffer <= s_data;                        data, valid also registered
      unique case (state)
        EMPTY  : if (state_next == PARTIAL) m_data <= s_data;
        PARTIAL: if (m_ready && s_valid)    m_data <= s_data;
        FULL   : if (m_ready)               m_data <= buffer;
      endcase
    end
endmodule
```

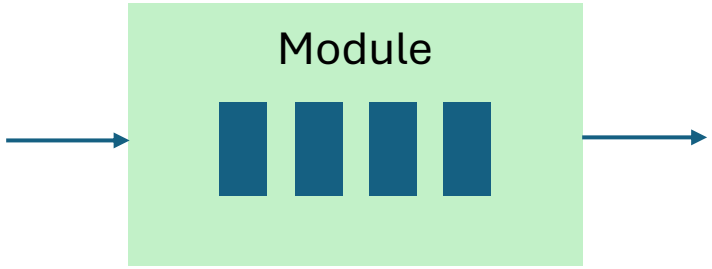
Full System – FIR Filter



Latency and Throughput

- Packet = Data sent over multiple clock cycles (e.g. output of parallel to serial)
- Latency = Time taken from first input beat to first output beat in a packet
- Throughput = How many inputs/outputs per second can you process?
- Interval (1/throughput) = Every (interval) clocks, we can take/release an input/output
- Both measured at steady-state (after pipeline is full)

- Activity: Find latency & interval in clock cycles
  - Parallel to serial (N=8)
  - FIR Filter (basic, not retimed, N=100)
  - Binary reduction tree to find min(X) (N=10)
- **If you are done, please fill the CSE140 evaluation (mail)**



Eg: purely pipelined module, with 4 pipeline registers

- Interval = 1 cycle
- Latency = 4 cycles
- Throughput = 1 per cycle

SystemVerilog Gotchas

- Signed arithmetic – all operands have to be signed or **$signed()**. 0 is signed, but 8'd0 is unsigned
- Testbench: never read & write at clock edge. Always: `@(posedge clk) #1ps x <= y; $display("%d", z);`
- Typos in instantiations will silently "infer 1-bit nets" : `logic [7:0] x,y,z;  mul M (.x(x), .y(u), .z(z));`
- If unspecified, default base of literals is 10: if (x==10)
- Default sign of unbased vs based literals: `logic signed [7:0] x=-5; x+1=>-4; x+1'b1 => 252`
- Filling all ones: x = 1 vs x = '1
- Port connection width mismatch
- Accidentally driving an input port

SystemVerilog Gotchas

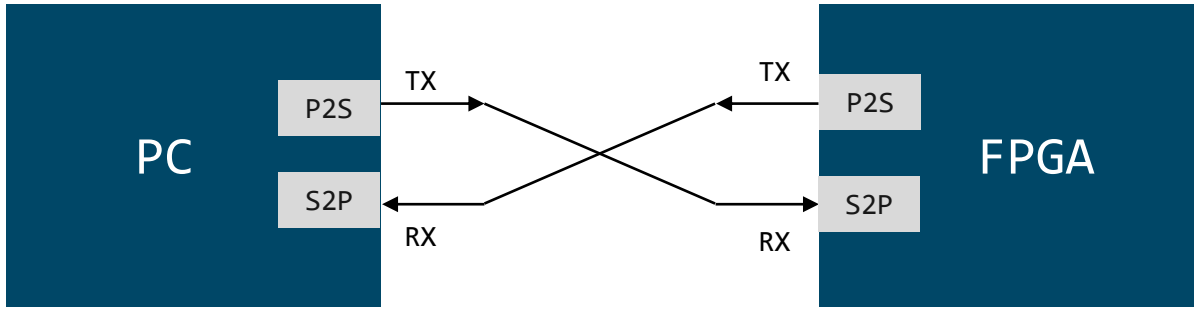
- Ensure both left & right hand side have same width in assignments (see warnings)
- Use always_comb, always_ff, always_latch. Never use always@
- Only use = for combinational logic, <= for sequential logic. Never mix in RTL design
- Be consistent with active high/low reset throughout the design

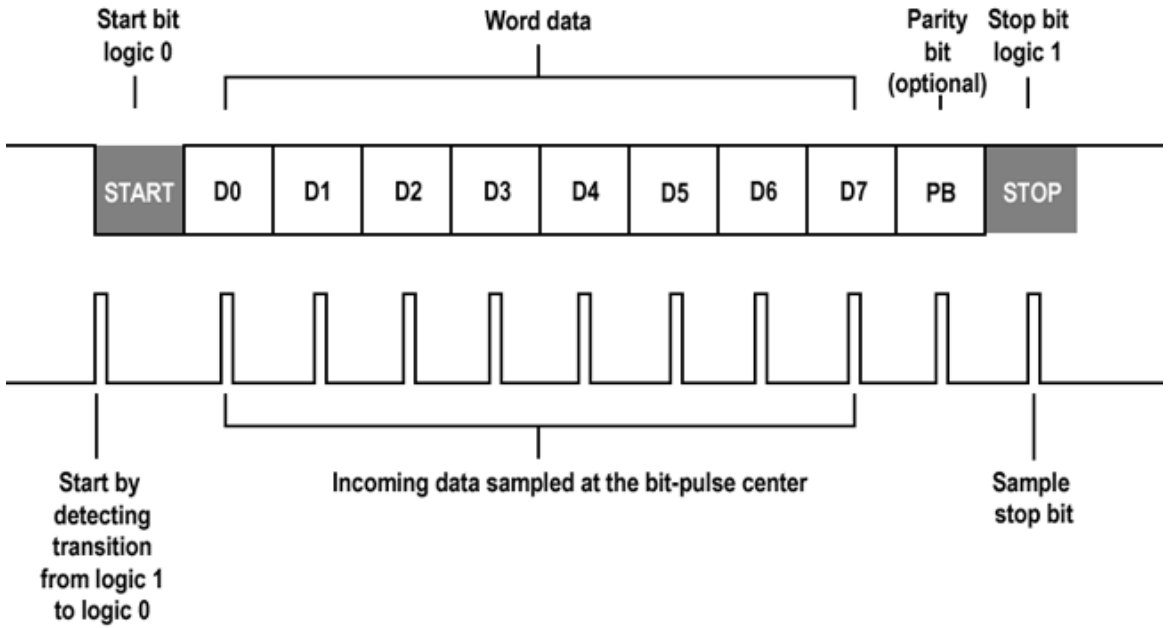
Guidelines for Clean Code

- Do not use /**/ for comments in code body. Instead, use //
- Used named or .* connections. Do not use port order connections
- Use logic type everywhere in RTL design. Do not use reg
- Use enums for states
- Use logical operators (&&, ||, !) even for 1 bit signals, rather than bitwise (&,|,~)
- Avoid mixing signed & unsigned variables in expressions. Make sure either all are unsigned, or all are signed.
- Declare functions used in RTL as automatic

UART Protocol

- Universal Asynchronous Receiver/Transmitter
- Serial – 1 bit at a time
- Asynchronous
  - Baud rate. eg: 2 million bits per second
  - FPGA (54 MHz): 27 clocks per bit
- Slow, but simple
- Used everywhere: microcontrollers, FPGA, your PC
- Serial communication can be fast: USB!





Activity: Find ALL Bugs

Pipelined module to compute: **relu((ax + b) >> s)**, where relu(x) = x > 0 ? x : 0; a,x,b are signed

```
module mac_shift_relu #(parameter W=8)(
  input logic clk, rstn, input logic [W-1:0] a,x,b,s,
  output logic [W-1:0] y);
  logic [W-1:0] ax, sum, shifted;


  always_ff @(posedge clk) begin
    if (!rstn) {ax, shifted} = '0;
    else begin
      ax <= a * x;
      end
  end

  always_comb begin
    sum <= $signed(ax) + b;
    shifted = sum >> s;
    if (shifted > 0) y = shifted;
  end
endmodule
```

```
module tb_mac_shift_relu;
  logic clk=0, rstn=0;
  logic [7:0] a,x,b,y,s,t;


  mac_shift_relu dut( .clk(clk),.rstn(rstn),.a(a),
                      .x(x), .b(b),.s(s),.y(v));


  initial forever #5 clk = ~clk;
  initial begin
    rstn = 1;
    @(posedge clk) a = -3; x = 4; b = 20; s = 1; t=(a*x+b)>>s);
    @(posedge clk) if (y != (t>0?t:0) $fatal("Wrong y=%0d",y);
  end
endmodule
```

Activity: Find ALL Bugs

Pipelined module to compute: **relu((ax + b) >> s)**, where relu(x) = x > 0 ? x : 0;

```
module mac_shift_relu #(parameter W=8)(
  input logic clk, rstn, input logic [W-1:0] a,x,b,s,
  output logic [W-1:0] y);
  logic [W-1:0] ax, sum, shifted;


  always_ff @(posedge clk) begin
    if (!rstn) {ax, shifted} = '0;
    else begin
      ax <= a * x;
    end
  end

  always_comb begin
    sum <= $signed(ax) + b;
    shifted = sum >> s;
    if (shifted > 0) y = shifted;
  end
endmodule
```

= in always_ff

$signed() missing – we want signed

Ax has same width - truncation

<= in always_comb

$signed() missing for one operand

Logical shift in place of arithmetic shift

Else branch/default not handled in always_comb: latch inference

Unsigned comparison

```
module tb_mac_shift_relu;
  logic clk=0, rstn=0;
  logic [7:0] a,x,b,y,s,t;


  mac_shift_relu dut( .clk(clk),.rstn(rstn),.a(a),
                      .x(x), .b(b),.s(s),.y(v));


  initial forever #5 clk = ~clk;
  initial begin
    rstn = 1;
    @(posedge clk) a = 3; x = 4; b = 20; s = 1; t=(a*x+b)>>s);
    @(posedge clk) if (y != (t>0?t:0)) $fatal("Wrong y=%0d",y);
  end
endmodule
```

Spelling mistake in port connection. A new, undriven wire “v” created silently

Reset removed before clock edge. Registers have garbage

Read/write at posedge from initial

Same signal (shifted) driven from two always blocks

ax is registered, b,s is not. We are adding wrong ax+b, and shifting wrong s

UART TX

```
module uart_tx #(
  parameter CLKS_PER_BIT  = 4,
            N_WORDS       = 2,
            BITS_PER_WORD = 8,
            W_PACKET      = BITS_PER_WORD+5
  )(
    input  logic clk, rstn, s_valid,
    input  logic [N_WORDS-1:0][BITS_PER_WORD-1:0] s_data,
    output logic tx, s_ready
  );
  localparam END_BITS = W_PACKET-BITS_PER_WORD-1;
  localparam CW_BITS = $clog2(N_WORDS*W_PACKET);
  localparam CW_CLKS = $clog2(CLKS_PER_BIT);

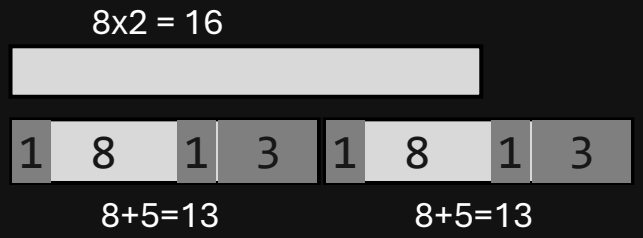
  logic [N_WORDS-1:0][W_PACKET-1:0] s_packets;
  logic [N_WORDS*W_PACKET-1:0]      m_packets;

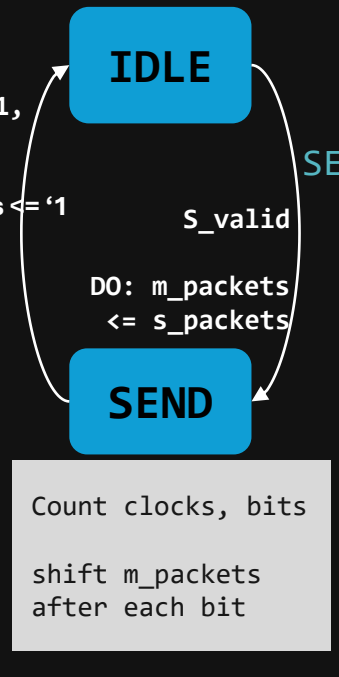
  always_comb begin
    tx      = m_packets[0];
    s_ready = (state == IDLE);
    for (int n=0; n<N_WORDS; n=n+1)
      s_packets[n] = {{END_BITS{1'b1}}, s_data[n], 1'b0};
  end
```

TX is connected to LSB of packet

Receive data via AXI Stream

Create set of packets. Each with start bit (0), data word and END_BITS number of ones (zeros inverted)






```
  logic [CW_BITS-1:0] c_bits;
  logic [CW_CLKS-1:0] c_clocks;

  enum {IDLE, SEND} state;

  always_ff @(posedge clk or negedge rstn) begin
    if (!rstn) begin
      state     <= IDLE;
      m_packets <= '1;
      {c_bits, c_clocks} <= '0;
    end else
      case (state)
        IDLE: if (s_valid) begin
                state     <= SEND;
                m_packets <= s_packets;
              end
        SEND: if (c_clocks == CW_CLKS'(CLKS_PER_BIT-1)) begin
                c_clocks <= '0;

                if (c_bits == CW_BITS'(N_WORDS*W_PACKET-1)) begin
                  c_bits    <= '0;
                  m_packets <= '1;
                  state     <= IDLE;
                end else begin
                  c_bits    <= c_bits + 1'b1;
                  m_packets <= m_packets >> 1;
                end
              end else c_clocks <= c_clocks + 1'b1;
      endcase
  end
endmodule
```

Two nested counters

Receive data via AXI Stream

Keep each bit stable for 4 clocks
Send all 26 bits

Shift the packet towards LSB

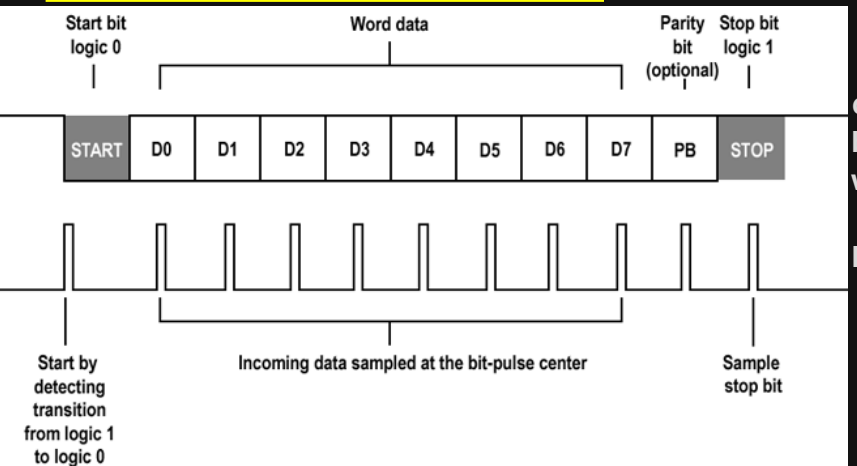
UART RX

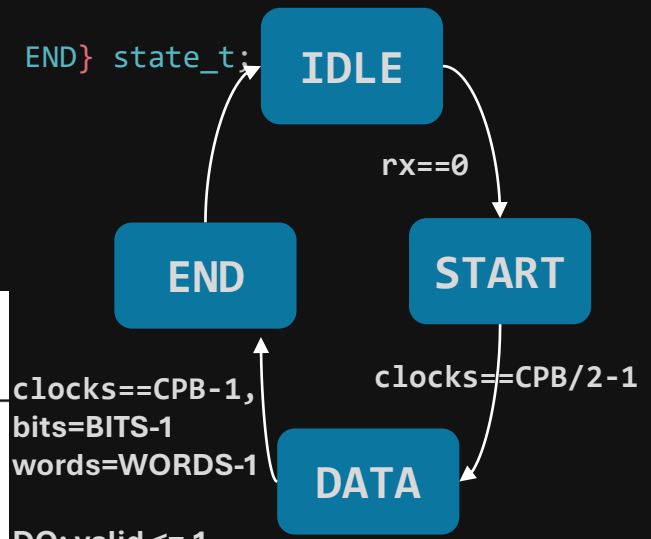
```
module uart_rx #(
  parameter CLKS_PER_BIT  = 4,
            N_WORDS       = 3,
            BITS_PER_WORD = 8
)(
  input  logic clk, rstn, rx,
  output logic m_valid,
  output logic [N_WORDS*BITS_PER_WORD-1:0] m_data
);
  localparam CW_CLK    = $clog2(CLKS_PER_BIT);
  localparam CW_BIT    = $clog2(BITS_PER_WORD);
  localparam CW_WORD   = $clog2(N_WORDS) == 0 ? 1 : $clog2(N_WORDS);

  typedef enum logic [1:0] {IDLE, START, DATA, END} state_t;
  state_t state;
```

Three counters

```
logic [CW_CLK -1:0] c_clocks;
logic [CW_BIT -1:0] c_bits;
logic [CW_WORD-1:0] c_words;
```





Count clocks, bits, words

shift {rx,m_data} after each bit

State Machine: One procedure coding style

```
always_ff @(posedge clk or negedge rstn) begin
  if (!rstn) begin
    state    <= IDLE;
    {c_clocks, c_bits, c_words, m_valid, m_data} <= '0;
  end else begin
    m_valid <= 1'b0;
    case (state)
      IDLE:   begin
                c_clocks <= '0;
                c_bits   <= '0;
                if (!rx) state <= START;
              end
      START:  begin
                if (c_clocks == CW_CLK'(CLKS_PER_BIT/2 - 1)) begin
                  state    <= DATA;
                  c_clocks <= '0;
                end else c_clocks <= c_clocks + 1'b1;
              end
      DATA:   if (c_clocks == CW_CLK'(CLKS_PER_BIT - 1)) begin
                c_clocks <= '0;
                m_data   <= {rx, m_data[N_WORDS*BITS_PER_WORD-1:1]};
                if (c_bits == CW_BIT'(BITS_PER_WORD - 1)) begin
                  state  <= END;
                  c_bits <= '0;
                  if (c_words == CW_WORD'(N_WORDS - 1)) begin
                    m_valid <= 1'b1;
                    c_words <= '0;
                  end else c_words <= c_words + 1'b1;
                end else c_bits <= c_bits + 1'b1;
              end else c_clocks <= c_clocks + 1'b1;
      END:    if (c_clocks == CW_CLK'(CLKS_PER_BIT - 1)) begin
                state    <= IDLE;
                c_clocks <= '0;
              end else c_clocks <= c_clocks + 1'b1;
    endcase end end
endmodule
```

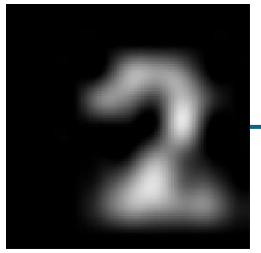
rx enters at the MSB of m_data and shifted towards LSB

Valid after all words

Count

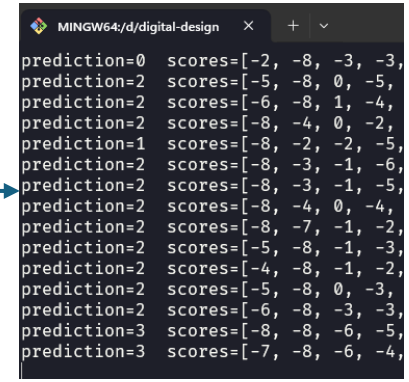
NN Accelerator

Webcam input



Your computer





Prediction Output

sys_nn (A4)

UART RX

Skid Buffer

axis_dense_relu [0]

dense_qrelu [0] (A3)

vector_mac

Multipliers

reduction_tree_add

qrelu (A2)

axis_dense_relu [1]

dense_qrelu [1] (A3)

vector_mac

Multipliers

reduction_tree_add

qrelu (A2)

UART TX

Daily Feedback

- One thing I liked about today’s class:
- One thing that could be improved:
- One place where I felt lost (if any):