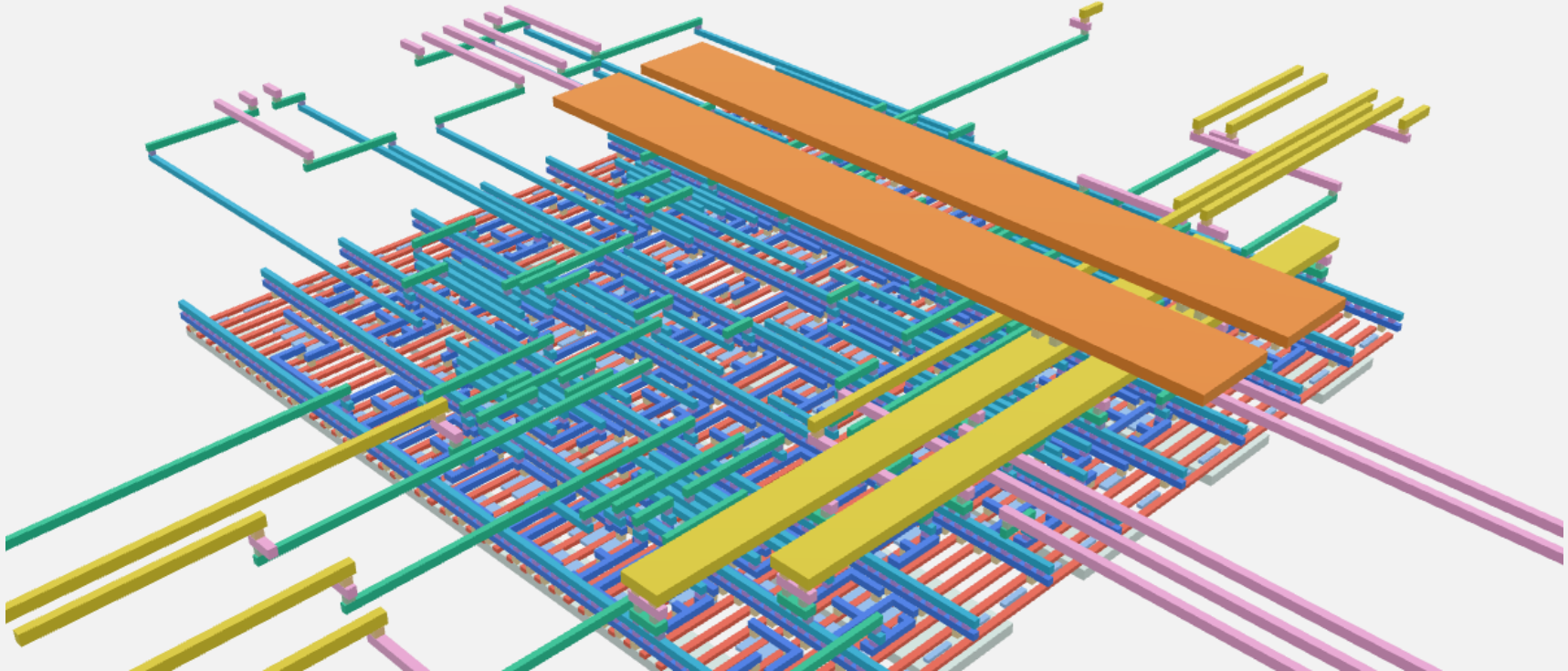
4.2
CSE140 – Digital Design

2026 Summer Session 1

Abarajithan G



Daily reminder:

- Open FeedbackFruits
  Canvas → CSE140 → Assignments → Slides 4.2

- Attendance is mandatory

- Start A3 & A4 early – they might take a lot of time

Previous Feedback

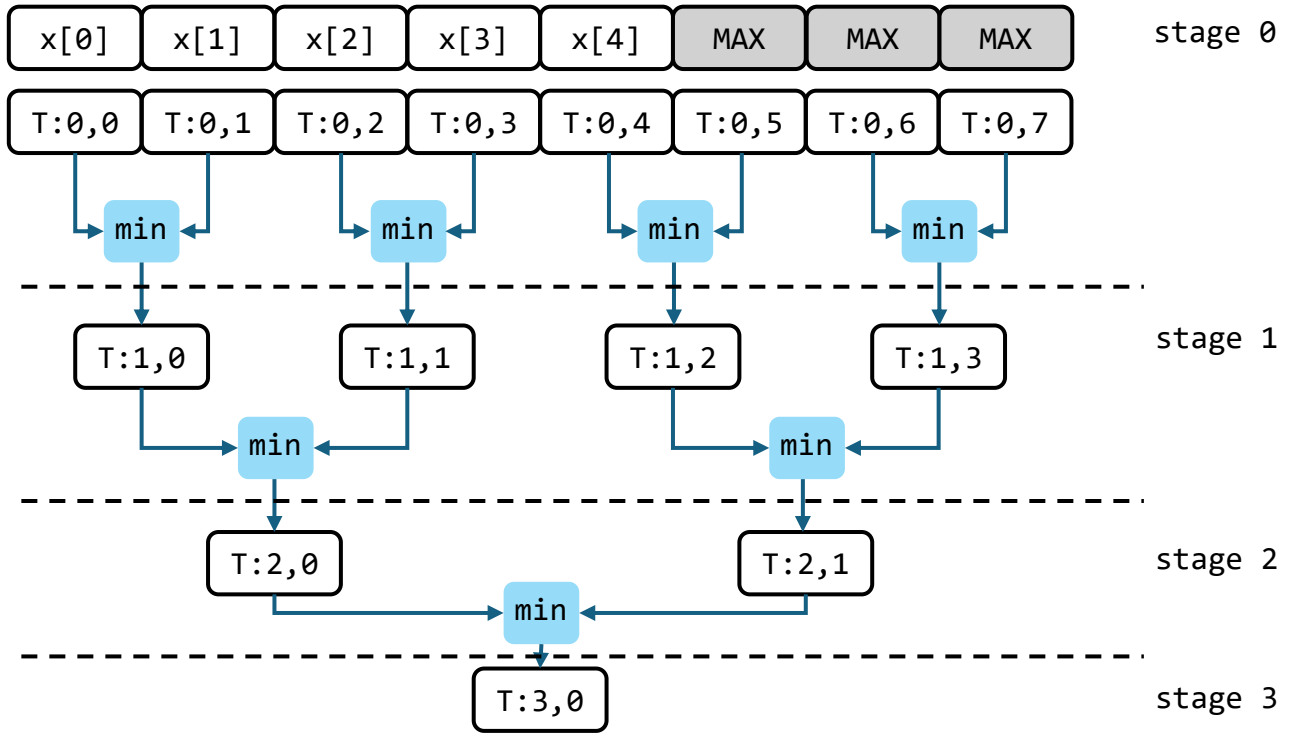
- FIR Filter explanation was super interesting
- I liked all the blackboard diagrams
- i took a signals class so seeing it connect to the fir was interesting

- I thought the waveform activity could've used a little bit more guidance;
- The directions/syntax in count activity was unclear and weird to apply to the questions

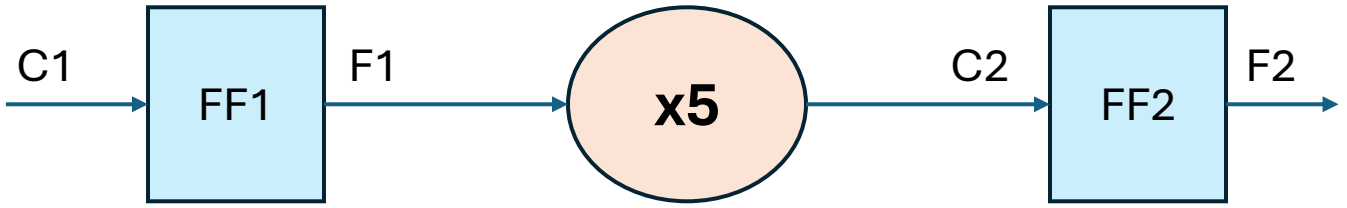
Optional Optimization – Bonus Points

- Must email me before Thursday 11:59pm
- Top 6 designs will be chosen for presentation
- Present in front of the class on Friday 31st
- Questions on code & design choices
- Must do without AI generation – risk "F"
- Bonus points: email + top 6 present + grading
- **Possible optimizations:**
  - Reuse multipliers & adders
  - Initialize acc with bias
  - FIR – share resources for repeated coef.
  - Anything else
- More info on Syllabus

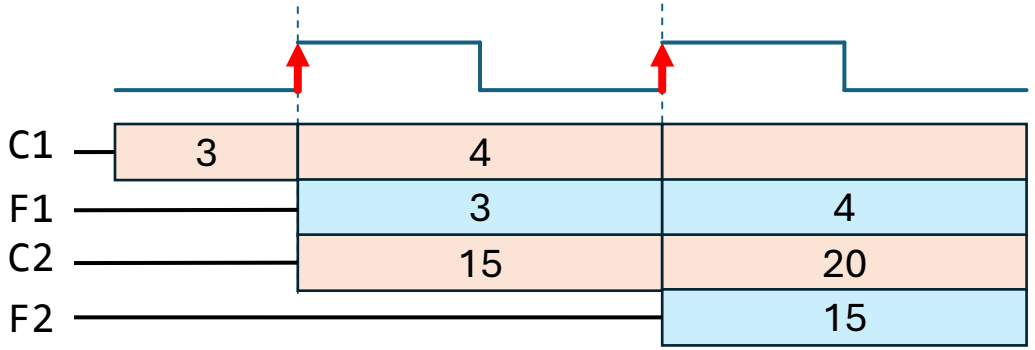
up to +5%	FIR Filter Resource Optimization
up to +10%	Neural Network Resource Optimization

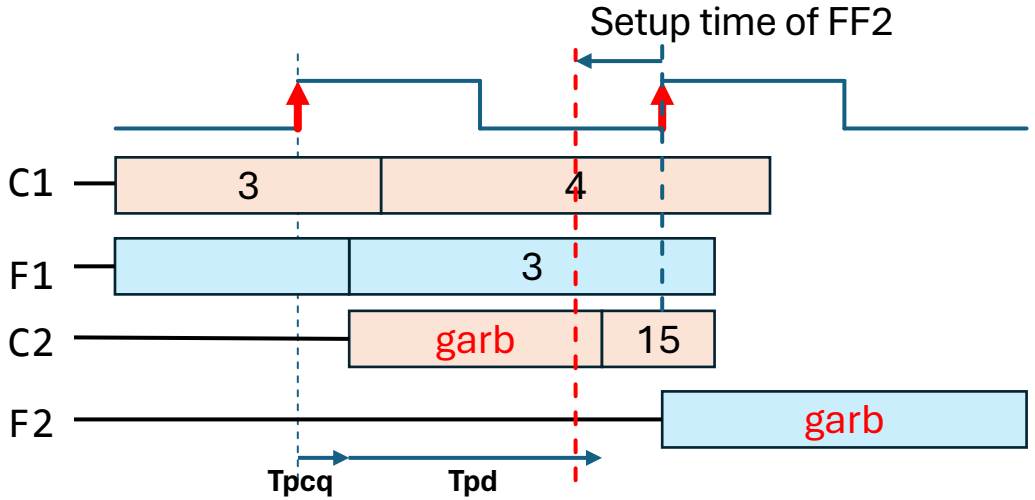


Setup Time

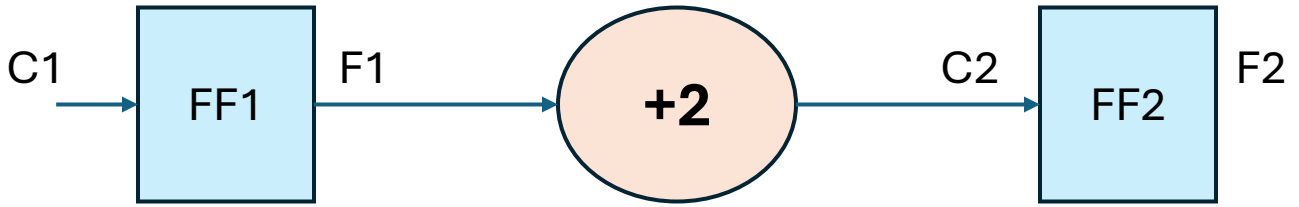


- Signal should go from FF to next FF within one clock cycle (T = period)
- But, it takes time to
  - Leave the first FF (Tpcq – max)
  - Run through comb logic (Tpd – max)
  - Be accepted at new FF (Ts – max)
- **Tpcq(FF1) + Tpd(comb) + Ts(FF2) <= T**
- Example:
  - Setup violation at FF2 around 2nd posedge.
  - Calculated result (15) comes after setup time
  - Garbage (partial result) gets stored in FF2

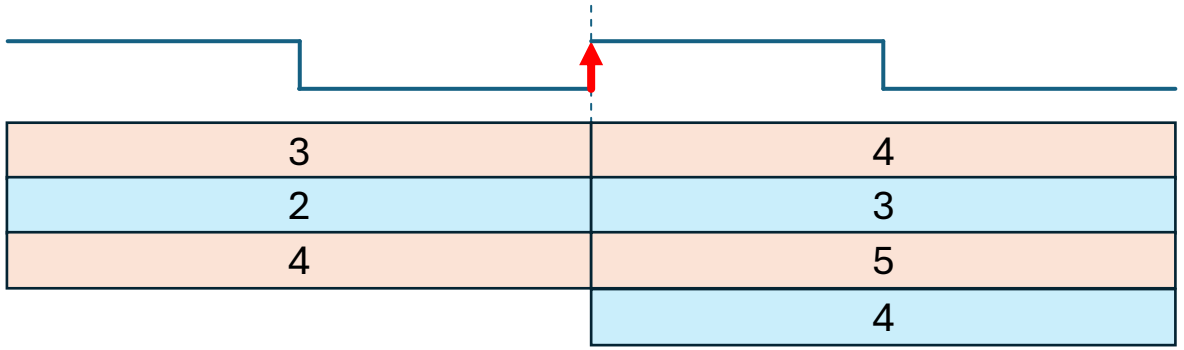


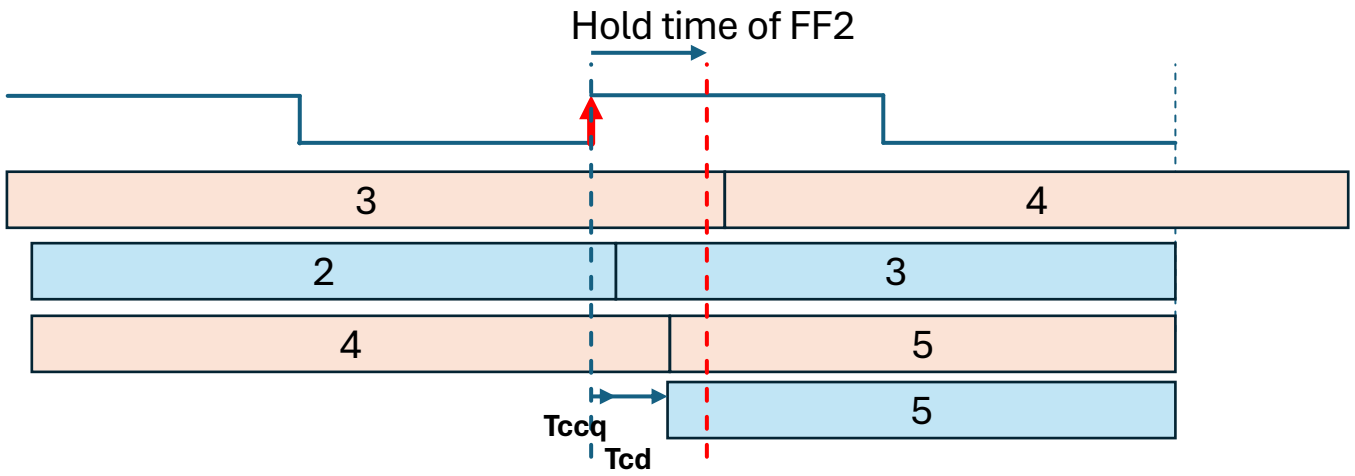


Hold Time



- A signal should stay constant (held) for a while after a posedge for an FF to store it correctly.
- Shortest time to
  - Leave FF1: Tccq
  - Run through comb: Tcd
  - Reach FF2: Tccq + Tcd
- **Tccq(FF1) + Tcd(comb) >= Th(FF2)**
- Example:
  - Hold violation at FF2 around 1st posedge
  - In the same posedge,
    - Signal leaves FF1
    - Runs through comb
    - Reaches FF2 – before its hold time
  - FF2 might store 5, instead of 4.





FIR Filter – Audio processing circuits

Digital Signals

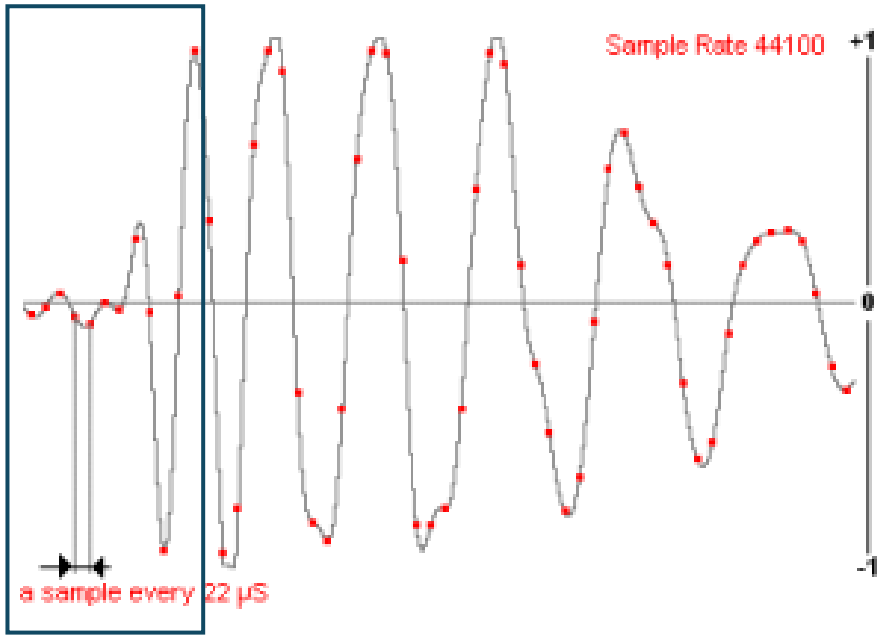
- Audio (songs you listen, my voice…) is a 1D signal
- Stored and transmitted as a series of numbers: [20, 5, 3, 8, 17,…]
- In uncompressed form, the numbers are proportional to sound amplitude & voltage of mic/speaker in known intervals.

Filtering

- We often need to extract, remove or boost different frequency ranges (bass, treble, midtones)
- Inside your mics, headphones & guitar amplifiers, this is done in custom chips, that apply various filters

FIR Filters

- Finite Impulse Response (FIR) filters are series of numbers multiplied & added with the sound signal for filtering.
- Numbers are mathematically computed – entire field of DSP




```
[-2,-1, 2,-2 ,-3, 0,-1, 7,-1,-23,1,23] – samples: x[n]
[ 0, 1, 2, 5 , 2, 1, 0]                – filter : k[n]
```

Dot product:

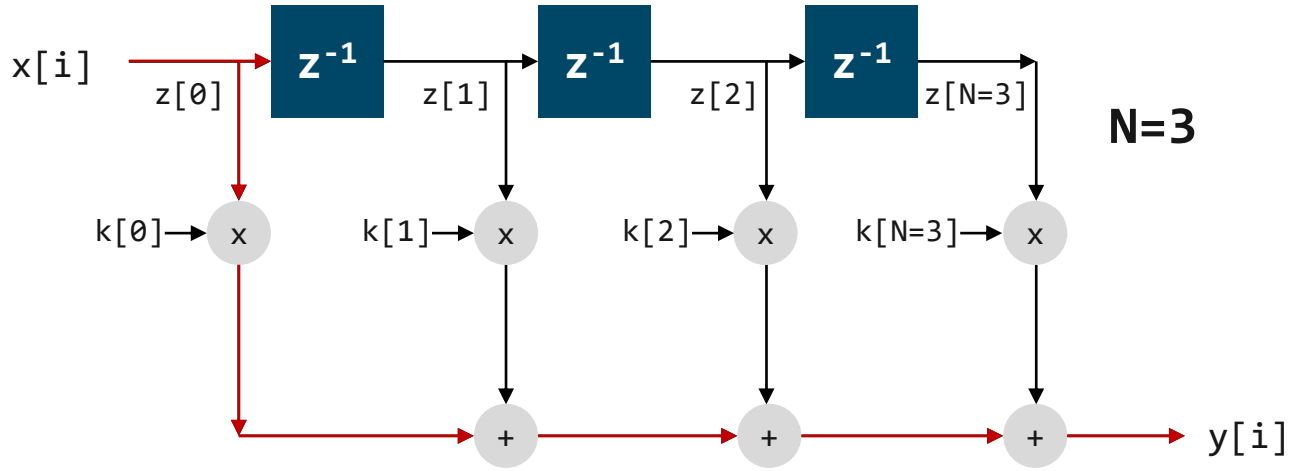
```
[ 0,-1, 4,-10,-6, 0, 0]  – multiplied: m[n]
[-13]                    – added     : y[0]
```

Shift x[n] & dot product:

```
[-2,-1, 2,-2, -3, 0,-1, 7,-1,-23,1,23]    – samples: x[n]
   [ 0, 1, 2,  5, 2, 1, 0]                – filter : k[n]
   [ 0, 2,-4,-15, 0,-1, 0]  – multiplied: m[n]
   [-18]                    – added     : y[1]
```

Repeat shift x[n] & dot product: Convolution

FIR Filter



$$y[n] = x[n]k[0] + x[n-1]k[1] + x[n-2]k[2] + ... + x[n-N]k[N]$$

$$y[n] = \mathbf{x[n]}k[0] + \mathbf{z^{-1}x[n]}k[1] + \mathbf{z^{-2}x[n]}k[2] + ... + \mathbf{z^{-N}x[n]}k[N]$$

Implementation:

- Every clock cycle, an input sample goes in, output sample comes out
- N+1 multipliers and N adders work in parallel for dot product
- Long path through adders. Fmax for N=100: 687.6 MHz

```
module fir_filter_naive #(
  parameter N = 5, W_X = 8, W_K = 4,
  parameter logic [(N+1)*W_K-1:0] K = {
    4'sd5, 4'sd4, 4'sd3, 4'sd2, 4'sd1, 4'sd0
    },//'{default:0},
  localparam W_Y = W_X + W_K + $clog2(N+1))(
    input  clk, rstn, en,
    input  logic [W_X-1:0] x,
    output logic [W_Y-1:0] y
  );
  logic [N:0][W_K-1:0] k_arr;
  always_comb k_arr = K;

  genvar n;
  localparam W_M = W_X + W_K;
  logic [N  :0][W_M-1:0] m;
  logic [N  :0][W_Y-1:0] a;
  logic [N  :0][W_X-1:0] z;

  always_ff @(posedge clk or negedge rstn)
    if (!rstn)   z[N:1] <= '0;
    else if (en) z[N:1] <= z[N-1:0];

  always_comb begin
    z[0] = x;
    y = 0;
    for (int n=0; n < N+1; n=n+1)
      y = $signed(y)+$signed(k_arr[n])*$signed(z[n]);
  end
endmodule
```

Flat array: K*W

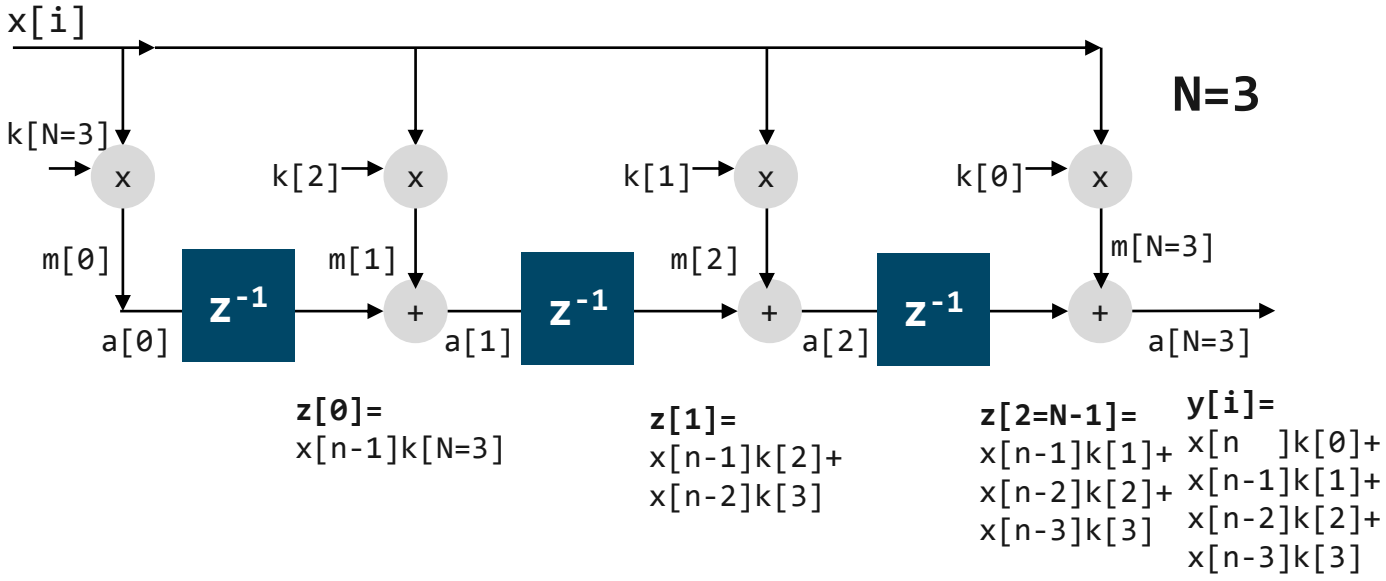
Constant coefficients, baked into the circuit

Flat array [K*W] can be assigned to packed array [K][W]

Delay registers to hold older x values

MAC Tree - combinational

FIR Filter - Retimed



$$y[n] = x[n]k[0] + x[n-1]k[1] + x[n-2]k[2] +...+ x[n-N]k[N]$$

$$y[n] = x[n]k[0] + z^{-1}(x[n]k[1] + z^{-1}(x[n]k[2] +...+ z^{-1}(x[n]k[N])))$$

Implementation:

- Delay is moved to break long combinational path
- Filter (k) order is reversed, Math is same
- Fmax for N=100: 977.9 MHz (42% better), but 62% more cells
- Still not perfect – high input fanout

```
module fir_filter #(                          Same interface & behaviour
  parameter N = 5, W_X = 8, W_K = 4,
  parameter logic [(N+1)*W_K-1:0] K = {
    4'sd5, 4'sd4, 4'sd3, 4'sd2, 4'sd1, 4'sd0
    },//'{default:0},
  localparam W_Y = W_X + W_K + $clog2(N+1))(
    input  clk, rstn, en,
    input  logic [W_X-1:0] x,
    output logic [W_Y-1:0] y);
  genvar n; localparam W_M = W_X + W_K;
  logic [N  :0][W_M-1:0] m;
  logic [N  :0][W_Y-1:0] a;
  logic [N-1:0][W_Y-1:0] z;
  logic [N  :0][W_K-1:0] k_arr;

  always_comb begin
    k_arr = K;                                N+1 Multipliers
    for (int n=0; n<N+1; n=n+1)
      m[n] = $signed(x) * $signed(k_arr[N-n]);

    a[0] = W_Y'($signed(m[0]));                N adders
    for (int n=1; n<N+1; n=n+1)
      a[n] =  W_Y'($signed(m[n]) + $signed(z[n-1]));

    y = a[N];
  end                                         N registers
  for (n=0; n<N; n=n+1)
    always_ff @(posedge clk or negedge rstn)
      if (!rstn)   z[n] <= '0;
      else if (en) z[n] <= a[n];
endmodule
```

Activity – Draw the waveform

Find the value in unsigned decimal

- Fast, at 30ns
- Fast, at 40 ns
- Slow at 60 ns

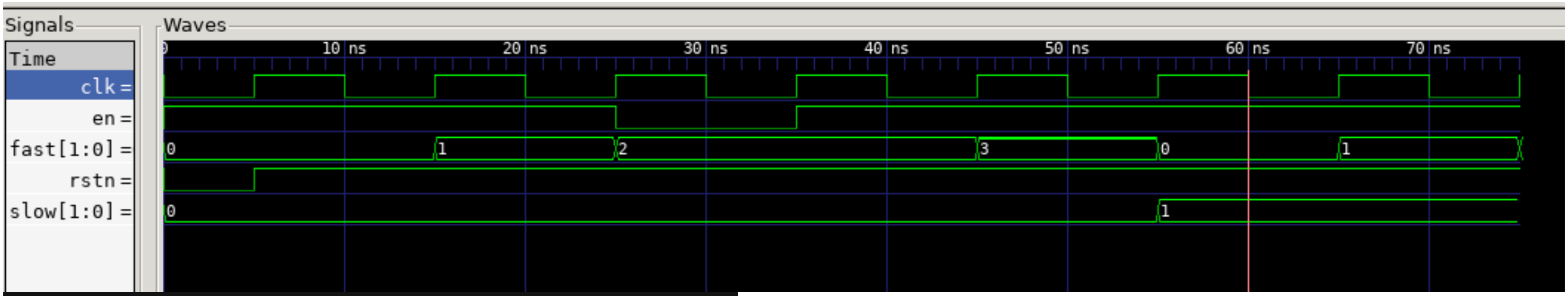
```
module nested_counter (
  input  logic clk, rstn, en,
  output logic [1:0] fast, slow);


  always_ff @(posedge clk)
    if (!rstn)
      {slow, fast} <= '0;
    else if (en) begin
      fast <= fast + 1'b1;
      if (fast == 2'd3)
        slow <= slow + 1'b1;
    end
endmodule
```

```
module tb;
  logic clk=0, rstn=0, en=1;
  logic [1:0] fast, slow;
  nested_counter dut (.*);
  initial forever #5 clk = ~clk;
  task automatic pclk(int n=1);
    repeat(n) @(posedge clk); #1ps;
  endtask


  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    pclk();   rstn=1; en=1;   // 5
    pclk(2);  en=0;           // 15, 25
    pclk();   en=1;           // 35
    pclk(4);  $finish;        // 45, 55, 65, 75
  end
endmodule
```

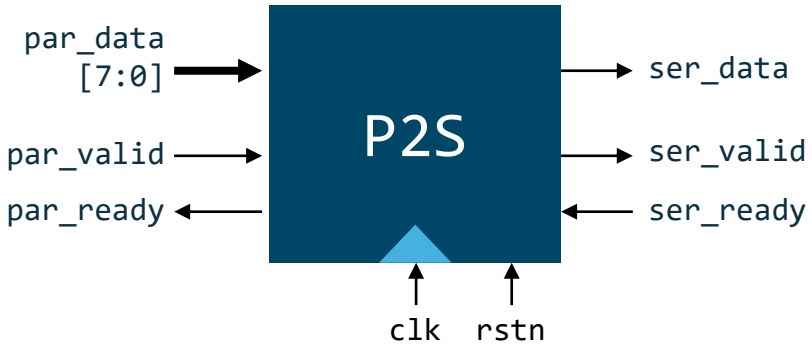
Activity – Draw the waveform

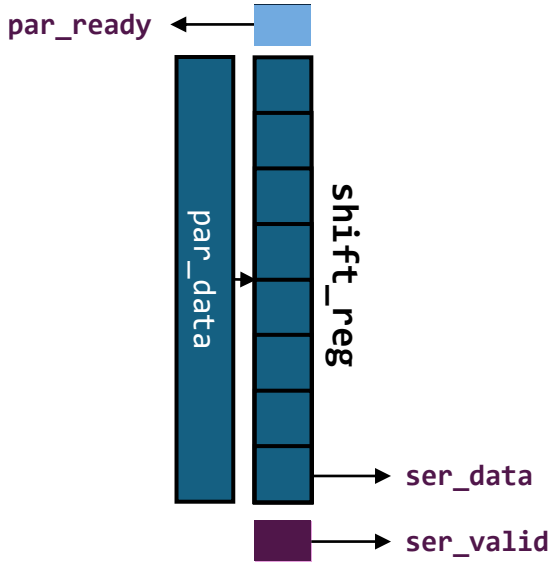



```
module nested_counter (
  input  logic clk, rstn, en,
  output logic [1:0] fast, slow);

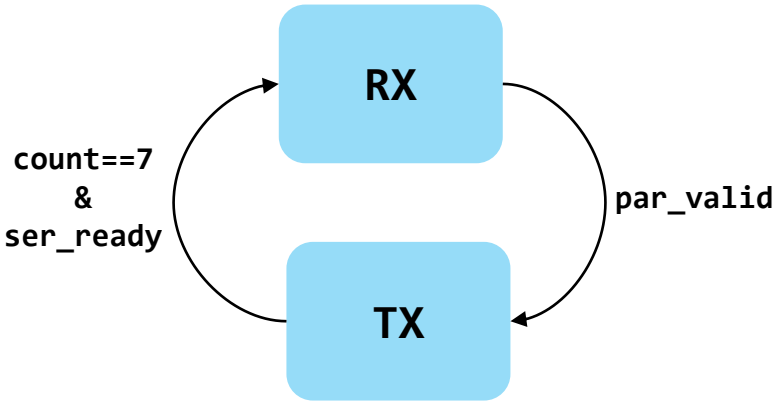
  always_ff @(posedge clk)
    if (!rstn)
      {slow, fast} <= '0;
    else if (en) begin
      fast <= fast + 1'b1;
      if (fast == 2'd3)
        slow <= slow + 1'b1;
    end
endmodule
```

Parallel to Serial – with Flow Control





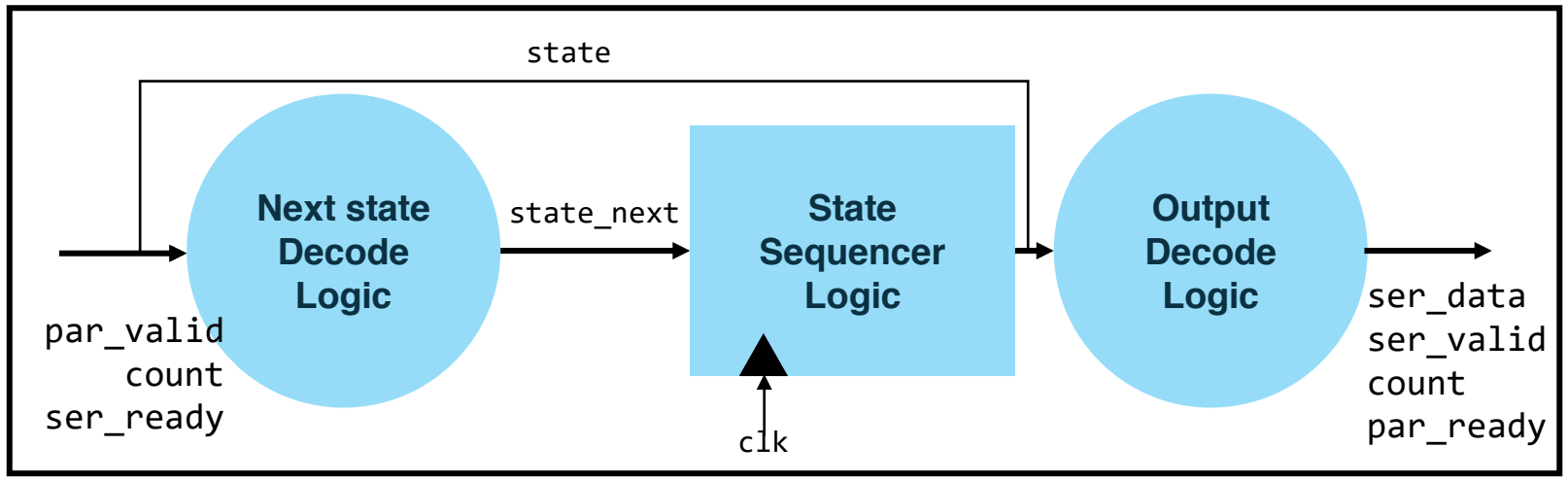

```
shift_reg <= par_data
par_ready    = 1
ser_valid    = 0
```



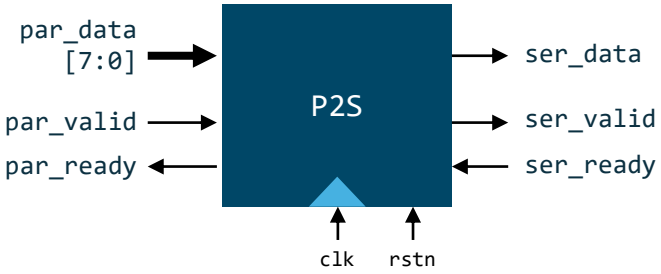

```
ser_valid   = 1
par_ready   = 0

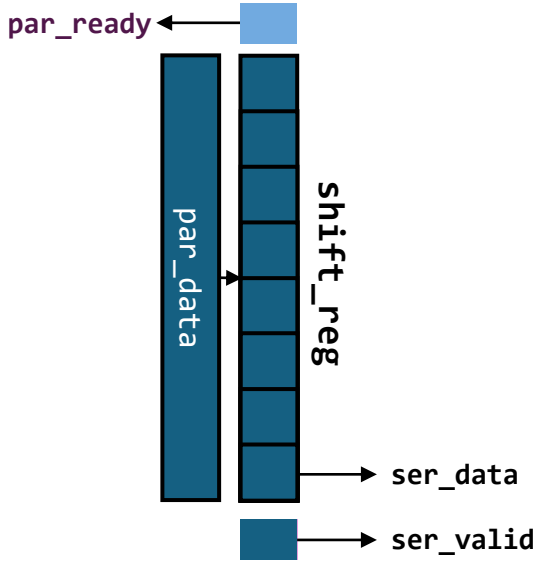
if (ser_ready)
  SHIFT
  count  <= count + 1
else
  DON'T SHIFT
  count  <= count
```

STATE MACHINE: THREE-PROCEDURE CODING STYLE

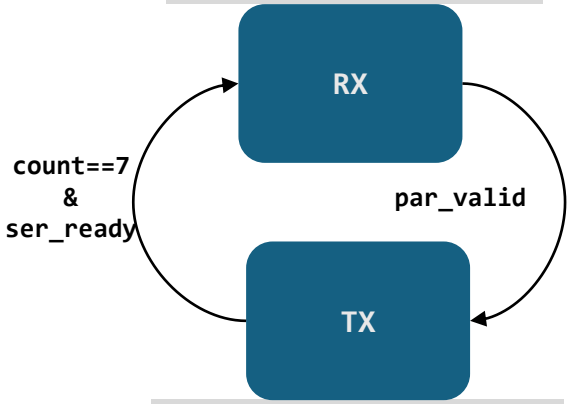


Parallel to Serial





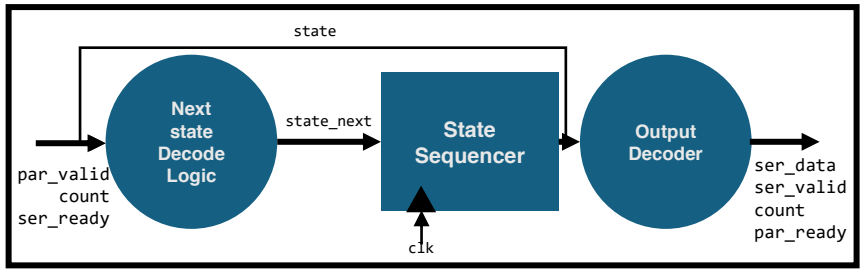

```
shift_reg <= par_data
par_ready    = 1
ser_valid    = 0
```




```
ser_valid   = 1
par_ready   = 0

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

THREE-PROCEDURE CODING STYLE




```
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;

                                                      OUTPUT DECODER
  always_comb begin
    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
```

P2S: Testbench

```
`timescale 1ns/1ps

module tb_parallel_to_serial;
  logic clk = 0, rstn = 0;
  initial forever #1ns clk = ~clk;
  task automatic posedge_clk(int n=1);
    repeat (n) @(posedge clk); #1ps;
  endtask

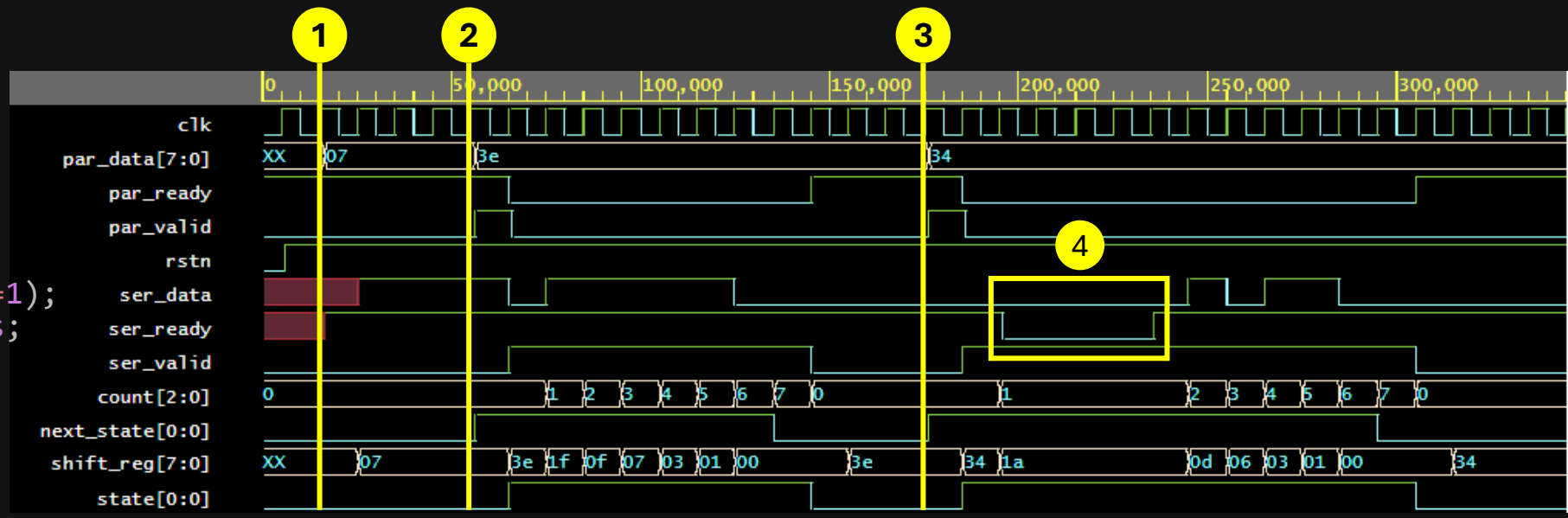
  parameter WIDTH = 8;
  logic [WIDTH-1:0] par_data;
  logic par_valid=0, par_ready,
    ser_data, ser_valid, ser_ready;

  parallel_to_serial
   #(.WIDTH(WIDTH)) dut (.*);

  initial begin
    $dumpfile(`FST_PATH); $dumpvars;
    posedge_clk;      rstn = 1;
```

Ready/Valid Flow Control

Data is transferred on a rising clock edge when both valid and ready are high.

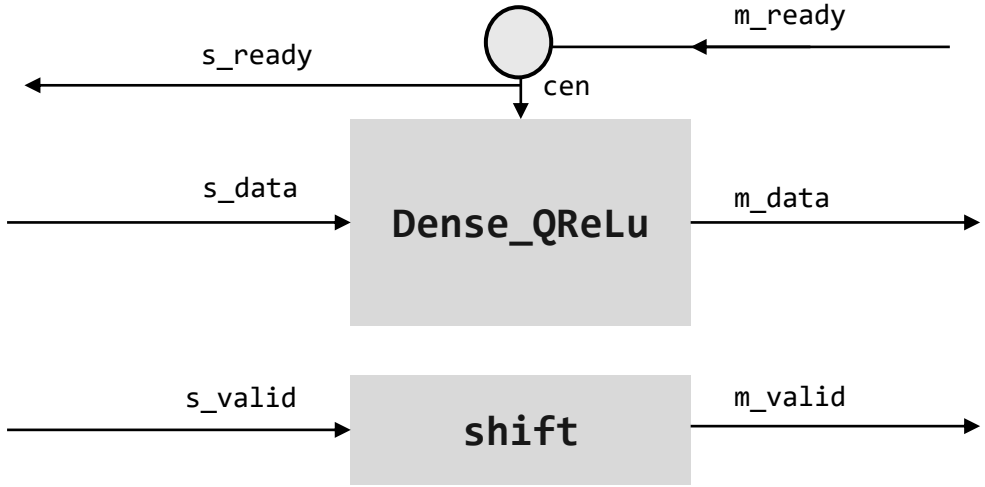



```
1   posedge_clk;      par_data  = 8'd7 ; par_valid = 0; ser_ready = 1;
    posedge_clk(3);
2   posedge_clk;      par_data  = 8'd62; par_valid = 1;
    posedge_clk;      par_valid = 0;
    posedge_clk(10);
3   posedge_clk;      par_data  = 8'd52; par_valid = 1;
    posedge_clk;      par_valid = 0;
4   posedge_clk;      ser_ready = 0;
    posedge_clk(3);
    posedge_clk;      ser_ready = 1;
    posedge_clk(10);
    posedge_clk;      ser_ready = 0;
    $finish();
  end
endmodule
```

Backpressure:
ready=0,
data & valid held

Valid/Ready Handshake

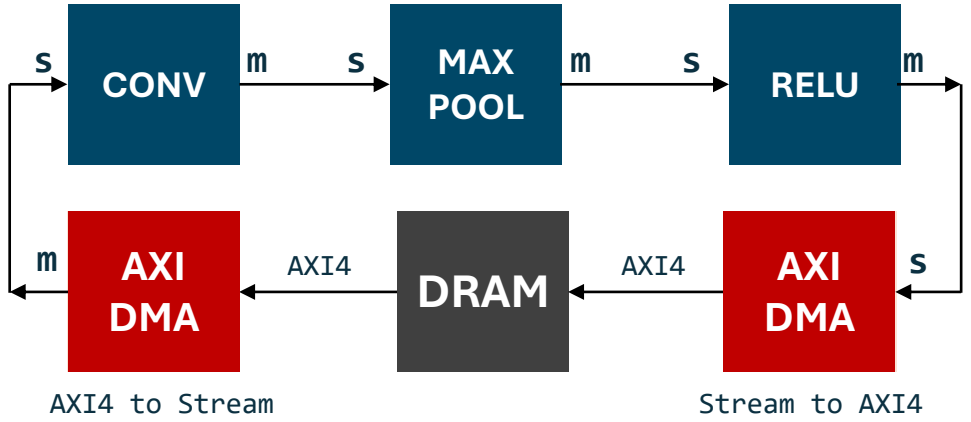
- Most common form of flow control
- Backpressure: ready = 0, upstream must freeze immediately
- Converting pipelined modules to AXI Stream (A4):
  - **`stall = m_valid & !m_ready`**
  - **`cen = !stall`**
  - **`s_ready = cen`**



AXI Stream Protocol

Bus

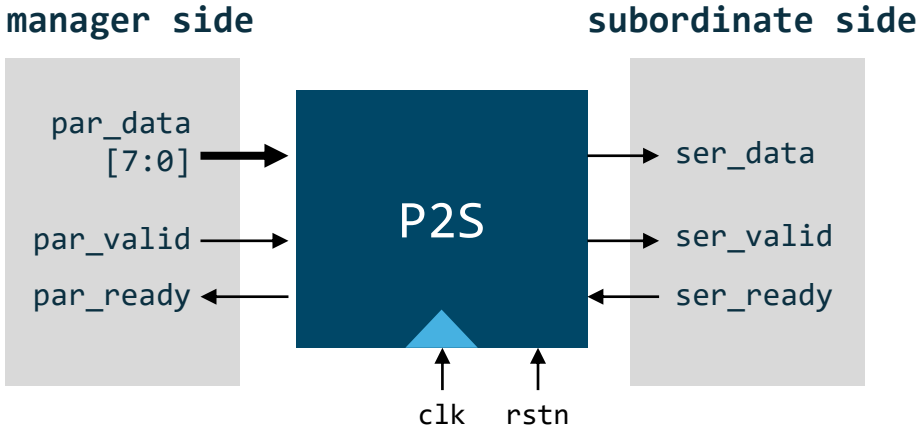
- Standardized set of wires and a “protocol”
- Used to keep modules independent & reusable when designing anything larger than this
- Eg: Bluetooth, Wi-Fi…etc. of mobile system-on-chips are made by different companies. But they follow the AXI (full) bus protocol.



AXI Stream

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

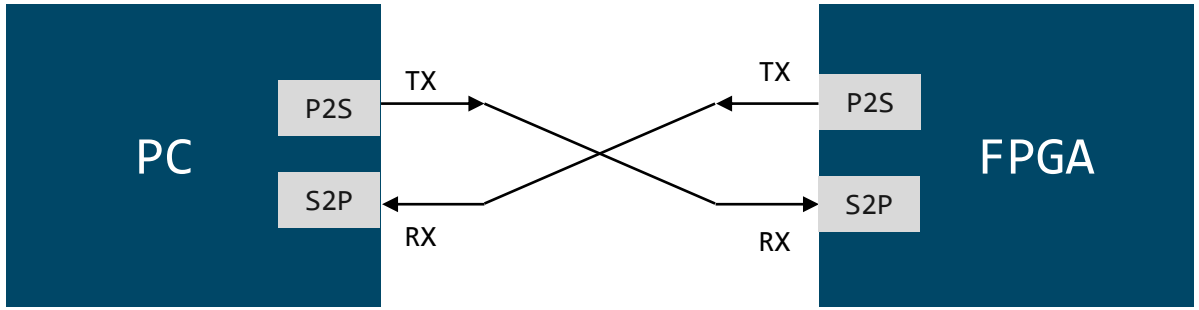
Protocol : If subordinate (downstream) gives ready and manager (upstream) gives valid in the same clock, data is transferred.

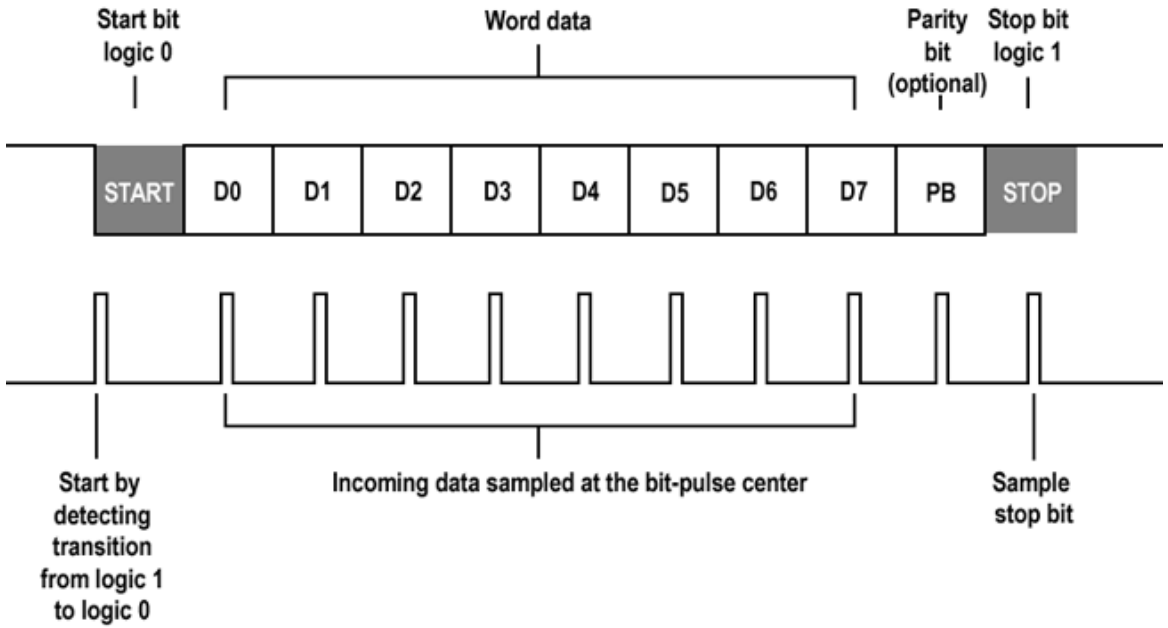


- Simplest & fastest bus I’ve seen. From ARM (AMBA/AXI)
- Useful for data pipelines where modules have independent data rates
- Backpressure: when a subordinate gets busy & pulls down ready
- Can be chained without long paths (unlike ours) using a skid buffer
- Each full AXI bus is made of 5 AXI Stream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

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!





Daily Feedback

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