EEL 4783: HDL in Digital System Design

Lecture 4: Architeching Area

Prof. Mingjie Lin

Digital Circuits: Area=Cost

- Choosing correct topology
  - Higher-level organization of the design
  - Not device specific
- Circuit-level reduction
  - Performed by the synthesis and layout tools
  - Minimization of the number of gates
  - May be device specific.
- Dilemma
  - Good topology for area
  - Reuse logic resources as much as possible

Main Points

- Rolling up the pipeline to reuse logic resources in different stages of a computation.
- Controls to manage the reuse of logic when a natural flow does not exist.
- Sharing logic resources between different functional operations.
- The impact of reset on area optimization.
    - Impact of FPGA resources that lack reset capability.
    - Impact of FPGA resources that lack set capability.
    - Impact of FPGA resources that lack asynchronous reset capability.
    - Impact of RAM reset.
    - Optimization using set/reset pins for logic implementation.

Rolling up the Pipeline

- “Rolling up” is direct opposite to “unrolling the loop”
- Unrolling a loop
  - Increase performance
  - Increase area → require more resource to hold immediate results and replicate computational structures for parallel
- Rolling up a pipeline
  - Optimize the area of pipelined designs with duplicated logic in the pipeline stages.

Example: Fixed Point Fractional Multiplier

```
module mult8(
 output [7:0]  product,
 input  [7:0]  A,
 input  [7:0]  B,
 input         clk);
 reg    [15:0] prod16;

 assign product = prod16[15:8];

 always @(posedge clk)
   prod16 <= A * B;

endmodule
```

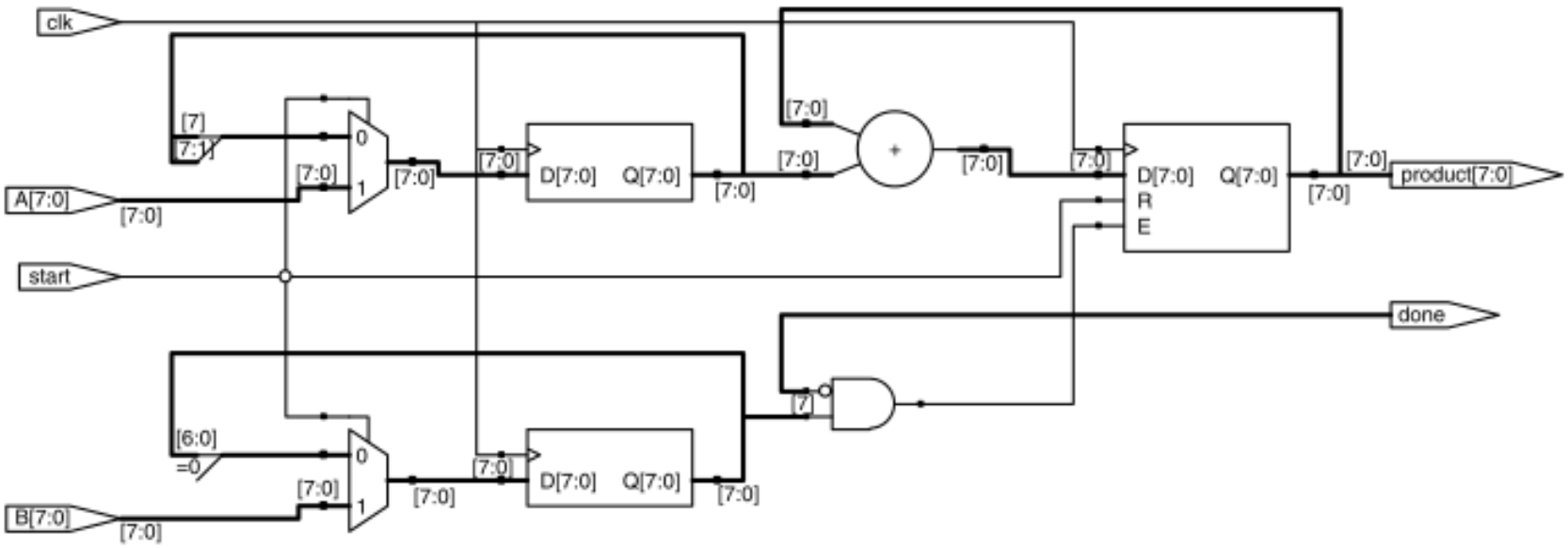
Pipelined Version

```
module mult8(
 output            done,
 output reg [7:0] product,
 input      [7:0] A,
 input      [7:0] B,
 input            clk,
 input            start);
 reg        [4:0] multcounter; // counter for number of
                                shift/adds
```

Pipelined Version +

```
reg        [7:0] shiftB; // shift register for B
reg        [7:0] shiftA; // shift register for A

wire adden; // enable addition

assign adden = shiftB[7] & !done;
assign done = multcounter[3];

always @(posedge clk) begin
  // increment   multiply counter for shift/add ops
  if(start)      multcounter <= 0;
  else if(!done) multcounter <= multcounter + 1;

  // shift register for B
  if(start) shiftB <= B;
  else shiftB[7:0] <= {shiftB[6:0], 1'b0};

  // shift register for A
  if(start) shiftA <= A;
  else shiftA[7:0] <= {shiftA[7], shiftA[7:1]};

  // calculate multiplication
  if(start)      product <= 0;
  else if(adden) product <= product + shiftA;
end
endmodule
```

Logic Diagram



- Throughput
- Latency
- Timing

Control-Based Logic Reuse

- Sharing logic resources oftentimes requires special control circuitry to determine which elements are input to the particular structure
- Previously, we described a multiplier that simply shifted the bits of each register, where each register was always dedicated to a particular input of the running adder. This had a natural data flow that lent itself well to logic reuse
- However, there are often more complex variations to the input of a resource, and certain controls may be necessary to reuse the logic
- Controls can be used to direct the reuse of logic when the shared logic is larger than the control logic.
- A state machine may be required as an additional input to the logic.

Low-Pass FIR Filter

$$Y = \text{coeffA} * X[0] + \text{coeffB} * X[1] + \text{coeffC} * X[2]$$

```
module lowpassfir(
 output reg [7:0] filtout,
 output reg       done,
 input            clk,
 input      [7:0] datain, // X[0]
 input            datavalid, // X[0] is valid
 input      [7:0] coeffA, coeffB; coeffC); // coeffs for
                                             low pass
                                             filter
 // define input/output samples
 reg        [7:0] X0, X1, X2;
 reg              multdonedelay;
 reg              multstart; // signal to multiplier to
                             begin computation
 reg        [7:0] multdat;
 reg        [7:0] multcoeff; // the registers that are
                             multiplied together
 reg        [2:0] state; // holds state for sequencing
                           through mults
 reg        [7:0] accum; // accumulates multiplier products
 reg                      clearaccum; // sets accum to zero
 reg        [7:0] accumsum;
 wire             multdone; // multiplier has completed
 wire       [7:0] multout; // multiplier product
```

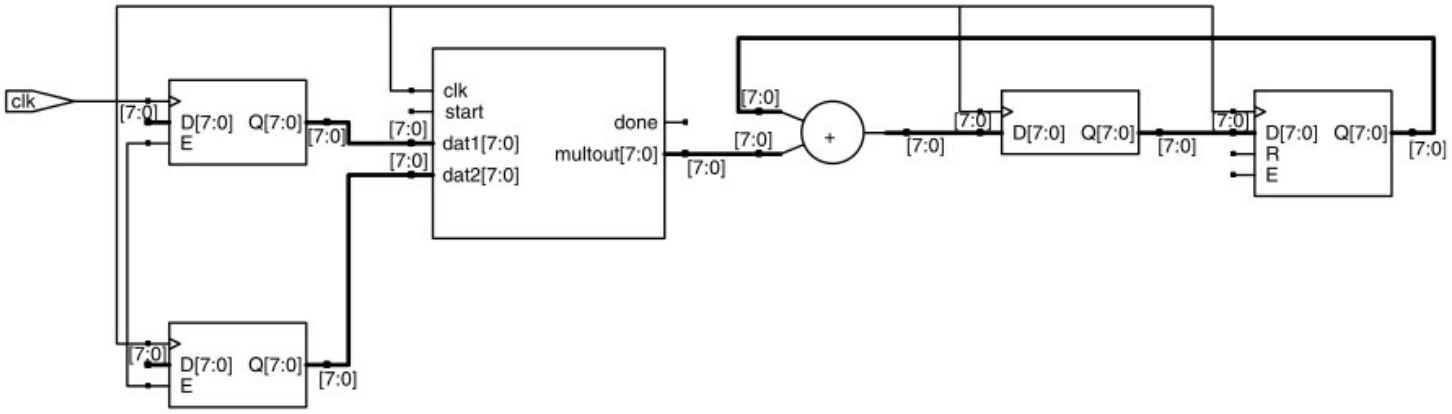
```
    // shift-add multiplier for sample-coeff mults
    mult8x8 mult8x8(.clk(clk), .dat1(multdat),
         .dat2(multcoeff), .start(multstart),
         .done(multdone), .multout(multout));

      multdonedelay <= multdone;

     // accumulates sample-coeff products
     accumsum <= accum + multout[7:0];

     // clearing and loading accumulator
     if(clearaccum)          accum <= 0;
     else if(multdonedelay) accum <= accumsum;
// do not process state machine if multiply is not done
case(state)
      0: begin
      // idle state
      if(datavalid) begin
       // if a new sample has arrived
       // shift samples
       X0     <= datain;
       X1     <= X0;
       X2     <= X1;
       multdat  <= datain;   // load mult
       multcoeff <= coeffA;
       multstart <= 1;
       clearaccum <= 1; // clear accum
       state   <= 1;
      end
```

```
else begin
 multstart <= 0;
 clearaccum <= 0;
 done    <= 0;
end
end
1: begin
if(multdonedelay) begin
 // A*X[0] is done, load B*X[1]
 multdat  <= X1;
 multcoeff <= coeffB;
 multstart <= 1;
 state   <= 2;
end
else begin
 multstart <= 0;
 clearaccum <= 0;
 done    <= 0;
end
end
2: begin
if(multdonedelay) begin
 // B*X[1] is done, load C*X[2]
 multdat  <= X2;
```

```
      multcoeff <= coeffC;
      multstart <= 1;
      state   <= 3;
     end
     else begin
      multstart <= 0;
      clearaccum <= 0;
      done    <= 0;
     end
     end
     3: begin
     if(multdonedelay) begin
      // C*X[2] is done, load output
      filtout  <= accumsum;
      done    <= 1;
      state   <= 0;
     end
     else begin
      multstart <= 0;
      clearaccum <= 0;
      done    <= 0;
     end
     end
     default
      state   <= 0;
    endcase
  end
endmodule
```

Observations

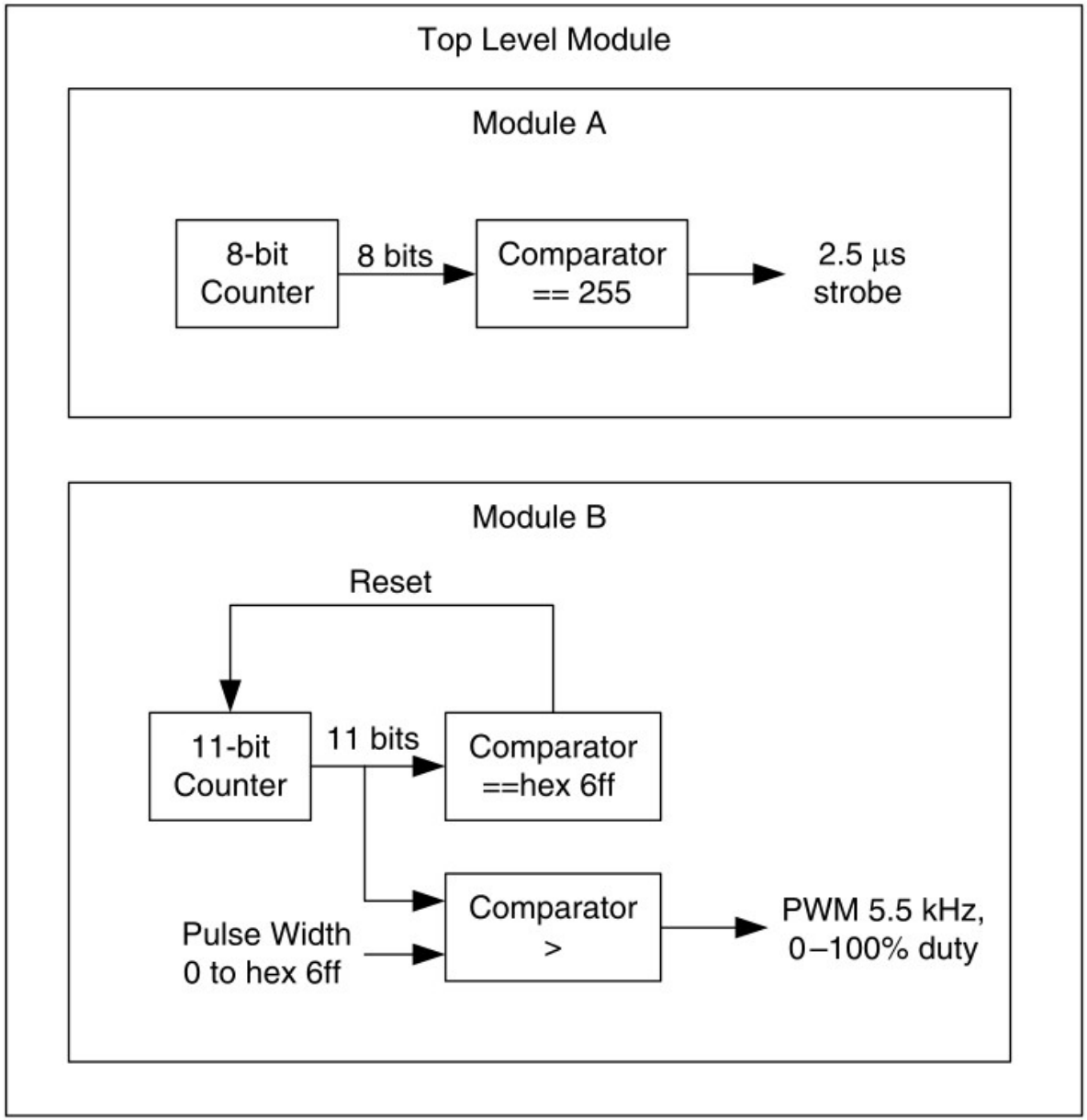
1. A single multiplier
2. A single accumulator
3. A FSM is used to load coefficients and registered samples into the multiplier. The state machine operates on every combination of coefficients and samples



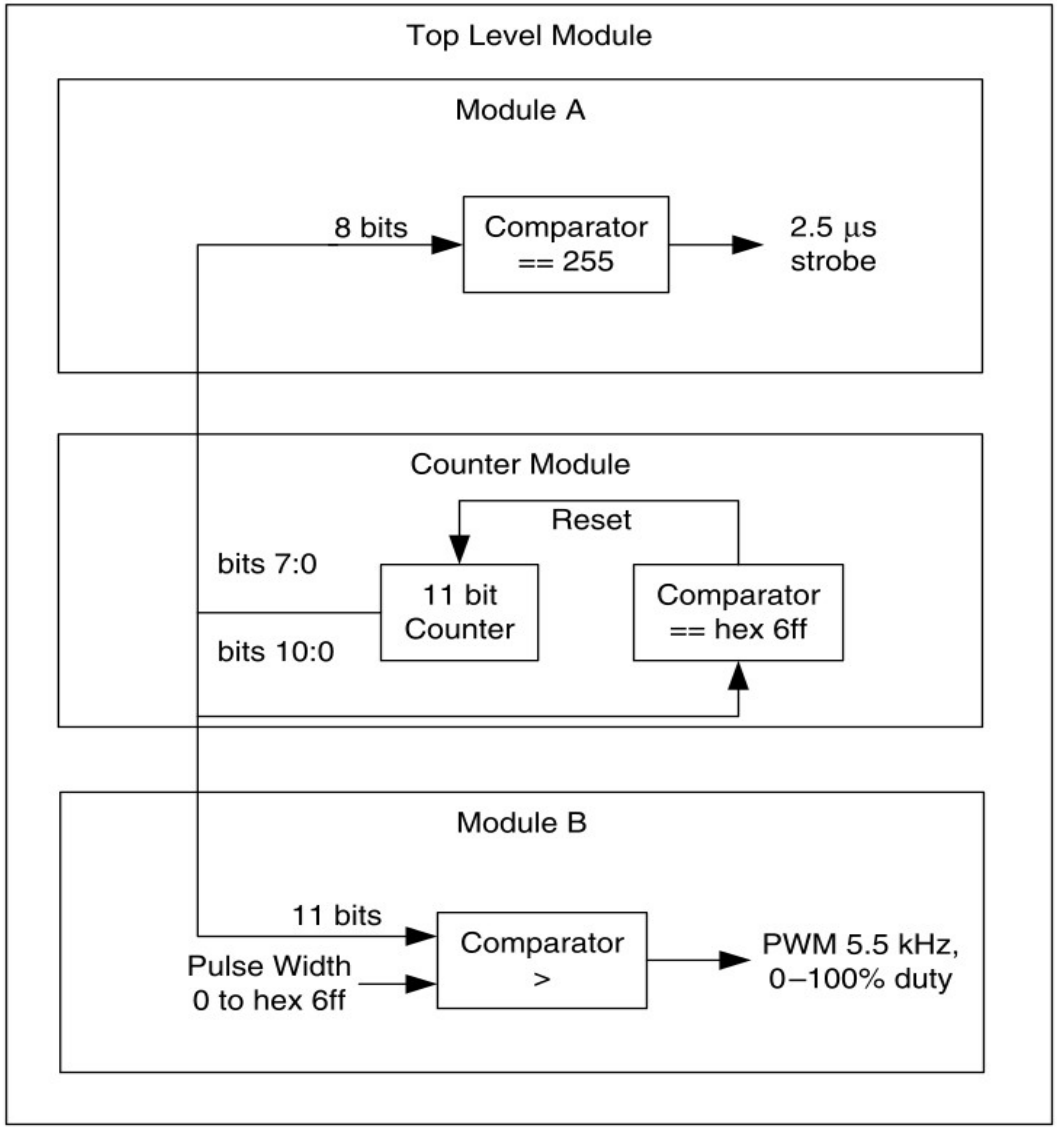
Resource Sharing

- Higher-level architectural resource sharing where different resources are shared across different functional boundaries
- This type of resource sharing should be used whenever there are functional blocks that can be used in other areas of the design or even in different modules

Example: Separate Counters



Example: Shared Counters



Impact of RESET on Area

- A common misconception is that the reset structures are always implemented in a purely global sense and have little effect on design size
- A global set/reset condition for every flip-flop. Although this may seem like good design practice, it can often lead to a larger and slower design
- An improper reset strategy can create an unnecessarily large design and inhibit certain area optimizations

Resources Without Reset

IMPLEMENTATION 1 : *Synchronous Reset*

```
always @(posedge iClk)
 if(!iReset) sr <= 0;
 else sr        <= {sr[14:0], iDat};
```

IMPLEMENTATION 2 : *No Reset*

```
always @(posedge iClk)
 sr <= {sr[14:0], iDat};
```

Final issues

- Please fill out the student info sheet before leaving
- Come by my office hours (right after class)
- Any questions or concerns?