EEL 4783: HDL in Digital System Design

Lecture 10: Synthesis Optimization

Prof. Mingjie Lin
What Can We Do?

- Trade-offs with speed versus area.
- Resource sharing for area optimization.
- Pipelining, retiming, and register balancing for performance optimization.
- The effect of reset on register balancing.
- Handling resynchronization registers.
- Optimizing FSMs.
- Handling black boxes.
- Physical synthesis for performance.
Speed vs. Area

Most synthesis tools provide switches that allow the designer to target speed versus area optimization

No-brainer:
- If you want it to run faster, choose speed
- If you want it to be smaller, choose area

This switch is mis-leading because it is a generalization of certain algorithms that can sometimes produce the opposite result
- i.e., the design becomes slower after telling it to go faster

Before we understand why this happens, we...
At the synthesis level, speed and area optimizations determine the logic topology that will be used to implement our RTL.

- At this level of abstraction, there is little known about the physical nature of the FPGA.
- The interconnect delay based on the place and route.

Synthesis tools use what are called wire load models, which are statistical estimates of interconnect delay based on various criteria of the design.

- In an ASIC, this is accessible to the designer.
- But with FPGA design this is hidden behind the scenes.

It is possible that the synthesis tool comes up with its estimates, which are often significantly different from the end result.

- Due to this lack of knowledge from the back end, synthesis tools will primarily execute gate-level optimizations.
- In high-end FPGA design tools, there exists a flow called placement-based synthesis to help close this loop.
The synthesis-based gate-level optimizations will include things like:
- State-machine encoding
- Parallel versus staggered muxing
- Logic duplication

As a general rule of thumb (although certainly not always true),
- Faster circuits require more parallelism, which equates with a larger circuit
- Therein lies the basic conceptual trade-off between speed and area: Faster circuits require more parallelism and an increase in area.
Unfortunately, for FPGA

- It isn’t until place and route is completed before the tool really knows how congested the device is or the difficulty in the place and route process.
- At this point in the flow, a particular logic topology has already been committed to by the synthesis tool.
- Thus, if an optimization effort was set to speed at the synthesis level and the back-end tool finds that the device is overly congested, it must still attempt to place and route all the extra logic.
- When the device is congested, the tool will have no choice but to place the components wherever they will fit and will therefore introduce long delays due to the suboptimal routes.
- Because of the fact that designers will often use the smallest FPGA possible for economic reasons, this situation occurs very frequently.
General Heuristic

As the resource utilization approaches 100%, a speed optimization at the synthesis level
Understand the Graph

- Underconstrained:
  - This is the flat region near the bottom where the constraint was defined to be less than 95 MHz. In this region, a compact implementation of the logic will run at approximately 95 MHz without any significant timing optimizations.

- Optimization region:
  - The linear region between 95 MHz and 135 MHz that represents the range where increases in the timing constraints can be fulfilled by corresponding optimizations in the logic implementation. In other words, the timing constraints in this region can be met by synthesizing higher speed (and correspondingly higher area) logic structures.

- Peak:
  - The upper peak represents the maximum constraint that can be satisfied with improvements to the logic structure given the parallel architectures for the specific design and the amount of space available in the FPGA.

- Overconstrained:
  - This is the flat region near the top where the constraint exceeds the maximum achievable frequency.
Resource Sharing

- Portions of the design that can be used for different blocks of functionality are reused via steering logic.
- At a high level, this type of architecture can dramatically reduce the overall area with a penalty that may include throughput if the operations are not mutually exclusive.
- Resource sharing on the synthesis optimization level typically operates on groups of logic between register stages.
- These simpler architectures can be boiled down to solving basic logic operations.
Example 1

module addshare
output oDat,
input iDat1, iDat2, iDat3,
inpu t iSel);

assign oDat = iSel ? iDat1 + iDat2: iDat1 + iDat3;
endmodule
Example 1
Example 2

module addshare (  
    output    oDat,  
    input     iDat1, iDat2, iDat3,  
    input     [1:0] iSel);  

    assign oDat = (iSel == 0) ? iDat1 + iDat2:  
                    (iSel == 1) ? iDat1 + iDat3:  
                            iDat2 + iDat3;  

endmodule
Example 2

- If resource sharing is activated, verify that it is not adding delay to the critical path.
PIPELINING, RETIMING, AND REGISTER BALANCING

- Pipelining was a method that was used to increase the throughput and flip-flop to flip-flop timing by adding register stages between groups of logic.
- A well-designed module can usually be pipelined by adding additional register stages and only impact total latency with a small penalty in area.
- The synthesis options for pipelining, retiming, and register balancing operate on the same structures but do not add or remove the registers themselves.
- Instead, these optimizations move flip-flops around logic to balance the amount of delay between any two register stages and therefore minimize the worst-case delay.
- Pipelining, retiming, and register balancing are very similar in meaning and often only vary slightly from vendor to vendor.
Balancing Combinatorial Logic

Delay $\ll T_{period}$ (critical path)

Delay $= T_{period}$

Delay $< T_{period}$

Clock

Comb logic

Comb logic
Pipelining typically refers to the first widely adopted method of load balancing whereby regular structures such as pipelined memories or multipliers could be identified by the synthesis tool and rearchitected with redistributed logic. Pipelining requires that a regular pipeline exists and that it is easily recognizable by the tool.

module multipipe #(parameter width = 8, parameter depth = 3) (  
    output [2*width-1: 0] oProd,  
    input  [width-1: 0]  iIn1, iIn2,  
    input       iClk);  
reg     [2*width-1: 0] ProdReg [depth-1: 0];  
integer    i;  
assign  oProd    = ProdReg [depth-1];  
always @(posedge iClk) begin  
    ProdReg[0] <= iIn1 * iIn2;  
    for(i=1;i <depth;i=i+1)  
        ProdReg[i] <= ProdReg [i-1];  
end  
endmodule
module genpipe (
    output reg oProd,
)
Improved Example

input [7:0] iIn1,
input iReset,
input iClk);
reg [7:0] inreg1;

always @(posedge iClk)
  if(iReset) begin
    inreg1 <= 0;
    oProd <= 0;
  end
  else begin
    inreg1 <= iIn1;
    oProd <= (inreg1[0]|inreg1[1]) & (inreg1[2]|inreg1[3]) &
  end
endmodule
Imbalanced Logic
Balanced Logic
The Effect of Reset on Register Balancing

- Reset can have a direct impact on the ability of the synthesis tool to use register balancing.
- Specifically, if two flip-flops are required to combine to balance the logic load, the two flip-flops must have the same reset state.
  - For instance, if one reset has a synchronous reset and another an asynchronous reset (which would typically be poor design practice).
  - Or if one had a set versus a reset, the two could not be combined, and register balancing would have no effect.
Adjacent flip-flops with different reset types may prevent register balancing from taking place.
FSM Compilation

- FSM compilation refers to the automatic identification of a finite state machine in the RTL and recoding as needed for the speed/area constraints.
  - This means that as long as a standard state-machine architecture is used, the exact coding in the RTL is unimportant.

- Due to the regular structure of a state machine coded with a standard style, the synthesis tool can easily extract the state transitions and output dependencies and transform the FSM into something that is more optimal for a given design and set of constraints.
Key to Success

- Design state machines with standard coding styles so they can be identified and reoptimized by the synthesis tool
  - Binary and sequential encoding will depend on all flip-flops in the state representation, and thus a state-decode will be necessary
  - FPGA technologies that are logic rich or that have multiple input gates for the decode logic will optimally implement these FSMs
State Encoding

• One-hot encoding is implemented such that one unique bit is set for each state
  – With this encoding, there is no state decode and the FSM will usually run faster.
  – The disadvantage is that one-hot encodings typically require many registers.

• Gray codes are a common alternative to one-hot encoding in two primary applications:
  – Asynchronous outputs
  – Low-power devices
Why Gray Encoding?

- If the output of the state machine, or any of the logic that the state machine operates on, is asynchronous, gray codes are typically preferred
  - This is due to the fact that asynchronous circuits are not protected from race conditions and glitches. Thus, the path differential between two bits in the state register can cause unexpected behavior and will be very dependent on layout and parasitics.
  - Consider the output encoding for a Moore machine as shown below. In this case, state transition events will occur where a single bit will be cleared and a single bit will be set, thereby creating the potential for race conditions
Illustration

State bit 1
State bit 2
State bit 1
State bit 3

async output

State bit 1
State bit 2
State bit 3

State bit 1 & !State bit 2
!State bit 1 & State bit 3
async output

→ Delay = AND gate
→ Delay = AND + Invert
→ Static-1 hazard

Figure 14.15 Potential hazard.
Why Gray Coding?

- One solution to this problem is to use gray encoding.
- A gray code only experiences a single bit change for any transition.
- The fact that gray codes can be used to safely drive asynchronous outputs is apparent after analyzing the structure of coding scheme.
- To construct a gray code, use the mirror-append sequence as described below
  - 1. Begin with a “0” and a “1” listed vertically.
  - 2. Mirror the code from the bottom digit.
  - 3. Append “0” to the upper half of the code (the section that was copied in the mirror operation).
  - 4. Append “1” to the lower half of the code (the section that was created in the mirror operation).
Creating Gray Codes

- To construct a gray code, use the mirror-append sequence as described below:
  - 1. Begin with a “0” and a “1” listed vertically.
  - 2. Mirror the code from the bottom digit.
  - 3. Append “0” to the upper half of the code (the section that was copied in the mirror operation).
  - 4. Append “1” to the lower half of the code (the section that was created in the mirror operation).