Lecture 16 - RTL Design

RTL Examples

RTL Design Pitfalls and Good Practices

Control and Data Dominated RTL Design

Video is a series of frames (e.g., 30 per second)

Most frames similar to previous frame

Compression idea: just send difference from previous frame

Digitized frame 2

Digitized frame 1

1 Mbyte

0.01 Mbyte

Just send difference

Need to quickly determine whether two frames are similar enough to just send difference for second frame

Compare corresponding 16x16 "blocks"

Treat 16x16 block as 256-byte array

Compute the absolute value of the difference of each array item

Sum those differences - if above a threshold, send complete frame for second frame; if below, can use difference method (using another technique, not described)

Want fast sum-of-absolute-differences (SAD) component

When go=1, sums the differences of element pairs in arrays A and B, outputs that sum

S0: wait for go

S1: initialize sum and index

S2: check if done (i>=256)

S3: add difference to sum, increment index

S4: done, write to output sad_reg

Step 2: Create datapath

Datapath

Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)
RTL Example: Video Compression – Sum of Absolute Differences

Step 3: Connect to controller
Step 4: Replace high-level state machine by FSM

Comparing software and custom circuit SAD
- Circuit: Two states (S2 & S3) for each i, 256 / 512 clock cycles
- Software: Loop (for i = 1 to 256), but for each i, must move memory to local registers, subtract, compute absolute value, add to sum, increment i – say about 6 cycles per array item \( 256 \times 6 \) = 1536 cycles
- Circuit is about 3 times (300%) faster
- Later, we’ll see how to build SAD circuit that is even faster

RTL Design Pitfalls and Good Practice
- Common pitfall: Assuming register is updated in the state it’s written
  - Final value of Q?
  - Final state?
  - Answers may surprise you
    - Value of Q unknown
    - Final state is C, not D
  - Why?
    - State A: \( R = 99 \) and \( Q = R \) happen simultaneously
    - State B: \( R \) not updated with \( R+1 \) until next clock cycle, simultaneously with state register being updated

Solutions
- Read register in following state (Q=R)
- Insert extra state so that conditions use updated value
- Other solutions are possible, depends on the example

Good practice: Register all data outputs
- In fig (a), output \( P \) would show spurious values as addition computes
- Furthermore, longest register-to-register path, which determines clock period, is not known until that output is connected to another component
- In fig (b), spurious outputs reduced, and longest register-to-register path is clear
Control vs. Data Dominated RTL Design

- Designs often categorized as control-dominated or data-dominated
  - Control-dominated design - Controller contains most of the complexity
  - Data-dominated design - Datapath contains most of the complexity
  - General, descriptive terms - no hard rule that separates the two types of designs
  - Laser-based distance measurer - control dominated
  - Bus interface, SAD circuit - mix of control and data
  - Now let’s do a data dominated design

Data Dominated RTL Design Example: FIR Filter

- Filter concept
  - Suppose X is data from a temperature sensor, and particular input sequence is 180, 180, 181, 240, 180, 181 (one per clock cycle)
  - That 240 is probably wrong!
  - Could be electrical noise
  - Filter should remove such noise in its output
  - Simple filter: Output average of last N values
    - Small N: less filtering
    - Large N: more filtering, but less sharp output

- FIR filter
  - “Finite Impulse Response”
  - Simply a configurable weighted sum of past input values
    - \( y(t) = c_0x(t) + c_1x(t-1) + c_2x(t-2) \)
    - Above known as “3 tap”
    - Tens of taps more common
    - Very general filter - User sets the constants \( c_0, c_1, c_2 \) to define specific filter

- RTL design
  - Step 1: Create high-level state machine
    - But there really is none! Data dominated indeed.
    - Go straight to step 2

- Step 2: Create datapath (cont.)
  - Instantiate registers for \( c_0, c_1, c_2 \)
  - Instantiate multipliers to compute \( c*x \) values

- Instantiate adders to sum the multiplied values
Data Dominated RTL Design Example:
FIR Filter

- Step 2: Create datapath (cont.)
  - Add circuitry to allow loading of particular c register

\[ y(t) = c_0 x(t) + c_1 x(t-1) + c_2 x(t-2) \]

**Design Challenge**

- Using the RTL design method, convert the following high-level state machine to datapath and controller. Design the datapath to structure, but design the controller to the point of an FSM only.

Due:  
- Next Lecture (Friday, October 28)
- Extra Credit (Homework)
  - 2 points

Data Dominated RTL Design Example:
FIR Filter

- Step 3 & 4: Connect to controller, Create FSM
  - No controller needed
  - Extreme data-dominated example
  - (Example of an extreme control-dominated design – an FSM, with no datapath)

Comparing the FIR circuit to a software implementation

**Circuit**
- Assume adder has 2-gate delay, multiplier has 20-gate delay
- Longest path goes through one multiplier and two adders
  - \( 20 + 2 + 2 = 24 \) gate delay
- 100-tap filter, following design on previous slide, would have about a 34-gate delay: 1 multiplier and 7 adders on longest path

**Software**
- 100-tap filter: 100 multiplications, 100 additions. Say 2 instructions per multiplication, 2 per addition. Say 10-gate delay per instruction.
  - \( (100 \times 2 + 100 \times 2) \times 10 = 4000 \) gate delays
- Circuit is more than 100 times faster (10,000% faster). Wow.