CSE140: Components and Design Techniques for Digital Systems

RTL Design Process

Tajana Simunic Rosing
Announcements and Outline

• CAPE season opens Monday, 5/24!!!!
  – Please go to http://www.cape.ucsd.edu/ and fill out your evaluation

• Pick up graded homework at my assistant’s office
  – Check webct grades; make sure everything is correct
  – Graded/regraded exams are in my office

• Next exam on 6/3, class time
  – Everything covered in lectures, whole book & all handouts
  – Format:
    • Problems similar to HW and previous exams
    • May have multiple choice and/or T/F questions on the assigned reading
  – Tuesday, 6/1 lecture will go over the previous year’s final and additional examples

• Discussion session this week will go over RTL problems
• Next: More RTL examples, CPU design
### RTL Design Method

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong></td>
<td><strong>Capture a high-level state machine</strong></td>
</tr>
<tr>
<td></td>
<td>Describe the system’s desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is “high-level” because the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs.</td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td><strong>Create a datapath</strong></td>
</tr>
<tr>
<td></td>
<td>Create a datapath to carry out the data operations of the high-level state machine.</td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td><strong>Connect the datapath to a controller</strong></td>
</tr>
<tr>
<td></td>
<td>Connect the datapath to a controller block. Connect external Boolean inputs and outputs to the controller block.</td>
</tr>
<tr>
<td><strong>Step 4</strong></td>
<td><strong>Derive the controller’s FSM</strong></td>
</tr>
<tr>
<td></td>
<td>Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.</td>
</tr>
</tbody>
</table>
RTL for Datapath & Control

(a) High-level block diagram

(b) Register-transfer-level block diagram
RTL Design example: Laser-Based Distance Measurer

- Laser-based distance measurement – pulse laser, measure time $T$ to sense reflection
  - Laser light travels at speed of light, $3 \times 10^8$ m/sec
  - Distance is thus $D = T \text{ sec} \times 3 \times 10^8 \text{ m/sec} / 2$
Step 4: Deriving the Controller’s FSM

- FSM has same structure as high-level state machine
  - Inputs/outputs all bits now
  - Replace data operations by bit operations using datapath

Inputs: B, S (1 bit each)  
Outputs: L (bit), D (16 bits)

Local Registers: Dctr (16 bits)

- **S0**:  
  - L = 0  
  - Dctr = 0  
  - Dreg_clr = 1  
  - Dreg_ld = 0  
  - Dctr_clr = 0  
  - Dctr_cnt = 0  
  - (laser off)  
  - (clear D reg)

- **S1**:  
  - L = 0  
  - Dctr = 0  
  - Dreg_clr = 0  
  - Dreg_ld = 0  
  - Dctr_clr = 0  
  - Dctr_cnt = 0  
  - B'  
  - S'  
  - B

- **S2**:  
  - L = 1  
  - Dctr = Dctr + 1  
  - Dreg_clr = 0  
  - Dreg_ld = 0  
  - Dctr_clr = 1  
  - Dctr_cnt = 0  
  - D = Dctr / 2  
  - (calculate D)

- **S3**:  
  - L = 0  
  - Dctr = Dctr + 1  
  - Dreg_clr = 0  
  - Dreg_ld = 0  
  - Dctr_clr = 0  
  - Dctr_cnt = 0  
  - (laser off)

- **S4**:  
  - L = 0  
  - Dctr = Dctr + 1  
  - Dreg_clr = 0  
  - Dreg_ld = 0  
  - Dctr_clr = 0  
  - Dctr_cnt = 0  
  - (count up)

---

Sources: TSR, Katz, Boriello, Vahid, Perkowski
RTL Design Method Example

• Soda dispenser
  – $c$: bit input, 1 when coin deposited
  – $a$: 8-bit input having value of deposited coin
  – $s$: 8-bit input having cost of a soda
  – $d$: bit output, processor sets to 1 when total value of deposited coins equals or exceeds cost of a soda
Step 1: Capture High-Level State Machine

- Declare local register \( tot \)
- **Init** state: Set \( d=0, \; tot=0 \)
- **Wait** state: wait for coin
  - If see coin, go to **Add** state
- **Add** state: Update total value: \( tot = tot + a \)
  - Remember, \( a \) is present coin’s value
  - Go back to **Wait** state
- In **Wait** state, if \( tot \geq s \), go to **Disp**(ense) state
- **Disp** state: Set \( d=1 \) (dispense soda)
  - Return to **Init** state

Not an FSM because:
- Multi-bit (data) inputs \( a \) and \( s \)
- Local register \( tot \)
- Data operations \( tot=0, \; tot<s, \; tot=tot+a \)

Useful high-level state machine:
- Data types beyond just bits
- Local registers
- Arithmetic equations/expressions

Source: TSR, Katz, Boriello, Vahid, Perkowski
Step 2: Create Datapath

- Need tot register
- Need 8-bit comparator to compare s and a
- Need 8-bit adder to perform \( \text{tot} = \text{tot} + a \)
- Connect everything
- Create control input/outputs

**Datapath**

- Inputs: c (bit), a (8 bits), s (8 bits)
- Outputs: d (bit)
- Local registers: tot (8 bits)
Step 3: Connect Datapath to a Controller

- Controller’s inputs
  - External input $c$ (coin detected)
  - Input from datapath comparator’s output, which we named $tot\_lt\_s$
- Controller’s outputs
  - External output $d$ (dispense soda)
  - Outputs to datapath to load and clear the $tot$ register
Step 4 – Derive the Controller’s FSM

- Same states and arcs as high-level state machine
- But set/read datapath control signals for all datapath operations and conditions
Completing the Design

- Implement the FSM as a state register and logic

Inputs: c, tot_lt_s (bit)
Outputs: d, tot_ld, tot_clr (bit)

<table>
<thead>
<tr>
<th>s1</th>
<th>s0</th>
<th>c</th>
<th>tot lt_s</th>
<th>n1</th>
<th>n0</th>
<th>d</th>
<th>tot ld</th>
<th>tot clr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**RTL Design Example: Bus Interface**

- **Example: Bus interface**
  - Master processor can read register from any peripheral
    - Each register has unique 4-bit address
    - Assume 1 register/periph.
  - Sets \( rd=1, \ A=\text{address} \)
  - Appropriate peripheral places register data on 32-bit \( D \) lines
    - Periph’s address provided on \( Faddr \) inputs (maybe from DIP switches, or another register)
Step 1: Create FSM

Inputs: rd (bit); Q (32 bits); A, Faddr (4 bits)
Outputs: D (32 bits)
Local register: Q1 (32 bits)

• Step 1: Create high-level state machine
  – State **WaitMyAddress**
    • Output “nothing” (“Z”) on D, store peripheral’s register value Q into local register Q1
    • Wait until this peripheral’s address is seen (A=Faddr) and rd=1
  – State **SendData**
    • Output Q1 onto D, wait for rd=0 (meaning main processor is done reading the D lines)
Step 2: Create a datapath

(a) Datapath inputs/outputs
(b) Instantiate declared registers
(c) Instantiate datapath components and connections

Inputs: rd (bit); Q (32 bits); A, Faddr (4 bits)
Outputs: D (32 bits)
Local register: Q1 (32 bits)
Step 3: Connect datapath to controller
Step 4: Derive controller’s FSM

WaitMyAddress

Inputs: rd, A_eq_Faddr (bit)
Outputs: Q1_ld, D_en (bit)

(rd’ and (A_eq_Faddr and rd)’)

D_en = 0
Q1_ld = 1

SendData

A_eq_Faddr
and rd

D_en = 1
Q1_ld = 0

SendData

D = Q1
D_en = 1
Q1_ld = 0

A_eq_Faddr

Datapath

Bus interface

D = \text{"Z"}

Q1 = Q

(A = Faddr) and rd

D_en = 0
Q1_ld = 1

A = Faddr and rd

A_eq_Faddr

Sources: TSR, Katz, Boriello, Vahid, Perkowski
RTL Example: Video Compression

- 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

(a) Digitized frame 1

1 Mbyte

(b) Digitized frame 2

1 Mbyte

Digitized frame 1

1 Mbyte

Digitized frame 2

1 Mbyte

Difference of 2 from 1

0.01 Mbyte

Only difference: ball moving

Just send difference
Video Compression – Sum of Absolute Differences

- If two frames are similar just send a difference instead
  - Compare corresponding 16x16 “blocks”
    - Treat 16x16 block as 256-byte array
  - Compute the absolute value of the difference of each array item
  - Sum the differences
    - if above a threshold, send a complete frame for second frame
    - Else send the difference

Each is a pixel, assume represented as 1 byte (actually, a color picture might have 3 bytes per pixel, for intensity of red, green, and blue components of pixel)
Video Compression – Sum of Absolute Differences

- 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
Step 1: High-level FSM

- **S0**: wait for \( go \)
- **S1**: initialize \( \text{sum} \) and \( \text{index} \)
- **S2**: check if done \( (i \geq 256) \)
- **S3**: add difference to \( \text{sum} \), increment index
- **S4**: done, write to output \( \text{sad\_reg} \)

Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: \( \text{sum}, \text{sad\_reg} \) (32 bits); i (9 bits)
Step 2: Create datapath

Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)

S0
!go

S1
sum = 0
i = 0

S2
i < 256
i = i + 1

S3
sum += abs(A[i] - B[i])

S4
!i < 256 (i < 256)
sad_reg = sum

Datapath

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Step 3: Connect to controller
Step 4: Replace high-level state machine by FSM
Control vs. Data Dominated RTL Design

- **Control-dominated design**
  - Controller contains most of the complexity
- **Data-dominated design**
  - Datapath contains most of the complexity
- Laser-based distance measurer – control dominated
- Bus interface, SAD circuit – mix of control and data
Data Dominated RTL Design Example: FIR Filter

- **FIR filter**
  - “Finite Impulse Response”
  - Simply a configurable weighted sum of past input values
  - \( y(t) = c_0 \cdot x(t) + c_1 \cdot x(t-1) + c_2 \cdot x(t-2) \)
    - Above known as “3 tap”
    - Tens of taps more common
    - Very general filter – User sets the constants (c0, c1, c2) to define specific filter

RTL design
Step 1: Create high-level state machine – there is none
Go straight to step 2
Step 2: Create datapath

- Begin by creating chain of xt registers to hold past values of X
- Instantiate registers for c0, c1, c2
- Instantiate multipliers to compute c*x values
- Instantiate adders
- Add circuitry to allow loading of particular c register

\[ y(t) = c0 \times x(t) + c1 \times x(t-1) + c2 \times x(t-2) \]

Step 3 & 4: Connect to controller, Create FSM
No controller needed
Comparing the FIR circuit to a software implementation

- **Circuit**
  - Adder has 2-gate delay, multiplier has 20-gate delay
  - Longest past goes through one multiplier and two adders
    - 20 + 2 + 2 = 24-gate delay
  - 100-tap filter, would have about a 34-gate delay: 1 multiplier and 7 adders on longest path

- **Software**
  - 100-tap filter: 100 multiplications, 100 additions.
  - If 2 instructions per multiplication, 2 per addition. Say 10-gate delay per instruction.
  - (100*2 + 100*2)*10 = 4000 gate delays

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

- Frequency limited by \textit{longest register-to-register delay}
  - Known as \textit{critical path}
  - If clock is any faster, incorrect data may be stored into register
  - Longest path on right is 2 ns
    - Ignoring wire delays, and register setup and hold times, for simplicity

\[ 5.4 \]

\[ a + b \]

\[ \text{clk} \]

\[ \text{c} \]

\[ 2 \text{ ns delay} \]
Critical Path Considering Wire Delays

- 1980s/1990s: Wire delays tiny compared to logic
- Today wire delay dominates
- Also consider register Tsu & Th

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Example: Critical Path Analysis

- 4 paths:
  - a to c through +: 2 ns
  - a to d through + and *: 7 ns - longest!!!
  - b to d through + and *: 7 ns - longest!!!
  - b to d through *: 5 ns
- Fastest frequency: $1 / 7 \text{ ns} = 142 \text{ MHz}$
A Circuit May Have Numerous Paths

Combinational logic

State register

Datapath

8-bit <
8-bit adder
Simple data encryption/decryption device

- $B = 1$, set offset $O = I[0:31]$
- $B=0$ $e=1$: encrypt mode: output $J = I+O$
- $B=0$ $e=0$: decrypt mode: get $I = J - O$
Reaction timer

- On reset (rst) reaction timer waits for 10 sec before turning on light (len=1)
- Measures the length of time $rtime$ (ms) until user presses button $B$
  - If reaction slower than 2 sec, output slow=1 and rtime=2000
Fast sum of 16 32-bit registers
Hot water detector

- Output warning when average temp over the past 4 samples exceeds a user defined value; $clr$ disables the system
- Inputs (32 bit): $CT$ – current temp; $WT$ – warning temp
- Output: $W$ – high if hot temperature; stays on until $clr$ pressed again
Design from “C” code

Inputs: byte a, byte b, bit go
Outputs: byte gcd, bit done

GCD:
while(1) {
    while(!go);
    done = 0;
    while ( a != b ) {
        if ( a > b ) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }
    gcd = a;
    done = 1;
}
Another “C” example

Inputs: byte a[256], bit go
Outputs: byte max_diff, bit done

MAX_DIFF:
while(1) {
  while(!go);
  done = 0;
  i = 0;
  max = 0;
  min = 255; // largest 8-bit value
  while( i < 256 ) {
    if( a[i] < min ) {
      min = a[i];
    }
    if( a[i] > max ) {
      max = a[i];
    }
    i = i + 1;
  }
  max_diff = max - min;
  done = 1;
Summary

- Datapath and Control Design
- RTL Design Steps
  1. Define the high level state machine
  2. Create datapath
  3. Connect datapath with control
  4. Implement the FSM
- Control vs. Data dominated RTL
- Timing analysis – critical path, processor frequency
Pitfall: Assuming register is updated in the state it’s written

- Value of Q unknown & final state is C, not D; solution: Read register in following state (Q=R)
- Reason:
  - State A: R=99 and Q=R happen simultaneously
  - State B: R not updated with R+1 until next clock cycle, simultaneously with state being updated
**RTL Design Issues**

- Outputs can only be written
- Solution: Introduce additional register, which can be written and read

**Inputs:** A, B (8 bits)  
**Outputs:** P (8 bits)  

- **Diagram (a):**
  - S
  - T
  - P = A
  - P = P + B

- **Diagram (b):**
  - S
  - T
  - R = A
  - P = A
  - P = R + B

**Notes:**
- Local register: R (8 bits)