ECEN 468
Advanced Digital System Design

Lecture 17: RTL Design

Instructor: J. Hu
High-Level State Machines (HLSMs)

- Some behaviors too complex for equations, truth tables, or FSMs
  - Ex: 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
  - FSM can’t represent...
    - 8-bit input/output
    - Storage of current total
    - Addition (e.g., 25 + 10)

![Diagram of soda dispenser](image)
HLSMs

- High-level state machine (HLSM) extends FSM with:
  - Multi-bit input/output
  - Local storage
  - Arithmetic operations

- Conventions
  - Numbers:
    - Single-bit: '0' (single quotes)
    - Integer: 0 (no quotes)
    - Multi-bit: “0000” (double quotes)
  - == for equal, := for assignment
  - Multi-bit outputs must be registered via local storage
  - // precedes a comment
**Ex: Cycles-High Counter**

- \( P = \) total number (in binary) of cycles that \( m \) is 1
- Capture behavior as HLSM
  - Preg required (multibit outputs must be registered)
    - Use to hold count

**Inputs:** \( m \) (bit)  
**Outputs:** \( P \) (32 bits)  
**Local storage:** Preg

\( \text{// Clear Preg to 0s} \)  
\( \text{Preg} := 0 \)

\( \text{// Wait for } m = '1' \)  
\( \text{Preg} := \text{Preg} + 1 \)

Note: Could have designed directly using an up-counter. But, that methodology is ad hoc, and won't work for more complex examples, like the next one.
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 $D = \frac{T \text{ sec} \times 3 \times 10^8 \text{ m/sec}}{2}$
Example: Laser-Based Distance Measurer

- **Inputs/outputs**
  - \(B\): bit input, from button, to begin measurement
  - \(L\): bit output, activates laser
  - \(S\): bit input, senses laser reflection
  - \(D\): 16-bit output, to display computed distance
Example: Laser-Based Distance Measurer

- Declare inputs, outputs, and local storage
  - Dreg required for multi-bit output
- Create initial state, name it $S_0$
  - Initialize laser to off ($L:='0'$)
  - Initialize displayed distance to 0 ($Dreg:=0$)

Recall: '0' means single bit, 0 means integer
Example: Laser-Based Distance Measurer

- Add another state, **S1**, that waits for a button press
  - **B'** – stay in **S1**, keep waiting
  - **B** – go to a new state **S2**

Q: What should **S2** do?   A: Turn on the laser
Example: Laser-Based Distance Measurer

- Add a state \( S_2 \) that turns on the laser (\( L:'1' \))
- Then turn off laser (\( L:'0' \)) in a state \( S_3 \)

Q: What do next?  A: Start timer, wait to sense reflection
Example: Laser-Based Distance Measurer

- Stay in **S3** until sense reflection (S)
- To measure time, count cycles while in **S3**
  - To count, declare local storage **Dctr**
  - Initialize **Dctr** to 0 in **S1**. In **S2** would have been O.K. too.
    - Don't forget to initialize local storage—common mistake
  - Increment **Dctr** each cycle in **S3**
Example: Laser-Based Distance Measurer

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

*Local storage:* Dreg, Dctr (16 bits)

- Once reflection detected (S), go to new state **S4**
  - Calculate distance
  - Assuming clock frequency is $3 \times 10^8$, $Dctr$ holds number of meters, so $Dreg := Dctr/2$

- After **S4**, go back to **S1** to wait for button again
**HLSM Actions: Updates Occur Next Clock Cycle**

- Local storage updated on clock edges only
  - Enter state on clock edge
  - Storage writes in that state occur on *next* clock edge
  - Can think of as occurring on outgoing transitions
- *Thus*, transition conditions use the OLD value, not the newly-written value
  - Example:

  **(a)**
  - **Inputs:** $B$ (bit)
  - **Outputs:** $P$ (bit) // if $B$, 2 cycles high
  - **Local storage:** $Jreg$ (8 bits)

<table>
<thead>
<tr>
<th>State</th>
<th>$P$</th>
<th>$Jreg$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S0$</td>
<td>'0'</td>
<td>1</td>
</tr>
<tr>
<td>$S1$</td>
<td>'1'</td>
<td>$Jreg + 1$</td>
</tr>
</tbody>
</table>

  $B'$, !(Jreg<2), Jreg<2

<table>
<thead>
<tr>
<th>Transition</th>
<th>$B'$</th>
<th>!(Jreg&lt;2)</th>
<th>Jreg&lt;2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S0$ to $S1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S1$ to $S0$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  **(b)**
  - **Clock**
  - **B**
  - **Jreg**
  - **P**

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>$B$</th>
<th>$Jreg$</th>
<th>$P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>S0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**RTL Design Process**

- **Capture** behavior
- **Convert** to circuit
  - Need target architecture
  - Datapath capable of HLSM's data operations
  - Controller to control datapath
Ctrl/DP Example for Earlier Cycles-High Counter

First clear Preg to 0s
Then increment Preg for each clock cycle that \( m \) is 1

\[
\text{inputs: } m \text{ (bit)} \\
\text{outputs: } P \text{ (32 bits)} \\
\text{locstr: } \text{Preg} \text{ (32 bits)}
\]

We created this HLSM earlier

CountHigh

We created this HLSM earlier

Controller

Create DP
Connect with controller
Derive controller
# RTL Design Process

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1: Capture behavior</strong></td>
<td></td>
</tr>
<tr>
<td><em>Capture a high-level state machine</em></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 single-bit inputs and outputs.</td>
</tr>
<tr>
<td><strong>Step 2: Convert to circuit</strong></td>
<td></td>
</tr>
<tr>
<td><strong>2A Create a datapath</strong></td>
<td>Create a datapath to carry out the data operations of the high-level state machine.</td>
</tr>
<tr>
<td><strong>2B Connect the datapath to a controller</strong></td>
<td>Connect the datapath to a controller block. Connect external control inputs and outputs to the controller block.</td>
</tr>
<tr>
<td><strong>2C Derive the controller’s FSM</strong></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>
Example: Soda Dispenser

- Quick overview example.
  More details of each step to come.

**Inputs:** c (bit), a (8 bits), s (8 bits)
**Outputs:** d (bit)  // '1' dispenses soda

**Local storage:** tot (8 bits)

---

**SodaDispenser**

**Step 1**

- Init
  - d:= '0'
  - tot:= 0
- Wait
  - c'*(tot<s)
  - tot:= tot + a
  - d:= '0'
- Disp
  - c

---

**Controller**

**Datapath**

**Step 2A**

- tot ld
- tot clr
- tot lt s
- ld
- clr
- tot

- 8-bit <
- 8-bit adder

**Step 2B**

- c
- d
- tot ld
- tot clr
- tot lt s

---

**SodaDispenser**

**Step 1**

- Init
  - d:= '0'
  - tot:= 0
- Wait
  - c'*(tot<s)
  - tot:= tot + a
  - d:= '0'
- Disp
  - c

---

**Controller**

**Datapath**

**Step 2A**

- tot ld
- tot clr
- tot lt s

- 8-bit <
- 8-bit adder

**Step 2B**

- c
- d
- tot ld
- tot clr
- tot lt s
Example: Soda Dispenser

• Quick overview example. More details of each step to come.

**Inputs:** c (bit), a (8 bits), s (8 bits)
**Outputs:** d (bit) // '1' dispenses soda
**Local storage:** tot (8 bits)

**SodaDispenser**

**Step 1**
- **Init**
  - d := '0'
  - tot := 0

- **Disp**
  - d := '1'

- **Wait**
  - tot := tot + a
  - c' * (tot < s)

**Step 2B**

**Controller**

**Datapath**

**Step 2C**

**Controller**

**Datapath**

**SodaDispenser**

- **Add**
  - tot ld = 1
  - tot clr = 1

- **Disp**
  - d = 0
  - tot ld = 1

- **Wait**
  - c' * tot lt s'

- **Init**
  - d = 0
  - tot clr = 1
Example: Soda Dispenser

- Quick overview example.
  More details of each step to come.

Use controller (Finite State Machine) design process to complete the design
RTL Design Process—Step 2A: Create a datapath

- **Sub-steps**
  - HLSM data inputs/outputs $\rightarrow$ Datapath inputs/outputs.
  - HLSM local storage item $\rightarrow$ Instantiated register
    - "Instantiate": Add new component ("instance") to design
  - Each HLSM state action and transition condition data computation $\rightarrow$ Datapath components and connections
    - Also instantiate multiplexors as needed

- **Need component library from which to choose**

```
\text{clock}$^{\uparrow}$ and $\text{clr}=1$: $Q=0$
\text{clock}$^{\uparrow}$ and $\text{ld}=1$: $Q=I$
else $Q$ stays same

S = A+B

\text{(unsigned)}
A<B: \text{lt}=1
A=B: \text{eq}=1
A>B: \text{gt}=1

\text{shiftL1}: \ll<1
\text{shiftL2}: \ll<2
\text{shiftR1}: \gg>1
...

s0=0: $Q=I0$
s0=1: $Q=I1$
```

\[ A \quad \text{add} \quad S \]
\[ A \quad \text{cmp} \quad \text{lt} \quad \text{eq} \quad \text{gt} \]
\[ I \quad \text{shift}<L/R> \quad Q \]
\[ \text{mux2x1} \]

\[ I1 \quad I0 \quad s0 \quad Q \]
Step 2A: Create a Datapath—Simple Examples

(a) \( P\text{reg} = X + Y + Z \)

(b) \( P\text{reg} = P\text{reg} + X \)

(c) \( P\text{reg} = X + Y; \quad \text{reg}Q = Y + Z \)

(d) \( k = 0: P\text{reg} = Y + Z \quad k = 1: P\text{reg} = X + Y \)

\( X \quad Y \quad Z \)

\[ \begin{array}{c}
0 \quad 1 \\
\text{clr} \quad \text{ld} P\text{reg} \\
\text{Q} \\
\end{array} \]

\[ \begin{array}{c}
0 \quad 1 \\
\text{clr} \quad \text{ld} \text{regQ} \\
\text{Q} \\
\end{array} \]
Laser-Based Distance Measurer—Step 2A: Create a Datapath

- HLSM data I/O → DP I/O
- HLSM local storage → reg
- HLSM state action and transition condition data computation → Datapath components and connections

DistanceMeasurer

*Inputs*: B (bit), S (bit)  
*Outputs*: L (bit), D (16 bits)  
*Local storage*: Dreg, Dctr (16 bits)

\[
\begin{align*}
B' \\
S' \\
S \rightarrow S_1 \rightarrow B \rightarrow S_2 \rightarrow L \rightarrow S_3 \rightarrow S_4 \\
L := '0' \\
Dctr := 0 \\
L := '1' \\
Dctr := Dctr + 1 \\
Dreg := Dctr/2 \\
// calculate D
\end{align*}
\]

Datapath

\[
\begin{align*}
\text{Add1: add(16)} \\
\text{Shr1: shiftR1(16)} \\
\text{Dreg: reg(16)} \\
\text{Dctr: reg(16)}
\end{align*}
\]
Laser-Based Distance Measurer—Step 2B: Connecting the Datapath to a Controller

Controller

Datapath

300 MHz Clock

from button B

to display D

L to laser

S from sensor

Dreg_clr

Dreg_ld

Dctr_clr

Dctr_ld

16
Laser-Based Distance Measurer—Step 2C: Derive the Controller FSM

- FSM has same states, transitions, and control I/O
- Achieve each HLSM data operation using datapath control signals in FSM
Laser-Based Distance Measurer—Step 2C: Derive the Controller FSM

- Same FSM, using convention of unassigned outputs implicitly assigned 0

Some assignments to 0 still shown, due to their importance in understanding desired controller behavior
5.4 More RTL Design

- Additional datapath components

\[ S = A - B \quad \text{(signed)} \]
\[ P = A \times B \quad \text{(unsigned)} \]
\[ Q = |A| \quad \text{(unsigned)} \]

\[ S = A - B \quad \text{(signed)} \]
\[ P = A \times B \quad \text{(unsigned)} \]
\[ Q = |A| \quad \text{(unsigned)} \]

- Additional RTL Design Components:
  - Subtraction (`sub`):
    - \[ S = A - B \; \text{(signed)} \]
  - Multiplication (`mul`):
    - \[ P = A \times B \; \text{(unsigned)} \]
  - Absolute Value (`abs`):
    - \[ Q = |A| \; \text{(unsigned)} \]

- Control Signals:
  - `clr`:
    - \[ Q = 0 \] (when `clk^` and `clr=1`)
  - `inc`:
    - \[ Q = Q + 1 \] (when `clk^` and `inc=1`)
  - `upcnt`:
    - \[ Q \] stays same (otherwise)

- Register File (RF):
  - Write Enable (`W_e`):
    - \[ RF[W_a] = W_d \] (when `clk^` and `W_e=1`)
  - Read Enable (`R_e`):
    - \[ R_d = RF[R_a] \] (when `R_e=1`)

- Additional Signals:
  - `W_d`, `W_a`, `W_e`, `R_a`, `R_e`, `R_d`
RTL Design Involving Register File or Memory

- HLSM array: Ordered list of items
  - Ex: Local storage: A[4](8-bit) – 4 8-bit items
  - Accessed using notation "A[i]", i is index
    - Array contents now: <9, 8, 7, 22>
    - X := A[1] will set X to 8
    - Note: First element's index is 0

- Array can be mapped to instantiated register file or memory
Simple Array Example

**ArrayEx**

Inputs: (none)
Outputs: A (11 bits)
Local storage: A[4](11 bits)

(a)

**Controller**

Init1
- Preg := 0
- A[0] := 9
- A[0] == 8'
Init2
- A[0] == 8
Out1
- Preg := A[1]

(c)

ArrayEx

Inputs: A_eq_8
Outputs: A_s, A_Wa0, ...

Init1
- Preg_clr = 1
- A_s = 0
- A_Wa0 = 0, A_Wa1 = 0
- A_We = 1
- (A_eq_8)'
Init2
- A_eq_8 = 1
- A_Wa0 = 0, A_Wa1 = 1
- A_We = 1
- A_Ra1 = 0, A_Ra0 = 0
- A_Re = 1
Out1
- Preg_ld = 1

(b)
RTL Example: Video Compression – Sum of Absolute Differences

- 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 1

Digitized frame 2

Frame 1

Frame 2

Difference of 2 from 1

0.01 Mbyte

Just send difference

Only difference: ball moving
RTL Example: Video Compression – Sum of Absolute Differences

• 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)

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)
Array Example: Video Compression—Sum-of-Absolute Differences

**Inputs:** A, B [256](8 bits); go (bit)
**Outputs:** sad (32 bits)
**Local storage:** sum, sadreg (32 bits); i (9 bits)

- **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
Array Example: Video Compression—Sum-of-Absolute Differences

Inputs: A, B [256](8 bits); go (bit)
Outputs: sad (32 bits)
Local storage: sum, sadreg (32 bits); i (9 bits)

```
sum := 0
i := 0
```

```
i < 256
sum := sum + abs(A[i] - B[i])
i := i + 1
```

```
sadreg := sum
```

Controller

Datapath
Circuit vs. Microprocessor

- Circuit: Two states (S2 & S3) for each i, 256 i’s → 512 clock cycles
- Microprocessor: 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 6 cycles per array item → 256*6 = 1536 cycles
- Circuit is about 3 times (300%) faster (assuming equal cycle lengths)
Data Dominated RTL Design Example

- Data dominated design: Extensive DP, simple controller
- Control dominated design: Complex controller, simple DP
- Example: Filter
  - Converts digital input stream to new digital output stream
  - Ex: Remove noise
    - 180, 180, 181, 180, 240, 180, 181
    - 240 is probably noise, filter might replace by 181
  - Simple filter: Output average of last $N$ values
    - Small $N$: less filtering
    - Large $N$: more filtering, but less sharp output
Data Dominated RTL Design Example: FIR Filter

- **FIR filter**
  - “Finite Impulse Response”
  - Simply a configurable weighted sum of past input values
  - \( y(t) = c_0 x(t) + c_1 x(t-1) + c_2 x(t-2) \)
    - 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 HLSM
    - Very simple states/transitions

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

Inputs: \( X \) (12 bits) Outputs: \( Y \) (12 bits)
Local storage: \( xt_0, xt_1, xt_2, c_0, c_1, c_2 \) (12 bits); Yreg (12 bits)

\[
\begin{align*}
  &\text{Init} &\rightarrow &\text{FC} \\
  &\text{Yreg} := 0 &\rightarrow &\text{c}_0 \cdot \text{xt}_0 + \\
  &\text{xt}_0 := 0 &\rightarrow &\text{c}_1 \cdot \text{xt}_1 + \\
  &\text{xt}_1 := 0 &\rightarrow &\text{c}_2 \cdot \text{xt}_2 \\
  &\text{xt}_2 := 0 &\rightarrow &\text{c}_0 := 3 \text{ \ \ xt}_0 := X \\
  &\text{c}_0 := 3 &\rightarrow &\text{c}_1 := \text{xt}_0 \\
  &\text{c}_1 := 2 &\rightarrow &\text{c}_2 := \text{xt}_1 \\
  &\text{c}_2 := 2 &\rightarrow &\text{Yreg} := Y
\end{align*}
\]

Assume constants set to 3, 2, and 2
FIR Filter

- Step 2A: Create datapath
- Step 2B: Connect Ctrlr/DP (as earlier examples)
- Step 2C: Derive FSM
  - Set clr and ld lines appropriately

Inputs: X (12 bits)  Outputs: Y (12 bits)
Local storage: xt0, xt1, xt2, c0, c1, c2 (12 bits);
  Yreg (12 bits)

Datapath for 3-tap FIR filter
Circuit vs. Microprocessor

- Comparing the FIR circuit to microprocessor instructions
  - Microprocessor
    - 100-tap filter: 100 multiplications, 100 additions. Say 2 instructions per multiplication, 2 per addition. Say 10 ns per instruction.
    - \((100*2 + 100*2)*10 = 4000\) ns
  - Circuit
    - Assume adder has 2 ns delay, multiplier has 20 ns delay
    - Longest path goes through one multiplier and two adders
      - \(20 + 2 + 2 = 24\) ns delay
    - 100-tap filter, following design on previous slide, would have about a 34 ns delay: 1 multiplier and 7 adders on longest path
  - Circuit is more than 100 times faster \((4000/34)\). Wow.
Determining Clock Frequency

- Designers of digital circuits often want fastest performance
  - Means want high clock frequency
- Frequency limited by *longest register-to-register delay*
  - Known as *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
Critical Path

- Example shows four paths
  - a to c through +: 2 ns
  - a to d through + and *: 7 ns
  - b to d through + and *: 7 ns
  - b to d through *: 5 ns
- Longest path is thus 7 ns
- Fastest frequency
  - $1 / 7 \text{ ns} = 142 \text{ MHz}$
Critical Path Considering Wire Delays

- Real wires have delay too
  - Must include in critical path
- Example shows two paths
  - Each is $0.5 + 2 + 0.5 = 3$ ns
- Trend
  - 1980s/1990s: Wire delays were tiny compared to logic delays
  - But wire delays not shrinking as fast as logic delays
    - Wire delays may even be greater than logic delays!
- Must also consider register setup and hold times, also add to path
- Then add some time to the computed path, just to be safe
  - e.g., if path is 3 ns, say 4 ns instead
A Circuit May Have Numerous Paths

- Paths can exist
  - In the datapath
  - In the controller
  - Between the controller and datapath
  - May be hundreds or thousands of paths

- Timing analysis tools that evaluate all possible paths automatically very helpful