Where we are now…

• What we have covered last time:
  – RTL introduction: 1st half of Zybook chap 6

• What we are covering today:
  – RTL design cont.; 2nd half of Zybook chap 6

• CAPEs are out!!! [https://cape.ucsd.edu/students/](https://cape.ucsd.edu/students/)
  – Your feedback is very important, please take the time to fill out the survey. I read all your feedback carefully and use it to guide the design of future courses. 😊
  – If response rate > 85%, I will drop the lowest quiz grade!

• Deadlines:
  – Quiz #6 today!!!
  – HW#6 due next Tuesday – the last HW!!!
  – Exam#3 during finals week – the last exam!!!
    • 80 minutes long
    • Comprehensive
    • Bring one 8 ½ x 11” paper with handwritten notes, but nothing else
## 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><strong>Capture a high-level state machine</strong></td>
</tr>
<tr>
<td><strong>Step 2: Convert to circuit</strong></td>
<td><strong>Create a datapath</strong></td>
</tr>
<tr>
<td>2A</td>
<td>Create a datapath to carry out the data operations of the high-level state machine.</td>
</tr>
<tr>
<td>2B</td>
<td><strong>Connect the datapath to a controller</strong></td>
</tr>
<tr>
<td>Connect the datapath to a controller</td>
<td>Connect the datapath to a controller block. Connect external control inputs and outputs to the controller block.</td>
</tr>
<tr>
<td>2C</td>
<td><strong>Derive the controller’s FSM</strong></td>
</tr>
<tr>
<td>Derive the controller’s FSM</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 Design Process: Datapath components

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

```
<table>
<thead>
<tr>
<th>clk^ and clr=1: Q=0</th>
<th>A B</th>
<th>A B</th>
<th>I</th>
<th>I1 I0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S = A+B</td>
<td>add</td>
<td>cmp</td>
<td>shiftL1: &lt;&lt;1</td>
<td>mux2x1</td>
</tr>
<tr>
<td>(unsigned)</td>
<td></td>
<td>lt</td>
<td>shiftL2: &lt;&lt;2</td>
<td></td>
</tr>
<tr>
<td>A&lt;B: lt=1</td>
<td></td>
<td>eq</td>
<td>shiftR1: &gt;&gt;1</td>
<td></td>
</tr>
<tr>
<td>A=B: eq=1</td>
<td></td>
<td>gt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A&gt;B: gt=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else Q stays same</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Sources: TSR, Katz, Boriello, Vahid, Perkowski
More RTL Design Components

\[
S = A - B \\
(\text{signed})
\]

\[
P = A \times B \\
(\text{unsigned})
\]

\[
Q = |A| \\
(\text{unsigned})
\]

\[
\text{clr}^\uparrow \; \text{and} \; \text{clr}=1: Q=0
\]

\[
\text{clk}^\uparrow \; \text{and} \; \text{inc}=1: Q=Q+1
\]

\[
\text{else} \; Q \; \text{stays same}
\]

\[
P = A \times B \\
(\text{unsigned})
\]

\[
Q = |A| \\
(\text{unsigned})
\]

\[
\text{clk}^\uparrow \; \text{and} \; \text{W}_e=1:
\]

\[
RF[\text{W}_a]= \text{W}_d
\]

\[
\text{R}_e=1:
\]

\[
\text{R}_d = RF[\text{R}_a]
\]

Sources: TSR, Katz, Boriello, Vahid, Perkowski
More Datapath Examples

(a) \[ Preg = X + Y + Z \]

(b) \[ Preg = Preg + X \]

(c) \[ Preg = X + Y; \quad regQ = Y + Z \]

(d) \[ k=0: Preg = Y + Z; \quad k=1: Preg = X + Y \]
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

(a) ArrayEx
Inputs: (none)
Outputs: P (11 bits)
Local storage: A[4](11 bits)
Preg (11 bits)

Init1
Preg := 0
A[0] := 9

(A[0] == 8)

Init2
A[0] == 8

Out1

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

Init1
Preg_clr = 1
A_s = 0
A_Wa1=0, A_Wa1=0
A_We = 1
(A_eq_8)'

Init2
A_s = 1
A_Wa1=0, A_Wa0=1
A_We = 1
A_Ra1=0, A_Ra0=0
A_Re = 1

Out1
Preg_Id = 1

Controller

(b) ArrayEx
Inputs: A_eq_8
Outputs: A_s, A_Wa0, ...

Preg_clr = 1
A_s = 0
A_Wa1=0, A_Wa1=0
A_We = 1
(A_eq_8)'

Init1
A_s = 1
A_Wa1=0, A_Wa0=1
A_We = 1
A_Ra1=0, A_Ra0=0
A_Re = 1

Out1
Preg_Id = 1

(c) ArrayEx
Inputs: A_eq_8
Outputs: A_s, A_Wa0, ...

Preg_clr = 1
A_s = 0
A_Wa1=0, A_Wa1=0
A_We = 1
(A_eq_8)'

Init1
A_s = 1
A_Wa1=0, A_Wa0=1
A_We = 1
A_Ra1=0, A_Ra0=0
A_Re = 1

Out1
Preg_Id = 1

Controller

Resources: TSR, Katz, Boriello, Vahid, Perkowski
**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)
Bus Interface: Create HLSM

- 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 ($A=\text{Faddr}$) and $rd=1$
  - **State SendData**
    - Output $Q1$ onto $D$, wait for $rd=0$ (meaning main processor is done reading the $D$ lines)

Inputs: $rd$ (bit); $Q$ (32 bits); $A$, $\text{Faddr}$ (4 bits)
Outputs: $D$ (32 bits)
Local register: $Q1$ (32 bits)
Bus Interface: Create a datapath

- **Steps for creating datapath:**
  1. Define datapath inputs/outputs
  2. Instantiate declared registers
  3. Instantiate datapath components and connections

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

- \( D = \text{"Z"} \)
- \( Q_{1} = Q \)

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Bus Interface: Connect datapath to controller & derive controller’s FSM

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

**Digitized frame 1**

**Digitized frame 2**

(a)

**Difference of 2 from 1**

(b)

1 Mbyte

1 Mbyte

0.01 Mbyte

Only difference: ball moving

Just send difference

Sources: TSR, Katz, Boriello, Vahid, Perkowski
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)
Sum-of-Absolute Differences: High-level FSM

- **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*

Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)
Sum-of-Absolute Differences: Datapath

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

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

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Sum-of-Absolute Differences: Connect datapath and controller & specify the controller FSM

\[
\text{sum} = 0 \quad \text{sum}_\text{clr} = 1 \\
i = 0 \quad i_\text{clr} = 1 \\
\text{if } i < 256 \text{ then } \text{go' else go} \\
\text{sum} = \text{sum} + \text{abs}(A[i] - B[i]) \\
i = i + 1 \quad i_\text{inc} = 1 \\
\text{sad}_\text{reg} = \text{sum} \quad \text{sad}_\text{reg}_\text{ld} = 1 \\
\text{Controller}
\]
Behavioral Level Design: C to Gates

• Earlier sum-of-absolute-differences example
  – Started with high-level state machine
  – C code is an even better starting point -- easier to understand

```c
int SAD (byte A[256], byte B[256]) // not quite C syntax
{
    uint sum; short uint I;
    sum = 0;
    i = 0;
    while (i < 256) {
        sum = sum + abs(A[i] - B[i]);
        i = i + 1;
    }
    return sum;
}
```

**Inputs:** A, B [256](8 bits); go (bit)
**Outputs:** sad (32 bits)
**Local storage:** sum, sadreg (32 bits); i (9 bits)
Converting from C to High-Level State Machine

- Convert each C construct to equivalent states and transitions
- **Assignment** statement
  - Becomes one state with assignment
- **If-then** statement
  - Becomes state with condition check, transitioning to “then” statements if condition true, otherwise to ending state
    - “then” statements would also be converted to states

```c
    target = expression;
    if (cond) {
        // then stmts
    }
```

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Converting from C to High-Level State Machine

**If-then-else**
- Becomes state with condition check, transitioning to “then” statements if condition true, or to “else” statements if condition false

**While loop statement**
- Becomes state with condition check, transitioning to while loop’s statements if true, then transitioning back to condition check
Converting from C to HLSM: Example

- Simple example: computing the maximum of two numbers
  - Convert if-then-else statement to states (b)
  - Then convert assignment statements to states (c)

Inputs: uint X, Y
Outputs: uint Max

```c
if (X > Y) {
    Max = X;
} else {
    Max = Y;
}
```

(a)

(b)

(c)
Example: SAD C code to HLSM

- Convert each construct to states
  - Simplify states
- Use RTL design process to convert to circuit
- However, only a subset of C can be easily converted
  - Can use language other than C