CSE140: Components and Design Techniques for Digital Systems

Register Transfer Level (RTL) Design

Tajana Simunic Rosing
Welcome to CSE 140!

- **What we covered last time:**
  - Timing constraints and FSMs, state assignment & partitioning, RTL
- **Where we are going today:**
  - RTL
- **Deadlines:**
  - HW5 due today, HW6 assigned
  - Zybook due by the beginning of the class
- **Regrades:**
  - Grades posted at the latest on Thursdays, regrade requests are due the following Thursday; updated grades posted on TED by the weekend
  - Regrade request means that your work will be thoroughly reviewed and a grade may change up or down
- **Discussion session:**
  - This Friday at 2pm!

Sources: TSR, Katz, Boriello, Vahid, Perkowski
# RTL Design Process

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1:</strong> Capture behavior</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 single-bit inputs and outputs.</td>
</tr>
<tr>
<td><strong>Step 2:</strong> Convert to circuit</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></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></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>

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Remember the Soda HLSM?

Inputs: c (bit), a(8 bits), s (8 bits)
Outputs: d (bit)
Local registers: tot (8 bits)

Controller

Add
Disp
Init
Wait

8-bit adder
Data path

d=0

tot= tot+a

tot= tot+a

tot= tot+a

Mealy or Moore

A) Me
B) Mo
C) No

Sources: TSR, Katz, Boriello, Vahid, Perkowski
RTL Delay: A Circuit May Have Numerous Paths

- Frequency is limited by **longest register-to-register delay**
  - **critical path**
- Paths can exist
  - In the datapath
  - In the controller
  - Between the controller and datapath
  - May be hundreds or thousands of paths

\[ T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew} \]
RTL Design Process: Datapath components

- **Adder**: $S = A + B$
  - \( \text{clk}\uparrow \) and \( \text{clr}=1 \): \( Q=0 \)
  - \( \text{clk}\uparrow \) and \( \text{ld}=1 \): \( Q=I \)
  - else \( Q \) stays same

- **Comparer**: $A < B$: \( \text{lt}=1 \)
  - $A = B$: \( \text{eq}=1 \)
  - $A > B$: \( \text{gt}=1 \)

- **Shift Register**
  - \( s_0=0 \): \( Q=I_0 \)
  - \( s_0=1 \): \( Q=I_1 \)
  - \( \text{clk}\uparrow \) and \( \text{clr}=1 \): \( Q=0 \)
  - \( \text{clk}\uparrow \) and \( \text{ld}=1 \): \( Q=I \)
  - else \( Q \) stays same

- **Subtractor**: $S = A - B$
  - \( \text{signed} \)

- **Multiplier**: $P = A \times B$
  - \( \text{unsigned} \)

- **Absolute Value**: $Q = |A|$
  - \( \text{unsigned} \)

- **Control Unit**
  - \( \text{clk}\uparrow \) and \( \text{clr}=1 \): \( Q=0 \)
  - \( \text{clk}\uparrow \) and \( \text{inc}=1 \): \( Q=Q+1 \)
  - else \( Q \) stays same

- **Register File**
  - \( \text{clk}\uparrow \) and \( \text{W}_e=1 \): \( RF[W_a]=W_d \)
  - \( R_e=1 \): \( R_d = RF[R_a] \)

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

(a) $P_{reg} = X + Y + Z$

(b) $P_{reg} = P_{reg} + X$

(c) $P_{reg} = X + Y$; $regQ = Y + Z$

(d) $k=0$: $P_{reg} = Y + Z$
    $k=1$: $P_{reg} = X + Y$

Sources: TSR, Katz, Boriello, Vahid, Perkowski
HLSM Array Example

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

1. Preg := 0
2. A[0] := 9
3. (A[0] == 8)'
5. A[0] == 8

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

1. Preg_clr = 1
2. A_s = 0
3. A_Wa1 = 0, A_Wa1 = 0
4. A_We = 1
5. (A_eq_8)'
6. A_s = 1
7. A_Wa1 = 0, A_Wa0 = 1
8. A_We = 1
9. A_Ra1 = 0, A_Ra0 = 0
10. A_Re = 1

(c) Controller
- **Controller**
- **Inputs:** A_eq_8
- **Outputs:** A_s, A_Wa, ...

1. Preg_clr = 1
2. A_s = 0
3. A_Wa1 = 0, A_Wa1 = 0
4. A_We = 1
5. (A_eq_8)'
6. A_s = 1
7. A_Wa1 = 0, A_Wa0 = 1
8. A_We = 1
9. A_Ra1 = 0, A_Ra0 = 0
10. A_Re = 1

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

![Video Compression Diagram](image)

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Video Compression – Sum of Absolute Differences

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)

• 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
Sum-of-Absolute Differences: High-level FSM

Sources: TSR, Katz, Boriello, Vahid, Perkowski

**S0**: wait for go
**S1**: initialize sum and index
**S2**: check if done ($i \geq 256$)
**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)

S0: !go, go
S1: go, sum = 0, i = 0
S2: !(i<256), sum = sum + abs(A[i]-B[i]), i = i+1
S3: !(!(i<256) & i<256), sad_reg = sum
S4: !(i<256) & i<256

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

\[ \text{S0: go' } \]
\[ \text{S1: } \text{sum=0; sum_clr=1; } i=0; \text{ i_clr=1 } \]
\[ \text{S2: } i < 256; \text{ i_lt_256 } \]
\[ \text{S3: } \text{sum=\sum_{i=0}^{255} |A[i] - B[i]|; } \text{sum_ld=1; AB_rd=1; } i=i+1; \text{ i_inc=1 } \]
\[ \text{S4: sad_reg=sum; sad_reg_ld=1 } \]

Controller

Sources: TSR, Katz, Boriello, Vahid, Perkowski
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

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

Sources: TSR, Katz, Boriello, Vahid, Perkowski
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
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

```c
if (cond) {
    // then stmts
} else {
    // else stmts
}
```

• **While loop** statement
  – Becomes state with condition check, transitioning to while loop’s statements if true, then transitioning back to condition check

```c
while (cond) {
    // while stmts
}
```
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

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

Sources: TSR, Katz, Boriello, Vahid, Perkowski
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

Inputs: byte A[256], B[256]
Output: int sad
main()
{
  uint sum; short uint i;
  while (1) {
    while (!go);
    sum = 0;
    i = 0;
    while (i < 256) {
      sum = sum + abs(A[i] – B[i]);
      i = i + 1;
    }
    sad = sum;
  }
}
Circuit vs. Microprocessor

• **Circuit:**
  – Two states (S2 & S3) for each $i$, 256 $i$'s $\rightarrow$ 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 $i$
  – Each loop iteration is approx 6 cycles per array item $\rightarrow$ 256*6 = 1536 cycles

• **Circuit is ~3 times (300%) faster**

• It is possible to build a circuit that is much faster than this
  – think about how to leverage parallelism in HW!

```plaintext
S2

S3

(i<256)

(i<256)

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

S2

S3
```
Data vs. Control Dominated RTL Design

- Data dominant design: extensive datapath, simple controller
- Control dominant design: complex controller, simple datapath

Example: Filter
Converts digital input stream to new digital output stream
- e.g: remove noise
  - 180, 180, 181, 180, 240, 180, 181
  - 240 is probably noise, filter might replace by 181
- Simple filter: output average of the 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”
  - A configurable weighted sum of past input values
  - \( y(t) = c_0 \times x(t) + c_1 \times x(t-1) + c_2 \times x(t-2) \)
    - 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 a specific filter

- **RTL design**
  - Step 1: Create HLSM
    - Very simple states/transitions

\[
\begin{align*}
  y(t) &= c_0 \times x(t) + c_1 \times x(t-1) + c_2 \times x(t-2) \\
  \text{Assumes constants set to 3, 2, and 2}
\end{align*}
\]
FIR Filter: Create datapath

Step 2: Create datapath
- Create a 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 c register

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

Steps 3 & 4:
Connect to controller, &
create FSM:
No controller needed!
FIR Filter: Design the Circuit

- Create datapath
- Connect control and datapath
- Derive FSM
  - Set clr and ld lines appropriately

Datapath for 3-tap FIR filter

Inputs: X (12 bits)  Outputs: Y (12 bits)

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

Yreg := 0
xt0 := 0
xt1 := 0
xt2 := 0
c0 := 3
c1 := 2
c2 := 2

xt0 := X
xt1 := xt0
xt2 := xt1

Yreg := c0*xt0 + c1*xt1 + c2*xt2

Sources: TSR, Katz, Boriello, Vahid, Perkowski
Comparing the FIR circuit to a software implementation

**Circuit**
- 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 has 34-gate delay**: 1 multiplier & 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.
- $(100 \times 2 + 100 \times 2) \times 10 = 4000$ gate delays

$$y(t) = c_0 x(t) + c_1 x(t-1) + c_2 x(t-2)$$
Another RTL Design Example: Bus Interface

- Master processor can read register from any peripheral
  - Each register has unique 4-bit address
  - Assume 1 register/peripheral
- Sets $rd=1$, $A=address$
- Appropriate peripheral places register data on 32-bit $D$ lines
  - Peripheral’s own address given on $Faddr$ inputs
Bus Interface: Create a datapath

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

D = “Z”
Q1 = Q

Steps for creating datapath:
1. Define its inputs/outputs
2. Instantiate registers
3. Instantiate components
4. Connect components

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

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

- D_en = 0, Q1_ld = 1
- (A_eq_Faddr and rd)
- D_en = 1, Q1_ld = 0

Sources: TSR, Katz, Boriello, Vahid, Perkowski