CDA 4253/CIS 6930 FPGA System Design
RTL Design Methodology

Hao Zheng
Comp Sci & Eng
Univ of South Florida
Structure of a Typical Digital Design

Datapath (Execution Unit)

Data Inputs

Control Signals

Control (Control Unit)

Control Inputs

Data Outputs

Status Signals

Control Outputs
Hardware Design with RTL VHDL

![Diagram of hardware design with RTL VHDL](image)
Steps of the Design Process

1. Text description
2. Define interface
3. Describe the functionality using pseudo-code
4. Convert pseudo-code to FSM in state diagram
   1. Define states and state transitions
   2. Define datapath operations in each state.
5. Develop VHDL code to implement FSM
6. Develop testbench for simulation and debugging
7. Implementation and timing simulation
   • Timing simulation can reveal more bugs than pre-synthesis simulation
8. Test the implementation on FPGA boards
Min_Max_Average
Pseudocode

Input: M[i]
Outputs: max, min, average

max = 0
min = MAX  // the maximal constant
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32

Data M[i] are stored in memory.

Results are stored in the internal registers.
Circuit Interface

clk  reset  in_data  in_addr  write  start

MIN_MAX_AVR

done  out_data  out_addr

n  5  2
## Interface Table

<table>
<thead>
<tr>
<th>Port</th>
<th>Width</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>System clock</td>
</tr>
<tr>
<td>reset</td>
<td>1</td>
<td>System reset – clears internal registers</td>
</tr>
<tr>
<td>in_data</td>
<td>n</td>
<td>Input data bus</td>
</tr>
<tr>
<td>in_addr</td>
<td>5</td>
<td>Address of the internal memory where input data is stored</td>
</tr>
<tr>
<td>write</td>
<td>1</td>
<td>Synchronous write control signal – validity of in_data</td>
</tr>
<tr>
<td>start</td>
<td>1</td>
<td>Starts the computations</td>
</tr>
<tr>
<td>done</td>
<td>1</td>
<td>Asserted when all results are ready</td>
</tr>
<tr>
<td>out_data</td>
<td>n</td>
<td>Output data bus used to read results</td>
</tr>
<tr>
<td>out_addr</td>
<td>2</td>
<td>01 – reading minimum</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 – reading maximum</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 – reading average</td>
</tr>
</tbody>
</table>
Datapath

Input: $M[i]$
Output: max, min, average

max = 0
min = max
sum = 0
for i=0 to 31 do
    \[ d = M[i]; \]
    \[ \text{sum} = \text{sum} + d \]
    if (d < min) then
        \[ \text{min} = d \]
    endif
    if (d > max) then
        \[ \text{max} = d \]
    endif
endfor
average = sum/32
Datapath

Input: $M[i]$  
Output: max, min, average

max = 0  
min = max  
sum = 0  
for $i=0$ to $31$ do
    $d = M[i]$;
    sum = sum + $d$
    if ($d < min$) then
        min = $d$
    endif
    if ($d > max$) then
        max = $d$
    endif
endfor
average = sum/32
State Diagram for Controller

Input: M[i]
Outputs: max, min, average

max = 0
min = MAX
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32
State Diagram for Controller

Input: M[i]
Outputs: max, min, average

max = 0
min = MAX
sum = 0
for i=0 to 31 do
    d = M[i];
    sum = sum + d
    if (d < min) then
        min = d
    endif
    if (d > max) then
        max = d
    endif
endfor
average = sum/32

start=1 / rst<=1
start=0 /
done<=0
i < 32 / i++
i==32 / done<=1

Output logic: in_addr <= i;
out_data <= ...

Sorting
Sorting - Example

<table>
<thead>
<tr>
<th>Addr</th>
<th>Before sorting</th>
<th>During Sorting</th>
<th>After sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Legend:
- position of memory indexed by $i$: $M_i$
- position of memory indexed by $j$: $M_j$
Pseudocode

\[
\text{for } i=0 \text{ to } k-2 \text{ do } \\
\quad A = M[i] \\
\quad \text{for } j=i+1 \text{ to } k-1 \text{ do } \\
\quad\quad B = M[j] \\
\quad\quad \text{if } A > B \text{ then } \\
\quad\quad\quad M[i] = B \\
\quad\quad\quad M[j] = A \\
\quad\quad\quad A = B \\
\quad\quad \text{end if} \\
\quad \text{end for} \\
\text{end for}
\]

\(K\) is a constant, the number of integers to be sorted in memory.

\(M\) denotes memory.

Memory address is either \(i\) or \(j\).
Sorting – Interface

Memory

Sort

clock

Clock

Reset

Start

Done

N

N

k

addr

we

din

dout
Sorting – Datapath

- Registers to hold A, B,
- Memory addresses i and j
- Incrementor
- Comparator

\[
\text{for } i=0 \text{ to } k-2 \text{ do} \\
\quad A = M[i] \\
\quad \text{for } j=i+1 \text{ to } k-1 \text{ do} \\
\qquad B = M[j] \\
\qquad \text{if } A > B \text{ then} \\
\qquad\quad M[i] = B \\
\qquad\quad M[j] = A \\
\qquad\quad A = B \\
\qquad\text{end if} \\
\quad \text{end for} \\
\text{end for}
\]
for $i=0$ to $k-2$ do
    $A = M[i]$
    for $j=i+1$ to $k-1$ do
        $B = M[j]$
        if $A > B$ then
            $M[i] = B$
            $M[j] = A$
            $A = B$
        end if
    end for
end for
Sorting – Datapath

\[
\text{for } i=0 \text{ to } k-2 \text{ do}
\]
\[
A = M[i] \\
\text{for } j=i+1 \text{ to } k-1 \text{ do}
\]
\[
B = M[j] \\
\text{if } A > B \text{ then}
\]
\[
M[i] = B \\
M[j] = A \\
A = B
\]
\text{end if}
\]
\text{end for}
\text{end for}
for $i=0$ to $k-2$ do
    $A = M[i]$
    for $j=i+1$ to $k-1$ do
        $B = M[j]$
        if $A > B$ then
            $M[i] = B$
            $M[j] = A$
            $A = B$
        end if
    end for
end for
Sorting – Controller

• Nested loops by two FSMs: one for the outer loop controls the one for the inner loop.
• Reuse the FSM for the single for loop in the previous example.

```plaintext
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
```
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
Behavioral Level Design

inputs \rightarrow \textit{Combinational Logic} \rightarrow \textit{reg\_next} \rightarrow \textit{register} \rightarrow \textit{clk} \rightarrow \textit{output} \rightarrow \textit{reg}
FSMD

for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for
for i=0 to k-2 do
    A = M[i]
    for j=i+1 to k-1 do
        B = M[j]
        if A > B then
            M[i] = B
            M[j] = A
            A = B
        end if
    end for
end for

i = 0;
while i < k-1 do
    addr = i
    A = M[addr]
    j = i + 1
    while j < k do
        addr = j
        B = M[addr]
        if A > B then
            addr = i
            M[addr] = B
            addr = j
            M[addr] = A
            A = B
        end if
        j = j + 1
    end while
    i = i + 1;
end while
FSMD

```
1   i = 0;
2   while i < k-1 do
3       addr = i
4       A = M[addr]
5       j=i+1
6           while j < k do
7               addr = j
8               B = M[addr]
9               if A > B then
10                  addr = i
11                  M[addr] = B
12                  addr = j
13                  M[addr] = A
14                  A = B
15                  end if
16               j=j+1
17           end while
18       i = i+1;
19   end while
```
FSMD

Current State | Next State | Cond     | Operations
-------------|------------|----------|--------------
1            | 2          | start='1' | i <= 0       
2            | 3          | i < k-1  | null         
2            | 18         | !(i<k-1) | done <= '1'  
3            | 6          | true     | addr <= i,  
             |            |          | A <= M[addr];  
             |            |          | j <= j+1;  
6            | 7          | j < k    | null         
6            | 17         | !(j<k)   | null         
7            | 10         | true     | j++; addr <= j; B <=  
             |            |          | M[addr];  
10           | 16         | A > B    | addr <= i; M[addr] <= B;  
10           | 16         | !(A > B) | null         
16           | 6          | true     | null         
17           | 2          | true     | null         

1 i = 0;
2 \textbf{while} i < k-1 \textbf{do}
3     addr = i
4     A = M[addr]
5     j = i+1;
6     \textbf{while} j < k \textbf{do}
7         j = j+1
8     addr = j
9     B = M[addr]
10    \textbf{if} A > B \textbf{then}
11    addr = i
12    M[addr] = B
13    addr = j
14    M[addr] = A
15    A = B
16    \textbf{end if}
17 \textbf{end while}
18 \textbf{end while}
i = 0;
while i < k-1 do
    addr = i
    A = M[addr]
    j = i+1
    while j < k do
        addr = j
        B = M[addr]
        if A > B then
            addr = i
            M[addr] = B
            addr = j
            M[addr] = A
            A = B
            end if
        end while
        j = j+1
    end while
    i = i + 1
end while
Optimization for Performance
Performance Definitions

• **Throughput**: the number of inputs processed per unit time.

• **Latency**: the amount of time for an input to be processed.

• Maximizing throughput and minimizing latency in conflict.

• Both require timing optimization:
  – Reduce delay of the critical path
Achieving High Throughput: Pipelining

• Divide data processing into stages
• Process different data inputs in different stages simultaneously.

\[
x_{\text{power}} = 1; \\
\text{for } (i = 0; i < 3; i++) \\
\quad x_{\text{power}} = x \times x_{\text{power}};
\]

Throughput: 1 data / 3 cycles = 0.33 data / cycle.
Latency: 3 cycles.
Critical path delay: 1 multiplier delay

```
process (clk) begin
  if rising_edge(clk) then
    if start='1' then
      cnt <= 3;
      done <= '0';
    elsif cnt > 0 then
      cnt <= cnt - 1;
      x_{\text{power}} <= x_{\text{power}} \times x;
    elsif cnt = 0 then
      done <= '1';
    end if;
  end if
end process;
```
Achieving High Throughput: Pipelining

```
xpower = 1;
for (i = 0; i < 3; i++)
    xpower = x * xpower;
```

Throughput: 1 data / cycle
Latency: 3 cycles + register delays.
Critical path delay: 1 multiplier delay

```
process (clk, rst) begin
    if rising_edge(clk) then
        if start='1' then -- stage 1
            x1 <= x;
xpower1 <= x;
            done1 <= start;
        end if;
        -- stage 2
        x2 <= x1;
xpower2 <= xpower1 * x1;
done2 <= done1;
        -- stage 3
        xpower <= xpower2 * x2;
done <= done2;
    end if;
end process;
```
Achieving High Throughput: Pipelining

• Divide data processing into stages
• Process different data inputs in different stages simultaneously.
Achieving High Throughput: Pipelining

- Divide data processing into stages
- Process different data inputs in different stages simultaneously.

*Penalty: increase in area as logic needs to be duplicated for different stages*
Reducing Latency

- Closely related to reducing critical path delay.
- Reducing pipeline registers reduces latency.

![Diagram of pipeline stages with registers](image.png)
Reducing Latency

• Closely related to reducing critical path delay.
• Reducing pipeline registers reduces latency.
Timing Optimization

• Maximal clock frequency determined by the longest path delay in any combinational logic blocks.
• Pipelining is one approach.
Timing Optimization: Spatial Computing

- Extract independent operations
- Execute independent operations in parallel.

\[ X = A + B + C + D \]

```defer
process (a, b, c, d) begin
  X1 := A + B;
  X2 := X1 + C;
  X  <= X2 + D;
end process;
```

```defer
process (a, b, c, d) begin
  X1 <= A + B;
  X2 <= C + D;
  X  <= X1 + X2;
end process;
```
**Timing Optimization: Spatial Computing**

\[
X = A + B + C + D
\]

```vhdl
process (a, b, c, d) begin
  X1 := A + B;
  X2 := X1 + C;
  X <= X2 + D;
end process;
```

Critical path delay: 3 adders
Timing Optimization: Spatial Computing

\[ X = A + B + C + D \]

```vhdl
process (a, b, c, d) begin
  X1 <= A + B;
  X2 <= C + D;
  X <= X1 + X2;
end process;
```

Critical path delay: 2 adders
Timing Optimization: Avoid Unwanted Priority

process (clk, rst) begin
    if rising_edge(clk) then
        if c0='1' then rout <= din1;
        elsif c1='1' then rout <= din2;
        elsif c2='1' then rout <= din3;
        elsif c3='1' then rout <= din4;
        end if;
    end if;
end process;

Critical path delay: 4 2x1MUX.
Timing Optimization: Avoid Unwanted Priority

In the context of FPGA design, we refer to a decision tree as the sequence of conditions that are used to decide what action the logic will take. Usually, this breaks down to if/else and case structures. Consider a very simple register write example:

```
module regwrite(
    output reg rout,
    input clk,
    input [3:0] in,
    input [3:0] ctrl);
```

```
always @(posedge clk)
    if(ctrl[0]) rout <= in[0];
    else if(ctrl[1]) rout <= in[1];
    else if(ctrl[2]) rout <= in[2];
    else if(ctrl[3]) rout <= in[3];
endmodule
```

This type of if/else structure can be conceptualized according to the mux structure shown in Figure 12.1.

This type of decision structure could be implemented in a number of different ways depending on speed/area trade-offs and required priority. This section describes how various decision trees can be coded and constrained to target different synthesized architectures.

12.1.1 Priority Versus Parallel

Inherent in the if/else structure is the concept of priority. Those conditions that occur first in the if/else statement are given priority over others in the tree. A higher priority with the structure above would correspond with the muxes near the end of the chain and closer to the register.

Critical path delay: 4 2x1 MUX.

cont’d from previous slide
process (clk, rst) begin
  if rising_edge(clk) then
    case c is
      when "0001" =>
        rout <= din0;
      when "0010" =>
        rout <= din1;
      when "0100" =>
        rout <= din2;
      when "1000" =>
        rout <= din3;
      when others => null;
    end case;
  end if;
end process;
Timing Optimization: Avoid Unwanted Priority

Critical path delay: 2-AND plus 4-OR.

cont’d from previous slide
Timing Optimization: Register Balancing

• Maximal clock frequency determined by the longest path delay in any combinational logic blocks.
Timing Optimization: Register Balancing

process (clk, rst) begin
  if rising_edge(clk) then
    rA <= A;
    rB <= B;
    rC <= C;
    sum <= rA + rB + rC;
  end if;
end process;

process (clk, rst) begin
  if rising_edge(clk) then
    sumAB <= A + B;
    rC <= C;
    sum <= sumAB + rC;
  end if;
end process;
Optimization for Area
Area Optimization: Resource Sharing

- Rolling up pipeline: share common resources at different time – a form of *temporal* computing

![Diagram of pipeline stages with common resource blocks](image_url)
Area Optimization: Resource Sharing

• Use registers to hold inputs
• Develop FSM to select which inputs to process in each cycle.

\[ X = A + B + C + D \]
Area Optimization: Resource Sharing

- Use registers to hold inputs
- Develop FSM to select which inputs to process in each cycle.

\[ X = A + B + C + D \]
Area Optimization: Resource Sharing

2.3 RESOURCE SHARING

When we use the term **resource sharing**, we are not referring to the low-level optimizations performed by FPGA place and route tools (this is discussed in later chapters). Instead, we are referring to higher-level architectural resource sharing where different resources are shared across different functional boundaries. This type of resource sharing should be used whenever there are functional blocks that can be used in other areas of the design or even in different modules.

A simple example of resource sharing is with system counters. Many designs use multiple counters for timers, sequencers, state machines, and so forth. Often-times, these counters can be pulled to a higher level in the hierarchy and distributed to multiple functional units. For instance, consider modules A and B. Each of these modules uses counters for a different reason. Module A uses the counter to...
Merge duplicate components together

Figure 2.4 Shared counter.

Area Optimization: Resource Sharing

Module A

8 bits

Comparator

Comparator == 255

2.5 μs strobe

Counter Module

bits 7:0

Reset

Comparator == hex 6ff

bits 10:0

11 bit Counter

Module B

11 bits

Comparator

PWM 5.5 kHz, 0–100% duty

Pulse Width

0 to hex 6ff
Impact of Reset on Area – Xilinx Specific

Do not set or reset Registers asynchronously.
– Control set remapping becomes impossible.
– Sequential functionality in device resources such as block RAM components and DSP blocks can be set or reset synchronously only.
– You will be unable to leverage device resources resources, or they will be configured sub-optimally.
– Use synchronous initialization instead.

Do not describe Flip-Flops with both a set and a reset.
– No Flip-Flop primitives feature both a set and a reset, whether synchronous or asynchronous.
– If not rejected by the software, Flip-Flop primitives featuring both a set and a reset may adversely affect area and performance.
Resetting Block RAM

- On-chip block RAM only supports synchronous reset.
- Suppose that Mem is 256x16b RAM.
- Implementations of Mem with synchronous and asynchronous reset on Xilinx Virtex-4.

Table 2.4 Resource Utilization for BRAM with Synchronous and Asynchronous Resets

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Slices</th>
<th>Flip-flops</th>
<th>4 Input LUTs</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous reset</td>
<td>3415</td>
<td>4112</td>
<td>2388</td>
<td>0</td>
</tr>
<tr>
<td>Synchronous reset</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Optimization for Power
Power Reduction Techniques

• In general, FPGAs are power hungry.
• Power consumption is determined by

\[ P = V^2 \cdot C \cdot f \]

where \( V \) is voltage, \( C \) is load capacitance, and \( f \) is switching frequency
• In FPGAs, \( V \) is fixed, \( C \) depends on the number of switching gates and length of wires connecting all gates.
• To reduce power,
  • turn off gates not actively used,
  • have multiple clock domains,
  • reduce \( f \).
Dual-EdgeTriggered FFs

• A design that is active on both clock edges can reduce clock frequency by 50%.

Example 1

Example 2

positively triggered

negatively triggered
Backup
Input: $M[i]$
Outputs: max, min, average

max = 0
min = $\text{MAX}$
sum = 0
for $i=0$ to $31$ do
  $d = M[i]$;
  sum = sum + $d$
  if $(d < \text{min})$ then
    min = $d$
  endif
  if $(d > \text{max})$ then
    max = $d$
  endif
endfor
average = sum/32