Producing Better Circuits Using VHDL

- Understanding cost of VHDL circuit elements
- Estimating resource usage of larger circuit modules
- Using synthesis report

Jon Turner
VHDL Coding and Circuit Quality

- Many ways to express circuit design in VHDL
- Best choice determined by many factors
  - design and maintenance cost – focus on simplicity, clarity
  - circuit cost – # of flip flops, gates, LUTs, wiring, memory
  - performance – minimum clock period
- Different choices can yield very large differences
- To improve circuit quality, important to understand (at least roughly) what synthesizer does
  - understand cost of common elements
    - adders, comparators, muxes, decoders, counters,…
  - study synthesis reports for clues to what synthesizer does
  - synthesize sub-circuits separately, experiment
Simple ALU Example

- ALU with 16 bit data inputs $A$, $B$ and control input $C$

$$X \leftarrow B \quad \text{when } C = "00" \text{ else }$$
$$\text{not } B + 1 \quad \text{when } C = "01" \text{ else }$$
$$A + B \quad \text{when } C = "10" \text{ else }$$
$$A - B;$$

- "default" implementation uses adder, subtracter, 2s-complement circuit and 4:1 mux
- synthesized circuit uses 49 LUTs

- Alternate architecture

```
A1 <= A and (15 downto 0 => C(1));
B1 <= B xor (15 downto 0 => C(0));
X <= A1 + B1 when C(0) = '0'
else A1 + B1 + 1;
```

- yields 16 LUT circuit
Arbiter Example

- Arbiter with 4 request inputs, 4 request outputs and uses “client list” for fairness
  
  entity arbiter is port(
    clk, reset: in std_logic;
    requests: in std_logic_vector(0 to 3);
    grants: out std_logic_vector (0 to 3));
  end arbiter;

- Six different designs
  
  flops range from 17 to 27
  LUTs from 35 to 127
  period from 7.5 to 12.9 ns

<table>
<thead>
<tr>
<th>design</th>
<th>flip flops</th>
<th>LUTs</th>
<th>min period</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>35</td>
<td>9.9</td>
</tr>
<tr>
<td>1</td>
<td>27</td>
<td>127</td>
<td>12.9</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>108</td>
<td>10.3</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>69</td>
<td>8.1</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>71</td>
<td>10.6</td>
</tr>
<tr>
<td>5</td>
<td>27</td>
<td>78</td>
<td>7.5</td>
</tr>
</tbody>
</table>
Barrel Shifter Example

- **Implements right rotate operation**

```
process (dIn, shift)
begin
    for i in 0 to n-1 loop
        dOut(i) <= dIn(i-shift);
    end loop;
end process;
```

- **Produces n-mux circuit**
  - uses about $n^2/2$ LUTs
**Alternate version**

- **Binary-shift version**
  
  ```vhdl
  process (dIn, shift)
  variable t: unsigned(n-1 downto 0);
  begin
    t := dIn;
    for j in 0 to lgN-1 loop
      if shift(j) = '1' then
        t := t(n-(1+2**j) .. 0) 
          & t(n-1 .. -2**j);
      end if; end loop;
    dOut <= temp;
  end process;
  ```

- **Uses $n \log_2 n$ LUTs (for $n=64$, 384 vs. 2048)**

- **Alternate specification**
  
  - `dout <= rotate_left(dIn, int(shift))`
  
  - note: avoid VHDL shift/rotate operators (sll, ror,...)
LUTs for Common Building Blocks

- Some cases depend on FPGA “helper circuits”
  - note: your mileage may vary
  - differences across synthesizers, FPGAs, ASICs,…

<table>
<thead>
<tr>
<th>width=n</th>
<th>no constant inputs</th>
<th>one constant input</th>
</tr>
</thead>
<tbody>
<tr>
<td>incremmer/adder</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>select bit (mux)</td>
<td>( n/2 )</td>
<td>0</td>
</tr>
<tr>
<td>decoder</td>
<td>( n+n/8 )</td>
<td>0</td>
</tr>
<tr>
<td>modify selected bit</td>
<td>( n+n/4 )</td>
<td>1</td>
</tr>
<tr>
<td>equality compare</td>
<td>( n/2 )</td>
<td>( n/4 )</td>
</tr>
<tr>
<td>less-than compare</td>
<td>( n )</td>
<td>( n/2 )</td>
</tr>
<tr>
<td>shift/rotate</td>
<td>( n \log_2 n )</td>
<td>0</td>
</tr>
<tr>
<td>loadable register</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>loadable counter</td>
<td>( n )</td>
<td>-</td>
</tr>
</tbody>
</table>
Example – Debouncer

- Flip flops – $2n+16$ (where $n$ is width)
- LUTs – 16 for count, $3n/4$ for comparators
  - for $n=8$, expect 28, synthesis report says 27
Estimating Resource Usage

- Flip flops
  - relatively easy to account for
    - just count signals on left side of “synchronized assignments”
  - state machines a little trickier – binary vs. one-hot
  - variables can cause synthesizer to generate flip flops
  - also flip flops sometimes replicated to improve timing

- LUTs
  - identify common building blocks (adders, comparators,...)
    - focus on multi-bit signals; these use most LUTs
  - consider combinational logic needed for each signal
    - combine all assignments

- Memory
  - configurable block RAMS – 18K bits each on proto board
  - distributed RAM uses LUTs – 16 bits per LUT
Example – Priority Queue

- $2 \times k$ array of cells
  - each column has registers, muxes for shifting and swapping, plus comparison circuit
  - $n = \text{width of key, value}$

- Estimate resources
  - $2k(2n+1)$ flip flops
  - $2k(2n+1)$ LUTs for muxes
  - $kn$ LUTs for comparators

- Example, $n = k = 16$
  - est. 1056 flip flops, 1312 LUTs
  - synthesis report:
    - 1059 flip flops, 1468 LUTs
Using Synthesis Report

- Use design summary to do quick “sanity check”
  - is flip flop count reasonably close to estimate?
    - check synth. report for other registers/counters, replication
  - is LUT count roughly comparable with flip flop count?
    - if many more LUTs than flip flops, try to understand why
    - do detailed estimate of LUTs; if synthesis report differs greatly from estimate, try to understand why

- Separate out contributions from “sub-components”
  - use high level analysis from synthesis report
    - lists counters, comparators, decoders, etc.
  - for more detail, synthesize sub-components separately
    - for finer-grained analysis, comment out parts of VHDL spec and re-synthesize (care needed here)
  - use spreadsheet to analyze resource usage
Reducing Resource Usage

- Flip flops
  - understand what data is stored in registers
  - eliminate unnecessary registers, share where possible
  - move large data arrays into memory

- LUTs
  - identify expensive sub-circuits and consider alternates that may be less expensive
  - avoid needless replication of combinational functions
    - if circuit contains \( f(X_1),...,f(X_n) \) with values used at different times, replace with \( f(Y) \) where \( Y=selectFrom(X_1,...,X_n) \)
  - adjust synthesis options – space/speed, 1-hot/binary,…

- Memory
  - use distributed memory for small memory blocks

- Wiring – limit “width” of interfaces
Example – WASHU-2

- Flip flop counts
  - area case: synthesizer drops extra tick bit
  - speed case: synthesizer replicates some state/tick flops

- LUT counts
  - 10% under-estimate (not bad)
  - 3.6 LUTs per flip flop
  - possible optimization targets?
    - ALU, decode, dispReg

<table>
<thead>
<tr>
<th>item</th>
<th>flip flops</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>IREG</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>IAR</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>ACC</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>this</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>ALU</td>
<td>0</td>
<td>80</td>
</tr>
<tr>
<td>aBus</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>dBus</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>target</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>decode</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>state</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>tick</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>dispReg</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>if-conditions</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>total</td>
<td>101</td>
<td>326</td>
</tr>
<tr>
<td>actual/area</td>
<td>100</td>
<td>361</td>
</tr>
<tr>
<td>actual/speed</td>
<td>106</td>
<td>389</td>
</tr>
</tbody>
</table>
High Level Synthesis Report

Synthesizing <cpu>... Found finite state machine for signal <state>.

| States | 17 |
| Transitions | 54 |

Found 16-bit tristate buffer for signal <aBus>.
Found 16-bit tristate buffer for signal <dBus>.
Found 16-bit 4-to-1 multiplexer for signal <dispReg>.
Found 16-bit register for signal <acc>.
Found 16-bit adder for signal <alu$addsub0000>.
Found 16-bit register for signal <iar>.
Found 16-bit register for signal <iReg>.
Found 16-bit register for signal <pc>.
Found 16-bit adder for signal <pc$addsub0000> created at line 146.
Found 16-bit adder for signal <target>.
Found 16-bit register for signal <this>.
Found 4-bit register for signal <tick>.
Found 4-bit adder for signal <tick$add0000> created at line 134.

Summary: inferred 1 Finite State Machine(s).
- inferred 84 D-type flip-flop(s).
- inferred 4 Adder/Subtractor(s).
- inferred 16 Multiplexer(s).
- inferred 32 Tristate(s).
Example - Console

- Subcomponent estimates from separate synth. runs
- Estimates
  - LUT count within 10%
  - 17 extra flip flops!
- Synthesis report explains
  INFO:Xst:2146 - In block <console>, Counter <lcdnCounter> <snoopCnt> are equivalent, XST will keep only <lcdnCounter>.

<table>
<thead>
<tr>
<th>console</th>
<th>estimate</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>flip flops</td>
<td>LUTs</td>
<td></td>
</tr>
<tr>
<td>debouncer</td>
<td>24</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>knob</td>
<td>43</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>display</td>
<td>36</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>aBus</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dBus</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>snoopAdr</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>snoopData</td>
<td>16</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>snoopCnt</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>lcdnCounter</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>regSelect</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>nuChar</td>
<td>0</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>hex2Ascii</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>prevDBtn</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>singletons</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>177</td>
<td>356</td>
<td></td>
</tr>
<tr>
<td>actual/area</td>
<td>160</td>
<td>389</td>
<td></td>
</tr>
</tbody>
</table>
Another Surprise from Synth. Report

- Extract

  Synthesizing Unit <console>.
  Found 16x8-bit ROM for signal <nuChar$rom0000> created at line 261.
  Found 16x8-bit ROM for signal <nuChar$rom0001> created at line 264.
  ...
  Found 16x8-bit ROM for signal <nuChar$rom0011> created at line 282.
  repeating hex2Ascii ROM 12 times – why??

- Not smart enough to identify exclusive cases

  process (cpuReg, snoopAdr, snoopData, selekt) begin
  case selekt is
    when "00000" | "00110" | "10000" | "10110" =>
      nuChar <= c2b(hex2Ascii(int(cpuReg(15 downto 12))));
    when "00001" | "00111" | "10001" | "10111" =>
      nuChar <= c2b(hex2Ascii(int(cpuReg(11 downto 8))));
    ...
    when "01100" => nuChar<=c2b(hex2Ascii(int(snoopAdr(15 downto 12))))
    when "01101" => nuChar<=c2b(hex2Ascii(int(snoopAdr(11 downto 8))));
    ...
  end case;
Alternate version

process (cpuReg, snoopAdr, snoopData, selekt) begin
  case selekt is
    when "00000" | "00110" | "10000" | "10110" =>
      showDigit <= cpuReg(15 downto 12);
    when "00001" | "00111" | "10001" | "10111" =>
      showDigit <= cpuReg(11 downto 8);
    when "00010" | "01000" | "10010" | "11000" =>
      showDigit <= cpuReg(7 downto 4);
    when "00011" | "01001" | "10011" | "11001" =>
      showDigit <= cpuReg(3 downto 0);
    when "01100" => showDigit <= snoopAdr(15 downto 12);
    when "01101" => showDigit <= snoopAdr(11 downto 8);
    when "01110" => showDigit <= snoopAdr(7 downto 4);
    when "01111" => showDigit <= snoopAdr(3 downto 0);
    when "11100" => showDigit <= snoopData(15 downto 12);
    when "11101" => showDigit <= snoopData(11 downto 8);
    when "11110" => showDigit <= snoopData(7 downto 4);
    when "11111" => showDigit <= snoopData(3 downto 0);
    when others => showDigit <= x"0";
  end case;
end process;
numChar<=x"20" when selekt(3 downto 1)="010" or selekt(3 downto 1)="101" else c2b(hex2Ascii(int(showDigit)));
Using VHDL Effectively

- Optimize circuits only when good reason to do so
- Ask “What would the synthesizer do?”
- Use CAD tools to evaluate alternatives
  - synthesize components separately and look at resource counts and timing analysis in synthesis report
- If you cannot figure out a circuit that the synthesizer might produce, consider rewriting
  - be cautious about VHDL specs for which you are unsure about the corresponding circuit
  - look for ways to simplify your code
- “A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.” – Antoine de Saint-Exupery