VHDL Synthesis Basics

TIE-50206 Logic Synthesis
Arto Perttula
Tampere University of Technology
Fall 2016
Acknowledgements

• Prof. Pong P. Chu provided "official" slides for the book which is gratefully acknowledged
  – See also: http://academic.csuohio.edu/chu_p/

• Most slides were originally made by Ari Kulmala
  – and other previous lecturers (Teemu Pitkänen, Konsta Punkka, Mikko Alho, Erno Salminen…)

• M. Perkows, ECE 590. DIGITAL SYSTEM DESIGN USING HARDWARE DESCRIPTION LANGUAGES, Portland State University, June 2008,
  http://web.cecs.pdx.edu/~mperkows/CLASS_VHDL_99/June2008/
Foreword: VHDL and Synthesis

• The main goal of writing VHDL is to generate *synthesizable description*
• This lecture presents some practical examples of how to write code that is good for synthesis
• The quality of the design is much affected by the coding style
  ➢ You must be able to choose structures that synthesize the best
RTL SYNTHESIS BASICS
Abstraction Levels – from Algorithm to Circuit

Algorithm
0% technology dependent

Architecture
10% technology dependent

Behavioral synthesis

Register level
20% technology dependent

Logic synthesis

Gate level
100% technology dependent

For $i = 0 : i = 15$

$\text{sum} = \text{sum} + \text{data}[i]$

Adapted from [M. Perkows, Class VHDL 99, June 2008]
Synthesis Phases
Technology Mapping

- First step in RTL synthesis produces generic netlist
- Generic components (AND, NOR, DFF…) must be mapped to suitable equivalents in target technology
  - Logic elements and embedded macros in FPGAs
  - Standard cells in ASIC
- Zillion legal mappings
- Tools decide heuristically based on number of inputs, drive strengths, leakage currents, wire delays…
Optimization Targets

- Logic can be optimized according to multiple criteria
  - Area
  - Delay
  - Power (dyn/leak/tot)
  - Code clarity, ease of verification, debugging, maintenance
- Trade-off! You cannot get all!
Logic Design Basics Still Apply

• Modularize the design to components
  – Easier to design single components
  – Easier to upgrade
  – Prefer registered outputs
• Use version control system (e.g., SVN or git)
• Thou shalt not mess with clock and reset
• Preferably active-high enable signals
• Asynchronous reset is used only to initialize
  – Not part of the functionality
  – Hence, you don’t force reset from your code
  – Use separate clear-signal or similar if needed. That is checked on the edge sensitive if-branch
Simplest Synthesis Example

- In this course, we concentrate on RTL synthesis: how is HDL converted into netlist of basic gates and flip-flops
  - Technology mapping, routing and placement are beyond the scope of this course
- **Example**: `arith_result <= a + b + c - 1;`
  - The resulting combinatorial logic is straightforward
  - Inclusion of DFF depends on the context (inside synchronous process or not)
Example: if-else

```plaintext
if boolean_expr_1 then
  if boolean_expr_2 then
    signal_a <= value_expr_1;
  else
    signal_a <= value_expr_2;
  end if;
else
  if boolean_expr_3 then
    signal_a <= value_expr_3;
  else
    signal_a <= value_expr_4;
  end if;
end if;
```

Conceptual structure of nested if-clauses in HDL

Conceptual hardware realization
(when none of the value_expr_x is "ZZ...Z". High-impedance covered in Lec 12.)
Example: Selected Assignment

Similar to if-else

```vhdl
signal a, b, c, x, y, r: std_logic;

r <= a when x = y else
    b when x > y else
    c;
```

Fig 1. Basic form of synthesized logic

Fig 2. Full logic
Logic from case-clause

- This example has 2 outputs but again the logic is similar to if-clause

```vhdl
case case_exp is
  when c0 =>
    sig_a <= value_expr_a_0;
    sig_b <= value_expr_b_0;
  when c1 =>
    sig_a <= value_expr_a_1;
    sig_b <= value_expr_b_1;
  when others =>
    sig_a <= value_expr_a_n;
    sig_b <= value_expr_b_n;
end case;
```

Conceptual structure of case-clause in HDL

Conceptual HW realization (when tri-state isn’t used)
More Complex Selected Assignment

Fig 1. Conceptual hardware realization

Fig 2. Full logic

```vhdl
signal a, b, r: unsigned(7 downto 0);
signal x, y: unsigned(3 downto 0);

r <= a+b when x+y>1 else
   a-b-1 when x>y and y!=0 else
   a+1;
```
Differences Between if-elsif and case

• Several conditions in if-elsif may evaluate to true
  – Priority order, the first true will be executed
• Only single condition may evaluate true in case
  – Order of statements in VHDL does not matter for synthesis
  – (Written order does matter for the human reader)
• If-elsif produces a chain of muxes, if conditions are not mutually exclusive
• Case produces a wide mux = tree of of multiplexers
  – Smart synthesis tools do the same for if-elsif if possible
• Area is the same in both cases
• Worst case delay is $O(n)$ for chain and $O(\log n)$ for tree
Example: loop

library ieee;
use ieee.std_logic_1164.all;

entity wide_xor is
  port(
    a, b: in std_logic_vector(3 downto 0);
    y: out std_logic_vector(3 downto 0)
  );
end wide_xor;

architecture demo_arch of wide_xor is
  constant WIDTH: integer := 4;
begin
  process(a, b)
  begin
    for i in (WIDTH-1) downto 0 loop
      y(i) <= a(i) xor b(i);
    end loop;
  end process;
end demo_arch;

- Bounds must be static, like here (3 down 0)
- The loop is "unrolled" in logic
- Everything happens in parallel!
- Hence, the loop is equivalent to

  \[
  \begin{align*}
  y(3) & \leftarrow a(3) \text{ xor } b(3); \\
  y(2) & \leftarrow a(2) \text{ xor } b(2); \\
  y(1) & \leftarrow a(1) \text{ xor } b(1); \\
  y(0) & \leftarrow a(0) \text{ xor } b(0);
  \end{align*}
  \]

Sidenote: \( y \leftarrow a \text{ xor } b \) is even better with std_logic_vectors, but then we would not have an example of a loop 😊
Loops: Example 2 in SW

• With software
  
  ```c
  for (i = 0; i < max_c; i++) {
    b(i) = a(i) + i;
  }
  ```

• Iterative calculation for \( b(i) \) (simplified)
  
  1. Calculate for-clause
  2. Fetch a
  3. Add
  4. Store b
  5. Increment i
  6. Go back to 1

• Takes a lot of clock cycles (several even with loop-unrolling)
Loops: Example 2 in HW

- **Hardware:**
  ```
  add_i: for i in 0 to max_c-1 generate
  b(i) <= a(i) + i;
  end generate add_i;
  ```

- **Generate `<max_c>` parallel computation units**
  - High area overhead

- **Results generated in 1 clock cycle, very fast!**

- **However, in HW we can adjust the area-performance ratio**
  - Needs FSM for looping
  - Pipeline, e.g., half of the result on the first cycle, rest on the second
  - Even fully sequential HW, still faster than SW
    - E.g., 1 result/cycle on HW, 1 result/ 6 cycles on SW
SW Technique: Loop Unrolling

• Branches are often slower than other assembly instructions in a CPU
• Loop unrolling: perform many operations on single loop iteration, less branches
  – http://en.wikipedia.org/wiki/Loop_unwinding
  – Some compilers do that automatically

\[
\text{for } i=0 \text{ to } 49 \\
\quad c[i] = a[i] \times b[i]
\]

\[
\text{for } i=0 \text{ to } 24 \\
\quad c[i] = a[i] \times b[i] \\
\quad c[i+25] = a[i+25] \times b[i+25]
\]

Unroll by 2

• Things are simple
  – If loop bounds are known at compile-time
  – If number of iterations is divisible by unrolling factor (e.g., 50 divisible by 2, like here)
  – Otherwise, see http://en.wikipedia.org/wiki/Duff%27s_device

• Loop pipelining: hide the latency of memory operations
  – Often load and store operations are also slower
  – Load \([i+1]\)s started on iteration \(i\)

• Loop parallelization: execute different iteration on different cores
  – Limited by data dependencies since iterations may be executed in random order
GUIDELINES FOR SYNTHESISABLE HDL
Synthesis Is Integral Part Right from the Start

• Synthesis tools are great but…
  – They behave differently. Some structures are not accepted by all tools.
• Separate non-synthesizable (test bench) code into their own entities
• Start trial syntheses early (e.g., day 1). Non-synthesizable structures will be detected early as well.
• Automate synthesis runs with scripts
  – Will be repeated dozens or hundreds of times during a project
  – Often it is good to perform parameter sweep, e.g., synthesize data widths 8, 16, …128 bits
• You can skip non-synthesizable parts with pragmas. Use with care.
  -- synthesis translate_off
  use std.textio.all;
  -- synthesis translate_on
General Guidelines And Hints

• **Do not optimize** your design before the HW cost (area, delay) have been proven!
• Initial values for signals are not generally synthesizable
  – Used only in simulators (and some synthesis tools)
  – You must reset all sequential (control) signals explicitly
  – You must NOT reset any combinatorial signals or those coming from component instance’s outputs
• Use `std_logic` data types in I/O
• Use `numeric_std` package for arithmetic
• Use only descending range in the arrays (e.g., `downto`)
  – `Signal write_r : std_logic_vector(data_width_g-1 downto 0)`
  – `Signal write_out : std_logic_vector(0 to data_width_g-1)`
General Guidelines And Hints (2)

- Parentheses to show the order of evaluation
  - \( A \text{ and} ( x \text{ or } b) \)

- Assignment delay, such as "\( a \leq b \text{ after } x \text{ ns} \)”, is problematic
  - Assignment will be synthesized but not the delay
  - This example will produce a simple wire
  - If you "fix" bugs in code like this, it won’t work after the synthesis
  - Only place to use non-synthesizable code is test benches

- Variables are synthesizable, but…
  - It is harder to figure out the resulting logic than with signals

- High-impedance state 'Z' is synthesizable but…
  - Simulation results and real HW do not always match
Notes on Combinational Circuit Synthesis

- Do not instantiate basic gates (AND, NOR…)
- Synthesis tools are very good at boolean logic optimization
  - Like Karnaugh map/Quine-McCluskey
  - Not all intermediate signal are preserved in synthesis
  - You should aim for code clarity
- Note that the minimal Boolean equation is not necessarily the fastest/smallest/lowest power depending on the used technology
  - Simple looking \((a \text{ AND } B)\) might turn to \((a' \text{ NOR } B')\)
  - Correct schematic may seem strange for a novice
Notes on Combinational Circuit Synthesis (2)

• Always write a complete sensitivity list
• In every combinatorial process invocation, every signal must be assigned a value
  – Assigned in every branch or with default assignment
  – Otherwise generates latches to hold the previous values
  – We practically never want to have latches from RTL combinatorial processes
  – Usual suspect: Incomplete if-else or such
• Avoid combinational loops!
  – The same signal on both sides of assignment inside combinatorial process
  – E.g., \( a <= a+1; \) -- aargh!
Notes on Sequential Circuit Synthesis

• Do not instantiate flip-flops
• Most synthesis tools do not move (optimize) combinatorial logic across flip-flops, at least not by default
• Possible modifications
  – Propagating constant values may save a lot in area and delay
  – Removing duplicated logic and registers saves area
  – Register duplication can cut long wiring delays
  – Register retiming can move register and combinatorial logic to balance the delays of a pipeline
• These are not always enabled by default, and not always automatically detected by the tools
  – Try first enabling them, and the try modifying your RTL
Local Optimization Steps

• Concentrate on clarity and functionality
  – You cannot beat the tools in Boolean algebra
• Define your targets
• Measure and identify critical areas
• Try automated optimizations
  – Source remains clear
• Optimize manually
  – Source gets obfuse and harder to port

Figure 6.20  Synthesis iterations and the impact of RT-level change.
Basic RTL Optimizations

• Prove and locate the problem first!
• Smallest possible data width is of course desired
• Iterative algorithms trade area for delay
  – One can share complex units via multiplexing
• Constant operands simplify Boolean equations
  – For example, consider 4 bit comparator
    a) \( x = y \cdot (x_3 \oplus y_3)' \cdot (x_2 \oplus y_2)' \cdot (x_1 \oplus y_1)' \cdot (x_0 \oplus y_0)' \)
    b) \( y = 0 \quad x_3' \cdot x_2' \cdot x_1' \cdot x_0' \)
• Even the most basic operations have different costs

8.11.2016
Arto Perttula
Resource Sharing

• Arithmetic operators
  – Large implementation
  – Limited optimization by synthesis software
  – Data width has a major impact
• Area reduction can be achieved by ”sharing” in RT level coding
  – Operator sharing
  – Functionality sharing
• Clever tools may do that automatically and not-so-smart ones need some guidance (different coding style)
Resource Sharing (2)

• Possible when "value expressions" in priority network and multiplexing network are **mutually exclusive**:
• Only one result is routed to output
• Generic format of conditional signal assignment guarantees this:

```plaintext
sig_name <= value_expr_1 when boolean_expr_1 else 
  value_expr_2 when boolean_expr_2 else 
  value_expr_3 when boolean_expr_3 else 
  . . . 
  value_expr_n;
```
Sharing Example 1

• Original code:
  \[ r <= a+b \text{ when boolean}_\text{exp} \text{ else } a+c; \]

• Revised code (enables sharing):
  \[ \text{src0} <= b \text{ when boolean}_\text{exp} \text{ else } c; \]
  \[ r <= a + \text{src0}; \]

NOTE: This example is coded outside a process
Area: 2 adders, 1 mux, Bool
Delay:
\[
\max(T_{adder}, T_{boolean}) + T_{mux}
\]

However, no free lunch in general: sharing reduces area A but increases delay T in this case
Sharing Example 2

- Original code:
  ```
  process(a,b,c,d,...)
  begin
    if boolean_exp then
      r <= a+b;
    else
      r <= a+c;
  end if;
  end process;
  ```

- Revised code:
  ```
  process(a,b,c,d,...)
  begin
    if boolean_exp then
      src0 <= a;
      src1 <= b;
    else
      src0 <= a;
      src1 <= c;
    end if;
  end process;
  r <= src0 + src1;
  ```

**NOTE:**
Coded inside a process
Equivalent with previous
Propagating Constants

- Generics and other system settings may disable parts of the logic in practice
  - E.g., \( \text{sel} \) is an input of ALU
- This may not be noted on the unit level (first step of synthesis)
  - Preceding unit sets constantly \( \text{sel}=0 \) and voilá!
- Disabling is noted when the neighbour units are also synthesized and connected
- Sometimes you must enable this optimization explicitly (depends on the tool)
- Sometimes this optimizes surprisingly long paths of logic away
  - Good for area, but might confuse debugging

Arto Perttula
Removing Duplicate Logic

• Instantiating modules is recommended but may result in duplicate logic
• Example bus uses counters to track the reservation state
• In practice, all the counters inside the interfaces tick synchronously
• Smart synthesis tool (or designer) detects this and instantiates **only one counter**
• Note that this also requires optimization across the unit boundaries
Register Duplication

- Wiring has a notable delay
  - Tens of percents in FPGA
- Hence, both logic and routing delays must be balanced
- Example above has 3 path and critical path delay is 9ns
- Long 5ns-wire is split with additional register
  - Little bit larger area
  - Critical path reduced from 9ns to 6ns
  - The outputs of highlighted registers (orig + duplicate) are identical

5+3+1=9 ns
5 ns
Register Retiming

- Moves combinatorial logic to the otherside of a DFF
- The function of DFF’s output changes naturally
- This optimization is not allowed if the original output is needed in 2+ places
- E.g., MUL has long delay, approximately the same as ADD and CMP together
- Extreme case places pipeline registers (e.g., 3) behind an purely combinatorial function in RTL
  - Register retiming splits combinatorial logic approximately evenly
  - Bumber of connections between small clouds decides the number of needed parallel registers
- Pipeline balancing is very tedious by hand!
Arithmetic Optimization/Resource Sharing

- Note that $a-b = a+(-b)$
- Moreover, negation logic is simpler than subtraction
- In 2’s complement number system – $X = X'+1$
- Very case-dependent optimization style
  - Possibly large savings
  - Not obvious to verify

Q: How to optimize the the MUX and INV at the bottom
A: Replace with 8 XOR gates. It’s your homework to show that :)
Fixed-Point Calculation

- Digital signal processing algorithms
  - Often developed in floating-point (in Matlab)
  - Later mapped into **fixed point** for digital hardware realization
- Fixed-point digital hardware
  - Lower area, lower power
- $n$-bit word is divided into
  - 1 sign bit: + or –
  - $I$ integer bits
  - $F$ fraction bits
- "Binary point" is not stored anywhere
Fixed Point (2)

- There’s a synthesis time trade-off between:
  - Dynamic range (large I)
  - Accuracy (large F)
- E.g., Q3.4
  - Range = -7.9375 - +7.9375
  - Resolution $1/2^4 = 1/16 = 0.0625$
- Q5.2
  - Range = -31.75 - +31.75
  - Resolution $1/2^2 = 1/4 = 0.25$

- Floating-point scales range vs. accuracy dynamically
- With fixed point designer must take care of it:
  - Wordwidth selection
  - #int bits and #fraction bits
  - Scaling

Cost $c(w)$

Distortion $d(w)$

Optimum wordlength

Arto Perttula
Operands And Data Types Have Large Impact Also in SW

- Floating-point operations can be really costly on a low-end softcore or microcontroller
  - No floating-point unit (FPU), and sometimes not even a multiplier/division unit
- Sometimes char is slower than int due to shifting and masking

<table>
<thead>
<tr>
<th>Double</th>
<th>Float</th>
<th>Int</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=b+c</td>
<td>1 101</td>
<td>465</td>
<td>6</td>
</tr>
<tr>
<td>a=b-c</td>
<td>1 099</td>
<td>493</td>
<td>6</td>
</tr>
<tr>
<td>a=b/c</td>
<td>5 312</td>
<td>1 345</td>
<td>144</td>
</tr>
<tr>
<td>a=b/c</td>
<td>4 021</td>
<td>654</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sin(b)</th>
<th>119 487</th>
<th>119 368</th>
<th>120 685</th>
<th>78 309</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cos(b)</td>
<td>119 366</td>
<td>119 637</td>
<td>121 106</td>
<td>78 561</td>
</tr>
<tr>
<td>Sqrt(b)</td>
<td>4 262</td>
<td>5 308</td>
<td>7 172</td>
<td>7 962</td>
</tr>
<tr>
<td>Abs(b)</td>
<td>345</td>
<td>247</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>B&lt;31</td>
<td>2 595</td>
<td>515</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>B Mod C</td>
<td>--</td>
<td>--</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>B Xor C</td>
<td>--</td>
<td>--</td>
<td>57</td>
<td>51</td>
</tr>
<tr>
<td>Keskiavo</td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Double</th>
<th>Float</th>
<th>Int</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=b+c</td>
<td>468</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>a=b-c</td>
<td>774</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>a=b/c</td>
<td>1 617</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>a=b/c</td>
<td>1 974</td>
<td>43</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sin(b)</th>
<th>33 461</th>
<th>35 189</th>
<th>41 671</th>
<th>35 695</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cos(b)</td>
<td>34 560</td>
<td>35 201</td>
<td>41 782</td>
<td>34 442</td>
</tr>
<tr>
<td>Sqrt(b)</td>
<td>2 037</td>
<td>3 000</td>
<td>3 266</td>
<td>3 174</td>
</tr>
<tr>
<td>Abs(b)</td>
<td>1 306</td>
<td>266</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>B&lt;31</td>
<td>298</td>
<td>203</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>B Mod C</td>
<td>--</td>
<td>--</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>B Xor C</td>
<td>--</td>
<td>--</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>Keskiavo</td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Nios II/f, mul=off, div=off, fpu=off, Area= 3 493 LEs

Nios II/f, mul=on, div=on, fpu=on, Area=11 662 LEs

Arto Perttula 8.11.2016
Do Not Set Unused Registers to Zero

- Do not set registers to null when value is not needed
  - E.g., if valid_in = '0' then data_r <= (others =>'0');
  - Unnecessary MUX at D input

- Nevertheless, it's good for visualization in simulation, since 'Z' looks different

```vhdl
if dbg_enable_g = '1' then
    reg <= dbg_value;
  -- e.g. 'Z'
```

In simulation, it's easy to see when these are valid

In real HW, validity determined according to signal `empty`

---

Arto Perttula 8.11.2016 42
Aim for "Good-Enough"

- Do not overoptimize HW, if performance limits is known
  - 100 frames/sec encoder is not better than 25 fps, if camera restricts the frame rate anyway
- Minimizing critical path causes larger area and power
  - Requires larger drive strength for gates (large and leaky)
  - Area and power increase super-linearly
- Minimizing cycle count needs many parallel sub-blocks (e.g., ALUs)
- Remember the good ol’ Amdahls’s law
- Add performance and statistics counters to your logic
  - Great help in profiling and debugging

Figure 3. Speed to area and power tradeoff (0.13m LVLK-OD)

Fig: [J. Wei, C. Rowen, “Implementing low-power configurable processors…”, DAC 2005]
Multiplexers

• Multiplexers form a large portion of the logic utilization, especially in FPGA
• E.g., 30% of Nios II/f soft-core processors area are multiplexers
• If-structure generates a priority multiplexer
  
  ```plaintext
  IF cond1 THEN z <= a;
  ELSIF cond2 THEN z <= b;
  ELSIF cond3 THEN z <= c;
  ELSE z <= d;
  END IF;
  ```

• It is preferred to use case clause
  
  ```plaintext
  CASE sel IS
  WHEN cond1 => z <= a;
  WHEN cond2 => z <= b;
  WHEN cond3 => z <= c;
  WHEN OTHERS => z <= d;
  END CASE;
  ```

  - Creates a balanced multiplexer tree since conditions are mutually exclusive by definition
  - Sometimes, if-elsif does the same, but it is not guaranteed
Multiplexers #2

- Do not let the simplicity of VHDL trick you
- Multiplexing four 32-bit words requires
  - 130 input bits (2 control bits + 128 data bits), 32 output bits
  - A lot of routing
  - 32 x 4-to-1 multiplexers
  - A 4-to-1 multiplexer requires three 2-to-1 multiplexer
  - One 2-to-1 multiplexer implementable in one basic logic element
  - => 3*32=96 2-to-1 multiplexers required, 96 LEs consumed
- Check your design and see if you can reduce the number of choices
  - E.g., having only 2 options instead of 3 as some options are fixed at synthesis-time
Shifters

• Variable amount shifting is area-hungry
  – Assume 32-bit vector that can be shifted arbitrary amount to left or right
  – Needs a 32-to-1 multiplexer for every result bit!
    • 32-to-1 multiplexer = 31 2-to-1 multiplexers = 31 Les
  – 32*31 = **992** 2-to-1 multiplexers (=LEs)
  – Non-constant shifters are generally not supported (automatically) by synthesis tools

• An FPGA-specific trick is to use the embedded multipliers to do the dynamic shifting
  – Multiplying by $2^n$ shifts the result to left by $n$
  – Faster and more area-efficient than doing this with LEs
Comparators

- $<$, $>$, $==$  
- Avoid implementing $==$ in general logic cells. Comparators are implementable using arithmetic operations and fast carry chains.
  - Calculate $a - b$ and check if the result is negative, zero, or positive.
  - Synthesis tools should be aware of this automatically...
- Recall that $x = a[6:0] < b[6:0]$ is the same as $x = \text{signed}(a[6:0] - b[6:0])[7]$
  - The last carry [overflow] of the subtraction.
  - Note: in ASICs it may not be feasible to use arithmetics for comparison.
Synteza instrukcji przypisań - operatory relacji

entity relational_ops_1 is  
  port (a, b: in bit_vector (0 to 3);  
         m: out boolean);  
end relational_ops_1;  

architecture example of relational_ops_1 is  
begin  
  m <= a = b;  
end example;  

entity relational_ops_2 is  
  port (a, b: in integer range 0 to 15;  
         m: out boolean);  
end relational_ops_2;  

architecture example of relational_ops_2 is  
begin  
  m <= a >= b;  
end example;  

Example Counter to Be Optimized...

process (clk, rst_n)
begin
  if rst_n... 
  elsif clk'event and clk='1' then
    if (count = value1) then -- check limit
      count <= 0;
    else
      count <= count + 1; -- increment
    end if;
  end if;
end process;

• Make a timer with run-time adjustable period
• Comparator takes here 2 variables, value[3:0] and q[3:0], as inputs
Optimized Basic Counter

process (clk, rst_n)
begin
  if rst_n...
  elsif clk'event and clk='1' then
    if (count_r = 0) then
      count <= value1; -- initialize
    else
      count <= count - 1; -- decrement
    end if;
  end if;
end process;

Now we count to constant zero

- Comparator takes 1 variable q[3:0] and a constant 0...0
- Smaller area and delay
An Example 0.55 um Standard-Cell CMOS Implementation

<table>
<thead>
<tr>
<th>width</th>
<th>nand</th>
<th>xor</th>
<th>area (gate count)</th>
<th>VHDL operator</th>
<th>delay (ns)</th>
<th>area (gate count)</th>
<th>VHDL operator</th>
<th>delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>nand</td>
<td>xor</td>
<td>&gt;_a</td>
<td>&gt;_d</td>
<td>=</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>22</td>
<td>25</td>
<td>68</td>
<td>26</td>
<td>27</td>
<td>33</td>
<td>51</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>44</td>
<td>52</td>
<td>102</td>
<td>51</td>
<td>55</td>
<td>73</td>
<td>101</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>85</td>
<td>105</td>
<td>211</td>
<td>102</td>
<td>113</td>
<td>153</td>
<td>203</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>171</td>
<td>212</td>
<td>398</td>
<td>204</td>
<td>227</td>
<td>313</td>
<td>405</td>
</tr>
</tbody>
</table>

Subscripts
a = area-optimized
d = delay-optimized

Asymptotic cost:
Nand: area is O(n) and time O(1), whereas
">" area is O(n) and time O(n)
Background: **Big-O Notation for Algorithmic Complexity**

- Way to approximate how the cost increases with the number of inputs $n$
- **Function** $f(n)$ belongs to class $O(g(n))$:
  - if $n_0$ and $c$ can be found to satisfy: $f(n) < cg(n)$ for any $n$, $n > n_0$
  - $g(n)$ is simple function: $1$, $\log_2 n$, $n$, $n\log_2 n$, $n^2$, $n^3$, $2^n$
- Following are $O(n^2)$:
  - $0.1n^2$
  - $n^2 + 5n + 9$
  - $500n^2 + 1000000$

Arto Perttula
Interpretation of Big-O

• Filter out the "interference": constants and less important terms
• Algorithms with $O(2^n)$ is intractable, but already $O(n^3)$ is very bad
  – Not realistic for a larger $n$
  – Frequently, for tractable algorithms, sub-optimal solution exists
• One may develop a heuristic algorithm
  – They do not guarantee optimal solution, but usually provide rather good one with acceptable cost
  – Often utilize pseudo-random choices
  – For example, simulated annealing and genetic algorithms
### Example

<table>
<thead>
<tr>
<th>input size</th>
<th>Big-O function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>2</td>
<td>2 $\mu s$</td>
</tr>
<tr>
<td>4</td>
<td>4 $\mu s$</td>
</tr>
<tr>
<td>8</td>
<td>8 $\mu s$</td>
</tr>
<tr>
<td>16</td>
<td>16 $\mu s$</td>
</tr>
<tr>
<td>32</td>
<td>32 $\mu s$</td>
</tr>
<tr>
<td>48</td>
<td>48 $\mu s$</td>
</tr>
<tr>
<td>64</td>
<td>64 $\mu s$</td>
</tr>
</tbody>
</table>

**intractable**
Specific to FPGA (details in lecture 10)

• A lot of registers – use them
  – Aggressive pipelining
  – Objective is to hide the routing delays as much as possible
    • Simple logic stages between registers

• Adders
  – Generally, it’s not beneficial to share adders
  – FPGAs often contain (e.g., Altera) special structures for adders
    • Sharing of adders may cost as much as the adder itself
  – Synthesis tool can select proper adder architecture (ripple-carry, carry-select…), no need to do it manually

• Hard macros
  – Use whenever appropriate
    • Higher performance than by building one with the FPGA native resources
    • ”They are there anyway”
  – Embedded multipliers and small SRAMs are common
FPGA #2

• Get to know properties of the device
• E.g., FPGA on-chip memories are typically multiples of 9 bit wide
  – The ninth bit can be used for
    • Control
    • Parity bit
  – Otherwise, it is wasted!
  – Memories are typically dual-ported, take advantage of this
DSP Blocks in Cyclone II

- E.g., there are 70 9x9 bit DSP blocks in Cyclone II FPGA chip
  - Synthesis tool can automatically combine DSP blocks
- One DSP block can save 50-170 LEs and several nanoseconds
- In addition to total #LEs, you must always report from FPGA synthesis
  - #4-input LUTs, # registers
  - # embedded multipliers
  - # embedded memory bits
- Example: Operation c=a*b using DSP or only LEs

<table>
<thead>
<tr>
<th>a width</th>
<th>b width</th>
<th>c width</th>
<th>Only LE's</th>
<th>Only DSP (9bit elements)</th>
<th>critical path (LE)</th>
<th>critical path (DSP)</th>
<th>Equivalent ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>58</td>
<td>1</td>
<td>9.971 ns</td>
<td>6.594 ns</td>
<td>282</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>16</td>
<td>101</td>
<td>1</td>
<td>9.704 ns</td>
<td>6.430 ns</td>
<td>457</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>18</td>
<td>115</td>
<td>1</td>
<td>11.293 ns</td>
<td>6.486 ns</td>
<td>580</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>163</td>
<td>2</td>
<td>11.551 ns</td>
<td>7.014 ns</td>
<td>881</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>18</td>
<td>202</td>
<td>2</td>
<td>12.434 ns</td>
<td>7.039 ns</td>
<td>1105</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>32</td>
<td>341</td>
<td>2</td>
<td>15.521 ns</td>
<td>7.144 ns</td>
<td>1768</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
<td>683</td>
<td>6 (+ 47 LE's)</td>
<td>19.254 ns</td>
<td>11.781 ns</td>
<td>3341</td>
</tr>
</tbody>
</table>
DSP-block area

Equivalent ports (NAND)

Sum of DSP-block input and output ports

Arto Perttula

8.11.2016
Division: Quartus Results

- Measure area and delay of division as a function of data width
- Huge differences! E.g., in 30 bit case, 47x area and 28x delay
- Note the peculiarity!
  - Variable divided by non-power-of-two constant (e.g., 7) is not optimized when using *signed* data type but it is optimized with type *integer*
Conclusions

• Coding style has a profound effect on the quality of the hardware
  – Area, maximum clock frequency
  – Loops
  – Complex assignment logic creates a sea of multiplexers
    • E.g., variable amount left-right shifter

• Synthesis tools create different but functionally equivalent netlists even for small designs

• Know your FPGA!
  – You might save area and time if using some hard-coded macros
  – However, these are tricks that you should only use on the final optimization phase
References

Sharing Example 3

Figure 7.3 Operator sharing based on a multiplexer.

with sel_exp select
    r <= a+b when "00",
    a+c when "01",
    d+1 when others;

with sel_exp select
    src0 <= a when "00" | "01",
    d when others;

with sel_exp select
    src1 <= b when "00",
    c when "01",
    "00000001" when others;

r <= src0 + src1;
Sharing Example 4

(a) Original diagram

(b) Diagram with sharing

process (a, b, c, d, ...)
begin
  if boolean_exp then
    x <= a + b;
    y <= (others=>'0');
  else
    x <= "00000001";
    y <= c + d;
  end if;
end process;

process (a, b, c, d, sum, ...)
begin
  if boolean_exp then
    src0 <= a;
    src1 <= b;
    x <= sum;
    y <= (others=>'0');
  else
    src0 <= c;
    src1 <= d;
    x <= "00000001";
    y <= sum;
  end if;
end process;

Figure 7.4 Complex operator sharing.

Arto Perttula
8.11.2016
Types

- Using own types may significantly clarify the code
- Declaration:
  ```
  TYPE location IS
  RECORD
    x: INTEGER range 0 to location_max_c-1;
    y: INTEGER range 0 to location_max_c-1;
    valid : std_logic;
  END RECORD;
  ```
  ```
  TYPE locations_type IS ARRAY (0 to 3) of location;
  SIGNAL loc_r : locations_type;
  ```
- Usage:
  ```
  For i in 0 to 3 loop
    loc_r(i).x <= i;
    loc_r(i).y <= 3-i;
    loc_r(i).valid <= '1';
  End loop;
  ```

<table>
<thead>
<tr>
<th>i</th>
<th>x</th>
<th>y</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>'1'</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>'1'</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>'1'</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>'1'</td>
</tr>
</tbody>
</table>
Types #2

• Initialization of an array constant:

```plaintext
custom a_bound_c : integer := 2;
type vector_2d is array (0 to a_bound_c-1) of std_logic_vector(1 downto 0);
type vector_3d is array (0 to a_bound_c-1) of vector_2d;
custom initial_values_c : vector_3d := (('00', '01'), ('10', '11'));
```

• You may split initialization to multiple lines to increase readability

<table>
<thead>
<tr>
<th>Initial_values_c</th>
<th>c0</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0 \ c1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>&quot;00&quot;</td>
<td>&quot;01&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;10&quot;</td>
<td>&quot;11&quot;</td>
</tr>
</tbody>
</table>

\[c_1 = \text{horizontal} \quad c_0 = \text{vertical}\]
• Special case, have to use positional assignment:

```vhdl
constant ip_types_c : integer := 1;
type ip_vect is array (0 to ip_types_c-1 ) of integer;
constant ip_amount_c : ip_vect := (0 => 1); -- right way

constant ip_amount2_c : ip Vect := (1); -- does not work!
constant ip_amount2_c : ip_vect := 1; -- does not work!
  -- ** Error: rtm_pkg.vhd(20): Integer literal 1 is not of type ip_vect.
```

• There is only a single value but it is an array nonetheless
Not Supported by Synthesis

- Signals in packages (global signals)
- Signal and variable initialization
  - Typically ignored (there are exceptions, e.g., Xilinx FPGA synthesis)
- Unconstrained while and for loops
- More than one 'event in a process
- Multiple wait statements
- Physical types, for example time
- Access types
- File types
- Guard expression
- (Sensitivity lists, delays and asserts are ignored)
DSP Block Synthesis

library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

entity mul_sync is
  generic (
    a_width_g : integer := 32;
    b_width_g : integer := 32;
    c_width_g : integer := 32);
  port (
    a     : in  std_logic_vector(a_width_g-1 downto 0);
    b     : in  std_logic_vector(b_width_g-1 downto 0);
    c     : out std_logic_vector(c_width_g-1 downto 0);
    rst_n : in  std_logic;
    clk   : in  std_logic);
end mul_sync;

architecture rtl of mul_sync is
begin -- rtl
mul : process (clk, rst_n)
begin -- process mul
  if rst_n = '0' then -- asynchronous reset (active low)
    c <= (others => '0');
  elsif clk'event and clk = '1' then -- rising clock edge
    c <= std_logic_vector(to_unsigned(to_integer(unsigned(a)) *
                               to_integer(unsigned(b)), c_width_g));
  end if;
end process mul;
end rtl;

• Synthesis script for Synopsys Design Compiler
analyze -library WORK -format vhdl ./SRC/mul_sync.vhd
elaborate MUL_SYNC -architecture RTL -library DEFAULT -parameters "a_width_g = 18, b_width_g = 18, c_width_g = 18"
check_design -multiple_designs
create_clock -name "clk" -period 4 -waveform (0 2) [clk]
set_clock_uncertainty 0.1 clk
set_clock_latency 0.2 clk
set_clock_transition 0.1 clk
set_dont_touch_network clk
set_dont_touch rst_n
set_ideal_network rst_n
set_max_area 1000
set_wire_load_model -lib umcl18g212t3_tc_180V_25C -name "suggested_10K"
set_wire_load_mode top
compile -exact_map -area_effort high
report_qor > ./SYN/RPT/report_qor_default.txt
report_timing > ./SYN/RPT/report_timing_default.txt
report_area > ./SYN/RPT/report_area_default.txt
report_area -hierarchy > ./SYN/RPT/report_area_hierarchy.txt
write_file -format ddc -hierarchy -output ./SYN/DDC/mul888_elab.ddc
write_file -format verilog -hierarchy -output ./SYN/NETLIST/mul_sync_default_compiled.v
The result of compilation of one cell (basic gate, dff, mux etc).

Two transistors in series

Diffusion n

Diffusion p

metalization

polysilicon

contact

Logic has been mapped to basic cells and they have placed-and-routed. In std-cell technology, there are specific rows for placing logic and for wiring. All logic cells in a row share the same voltage supply and ground. Note that cells have uniform height but different width.
Standard-Cell Layout Example

Same as previous but shown multiple silicon layers at once.