Lecture 9
Tcl Basics and Logic Synthesis Intro

Xuan ‘Silvia’ Zhang
Washington University in St. Louis

http://classes.engineering.wustl.edu/ese461/
Outline

Tcl Basics

Logic Synthesis Intro
My First Tcl Script

- Hello world

```tcl
#hello.tcl
set s "hello world!"
puts "* $s *
exit 0
#end of hello.tcl

$ tclsh hello.tcl
* hello world! *0
```
Tcl Basics

• String based interpreted command language
• Tcl scripts
  - consist of one or more commands
  - command followed by parameters or arguments
  - separated by white spaces or tabs
• Example
  - print “My first Tcl script”
  - puts: output a new line
  - stdout: output to the terminal

ex: puts stdout "My first Tcl script."

=> My first Tcl script.
Tcl Basics

• Two step process
  - parsing: define substitutions
  - execution: meaning is applied to command arguments

• Example
  - set: read and write variables

\[
\begin{align*}
\text{set a 5} \\
\text{set b a+8} & \leq \text{ b is “a+8” b is “13”}\Rightarrow \\
& \text{set a 5} \\
& \text{set b [expr $a+8]} \\
\end{align*}
\]
  - pair of brackets invokes an additional evaluation
Tcl Basics

• Declare and instantiate variable
  - `set`

• Retrieve variable value
  - `$`: variable substitution
  - `[ ]`: command substitution
  - `\`: escape for $, new line

• Quote and comment
  - `""`: space, tabs, semicolons treated as ordinary characters
  - `{ }`: all special character lose their meanings, defer evaluation
  - `#`: the remaining line is a comment
Tcl Basics

- **Variable**
  - has a name and a value (stored as a string)

- **Expression**
  - combine operands and operators to create new values

- **Operators**
  - relational operators: <, <=, >, >=, ==, !=
    - return 0 (false) or 1 (true)
  - bitwise operators: &, |, ^, <<, >>, ~
    - require operands to be integers
  - ternary operator: ?
    - e.g. `expr {($a < $b)? $a : $b}`
Tcl Basics

• **Mathematical expressions**
  - `expr` command evaluate math expressions
  - concatenate all arguments into string
  - parse it as math expression according to C-syntax

• **Example**

```bash
#!/usr/bin/tclsh
set a 2.0
set b 2.0
set c [expr $a*$b]
set d $a*$b
puts "$d=$c" exit 0
```
Tcl Basics

• Control flow
  - if, elseif, else
  - while
  - for
  - foreach

• Procedure
  - reusable packaged modules
    ```tcl
    proc procedure_name arguments ?args? body
    ```
    ```tcl
    proc plus {a b} {expr $a+$b}
    ```

• Output format
  - string: %s
  - decimal: %d
  - real number: %f
  - character: %c
Example Synthesis Script

# Script for Cadence RTL Compiler synthesis
# Erik Brunvand, 2008
# Use with syn-rtl -f rtl-script
# Replace items inside <> with your own information

# Set the search paths to the libraries and the HDL files
# Remember that "." means your current directory. Add more directories
# after the . if you like.
set_attribute hdl_search_path { ./ }
set_attribute lib_search_path { ./ }
set_attribute library [ list < your-target-library > . lib ]
set_attribute information_level 6

set myFiles [ list < HDL-files > ] ;; # All your HDL files
set basename < top-module-name > ;; # name of top level module
set myClk < clk > ;; # clock name
set myPeriod_ps < num > ;; # Clock period in ps
set myInDelay_ns < num > ;; # delay from clock to inputs valid
set myOutDelay_ns < num > ;; # delay from clock to output valid
set runname < string > ;; # name appended to output files
Example Synthesis Script

# Analyze and Elaborate the HDL files
read_hdl ${myFiles}
elaborate ${basename}

# Apply Constraints and generate clocks
set clock [define_clock -period ${myPeriod_ps} -name ${myClk} [clock_ports]]
external_delay -input $myInDelay_ns -clock ${myClk} [find / -port ports_in/*]
external_delay -output $myOutDelay_ns -clock ${myClk} [find / -port ports_out/*]

# Sets transition to default values for Synopsys SDC format,
# fall/rise 400ps
dc::set_clock_transition .4 $myClk

# check that the design is OK so far
check_design -unresolved
report timing -lint

# Synthesize the design to the target library
synthesize -to_mapped

# Write out the reports
report timing > ${basename}_${runname}_timing.rep
report gates > ${basename}_${runname}_cell.rep
report power > ${basename}_${runname}_power.rep

# Write out the structural Verilog and sdc files
write_hdl -mapped > ${basename}_${runname}.v
write_sdc > ${basename}_${runname}.sdc
Tcl Reference

- https://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html
- http://wiki.tcl.tk/1304
Outline

Tcl Basics

Logic Synthesis Intro
Standard Cell Design Flow
Design Flow Division

- **Front End**
  - specify, explore
  - design, capture
  - synthesize
  - output: structural RTL

- **Back End**
  - input: structural
  - place and route
  - mask making
  - fabrication
Front End Design Flow

1. System Requirement (marketing)
   - Third party
   - Legacy in-house
   - New Write in-house

2. Architectural Modelling
   - Design Possibilities

3. System Architecture
   - Design Decisions

4. APIs
   - Firmware
   - Application Code
   - RTL
   - Assertions

5. Structure
   - Behavioral HW Desc

6. Design Database
   - SystemC Models

7. HIGH-LEVEL SYNTHESIS

8. Instruction Set Simulator(s)
   - RTL design path

9. BEHAVIOURAL RTL
   - Assertion Checking
   - PASS/FAIL

10. RTL SYNTHESIS
    - Target Technology Cell Library

11. STRUCTURAL RTL
    - Timing Closure

Flowchart illustrating the design process, starting from system requirements and progressing through architectural modeling, system architecture, APIs, structure, design database, high-level synthesis, instruction set simulation, behavioral RTL, assertion checking, RTL synthesis, and structural RTL.
Back End Design Flow

1. **Regression library**
   - RTL Simulation
   - Timing Closure Path

2. **BEHAVIOURAL RTL**
   - Assertion Checking
   - Target Technology Cell Library

3. **RTL SYNTHESIS**

4. **STRUCTURAL RTL**
   - Formal Equiv Checking
   - Formal Equiv Checking

5. **LAYOUT PLACE AND ROUTE**
   - Back Annotation

6. **ANNOTATED RTL**
   - Formal Equiv Checking

7. **MASK MAKING**

8. **Fabrication at Foundry (e.g., TSMC)**

9. **Silicon Chips**
Logic Synthesis

- HDL (Verilog or VHDL) <-> gate-level netlist
  - C compiler: C code <-> machine language
  - convert a high-level description of design into an optimized gate level representation
  - e.g. verilog by connecting standard cells

- Target library
  - standard cell library
  - basic logic gates like and, or, not, xnor, or macro cells like adder, mux, and flip-flops

- Mapping and Optimizing
  - timing engine
  - power, delay, area (but not number of cells)
Example: Comparator/MUX

```vhdl
// comp_mux.v

module comp_mux(a, b, outp);
input [2:0] a, b;
output [2:0] outp;
function [2:0] compare;
input [2:0] ina, inb;
begin
if (ina <= inb) compare = ina;
else compare = inb;
end
endfunction
assign outp = compare(a, b);
endmodule
```
Example: Comparator/MUX

```verilog
`timescale 1ns / 10ps

module comp_mux_o (a, b, outp);

input [2:0] a; input [2:0] b;

output [2:0] outp;

supply1 VDD; supply0 VSS;

in01d0 B1_i1 (.I(a[2]), .ZN(B1_i1_ZN));
in01d0 B1_i2 (.I(b[1]), .ZN(B1_i2_ZN));
oa01d1 B1_i3 (.A1(a[0]), .A2(B1_i4_ZN), .B1(B1_i2_ZN), .B2(a[1]), .ZN(B1_i3_Z);
fn05d1 B1_i4 (.A1(a[1]), .B1(b[1]), .ZN(B1_i4_ZN));
fn02d1 B1_i5 (.A(B1_i3_ZN), .B(B1_i1_ZN), .C(b[2]), .ZN(B1_i5_ZN));
mx21d1 B1_i6 (.I0(a[0]), .I1(b[0]), .S(B1_i5_ZN), .Z(outp[0]));
mx21d1 B1_i7 (.I0(a[1]), .I1(b[1]), .S(B1_i5_ZN), .Z(outp[1]));
mx21d1 B1_i8 (.I0(a[2]), .I1(b[2]), .S(B1_i5_ZN), .Z(outp[2]));

endmodule
```

<table>
<thead>
<tr>
<th></th>
<th>Delay /ns</th>
<th>No. of standard cells</th>
<th>No. of transistors</th>
<th>Area /mils $^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand design</td>
<td>4.3</td>
<td>12</td>
<td>116</td>
<td>68.68</td>
</tr>
<tr>
<td>Synthesized</td>
<td>2.9</td>
<td>15</td>
<td>66</td>
<td>46.43</td>
</tr>
</tbody>
</table>
Example: Comparator/MUX

- Critical path

```
instance name
inPin --> outPin incr arrival trs rampDel cap cell
      (ns) (ns) (ns) (pf)

a[1] .00 .00 R .00 .04 comp_m...
B1_i4
A1 --> ZN .33 .33 R .17 .03 fn05d1
B1_i3
A2 --> ZN .39 .72 F .33 .06 oa01d1
B1_i5
A --> ZN 1.03 1.75 R .67 .11 fn02d1
B1_i6
S --> Z .68 2.43 R .09 .02 mx21d1
```
Simplified Synthesis Process

- Mature commercial tool
  - over 1 million lines of code

- Parse (Analysis) HDL code

- Elaborate (Translate) HDL code
  - data structure: a graph represented by linked list
  - then covert to a network of generic logic cells (technology-independent)
  - output: synthesized network

\[
\begin{align*}
\text{sel} &= a_1 \cdot b_1 \cdot b_2 + a_0 \cdot b_1 \cdot b_2 + a_0 \cdot a_1 \cdot b_2 + a_1 \cdot b_1 \cdot a_2 + a_0 \cdot b_1 \cdot a_2 + a_0 \cdot a_1 \cdot a_2 + a_2 \cdot b_2;[12.1] \\
\text{outp}_2 &= \text{sel} \cdot a_2 + \text{sel} \cdot b_2;[12.2] \\
\text{outp}_1 &= \text{sel} \cdot a_1 + \text{sel} \cdot b_1;[12.3] \\
\text{outp}_0 &= \text{sel} \cdot a_0 + \text{sel} \cdot b_0;[12.4]
\end{align*}
\]
Simplified Synthesis Process

- Logic optimization
  - attempt to improve the network (area, power, speed)
  - a series of factoring, substitution, and elimination steps
  - still technology-independent

- Technology decomposition
  - build a generic network from optimized logic network

```
sel = [100] * [101] * [102]; [12.5]
[100] = !(a2 * [103]);
[101] = !(b2 * [103]);
[102] = !(a2 * b2);
[103] = !( [104] * [105] * [106]);
[104] = !(a1 * b1);
[105] = !(b0 * [107]);
[106] = !(a0' * [107]);
[107] = !(a1 * !b1);
outp2 = !( [108] * [109]); [12.6]
[108] = !(a2 * !sel);
[109] = !(sel * b2);
```
Simplified Synthesis Process

• Technology mapping
  - match generic network to a specified technology-dependent target cell library

```plaintext
outp2 = MUX(a, b, c) = ac + b!c[12.7]

a = b2 ; b = a2 ; c = sel

sel = MAJ3(w, x, y) = !(wx + wy + xy) [12.8]

w = !a2 ; x = b2 ; y = [103] ;

[103] = OA122(w, x, y, z) = !(((w + x)(y + z)) = !(w!x + !y!z) [12.9]

w = a0 ; x = a1 ; y = !b1 z = [107] ;

[107] = !(b1 + !a1) ; [12.10]

sel = !(((!a0 * (!(a1&!b1) | (b1!*a1) ) * (!a2|b2) ) | (!a2*b2)) ;[12.11]

outp2 = !sel * a2 | sel * b2;

outp1 = !sel * a1 | sel * b1;

outp0 = !sel * a0 | sel * b0;
```
Logic Synthesis Reference

- “Logic Synthesis in a Nutshell”
Lab #3: FPU Adder

- Due 10/10 (Monday)
- floating-point adder
- 32-bit IEEE-754 format

- Note: can’t “cheat” with a “real” variable
- real
  - predefined variable data type in Verilog
  - typically a 64 bit IEEE-754 floating point number
Questions?

Comments?

Discussion?