RTL Synthesis

- RTL = Register Transfer Level
- RTL code (Verilog, VHDL, or something else) completely specifies
  - all registers
  - Logic operations
  - Arithmetic operation
- Synthesis will convert these to meet some combination of an area + delay constraint
  - Boolean minimization techniques used to improve both area and speed
  - Different area, speed constraints will produce different gate level netlists, but will be functionally equivalent

Random Logic vs Arithmetic RTL

- General boolean minimization techniques work well for random logic to meet area/speed constraints
- Not true for Arithmetic operations (addition, multiplication, etc)
  - Design space is too large, and resulting netlists are usually sub-optimal when compared to structured netlists

An Example

- What should be synthesized for 'y <= a + b' where y, a, b are 32 bit values?
- Many different adder structures to choose from:
  - Ripple carry – slow, but area efficient
  - Carry select adder - faster than ripple, but more gates
  - Carry Save adder – fastest adder architecture for general logic gates, but requires lots of gates
- Need a methodology that the RTL synthesis tool can use to choose between various architectures for an arithmetic operation based on speed/area constraints

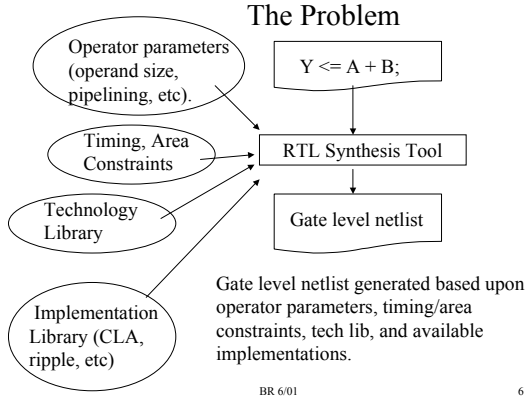
Technology Mapping

- Technology mapping refers to how an RTL synthesis tool maps boolean operations to a set of available gates in a chosen technology
  - ASIC library (nands, nors, complex gates, DFFs)
  - Gate array library (all primitive nands)
  - FPGA library (lookup tables + DFFs)
- Part of technology mapping should also include determining the best structure for arithmetic operations for a given set of constraints
  - I.E. for one technology a 10-bit ripple adder might be a faster than a 10-bit CLA, while in a different technology the opposite is true.

Example Technology Mapping

- In just about any ASIC technology (ie standard cell or gate array), a 12-bit adder is faster done via a CLA structure than a ripple structure
- In the LUT4 (4-input lookup tables) FPGA technology from Xilinx and Altera, the opposite is true
  - Basic programmable cells can implement a two bit sum and has fast carry logic as part of the cell
  - The delay through a LUT4+programmable routing is much slower than the dedicated carry logic+routing between cells
  - This means that ripple chains are more effective than CLA structures for higher values of N than other technologies

The Problem



Gate level netlist generated based upon operator parameters, timing/area constraints, tech lib, and available implementations.

Synopsys Design Ware

- *Design Compiler* is the basic RTL synthesis tool from Synopsys
- *DesignWare* components and libraries is the method by which a user can define custom implementations and technology mappings for arithmetic operations
- The DesignWare Foundation Basic library already has architectures that tradeoff area/speed for many arithmetic operations
  - These architectures (i.e. Ripple vs CLA) are based on generic logic gates and use timing information from the technology library plus area/time constraints to pick an architecture

adder.vhd Example

```
library ieee,synopsys;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use synopsys.attributes.all;
entity adder is
  generic( N : integer := 4);
   port ( a,b: in std_logic_vector(N-1 downto 0);
          sum: out std_logic_vector(N-1 downto 0) );
end adder;
architecture a of adder is
  signal tmpa,tmpb: unsigned(N-1 downto 0);
  signal tmpsum : unsigned(N-1 downto 0);
begin
 tmpa <= unsigned(a);  -- type conversion
 tmpb <= unsigned(b);  -- type conversion
 tmpsum <= tmpa + tmpb;
 sum <= std_logic_vector(tmpsum); -- type conversion
end a;
```

Std_logic_arith defines *unsigned,signed* types needed for arithmetic operations on *std_logic*.

'+' operator only defined for signed, unsigned types.

Unsigned vs Signed types

- For addition, unsigned and signed addition uses the same hardware, so it does not matter which we use
- For other operations like multiplication, it makes a difference
  - Signed means 2's complement representation
  - Different hardware required for signed vs. unsigned multiply

Synthesizing with Synopsys

To place Synopsys on your path, do "swsetup synopsys".

Synopsys synthesis is best used by giving it a command script that has the synthesis commands.

The directory structure that we will use is:
vhdl_course/synopsys/dware_tut/

Script, library file locations

/rtl — RTL files

/gate — Synthesized gate level files

/behv — High level synthesis files

To run a dc_shell script do:

% dc_shell –f *scriptfilename*

A Sample Synopsys Script

```
link_library =  {gcmos_unit.db}
target_library = {gcmos_unit.db}
analyze -f vhdl rtl/addN.vhd
elaborate addN -parameters "N=>8"
set_max_delay 10 -to {sum}
compile -ungroup_all -map_effort high

report_timing -path full -delay max  -to {sum} -max_paths
3  -nworst 1  > add8_dly10.rpt

report_area >> add8_dly10.rpt
change_names -rule vhdl
write -f vhdl -output gate/add8_dly10.vhd
quit
```

Specify target library

Analyze design (create internal representation)

Generate design instance

Set delay constraint

Do synthesis

Report timing, area

Write gate level file in VHDL format

Operator Inference

- The previous Synopsys script and *adder.vhd* file uses *inference* for choosing an operator architecture
  - Inference means that the operator architecture is chosen based on constraints + technology library
- A delay constraint of 10 is specified from input to output via *set_max_delay* command
  - This delay has to match the units specified in the target library
  - The *gcmos.db* library is a generic gate library with only a few gate primitives
  - Has unit delays (delay = 1), unit areas (area = 1) for all delays, areas

Dware Cache

- When Design compiler builds a Design Ware component of a particular type, architecture and size (I.e, adder, ripple, 8 bits) this is cached so that next time will be faster
  - Caches both structure and timing information
  - Cache resides under ~/synopsys_cache_* (exact directory name is version dependent.
- Choosing a particular architecture means that DC has to build architectures of different types to meet the word size, then evaluate each against area/time constraints

Controlling the Architecture Choice

- Can directly control which architecture is used for a particular operator by using a *Synopsys pragma* to specify the architecture
  - *a pragma* is a control directive embedded in a comment
- This can be useful if a particular implementation is required as a starting point for optimization
  - NOTE: the chosen architecture is used a starting point – it will still be modified according to synthesis options
  - I.e., you start with a Ripple adder, and are synthesizing for speed, synthesis transformations will modify it beyond recognition and will probably not be as good as one that started from a CLA structure

Manual Architecture Selection

```
architecture a of adder is
begin
  process(a,b)
  variable tmpa,tmpb: unsigned(N-1 downto 0);
  variable tmpsum : unsigned(N-1 downto 0);
  constant r0 : resource := 0;
  attribute map_to_module of r0: constant is "DW01_add";
  attribute implementation of r0: constant is "rpl";
  attribute ops of r0: constant is "a1";
  begin
   tmpa := unsigned(a);  -- type conversion
   tmpb := unsigned(b);  -- type conversion
   tmpsum := tmpa + tmpb;  -- pragma label a1
   sum <= std_logic_vector(tmpsum); -- type conversion
  end process;
end a;
```

DW component name

DW architecture name

Pragma label must be on line where synthetic operator occurs

Reporting Available Architectures

How does one know the available operators/architectures in a design ware library?

% dc_shell

dc_shell> report_design_lib DW01

Will list all synthetic operators for a particular library. DW01 is the library that contains the basic operators for addition, subtraction, add/sub, inc/dec, multiplication, comparators, shifts

Component Instantiation

```
library ieee,synopsys,DW01;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use synopsys.attributes.all;
use DW01.DW01_components.all;
-- instantiate DW01 component directly
entity adder is
  generic( N : integer := 16  );
   port ( a,b: in std_logic_vector(N-1 downto 0);
          sum: out std_logic_vector(N-1 downto 0) );
end adder;
architecture a of adder is
 attribute implementation:STRING;
 attribute implementation of U1: label is "cla";
 signal l0: std_logic;
begin
   l0 <= '0';
   U1: DW01_add generic map (width => N)
       port map (CI =>l0, A =>a, B=>b,  SUM => sum,
                 CO => open);
end a;
```

Can also instantiate a component directly instead of using operator inference.

Optional blue lines allow manual architecture selection.

Gives user access to ports not available via synthetic operator (e.g., Ci)

Creating a Custom DW library

- Suppliers of implementation technologies (ie, ASIC libraries, PLDs, FPGAs, etc) will also supply a Design Ware library with custom architectures for the basic operators
- This library will take advantage of the unique features of the implementation technology to create a better mapping than what can be done using the basic Design Ware mappings

Tutorial on Creating Dware libraries

- The synopsys software resides at */opt/ecad/synopsy/default* (call this $synopsys)
- *$synopsys/doc/online/dw/dwdg/dwdg_2.pdf* contains a tutorial on creating a custom Dware library
- Creates an adder that has an ov output (overflow output) and has two architectures – ripple ("rpl") and ("cla").
  - I will not attempt to repeat this entire tutorial here, just hit the highpoints
  - You will need to read this tutorial in order to complete the next assignment

DWSL_addov.vhd

- The tutorial creates a new Dware library called DWSL and a component called *DWSL_addov*

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DWSL_addov is
  generic(width : POSITIVE);
  port(A,B : std_logic_vector(width-1 downto 0);
       CI : std_logic;
       SUM : out std_logic_vector(width-1 downto 0);
       OV, CO : out std_logic);
end DWSL_addov;
```

The *ov* output is not a true overflow but simply the carry into the MSB (the *xor* of *ov*, *co* external to model will produce a true overflow)

DWSL_addov_rpl.vhd

```
Library IEEE, gcmos;
use IEEE.std_logic_1164.all;
use gcmos.gcmos_components.all;
architecture rpl of DWSL_addov is
 signal carry : std_logic_vector(width downto 0);
begin
    carry(0) <= CI;
    L1: for I in 0 to width-1 generate
       U1: fa port map(A(i), B(i), carry(i), SUM(i),
              carry(i+1));
    end generate;
    OV <= carry(width-1);
    CO <= carry(width);
end;
```

This has been modified from the original tutorial file to use the gcmos library.

A full adder cell which produces sum, carry-out from a, b, and carry-in

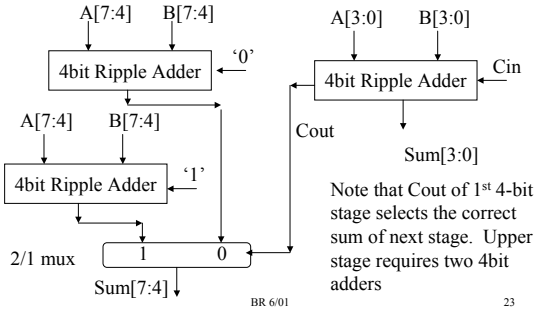
The ripple architecture ("rpl")

DWSL_addov_cla.vhd

- This file is too complex to show on slide but implements a CLA for N < 17.
  - You should look at it to get a feel for how GENERATE statements can be use to create a complex architecture
- For complex architectures, not possible or difficult to work for any value of N
  - "rpl" works for any value of N, but for "cla" the width must be less than 17
- For the next assignment, create an architecture for a carry select adder (see next page)
  - Call this architecture "csel" and has to work for N < 33.
  - Name file DWSL_addov_csel.vhd

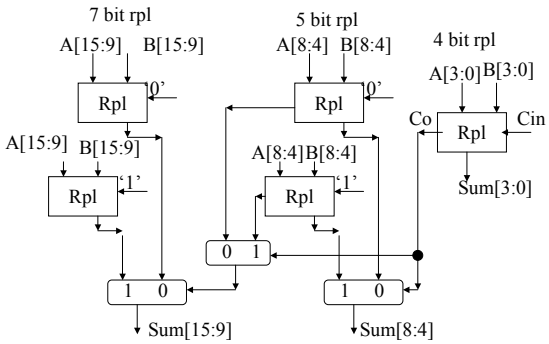
Carry-Select Adder

The Carry path is the slowest path in the ripple carry adder. We can speed it up with the following scheme (8-bit adder):



Note that Cout of 1st 4-bit stage selects the correct sum of next stage. Upper stage requires two 4bit adders

Carry-Select Adder (larger N)



Carry Select comments

- Critical path is still the carry
- The goal is to match the delay along the carry path to the final select on the sum mux to delay of the rpl adder
  - Can increase the ripple size at each stage because the carry delay to the mux select gets longer
  - Exact choice of sizes for each stage depends on gate delays
- In your implementation, choose your own stage sizes
  - CANNOT make them all them size – you must choose some scheme for gradual increase
  - You know that N will be a maximum of 32, so just pick some progression of sizes (like 4-5-7-7-9 or 4-5-7-8-8 or whatever).

Archive *dware.zip*

Unpacks a directory *dware_tut.students* . Important files:

• *gcmos.lib, gcmos.db* GCMOS library

• *DWSL_addov.vhd, DWSL_addov_cla.vhd, DWSL_addov_rpl.vhd, DWSL_addov_csel.vhd* - you must modify the 'csel' architecture.

• *analyze_dwsl.script* - pass this script to *dc_shell* to compile all of the DWSL*.vhd files. Must execute this after any changes to DWSL*.vhd files

• *rtl/{adder_cla.vhd, adder_rpl.vhd, adder_csel.vhd}* -- VHDL files that have manual component instantiations for addov the three respective architectures.

Archive *dware.zip*

- *adder_sample.script* - a sample *dc_shell* script for synthesizing one of the 'rtl/adder*.vhd' designs for a particular bit width. Modify this script or use the Perl script below. The synthesized design is written to the 'gate/' directory.
- *make_design.pl* – a perl script to assist in generating designs for different architectures, N values instead of writing a separate *dc_shell* script for each case. A sample run is:

make_design.pl adder.template %arch%=rpl %dly%=0 %N%=16

will substitute the values shown for the corresponding strings in the adder.template file to create a new *dc_shell* script.

If you don't feel comfortable using this script, simply write your own dc_shell script for each case you want to test.

Archive *dw_test.zip*

- Contains *gcmos/* directory that has VHDL entity/architecture, component package for the gcmos cell library
  - Install as a modelsim library
  - Makefile is gcmos/Makefile.gcmos
  - Look at these files for any questions on cell names, pin names, cell functionality.
- Contains *dw_test.student* directory that should be used for testing your synthesized
  - Rename to *dw_test* and install as modelsim library
  - Makefile.dw_test is included in this directory

Testbench in *dw_test*

- Files *tb16.vhd*, *tb28.vhd* are two testbenches for testing 16-bit and 28-bit adder implementations
  - Generates 100 pairs of random numbers, does sum using *addov* component, prints result
  - Configurations are included in each testbench file for the three gate level architectures
- After generating a gate level implementation (ie., *dware_tut/gate/adder16_rpl.vhd*)
  - Copy to *dw_test* directory, edit file to remove the entity declaration for adder
  - Make sure the architecture name, file name for gate level architecture matches was is expected by the configurations in the testbench files, and also the makefile.

Approach

- Read through the DW tutorial referenced previously
  - You do not have to make any modifications to the files as mentioned in the tutorial, I have already made the changes and converted them to use the GCMOS library
- Try generating a couple of different sized adders for "rpl", "cla" architectures
- Make sure you can simulate these using the *dw_test* library (you might even want to write a testbench for a different sized adder like N=20 to ensure that you understand the files).

Approach (cont.)

- Look at the code in *DWSL_addov_rpl.vhd*, *DWSL_addov_cla.vhd* to understand the GENERATE approach for creating the adder structure
  - Look at the PLD model we covered for more examples of GENERATE statements
- Fill in the architecture of *DWSL_addov_csel.vhd* to create a parameterized carry-select adder
  - Should use the *fa* (full adder), mux2to1 cells primarily
  - The full adder (*fa*) cell function in *gcmos.lib* is not specified so that Synopsys will treat it as a black box – this meant the gate level structure will not be modified via synthesis constraints – easier to debug.
- Generate designs of size N=16, N=28 and test via the *dw_test* testbench – your adder should generate the same results as the other architectures.

How to get help in Synopsys

Within *dc_shell*, can do "man *command_name*" to bring up a man page on that command.

Extensive PDF documents at $synopsys*/doc/online*

synth/ directory contains all documents for synthesis tools.
Synt/dcrm has *dc_shell* reference manual.
Synth/dcug has *dc_shell* user guide. Both of these are good places to look for answers to questions about *dc_shell*.

dw/dwug has user guide for Design Ware (basic concepts, usage examples).
dw/dwdg has notes for creating custom libraries including tutorial.

Before you ask questions

- Have you looked at all of the files/examples ?
  - Have you looked inside the files and attempted to understand the particular VHDL or *dc_shell* commands being used?
  - Have you looked at the input files required by the script and output files produced by it?
- Have you looked at the Synopsys PDF documentation?
- Have you used the 'man' facility in dc_shell?