MICROPROCESSOR LABORATORY

African Regional Course on Advanced VLSI Design Techniques

24 November - 12 December 2003

Kumasi - Ghana

Exercise 4

Design of a 4 bit Presettable Synchronous Binary Counter using VHDL dataflow

Problem Description

In this example you will design a 4-bit presettable synchronous binary counter using VHDL dataflow. In this design example you will learn to:

- Specify the behaviour of the counter using VHDL and simulate it.
- Generate the structural description of the counter and simulate it.
- Place the necessary pads and re-simulate the structural description of the counter.
- Make the layout of the chip.
- Extract the circuit from the layout.
- Extract the behavioural description from the netlist and compare with the original behaviour file we created, to complete formal verification.

This design example consists of two phases. The first phase is to describe the behaviour of the chip as is seen at the pins of the chip. The second phase is to describe the functions of the core of the chip, and then connect it to the pads.

In the first phase you will:

- Describe the counter's behaviour using VHDL (counter.vbe).
- Write test patterns files.
- Simulate the behavioural description using the pattern file by using Asimut.

In the second phase you will:

- Describe the behaviour of the core in VHDL as is seen inside the chip by the pads (countcore.vbe).
- Synthesise the logic and structural descriptions using **Bop** and **Scmap** (countcorel.vst).
- Use **Glop** to optimise for critical path and fanout (countopt.vst).
- Use the standard cell router called **Scr** to place and route the core (countopt.ap).
- Add the necessary pads for the chip and compile using **Genlib** (countchip.vst).
- Use Asimut to simulate the 'countchip.vst' file using the pattern file 'counter.pat'.
- Place the pads and generate the layout of the chip with pads using **Ring** (countchip.ap).
- Use **Tas** to perform the static timing analysis.
- Use Lynx to extract the netlist from the layout file 'countchip.ap' (countchip.al).
- Use **Lvx** to compare the extracted circuit 'countchip.al' and the original 'countchip.vst' file created by **Genlib**.
- Use **Yagle** to extract the behaviour, 'countchip.vbe' from the 'countchip.al' netlist file.
- Use **Proof** to compare the extracted behaviour file, 'countchip.vbe' and the behavioural file created in the first phase, 'counter.vbe'.

1



Fig 1. Design Flow for the counter chip.

A 4-Bit Presettable Synchronous Binary Counter

The present exercise is a 4-bit presettable synchronous binary counter. The counter has an "enable" which when at logic '1' allows the counter to count. The counter is presettable to the value given on the input bus when "preset=1". The counter counts forward starting from this value. There is a synchronous reset, which puts the counter to zero when it is '0'.

A possible pin diagram of the counter is shown in Fig. 2.



Fig. 3 Counter chip (a possible pinout diagram).

СК	RESET	PRESET	ENABLE	COUT(3:0)
Rising Edge	0	Х	Х	0000
Rising Edge	1	1	Х	PIN(3:0)
Rising Edge	1	0	1	COUT(3:0) +1
Rising Edge	1	0	0	COUT(3:0)
No Rising Edge	Х	Х	Х	COUT(3:0)

Table 1. Truth Table for the 4-bit presettable counter



Course on VLSI Design Techniques.

Solution

Legend



Give the command that appears immediately after this symbol, at the command line.



Edit and save into a file, all that appears after this symbol.



Explanation of a topic

Set the environmental variables as shown immediately after this symbol.

Creating the Design

Begin by creating a design directory, at a convenient position in your work space:



mkdir counter

Change into this directory:



cd counter

Before starting the design you will have to set the environmental variables as shown below so that you will not run into problems later.



Create with the text editor a file called "counter.vbe". Enter the following and save the file.

```
-- Behavioural description using VHDL
-- 6th Workshop on VLSI Design - Trieste
ENTITY counter IS
PORT(
Vdd, Vss, Vdde, Vsse: in BIT;
Pin: in BIT_VECTOR (3 downto 0);
Cout: out BIT_VECTOR (3 downto 0);
ck: in BIT;
reset: in BIT;
enable: in BIT;
preset: in BIT;
```

ARCHITECTURE dataflow OF counter IS

SIGNAL count: REG_VECTOR (3 downto 0) REGISTER;

BEGIN

```
lcount : BLOCK(ck='1' and not ck'STABLE)
  BEGIN
       count <= GUARDED B"0000" when (reset ='0') else
       Pin
             WHEN (preset = '1') else
       B"0001" WHEN ((enable='1') and (count = B"0000")) else
       B"0010" WHEN ((enable='1') and (count = B"0001")) else
       B"0011" WHEN ((enable='1') and (count = B"0010")) else
       B"0100" WHEN ((enable='1') and (count = B"0011")) else
       B"0101" WHEN ((enable='1') and (count = B"0100")) else
       B"0110" WHEN ((enable='1') and (count = B"0101")) else
       B"0111" WHEN ((enable='1') and (count = B"0110")) else
       B''1000'' WHEN ((enable='1') and (count = B''0111'')) else
       B''1001'' WHEN ((enable='1') and (count = B''1000'')) else
       B"1010" WHEN ((enable='1') and (count = B"1001")) else
       B"1011" WHEN ((enable='1') and (count = B"1010")) else
       B"1100" WHEN ((enable='1') and (count = B"1011")) else
       B"1101" WHEN ((enable='1') and (count = B"1100")) else
       B"1110" WHEN ((enable='1') and (count = B"1101")) else
       B"1111" WHEN ((enable='1') and (count = B"1110")) else
       B"0000" WHEN ((enable='1') and (count = B"1111")) else
       count;
```

END BLOCK lcount;

Cout <= count;

 $rp \le count(0)$ and count(1) and count(2) and count(3);

end dataflow;

Test Pattern File and Simulation of the Behavioural Description

Write a pattern file for simulation.

(You can write a C file that when treated with **Genpat** will generate the pattern file for you. See exercise 3). Modify the pattern file if it is necessary by editing it and simulate using **Asimut** with the **-b** option and check that the counter performs satisfactorily.

Describing the core of the chip



The behavioural file "counter.vbe" is the description of the counter as is seen at the pins of the chip. We have not thought about the pads that drive the pins. When the chip is described physically in Alliance, it consists if two separate parts that are brought together, the core and the pads. In Alliance, the core and the pads are brought together in a C description file. This file when treated with **Genlib**, produces the structural description of the chip with the pads. In practice the core can be synthesised automatically form a behavioural description, whereas the pads should be placed physically, one by one in the C file. Placing the pads require the structural knowledge of the pads. One of the type of pads that is used in this example is the pck_sp clock pad, a cell of "padlib", a library of pads provided with Alliance.

Give the following command at the command line to see a description of this pad.



man pck_sp

Behavioural Description of the Core

Copy the file "counter.vbe" to the file "countcore.vbe", edit it and delete the Vdde and Vsse input signals since they are used only for the Pads.

Logic and Structural Synthesis of the Core

Now **Bop** and **Scmap** can be used to optimise and synthesise the core of the chip from the above behavioural description.

Give the command:

bop -o countcore countcorel

This takes as input the "countcore.vbe" description and creates an optimised behavioural description file "countcorel.vbe". Now to synthesise the structural description give the command:



scmap countcorel countcorel

This takes as input the optimised behavioural description "countcorel.vbe" and creates a structural description file "countcorel.vst" using the elementary components from the standard cell library.

Optimising for Fanout and Timing

The structural description created above has been created without worrying about the standard cells fanout limits and critical path signals. Alliance **Glop** can analyse the structural description and create a new description by adding buffers to the appropriate nets.

Give the command:

glop -g countcorel countopt -i -t

-	invokes timing optimization.
-	countcorel.vst input file
-	countopt.vst output file
-	gives fanout information about the gate netlist.
-	gives timing information about the gate netlist.
	- - - -

This command takes "countcorel.vst" structural description and generates a "countopt.vst" file after buffers have been added to the critical paths.

Give the command:

glop -f countopt countopt

-f	-	invokes fanout optimization.
countopt	-	countopt.vst modified structural file

This command should add buffers to the appropriate nets to resolve fanout problems and write over the "countopt.vst" file created above.

Placement and Routing of the core

The core can now be routed using **Scr**. Give the following command at the command line:

scr -p -r -l 4 -i 100 countopt

-p	-	placement option
-r	-	routing option
-14	-	asks to place and route the core in 4 rows
-i 100	-	use 100 iterations to improve placement quality

A "countopt.ap" layout file is created which can be viewed with **Graal**. Inspect it using **Graal**.

Describing the Pads and Core using the Procedural Design Language

The procedural description language is actually a set of C functions that allows you to describe circuit objects like pads and the core and their connectivity. Edit and save into the file "countchip.c" the following:

#include <genlib.h>

main() { int i;

DEF_LOFIG("countchip");

LOCON("ck", IN, "ck"); LOCON("reset", IN, "reset"); LOCON("preset", IN, "preset"); LOCON("enable", IN, "enable"); LOCON("vdd", IN, "vdd"); /* core power supply */ LOCON("vss", IN, "vss"); /* core ground LOCON("vdde", IN, "vdde"); /* pads power supply */ LOCON("vsse", IN, "vsse"); /* pads ground */ LOCON("PIN[3:0]", IN, "PIN[3:0]"); /* preset input */ LOCON("COUT[3:0]", OUT, "COUT[3:0]"); /* */ output LOCON("rp", OUT, "rp");

/*

power supplies: pxxxe_sp are external power supplies, i.e. used only by the buffers inside the pads.

pxxxi_sp are internal power supplies, for core logic only.
*/

LOINS ("pvsse_sp", "p30", "cki", "vdde", "vdd", "vsse", "vss", 0); LOINS ("pvsse_sp", "p31", "cki", "vdde", "vdd", "vsse", "vss", 0); LOINS ("pvdde_sp", "p32", "cki", "vdde", "vdd", "vsse", "vss", 0); LOINS ("pvdde_sp", "p33", "cki", "vdde", "vdd", "vsse", "vss", 0); LOINS ("pvssi_sp", "p34", "cki", "vdde", "vdd", "vsse", "vss", 0); LOINS ("pvddi_sp", "p35", "cki", "vdde", "vdd", "vsse", "vss", 0);

LOINS("pck_sp", "p0", "ck", "cki", "vdde", "vdd", "vsse", "vss", 0);

LOINS("pvssick_sp", "p1","clock", "cki","vdde", "vdd", "vsse", "vss", 0);

LOINS("pi_sp", "p2", "reset", "res", "cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("pi_sp", "p3", "preset", "pres", "cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("pi_sp", "p4", "enable", "en", "cki", "vdde", "vdd", "vsse", "vss", 0);

LOINS("po_sp", "p5", "rprp", "rp", "cki", "vdde", "vdd", "vsse", "vss", 0);

LOINS("po_sp", "p10","usc[0]", "cout[0]","cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("po_sp", "p11","usc[1]", "cout[1]","cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("po_sp", "p12","usc[2]", "cout[2]","cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("po_sp", "p13","usc[3]", "cout[3]","cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("pi_sp", "p20","pin[0]", "ingr[0]","cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("pi_sp", "p21","pin[1]", "ingr[1]","cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("pi_sp", "p22","pin[2]", "ingr[2]","cki", "vdde", "vdd", "vsse", "vss", 0); LOINS("pi_sp", "p23","pin[3]", "ingr[3]","cki", "vdde", "vdd", "vsse", "vss", 0);

LOINS("countopt", "core", "vdd", "vss", "ingr[3:0]", "usc[3:0]", "clock", "res", "en", "pres", "rprp", 0);

```
SAVE_LOFIG();
exit(0); /* necessary for the proper run of the Makefile */
}
```

Give the command at the command line:



genlib -v countchip

This creates the "countchip.vst" structural description file with pads.

Simulating the Structural Description

You can now simulate this structural description with the test vector file that you used for "counter.vbe". Simulate the structural description and confirm the functioning of the structural description.

Placing and routing the pads

Now the chip's pads and the core has to be connected together physically in a layout. This is done by using **Ring**.

Edit and save the following in the file "countchip.rin":

1101010		++++++	12 - 4	
旧社	hana	mmm	ťΟ	ш
HH	·ΨΨΨ	uuu	-B -	

width (vdd 20 vss 20)
west (p0 p32 p35 p33 p23)
south (p3 p2 p1 p4 p5)
north (p10 p30 p34 p31 p21)
east (p11 p12 p13 p20 p22)

This file describes the relative position of the pads on the four sides of the chip.

Give the command at the command line:



ring countchip countchip

The "countchip.ap" layout file is created that can be examined by using **Graal**. Examine the layout using **Graal**.

Static Timing Analysis

	The "countchip.ap" contains the layout information. However we do not know if the physical description produced reflect the desired behaviour. Therefore to check the layout we use the tools Lynx and Tas . Lynx is a netlist extractor. It extracts a netlist representation of the circuit from the layout. The file created by Lynx will be the input file for Tas . Tas is a switch level timing analyser for CMOS circuits. Give the following command at the command line:
	setenv MBK_OUT_LO al
	This tells that the output file should be in the ".al" (Alliance) format.
	lynx -v -t countchip
8.04	-v-verbose-t-build the netlist to the transistor level.first countchip-take the "countchip.ap" layout file as input.second countchip-generate the "countchip.al" netlist file.Give the following command at the command line:
	setenv MBK_IN_LO al
	This tells that the input file for Tas must be in the ".al" (Alliance) format.
	tas -tec=/alliance/archi/Linux_elf/etc/prol10.elp countchip
	-tec - selects the technology file prol10.elp.

Layout Extraction and Netlist Comparison

The "countchip.ap" contains the layout information. However we do not know if the physical description produced reflect the initial behavioural description. Therefore to check the layout we use two tools, Lynx and Lvx.

Lynx is a netlist extractor. It extracts a netlist representation of the circuit from the layout.

For this you have to set some environmental variables. You have to specify the format in which the extracted netlist is generated.

Give the following command at the command line:



setenv MBK_OUT_LO al

This tells that the output file should be in the ".al" (Alliance) format.

Give the command at the command line:



lynx -v -f countchip countchip

-	verbose
-	asks Lynx to generate the netlist at the Standard-
	cells level.
-	Take the "countchip.ap" layout file as input.
-	Generate the "countchip.al" netlist file.
	- - -

Lvx is a netlist comparison software that compares two netlists. Along with the comparison it re-orders the interface terminals to produce a consistent netlist interface.

Give the command at the command line



lvx vst al countchip countchip -f -o

vst	-	take the first file in .vst format.
al	-	take the second file in .al format.
first countchip	-	"countchip.vst" input file.
second countchip	-	"countchip.al" output file.
-f	-	build the netlist to the standard cell level.
-0	-	to have ordered connectors in the output netlist

The comparison should not produce any errors. If errors are produced by the program, then there is something wrong with the layout. The router has done something funny and corrective action is to be taken at the layout level by studying the error messages.

The Lvx has also re-ordered and built the netlist in the ".al" to the standard cell format. This file can be simulated using **Asimut**.

Simulating the Extracted netlist file

The netlist file "countchip.al" can be simulated using **Asimut** and the test vector file that has been created to test the behavioural file "counter.vbe". Give the following command at the command line:

setenv MBK_IN_LO al

to set the input file format for **Asimut** for the ".al" format, before doing the simulation. Any error during simulation means that you will have to retrace your steps back to find out the source of the error.

Functional Abstraction

Yagle is a program that extracts from several structural descriptions (man yagle), the behaviour of the circuit. Essentially a VHDL file is created from a standard cell connectivity description or from a SPICE transistor netlist! This VHDL file can be simulated in turn to verify the function of the chip

Give the command at the command line:



yagle -v countchip

-V	-	vectorize
countchip	-	Takes the "countchip.al" as input.

The extracted VHDL description is put in the file "countchip.vbe".

Simulate the extracted behavioural description to verify the extracted behavioural description.

Alliance has a program that compares the extracted behavioural file with the original behavioural file to formally prove the functional congruence of the described and the extracted circuit. However this step requires that the registers in the two behavioural descriptions have the same names. This can be done automatically by **Yagle** by giving it a list of registers to be renamed, in an information file "countchip.inf". If we do a "more" of the "countchip.vbe" file we see that the registers have a different name from the one that we have given in "counter.vbe".

Edit and save a file "countchip.inf" with the following:

rename		
<pre>core.count_1.dff_s : count_1</pre>	;	
<pre>core.count_0.dff_s : count_0</pre>	; ;	
<pre>core.count_2.dff_s : count_2</pre>	;	
<pre>core.count_3.dff_s : count_3</pre>	;	
end		

Give the command:

yagle -i -v countchip

i	-	asks	Yagle to read the "countchip.inf" file and rename the registers
			in the "countchip.vbe" file as given in the list.

-v - vectorize

Now a formal verification compares the *extracted* and the *original* behavioural descriptions. Give the command:

proof -p -d counter countchip

-p	-	negates the input and output signal expressions of
		the registers.
-d	-	display errors to screen.

If no errors are reported, then the two behavioural descriptions concur.

Real Technology Conversion

Up till now all the files describe the circuit only as symbolic cells. The foundry requires the layout of the chip, described in terms of rectangles and layers in the **gds** or the **cif** format. This can be done in Alliance, by using **S2r**.



setenv RDS_TECHNO_NAME /alliance/archi/Linux_elf/etc/prol10_7.rds
setenv RDS_OUT cif
setenv RDS_IN cif

This chooses the $1.0\mu m$ CMOS process, chooses the output form of the chip in cif format and, replaces the symbolic pads with their real equivalent.

Give the command:



s2r -cv countchip countchip

-с	-	deletes connectors at the highest hierarchy. (Use
		man to see full description)
-V	-	verbose mode on
first countchip	-	"countchip.ap" file as input
second countchip	-	"countchip.cif" file as output.

This completes the design of the counter chip.

