MICROPROCESSOR LABORATORY

African Regional Course on Advanced VLSI Design Techniques

24 November - 12 December 2003

Kumasi - Ghana

Exercise 5

Design of a 4 bit Adder Accumulator using VHDL Dataflow

Problem Description

In this example you will design a 4-bit binary adder accumulator using VHDL dataflow. In this design example you will learn to:

- Specify the behaviour of the adder using VHDL and simulate it.
- Generate the structural description of the adder and simulate it.
- Place the necessary pads and re-simulate the structural description of the adder.
- Synthesise 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 adder's behaviour using VHDL (adder.vbe).
- Write test pattern 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 (addercore.vbe).
- Synthesise the logic and structural descriptions using **Bop** and **Scmap** (addercorel.vbe & addercorel.vst).
- Use **Glop** to optimise for critical path and fanout (addopt.vst).
- Use the standard cell router called **Scr** to place and route the core (addopt.ap).
- Add the necessary pads for the chip and compile using **Genlib** (addchip.vst).
- Use **Asimut** to simulate the 'addchip.vst' file using the pattern file 'adder.pat'.
- Place the pads and generate the layout of the chip with pads using **Ring** (addchip.ap).
- Use **Tas** to perform the static timing analysis.
- Use **Lynx** to extract the netlist from the layout file 'addchip.ap' (addchip.al).
- Use **Lvx** to compare the extracted circuit 'addchip.al' and the original 'addchip.vst' file created by **Genlib**.
- Use **Yagle** to extract the behaviour, 'addchip.vbe' from the 'addchip.al' netlist file.
- Use **Proof** to compare the extracted behaviour file, 'addchip.vbe' and the behavioural file created in the first phase, 'adder.vbe'.

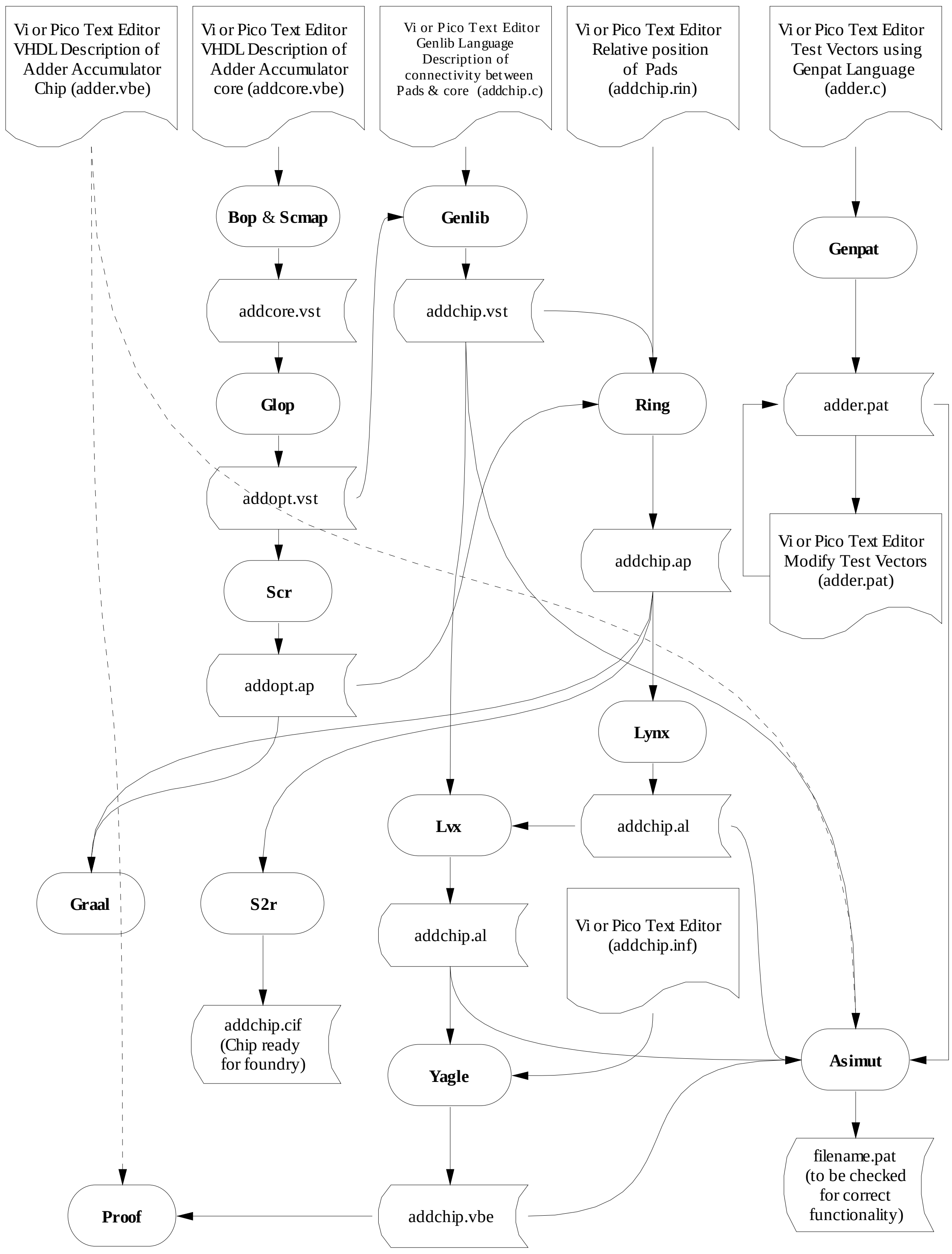
Vi or Pico Text Editor VHDL Description of Adder Accumulator Chip (adder.vbe)
Vi or Pico Text Editor VHDL Description of Adder Accumulator core (addcore.vbe)
Vi or Pico Text Editor Genlib Language Description of connectivity between Pads & core (addchip.c)
Vi or Pico Text Editor Relative position of Pads (addchip.rin)
Vi or Pico Text Editor Test Vectors using Genpat Language (adder.c)
Bop & Scmap
Genlib
Genpat
addcore.vst
addchip.vst
Glop
Ring
adder.pat
addopt.vst
addchip.ap
Vi or Pico Text Editor Modify Test Vectors (adder.pat)
Scr
addopt.ap
Lynx
Lvx
addchip.al
Graal
S2r
addchip.al
Vi or Pico Text Editor (addchip.inf)
addchip.cif (Chip ready for foundry)
Yagle
Asimut
filename.pat (to be checked for correct functionality)
Proof
addchip.vbe

Fig 1. Design flow of the adder accumulator chip

A 4-Bit Binary adder accumulator

The present exercise is a 4-bit binary adder accumulator. The adder has a “select” which when at logic ‘0’ allows the adder to sum the two inputs 4-bit buses: A and B, and when at logic '1' the input 4-bit bus A is added to the result stored in a 4-bit register which we call the accumulator. The accumulator is updated at the rising edge of the clock. The result of the sum is presented at the 4-bit output bus Y.

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

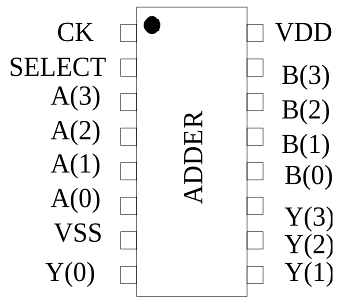

Fig 2. Adder chip (a possible pinout diagram).

CK	**SELECT**	**Y(3:0)**
Rising Edge	0	A+B
Rising Edge	1	A+Y
No Rising Edge	X	Y(3:0)

Table 1. Truth Table for the 4-bit binary adder

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 adder
```

Change into this directory:

```
cd adder
```

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

```
setenv MBK_CATA_LIB .:/alliance/archi/Linux_elf/cells/sclib:
                      /alliance/archi/Linux_elf/cells/padlib
setenv MBK_IN_LO vst
setenv MBK_OUT_LO vst
setenv MBK_IN_PH ap
setenv MBK_OUT_PH ap
setenv MBK_WORK_LIB .
```

Create with the text editor a file called “adder.vbe”. Enter the following and save the file.

```
ENTITY adder IS

PORT(
  vdd, vss, vdde, vsse : in BIT ;
  ck                   : in BIT ;
  sel                  : in BIT ;
  a                    : in  BIT_VECTOR (3 DOWNTO 0) ;
  b                    : in  BIT_VECTOR (3 DOWNTO 0) ;
  y                    : out BIT_VECTOR (3 DOWNTO 0)
  );

END adder;

ARCHITECTURE data_flow OF adder IS

SIGNAL regstr        : REG_VECTOR (3 DOWNTO 0) REGISTER;
SIGNAL mux           : BIT_VECTOR (3 DOWNTO 0) ;
SIGNAL sum           : BIT_VECTOR (3 DOWNTO 0) ;
SIGNAL carry         : BIT_VECTOR (2 DOWNTO 0) ;

BEGIN


  WITH sel SELECT

  mux      <=  b WHEN '0', regstr WHEN '1' ;

  sum(0)    <=  a(0) xor mux(0) ;
  carry(0)  <=  a(0) and mux(0) ;

  sum(1)    <=  a(1) xor mux(1) xor carry(0) ;
  carry(1)  <=  (a(1) and mux(1))     or
                (mux(1) and carry(0)) or
                (a(1) and carry(0)) ;

  sum(2)    <=  a(2) xor mux(2) xor carry(1) ;
  carry(2)  <=  (a(2) and mux(2))     or
                (mux(2) and carry(1)) or
                (a(2) and carry(1)) ;

  sum(3)    <=  a(3) xor mux(3) xor carry(2) ;


   store : BLOCK ((ck = '1') and not ck'STABLE)

         BEGIN
         regstr  <=  GUARDED  sum ;

         END BLOCK ;


   y <= regstr;

END;
```

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 "adder.vbe" is the description of the adder 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 of 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 from 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 types of pads that is used in this example is the pi_sp input pad, a cell of PAD-Lib, a library of pads provided with Alliance.

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

```
man pi_sp
```

Behavioural Description of the Core

Copy the file "adder.vbe" to the file "addercore.vbe", edit it and delete the Vdde and Vsse input signals since they are not necessary for the core.

Logic and Structural Synthesis of the Core

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

Give the command:

```
bop -o addercore addercorel
```

This takes as input the "addercore.vbe" description and creates an optimised behavioural description file "addercorel.vbe".

To synthesise the structural description give the command:

```
scmap addercorel addercorel
```

This takes as input the optimise behavioural description “addercorel.vbe” and creates a structural description file “addercorel.vst” using the 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 addercorel addopt –i –t
```

-g	-	invokes timing optimization.
-i	-	gives fanout information about the gate netlist.
-t	-	gives timing information about the gate netlist.

This command takes “addercorel.vst” structural description and generates a “addopt.vst” file after buffers have been added to the critical paths.

Give the command:

```
glop -f addopt addopt
```

This command should add buffers to the appropriate nets to resolve fanout problems and write over the “addopt.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 1000 addopt
```

-p	-	placement option
-r	-	routing option
-l 4	-	asks to place and route the core in 4 rows
-i 1000	-	use 1000 iterations to improve placement quality

A “addopt.ap” file is created which can be viewed with **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.
Create and edit and save into the file “addchip.c” the following:

```
#include <genlib.h>
main()
{
 DEF_LOFIG("addchip");
 LOCON("a[3:0]",'I',"a[3:0]");
 LOCON("b[3:0]",'I',"b[3:0]");
 LOCON("y[3:0]",'O',"y[3:0]");
 LOCON("sel",'I',"sel");
 LOCON("ck",'I',"ck");
 LOCON("vdde",'I',"vdde");
 LOCON("vsse",'I',"vsse");
 LOCON("vdd",'I',"vdd");
 LOCON("vss",'I',"vss");

   LOINS ("pvsse_sp", "Vss", "cki", "vdde", "vdd", "vsse", "vss", 0);
   LOINS ("pvdde_sp", "Vdd", "cki", "vdde", "vdd", "vsse", "vss", 0);
   LOINS ("pvssi_sp", "Vssi", "cki", "vdde", "vdd", "vsse", "vss", 0);
   LOINS ("pvddi_sp", "Vddi", "cki", "vdde", "vdd", "vsse", "vss", 0);

   LOINS("pi_sp","sl","sel","sl","cki","vdde","vdd","vsse","vss",0);

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

   LOINS("pvsseck_sp", "clkcore", "clkcore", "cki",
                                              "vdde", "vdd", "vsse", "vss", 0);

   LOINS("pi_sp","a0","a[0]","ina[0]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","a1","a[1]","ina[1]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","a2","a[2]","ina[2]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","a3","a[3]","ina[3]","cki","vdde","vdd","vsse","vss",0);

   LOINS("pi_sp","b0","b[0]","inb[0]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","b1","b[1]","inb[1]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","b2","b[2]","inb[2]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","b3","b[3]","inb[3]","cki","vdde","vdd","vsse","vss",0);

   LOINS("po_sp","y0","out[0]","y[0]","cki","vdde","vdd","vsse","vss",0);
   LOINS("po_sp","y1","out[1]","y[1]","cki","vdde","vdd","vsse","vss",0);
   LOINS("po_sp","y2","out[2]","y[2]","cki","vdde","vdd","vsse","vss",0);
   LOINS("po_sp","y3","out[3]","y[3]","cki","vdde","vdd","vsse","vss",0);

LOINS("addopt","adder1","vdd","vss","clkcore","sl","ina[3:0]","inb[3:0]","out[3:0]"
,0);

SAVE_LOFIG();
   }
```

Give the command at the command line:

```
genlib –v addchip
```

This creates a “addchip.vst” structural description file with pads.

Simulating the Structural Description

You can now simulate this structural description with the test vector file that you developed for “adder.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 “addchip.rin”:

```
north ( clk sl b0 b1 b2 b3 )
west ( a0 a1 vssi a2 a3 )
south ( y0 y1 clkcore y2 y3 )
east ( vdd vddi vss  )
```

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

Give the command at the command line:

```
ring addchip addchip
```

A “addchip.ap” file is created that can be examined by using **Graal**.

Static Timing Analysis

The “addchip.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 two 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 analyzer 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 addchip addchip
```

-v	-	verbose
-t	-	build the netlist to the transistor level.

first addchip	-	take the “addchip.ap” layout file as input.
second addchip	-	generate the “addchip.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 addchip
```

-tec	-	selects the technology file prol10.elp.

Layout Extraction and Netlist Comparison

The “addchip.ap” contains the layout information. However we do not know if the physical description produced reflect the 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 addchip addchip
```

-v	-	verbose
-f	-	asks Lynx to generate the netlist from the Standard- cells level.
first addchip	-	Take the “addchip.ap” layout file as input.
second addchip	-	Generate the “addchip.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 addchip addchip –f –o
```

vst	-	take the first file in .vst format.
al	-	take the second file in .al format.
first addchip	-	“addchip.vst” file.
second addchip	-	“addchip.al” file.
-f	-	build the netlist to the standard cell level.
-o	-	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 some problem 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.

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 “addchip.al” can be simulated using **Asimut** and the test vector file that has been created to test the behavioural file “adder.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 a standard cell level, the behaviour of the circuit. Essentially a VHDL file is created from a standard cell connectivity description! This VHDL file can be simulated in turn to verify the function of the chip

Give the command at the command line:

```
yagle -v addchip
```

-v	-	vectorize
addchip	-	Takes the “addchip.al” as input.

The extracted VHDL description is put in the file "addchip.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 "addchip.inf". If we do a "more" of the "addchip.vbe" file we see that the registers have a different name from the one that we have given in "adder.vbe".

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

```
rename
adder1.regstr_0.dff_s : regstr_0 ;
adder1.regstr_1.dff_s : regstr_1 ;
adder1.regstr_2.dff_s : regstr_2 ;
adder1.regstr_3.dff_s : regstr_3 ;
end
```

Give the command:

```
yagle -i -v addchip
```

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

Give the command:

```
proof -p -d adder addchip
```

-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.

Give the command:

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

This chooses the 1.0μm 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 addchip addchip
```

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

This completes the design of the counter chip.