Digital ASIC Design
A Tutorial on the Design Flow

Digital ASIC Group

October 20, 2005

Lund University

Contents

Preface

When buying a book on hardware design, the focus is often limited to one area. It could be on signal processing, system level design, VHDL and other programming languages or arithmetic. In this manual, we will try to describe the design flow from developing code to chip layout, see Figure 1. The manual is divided into the following main sections:

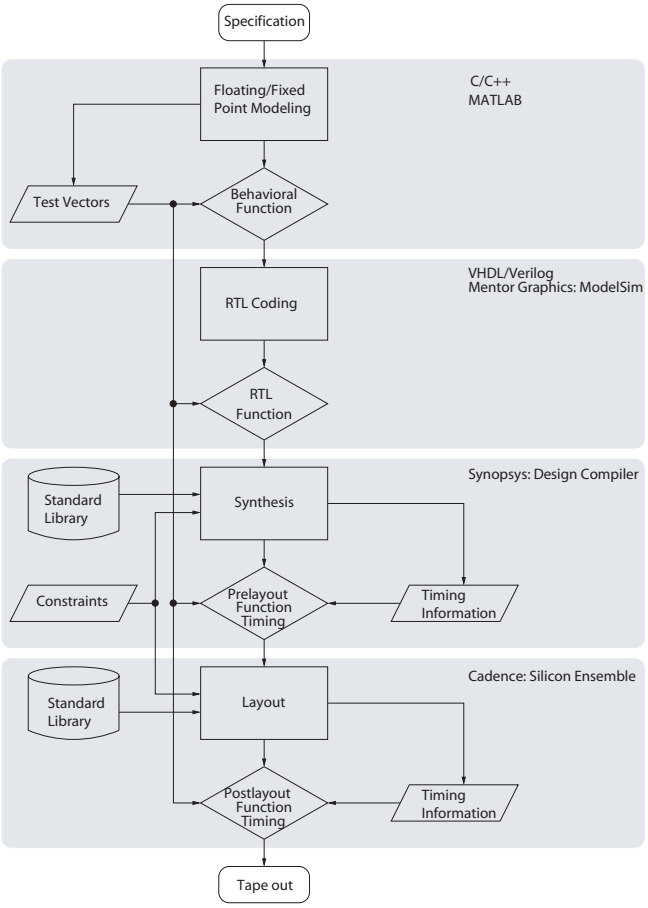


Figure 1: Common digital ASIC design flow.

VHDL

Chapter 1 presents an introduction to VHDL and the basic concepts of describing and simulating hardware. This is however not a complete reference to the language itself, more an overview and a selection of the most important parts of the language. Chapter 2 presents a strucured way of writing and using VHDL. The design methodology described here is developed by Jiri Gaisler at Chalmers. This chapter also describes how to develop platform independent code, supporting several technologies such as programmable logic and various cell libraries.

Arithmetic

The chapter on arithmetic describes how to infer a component for an arithmetic operation in a digital design. The most commonly used arithmetic operations are described and the connection between the VHDL description and the final hardware structure realized during synthesis is illustrated. This chapter gives the reader an overview on how the determine a highly effective hardware structure by coding and controlling the synthesis tool.

Memories

One of the most important topics in digital ASIC design today is memories. Not only the amount of memory but also the memory hierarchy, including caches and off-chip memories, has to be considered. Each memory hierarchy should be optimized for high speed, low power, small area or a combination of these, depending on the application. However, this is a too large topic to cover in this compendium. Instead three different designs are presented in this Chapter in order to show possible optimizations to various memory structures.

Synthesis

This chapter gives a overview of the whole synthesis process, and covers the basic synthesis concepts and guidelines for design optimization. It starts with the basic concept of what synthesis is, followed by environment setups for synthesis, and wireload model basics. In order to better understand the effect of setting constraints, the synthesis process is described. At last, VHDL coding style is mentioned as a guideline for better synthesis results.

Place & Route

Place and Route is the final step before sending the design for fabrication. The standard cells determined during synthesis are placed automatically on the cell core. Location of the pads and placement of the on-chip memory has to be defined and carried out by the user. The core supply, power and ground rings, are added. A design-rule check (DRC) verifies that the fabrication constraints are met and finalizing your design for fabrication.

Power Estimation

This chapter gives an introduction to power estimation and optimization techniques in an ASIC design flow with Synopsys **Power Compiler**. After a short review of the sources of power consumption in a digital circuit, tool-independent optimization techniques are presented for different abstraction levels. It is also shown how the design tool interacts with information from the cell library and external simulation tools to estimate and optimize power at gate level. For those who like to probe further, additional information is found in the reference list at the end of the chapter.

Chapter 1

Introduction to VHDL

This chapter is a brief introduction to hardware design using a Hardware Description Language (HDL). A language describing hardware is quite different from C, Pascal, or other software languages. A computer program is dynamic, i.e., sharing the same resources, allocating resources when needed and not always optimized for maximum speed, optimal memory management, or lowest resource requirements. The main focus is functionality, but it is still not uncommon that software programs can behave quite unexpected. When problems arise, new versions of the programs are distributed by the vendor, usually with a new version number and a higher price tag.
The demands on hardware design are high compared to software. Often it is not possible, or at least very tricky, to patch hardware after fabrication. Clearly, the functionality must be correct and in addition how the code is written will affect the size and speed of the resulting hardware. Each mm^2 of a chip costs money, lots of money. The amount of logic cells, memory blocks and input/output connections will affect the size of the design and therefore also the manufacturing cost. A software designer using a HDL has to be careful. The degrees of freedom compared with software design have dramatically increased and must be taken into account.
The focus of this chapter is not to present VHDL in detail. A programming language often specifies more functionality than actually needed. This is especially true for VHDL, since it is only a small part of the program language that can be synthesized to hardware. In this chapter, the basics parts of VHDL programming that can be used in hardware design will be presented. In the next chapter, a design methodology is applied and a structured way of writing VHDL will be presented. There are numerous available books that addresses VHDL coding and to probe further, the book written by J. Bhasker [1] is recommended as first choice. Another book worth mentioning is written by P. J. Ashenden [2] and is used at many Universities.

1.1 Background

VHDL is a acronym which stands for Very High Speed Integrated Circuit Hardware Description Language. VHDL has been standardized by IEEE and the most widely used version is called VHDL'87 (1079-1987). In 1994, an updated

version called VHDL'93 introduced new functionality and a more symmetric syntax. However, using the VHDL'87 syntax guarantees compatibility with both old and new synthesis and simulation tools. A recommendation is to use VHDL'87 syntax for synthesizable code, while testbenches and other non-synthesizable parts can be written using VHDL'87/93. Although being a hardware description language, VHDL still supports file access, floating point, and complex numbers, and other features not directly associated with hardware design. This comes in handy when writing reference designs or testbenches, but is not even close to being synthesizable, i.e., transferred into physical gates and how they are connected which is equivalent to a gate-level netlist or simply called a netlist.

1.2 Event-driven simulation

A computer program running on a microprocessor executes one instruction at a time. Therefore, it is easy to debug and understand the behaviour of the program. A compiler translates the source code into machine language and executes the instructions sequentially. Hardware is by nature concurrent, i.e., statements are executed in parallel depending on the chosen architecture. Instead of compiling the VHDL code into a sequential program, it is *executed* using an event-driven simulator. This simulator understands the concept of time and delay. The smallest time quantum is called delta (Δ), which is the time it will take to assign a value to a signal. When an external event triggers the system, the timescale is incremented with Δ until all signals are stable, i.e. until there are no more events in the system to process. The external event could be, e.g., the master clock, updating the system registers on a rising or falling edge. Figure 1.1 shows an example of event driven simulation. It is shown that it takes two (Δ) to propagate signals from input to output in this design.

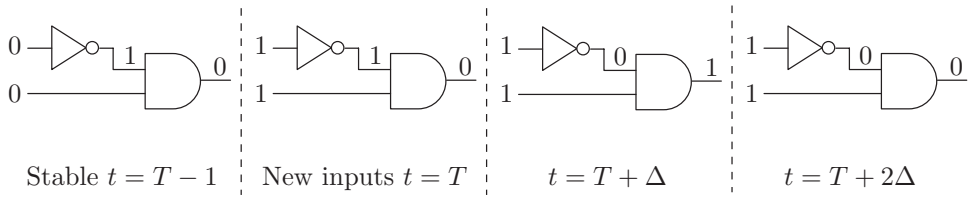


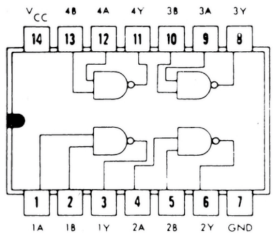
Figure 1.1: Event-driven simulation of a small design.

1.3 Design units

This chapter will present the five design units in VHDL. The **entity**, describing the interface between building blocks. The **architecture**, describing the structure or behaviour of a block. The **configuration** associating an architecture with an entity. The **component** that is used to include a building block as part of a new building block. Finally, **packages** to store common or global information.

1.3.1 Entity

The concept of VHDL coding can be compared to building a design using discrete logical circuits and wires. The once so popular $74xx$ circuits are series of TTL logic blocks with different logical functions, Boolean logic, inverters, binary counters and flip-flops. They come in small packages with a number of pins connected to the outside world. This can be compared to an **entity** in VHDL. The entity describes the interface, the input signals, and the output signals, without any information about the functionality. In Figure 1.2 the interface of a TTL circuit is described in VHDL. All signals except for power and ground are declared in the entity together with signal direction, i.e., input or output. Power and ground signals will not be part of the design flow until the design is placed and routed, see Chapter 6 for more details.




```
entity TTL is
  port( A1,B1 : in  std_logic;
        ...
        A4,B4 : in  std_logic;
        Y1    : out std_logic;
        ...
        Y4    : out std_logic
      );
end;
```

Figure 1.2: The entity describes the interface of the block.

1.3.2 Architecture

Every entity has one or several **architectures** describing the behaviour of the block. For example, the circuit in Figure 1.2 contains 4 NAND gates. However, using the same entity but replacing the architecture with a description of 4 NOR gates changes the behaviour but not the interface to the block.

```
architecture OR of TTL is
begin
  Y1 <= A1 or B1;
  Y2 <= A2 or B2;
  Y3 <= A3 or B3;
  Y4 <= A4 or B4;
end;
```

```
architecture NAND of TTL is
begin
  Y1 <= not (A1 and B1);
  Y2 <= not (A2 and B2);
  Y3 <= not (A3 and B3);
  Y4 <= not (A4 and B4);
end;
```

1.3.3 Configuration

A **configuration** specifies which architecture to use for a certain instantiation of the entity. Normally, there is only one architecture for each entity, and in this case no configuration is needed. However, it is useful to write a configuration for post-synthesis simulations, selecting and simulating the gate-level netlist generated by the synthesis tool. This way, it is possible to create several configurations to simulate before synthesis, after synthesis, and finally after place and route. To use two architecture to describe different behaviourals of the same entity, as shown in Section 1.3.2, is not commonly used, since it is not always supported in the synthesis tool. However, to use it in the testbench is okay, since it is not synthesized.

```
configuration cfg_pre_syn_adder of tb_adder is      -- configuration before synthesis
  for behavioural                                   -- testbench architecture name
    for iadd: adder                                 -- DUT instantiation name
      use entity WORK.ADDER(STRUCTURAL);            -- point to behavioural adder
    end for;
  end for;
end;

configuration cfg_post_syn_adder of tb_adder is     -- configuration after synthesis
  for behavioural                                   -- testbench architecture name
    for iadd: adder                                 -- DUT instantiation name
      use entity WORK.ADDER(SYN_STRUCTURAL);        -- point to synthesized adder
    end for;
  end for;
end;
```

If no configuration exists, a *default architecture* will be selected. The default architecture is selected by the simulation or synthesis tool, taking the most recent compiled architecture with the exact same name and port description as the entity. If no such architecture exists, the entity will be linked to an empty *black box*. This could be used for inserting components in the empty space after synthesis.

1.3.4 Component

Component declarations is used to include a pre-made design as part of a new design. To be able to use the pre-made design in the new design it has to be declared in the code as a component. The component declaration is either included in a package or placed in the architecture part of the code before begin. For example, to include the previously mention TTL design in a new larger design, the code could look like this:

```
entity larger_design is
  port( A : in  std_logic;
        Y : out std_logic);
end;

architecture my_first_arch of larger_design is

component TTL
  port( A1,B1 : in  std_logic;
        ...
        A4,B4 : in  std_logic;
        Y1    : out std_logic;
        ...
        Y4    : out std_logic
      );
end comonent;

begin
  -- use TTL here
end;
```

1.3.5 Package

A package can be used to store global information such as constants, types, component declarations or subroutines/functions. A package in VHDL is like the header files used in C. The package is separated in a declaration and a body. The body contains implementations of the functions in the declaration part.

```
package MY_STUFF is
  constant ZERO : std_logic := '0';
  constant ONE  : std_logic := '1';
  function invert(v : std_logic_vector) return std_logic_vector;
```

```
end;

package body MY_STUFF is
  function invert(v : std_logic_vector) return std_logic_vector is
    variable result : std_logic_vector((v'length)-1 downto 0);
  begin
    result := not v;  -- Input v is returned bit wise inverted
    return result;    -- from the function invert
  end;
end;
```

The package can be included at the beginning of a VHDL file using

```
library MY_LIB
use MY_LIB.my_stuff.all;
```

1.4 Data types and modelling

After describing the design units, the next step is to connect the building blocks. The entity describes the interface of each block and these interfaces are connected using **signals**. Signals are like wires on a PCB (Printed Circuit Board), routing the traces between the individual chips. Signals are connected to the building blocks with port maps that specifies which input/output is connected to which signal. This chapter also presents **generics**, specified in the entity, which makes it possible to pass constant values to a module and to dynamically configure the wordlength of input and output signals in the port map. Finally, **constants** and **variables** will be presented.

1.4.1 Type declaration

A constant, variable, signal, or generic can be of any type. To simplify this section, only the most useful types will be described. Types and subtypes can be declared in the architecture, or in a package if the types are going to be used between entities. Subtypes are limited sets of a type, e.g., the range 0 to 7 is a subtype of the type integers. There are two reasons to use subtypes; One, to give informative names to the subtypes and secondly, the simulator will raise a warning if values exceeds the subtype range. Integers and enumerated data types are built-in (package standard), while **std_logic** is defined in a package standardized by **IEEE**. Records are structures containing several values of any type, and will be further discussed in the next chapter about design methodology.

```
library IEEE;
use IEEE.std_logic_1164.all;                          -- std_logic package

architecture ... of ... is
  type state_type is (IDLE, START, STOP);             -- enumeration
  subtype byte is integer range 0 to 255;             -- 8 bit integer
  subtype data_type is std_logic_vector(7 downto 0);  -- 8 bit vector
  type reg_type is record                             -- define record
    state : state_type;                               -- using types above
    data  : data_type;
  end record;
begin
```

Type	Description	Assigning	Representation
integer	numbers	i := 32	32-bit number max
std_logic	a single bit	v := '0'	U,X,0,1,Z,W,L,H,-
std_logic_vector	bit vector	v := "00"	U,X,0,1,Z,W,L,H,-
enumeration	set of names	s := IDLE	any legal name
record	set of types	r.state := IDLE	

1.4.2 Signals

Signals can be viewed as physical connections between blocks and inside modules. Signals are used as interconnects between ports and are assigned a value using the <= operator. Inputs and outputs described in the entity are also signals but with a predefined direction. The value of an input signal can only be read, and an output signal can only be assigned a value. For signals declared in the architecture body, the direction is not specified.

```
entity INVERTER is
  port( x : in  std_logic;
        y : out std_logic;
       );
end entity;

architecture behavioural of INVERTER is
signal internal : std_logic;              -- internal signal, no direction is specified
begin
  internal <= '0' when x = '1' else '1';  -- using WHEN statement
  y <= not x;                             -- boolean function
end;
```

1.4.3 Generics

One important feature in VHDL is the possibility of building generic components, i.e., creating dynamic designs. For example, when designing a component for adding two values, it would be inconvenient if the number of bits of the input operands were static values. If the adder component is reused in several places in the design, a static approach would force the designer to create several versions of the same component with various wordlengths. Therefore, the entity can be described using generic values in addition to the interface. The generic value of a specific instantiation is given a value with the generic map command that is similar to the port map command. The value of a generic parameter can be used in the port specification of the entity as well as in the architecture.

```
...

architecture struct of ALU is

component adder                                          -- component declaration
  generic( WIDTH : integer );                            -- generic parameter
  port( a : in  std_logic_vector(WIDTH-1 downto 0);      -- signals controlled by
        b : in  std_logic_vector(WIDTH-1 downto 0);      -- generic parameter
        z : out std_logic_vector(WIDTH   downto 0)
       );
end component;

begin
signal c,d : std_logic_vector(8 downto 0);
signal q   : std_logic_vector(9 downto 0);

adder_1 : adder generic map(WIDTH => 9) port map(a => c, b => d, z => q);
-- WIDTH is given the value 9 and the signals c, d, and q are connected to
-- the in and out ports of the adder instantiation adder_1.

...
```

1.4.4 Constants

It is often useful to define constants instead of hard-coding values in the VHDL code. If the constant names are chosen with care, e.g., **MEM_ADDR_BITS** to describe the width of an address bus to a memory, it will improve the readability and simplify the maintenance of the code. Constants can be declared in a package or in the architecture. Constants declared in the architecture are only local, while constants defined in a package can be used as global values.

```
package my_constants is                                      -- in package
  constant GND   : std_logic := '0';                         -- logical zero
  constant VDD   : std_logic := '1';                         -- logical one
  constant MAGIC : std_logic_vector(3 downto 0) := "1010";   -- constant vector
end;
```

```
architecture arch_counter of counter is                      -- in architecture
  constant MAXVAL : std_logic_vector(3 downto 0) := "1111";  -- all one vector
begin
  ...
end;
```

1.4.5 Variables

Variables can be used to temporarily store values in a process. Assigning a value to a variable has a different syntax than for a signal, using the := operator. When discussing design methodologies, we will show how variables can be used to avoid signal assignment and to change the concurrent behaviour into sequential execution. The advantage of variables is that they are sequentially executed since a value is assigned immediately, as in software programs. From a simulation point of view, it is also faster to work with variables than signals. Normally, variables are only used inside a process, but in later versions of VHDL it is also possible to declare shared variables directly in the architecture.

It is important to note that the hardware will be the same whether variables or signals are used! The hardware will not be sequential, it is only the code that is executed sequentially. Variables are used to increase the readability of the code and to reduce simulation time. Below is an example of the differences using variables and signals, but the hardware will be exactly the same in both cases. The process to the right uses only signals, which all change values at the same time. The first time this process is executed due to a change in z, signals a will become $z+1$ and b will become $3+1$, the old value of a plus one. These changes will trigger the process one more time, since a and b are in the sensitivity list of the process. The second time correct values will be assigned to b and q. The process to the left that uses variables, will assign correct values to a, b, and q the first and only time the process is triggered by a change in z. For more details of processes see Section 1.5.

```
process(z) -- sensitive to changes on z
  variable a,b integer;
begin    -- assume that z=5 then
  a:=z+1;  -- a=6
  b:=a+1;  -- b=7
  q<=b;    -- signal q=7 after one delta
end process;
```

```
process(a,b,z) -- sensitive to a, b, z
begin    -- assume that z=5 and a=3 then
  a<=z+1;  -- a=6 after 1 delta
  b<=a+1;  -- b=4 after 1 delta and b=7 after 2 delta
  -- changes to a and b will trigger process again
  q<=b;    -- q=4 after 1 delta and 7 after 2 delta
end process;
```

1.4.6 Common operations

Table 1.1 shows a list of common operators and which data types that can be used with them. All logical operations, e.g., **and** and **or**, are performed bit wise and both input most be of the same length. The input to comparison operations do not have to be of the same length, but the smallest input will be extended to the same length as the widest input in the hardware. For wordlengths concerning arithmetic operations see Chapter 3.

1.5 Process statement

Processes are used to increase code readability, reduce simulation time, and for the synthesis tool to recognize some common hardware blocks, e.g., flip-flops and latches. A hardware architecture can contain an arbitrary amount of processes. However, to increase code readability it is recommended that a maximum of two processes are used, one containing sequential logic and one containing combinatorial logic, e.g., flip-flops and latches.

A process can be viewed as a self-contained module, sensing changes in the input signals and changing the output signals accordingly. A process has a **sensitivity list** which reacts to changes in the input signals. If a signal in the sensitivity list changes, all statements in the process will be executed once. Therefore, it is very important to add all dependent signals, i.e., signals that are being read in the process, to the sensitivity list. Otherwise, the simulation result will not be correct.

A process without a sensitivity list is executed continuously and has to contain some kind of **wait** statement. Otherwise, the simulation tool will lock up. A **wait** statement could for example be *wait until sel='1'*, continuing execution when the condition is met. A simple MUX is presented below. The process is sensitive to changes to the input operands **a** and **b** and the select signal **sel**.

```
entity mux is
  port( a   : in  std_logic;
        b   : in  std_logic;
        sel : in  std_logic;
        q   : out std_logic
       );
end entity;

architecture behavioural of mux is
begin
  process(a,b,sel)                    -- sensitive to changes on A, B, SEL
  begin
    if (sel = '1')
      q <= a;
    else
      q <= b;
    end if;
  end process;
end;
```

1.5.1 Sequential vs. combinatorial logic

All the examples so far have demonstrated combinatorial circuits, for example Boolean functions, adders or multiplexers. Normally, the combinational blocks are separated by sequential logic, updated only on the rising or falling edge of the clock signal, see Figure 1.3. By combining sequential and combinatorial logic, it is possible to describe state machines, counters, and other sequential units. The accumulator in Figure 1.4 uses a sequential unit to store the sum of all previous input values. It uses a feedback signal to add the current input to the accumulated sum. The reset signal forces the sequential unit back to zero. The code example below shows the implementation of the accumulator, using one process to describe the combinatorial logic and one process for the sequential unit. The command $a <= (others =>' 0')$ is used to assign zero to all bits in a, independently of the wordlength of a.

```
entity ACCUMULATOR is
```

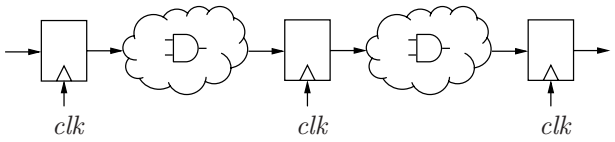


Figure 1.3: Combinatorial logic divided by sequential registers.

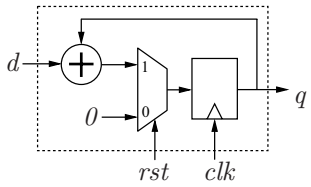


Figure 1.4: Simple accumulator with reset. The adder and MUX are the combinatorial part of the circuit. The register is sequentially updated by the clock signal.

```
  generic( BITS : integer );
  port( clk : in  std_logic;
        rst : in  std_logic;
        d   : in  std_logic_vector(BITS-1 downto 0);
        q   : out std_logic_vector(BITS-1 downto 0));
end;

architecture MY_GOD_I_HOPE_THIS_WORKS of ACCUMULATOR is
  signal sum, feedback : std_logic_vector(BITS-1 downto 0);
begin

  comb: process(rst, d, feedback)                -- combinatorial update
  begin
    if (rst = '0') then
      sum <= (others => '0');                    -- assign '0' to all vector elements
    else
      sum <= feedback + d;
    end if;
  end process;

  seq: process(clk)                              -- sequential update
  begin
    if rising_edge(clk) then                     -- update on the rising clock edge
      feedback <= sum;
    end if;
  end process;

  q <= feedback;

end;
```

1.6 Instantiation and port maps

Hardware design is well suited for a hierarchical approach. Small logical cells build arithmetic units such as adders. Adders are basic building blocks in signal processing, and several signal processing algorithms can form a complete system. This way of designing is called bottom-up, first specifying the smallest and most fundamental building block, then using (instantiating) these basic blocks to build more complex designs. Using previously designed components in a new component is called *instantiation*. For example, an FIR filter instantiates a number of adders and multipliers and connects wires between the instantiated blocks. The procedure for instantiation is presented below. Examples of how to instantiate components can be found in Section 1.9.

1. Create a new entity **adder**, as shown in Section 1.3.1.
2. Specify the **adder** architecture, as shown in Section 1.3.2.
3. Create a new entity and architecture to the **FIR**.
4. Include a component declaration of the adder, in the **FIR** architecture (before begin).
5. Instantiate the **adder** in the **FIR** architecture (after begin)
6. Connect the signals in the port map to create a filter function

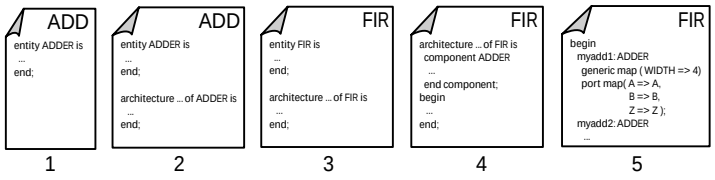


Figure 1.5: FIR filter instantiating a generic adder component.

1.7 Generate statements

When building highly regular structures, for example a direct map FIR filter, instantiating components by hand would be a cumbersome procedure. In the case of an FIR filter, the number of components depends on the actual filter size. Changing the filter size would require manual work to instantiate a different number of components. However, VHDL has a construct to cope with the situation of dynamic instantiation. The *generate* statement, similar to the *for* statement, can be used for instantiating blocks or connecting signals using the loop counter as index.

A frequently used signal processing element is the CORDIC (Coordinate Rotation Digital Computer). It performs a series of pseudo-rotations based on shift and add operations, useful for evaluating trigonometrically functions. An N bit CORDIC element can be implemented as a time multiplexed unit performing N iterations or by pipelining N units, as shown in Figure 1.6. The

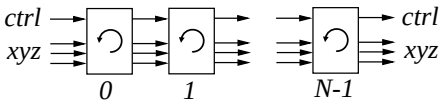


Figure 1.6: A pipelined CORDIC element with N bit precision.

example in this section shows how to instantiate N units and connecting them in a pipeline.

```
architecture structural of cordic is

  subtype data_type is std_logic_vector(N-1 downto 0);

  type cordic_data is record
    ctrl    : cordic_control_type;
    x, y, z : data_type;
  end record;

  type cordic_pipeline is array(0 to N) of cordic_data;

  signal pipeline : cordic_pipeline;

begin

  gen_pipe: for i in 0 to N-1 generate
    inst_unit: cordic_unit
      generic map( BITS  => N,
                   STAGE => i )
      port map( clk    => clk,
                rst    => rst,
                ctrli  => pipeline(i).ctrl,
                ctrlo  => pipeline(i+1).ctrl,
                xi     => pipeline(i).x,
                yi     => pipeline(i).y,
                zi     => pipeline(i).z,
                xo     => pipeline(i+1).x,
                yo     => pipeline(i+1).y,
                zo     => pipeline(i+1).z );
  end generate;

end;
```

1.8 Simulation

1.8.1 Testbenches

When writing a design, it is important to verify its functionality. The most common method of doing this is to create a testbench, i.e., instantiating a **DUT** (Device Under Test), generate test vectors (a set of inputs), and monitor the output, as shown in Figure 1.7. Common testbench tasks are to generate clock and reset signals, and read/write information to file. Writing the output values to a file makes it possible to verify the result using, e.g., **Matlab**. An example testbench is found below.

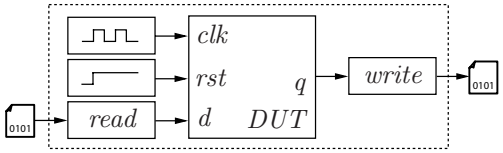


Figure 1.7: Testbench providing stimuli to a device under test (DUT).

```
library IEEE;
use IEEE.std_logic_1164.all;
use std.textio.all;

entity tb_exmple is
end tb_exemple;

architecture behav of tb_example is
  file in_data:text open read_mode is "../stimuli/input";
  file out_data:text open write_mode is "../stimuli/output";

  constant half_period: time :=10 ms;   -- half period = half clock period

  component test_design
      port (clk   : in  std_logic;
            rst   : in  std_logic;
            input : in  std_logic_vector(1 downto 0);
            output: out std_logic_vector(1 downto 0));
  end component;

  signal clk: std_logic:='0';           -- start value of clk is '0'
  signal rst: std_logic:='1';
  signal input_sig, output_sig : std_logic_vector(1 downto 0);

begin

  rst <= '0' after 20 ms;

  DUT : test_design port map (
    clk   => clk,   -- device pin (clk) connected with (=>) signal in tb (clk)
    rst   => rst,
    input => input_sig,
    output=> ouput_sig);

  process(clk)
  variable buffer_in,buffer_out:line;
  variable d,result :integer;
  begin
  if (clk='1') and (clk'event) then  -- at each positive clock edge a new input
    readline(in_data,buffer_in);     -- is written to the DUT and the output is
    read(buffer_in,d);               -- written to the output file
    inp_sig <= CONV_STD_LOGIC_VECTOR(d,2) after 2 ns; -- delay input 2 ns

    result:=CONV_INTEGER(output_sig);
    write(buffer_out,result);
```

```
    writeline(out_data,buffer_out);
  end if;
  end process;

  process --clock generator
  begin
    wait for half_period;
    clk <= not clk;
  end process;
end;
```

1.8.2 Modelsim

Modelsim is a widely used simulator for VHDL and Verilog-HDL. The screenshots from the program in Figure 1.8 and Figure 1.9 show the simulation of the accumulator example from Section 1.5.1. Simulation is performed by compiling a VHDL file using the Modelsim command *vcom*, loading the design with *vsim* and finally starting the simulation with the *run* command. All commands can either be written at the prompt or found in the GUI. The waveform window shows how the output value is incremented with *d* for each clock cycle.

```
ModelSim> vcom -work MYLIB accumulator.vhd
ModelSim> vsim MYLIB.accumulator
ModelSim> run 1000 ns
```

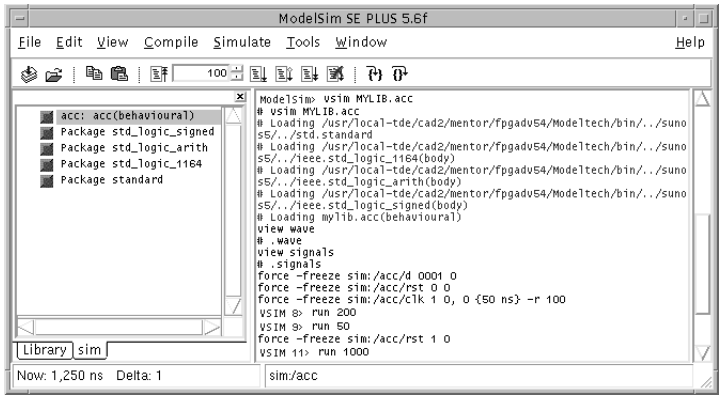


Figure 1.8: Modelsim main window. Can be used for compiling and simulating designs.

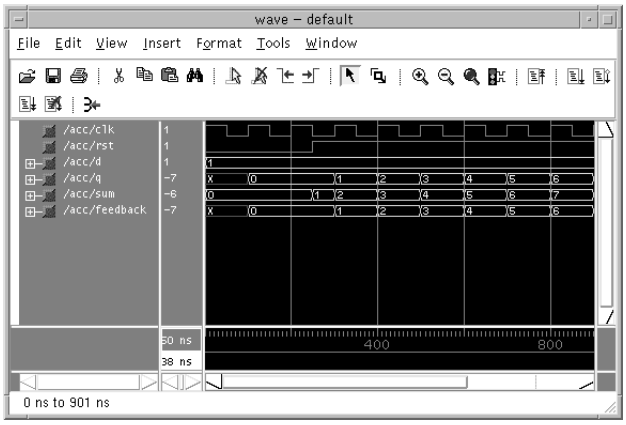


Figure 1.9: Modelsim waveform window. The accumulated values increases with d for each clock cycle.

1.8.3 Non-synthesizable code

Sometimes it is convenient to include functions that are non-synthesizable during simulation. It could be printing a message in the simulation window, or writing important information to a file. During synthesis, these lines of code must be discarded. This can be done by placing a special line before and after the non-synthesizable code. Place the line

```
-- pragma translate_off
```

before, and

```
-- pragma translate_on
```

after the part you want to hide from the synthesizing tool. When using **Synopsys**, a system variable must also be set in order for **Synopsys** to understand the `translate_off` and `translate_on` commands. This is done by setting the following variable in the synthesis script:

```
hdlin_translate_off_skip_text = true
```

An example of non-synthesizable code is **signal delays**. During simulation, it is possible to delay the signal assignment to emulate the behaviour of gates, memories or external interfaces. However, most synthesis tools understand this syntax and will simply ignore it; generating a warning message. It is not necessary to use *translate_off* when adding signal delays.

1.9 Design example

A small design example of an ALU (arithmetic logical unit) will be presented in this section. The ALU supports addition, subtraction, and some logical functions. There are two input operands, input control signal, and one output for the ALU result. The architecture for the design example is found in Figure 1.10.

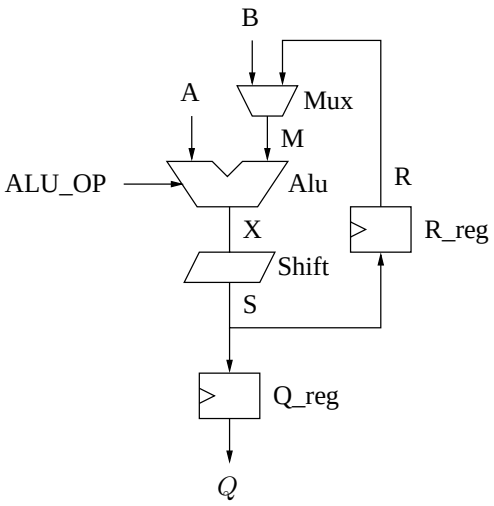


Figure 1.10: Schematic view of the ALU design.

ALU unit

```
entity ALU is
  generic (WL  :     integer);
  port( ALU_OP : in  std_logic_vector(2 downto 0);
        A, M   : in  std_logic_vector(WL-1 downto 0);
        X      : out std_logic_vector(WL-1 downto 0));
end;
```

```
architecture BEHAVIOURAL of ALU is
begin
  process (A,ALU_OP,M)
  begin
    case ALU_OP is
      when "000" => X <= A;
      when "001" => X <= A+M;
      when "010" => X <= A-M;
      when "011" => X <= M-A;
      when "100" => X <= -(A+M);
      when "101" => X <= (A and M);
      when "110" => X <= (A or M);
      when "111" => X <= (A xor M);
      when others => null;
    end case;
  end process;
end;
```

Shift unit

```
entity SHIFTOP is
  generic(WL  :     integer);
  port( SHIFT : in  std_logic_vector(1 downto 0);
        X     : in  std_logic_vector(W_L-1 downto 0);
        S     : out std_logic_vector(W_L-1 downto 0));
end;
```

```
architecture BEHAVIOURAL of SHIFTOP is
begin
  process (X,SHIFT)
    variable S_temp : std_logic_vector(WL-1 downto 0);
  begin
    case SHIFT is
      when "01" => S <= X(WL-1) & X(WL-1 downto 1);
      when "11" => S <= X(WL-2 downto 0) & '0';
      when "10" => S <= X(WL-3 downto 0) & "00";
      when others => S <= X;
    end case;
  end process;
end;
```

MUX unit

```
entity MUX is
  generic(WL :     integer);
  port( MUX  : in  std_logic;
        B,R  : in  std_logic_vector(WL-1 downto 0);
        M    : out std_logic_vector(WL-1 downto 0));
```

```
architecture BEHAVIOURAL of MUX is
begin
  process (MUX,B,R)
  begin
    if MUX='0' then
```

```
end;
```

```
      M <= B;
    else
      M <= R;
    end if;
  end process;
end;
```

Registers

```
entity REGISTER is
  generic(WL  :     integer);
  port( clk   : in  std_logic;
        WR_R  : in  std_logic;
        S     : in  std_logic_vector(WL-1 downto 0);
        R     : out std_logic_vector(WL-1 downto 0));
end;
```

```
architecture BEHAVIOURAL of REGISTER is
begin
  process (clk,WR_R,S)
  begin
    if (clk='1') and (clk'event) then
      if WR_R='1' then
        R <= S;
      end if;
    end if;
  end process;
end;
```

ALU toplevel

The ALU toplevel instantiates the previously created designs and connects the port maps using signals.

```
entity ALU_TOP is
  generic(WL     : nteger := 8);
  port( A, B     : in  std_logic_vector(WL-1 downto 0);
        ALU_PORT : in  std_logic_vector(7 downto 0);
        clk      : in  std_logic;
        Q        : out std_logic_vector(WL-1 downto 0));
end;

architecture BEHAVIOURAL of ALU_TOP is

  component alu                                                    -- declare components
    generic (WL   :     integer);
    port( ALU_OP  : in  std_logic_vector(2 downto 0);
          A, M    : in  std_logic_vector(WL-1 downto 0);
          X       : out std_logic_vector(WL-1 downto 0));
  end component;

  ...
                                                                   -- declare signals
  signal OEN, WR_R, MUX : std_logic;
  signal SHIFT          : std_logic_vector(1 downto 0);
  signal ALU_OP         : std_logic_vector(2 downto 0);
  signal R,S,M,X        : std_logic_vector(WL-1 downto 0);
begin
  OEN    <= ALU_PORT(7);                                           -- connect signals
  WR_R   <= ALU_PORT(6);
  SHIFT  <= ALU_PORT(5 downto 4);
  MUX    <= ALU_PORT(3);
  ALU_OP <= ALU_PORT(2 downto 0);

  A_OP : alu                                                       -- instatiate units
    generic map (WL => WL)
    port map (ALU_OP => ALU_OP, A => A, M => M, X => X);

  S_OP : shiftop
    generic map (WL => WL)
    port map (SHIFT => SHIFT, S => S, X => X);

  M_OP: mux
    generic map (WL => WL)
    port map (MUX => MUX, B => B, R => R, M => M);

  R_OP: register
    generic map (WL => WL)
    port map (clk => clk, WR_R => WR_R, S => S, R => R);

  Q_OP: register
    generic map (WL => WL)
    port map (clk => clk, WR_R => OEN, S => S, R => Q);

end;
```

Table 1.1: List of operations.

Operator	Description	Operand type	Example
and	logical and	bit, vector	a := b and c
or	logical or	bit, vector	a := b or c
nand	logical nand	bit, vector	a := b nand c
nor	logical nor	bit, vector	a := b nor c
not	logical not	bit, vector	a := not b
xor	logical xor VHDL'93	bit, vector	a := b xor c
xnor	logical xnor VHDL'93	bit, vector	a := b xnor c
=	equal	bit, vector, integer	if (a = 0) then
/=	not equal	bit, vector, integer	if (a /= 0) then
<	less than	bit, vector, integer	if (a < 2) then
<=	less or equal	bit, vector, integer	if (a <= 2) then
>	greater than	bit, vector, integer	if (a > 2) then
>=	greater or equal	bit, vector, integer	if (a >= 2) then
+	addition	bit, vector, integer	a := b + "0100"
−	subtraction	bit, vector, integer	a := b - "0100"
*	multiplication	bit, vector, integer	a := b * c
**	exponent (shift)	bit, vector, integer	a := 2 ** b
&	concatenation	bit, vector, string	a := "01" & "10"

Chapter 2

Design Methodology

This chapter proposes an HDL Design Methodology (DM) that is based on utilizing certain properties of the VHDL language. After having gotten acquainted with VHDL, it seems that the language contains many useful properties. As a result, an ad-hoq coding style often arises, resulting not only in non-synthesizable code but also in many concurrent processes, signals and different source file dispositions. As the design grows, i.e., the actual length and number of source files, together with the time spent on debugging, the need for an effective DM becomes evident. This is especially important when several designers are involved in the same project. In the subsequent sections, a DM is proposed, based on the methodology proposed by Jiri Gaisler [3], together with some additional guidelines. The proposed DM main objectives are to improve the following important properties of HDL design namely:

- Increase readability.
- Simplify debugging.
- Simplify maintenance.
- Decrease design time.

The readability can be increased by exploring several important properties of HDL design and VHDL, e.g., hierarchy, i.e., abstraction level; not use more than two processes per entity, i.e., one combinatorial and one sequential; the way a Finite State Machine (**FSM**) is written, using case statements; etc. As a result, the source code will have a uniform disposition simplifying code orientation and the way to address a certain design problem.

As the design grows, the time spent on functionality debugging becomes a larger part of the design time. Using a proper DM will certainly help to minimize the time spent on debugging.

An important issue after having released an IP-block is proper maintenance and support, which might lead to an advantage over competitive designs. An effective DM can simplify these procedures, since code orientation and single block functionality verification becomes easier.

2.1 Structured VHDL programming

Structured VHDL programming involves specific properties of the VHDL language as well as addressing design problems in a certain way. This section will propose some simple guidelines to achieve the goals mentioned in the previous section. The guidelines are summarized below:

- Use a uniform source code disposition, e.g., two processes per entity.
- Utilize records.
- Use synchronous reset and clocking.
- Explore hierarchy.
- Use local variables.
- Utilize sub-programs.

The proposed DM is based on these properties and they have to be utilized throughout the complete design process. At first glance, adopting all paragraphs may seem excessive, but as the skill of the designer increases, the individual benefits will become evident. Furthermore, the presented guidelines are applicable to any synchronous single-clock design, but many will, with some modification, work for all types of designs, e.g., Globally Asynchronous Locally Synchronous (**GALS**). Furthermore, the guidelines should be seen as recommendations and following them will most certainly result in synthesizable and well structured designs. The benefits of each paragraph will be explained in detail in the subsequent sections.

2.1.1 Source code disposition

Limiting the number of processes to two per entity improves readability and results in a uniform coding style. One process contains the sequential logic (synchronous), i.e., registers, and the other contains the combinatorial part (asynchronous), i.e., behavioral logic. The sequential part always looks the same but the combinatorial part looks different in every source file due to the functionality of the block. Due to this fact, the sequential part is always put after the combinational part. A typical source code disposition is compiled below:

1. Included library(s).
2. Entity declaration(s).
3. Constant declaration(s).
4. Signal declaration(s).
5. Architecture declaration.
6. Component instantiation(s).
7. Combinatorial process declaration.

8. Sequential process declaration .

The source file disposition should always have this given order, simplifying readability, code orientation, and maintenance.

2.1.2 Records

A typical VHDL design consists of several hundreds of signal declarations. Each signal has to be added on several places in the code, i.e., entity, component declaration, sensitivity list, component instantiation, which obviously can become quite hard to keep track of as the number of lines of code increase. A simple way of avoiding this is to use records since once the signal is included in a record; all these updates are done automatically. This is due to the fact that the record is already included in the necessary places and the actual signal is hidden within the record.

A **record** in VHDL is analogous to a **struct** in C and is a collection of various type declarations, e.g., standard logic vectors, integers, other records, etc. The declarations should not contain any direction, i.e., in or out. Below is an example how a simple record is defined VHDL:

```
type my_type is record
  A : std_logic_vector(WIDTH-1 downto 0);
  B : integer;
  Z : record_type;
end record;
```

The record declarations should be compiled in a declaration package that can be imported to the various files, thus enabling reuse. Using records not only reduces many lines of code but also simplifies signal assignment since none of the default signal assignments are forgotten, which otherwise can lead to unnecessary latches in the synthesized design.

2.1.3 Clock and reset signal

The clock signal solely controls the sequential part, i.e., updating the registers, of the source file and should be left out of any record declaration. This is mainly because the clock signal is treated differently by the synthesis tools, since it is routed from an input pad throughout the complete design. Also, many clock tree generation and timing analysis tools do not support clock signals that are part of a bus. The sequential part always looks the same and should only have the clock signal in the sensitivity list, see example below:

```
sequential_part : process(clk)
begin
  if rising_edge(clk)
    r <= rin;
  end if;
end process;
```

There are two methods of implementing the reset signal, namely synchronous and asynchronous. There are also two places to put the reset assignment in the

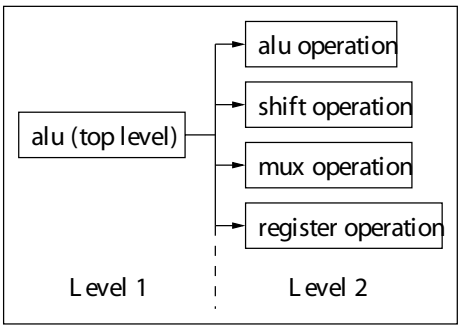


Figure 2.1: An example of how hierarchy can be used in the ALU example.

source code, i.e., in the sequential or combinational part. Choosing between the two methods and deciding where to put the assignment seems to be a religious matter and many of the arguments for choosing either of them are obsolete, since most of the commercial synthesis tools now support them. In the proposed DM, synchronous reset assigned in the combinational part is utilized, and for the same reasons as for the clock signal; the reset signal should not be included in any kind of record declaration[1]. The reset assignment is put in the latter part of the combinational block, just before the current state record is updated, and should be triggered on a logic zero, which is a commonly used convention. The following example illustrates where the synchronous reset assignment is placed in the source file:

```
combinational_part : process (various signals and records, reset)
  begin
  ...
  if (reset = '0') then
    v.state := IDLE;
    ...
  end if;
  ...
  rin <= v;
end process;
```

2.1.4 Hierarchy

Hierarchy is an important concept and is general for all types of modeling since altering the abstraction level tends to simplify a given problem. In VHDL, hierarchy can be utilized to split a given problem or model into individual modules, designs or sub programs. Each program level is controlled by component instantiation, enabling debugging of each block individually. Clearly, there is no general methodology to explore hierarchy rather that it can and should be utilized in every design. A block diagram of how hierarchy can be used in the ALU example can be seen in Figure 2.1:

[1]A synchronous reset signal can be added in a record since it is then treated as any other signal but it is not advisory since it does not support a uniform coding style.

2.1.5 Local variables

By utilizing local variables instead of signals in the combinational part, a given design problem is transformed from concurrent to sequential programming, i.e., each line of code is executed in consecutive order during simulation. This is advantageous since humans find it easier to think in sequential terms rather than in parallel. Using local variables will also increase simulation speed since simulation is often event triggered and the tool can execute events in order of appearance and does not have to consider parallel events. Basically, there is only one restriction when using local variables namely that a computational result stored in a variable should not be reused within the same clock cycle but rather stored in a register to be used in the succeeding clock cycle. This is mainly to avoid long logic paths and combinational feedback loops. Below is an example of how to use local variables in the ALU example:

```
ALU_example : process("additional signals and records", r, reset)
  variable v : reg_type;
begin
  v := r;
  ...
  case r.state is
    when IDLE => v.state := CALC;
    ...
  end case;
  ...
  rin <= v;
end;
```

Initially, the local variable v is set to the current values of the registers stored in r. Then, the local variable v is used throughout the combinational part in a sequential programming fashion. Finally, the input signal to the registers rin is updated to the computed values of the local variable v. To see the complete code of the ALU example, please refer to Section 2.2. Variable and signal assignments should be placed exactly as in the example above.

2.1.6 Subprograms

Using sub-programs is often a good way of compiling commonly used functions and algorithms, thus increasing readability. Subprograms can be used to increase the hierarchy and abstraction level, hiding complexity and enabling reuse. The subprograms are defined in packages and imported into the different source files. A restriction in the use of subprograms is that functions should only contain purely combinatorial logic. Below is an example of how subprograms can be used in the ALU example:

```
-- package declaration
function shift_operation (A : in data_type, m : in data_type,
                          control : in control_type)
                          return data_type;

-- package body
```

```
function shift_operation(A : in data_type , m : in data_type,
                         control : in control_type)
                         return data_type is
begin
  case control is
    when "000" => return A;
    when "001" => return A + m;
    when "010" => return A - m;
    when "011" => return m;
    when "100" => return (others => '0');
    when "101" => return A and m;
    when "110" => return A or m;
    when "111" => return A xor m;
    when others => return null;
  end case;
end function shift_operation;
```

Each subprogram can be debugged individually with a separate testbench to ensure correct functionality of the block.

2.1.7 Summary

The previous sections have proposed guidelines to establish a sound DM to be used throughout the whole design process. As a result, a uniform coding style can be established to simplify and increase readability, debugging, and maintenance. There are also several other benefits of adopting this DM where most important are summarized in the table below:

- Less lines of code.
- Altering the abstraction level, hiding complexity, and enabling reuse.
- Parallel coding is transformed into sequential.
- Improved simulation and synthesis speed.

Finally, adopting proposed guidelines hopefully results in synthesizable and well disposed designs that will help the design team on the way towards the final goal, which is a robust design using a minimal design time.

2.2 Example

The same example as in the previous chapter is now presented, using the proposed design methodology. The ALU code is divided into a global package and the actual implementation. The package contains constants and type declarations to improve the readability and portability, and can be found below. The rest of the ALU implementation is presented in Section 2.3.5.

```
library IEEE;
use IEEE.std_logic_1164.all;          -- for std_logic
use IEEE.std_logic_signed.all;        -- for signed add/sub

package alu_pkg is

  constant WL : integer := 8;

  constant ALU_STORE : std_logic_vector(2 downto 0) := "000";
  constant ALU_ADD   : std_logic_vector(2 downto 0) := "001";
  constant ALU_SUB   : std_logic_vector(2 downto 0) := "010";
  constant ALU_FB    : std_logic_vector(2 downto 0) := "011";
  constant ALU_CLEAR : std_logic_vector(2 downto 0) := "100";
  constant ALU_AND   : std_logic_vector(2 downto 0) := "101";
  constant ALU_OR    : std_logic_vector(2 downto 0) := "110";
  constant ALU_XOR   : std_logic_vector(2 downto 0) := "111";

  constant SHIFT_NONE  : std_logic_vector(1 downto 0) := "00";
  constant SHIFT_RIGHT : std_logic_vector(1 downto 0) := "01";
  constant SHIFT_2LEFT : std_logic_vector(1 downto 0) := "10";
  constant SHIFT_1LEFT : std_logic_vector(1 downto 0) := "11";

  subtype data_type is std_logic_vector(WL-1 downto 0);

  type control_type is record
    oen    : std_logic;
    fb     : std_logic;
    mux    : std_logic;
    shift  : std_logic_vector(1 downto 0);
    alu_op : std_logic_vector(2 downto 0);
  end record;

  component alu
    port ( clk     : in  std_logic;
           A       : in  data_type;
           B       : in  data_type;
           control : in  control_type;
           Q       : out data_type );
  end component;

end;
```

2.3 Technology independence

During code development, the target technology may not yet be known or changed at a later stage. An important design aspect is therefore to keep the code free from parts that are technology specific. All parts that are not standard cells and pure logic, for example **on-chip memories** and the **pads** connecting the design to the outside world, are technology dependent.
Memories and pads are instantiated in the VHDL code **by name**. At the same time, avoid hard coding the memory name or pad name in the design. A simple solution is to use abstraction. Start with creating a *mapping* file, which selects the proper component based on the user constraints and the chosen technology. For example, when creating a memory, specify only the number of data bits and address bits in the code. Then let the mapping file choose the correct memory

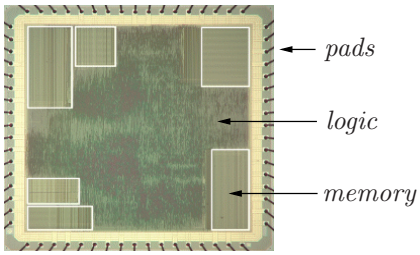


Figure 2.2: A chip layout with standard cells (logic), memories and pads.

based on the current technology and by using your specifications about the address and data bus.

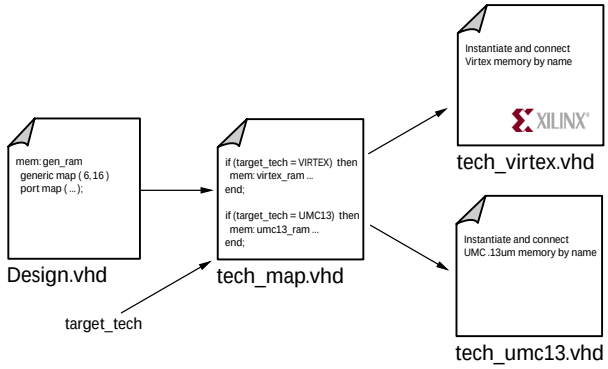


Figure 2.3: Using a mapping file to handle target specific modules.

In your design, instantiate a memory from the mapping file as:

```
use work.tech_map.all;

mem: gen_ram
  generic map ( ADDR_BITS => 5,
                DATA_BITS => 16 );
  port map ( clk, addr, d, q );
```

The instantiation will point to the technology mapping file, instead of the actual memory, which selects and connects the chosen technology dependent module. The mapping file contains the following information:

```
use work.tech_virtex.all;                                   -- one library for
use work.tech_umc13.all;                                    -- each technology

type target_type is ( VIRTEX, UMC13 );                      -- available technologies
constant target_tech : target_type := VIRTEX;               -- set technology here

entity gen_ram
  generic map ( ADDR_BITS : integer;
                DATA_BITS : integer );
  port map ( clk  : std_logic;
             addr : std_logic_vector(ADDR_BITS-1 downto 0);
             d    : std_logic_vector(DATA_BITS-1 downto 0);
             q    : std_logic_vector(DATA_BITS-1 downto 0) );
end;

architecture structural of gen_ram is
```

```
begin
  if (target_tech = VIRTEX) generate                          -- VIRTEX memory
    mem: virtex_ram
      generic map ( ADDR_BITS, DATA_BITS );
      port map ( clk, addr, d, q );
  end generate;

  if (target_tech = UMC13) generate                           -- UMC ASIC memory
    mem: umc13_ram
      generic map ( ADDR_BITS, DATA_BITS );
      port map ( clk, addr, d, q );
  end generate;
end;
```

2.3.1 ASIC Memories

ASIC memories must be created using a memory generator or by contacting the vendor. The ideal memory generator creates a VHDL simulation model, entity and architecture, and a layout file specifying the size and connections of the memory. The memory is instantiated from the VHDL code by using exactly the same name and port connections.

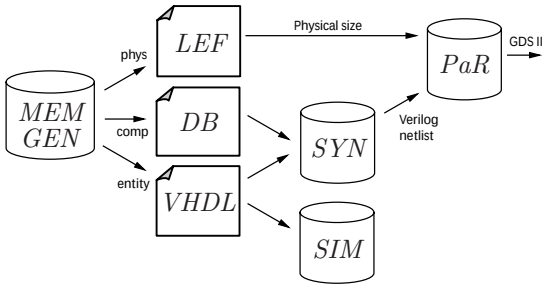


Figure 2.4: The memory generator creates simulation files, library files and physical description files.

2.3.2 ASIC Pads

For smaller projects, pads are normally inserted during synthesis. The reason is simple: it is less work. For ASIC technologies, pad insertion is only a couple of lines in the synthesis script. For Synopsys, use:

```
dc_shell> set_port_is_pad all_inputs()
dc_shell> set_port_is_pad all_outputs()
dc_shell> set_pad_type -exact IBUF all_inputs();
dc_shell> set_pad_type -exact B2CR all_outputs();
dc_shell> insert_pads -respect_hierarchy
```

The only thing you have to select is the pad name from your target library, in this case IBUF for input ports and B2CR for output ports, names that are technology specific. The number associated with the output pad is usually the driving strength in mA. However, for larger designs, several different pad types are usually required. It could be pads with different driving strength, Schmitt-trigger, open-drains or bi-directional pads. Assigning pads individually and at

the same time creating multiple version, one for each technology, will be difficult. Adding one pad requires changing the synthesis script for all technologies.

The solution is to apply the same technique as described for memories; create generic pads in a *mapping* file, and point to the current technology. Adding a new pad will only require a pad instantiation at the top level of your design, and the correct pads will be chosen from the *mapping* file.

2.3.3 FPGA modules

For FPGAs, memories and pads are only two examples of technology specific components. Xilinx has created a tool for automatically creating small cores that can be used in the design. The tool is called **Core Generator** and creates modules (.edn) that can be inferred during the place and route step. The modules are instantiated by name (which will create a black box during synthesis), replacing the black box when reading the .edf file. For example, core generator can create memories and multipliers of any size, combining the resources in the FPGA. Usually the on-chip memories are small, but can be grouped to form a larger memory.

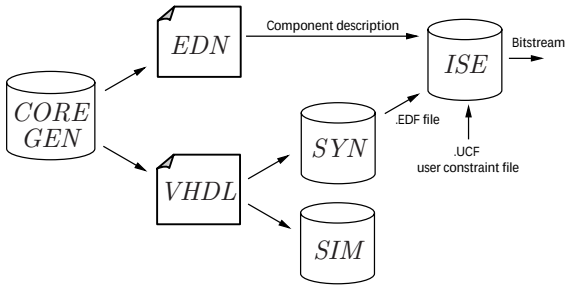


Figure 2.5: Core Generator can create all kinds of modules, instantiated by name in the VHDL code.

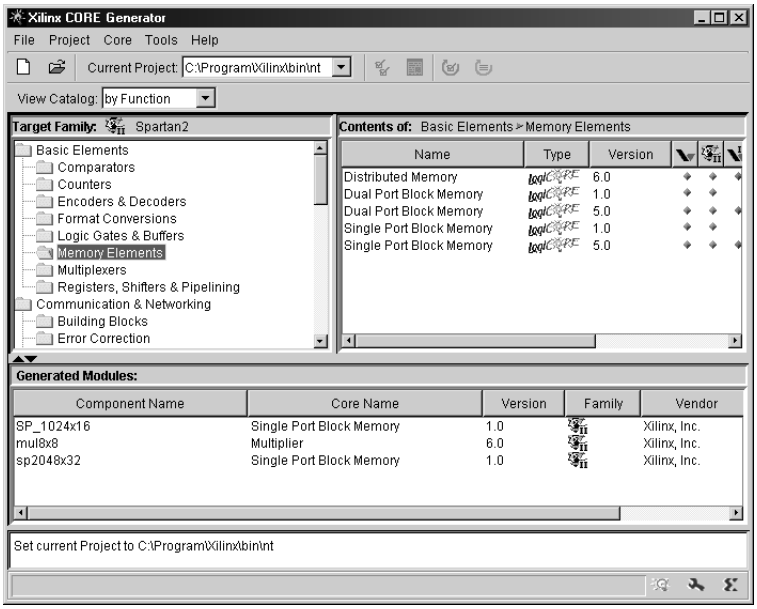


Figure 2.6: Core Generator main window. Generated cores can be instantiated in the VHDL code.

2.3.4 FPGA Pads

Memory and pad insertion is more simple for FPGAs than for ASIC. The reason is that the synthesis tool for FPGA already knows what kinds of memories and pads that are available for a certain FPGA. The synthesis tool can therefore handle a lot of things automatically.

2.3.5 ALU unit

The ALU unit imports the package on the previous page and uses the type and constant declarations. The ALU is implemented using the two process methodology, separating the sequential and combinatorial logic. All sequential elements are declared in a record type.

```
use work.alu_pkg.all;                                   -- use ALU types

entity alu is
  port ( clk     : in  std_logic;
         A, B    : in  data_type;
         control : in  control_type;
         Q       : out data_type );
end;

architecture behavioural of alu is
  type reg_type is record                               -- all registers
    output   : data_type;
    feedback : data_type;
  end record;

  signal r, rin : reg_type;
begin

  process(r, A, B, control)
    variable m, x, s : data_type;
    variable v : reg_type;
  begin
    v := r;

    if control.mux = '1' then                           -- select input or feedback
      m := B;
    else
      m := r.feedback;
    end if;

    case control.alu_op is                              -- arithmetic operation
      when ALU_STORE => x := A;
      when ALU_ADD   => x := A + m;
      when ALU_SUB   => x := A - m;
      ...
      when others => null;
    end case;

    case control.shift is                               -- shift operation
      when SHIFT_NONE  => s := x;
      when SHIFT_RIGHT => s := x(WL-1) & x(WL-1 downto 1);
      when SHIFT_2LEFT => s := x(WL-2 downto 0) & '0';
      when SHIFT_1LEFT => s := x(WL-3 downto 0) & "00";
      when others => null;
    end case;

    if (control.fb = '1') then                          -- feedback enable
      v.feedback := s;
    end if;
    if (control.oen = '1') then                         -- output enable
      v.output := s;
    end if;

    Q   <= r.output;                                    -- drive output from register
    rin <= v;
  end process;

  process(clk)                                          -- update registers
  begin
    if rising_edge(clk) then
      r <= rin;
    end if;
  end process;
end;
```

Chapter 3

Arithmetic

3.1 Introduction

In the original meaning, arithmetic describes the four fundamental rules of mathematics: addition, subtraction, multiplication and division. In computer technology, a device that performs these operations is called Arithmetic and Logical Unit (ALU). In addition to the four fundamental rules of arithmetic, an ALU can perform several other operations, e.g. shifting and logical operations.

Arithmetic and logical operations are of course very important to understand when designing hardware, as we want an efficient hardware structure as possible to perform the operation. Different implementations of a particular operation can be distinguished in terms of area, performance or power.

To understand the implementation of arithmetic operations we must first consider and distinguish two important concepts: the *operation* and the *data type*. When it comes to hardware design with VHDL, the data type and the operation is specified in the VHDL language. The description in this chapter will concentrate on the VHDL data types based on an array of `std_logic` and the operations that can be performed on this specific data type. Although the VHDL language specify both the data type and the operations to be performed on this type, it does not say anything about how the operations are to be implemented as a hardware structure. This is decided in the step which hardware designers refer to as synthesis, which in this chapter will be discussed in the context of the Synopsys synthesis tool, **Design Compiler** or **DC** for short. Hence, this chapter addresses the implementation of arithmetic operations in a digital ASIC.

3.2 VHDL Packages

Operations and data types are specified in the VHDL language in packages and are made accessible by use of a *library clause* at the beginning of the VHDL file.

```
-- Library clause in VHDL
library IEEE;
use IEEE.std_logic_1164.all;
```

As there may be several definitions of the same operator in a standard package, the actual operator selected, by the compiler, is controlled by *operator overloading*. Operator overloading means that the compiler checks input and output data types, to find the correct operation from the list of all operations currently accessible through the library clauses in the file.

The most important and widely used standard packages when it comes to ASIC design, simulation and synthesis, are:

- **std_logic_1164**
- **std_logic_arith**
- **std_logic_signed**
- **std_logic_unsigned**

The standard package called **std_logic_1164** is an IEEE standard and the other three are Synopsys de facto industry standards. We will briefly explain the content of the packages which also are found in Appendix A.

3.2.1 Data types

The basic data type from which the data types, used for writing synthesizable VHDL code, originate is `std_ulogic`. The type describes a 9-level information carrier, see Table 3.1. The information carriers are 'U' (uninitialized), 'X' (forcing unknown), '0' (forcing 0), '1' (forcing 1), 'Z' (high impedance), 'W' (weak unknown), 'L' (weak 0), 'H' (weak 1) and '-' (don't care). These are the values you can find on a signal when simulating the design in **Modelsim**.

The data type `std_logic` is a resolved `std_ulogic`, which means that if several processes assign the same signal, there will always be a defined value of that signal. This is an important concept in hardware design, to describe shared busses. When a conflict occurs in a resolved signal, a *resolution function* will be automatically invoked to solve the conflict and give the signal a well defined value.

When writing synthesizable VHDL code, the type `std_logic` and the types based on an array of `std_logic` should be used exclusively. In **std_logic_1164**, the important types `std_logic` and `std_logic_vector` are defined. The standard package **std_logic_arith** defines two additional types: `signed` and `unsigned`. These types have definitions identical to the type `std_logic_vector`. Table 3.1 summarizes the data types and the standard package from which they originate.

Table 3.1: Important VHDL types

Type	Description	Package
`std_ulogic`	'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'	**std_logic_1164**
`std_logic`	Resolved `std_ulogic`	**std_logic_1164**
`std_logic_vector`	Array of `std_logic`	**std_logic_1164**
`unsigned`	Array of `std_logic`	**std_logic_arith**
`signed`	Array of `std_logic`	**std_logic_arith**
`integer`	$-(2^{31}-1)$ to $(2^{31}-1)$	**standard**

3.2.2 Operations

It should be familiar to the reader that a number can be coded into a binary representation in different ways. A binary number representation must be specified to perform a predictable operation on the type. Since hardware for unsigned arithmetic is less complex than hardware for signed arithmetic in a given number range, these cases have been separated in the standard packages. Signed operations are done with *two's complement* number representation and unsigned operations with *natural binary* number representation. In the VHDL language, there are two ways of controlling if signed or unsigned operations should be performed.

If `std_logic_vector` is used in the design, the operation is controlled in the library clause by including the package **std_logic_signed** or **std_logic_unsigned** to select signed or unsigned operations, respectively. Hence, the type is just a container for either unsigned or signed numbers. An obvious drawback is that it is not straightforward to mix signed and unsigned operators in the same VHDL file.

```
-- Library clause for unsigned operations with std_logic_vector
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- Library clause for signed operations with std_logic_vector
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
```

The other method is to use the types `signed` and `unsigned` and hence explicitly, in the type definition, specify the contents and thereby the operation. The operators for `signed` and `unsigned` data types are all specified in the **std_logic_arith** package. Whether a signed or an unsigned operator are selected is then controlled by operator overloading.

```
-- Library clause for signed or unsigned operations with signed or
-- unsigned data types
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

As already stated, several functions for the same operation are defined in each specific package. Table 3.2 summarizes the arithmetic operations in the packages. For details about operator overloading and all defined functions, we refer the reader to Appendix A where all packages described are found.

3.2.3 VHDL examples

Here is a VHDL code example where `std_logic_vector` is used as the data type in a multiplication. The `std_logic_vector` is synonymous to an *unsigned* by including the **std_logic_unsigned** package.

```
library IEEE;
```

Table 3.2: Arithmetic operations in packages

Operation	Description	Arith	Signed	Unsigned
ABS	Absolute value	Yes	Yes	No
$+$	Unary addition	Yes	Yes	Yes
$-$	Negation	Yes	Yes	No
$+$	Addition	Yes	Yes	Yes
$-$	Substraction	Yes	Yes	Yes
$*$	Multiplication	Yes	Yes	Yes
$<$	Less than	Yes	Yes	Yes
$<=$	Less than or equal to	Yes	Yes	Yes
$>$	Greater than	Yes	Yes	Yes
$>=$	Greater than or equal to	Yes	Yes	Yes
$=$	Equal to	Yes	Yes	Yes
$/=$	Not equal to	Yes	Yes	Yes
SHL	Left shift	Yes	Yes	Yes
SHR	Right shift	Yes	Yes	Yes
EXT	Zero extension	Yes	No	No
SXT	Sign extension	Yes	No	No

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity multiplication is
  generic(wordlength: integer);
  port(in1, in2    : in std_logic_vector(wordlength-1 downto 0);
       product     : out std_logic_vector(2*wordlength-1 downto 0));
end multiplication;

architecture rtl of multiplication is

begin

  process(in1, in2)
  begin
    product <= in1 * in2;
  end process;

end;
```

Here is a VHDL code example where `std_logic_vector` is used as the data type in a multiplication. The `std_logic_vector` is synonymous to a *signed* by including the **std_logic_signed** package.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity multiplication is
  generic(wordlength: integer);
```

```
  port(in1, in2    : in std_logic_vector(wordlength-1 downto 0);
       product     : out std_logic_vector(2*wordlength-1 downto 0));
end multiplication;

architecture rtl of multiplication is

begin

  process(in1, in2)
  begin
    product <= in1 * in2;
  end process;

end;
```

Here is a VHDL code example where `signed` and `unsigned` are used as the data types in a multiplication. Only the **std_logic_arith** package are included. Hence, no arithmetic operations can be performed directly on `std_logic_vector`.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiplication is
  generic(wordlength: integer);
  port(in1, in2      : in std_logic_vector(wordlength-1 downto 0);
       product_u     : out std_logic_vector(2*wordlength-1 downto 0);
       product_s     : out std_logic_vector(2*wordlength-1 downto 0));
end multiplication;

architecture rtl of multiplication is

begin

  process(in1, in2)
  variable product_u : unsigned(2*wordlength-1 downto 0);
  variable product_s : signed(2*wordlength-1 downto 0);
  begin
    -- Unsigned multiplication
    product_u := unsigned(in1) * unsigned(in2);

    -- Signed multiplication
    product_s := signed(in1) * signed(in2);
  end process;

  product_u <= std_logic_vector(product_u);
  product_s <= std_logic_vector(product_s);
end;
```

3.3 DesignWare

In a VHDL description, the operations are specified, but it is not decided how these operations are to be implemented, or mapped, to hardware components. That is decided in the synthesis step. The synthesis tool comes with pre-built components that implement the built-in VHDL operators. These pre-built components are *technology independent* and several components for implementing the same operator may exist. Since, the different implementations have different properties when it comes to power, area, and delay, the designer provides power, area, or delay *constraints* to the synthesis tool, which then selects the most suitable implementation.

3.3.1 Arithmetic Operations using DesignWare

In synposys synthesis tool **Design Compiler** the libraries with pre-built components are called *DesignWare libraries*. In these libraries there are technology independent implementations of the in-built VHDL operators as well as other more specialized operations. Technology independent means that the implementations are described as a netlist of common *logic elements*, which could later be mapped to standard cells in a specific cell library. The DesignWare libraries which will be loaded into **DC** and used during the synthesis process are controlled in the file `.synopsys_dc.setup`. You can list all libraries currently loaded, using the dc_shell command `list -libraries`. A DesignWare library has the file extension `.sldb`.

```
dc_shell> list -libraries
   Library             File             Path
   -------             ----             ----
   dw01.sldb           dw01.sldb        /usr/local-tde/...
   dw02.sldb           dw02.sldb        /usr/local-tde/...
   standard.sldb       standard.sldb    /usr/local-tde/...
```

A DesignWare component can be included in a design using basically three different methods.

Operator inference: In operator inference, which is the most commonly used method, the synthesis tool automatically maps operators in the VHDL code to components in the included DesignWare libraries. This is done in a hierarchy of abstractions. The *VHDL operator* is associated to a *Synthetic operator*, which is bound to *Synthetic modules*. Each *Synthetic module* can have several *implementations*. The operator inference is performed by the synthesis tool during the *Analyze and Elaborate* phase, during which the VHDL operator are mapped to a Synthetic operator. Next, the designer sets the design constraints and compiles. During this phase the synthetic operator is mapped to a synthetic module and a implementation is selected. Table 3.3 list the most commonly used synthetic modules and implementations used for operator inference. A complete list of the DesignWare components can be found by typing *sold* at the UNIX prompt, select **DesignWare Library** and then select **DesignWare Building Block IP**. Figure 3.3.1 illustrates the flow for operator inference using the addition operator.

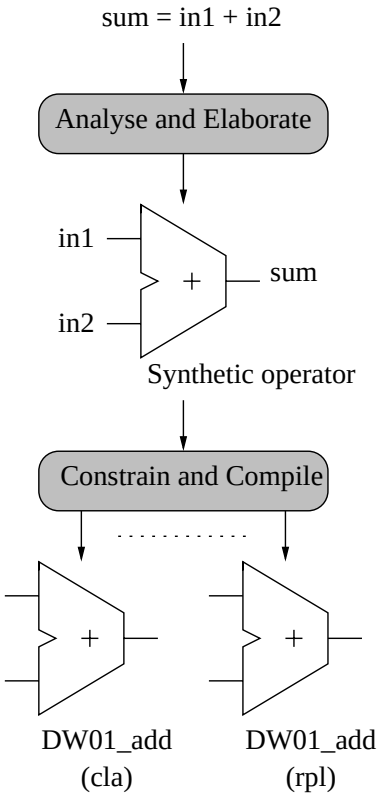


Figure 3.1: The figure illustrates the steps and abstraction levels used for operator inference of the addition operator

Function inference Including the DesignWare component through function inference is done by first making the function available though a library clause in the VHDL file, and then call the function inside the architecture body.

Component instantiation Including a DesignWare component through component instantiation is done similarly as including a component through function inference. The component is made accessible through a library clause, an entity declaration of the component is included and the architecture body contains a port map.

A complete description on how to include a specific DesignWare component through function inference or component instantiation is found in **DesignWare Building Block IP** under the specific component. Normally, when including a DesignWare component through one of the above mentioned methods, the designer selects only the synthetic module and leaves selection of implementation to the synthesis tool. However, it is possible to control also the implementation. This can be done in two different ways. One method is to use component instantiation or function inference and add VHDL code that specifies also the implementation. The other method, which makes the VHDL code implementation

Table 3.3: Basic arithmetic synthetic modules and their implementations

Synthetic Module	Operator	Description	Implementations
DW01_absval	ABS	Absolute value	Ripple (rpl) Carry Look-Ahead(cla) Fast Carry Look-Ahead (clf)
DW01_add	+	Adder	Ripple (cla) Carry Look-Ahead (cla) Fast Carry Look-Ahead (clf) Brent-Kung architecture (bk) Conditional Sum (csm)
DW01_addsub	+, −	Adder and subtractor	Ripple (rpl) Carry Look-Ahead (cla) Fast Carry Look-Ahead (clf) Brent-Kung architecture (bk) Conditional Sum (csm)
DW01_cmp2	<, >	2-Function comparator	Ripple (rpl) Fast Carry Look-Ahead (clf) Brent-Kung architecture (bk)
DW01_cmp6	<, >, <=, >=	6-Function comparator	Ripple (rpl) Fast Carry Look-Ahead (clf) Brent-Kung architecture (bk)
DW01_dec	−	Decrementer	Ripple (rpl) Carry Look-Ahead(cla) Fast Carry Look-Ahead (clf)
DW01_inc	+	Incrementer	Ripple (rpl) Carry Look-Ahead(cla) Fast Carry Look-Ahead (clf)
DW01_incdec	+, −	Incrementer and Decrementer	Ripple (rpl) Carry Look-Ahead(cla) Fast Carry Look-Ahead (clf)
DW01_add	−	Subtractor	Ripple (cla) Carry Look-Ahead (cla) Fast Carry Look-Ahead (clf) Brent-Kung architecture (bk) Conditional Sum (csm)
DW01_mult	*	Multiplier	Carry Save Array (csa) Non-Both-recoded Wallace (nbw) Wallace Tree (wall)

independent, is to use operator inference, function inference or component instantiation in the VHDL file. Then, for a specific cell, after the elaborate stage, select the implementation using the dc_shell command `set_implementation`. For example:

```
dc_shell> set_implementation DW01_add/rpl cell_name
```

selects the cell cell_name to be implemented as a Ripple Carry adder (rpl) from the synthetic module `DW01_add`. This method is preferable since it keeps the VHDL code free from implementation specific details, which is then handled in the Synthesis script. With some knowledge of the dc_shell script commands, it is for example possible to map all addition operators in a design to a specific synthetic module and implementation.

The dc_shell script command `report_resources` can be used after the compile step to check which synthetic modules and implementations the synthesis tool has selected for the design.

3.3.2 Manual Selection of the Implementation in *dc_shell*

The *Synopsys Design Compiler* automatically determines the hardware implementation to meet the design constraints speed, area and power in the given order. However, if one wants to realize hardware that meets special needs, it can be necessary to select the implementation manually.

Before the implementation type can be chosen the following steps are required a priori:

- analyze
- elaborate
- uniquify
- compile

Information on the used resources, e.g., number and type of adders, can be obtained by

```
dc_shell > report_resources
```

The table below shows an example of how the structure in Figure 3.14 is implemented. The chosen implementation is a ripple-carry structure for all adders. If one wants to change the implementation of one of the adders the desired type has to be determined manually, see Table 3.4 and needs to be assigned to the target cell as

```
dc_shell > set_implementation cla add_37/plus/plus
```

The implementation of the targeted adder cell has now been changed to a carry-lookahead structure,see the table below.

```
\label{sc:report_resources}
****************************************
Report : resources
Design: adder
Version: 2003.06-SP1-3
Date   : Mon Jul  5 16:58:42
```

Table 3.4: Adder Synthesis Implementation

Implementation name	Function	Library
rpl	Ripple-carry	DWB
cla	Carry-look-ahead	DWB
clf	Fast carry-look-ahead	DWF
bk	Brent-Kung architecture	DWF
csm	Conditional-Sum	DWF
rpcs	Ripple-carry-select	DWF
clsa	Carry-look-ahead-select	DWF
csa	Carry-select	DWF
fastcla	Fast-carry-look-ahead	DWF

```
2004****************************************

Resource Sharing Report for design adder in file
        /home/toll/jrs/Synopsys/vhdl/course/adder.vhd
===============================================================================
|          |              |            | Contained     |                      |
| Resource | Module       | Parameters | Resources     | Contained Operations |
===============================================================================
| r139     | DW01_add     | width=12   |               | add_37/plus/plus     |
| r142     | DW01_add     | width=12   |               | add_37/plus/plus_1   |
| r145     | DW01_add     | width=12   |               | add_37/plus/plus_2   |
| r148     | DW01_add     | width=12   |               | add_37/plus/plus_3   |
===============================================================================

Implementation Report

============================================================================
|                  |                  | Current          | Set              |
| Cell             | Module           | Implementation   | Implementation   |
============================================================================
| add_37/plus/plus | DW01_add         | rpl              | cla              |
| add_37/plus/plus_1 | DW01_add       | rpl              |                  |
| add_37/plus/plus_2 | DW01_add       | rpl              |                  |
| add_37/plus/plus_3 | DW01_add       | rpl              |                  |
============================================================================
```

If all the implementations in one design needs to be set manually a script can be executed in the *dc_shell*

```
SELECT_DW_CELLS = true /* Choose DW implementations (rpl or
cla)*/

DW_ADD_IMPL = rpl
DW_SUB_IMPL = rpl
DW_CMP_IMPL = rpl

design_list = find (design, "*", -hierarchy) design_list =
design_list + current_design

if (SELECT_DW_CELLS == true) {
  foreach (de, design_list) {
    current_design de
    plus_list = find (cell, "*plus*")
    if (plus_list != {}) {
      set_implementation DW01_add/ + DW_ADD_IMPL plus_list }
    if (BF == sub) {
      minus_list = find (cell, "*minus*")
      if (minus_list != {}) {
        set_implementation DW01_sub/ + DW_SUB_IMPL minus_list
      }}
    else {
      cmp_list = find (cell, "*lt*")
      if (cmp_list != {}) {
```

```
  set_implementation DW01_cmp2/ + DW_CMP_IMPL cmp_list
}}}}
```

3.4 Addition and Subtraction

Addition and substraction are two of the most frequently used arithmetic operations. They are used both in data paths as well as in controllers. Addition and subtraction are closely related operations. A substraction can be performed by negating the subtrahend and add the result to the minuend with a carry in. Hence, the same basic architectures used for the implementations of adders are found in the implementations of subtractors, see Table 3.3.

When a + or − sign is used in the VHDL code, either the *DW01_add*, *DW01_sub*, *DW01_inc*, *DW01_dec*, *DW01_incdec*, or *DW01_addsub* synthetic module will be instantiated through operator inference during compilation. The same synthetic modules and implementations are used, independent of whether unsigned or signed data types are used in the addition or subtraction. Although there is no difference between an adder/subtractor for unsigned or signed operands, there are indeed differences when it comes to exceptions, e.g., overflow conditions or increasing the wordlength of the operands.

Figure 3.4 demonstrates signed and unsigned addition for a 3-bit number representation. From this example we see that the same basic addition operation can be used for both unsigned and signed numbers. The same number of bits should be used in the result as used in the input operands. Hence, we see that adding 011 with 101 generates the result 000. This is the correct result for the signed addition, but it is an overflow for the unsigned addition. A similar example for unsigned or signed substraction shows that the same basic subtractor can be used for both signed and unsigned numbers.

unsigned	signed	code
0	0	000
1	1	001
2	2	010
3	3	011
4	−4	100
5	−3	101
6	−2	110
7	−1	111

signed	unsigned	
2	2	010
+ (−3)	+ 5	+ 101
−1	7	111
3	3	011
+ (−3)	+ 5	+ 101
0	8	(1)000

Figure 3.2: 3-bits unsigned and signed addition

3.4.1 Basic VHDL example

The wordlength of an addition or substraction specifies the number of bits used in the operands. An unsigned number with wordlength w bits lies in the range 0 to $2^w - 1$ and a signed number with wordlength w lies in the range -2^{w-1} to $2^{w-1} - 1$. Generally, the wordlength of the input operands and the result should be the same in the VHDL code. The wordlength of an addition or

subtraction is one of the most important parameters used to control the final area and performance of the implementation. To be able to do wordlength optimizations, the designer must be aware of the dynamic range of the input signals and the precisions needed in the operation.

The following VHDL code will instantiate the synthetic module `DW01_add` for the addition operator and the synthetic module `DW01_sub` for the substraction operator, through operator inference during compilation. The implementation will be selected to meet the design constraints as well as possible. The VHDL code used in the examples will not be complete when it comes to entity declaration, architecture body, etc. Instead, for the sake of readability, we show only the signal declarations and the actual operation.

```
-- Unsigned addition
in1, in2, sum  : unsigned(wordlength-1 downto 0);
sum <= in1 + in2;

-- Signed addition
in1, in2, sum  : signed(wordlength-1 downto 0);
sum <= in1 + in2;

-- Unsigned substraction
in1, in2, diff  : unsigned(wordlength-1 downto 0);
diff <= in1 - in2;

-- Signed substraction
in1, in2, diff  : signed(wordlength-1 downto 0);
diff <= in1 - in2;
```

The synthetic module `DW01_addsub` can perform both addition and substraction and will be inferred into the design if it is possible to do resource sharing between an addition and subtraction.

3.4.2 Increasing wordlength

Addition or subtraction of two operands with the dynamic range -2^{w-1} to $2^{w-1}-1$, gives a result with the dynamic range -2^{w} to $2^{w}-1$, as seen in Figure 3.4.2. Hence, to avoid overflow in the result, the wordlength of the operands must be increased by one bit before the operation, as the result must have the same wordlength as the largest wordlength of the input operands in the VHDL code.

This is handled differently depending on, whether the operands are signed or unsigned. An unsigned number should be extended with zeroes. A signed operand should be sign extended, i.e. extend with the most significant bit of the operand.

```
-- Unsigned addition with extended wordlength
in1, in2 : std_logic_vector(wordlength-1 downto 0);
sum      : std_logic_vector(wordlength downto 0);
sum <= ext(in1, wordlength+1) + ext(in2, wordlength+1);

-- Signed addition with extended wordlength
```

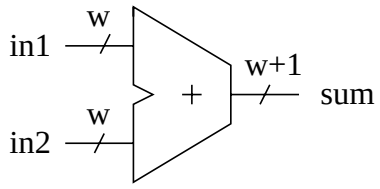


Figure 3.3: The dynamic range of the result of a two-operand addition or substraction is increased by one bit compared to the dynamic range of the input operands

```
in1, in2 : std_logic_vector(wordlength-1 downto 0);
sum      : std_logic_vector(wordlength downto 0);
sum <= sxt(in1, wordlength+1) + sxt(in2, wordlength+1);
```

An important condition for addition or substraction in VHDL is that the output wordlength must be equal to the largest wordlength of the operands. The operand with shortest wordlength will be zero or sign extended, depending on whether it is unsigned or signed, respectively. In the following VHDL example *in1* will be automatically sign extended to 16 bits.

```
-- Signed addition with implicit sign extension
in1 : signed(7 downto 0);
in2 : signed(15 downto 0);
sum : signed(15 downto 0);
sum <= in1 + in2 ;
```

3.4.3 Counters

Counters and addressing arithmetic are used extensively in controllers. A counter is defined as an addition or subtraction where one of the operands is a constant. In such operations the addition or substraction can be optimized compared to the two-operand operation. In VHDL, a counter is best described by using an integer to represent the constant.

```
-- Counter adding 5
input     : std_logic_vector(wordlength-1 downto 0);
counter   : std_logic_vector(wordlength-1 downto 0);
counter <= input + 5;
```

In the special case when the constant is equal to +1, we have a incrementer. When the constant is equal to −1, a decrementer. The incremeter/decrementer is the only case among counters that has a dedicated synthetic module in the DesignWare library. An incrementer will be mapped to the synthetic module `DW01_inc` and a decrementer to the synthetic module `DW01_dec`. If it is possible to perform resource sharing between incrementation and decrementation, the synthetic module `DW01_incdec` will be used.

The area and delay through a counter depends on the constant. For example, a logic 0 at bit position i in the constant will prevent carry propagation to the next stage in a ripple carry structure. Firstly this means that a half adder is

sufficient at bit position $i + 1$ and secondly that the critical path through the carry chain will be reduced.

The beneficial architecture of the incrementer/decrementer in the DesignWare library may also be used, with some efforts, to make a counter with a constant that is a power of two. The following VHDL example illustrates this procedure with the constant equal to 8.

```
-- Counter adding 8
input     : std_logic_vector(wordlength-1downto 0);
counter   : std_logic_vector(wordlength-1 downto 0);
counter <= (input(wordlength-1 downto 3) + 1) & input(2 downto 0);
```

This example gives the area 205 area units compared to 224 area units, if the operation where expressed writing +8.

3.4.4 Multioperand addition

Multioperand addition means addition with more than two input operands. To perform this in a single cycle a binary tree of adders is used. Single cycle multioperand addition are expressed in VHDL by using the + operator between all operands. The following example shows a single cycle 4-operand addition.

```
-- Multioperand addition
in1, in2, in3, in4, sum :signed(wordlength-1 downto 0);
sum <= in1 + in2 + in3 + in4 ;
```

The code will instantiate 3 `DW01_add` synthetic modules during synthesis, see Figure 3.4.4. To avoid overflow, the wordlength must be manually controlled

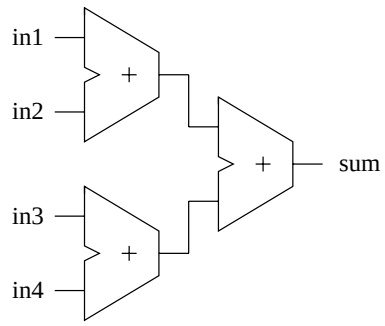


Figure 3.4: Multioperand addition, adding four operands, implemented as a binary adder tree

and selected by the designer. Implementation as a binary adder tree dictates that the wordlength of the last addition will be $w+\lceil \log_2 M \rceil$, where w is the wordlength of the input operands and M is the number of input operands. Hence, for each depth in the binary adder tree, 1 bit is added to the wordlength of the input operands.

Multioperand addition can also be performed using the specialized synthetic module `DW01_sum`, which contains optimized architectures for multioperand addition. The module is inferred into the design through function inference or component instantiation. There is no support for operator inference.

A recurrent problem closely connected to multioperand addition, is that of calculating a sum of M operands using a single time-shared adder and a register to store the partial results. The wordlength of the adder and the register should then be selected as the wordlength of the input operands plus $\lceil \log_2(M) \rceil$ bits, to avoid overflow.

3.4.5 Implementations

The design space for an operator is a multidimensional space. First, the wordlength of the operator is crucial for the designer to decide. Secondly, there are several possible implementations for each operator. Finally, for each implementation there exists an area-delay-power space, depending on the mapping of the structure to a cell library. Such a multidimensional space is not easily illustrated.

To make some benchmarks which can be illustrated, we have used a 16-bits adder (`DW01_add`) and a 16-bits subtractor (`DW01_sub`) and plotted delay versus area for the different implementations in a 0.13 μm CMOS process, see Figure 3.4.5 and Figure 3.4.5. The plots are generated by running **DC** several times with different delay constraints. The point most far to the right in each implementation is the smallest area possible to achieve with that specific implementation.

With an area-delay plot for all different implementations we can find the optimal area-delay curve, which spans through the different implementations. This optimal area-delay curve is also plotted in the figures and the optimal implementation for a given delay interval is printed.

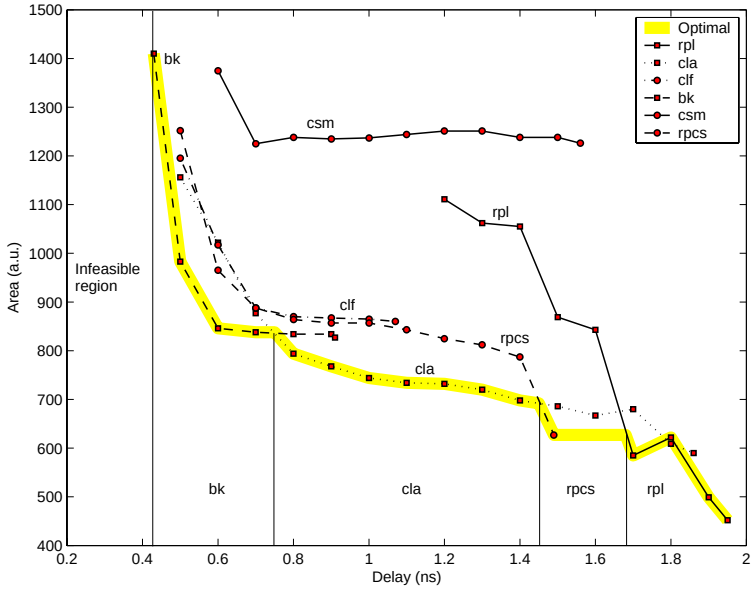


Figure 3.5: Area-Delay plot for 16-bits `DW01_add` implementations

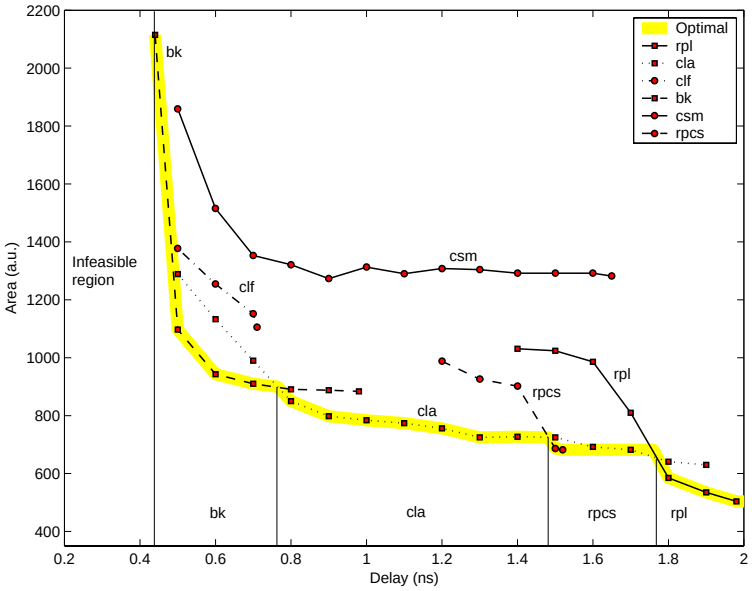


Figure 3.6: Area-Delay plot for 16-bits `DW01_sub` implementations

3.5 Comparison operation

Comparison operators are used frequently in controlling the data flow in data paths and selecting operands and outputs. The result of a comparison operation is a boolean type (true or false), which often is used to control multiplexors in the data path.

When it comes to comparison operation it is important to distinguish between signed and unsigned operators. For example, $100 < 011$ if the operands are signed and $100 > 011$ if the operands are unsigned. In VHDL there are six comparison operators: =, <, >, <=, >= and \ =. The only two operators where it does not matter if the operators are unsigned or signed is equal to (=) and not equal to (\ =).

A comparison operation in the VHDL code will instantiate the synthetic module `DW01_cmp6` or `DW01_cmp2` through operator inference during compilation. Figure 3.5 shows the input and outputs to the synthetic module `DW01_cmp6`. If

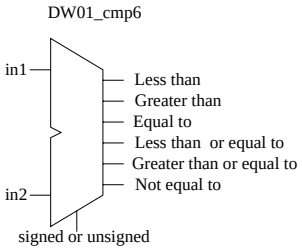


Figure 3.7: Synthetic module for the six-function comparator `DW01_cmp6`

not all the outputs of the synthetic module for comparison operations are used, the unused logic will be removed. Thus, adding no extra area to the design.

3.5.1 Basic VHDL example

The most basic example is to compare operands of equal size and data type, as shown in the following VHDL example.

```
-- Comparison
in1    : std_logic_vector(wordlength-1 downto 0);
in2    : std_logic_vector(wordlength-1 downto 0);
result : boolean;
result <= in1 < in2;
```

It is also possible to make a comparison between operands with different wordlengths, which will result in sign or zero extension of the operand with smallest wordlength.

```
-- Comparison with implicit sign extension
in1    : signed(wordlength-1 downto 0);
in2    : signed(wordlength-9 downto 0);
result : boolean;
result <= in1 < in2;
```

Comparing two operands of different types will result in an implicit type conversion. For example, the following VHDL code will increase the wordlength of the comparison by one bit. Then sign extend *in1* and zero extend *in2*.

```
-- Comparison between different types
in1    : signed(wordlength-1 downto 0);
in2    : unsigned(wordlength-1 downto 0);
result : boolean;
result <= in1 < in2;
```

The counter as a special case when building an adder/substractor structure corresponds to doing a comparison with a constant, in the context of a comparator structure. Hence, comparing one input operand with a constant makes it possible to reduce the area and delay of the operation. A constant is best described in VHDL by using an integer.

As the output of the comparison operation is very often used as an input to a multiplexor, we demonstrate such an example as well.

```
-- Comparison used to control a multiplexer
in1    : signed(wordlength-1 downto 0);
in2    : signed(wordlength-1 downto 0);
diff   : signed(wordlength-1 downto 0);
if (in1 < in2) then
  diff <= in2 - in1;
else
  diff <= in1 - in2;
end if;
```

The code may be translated to the hardware blocks (or synthetic modules) shown in Figure 3.5.1.

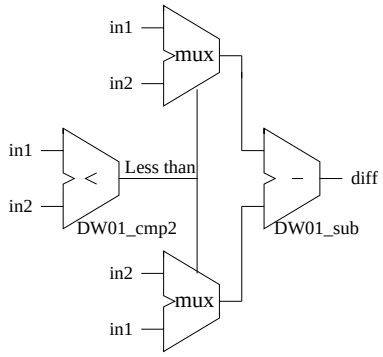


Figure 3.8: The output from the comparator used as input to multiplexors, to control the data flow

Table 3.5: Various Multiplier Synthesis Implementation

Synthetic Module	Inferencing	Description * Implementation	
DW02_mult	*	Multiplier	csa, nbw, wall
DW02_mult_stage	*	2, 3, 4, 5, 6 -stage Multiplier	csa, nbw, wall
DW02_prod_sum	component instantiation	Sum of products	csa, nbw, wall

3.6 Multiplication

The implementation of a multiplier in digital hardware can be done in various ways, see Table 3.5. Dependent on the design constraints, e.g. speed, area, and power consumption, *DC* chooses the most suitable implementation to meet the design constraints. However, if an operation such as a a *product-sum multiplier* is desired the instantiation has to be done in the VHDL code.

A multiplier is basically a construct the adds the partial products of the multiplicand and multiplier. The main differences between the different implementations is the Adder implementation. The multiplication of 2 two's complement numbers is illustrated in Figure 3.9. The width of the required Adders is $N_a + N_x - 1$. The width of the adders can be reduced to N_x by carrying out the multiplication with right-shifts, see Figure 3.12, 3.13.

				a_4	a_3	a_2	a_1	a_0
			x	x_4	x_3	x_2	x_1	x_0
				$-a_4x_0$	a_3x_0	a_2x_0	a_1x_0	a_0x_0
			$-a_4x_1$	a_3x_1	a_2x_1	a_1x_1	a_0x_1	
		$-a_4x_2$	a_3x_2	a_2x_2	a_1x_2	a_0x_2		
	$-a_4x_3$	a_3x_3	a_2x_3	a_1x_3	a_0x_3			
a_4x_4	$-a_3x_4$	$-a_2x_4$	$-a_1x_4$	$-a_0x_4$				
p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Figure 3.9: Sequential multiplication of 2 two's complement numbers.

3.6.1 Multiplication by constants

For applications where multiplications with fixed coefficients are required, *DC* optimizes the multiplication with smaller multipliers. Therefore, the multipliers must be implemented as constants rather than having them stored in a ROM. If fixed coefficients are read from a ROM, full-sized multipliers will be implemented.

However, if the multiplier, c, is a power of 2 number, $2, 4, 8, \ldots$, *DC* replaces the multiplier automatically by a bit-shift operation. Bit-shifts are hardwired and do not require any gates. Therefore, it is recommended to use as many power of 2 numbers as possible when designing hardware. Moreover, if $c \neq 2^k$ it might be beneficial to express the multipliers by bit shifts and additions/subtractions. A multiplication such as $126 \cdot x$ can be written as
$128 \cdot x - 2 \cdot x$ A VHDL example on how to write a multiplication with a constant

is given below:

```
architecture STRUCTURAL of mult126 is

  signal input     : std_logic_vector(N-1 downto 0);
  signal hundred26 : std_logic_vector(N+9-1 downto 0);

begin
  process(input)
  begin
    hundred26 <= conv_std_logic_vector(126,9)*input;
  end process;
```

A VHDL example on how to implement a multiplication as a shift-and-add operation is given below:

```
  architecture STRUCTURAL of adder126 is

  signal input     : std_logic_vector(N-1 downto 0);
  signal hundred28 : std_logic_vector(N+9-1 downto 0);
  signal hundred26 : std_logic_vector(N+9-1 downto 0);

begin

  process(input,hundred28)
  begin
    hundred28 <= conv_std_logic_vector(128,9)*input(0);
    hundred26 <=hundred28 - conv_std_logic_vector(2,3)*input(0);
  end process;
```

The area and delay can be reduced significantly by expressing a multiplication by a *bit-shift* and *add* operations. Synthesis results show that such a multiplication results in least area and delay if implemented as *carry-lookahead*, see Figure 3.10. There are various articles on how the find the minimum number of adders for the implementation of a multiplier. The more interested reader is referred to [].

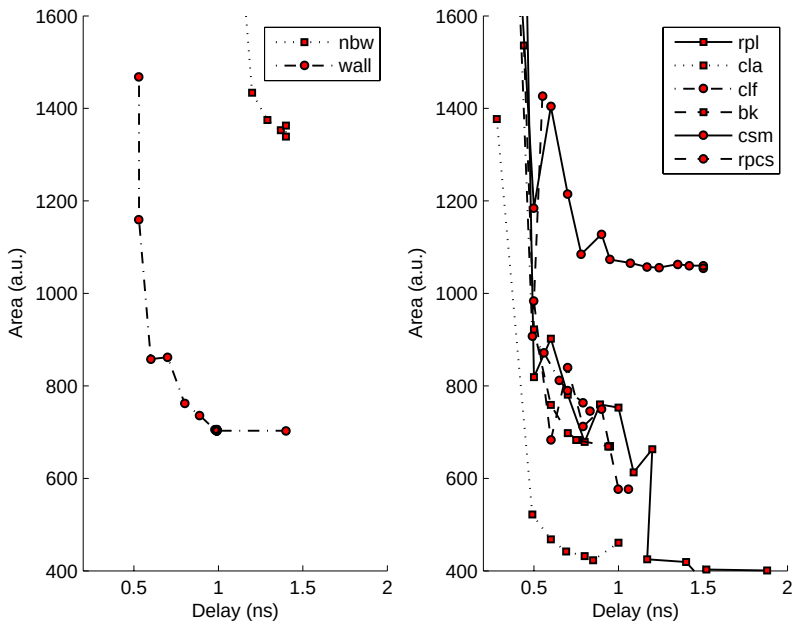


Figure 3.10: Area/Delay comparison of a implemented multiplication. The graphs on the left panel refer to an implementation as $126 \cdot x$ and on the right panel as $128 \cdot x - 2 \cdot x$

3.6.2 Multiplication of signed numbers

A multiplication can be implemented as *carry-save array (csa)*, *Non-Booth-recoded Wallace-tree (nbw)* and *Booth recoded Wallace-tree (wall)*. Synthesis results show that the *nbw* implementation is superior in terms of area and delay compared *csa* and *wall*, see Figure 3.11. The synthesis results of the *csa* implementation is not shown as it can not compete with the *csa* and *wall* implementation.

3.7 Datapath Manipulation

This section shows how the construction of the datapath can be determined by writing appropriate VHDL code. A simple arithmetic instruction as an addition can be written as:

```
z <= A + B + C + D;
```

The instruction is mapped to hardware as a balanced tree if possible in order to minimize the delay. The same hardware construct can be obtained as follows:

```
z <= (A + B) + (C + D);
```

If information on the arrival time of the signal is available the delay can be shortened by rearranging the order of the adders. Assuming that signal E arrives late, the delay time can be shortened by using parentheses as

```
z <= (A + B + C + D) + E;
```

The parser is forced to place E latest in the adder chain. The signals within the parentheses will be arranged as a balanced adder tree, see Figure 3.15.

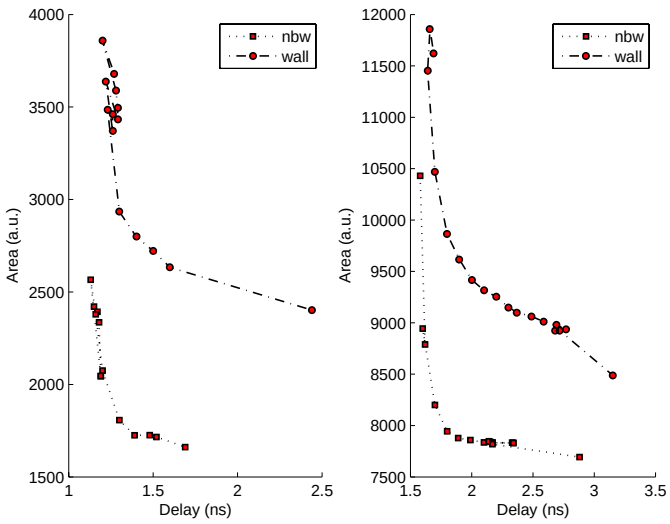


Figure 3.11: Area/Delay comparison of multipliers. On the left-hand side a 8-bit multiplier and on the right-hand side a 16-bit multiplier.

```
                  1 0 1 0 1
              x   1 0 1 0 1
              -----------------
p(0)              0 0 0 0 0
+x0a              1 0 1 0 1
              -----------------
2p(1)         1   1 0 1 0 1
         ---->    1 1 0 1 0 1
+x1a              0 0 0 0 0
              -----------------
2p(2)         1   1 1 0 1 0 1
         ---->    1 1 1 0 1 0 1
+x2a              1 0 1 0 1
              -----------------
2p(3)         1   1 0 0 1 0 0 1
         ---->    1 1 0 0 1 0 0 1
+x3a              0 0 0 0 0
              -----------------
2p(4)         1   1 1 0 0 1 0 0 1
         ---->    1 1 1 0 0 1 0 0 1
+(-x4a)           0 1 0 1 1
              -----------------
  p(5)            0 0 1 1 1 1 0 0 1
```

Figure 3.12: Negative Right-Shift Multiplier.

```
                  1 0 1 0 1
              x   0 1 0 1 1
              -----------------
p(0)              0 0 0 0 0
+x0a              1 0 1 0 1
              -----------------
2p(1)         1   1 0 1 0 1
         ---->    1 1 0 1 0 1
+x1a              1 0 1 0 1
              -----------------
2p(2)         1   0 1 1 1 1 1
         ---->    1 0 1 1 1 1 1
+x2a              0 0 0 0 0
              -----------------
2p(3)         1   1 0 1 1 1 1 1
         ---->    1 1 0 1 1 1 1 1
+x3a              1 0 1 0 1
              -----------------
2p(4)         1   1 0 0 0 0 1 1 1
         ---->    1 1 0 0 0 0 1 1 1
+x4a              0 0 0 0 0
              -----------------
2p(5)         1   1 1 0 0 0 0 1 1 1
 p(5)             1 1 1 0 0 0 0 1 1 1
```

Figure 3.13: Positive Right-Shift Multiplier.

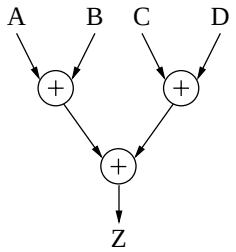


Figure 3.14: Balanced Adder Tree

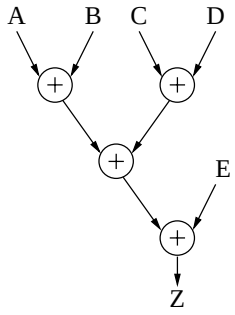


Figure 3.15: Unbalanced Adder Tree

Chapter 4

Memories

One of the most important topics in digital ASIC design today is memories. Memories occupy a lot of space and consume a lot of power and the situation becomes even worse if off-chip memories are considered. Even though each new generation of silicon processes can include more transistors on a single chip, it is predicted that the percentage of memory on a chip will increase. In the road map from Japanese system-LSI industry it is predicted that in 2014 more than 90% of a chip is covered with memory, compared to todays 50%.

With these numbers in mind, it is reasonable to spend some time to optimize the memory system when designing a new ASIC. Each saved square millimeter of silicon corresponds to a neat pile of money and to be able to use ASICs in mobile or extremely fast applications, low power consumption is required. In mobile applications to maximize battery life time and in fast applications to avoid expensive cooling equipment. Another memory issue is that the larger a memory is the slower and more power hungry it becomes. The consequence is, that the designer have to split large memories into smaller once in order to be more power efficient or to meet timing requirements. This will create a more complex memory hierarchy and introduce more design criterias.

This section does not try to cover every aspect of memory design; instead some examples on what can be done to optimize memory systems. These examples do not serve as the final truth but rather show the reader that memory systems have to be tailor made for each specific application. The first example is about minimizing the area of a Shift Register (SR) [4]. The second example deals with a memory that is not always utilized 100% [5]. A cache system to read from a off-chip memory in a image convolution application is presented in the last example [6]. For more information on memory designs consult [7].

4.1 SR example

In this example shift register (SRs) are considered. SRs are commonly used to delay or align data in data paths, for example, in a pipelined Fast Fourier Transform (FFT) processor. Here, four different ways to implement a SR are presented together with an estimate of the required silicon area.

Since one input is read and one output is written each clock cycle, the SR could be implemented in at least four different ways, as shown in Figure 4.1:

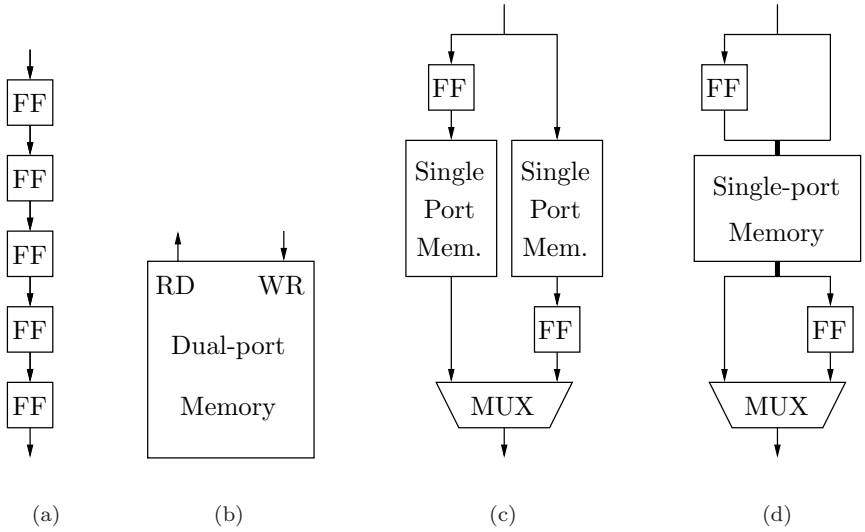


Figure 4.1: Four ways to realize a SR. Flip-flop based (a), dual-port memory (b), two single-port memories (c), and one single-port memory of twice width (d).

a. Flip-flops connected in series: This is an easy solution since there is no control logic needed. Additionally, flip-flops have a fast access time and thus allow high operation speed. However, flip-flops are not optimized for area.

b. One dual-port memory: A dual-port memory can perform both a read and write operation each clock cycle and thus suits the requirements perfectly. However, in order to perform simultaneous read/write additional logic is required in each memory cell, compared to a single port memory.

c. Two single-port memories of half length and alternating read/write: The input is gathered in blocks of two and written to the memories every other clock cycle. In a similar manner, two words are read from the memories every other clock cycle. Single-port memories have smaller memory cells than dual-port memories, but additional logic is required outside the memories in order to gather and separate input and output.

d. One single-port memory of half length and double width, which reads and writes every other clock cycle. This solution works in the same way as the previous solution, but instead of storing the two inputs in separate memories they are stored as one word in the same memory location. Thus, the memory has to be twice as wide.

Figure 4.2 shows the estimated area for the different implementations of SRs in a 0.35μm CMOS technology. There are no interconnections included in the flip-flop area. All memory SRs include control logic and a 50μm power ring on three sides of the block. In the figure it can be seen, that flip-flops is only the best solution, from an area perspective, if the SR is less than approximately 400

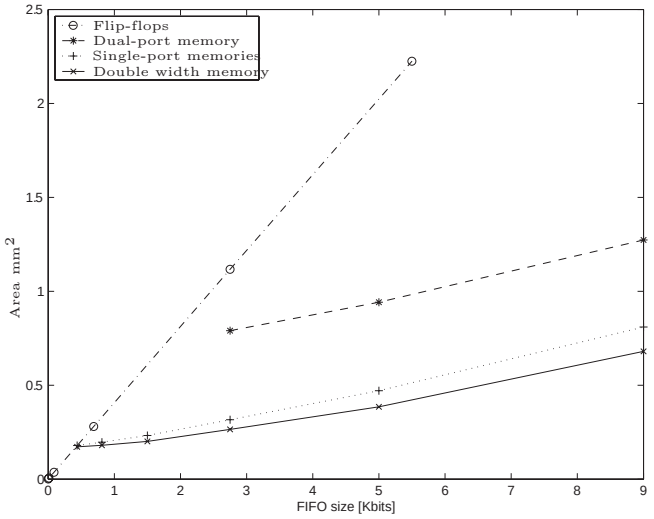


Figure 4.2: Area versus SR size for different implementations.

bits and that one single port memory of double width is the most area efficient solution for SRs larger than 400 bits. It can also be seen that memory size does not increase linearly with the number of bits such as flip-flops do. This is due to the decreased relative overhead from address logic and buffers, these parts do not increase in size as fast as the cell matrix does. A rough number on the overhead can be estimated if the memory areas in Figure 4.2 are extrapolated down to a zero bit SR. Particularly, it can be seen that the overhead for a dual-port memory is extremely large compared to single-port memories in this process. This is due to extra transistors in each memory cell and double address logic.

4.2 Memory split example

The target application in this example is a memory that continuously stores data sequences of different lengths. The length of the sequence, N, can be any power of two between 32 and 1024. The application will write N words of data into the memory and then read the same data in a different order from the memory, this operation will be performed continuously.

Table 4.1: Average current drawn for 128-1024 words memories. With V_{dd}=3.3 V and a wordlength of 24 bits.

Size [words]	1024	512	256	128
IDD [μA/MHz]	516	463	436	74

Since the largest N in this design is 1024, the memory has to be 1024 words long. If the memory is implemented as one block it will consume much power even if only a subset of the addresses is used. Hence, to reduce power consumption for cases when N is less than 1024, the memory is split into smaller blocks. As a trade-off between power reduction and the difficulty to place and

Table 4.2: Average current drawn for the memory bank for different sizes of N. The values are given for V_{dd}=3.3 V and a wordlength of 24 bits.

N	1024	512	256	128	64	32
IDD [μA/MHz]	359	255	74	74	74	74

Table 4.3: Average power consumption for different sizes of N. The values are given for V_{dd}=3.3 V and a clock frequency of 50 MHz.

N	1024	512	256	128	64	32
One memory [mW]	85	85	85	85	85	85
Memory bank [mW]	59	42	12	12	12	12
Power savings [%]	30	50	86	86	86	86

route many memories, the memory is split into four blocks of size 128, 128, 256, and 512 words. Table 4.1 shows the average current that the memory draws. The values are given for memories in a standard 0.35 μm CMOS process with five metal layers, provided as macrocells. Note that the 128-words memory is a low power memory and this was the largest low power memory available for this process. The average current drawn for the memory bank for different sizes of N is shown in Table 4.2. For example, if N = 512-points, both 128-words memories and the 256-words memory is used. Each 128-words memory is used a quarter of the time and the 256-words memory is used half the time. In this case the average current is calculated as $436/2 + 74/4 + 74/4 = 255$ μA/MHz. With N smaller than 512-points, only the low power memories are used.

Table 4.3 shows a comparison of the power consumption between the one memory and the memory bank solution. The values are given for a clock frequency of 50 MHz and V_{dd}=3.3 V. The last row shows the power savings in percent. The power savings are between 30 and 86 percent. However, the drawback is that a larger chip is required, since the silicon area increases with the number of memories due to the overhead in address logic and power supply.

4.3 Cache example

The target application is a custom image convolution processor. Two dimensional image convolution is one of the fundamental processing elements among many image applications and has the following mathematical form:

$$y(K_1, K_2) = \sum\sum x(K_1 - m_1, K_2 - m_2)h(m_1, m_2), \tag{4.1}$$

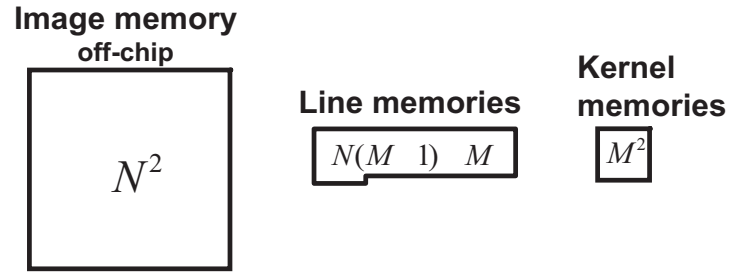
where x and h denote the image to be filtered and kernel functions, respectively. K_1 and K_2 is the pixel index, i.e., this operation is performed once for each output pixel. In this example x is 256×256 pixels and h is 15×15 pixels large.

As a consequence of the image size, the complete image has to be stored on off-chip memory. During the convolving operations, each kernel position requires 15×15 pixel values from off-chip memory directly. When this is performed in real-time, a very high data throughput is needed. Furthermore, accessing large off-chip memory is expensive in terms of power consumption and so is the signaling due to the large capacitance of package pins and PCB wires.

Table 4.4: memory hierarchy schemes and access counts for an image of 256×256 (M0=off-chip 0.18μm, C0 and C1=0.35μm)

Scheme	M0	C0	C1	energy cost
A: M0	13176900			790 mJ
B: M0→ C0	65536	13176900		56.6 mJ
C: M0→ C1	929280		13176900	68.9 mJ
D: M0→C0→C1	65536	929280	13176900	20.8 mJ

By the observation that each pixel value, except the one in the extreme corners, is used in several calculations, a two level memory hierarchy was introduced. Instead of accessing off-chip memories 225 times for each pixel operation, 14 full rows of pixel values and 15 values on the 15th row are filled into 15 on-chip line memories before any convolution starts. After the first pixel operation, for each successive pixel position, the value in the upper left is discarded and a new value is read from the off-chip memory. As a result, pixel values from the off-chip memories are read only once for the whole convolving process from upper left to lower right. Thus the input data rates are reduced from 15×15 accesses/pixel to only 1 read. In Figure 7.16 a three level memory hierarchy is shown and the number of memory access required for Equation 4.1 when using one, two, or three levels of the hierarchy.



Scheme	Image	Line	Kernel
Image	$M^2(N-M+1)^2$		
Image line	N^2	$M^2(N-M+1)^2$	
Image kernel	$MN(N-M+1)$		$M^2(N-M+1)^2$
Image line kernel	N^2	$MN(N-M+1)$	$M^2(N-M+1)^2$

Figure 4.3: Three levels of memory hierarchy and the number of reads from each level.

Using a two level memory structure can reduce both power consumptions and I/O bandwidth requirements, one extra cache level is introduced to further optimize for power consumptions during memory operations, as shown in Figure 7.16. Since accessing large memory consumes more power, 15 small cache memories are added to the two level memory hierarchy. The small cache memories are composed of 15 separate memories to provide one column pixel values

during each clock cycle. Instead of reading pixel values directly from line memories 15 times for each pixel operation, one column of pixel values are read to cache memories first from the line memories for each new pixel operation except for the first one. During each pixel operation, direct line memory access is replaced by reading from the cache. As a result, reading pixel values from line memories 15 times could be replaced by only once plus 15 times small cache accesses. Assuming that the embedded memories are implemented in a 0.35μm process CMOS, by the calculation method in [7], it is shown in Table 4.4, that the power consumption for the total memory operation could be reduced over 2.5 times compared to that of a two level hierarchy.

In addition to the new cache memory, one extra address calculator is synthesized to handle the cache address calculation. In order to simplify address calculation, the depth for each cache is set to 16. This allows circular operations on the cache memories. During new pixel operations each new column of data is filled into the cache in a cyclic manner. However, the achieved low power solution has to be traded for extra clock cycles introduced during the initial filling of the third level cache memories. In the case of an image size of 256×256, this will contribute 61696 clock cycles in total. Compared with the total clock cycles in the magnitude of 10^7, such a side effect is negligible.

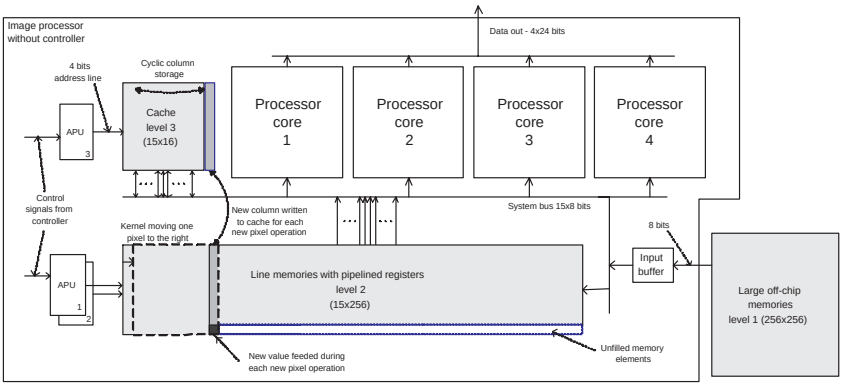


Figure 4.4: Controller Architecture with incremental circuitry

In fact, alternative schemes exists by different combinations of the three memories in the hierarchy. All these possible solutions differ substantially in memory accesses. In Table 4.4, different memory schemes are shown where M0, C0, C1 denote off-chip memory, line memories and smaller caches, respectively. Although the amount of data provided to the datapath remains the same for all four schemes, the access counts to the large off-chip memories varies. For the two and three level hierarchy structures, the counts to the large memory M0 are reduced by nearly 225 times compared to that of the one level solution. Between scheme B and D, the least access counts to both external memories and line memories are identified in three level memory structure, but this is achieved at the cost of extra clock cycles introduced during each pixel operation. Thus, trade off should be made when different implementation techniques are used. For the FPGAs where power consumption is considered less significant, two level hierarchies prevails due to its lower clock cycle counts. While for ASIC

solutions, three level hierarchy is more preferable as it results in a reasonable power reduction.

Chapter 5

Synthesis

5.1 Basic concepts

Synthesis is the process of taking a design written in a hardware description language, such as VHDL, and compiling it into a netlist of interconnected gates which are selected from a user-provided library of various gates. The design after synthesis is a gate-level design that can then be placed and routed onto various IC technologies.

There are three types of hardware synthesis, namely, logic synthesis, which maps gate level designs into a logic library, RTL synthesis, creating a gate level netlist from RTL behavior, behavioral(high-level)synthesis, that creats a RTL description from an algorithmic representation.

Such a process is generally automated by many commercial CAD tools. However, this tutorial will only cover the synthesis tool named *Design Compiler* by **Synopsys**, which is widely used in industry and universities. The much simplified version of the design flow is shown in Figure 5.1. The Design Compiler product is the core of the Synopsys synthesis software products. It comprises tools that synthesize your HDL designs into optimized technology-dependent, gate-level designs. It supports a wide range of flat and hierarchical design styles and can optimize both combinational and sequential designs for speed, area, and power.

Design Compiler deals with both RTL and logic synthesis work. Its process of synthesis can be described as translation plus logic optimization plus mapping. This is illustrated in Figure 5.2.

Translation is the process of translating a design written in behavioral hdl into a technology independent netlists of logic gates. In design compiler, this is done by either the command **read_vhdl** or **analyze/elaborate**. It performs syntax checks, then builds the design using generic (GTECH) components.

After translation, the design in the form of netlists of generic components needs to be mapped into the real logic gates (contained in cell libraries provided by IC vendors). This process is accompanied by design optimization. According to various user specified constraints and optimization techniques, design compiler maps and optimizes the design with a range of algorithms, eg., transformation, logic flattening, design re-timing, and selecting alternative cells, etc. Aiming to meet the design constraints, the process makes trade-offs between

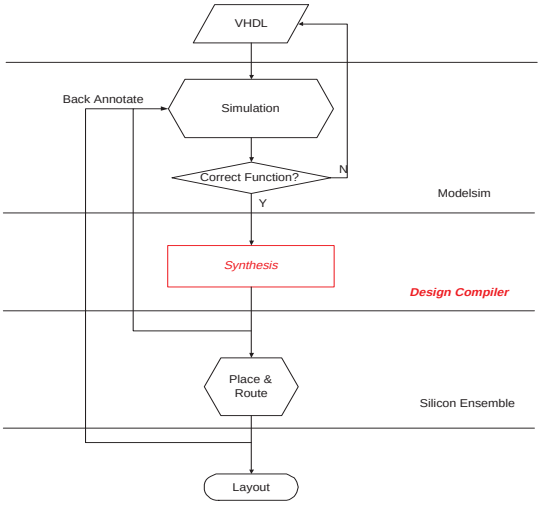


Figure 5.1: Simplified ASIC Design flow

speed, area and power consumption. Upon completion, the unmapped design in Design Compiler memory is overwritten by the new, mapped design, ready for use in later physical design of place and route.

5.2 Setting up libraries for synthesis

Before Design Compiler could synthesize a design, it is necessary to tell the tool which libraries it should use to pick up the cells. In total, there are three types of libraries that should be specified in Synopsys setup file ".synopsys_dc.setup", namely, technology libraries, symbol libraries, and DesignWare libraries.

The technology libraries, which contain information about the characteristics and functions of each cell, are provided and maintained by the semiconductor vendors and must be in .db format. If only library source code is provided, you have to use Synopsys Library Compiler to generate .db format, see Library Compiler manual for further information. In addition to cell information, Wireload models are also provided to calculate wire delays during optimization process. Wireload models will be discussed in detail in the following sections.

The symbol libraries contain graphical symbols for each library cells. As with technology libraries, it is also provided and maintained by semiconductor vendors. Design compiler uses it to display symbols for each cell in the design schematic.

The DesignWare libraries are composed of pre-implemented circuits for arithmetic operations in your design, and such designware parts are inferred through arithmetic operators in the HDL code, such as "$+ - * <><=>=$". The libraries come with two types: the standard DesignWare library and the Foundation DesignWare library. The standard version provides some basic implementation of arithmetic operations, for example, the classic implementation of multiplication by a 2-D array of full adders. The Foundation provided some advanced implementation to provide significant performance boost at the cost of a higher gate count. For example, a Booth-coded Wallace tree multiplier is provided

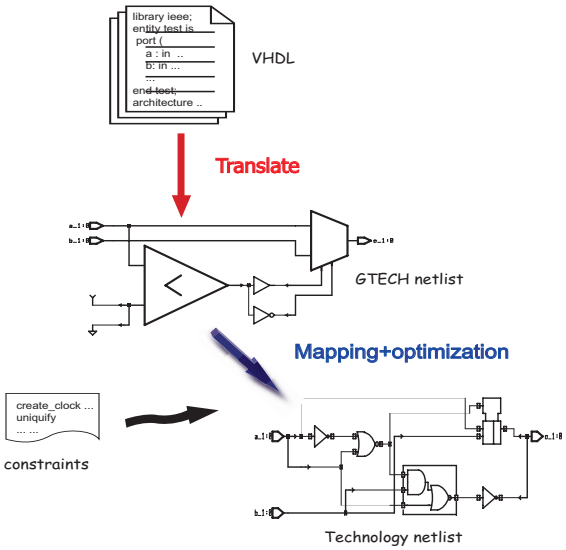


Figure 5.2: Synthesis Process

for multiplications. In addition to basic arithmetic operations, several complex arithmetic operations are implemented to obtain even higher performance. These are the MAC (Multiply-Accumulate), SOP (Sum-of-Products), vector adder(A+B+C+D), their counterpart DesignWare components are DW02_mac, DW02_prod_sum, DW02_sum respectively. One thing to note is these parts can not be inferred through HDL operators, modifications to the HDL code has to be done by instantiating or using function calls to utilize those parts. The standard library comes for free, while the foundation library need a extra license.

To setup these libraries in Synopsys setup file, use four library variables, "target_library, link_library, symbol_library, synthetic_library".

Here is one example segment of the setup files for AMS035 technology at the department.

```
/* -- .synopsys_dc.setup -- -- setup file for Synopsys synthesis
Synopsys v2002.05 --
AMS 0.35u CMOS v3.50 -- Feb 2003  / S Molund -- */

company = "TdE LUND"

cache_read  = ./; cache_write = ./;


search_path = {
/usr/local-tde/cad1/amslibs/v3.51/synopsys/c35_3.3V } +
search_path

link_library = { c35_CORELIB.db c35_IOLIB_4M.db standard.sldb
dw01.sldb dw02.sldb}

target_library = { c35_CORELIB.db c35_IOLIB_4M.db }

symbol_library = { c35_CORELIB.sdb c35_IOLIB_4M.sdb }
synthetic_library = { standard.sldb dw01.sldb dw02.sldb }
```

From Synopsys manual, target libraries set the target technology libraries where all the cells are mapped to, in the example above, "c35_CORELIB.db

c35_IOLIB_4M.db" in the directory indicated in "search_path". The link libraries resolve cell references in your design, so both target library(c35_CORELIB.db c35_IOLIB_4M.db), your own design(* by default), and DesignWare libraries(standard.sldb dw01.sldb dw02.sldb) should be specified here. The variable "symbol_library", "synthetic_library" set symbol libraries and DesignWare libraries respectively, where "stdandard.sldb" represents standard DesignWare library while "dw01.sldb dw02.sldb" is a foundation DesignWare library.

5.3 Baseline synthesis flow

After correctly setting up the environments, the synthesis tool is ready to start. Synopsys Design Compiler provides two ways to synthesize the design. One is the command line user interface, called *dc shell*, which works much like the way of a unix shell, and even some common unix command is also supported in such a shell, eg., ***ls***, ***cd***. The shell can be invoked by typing the command: **dc_shell**. If the designer is more into the graphical interface, there exists two programs provided by Synopsys, namely, *design analyzer* and *design vision*. These two programs are nothing but simple graphical interfaces for dc shell. Most common operations in synthesis can be done by "pushing-button" procedure in the graphical tools. To use any of the two graphical tools, type in **design_analyzer &** or **desgin_vision -dcsh_mode &**.

To synthesize a design using command line based dc shell is usually confusing for beginners, however, it is a more efficient way to do synthesis for the experienced designer. By running *synthesis script* (a collection of synthesis commands and constraints) on a dc_shell, a synthesis flow can be fully automated without any intervention. For this reason, only synthesis commands/constraints are given for each synthesis steps in the following sections. Beginners who uses graphical tools can easily find these commands/constraints in the menus, refer to *Design Vision User Guide* or *Design Analyzer Reference manual* for more details about the graphical tools.

5.3.1 Sample synthesis script for a simple design

The following is a sample synthesis script for a simple design. Assume there is one project containing a hierarchy of design files, which is shown in Figure 5.3. To synthesize the design, the design files are first analyzed one by one from bottom up in the hierarchy. During this process, each design is checked for syntax error and loaded into Design Compiler memory if no errors are found. The top design is then elaborated, which means the whole design is translated into a circuit interconnected by technology independent logic gates (GTECH netlist). Given some goals (eg. speed or area of the circuit) by constraint specification, Design Compiler starts mapping and optimization processes at the same time. That is, trying to pick up cells from the technology libraries to meet the design goals. This is called optimization, which can be done by the command **compile**. Whenever the design is optimized and mapped to the technology gates, it can be saved as different format for future use, eg., .sdf and .vhd format for back-annotate simulation, .v format(verilog) for place&route. To check if the synthesized design meets the goals, report has to be generated. Use the command **report_timing** or **report_area**.

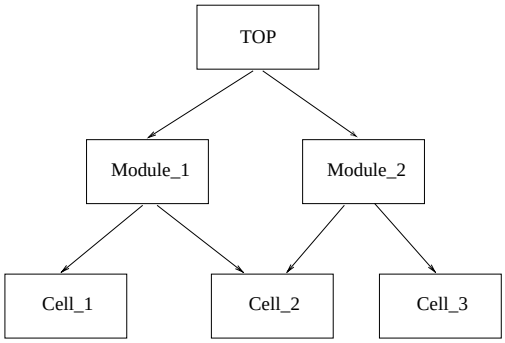


Figure 5.3: Sample hierarchical design

```
/* Analyze and ELaborate design */
analyze -f vhdl -lib WORK cell_1.vhd
analyze -f vhdl -lib WORK cell_2.vhd
analyze -f vhdl -lib WORK cell_3.vhd
analyze -f vhdl -lib WORK module_1.vhd
analyze -f vhdl -lib WORK modele_2.vhd
analyze -f vhdl -lib WORK top.vhd

elaborate top -lib WORK

/* constraints (design goals) */
create_clock -period CLK_PERIOD -name CLK find(port CLK)
uniquify


/* compile (mapping & optimization) */
current_design top
compile

/* outputs */
write -format db -hier -output TOP.db
write -f vhdl -hier -output TOP.vhd
write -f verilog -hier -output TOP.v
write_sdf -version 1.0 top.sdf

/* reports */
current_design top
report_timing >"timing.rpt"
report_area > "area.rpt"
```

5.4 Advanced topics

So far, simple synthesis steps have been given to show how to derive a hardware (netlist) from a behavioral HDL descriptions using Design Compiler. However, due to its inherent complexity in synthesis process, further knowledge on inner working mechanism of synthesis process would be beneficial for a designer to control the synthesis process in order for better synthesis results. The following sections will try to address some of important issues that will lead to a better understanding of the synthesis process, hopefully resulting in better script writing to control the synthesis process.

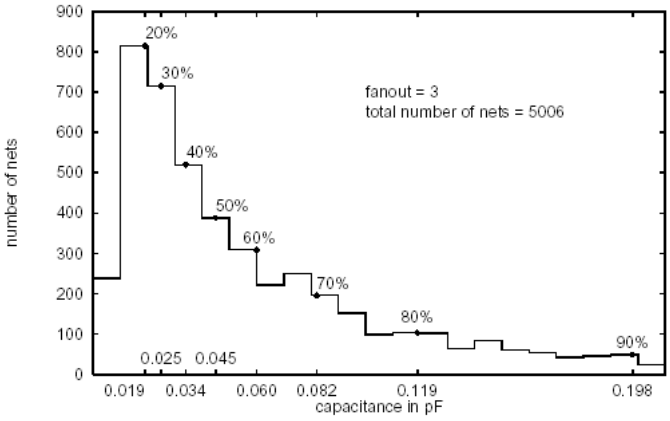


Figure 5.4: Statistic distributions of wire capacitance value

5.4.1 Wireload model

During optimization phase, design compiler needs to know both cell delays and wire delays. The cell delays are relatively fixed and dependent on the fan-ins and fan-outs of those cells. Wire delays, however, can not be determined before post-synthesis place and route. One has to estimate such delays in certain "pessimistic" way to guarantee the estimated delay is equal to or larger than the real wire delays after place and route, otherwise the circuit might malfunction. Usually, Wireload models are developed by semiconductor vendors, and this is done based on statistical information taken from a variety of example designs. For all nets with a particular fanout, the number of nets with a given capacitance is plotted as a histogram, one capacitance is picked to represent the real values for that fanout depending on how conservative criteria you want the model to be. Figure 5.4 is one example of such histogram. If a criteria of 90 percent is chosen, the value is 0.198 pF in this case.

Since the larger a design gets, the larger capacitance the wires might have, Wireload models usually come with a range of area specifications. Use the command **report_lib**, you will see the segment describing the wireload models as the example below.

```
****************************************

report : library

Library: c35_CORELIB

Version: 2003.06-SP1-3

Date : Wed Aug 18 13:06:21 2004
****************************************


Library Type            : Technology
Tool Created :2000.11 Date Created            : Date:2003/03/14
Library Version : c35_CORELIB 1.8 - bka Comments                 :
Owner: austriamicrosystems AG HIT-Kit: Digital Time Unit : 1ns

Capacitive Load Unit    : 1.000000pf Pulling Resistance Unit :
1kilo-ohm Voltage Unit            : 1V Current Unit            :
1uA Leakage Power Unit      : Not specified.
```

```
Bus Naming Style        : %s[%d] (default)


Wire Loading Model:

Name           :   10k Location       :   c35_CORELIB Resistance
:   0.0014 Capacitance    :   0.001633 Area           :   1.8
Slope          :   5 Fanout   Length   Points Average Cap Std
Deviation
------------------------------------------------------------
     1     5.00

Name           :   30k Location       :   c35_CORELIB Resistance
:   0.0014 Capacitance    :   0.001633 Area           :   1.8
Slope          :   6 Fanout   Length   Points Average Cap Std
Deviation
------------------------------------------------------------
     1     6.00

Name           :   100k Location       :   c35_CORELIB Resistance
:   0.0014 Capacitance    :   0.001633 Area           :   1.8
Slope          :   8 Fanout   Length   Points Average Cap Std
Deviation
------------------------------------------------------------
     1     8.00

Name           :   pad_wire_load Location       :   c35_CORELIB
Resistance     :   0.0014 Capacitance    :   0.001633 Area
:   1.8 Slope          :   15 Fanout   Length   Points Average Cap
Std Deviation
------------------------------------------------------------
     1    15.00


Wire Loading Model Selection Group:

   Name           : sub_micron

        Selection             Wire load name
    min area   max area
   -----------------------------------------
        0.00   1427094.00        10k
    1427094.00 4280637.00        30k
    4280637.00 219533760.00      100k


Wire Loading Model Mode: enclosed.
```

As to how to select Wireload models for wires crossing design hierarchy, synopsys provides three modes, namely, top, enclosed and segmented.

This is shown in Figure 5.5.

For the top mode, Design Compiler sees the whole design as if there is no hierarchy, and use the Wireload model used for the top level for all nets.

In the enclosed mode, Design Compiler uses the Wireload model specified for the smallest design that encloses the wires across the hierarchy.

Segmented mode is the most optimistic mode among all these three, it uses several Wireload models for segments of the wire for each hierarchy.

You can choose which mode to use by the command **set_wire_load_model** in your constraint file. Otherwise, design compiler uses the mode specified in the technology libraries or it's own default mode.

5.4.2 Constraining the design for better performance

After translating process, the design in behavioral HDL is translated into technology independent gates, after which optimization phase starts. Two major

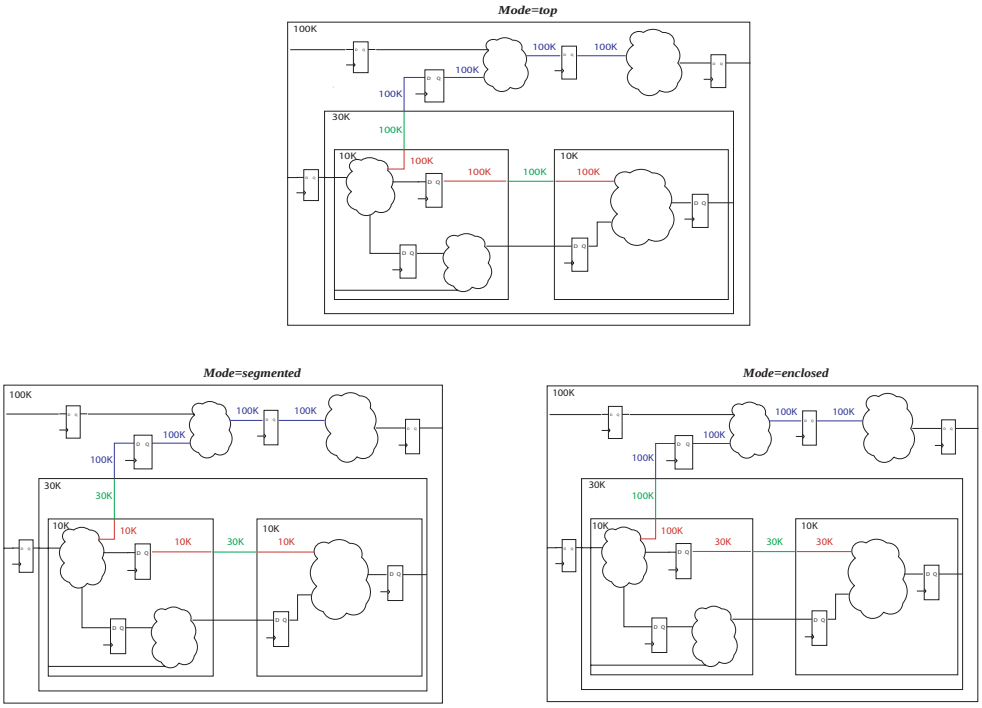


Figure 5.5: Wireload mode

tasks take place during this process, combinational logic optimization plus mapping and sequential logic optimization.

For combinational logic optimization, a phase called *technology-independent optimization* is performed to meet your timing and area goals. Two most commonly used methods to do this are logic flattening and structuring which are described in the section about logic optimization. Combinational logic mapping takes in the technology-independent gates optimized from an early phase and maps those into cells selected from vendor technology libraries. It tries different cell combinations, using only those that meet or approach the required speed and area goals.

Sequential optimization maps sequential elements (registers, latches, flip-flops, etc.) in the netlists into real sequential cells (standard or scan type) in the technology libraries, and also tries to further optimize sequential elements on the critical path by attempting to combine sequential elements with it's surrounding combinational logic and pack those into one complex sequential cells, refer to *Design Compiler Reference Manual: Optimization and Timing analysis*.

5.4.3 Compile mode concepts

Before going into details about optimization techniques, the basic concepts regarding different compile modes should be explained. It is crucial for understanding different compile strategies and scripts writing. If properly used, these compile modes will significantly reduce the compile time and improve the final

synthesis results.

- Full compilation

  The most commonly used compile mode is the full compilation. It starts the optimization and mapping process all over again no matter whether the design was compiled or not before. If prior optimized gate structure already exists, it removes it in the design and rebuild a brand new gate structure. Usually this results in better circuit implementation after the design has been changed, but such improvements come at the cost of increased compilation time. Full compilation is simply invoked by the command **compile** in the scripts.

- Incremental compilation

  Incremental compilation is useful if only part of the design fails to meet the constraints. It starts with an optimized gate structure and works on the parts with violations. It tries to re-map these parts and change the structure only if they are improved. BY working on only problematic parts without first phase of logic optimization, the compile time is reduced substantially. Use the command **compile -inc** for incremental compilation.

5.4.4 Compile strategy

Most designers would not bother with design strategy if their design is within reasonable size (for example, within 150k equivalent gates), which is the case for the designs at the department. Just take the top level design in the hierarchy and compile it, a better results among all alternative strategies can be achieved. This is so called *top-down compile strategy*. In addition to the QOR(Quality of Results), the script writing is much simpler and easy to understand.

If the design, however, turns into a huge project, like the ones a group of people work on, the run time of compiling the whole design could be prohibitive. In such cases, the synthesis process has to take place in parallel. This demands the whole design be separated into several sub-blocks, synthesized separately, and integrated together on the top level, a typical *divide-and-conquer* procedure. In practice, such bottom-up and then top-down strategy would not be manipulated as easily as it sounds. Problems always emerge in the sub-block interfacing. If snake path is present due to not fully registered sub-blocks, manually setting constraints on the sub-block borders could be laborious and apt to mistakes. Some blocks might be over-constrained while others under-constrained. This in turn will direct Design Compiler to work on the wrong part of a timing path. Solution to the above problems is automatic *design budgeting*, one capability that comes with Design Compiler.

Design budgeting can be invoked on dc_shell with the command **dc_allocate_budgets** at three different levels. The budgeting starts either in unmapped RTL level or mapped gate level or the mix of these two. It utilizes top level constraints and delay information (cell delays, Wireload models, physical data from P&R, etc.) to create a set of constraints among all hierarchies, a designer could then apply these constraints to the sub-blocks interested, and compile top-down for the sub-modules separately, and do top level compile to fix interconnect violations

for top level modules. Refer to *Synopsys User Guide, Budgeting for Synthesis* for more information.

5.4.5 Design Optimization and constraints

Design optimization is performed by Design Compiler according to user specified constraints. Two types of constraints exist for constraining your design. The following sections will discuss these in detail.

One of the two types is the design rule constraints, which are implicit constraints defined by the technology libraries. These constraints are required to guarantee the circuit to function correctly. Some examples are:

- The maximum transition time for a net, specifying a cell with input pin connected to the net to function correctly.
- The maximum fanout of a cell that drives many load cells.
- The maximum load capacitance a net could have.
- A table listing maximum capacitance of a net as a function of the transition times at the inputs of the cell.

When violations are found during the optimization process, measures are taken to meet the requirements, eg. choosing cells with different size, buffering the outputs of a driving cell. Although the design will meet design rule constraints with default values by technology libraries, the designer could still specify more restrict design rule constraints.

The other type is optimization constraints. These constraints are defined by the designer with the goal to get better synthesis results such as higher speed or less chip area. It has a lower priority over design rule constraints. In other word, design compiler can not violate design rule constraints, even if it means violating optimization constraints. Comprised of four types of constraints (namely speed, power, area, and porosity), optimization constraints are commonly used to optimize for the speed of your design.

To use optimization constraints to obtain better circuit performance, you should have in mind that your design is composed of two parts, synchronous and asynchronous paths. Synchronous paths are usually the timing paths of combinational logics between any two sequential elements (eg. registers). To specify the maximum delay of these timing paths, you only have to set clock period constraints on the clock pins of the sequential elements. Use the command **create_clock** to specify the clocks. In addition to timing paths between sequential elements, any paths starting from an input port and end at the sequential elements or paths from sequential elements to output ports are regarded as synchronous paths as well. Apart from clock period setting, input and output delays have to be specified to complete timing paths constraints. Use the **set_input_delay** and **set_output_delay** for this purpose. Asynchronous delay is defined as point to point paths with partial or no association with sequential elements. To set the constraints on these paths like maximum and minimum delays, use the command **set_max_delay** and **set_min_delay**.

In the example, only clock period is set to constrain any synchronous paths between synchronous elements under the assumptions of 0 input and output delay.

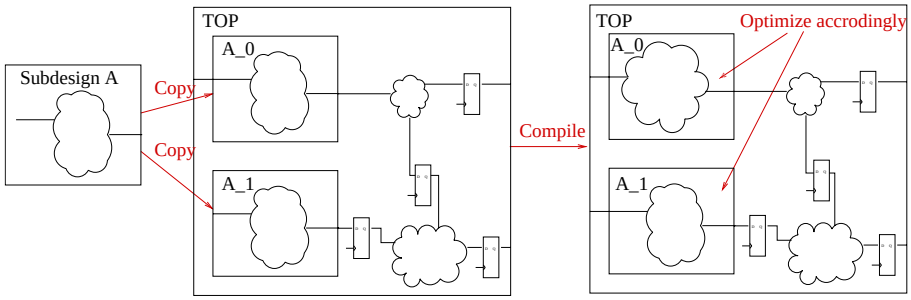


Figure 5.6: Uniquify method

5.4.6 Resolving multiple instances

Before compilation, the designer need to tell Design Compiler how multiple instances should be optimized according to the environment around. In a hierarchical design, a sub-design is usually referenced more than once in the upper level, thus reducing the effort of recoding the same functionality by design reuse. In order to obtain better synthesis results, each copy of these sub-designs has to be optimized specifically to be tailored to the surrounding logic environment. Design Compiler provides three different ways of resolving multiple instances.

- Uniquify

  When the environment around each instance of a sub-design differs significantly, it would be better to optimize each copy independently to make it fit to the specific environment around them. Design Compiler provides the command **Uniquify** to do this. The command makes copies of the sub-design each time it is referenced in the upper level of the hierarchy, and optimize each copy in a unique way according to the conditions and constraints of its surrounding logic. Each copy of the sub-design will be renamed in certain conventions. The default is %s_%d, where %s is the sub-design name, and %d is the smallest integer count number that distinguish each instance. This is shown in Figure 5.6. The corresponding sample script is as follows:

```
dc_shell> current_design top
dc_shell> uniquify
dc_shell> compile
```

- Don't touch

  There is one another way called *Don't touch* that compile the sub-design once and use the same compiled version across all the instances that reference the sub-design. This is shown in Figure 5.7. This is used when the environment around all the instances of the same sub-design is quite similar. Use the command **characterize** to get the worst environment among all the instances, and the derived constraints are applied when compiling the sub-design. A **dont_touch** attribute is then set on the sub-design so that the following compilation in the upper level will leave the all the instances intact. The major motivation of doing optimization in this way is to save compilation time of the whole design. The script is as follows:

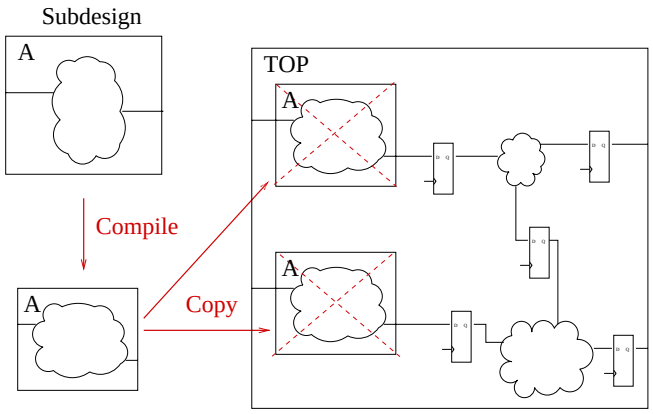


Figure 5.7: Dont_touch method

```
dc_shell> current_design top
dc_shell> characterize U2/U3
dc_shell> current_design C
dc_shell> compile
dc_shell>current_design top

dc_shell> set_dont_touch {U2/U3 U2/U4}
dc_shell>
compile
```

- Ungroup

  If the designer wants the best synthesis results in optimizing multiple instances, use the command **Ungroup**. Similar to Uniquify, Ungroup makes copies of a sub-design each time it is referenced. Instead of optimizing each instance directly, design compiler removes the hierarchy of the instance to "melt" it into its upper level design in the hierarchy. This is shown in Figure 5.8. In this way, further design optimization could be done at the logics that were originally at the instance border which are impossible to optimize with design hierarchy. The better synthesis result comes at the cost of even longer compilation time. In addition, design hierarchy is destroyed, which makes post-synthesis simulation harder to perform. The sample script is as follows:

```
dc_shell> current_design B
dc_shell> ungroup {U3 U4}
dc_shell>current_design top
dc_shell> compile
```

5.4.7 Optimization flow

Until now, enough "peripheral" knowledge has been introduced regarding design optimization. This section will move on to give an overview of the whole optimization process, intended for a better understanding of what is actually happening in synthesis process. This, to the author's view, is beneficial for constraints settings in optimization.

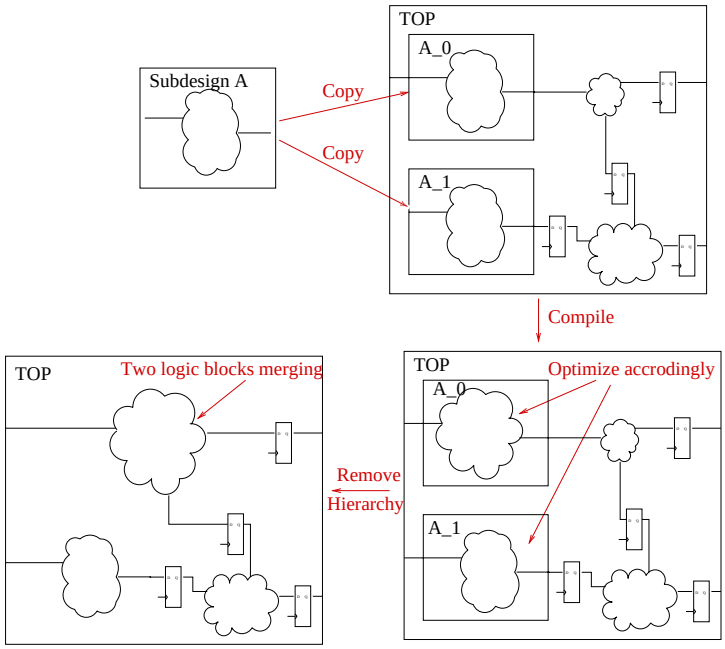


Figure 5.8: Ungroup method

In fact, design optimization starts already when it reads in designs in HDL format. Aside from performing grammatical checks, it tries to locate common sub-expressions for possible resource sharing, select appropriate DesignWare implementations for operators, or reorder operators for optimal hardware implementations, etc. These high level optimization steps are based on your constraints and HDL coding style. Synopsys refers this optimization phase as *Architectural Optimization*.

After architectural optimization, the design represented as GTECH netlist goes through logic optimization phase. Two major process happens during this phase:

- Logic Structuring

  For paths not on critical timing ranges, **structuring** means smaller area at the cost of speed. It checks the logic structure of the netlist for common logic functions, attempting to evaluate and factor out the most commonly used factors, and put them into intermediate variables. Design resource is reduced by hardware sharing. Introducing extra intermediate variables increases the logic levels into the existing logic structure, which in turn makes path delay worse. This is the default mode that design compiler optimize a design.

- Logic Flattening

  Opposite to logic structuring, *logic flattening* converts combinational logic paths into a two level, sum-of-products structure by flattening logic expressions with common factors. It removes all the intermediate variables and form low depth logic structure of only two levels. The resulting en-

hanced speed specification is a tradeoff for area and compilation time, and some times it is even not practical to perform such tasks due to limited CPU capability. By default, design compiler does not flatten the design. It is even constraints independent, meaning design compiler would not invoke it even timing constraints is present. Designer, however, can enable such capability by using the command **set_flatten**.

Gate level optimization is about selecting alternative gates from the cells in the technology libraries, which tries to meet timing and area goals based on your constraints. Setting constraints (both design rule and optimization) is already covered in the previous section. Various compile mode can be used to control the mapping process in gate level optimization.

5.4.8 Behavioral Optimization of Arithmetic and Behavioral Re-timing

For designs with large number of arithmetic operations such as multimedia applications, manually setting constraints to optimize datapath is a tedious work. Designer has to go back to HDL descriptions, manually inserting pipeline stages to the parts on the critical ranges. Specialized arithmetic components have to be coded or instantiated in the HDL file, making it harder to maintain. To address this issue, Design Compiler provides two capabilities that will automate such a process.

BOA (Behavioral Optimization of Arithmetic) is a feature that applies various optimization on arithmetic expressions. It is effective for datapaths with large number of computations in the form of tree structures and large word-length operands. BOA replaces arithmetic computations in the form of adder trees with carry save architecture. This includes decomposing a multiplier into a wallace tree and an adder. It also optimizes multiplication with a constant into a series of shift-and-add operations. Compared to similar implementations from DesignWare components, BOA can also work on much more complicated arithmetic operations and does not need to change the source code manually. Acting as a behavioral optimization techniques, it only recognizes high level synthetic operators $(+, -, *)$, and is invoked by the command **transform_csa** after elaboration (but before compile).

BRT (Behavioral Re-timing) is a technique to move registers around combinational paths to achieve faster speed or less register area. The concept of retiming is introduced in the course "*DSP-Design*". By inserting pipeline registers into timing paths, the path delays between any two registers are reduced. As a result, the total circuit could run at a higher speed at the cost of latency. The idea is easy to understand but hard to implement when this is done in the mapped gate level. Use the command **pipeline_design** provided by Design Compiler. This command will automatically place the pipeline registers on your design to achieve the best timing results according to your requirements. One thing to note is that the targeted design has to be purely combinational circuits. There is another command, **optimize_registers**, which optimizes the number of the registers by moving registers in a sequential design. Refer to *Creating High-speed Data-Path Components* for more details.

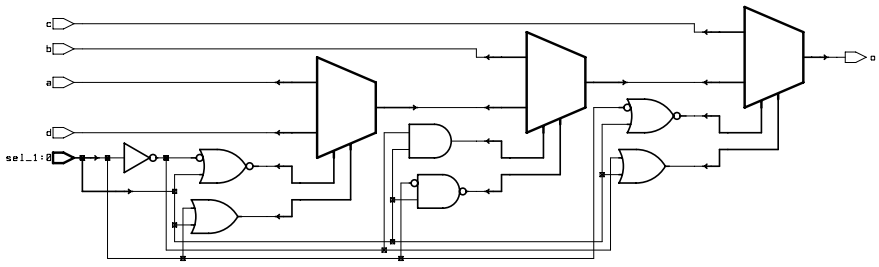


Figure 5.9: synthesis for if_bad

5.5 Coding style for synthesis

Although synthesis tools will optimize the design to a certain extent, only 10-20 percent of performance enhancement is usually expected. The higher the design level you are working on, the more performance boost you will get eventually. In this context, the most important factor for a good design lies mostly in everything in the upper level, ie, how well the algorithm is, what kind of architecture the whole system is utilized, how good the whole system is partitioned, whether a good coding style is used in VHDL.

Since this will cover everything from algorithm down to logic gates, which is beyond the range of the tutorial, only coding styles will be covered in the following section. Two examples on how the synthesis results could differ in a good and bad coding style will be shown.

5.5.1 Coding style *if* statement

This is a "classical" problem regarding ***if*** statement coding style in vhdl, which comes to different synthesis results. Below is two coding styles using *if* statement for the same functionality.

```
library ieee;
use ieee.std_logic_1164.all;

entity if_bad is
  port (
    a, b, c, d   : in  std_logic;
    sel          : in  std_logic_vector(1 downto 0);
    o          : out std_logic);
end if_bad;

architecture behave of if_bad is
begin
  process (sel, a, b, c, d)
  begin  -- process
    if sel="00" then
      o<=a;
    end if;
    if sel="01" then
      o<=b;
    end if;
    if sel="10" then
      o<=c;
    end if;
    if sel="11" then
      o<=d;
    end if;
  end process;
end behave;
```

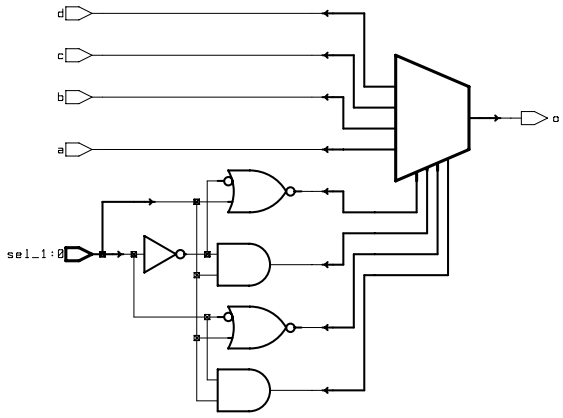


Figure 5.10: synthesis for if_good

```
library ieee;
use ieee.std_logic_1164.all;
 entity if_good is
  port (
    a, b, c, d   : in  std_logic;
    sel          : in  std_logic_vector(1 downto 0);
    o          : out std_logic);
 end if_good;

architecture behave of if_good is begin
  process (sel, a, b, c, d)
  begin  -- process
    if sel="00" then
      o<=a;
    elsif sel="01" then
      o<=b;
    elsif sel="10" then
      o<=c;
    else
      o<=d;
    end if;
  end process;
end behave;
```

Here is the synthesis results from synopsys for both designs. For if_bad design, critical path increases due to the delay propagated from input pin d through all the way to the output o. Design Compiler is not smart enough to take the advantage of all these exclusive *if* conditions, that should be translated into a single MUX. Instead, it uses priority coding for *if* statements without following *else* statements. This results in three cascaded MUXes.

5.5.2 Coding style for loop statement

For *for loop* construct in the vhdl design, Design Compiler simply unrolls the loop, making a hardware for each iteration of the loop. So, when writing such loops, the designer should aware of the hardware inference in such *for loop*, and try to use as less hardware as possible within the loop. The following is an example of two *for loop* coding styles.

```
 architecture RTL of LOOP_BAD is
    type TABLE_8_x_5 is array (1 to 8) of std_logic_vector (4 downto 0);

    begin

    process (BASE_ADDR, IRQ)

       variable INNER_ADDR: std_logic_vector(4 downto 0);
```

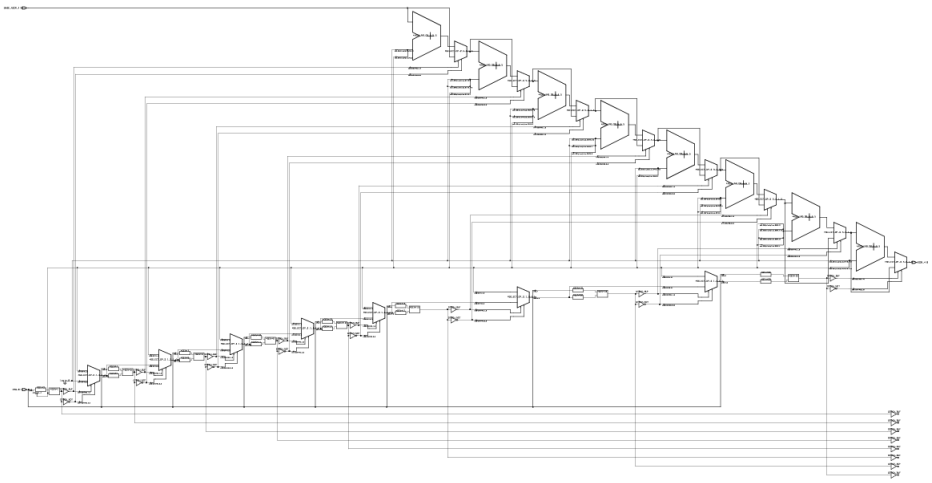


Figure 5.11: synthesis for loop_bad

```
        variable DONE: std_logic;

    constant OFFSET: TABLE_8_x_5 :=("00001","00010", "00100",
                     "01000","10000","10001","10100","11000");

 begin

 DONE := '0';
 INNER_ADDR := BASE_ADDR;

for I in 1 to 8 loop
   if ((IRQ(I) = '1') and (DONE = '0')) then
   INNER_ADDR := INNER_ADDR + OFFSET(I);
   DONE := '1';
   end if;
end loop;

ADDR <= INNER_ADDR;
end process;
end RTL;


architecture RTL of LOOP_BEST is
  type TABLE_8_x_5 is array (1 to 8) of std_logic_vector (4 downto 0);
begin
 process (BASE_ADDR, IRQ)
variable TEMP_OFFSET: std_logic_vector(4 downto 0);
variable DONE: std_logic;
 constant OFFSET: TABLE_8_x_5 := ("00001","00010","00100","01000",
                                  "10000","10001", "10100","11000");
 begin
TEMP_OFFSET := "00000";
for I in 8 downto 1 loop if (IRQ(I) = '1') then
TEMP_OFFSET := OFFSET(I);
 end if;
end loop;
 -- Calculate Address of interrupt Vector
ADDR <= BASE_ADDR + TEMP_OFFSET;
end process;
end RTL;
```

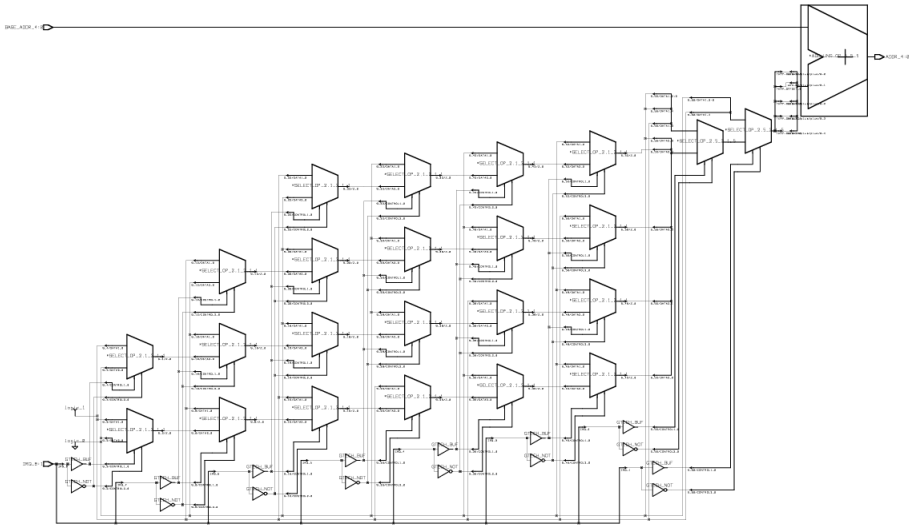


Figure 5.12: synthesis for loop_good

Chapter 6

Place & Route

6.1 Introduction and Disclaimer

This manual is intended to guide an ASIC designer carrying out the designs steps from netlist to tapeout. The reader is guided by using the Silicon Ensemble GUI and by the use of script (mac) files. However, this manual was never intended to be complete. The design steps considered in this manuscript are presented in Figure 6.1. Several digital ASIC's have been successfully produced (in our digital ASIC group) according to the presented design flow. For a more detailed description the reader is referred to the Silicon Ensemble manual.

Although every precaution has been taken in the preparation of this manual, the publisher and the authors assume no responsibility for errors or omissions. Neither is there any liability assumed for damages or financial loss resulting from the use of the information contained herein.

6.2 Starting Silicon Ensemble

Before you start SE you should ensure that your design has been tested extensively as the place and routing job can be very time consuming dependent on the design size.

The required library elements for the design flow are:

- LEF/DEF Files (usually provided from the chip manufacturer)
- Netlist in Verilog format (generated with Synopsys)

!As SE needs a lot of disk quota to do the routing you should check that you have enough free disk space**!**

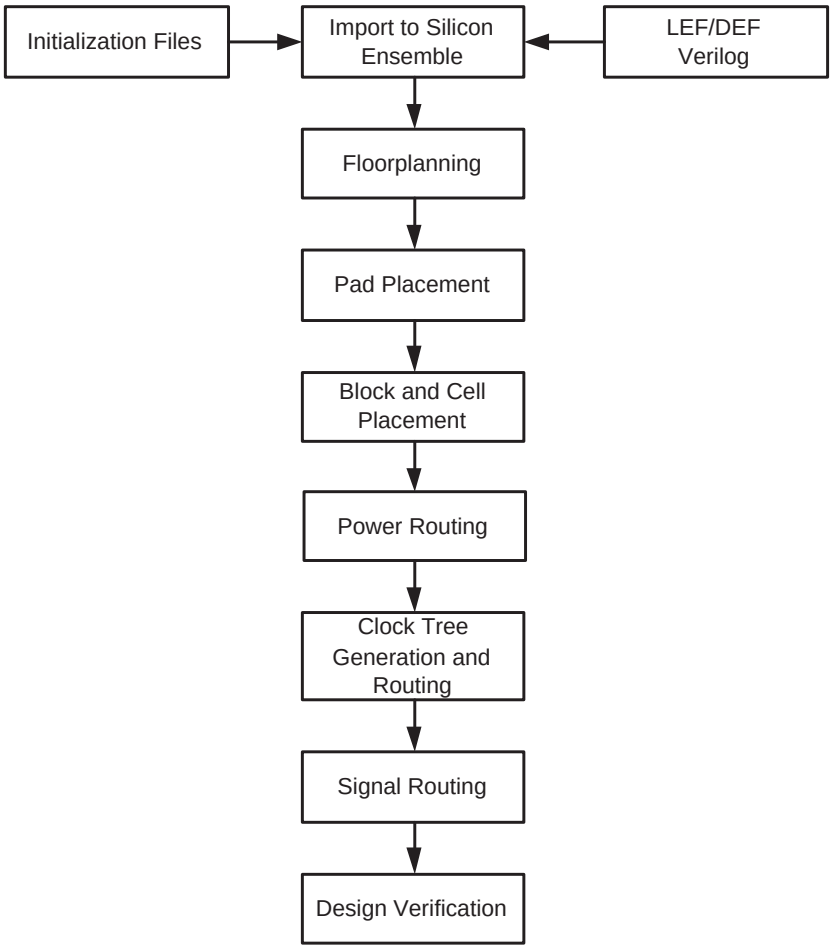


Figure 6.1: Silicon Ensemble Design Flow

To start SE go to the cadence work directory: ***cd NameofDirectory***

Setup the SE environment with ***source ....***

To view the SE manual type: ***openbook***

Start SE graphical user interface with ***seultra -m=500&***

and following actions will take place:

- 200 MB RAM will be allocated for the use of SE.
- A log file *se.jnl* will be created
- Commands in the initialization file *se.ini* will be run
- The SE GUI pops up and is ready to use

Type *seultra -h* for more information.

6.3 Import to SE

To create a database for your design the LEF files have to be imported to create a physical and logical library. LEF files contain information on the target technology, the cells and the pads. If memory cells are used the respective LEF files have to be read as well. The import can be done by using the GUI.

⇒ Select *File* → *Import* → *LEF*
Import ***cmos035tech.lef*** (process parameters), **MTC45000.lef, MTC45100.lef** (cells and pads) and ***memory.lef*** (custom memeory) in the given order and terminate each import routine with **ok**.

The description of the memory and a netlist of the design in *verilog* format needs to be imported to SE:
⇒ Select *File* → *Import* → *Verilog*
and browse for the memory verilog file *memory.v*. Repeat the same procedure and import *filter.v*. Type the name of your top module under *Verilog Top Module*. The name of the top module is created after the synthesis and can be quite long as all the generics are included in the name. Set *Compiled Verilog Output Library* to ***projdir***, see Figure 6.2.

In the next step the supply pads definition file that specifies the names of the different power pins is imported:
⇒ Select *File* → *Import* → *DEF*
and browse for the *supplyPads.def* file.
Such files contains the minimum set and a few additional supply pads. All the files required to start the SE design flow have now been imported.

⇒ Type ***save design "loaded";***
in the command line of the SE GUI.

Figure 6.2: Import of the netlist in verilog format

6.4 Creating a mac file

Instead of using the GUI to import the data, a script file can be executed to perform the same operations. All the instructions needed to set up such a script file can be found in *se∗.jnl*. The **.jnl* is a screen dump of the report window. If the structure of a script file and the SE design flow is understood, script files can be more than a competitive alternative to the GUI. If parameters needs to be changed it is more convenient to do the changes in the script file rather than doing the changes via the GUI. For future designs the script files can be easily modified and the time needed to do the place and route job is significantly reduced. Furthermore, a complete *mac* file serves as documentation and makes it possible to repeat the flow exactly.

⇒ Open a new file, e.g. *1_verilogin.mac*, with an editor, e.g emacs. Scan *se.jnl* for the instructions that have been carried out with the GUI to load the LEF, Verilog and DEF files. Paste the instructions in your newly generated file. You script file may look as follows:

```
##-- Import Library Data ##--
FINPUT LEF F LEF/cmos035tech.lef;
INPUT  LEF F LEF/MTC45000.lef ;
INPUT  LEF F LEF/MTC45100.lef ;
INPUT  LEF F LEF/memory.lef;
##-- your design.
INPUT VERILOG FILE "/../memory.v"; LIB "cds_vbin" REFLIB "MTC45000 MTC45100
cds_vbin";
INPUT VERILOG FILE "/../your_design.v"; LIB "projdir" REFLIB "MTC45000 MTC45100
cds_vbin";
DESIGN "projdir.your_design:hdl" ;
INPUT DEF F DEF/supplyPads.def ;
SAVE DESIGN "LOADED" ;
FLOAD DESIGN "LOADED";
```

Change the first command (in your *mac* file) *INPUT* to *FINPUT*. The difference between these commands is that it is possible to add data to an existing library with *INPUT* but not with *FINPUT*. However, as SE provides a huge set of commands, are variations in commands that result in the same design task possible.

⇒ Select *File* → *Execute* → *1_verilogin.mac*

All the necessary files have been imported as it has been equivalently done with the GUI.

6.5 Floorplanning

The floorplanning is done by using a combination of automatic functions provided with the GUI in combination with modifications done by the user, see Figure 6.3. By ***Initialize Floorplan*** following actions will take place:

- A starting floorplan is created by an estimate of the required layout.
- Global and detailed routing grids are created.
- The core rows are created
- Sites for corner cells are created if necessary

⇒ Select *Floorplan* → *Initialize Floorplan* and Figure 6.3 pops up.

In Figure 6.3 the design statistics such as Number of Cells, IO's, Nets is provided by SE. The aspect ratio is set to one by default, which is square.

The distance of the IO to the core needs to be set, e.g. 150μ in both cases. The distance has to be large enough to leave room to route power and ground wires.
⇒ **Select** *Flip Every other row* and *Abut Rows* to create rows where VDD and GND alternates. Set the value for *Block Halo Per Side* (distance between blocks and rows) to 30μ for this design. The calculate button describes the floorplan in terms of number of rows, row utilization, chip area etc., before creating the floorplan. The floorplan, as described in *Expected Results*, will be created by confirming your settings with *apply* or *ok*.

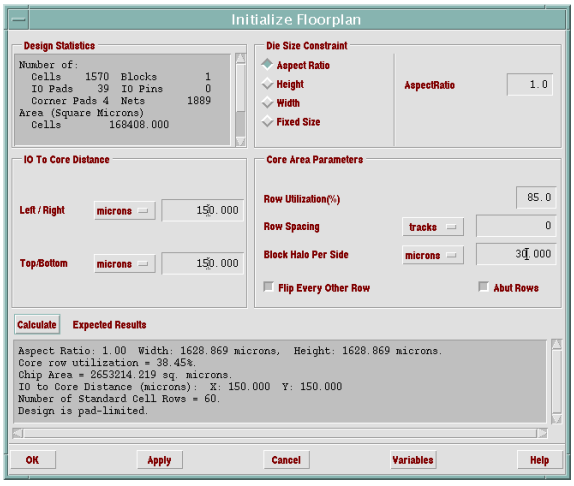


Figure 6.3: Initialize Floorplan Dialog Window

You can try different setting, e.g. IO to core distance, and observe how the floorplan changes. Before continuing in the design flow return to your initial settings for the floorplan. A floorplan and a memory cell are visible now, see Figure 6.4.

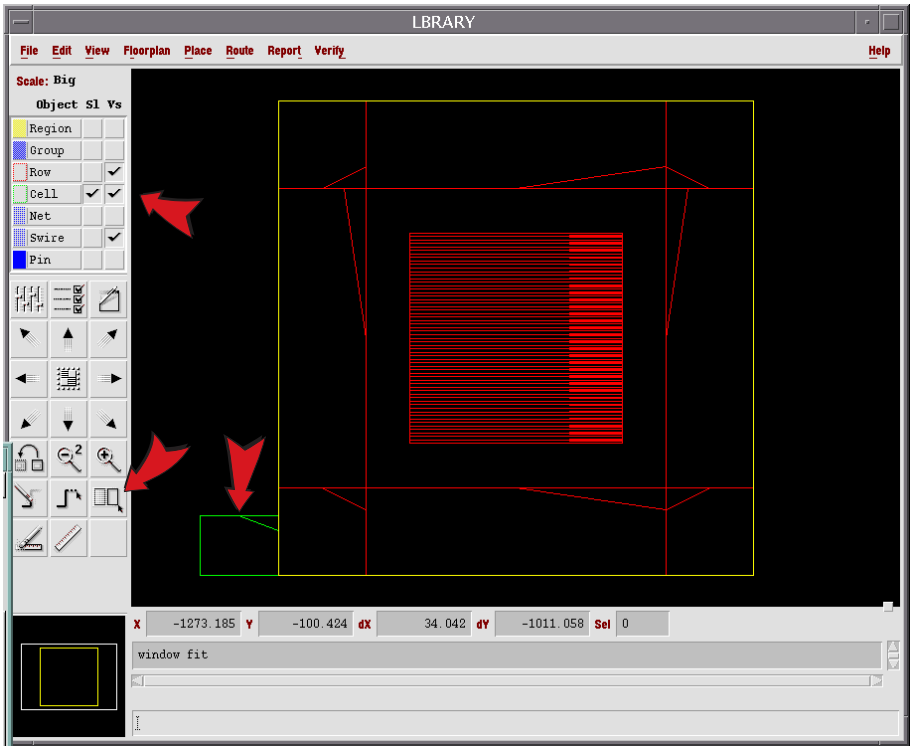


Figure 6.4: Initialized floorplan with memory cell

6.6 Block and Cell Placement

The memory cell, currently placed on the lower left site, needs to be placed on the core, see Figure 6.4.
⇒ Activate *cell* as selectable,
⇒ Select *Move* on the icon panel on the left hand side.
Drag the memory cell to one of the corners on the core.
Assure that the memory supply faces to one of the core borders (the memory pins become visible if net is selected as visible). Thus the memory power ring has only to be drawn on two or three sides of the memory. The coordinates for cell orientation are illustrated in Figure 6.5.

⇒ Select *Floorplan* → *Update Core Rows* → *ok*.
The rows underneath the memory are cut and the rows surrounding the memory are shortened (halo per side) to give space for the memory power ring.

⇒ Scan *se.jnl* for the instructions and paste them a new file, e.g. *2_floorplan.mac*.

Next, the power pins and signal IO's needs to be placed.
⇒ Select *Place* → *IO's* → *Random* → *ok*.
⇒ Select *Place* → *IO's* → *I/O constraint file* and select *Write*.

A file *ioplace.ioc* is generated that specifies the location/orientation of the pads. The order of the cell names in the generated file is the placement order: Left → right in the top and bottom rows and bottom → top in the left and right rows.
⇒ Select *Edit* if you want to do changes in the pad placement. The names of the IO's as specified in the vhdl code are listed with their respective net name in the verilog file.
⇒ Finally, place the IO pads with *ok*.

The aim of placing the pads manually by using *ioplace.ioc* is to make your ASIC fit better with surrounding blocks. However, by setting such constraints it might be a much harder job for the router and some configurations might not be routeable at all. As a rule of thumb: Place the supply pads evenly in the center of each side of the chip. Interrupt a series of IO pads with pad supply to obtain a better driving of the pads. The more supply pads the better. Avoid core supply pads in the corners of the ASIC.

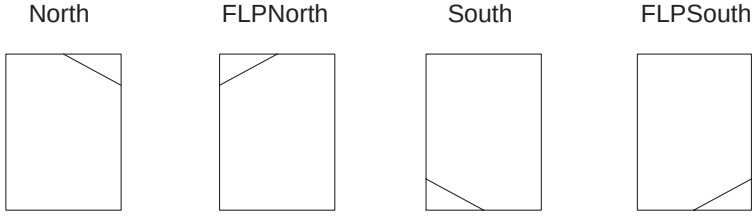


Figure 6.5: Initialized floorplan with memory cell

The floorplan is now ready to insert the cells. The placement will be done automatically (QPlace). Dependent on the design size this can take some time.
⇒ Select *Place* → *Cell* and *ok*
and the cells will be placed on the rows, see Figure 6.6.
⇒ Save you design as *placed*.
⇒ Scan *se.jnl* for the instructions carried out under block and cell placement and paste them in a new file, e.g. *3_place.mac*.

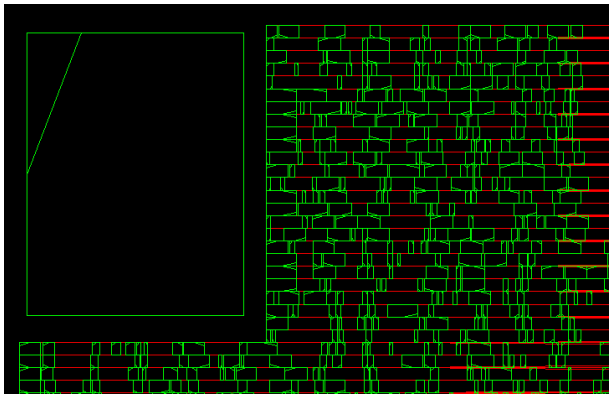


Figure 6.6: Closeup of the memory and placed cells

6.7 Power Routing

The power ring structures surrounding the core and the memory cell has to be planned and routed. With *Plan Power* the following design steps are done:

- Power paths can be planned and modified before routing them
- Generation of power rings that are surrounding all blocks and cells
- Creates stripes over standard cell rows
- Connects created ring, stripes and power pads

To start *Plan Power*
⇒ Select*Route* → *Plan Power* → *Add Rings*.
Only *vdd* and *gnd* is needed as core supply. Delete other supplies in the dialog window, see Figure 6.7. The order of the rings can be defined by the order of the net names. The first net name is the left or bottom-most ring.
⇒ Select the type of metal you want to use for the power ring. To prevent routing violations choose ***metal3, metal4***, as at least metal1 is already reserved for signal routing. Specify the width for the core supply ring and channel, located between the memory and core, to 30μ and 10μ, respectively. If the IO to core distance (defined during initialize floorplan) is to narrow you will get a warning.

In order to improve the power supply throughout the rows, wires crossing the core can be added.
⇒ Select *Route* → *Plan Power* → *Add Stripes*.
With the dialog box that pops up, see Figure 6.9, the following can be specified:

- layer for the stripes
- width and spacing
- number of stripes (equally spaced across the core)

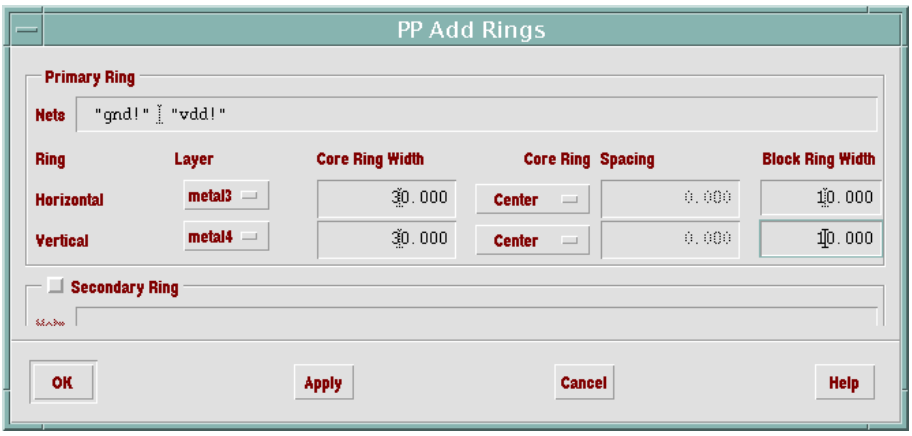


Figure 6.7: Dialog box for Add Rings

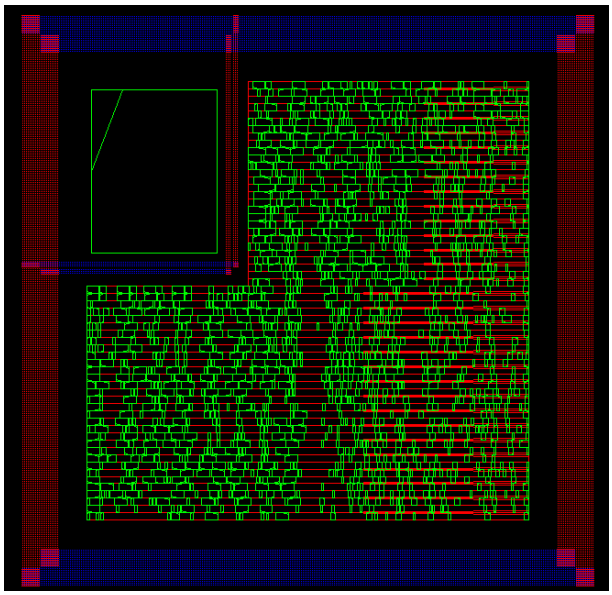


Figure 6.8: Core with power rings

To create stripes across the core you may use following parameters:

layer	metal4
width	5.000
spacing	1.000
mode	all
Set to Set spacing	100.000

The strips are only generated to the edge of the core and will be connected later to the power rings.
⇒ Scan *se.jnl* for the instructions carried out during power routing and paste them in a new file, e.g. *4_powerroute.mac*.

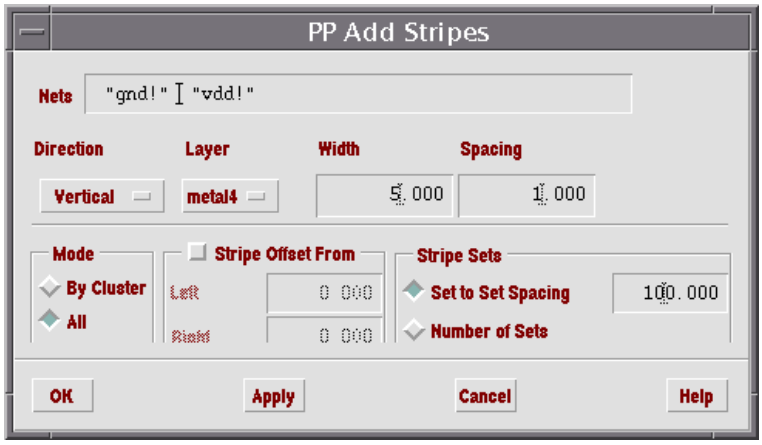


Figure 6.9: Dialog box for Add Stripes

Filling the gaps between the IO pads

The gaps between the the IO pads needs to be closed to maintain continuity in the power supply ring. This can be done by inserting fillercells (dummy-cells) in the IO ring, see Figure 6.10.
⇒ Select *Place* → *FillerCells* → *AddCells*
to open the dialog box. Fill in the Model as **IOFILLER64** and the prefix as **FP64**.
⇒ Select *North, Flip North, South, Flip South* under Placement, see Figure 6.11. Iterate this procedure with narrower filler cells (32,16,8,4,2,1). The different filler cells can be found in *MTC45100.lef*. Observe how the gaps between the IO pads are vanishing.

⇒ Scan *se.jnl* for the instructions and paste them in a script file.
It is recommended to save the commands for the filler cells in a separate script file, e.g. *IOdummys.mac*, as the script can easily be reused in the AMIS 0.35 CMOS technology. You only need to set the area parameter to a rather big value to make it applicable for future designs.

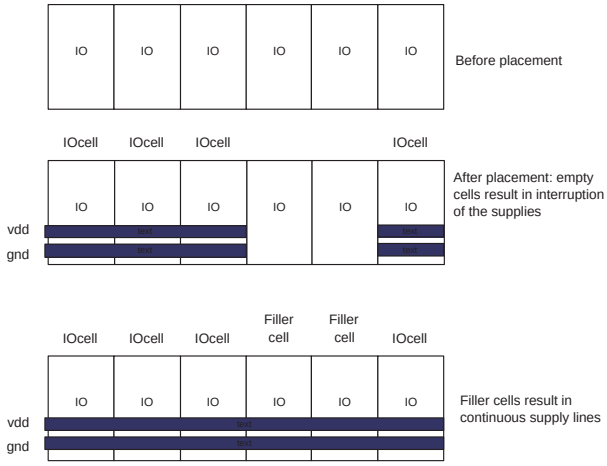


Figure 6.10: Adding Filler Cells

6.8 Connecting Rings

To connect the rings use the *Connect Rings* command. The following is completed with the *Connect Ring* command:

- Connections between IO power pins within IO rows
- Connections between CORE IO ring wires and the IO power pins
- Connections between stripes and core rings
- Connections between block power pins and the core IO wires

⇒ Select *Route* → *Connect Ring*
Assure that *Stripe, Block, All Ports, IO Pad, IO Ring, Follow Pins* are selected. All the supplies are now connected, see Figure 6.12.
⇒ Save the design as connected.

⇒ Scan *se.jnl* for the instructions carried out during connecting the rings and paste them in a script file, e.g *5_connectrings.mac*.

Export your design as *b4CT.def*.
⇒ *File* → *Export* → *DEF*

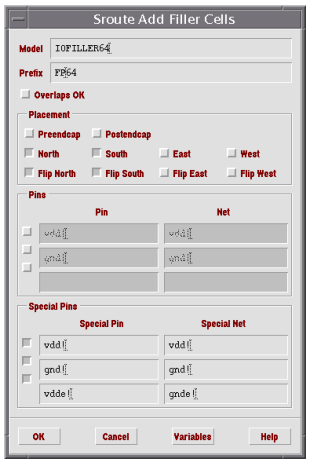


Figure 6.11: IO Filler Cell Window

6.9 Clock Tree Generation

As an optional tool to SE comes a clock-tree generator (CT-Gen). It can be controlled with the GUI or as a stand-alone tool. We will use CT-gen as a stand-alone tool. You need to export your design as *b4CT.def* before generating the clock tree.

It is recommended to route the clock net only after all the core cells are connected to the supplies as eventual routing violations might be prevented.

With the use of data from your DEF file CT-Gen produces clock tree(s), computes wire estimates and inserts buffers to reduce clock skew. The input DEF file must be fully and legally placed. The output of CT-Gen is a fully placed netlist with new components such as buffers and inverters.

CT-Gen provides following features:

- Construction of an optimized clock tree.
- Minimizes skew.
- Produces code for Silicon Assemble router.
- Controls buffer and inverter selection

Following constraints for the clock tree need to be defined before using CT-Gen:

- Clock root.
- t_{rise} and t_{fall} of the waveform.
- Minimum/maximum insertion from clock root to any leaf
- Maximum skew between insertion delay and any leaf pin

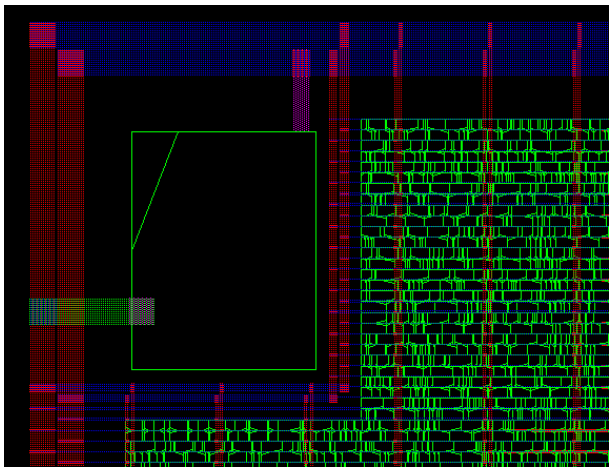


Figure 6.12: Closeup of the layout after the Connect Ring command

⇒ Open the file *ctgen.constrainsts* and set the constraints (see the comments for explanation). In the *ctgen.commands* parameters such as the input LEF file, supply definitions need to be set.

The clock generator tool uses the library *CTGEN* (defined in ctgen.commands) as work library.
⇒ Delete all the files in *CTGEN* that may remain from previous designs with ***\rm -r CTGEN/***.

⇒ Start CT-Gen with ***ctgentool ctgen.commands***

If the the clock generation was successful, a number of report files will be produced in *CTGEN/rpt*. The report files are of the syntax <stage>.<type> where <stage> is one of

- initial: without modifications.
- clock: after the clock tree has been added.
- final: when the clock tree components are placed correctly.

and <type> is one of

- analysis: complete description of best and worst path found.
- timing: the same without the details.
- trace: describes the clock path/tree with its components.
- violations: lists any violations against the given constraints.

The DEF file generated with CT-Gen needs to be loaded to SE. However, the technology and memory LEF files needs to be imported first.

⇒ Open the *verilogin.mac* and paste the instructions in the SE command line.

6.10 Filler Cells

The row utilization as calculated during floorplan initialization is visible on the floorplan. The red gaps between the cells need to be closed to avoid design rule violations. By placing filler cells continuity is maintained in the rows. Furthermore, substrate biasing is improved by filler cells substrate connections.

⇒ Select *File* → *Import* → *DEF* and select *afterCT.def*
⇒ Select Place → *FillerCells* → *AddCells*
to open the dialog box. Fill in the Model as **FILLERCELL16** and the prefix as **FC16**. Select *North, Flip North, South, Flip South* under Placement, see Figure 6.13.

⇒ Open a new file, e.g *6_coredummy.mac* and scan *se.jnl* for the place filler cell command. Paste the command in your newly generated file. Iterate this procedure and change the width of the filler cells (8,4,2,). The different filler cells can be found in *MTC45000.lef*.

!With increased area in your *6_coredummy.mac* file such file can be reused in other designs!

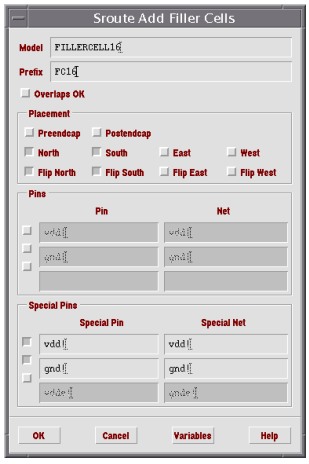


Figure 6.13: Core Filler Cell Window

6.11 Clock and Signal Routing

For final routing the **Envisia Ultra Router** (WRoute) is used. WRoute is included in the SE package and can be used with the GUI or as stand-alone tool. The benefits of WRoute are:

- 3 to 20 times faster than GRoute/FRoute
- Timing driven global and final routing
- Automatic search and repair in final routing
- Supports prerouted nets
- Final clean up

WRoute requires a lot of memory. If it crashes decrease memory allocation for SE as WRoute is run outside SE and thus will get more memory. **!**It is recommended to route the clock before routing the remaining signals (shortest wiring possible)**!**

⇒ Select *Route* → *Clock route*. Assure that *ALL* is selected and proceed with *ok*

To start the final routing
⇒ Select *Route* → *WRoute* to open the dialog box. Select *Global and Final Route* and *Auto Search and Repair*.

⇒ Scan *se.jnl* for the instructions carried out during final routing the rings and paste them in a script file, e.g *7_finalroute.mac*.

If the final routing is done identify different parts of the ASIC.
⇒ Select *nets* as *visible*.
Choose one or several pins and find the clock signal. Trace the highlighted signal path and identify the cell where the net ends.

⇒ Select *Edit* → *Find*. Choose *net* as *type*. Specify the net name as: ***n****xxxx*∗.
The ∗ option ⇒ Selects every netname that starts with nxxx.
⇒ Select *highlight* and the entire clock tree becomes visible.

Scan the last lines in *se.jnl* for *"Total wire length"* and *"Total number of vias"*.
⇒ Select *Route* → *WRoute*. **Enable** *Incremental Final Route*.
⇒ Select *Options* **and enable** *Optimize Wire Length*. *OK* in both boxes.
After the improved routing is done find the new *"Total wire length"* and *"Total number of vias"*. Check if the design could be improved.

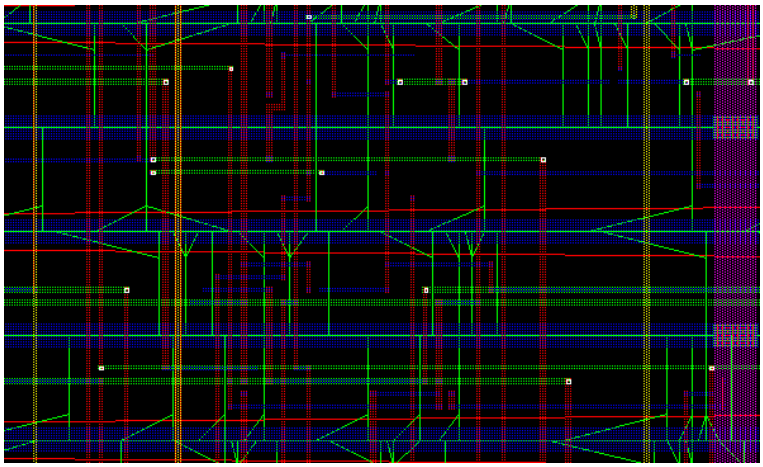


Figure 6.14: Closeup of the final layout

6.12 Verification and Tapeout

As a last step in the SE design you may test if the place and route process went smoothly. To check if there are any unconnected pins or routing violations
⇒ Select *Verify* → *Connectivity*.
Assure that *Types* and *Nets* are set to *ALL*.

If no errors are found the design is ready for tapeout.

Tapeout files

The finished design needs to be saved as *DEF* or as *GDS II* format. However, for tapeout, the manufacturer requests the *GDS II* format.

⇒ Select *File* → *Export* → *GDS II*.

⇒ Select a name of the GDS-II file and a name of the Report file.
A map-file is used to translate the design to *GDS II*.
⇒ Choose the map-file *cmos035gds2.map* and terminate with *OK*.

The *GDS II* file is ready to send to the manufacturer. The manufacturer replaces the cell abstracts with layout cells.

6.13 Acknowledgements

Thanks to Stefan Molund for setting up a system for our needs. Thanks to Shousheng He, Anders Berkemann, Fredrik Kristensen and Zhan Guo for contributing with their experience and knowledge.

Chapter 7

Optimization strategies for power

7.1 Sources of power consumption

Two major sources of power consumption are considered: *Dynamic* and *static* power. Dynamic power is dissipated when a circuit is active, that is, the voltage on a net changes due to changes at the input. However, transitions on a cell input do not necessarily result in a logic transition at the output. Hence, one further distinguishes between *switching* and *internal* power consumption. On the other hand, static power is considered to be consumed when the circuit is inactive, that is, not switching. As technology is scaled down, static power consumption starts to be the major part of the power consumption even during switching.

7.1.1 Dynamic power consumption

Switching power

The switching power is due to charging and discharging of capacitances and can be described by superposition of the individual switching power contributed by every node in a design. The load capacitance as seen by a cell is composed of net and gate capacitances on the driving output. Since charging and discharging are result of logic transitions, the switching power increases as logic transitions increase.

Internal power

The internal power consumption is due to voltage changes on nets that are internal to a cell. Here, short-circuit power is the major contributor and often used synonymously. This power is consumed during a momentary short caused by a non-perfect signal transition, that is, nonzero rise or fall time. In a static CMOS cell, both P and N transistors are conducting simultaneously for a short time, providing a direct path between V_{dd} and ground.

Internal power consumption of a cell is said to be both state and path dependent. Clearly, a complex cell can have several levels of logic and hence

transitions on different input pins will consume a different amount of internal power, depending on the number of logic levels that are affected by the transition, as exemplified in Figure 7.1. Input A and D can each cause an output transition at Z. However, D only affects one level of logic, whereas A affects all three, thus consuming more internal power.

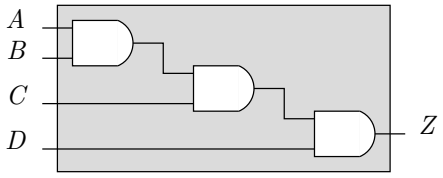


Figure 7.1: Cell with different logic depths.

A typical example for a cell with state dependent internal power is a RAM cell. It consumes a different amount of internal power, depending on whether it is in read or write mode.

7.1.2 Static power consumption

In digital CMOS technology, static power consumption is caused by leakage through the transistors. In [8], six different types of leakage are described. The current I_1 in Figure 7.2 comes from the pn-junction reverse-bias, I_2 is a combination of subthreshold current and channel punchthrough current, I_3 is gain-induced drain leakage and finally, I_4 is a combination of oxide tunnelling current and gate current due to hot-carrier injection.

In CMOS circuits currently used today, the major sources of leakage are the subthreshold current (part of I_2) and gate leakage due to oxide tunnelling (part of I_4). In future CMOS technologies, the gain-induced drain leakage (I_3), and the pn-junction leakage (I_1) may also become important factors.

For submicron CMOS technology, the supply voltage is decreased to reduce electrical field strengths and power dissipation [9]. Since the ratio of supply voltage to threshold voltage affects the circuit delay, the threshold voltage is also decreased to sustain an improvement in gate delay for each new technology generation [8]. Leakage power due to subthreshold current increases exponentially with threshold voltage scaling [8], which makes leakage increasingly troublesome. Traditionally, leakage has been a rather small contributor to the overall power consumption in digital circuits. However, the increase in static leakage power cannot be ignored in CMOS circuits of today and will be an even greater issue in the future.

7.2 Optimization

Where and when to apply power optimization strategies is simply answered, as early as possible and at the highest possible abstraction level. Figure 7.3 shows that the attainable power savings are the greatest at system level and still significant downto register transfer level (RTL)—before the design is committed to a specific technology. However, the highest accuracy on a power estimate is at gate and transistor level. Despite possible savings at higher abstraction levels,

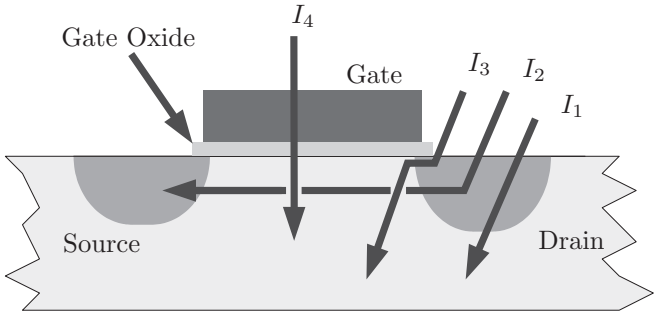


Figure 7.2: Leakage currents in a MOS transistor.

most IC's are still developed at RTL due to the maturity of design, synthesis, and verification flows that come along with it.

Figure 7.3: Power savings and accuracy attainable at different abstraction levels [10].

RTL power estimation is usually fast and gives early answers to the questions:

- Which architecture consumes least power?
- Which module in a design is consuming the most power?
- Where is power being consumed within a given block?

Doing such an estimation does not introduce much overhead to your design flow and should be carried out if power is a serious objective in your design, which is usually true. Based on the obtained results, design modifications can be carried out[1]. Following are some of the options that can be thought of.

7.2.1 Architecural issues—Pipelining and parallelization

The dynamic power consumption P_d for a CMOS design depends on the operating speed as

$$P_d \propto C_L \cdot f \cdot V_{dd} \cdot V_{swing} \approx C_L \cdot f \cdot V_{dd}^2, \tag{7.1}$$

where C_L is the total capacitive load on the chip that is switched, V_{dd} is the operating voltage, V_{swing} is the voltage swing, and f is the clock frequency. For simplicity, it is assumed that the voltage swing is equal to V_{dd}.

[1] From our experience, RTL power estimation in Synopsys involves basically the same steps as the more accurate gate-level estimation and will hence be neglected in favour of the latter.

Let f_{req} be the frequency that is required at the output of a design to achieve a desired throughput. Then, the architecture of the design will affect both f_{req} and thus the dynamic power consumption, see Figure 7.4 [11].

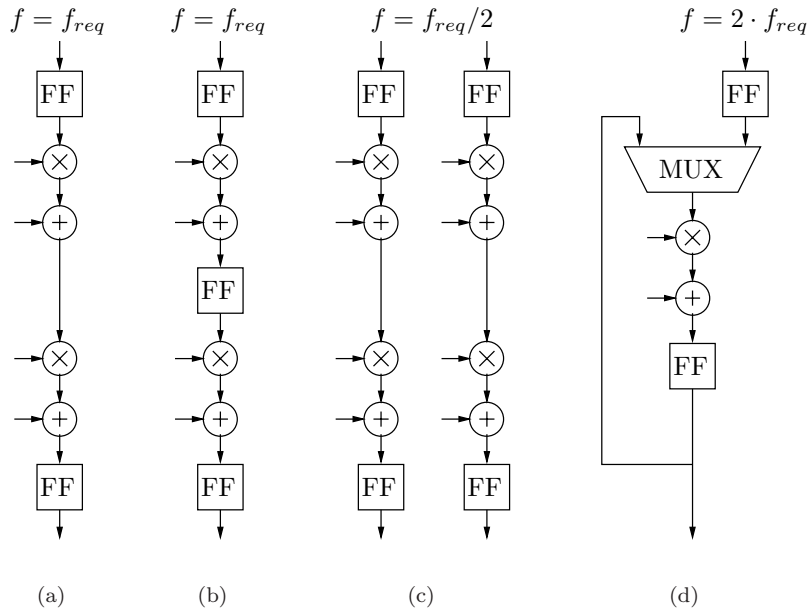


Figure 7.4: HW mapped (a), pipelined (b), parallel (c), and time shared (d) implementation.

The hardware mapped and pipelined designs both produce one sample per clock cycle and hence have f equal to the required frequency f_{req}. Note that the pipelined design has a shorter critical path and can therefore reach a higher maximum clock frequency. The parallel design produces two samples per clock cycle and f can be reduced to $f_{req}/2$, while the time shared design has to double f to reach f_{req}.

Comparing the hardware mapped and the parallel implementation, it seems as if there is nothing to gain from a power perspective by decreasing f since C_L increases accordingly. However, with a lower clock frequency, V_{dd} can be decreased since

$$f = \frac{1}{t} \tag{7.2}$$

and

$$t_{pd} \propto \frac{C_{crit} V_{dd}}{K(V_{dd} - V_t)^{\alpha}}, \tag{7.3}$$

where t_{pd} is the propagation delay, K and α are constants, C_{crit} is the capacitive load in the critical path, and V_t is the threshold voltage. K, α, and V_t depend on the silicon process. Note that α accounts for velocity saturation in processes characterized by short channel devices. Consequently, a circuit designed for high

speed can, when operated at low speed, reduce the power consumption below that of a circuit only designed to operate at low speed.

As an example, consider the designs from Figure 7.4(a) and Figure 7.4(c). The latter produces two samples each clock period and thus the frequency can be halved while the original throughput is maintained. With (7.1) and (7.3) and, for simplicity and rather traditional, V_t is assumed to be much smaller than V_{dd} and $\alpha = 2$, the dynamic power consumption is calculated as

$$f_{par} = \frac{f_{org}}{2}. \tag{7.4}$$

Hence, with (7.2) and (7.4)

$$t_{par} = 2 \cdot t_{org}. \tag{7.5}$$

Let the capactive load of a multiplier, an adder, a register, and a multiplexer be $24c$, $3c$, $1c$, and $1c$, respectively. Then, the total load of the original design is $C_{L,org} = 56c$ and for the parallel design $C_{L,par} = 112c$. The load along the respective critical path is then $C_{crit,org} = 55c$ and $C_{crit,par} = 56c$.

$$t_{par} \approx \frac{C_{crit,par}V_{dd}}{K(V_{dd} - V_t)^2} \approx \frac{C_{crit,par}}{KV_{par}} = \frac{56c}{KV_{par}} \tag{7.6}$$

$$t_{org} \approx \frac{C_{crit,org}V_{dd}}{K(V_{dd} - V_t)^2} \approx \frac{C_{crit,org}}{KV_{org}} = \frac{55c}{KV_{org}} \tag{7.7}$$

Substituting (7.6) and (7.7) into (7.5) yields

$$\frac{56c}{KV_{par}} = \frac{2 \cdot 55c}{KV_{org}} \Rightarrow V_{par} = \frac{V_{org}}{1.96}, \tag{7.8}$$

and finally, using (7.1),

$$\frac{P_{par}}{P_{org}} = \frac{C_{L,par}f_{par}V_{par}^2}{C_{L,org}f_{org}V_{org}^2} = \frac{112c\frac{f_{org}}{2}(\frac{V_{org}}{1.96})^2}{56cf_{org}V_{org}^2} \approx 0.26. \tag{7.9}$$

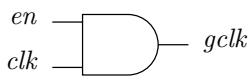
The parallel design has the same throughput at approximately a quarter of the original design's power consumption and can achieve double throughput at double power consumption. Note that the approximation $V_t \ll V_{dd}$ becomes too rough and unreliable as silicon processes shrink since V_t does not scale with V_{dd}. Hence, the power savings will not be as large as in the presented example for processes below 0.18μm. Also, α approaches numbers smaller than 2, for example, between 1.3 and 1.5 for a 0.25μm process [12].

The drawback with the parallel design is that the amount of hardware is doubled. In addition, if the design should be able to switch between high throughput/power and low throughput/power mode during run time, an active power controller will be required. To design a device with active power control will increase the number of constraints on the design and add additional verification time. A cell library that is characterized for multiple voltages is needed as well as hardware to generate the voltage levels and logic to control it. However, it is possible and an example of a commercial processor with active power control is found in [13].

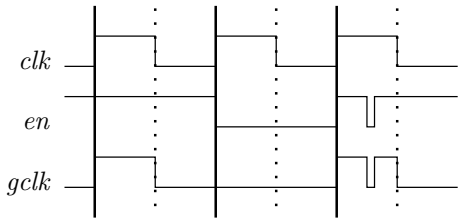
To insert additional pipeline stages in the original design has the same effect as parallelizing, either the throughput is increased or the power consumption is reduced. However, there are some differences worth noticing. First and most important, the amount of additional hardware is normally much less than in the parallel case since only some extra registers are needed. Secondly, whereas parallelizing is only restricted by area, pipelining has an upper limit to the number of pipeline stages that can be inserted. At some point there will be no more combinational logic that can be separated by pipeline registers. Finally, for pipelining to be really efficient the delay through each stage should be balanced since frequency and voltage are proportional to the critical path. In addition, pipelining increases latency, that is, the number of clock cycles from which data is present at the input until the corresponding result is presented at the output.

7.2.2 Clock gating

When a design increases in size and functionality there is a great chance that parts of the hardware are only used at certain times or in certain operation modes. These blocks will consume switching power and contribute with unnecessary load to the clock tree, even though the result is unused. One simple way to deal with this problem is to gate the clocks into these blocks, that is, turn off the clocks. In addition to reducing the clock load it will prevent unnecessary switching activity since almost all hardware modules have registers at their input.



(a)



(b)

Figure 7.5: A simple clock gate (a) and its timing diagram (b).

The simplest clock gate is implemented as an AND-gate, as shown in Figure 7.5(a). The timing diagram is shown in Figure 7.5(b). When the enable signal *en* is low the gated clock is turned off. This design is simple but sensitive to glitches, which must be avoided under all circumstances.

A safer, but larger, implementation is shown in Figure 7.6(a). Here, *en* is latched in order to avoid transitions while the clock signal is high and glitches on the output. To turn off the gated clock for one clock cycle, *en* is lowered at

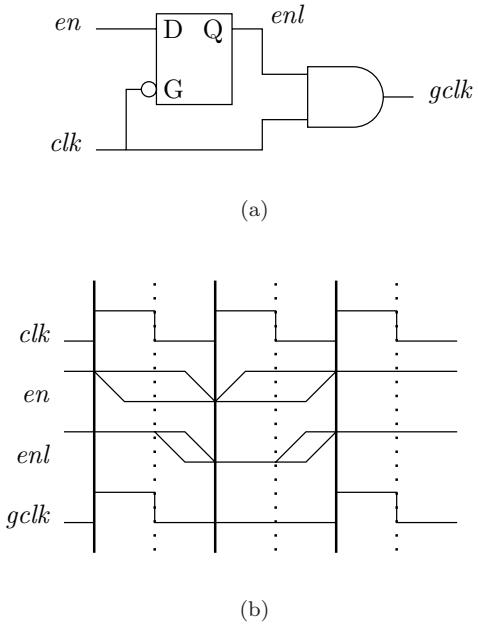


Figure 7.6: A latched clock gate (a) and its timing diagram (b).

any time in the preceding clock cycle and then raised again in the next clock cycle, as shown in Figure 7.6(b). This design is safer since the latch is only transparent when the clock is zero keeping the output of the AND-gate at zero.

7.2.3 Operand isolation

Operand isolation, also called guarded evaluation [14], prevents unnecessary operations to be performed. However, instead of halting the clock signal, datapath signals are directly disabled and hence no new results are evaluated.

Traditionally, datapath operators are always active. Due to changing inputs, they dissipate power even when the output of the operators is not used. In a particular clock cycle, the datapath operator output is not used if it is an unselected input to a multiplexer or if the datapath operator is an input to a register that is currently disabled.

With the operand isolation approach, additional logic (AND or OR gates) called isolation logic is inserted along with an activation signal to hold the inputs of the data-path operators stable whenever their output is not used. Isolation candidates can be identified in the HDL source by using a pragma annotation or at the GTECH level by using commands in the synthesis script.

```
do_operand_isolation = true
read -f vhdl /simple.vhd
set_operand_isolation_style -logic or
set_operand_isolation_cell DW_MULT
set_operand_isolation_slack 5
/* define constraints */
```

```
compile
```

In Synopsys, operand isolation automatically builds isolation logic for the identified operators. A rollback mechanism is also provided to remove the isolation logic if you determine that the delay penalty is not acceptable.

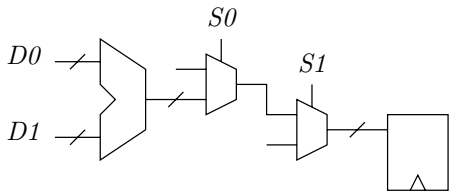


Figure 7.7: Datapath operator and selection circuitry.

Consider the example from Figure 7.7. Here, the result from the datapath operator is not used in further steps if *S0=0* or if *S1=1*. By creating an activation signal (*AS*) based on this observation, the inputs to the operator are kept stable, see Figure 7.8.

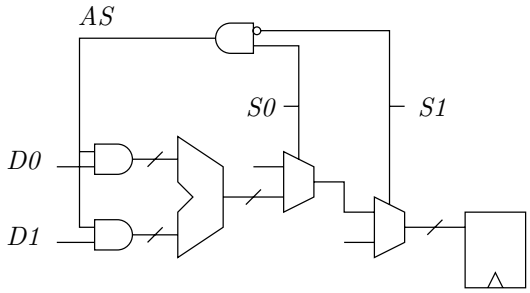


Figure 7.8: Operands isolated from the datapath operator.

For inputs that mostly stay at logic 1, use OR gates to create the operand isolation since this holds the inputs to the operator at logic 1 during inactive mode, minimizing switching activity at the transition from active to inactive mode. Similarily, AND gates are used for inputs that mostly stay at logic 0.

Ideal candidates for operand isolation are arithmetic modules such as arithmetic logical units (ALUs), adders, and multipliers that frequently perform redundant computations.

7.2.4 Leakage power optimization

While the active power consumption varies with the workload and can be locally eliminated using for instance clock gating, power consumption due to leakage remains, as long as the power supply voltage is switched on. For burst mode applications, such as mobile phones, a large part of the system is in idle mode most of the time. This makes the energy consumed by leakage a large contributor to the overall energy consumption and, thus, making simple saving schemes such as clock gating less attractive.

One efficient and often-used method to combat leakage is to use dual or variable threshold technologies [15, 16]. In a dual threshold technology, there

are typically two versions of each digital gate, one with high threshold voltage transistors for low power and one with low threshold voltage transistors for low delay. Using low threshold devices only in timing critical paths can radically decrease leakage currents both in standby and operational mode.

Due to transistor stacking in CMOS gates, the input pattern has great impact on the leakage power. The leakage current may differ several orders of magnitude for a simple CMOS gate depending on the input pattern [17]. Reduced leakage power is therefore achievable during idle mode depending on the selected patterns for internal nodes.

In [18], a comparison is made between four different leakage reduction techniques involving supply voltage scaling, increased channel length, stacking transistors, and reverse body bias. Increased channel length is a static design method while stacking transistors, supply voltage scaling and reverse body bias are suitable as adaptive or reconfigurable power management methods. Results in [18] indicate that reverse body bias is an efficient method to reduce leakage.

Supply voltage scaling directly affects leakage, making it a feasible technique for power reductions. In [8], power consumption due to leakage is described as $I_{leak} \cdot V_{dd}$, and in [19], it is shown that the leakage current decreases when the supply voltage is decreased.

Stacking transistors increases the resistance between supply and ground, and thereby reduces both leakage current and maximum transistor current [8]. In Figure 7.9(a), the potential V_m is higher than 0V due to the leakage current. The increased potential at V_m makes the voltage between gate and source negative for transistor T2, which reduces the leakage current. The voltages between bulk and source, and between drain and source also change due to the stacking effect, which also results in less leakage current [8].

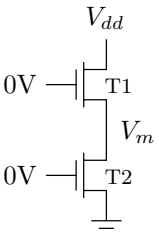
Reverse body bias as well as increased channel length can be utilized to increase the threshold voltage and thereby reduce leakage [8]. Reverse body bias involves increasing the bulk potential for the PMOS transistors and decreasing the bulk potential for the NMOS transistors, see Figure 7.9(b). A technique using reverse body bias for standby power management is described in [20].

7.2.5 Beyond clock gating

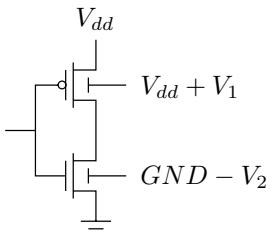
With aggressive gate length scaling follows an excessive gate leakage current that is due to very thin oxy-nitride gate dielectrics. Therefore, the static power dissipation will rapidly increase with the development of the technology and methods for power saving, such as clock gating, will cease to work.

A simple way to save power is to shut down a function block when it is not used. This method is applicable if it can be predicted when a block is going to be used again. This stems from the fact that the time to power up a block is in the order of a few microseconds. This rather long time duration is due to the risk of interference with other parts on a chip. A way to reduce the power-up time and still save power is to just lower the voltage in a block when it is not used. A disadvantage with these methods is that the block loses its information. To retain this information some memory stages can be inserted in the block that stores the state the block was in before it was shut down.

For handheld applications a low power dissipation is very important. Therefore, special low power cell libraries has been developed for new technologies. The difference with these cell libraries is that the oxy-nitride dielectrics are as



(a)



(b)

Figure 7.9: The stacking effect (a). Reverse body bias (b).

thick as in the previous technology. As an example, the 90 nm and the 130 nm technology use oxy-nitride dielectrics with same thickness. The drawback with this is that the performance with respect to speed will also be similar to the previous technology.

Using the methods above to decrease the power dissipation of the block has an increase of 5% to 20% in chip area of a block.

One can also predict that technologies have to be developed with high-κ gate dielectrics to meet stringent gate leakage and performance requirements. Also, the downscaling of technology causes the drain current in a transistor (Gain Induced Drain Leakage) to increase to be in the same order of the gate current. More information about future technology predictions, see [21].

7.3 Power estimation with Synopsys Power Compiler

This section tries to give some practical examples of how to carry out power estimation using Synopsys **PowerCompiler**. First, a format that models different properties of a cell's and a design's behaviour is presented. Information provided by a cell library should be briefly verified by inspecting the respective files in order to be able to manually calculate the power consumption of a cell.

7.3.1 Switching Activity Interchange Format (SAIF)

In order to allow for power estimation and optimization there has to be input in form of toggle rate and static probability data based on switching activity, typically available through simulation. Here, the toggle rate is a measure of how often a net, pin, or port went through either a $0 \rightarrow 1$ or a $1 \rightarrow 0$ transition. The static probability is the probability that a signal is at a certain logic state. With SAIF [22] data, **DC** can perform a more precise power estimation, for example, modelling state and path dependent power consumption of cells. Generally, there are two types of SAIF files: *forward annotation* and *back annotation*. The former is provided as input to the simulator, the latter is generated by the simulator as input to the following power estimation and optimization tool.

For RTL simulation, the forward annotation SAIF file contains directives that determine which design elements are to be traced during simulation. These directives are generated by the synthesis tool from the technology-independent elaborated design. Furthermore, it lists synthesis invariant points (nets, ports) and provides a mapping from RTL identifiers to synthesized gate level identifiers (variables, signals). Following is the command that can be used from the **dc_shell** prompt.

```
dc_shell> rtl2saif -output rtl_fwd.saif -design <design_name>
```

For gate level simulation, the forward annotation SAIF file contains information from the technology library about cells with state and/or path dependent power models. The extra information provided results in more accurate estimates.

```
dc_shell> read -format db <library_name>.db
dc_shell> lib2saif -output <library_name>.saif <library_name>.db
```

Following is an example of a forward annotation SAIF file for gate level simulation that takes state and path dependent information into account. Considered is a 2-input XOR cell from the UMC 0.13μm process.

```
(MODULE "HDEXOR2DL"
   (PORT
      (Z
         (COND !A2 RISE_FALL (IOPATH A1 )
          COND !A1 RISE_FALL (IOPATH A2 )
          COND A2 RISE_FALL (IOPATH A1 )
          COND A1 RISE_FALL (IOPATH A2 ) )
      )
   )
   (LEAKAGE
      (COND (!A2 * !A1)
       COND (A2 * !A1)
       COND (!A2 * A1)
       COND (A2 * A1)
       COND_DEFAULT)
   )
)
```

When a transition at output Z occurs, the simulator should determine which path caused the transition and evaluate the associated state condition at the input transition time. For example, if a transition at Z is caused by $A1$ and the value of $A2$ was "0" at the time, the transition belonged to COND !A2 RISE_FALL (IOPATH A1). In this example, all four possible outcomes can

be assigned different power values in the back annotation SAIF file. Also, the leakage power section is divided into four outcomes for valid cell states and a default outcome for cell states not covered by the aforementioned ones.

The back annotation SAIF file from the simulator includes the captured switching activity based on the information and directives in the appropriate forward annotation file. Switching activity can be divided into two categories, *non-glitching* and *glitching*. The former is the basis of information about toggle rate and static probability. This information usually yields reasonable analysis and optimization results. To model glitching behaviour, a full-timing simulation is required. Transport glitches (TG), for example, are extra transitions at the output of a gate before the output signal settles to a steady state. They consume the same amount of power as a normal toggle transition does and are an ideal candidate for power optimization. Unbalanced logic often give rise to these kind of glitches.

The second category of glitches is called inertial glitches (IG). Unlike TG, their pulsewidth is smaller than a gate delay and would be filtered out if an inertial delay algorithm were applied in the simulator. However, the number of transitions is not enough to accurately estimate their power dissipation. Therefore, the simulation tool provides a de-rating factor (IK) to scale the IG count to an effective count of normal toggle transitions.

In a back annotated SAIF file there are timing and toggle attributes. The former group includes:

- T0: Total time design object is in “0” state
- T1: Total time design object is in “1” state
- TX: Total time design object is in unkown “X” state
- TZ: Total time design object is in floating “Z” state
- TB: Total time design object is in bus contention (multiple drivers simultaneously driving “0” or “1”)

The toggle attributes can be summarized as:

- TC: Number of “0” to “1” and “1” to “0” transitions
- TG: Number of transport glitches
- IG: Number of inertial glitches
- IK: Inertial glitch de-rating factor

Following is an example of how state dependent timing attributes appear in the back annotated file and how they are interpreted.

```
( COND (A*B*Y)          (T1 1) (T0 8)
  COND (!A*B*Y)         (T1 1) (T0 8)
  COND (A*!(B*C))       (T1 2) (T0 7)
  COND B                (T1 1) (T0 8)
  COND C                (T1 1) (T0 8)
  COND_DEFAULT          (T1 3) (T0 6))
```

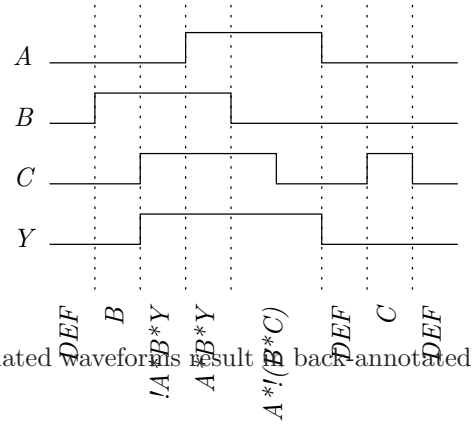


Figure 7.10: Simulated waveforms as result of back-annotated SAIF data as listed before.

The respective simulated waveforms are depicted in Figure 7.10. There are 9 cycles in total and for every condition defined, the number of cycles for which this condition is fulfilled are counted.

Extending this, we list the back annotated SAIF-file that results from a gate level simulation of a simple XOR gate in `xor_back.saif`. Each net is modeled with its timing and toggle attributes. From the simulation time t_{sim}, the toggle rates TR and static probabilities $Pr\{I\}$, are obtained according to

$$TR = TC/t_{sim} \tag{7.10}$$

and

$$Pr\{I\} = T1/t_{sim}. \tag{7.11}$$

This is the basic information needed to calculate switching and leakage power.

7.3.2 Information provided by the cell library

As an example, the standard cell library to UMC's 0.13μm process is considered. The information is bundled in [23]. Included are general definitions of base units, operating conditions, and wire load models. Each cell's behaviour is described in conjunction with timing and power model templates. These templates use weighted input transition times and output net capacitances as indeces into the respective look-up tables that specify the energy per transition. Figure 7.11 shows an example of how the internal energy per transition is obtained using a look-up table.

Considering a simple cell, this look-up is done for every input pin. The path dependent power model distinguishes which state the pin is in and whether a rising or falling transition occured. Furthermore, as indicated in the figure, an extrapolation takes place if the exact index into the table is not available.

7.3.3 Calculating the power consumption

As a simple example, we consider a 2-input XOR cell. The information needed to calculate the power consumption is in a file called `HDEXOR2DL.rpt`. Included is the necessary wire load model and the cell description. In order to calculate

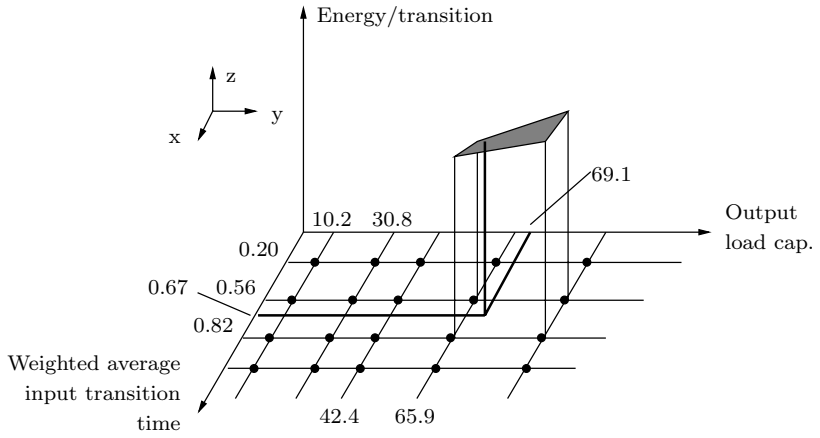


Figure 7.11: Look-up table based internal energy calculation.

toggle rates and static probabilities of the nets, use the back annotated SAIF file listed earlier in this section.

The total leakage power of a cell is determined by a superposition of the probabilities of being in a certain state times the leakage power for this state. Here you have to make use of the state dependent leakage power information that is provided in the cell library.

$$P_{leak} = \sum_{i=1}^{N} Pr\{I = i\} \cdot P_{leak,i}, \tag{7.12}$$

where $N = 2^m$ is the number of combinations of m binary inputs and $I = [1 \dots 2^m]$.

The switching power of a cell can be expressed as

$$P_{switch} = 1/2 \cdot \frac{TR}{\text{t.u.}} \cdot (C_{wire} + C_{pin}) \cdot V_{dd}^2. \tag{7.13}$$

Here, one has to take the load capacitance into account which is usually comprised of wire load and pin load. The factor 1/2 results from the fact that TR accounts for both $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions, but only half of them affect the power consumption. Also, the number of transitions is counted on a per second basis and hence one has to normalize TR to the default time unit (t.u.) of the cell library, usually 1ns.

Based on these facts try to calculate both switching and leakage power of the cell. The internal power is a lot more subtle to calculate since the tool's extrapolation algorithm is hard to track. For verification of your results, the power estimation report is found in `xor.rpt`. Surely, one gets a feeling for the amount of information that has to be processed to get an estimate of a design's power consumption.

7.4 Power-aware design flow at the gate level

After having introduced the basic concepts of power estimation, it is time to conclude with a complete design flow. Figure 7.12 shows an example of a design

flow that estimates and optimizes power consumption of a digital design at the gate level.

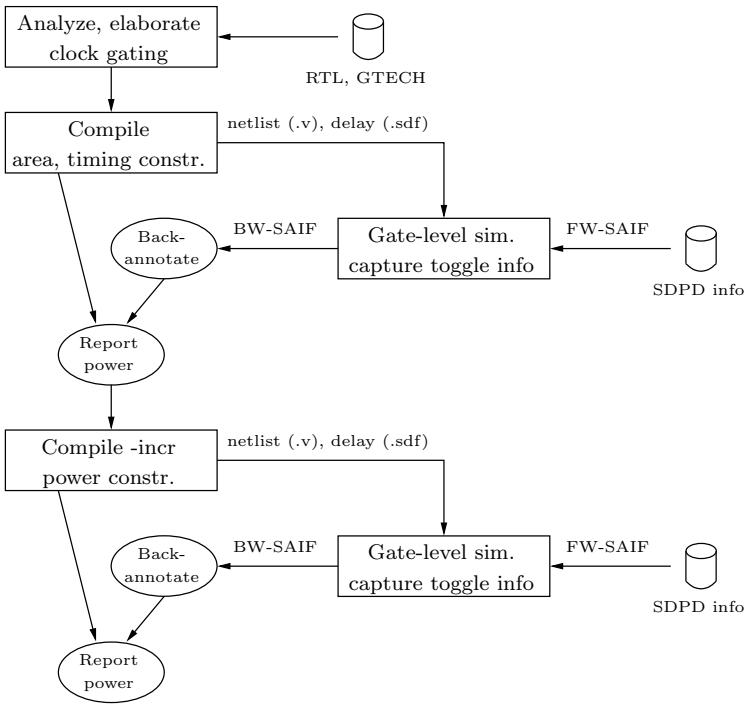


Figure 7.12: Gate-level power estimation and optimization flow.

After RTL clock gating is completed, timing and area constraints are set to the design. In most cases clock gates have to be inserted manually in the design together with control logic but in some cases there are tools that can perform automatic low level clock gating. These tools automatically find flip-flops that have an enable control and exchange these to a clock gate and standard flip-flops. Figure 7.13(a) shows a flip-flop with enable input. These flip-flops are implemented with a feedback from output to input and a multiplexer that selects if the previous output is to be kept or a new input is to be accepted, see Figure 7.13(b). Figure 7.14 shows the result of automatic clock gating on a bank of enable flip-flops. Unlike manual clock gating, no extra control signals are needed since all control logic is already there.

Compiling the elaborated RTL or GTECH description results in a gate level netlist on which you can annotate switching activity from a back annotation file (BW-SAIF). This file is created by a gate-level simulator, preferably **nc-verilog** or **vcs** since these simulators make use of state and path dependent information provided by the cell library (FW-SAIF). Note that **MS** can not make use of this information and hence the power estimates based on the toggle files from this simulator will be inferior. Now, the switching activity is annotated and power constraints are set to the design. An incremental compile will then try to optimize the netlist based on the premises it was provided with. If the design was fully annotated, Synopsys claims that the power estimates lie within 10-25% of **SPICE** accuracy.

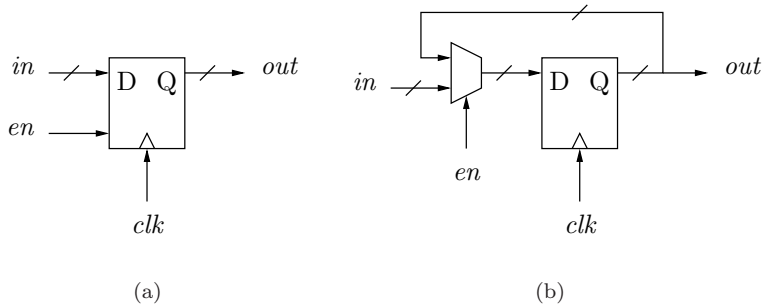


Figure 7.13: An enable flip-flop (a) and its implementation in Synopsys (b).

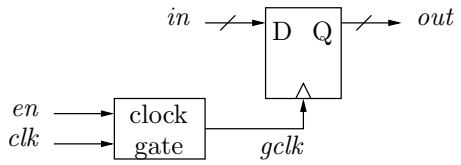


Figure 7.14: The result of automatic clock gating on a bank of enable flip-flops.

Optimization takes place by evaluating optimization moves and their impact on the overall cost function. There is a predefined optimization priority that can be changed. By default, design rule constraints set by the technology library, for example, max fanout, max transition, and max capacitance, are the most important ones. Then, timing is the next critical issue. Following are in descending order dynamic power, leakage power, and area constraints.

Generally, a positive timing slack will leave more possibilities for optimization whereas already tight timing constraints might not result in any power improvement at all. Also, diverse cell libraries make life easier for the optimization tool. Furthermore, designs dominated by sequential logic often achieve less power reduction since different sequential cells tend to have less variation in power than combinational cells.

7.5 Scripts

This section provides a short description on scripts for **DC** and Cadence **ncverilog** that deal with power estimation and optimization. These scripts can be downloaded from [**?**]. A general knowledge about UNIX and **DC** is assumed. The scripts are written for the UMC 0.13 μm technology. No responsibility is taken for the consequences of using these scripts, although great care has been taken to debug the scripts. All feedback is welcome.

7.5.1 Design flow

Figure 7.15 shows the design flow for the power optimization scripts, the names of each script and environments they are executed in. Inputs are `header.scr`

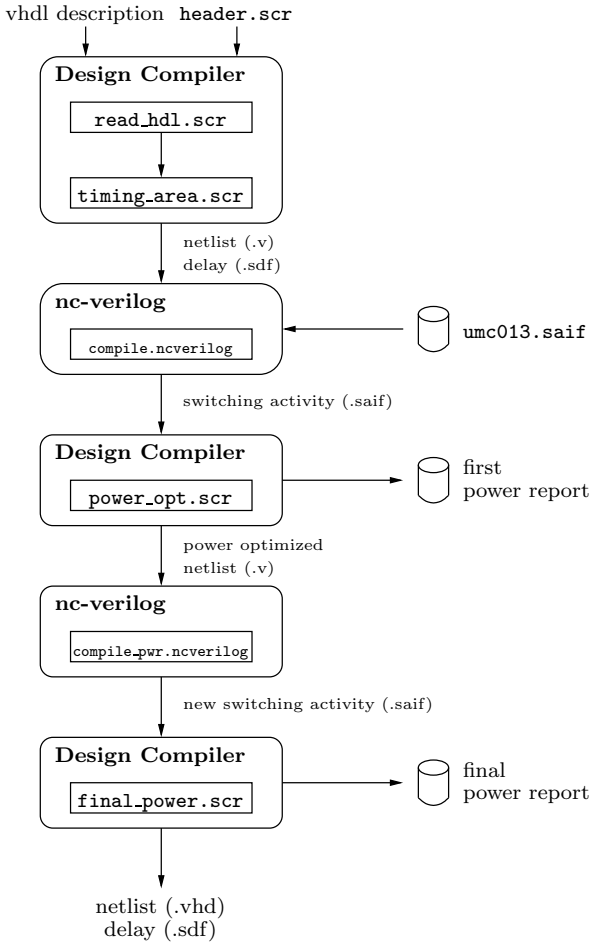


Figure 7.15: Design flow for power optimization with **DC** and **nc-verilog**.

and VHDL code. All information about the design and the target values for the synthesis is listed in `header.scr`.

The first step in the optimization takes place in **DC**. Here the design is be read, elaborated, and synthesized. Timing and area constraints are set in this stage and **DC** will look for places to insert clock gates. The result is an initial VERILOG netlist together with a standard delay format (SDF) file that will be used in **nc-verilog** to estimate switching activity in each node of the design. This is carried out by running

```
ncverilog -f compile.ncverilog
```

Then, the switching activity report is used to estimate the power consumption in **DC** (`power_opt.scr`). All important results are collected in a report file. With this results as a guideline, power constraints on both dynamic and leakage power can be set in the `header.scr`. An incremental synthesis is performed and the result, a power optimized netlist, is used in **nc-verilog** to collect the new switching activity by running

```
ncverilog -f compile_pwr.ncverilog
```

The remaining step is to estimate the final power consumption and collect all important information in a report file. This is performed in **DC** with `final_pwr.scr`. The final output are a VHDL netlist, an SDF file, and a report file with the final power consumption estimate. To create an SDF file for VHDL the `write.sdf` script has to be executed in **PrimeTime** due to some naming conventions problems between the UMC libraries and the DC SDF file. Then, this netlist and the SDF file can be used to perform a post synthesis simulation for verification purposes in **MS**.

7.5.2 Patches

The aim was to generate a fully automatic design flow that could be executed with one script and where all parameters are set in one constraint file. However, since more than one program is involved there are some parts of the scripts that have to be patched manually, which files and what to change are listed below. In addition to run an automated flow a file hierarchy as shown in Figure 7.16 must be provided. The complete structure with all script files and an example can be downloaded from [?].

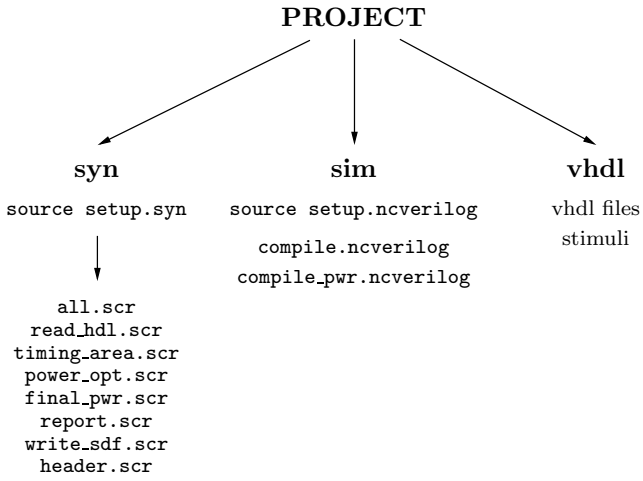


Figure 7.16: The required file hierarchy for an automated design flow.

To initialize the project, create the hierarchy shown in Figure 7.16. Then `source setup.syn` in the **syn** folder and `source setup.ncverilog` in the **sim** folder. Then, open the `header.scr` and fill in search paths, constraints, and design names. Finally, the parts that have to be manually patched are shown in Table 7.1.

If the above procedure is performed correctly, the `all.scr` script executed in the **syn** folder will perform all steps in the design flow and the result can be read in the report files. However, it is strongly recommended that the scripts are executed one at a time and the observed results are correct, at least once before the `all.scr` script is executed.

Table 7.1: Manual patches.

File	Parameter	Values and example
write.sdf	design data base	db/ALU_W_L8_PWR.db
write.sdf	result file name	netlist/ALU_W_L8_PWR.sdf
verilog tb	file name	must be tb_"MODULE".v, e.g., tb_ALU.v
verilog tb	Design under test	must be labeled as dut
verilog tb	inputs and parameters	see the example, tb_ALU.v
compile.ncverilog	tb name	tb_"MODULE".v, e.g., tb_ALU.v
compile.ncverilog	netlist name	../syn/netlists/ALU_W_L8.v
compile_pwr.ncverilog	tb name	tb_"MODULE".v, e.g., tb_ALU.v
compile_pwr.ncverilog	netlist name	../syn/netlists/ALU_W_L8_PWR.v

7.6 Commands in DC

Some commands that are frequently used in **DC** and **nc-verilog** for power optimization are briefly explained in this section. It is assumed that the reader already has basic knowledge about common commands in **DC**. For a complete explanation use the `man` command in **dc_shell**, for example,

```
dc_shell> man <command_name>
```

To get a glimpse of how many commands there are, feel free to type:

```
dc_shell> list -commands
```

For a complete explanation of the commands in **nc-verilog**, type `sold` and look in the Power Management/Power Compiler reference manual/Using Verilog.

7.6.1 Some useful DC commands

This section is an excerpt of the commands that appear in the power estimation scripts.

Clock gating

`set_clock_gating_style -sequential_cell latch -minimum_bitwidth 3`
Description: Tells **DC** to use latch based clock gates if it finds enable registers that are at least `-minimum_bitwidth` bits wide when elaborating.
Warning: Use clock gates with latch to avoid glitch sensitivity. Might cause timing errors in **DC**, see also `set_false_path`.

`elaborate ALU -lib WORK -update -gate_clock`
Description: Elaborates the design and looks for places to insert clock gates.

`set_false_path -to find(-flat, -hierarchy, pin "latch/D")`
Description: Tells **DC** to NOT perform timing checks on the enable signal in clock gates. **DC** might otherwise insert one clock cycle of delays on the enable signal.
Warning: Will only function correct if the latches in the clock gates are the only latches in the design (this is normally the case).

`report_clock_gating -hier -gating_elements`
Description: Reports all inserted clock gates.

Power estimation and optimization

`link`
Description: Links all used libraries to the current design.

`power_preserve_rtl_hier_names=true`
Description: Preserves RTL-hierarchy in RTL design.
Warning: Must be set to true if `rtl2saif` is to be used.

`rtl2saif`
Description: Used to create a forward annotated SAIF file. This file is then used in a simulation tool to capture switching activity.
Warning: Only necessary if power estimation is performed on RTL. The forward SAIF file for gate-level simulation is provided by the technology library (`lib2saif`).

```
set_max_dynamic_power 8 mW
set_max_leakage_power 20 uW
```
Description: Sets target values for power optimization.

`compile -incremental`
Description: Will only try to solve issues due to new constraints set on an already compiled design. Used to recompile the design after power constraints are specified.
Warning: Must be used if a second compile is performed on the same design.

`report_power -analysis medium -hier -hier_level 3 >> REPORT_FILE`
Description: Reports power consumption in the design and lists it for each module down to the third hierarchy level.

```
reset_switching_activity -all
read_saif -input "backward.saif" -instance "tb_ALU/dut" -unit_base ns
```
Description: Removes previously annotated switching activity and reads new switching information from the **backward.saif** file into the design. This information is used to estimate power consumption. See also the **nc-verilog** command toggle report.

Miscellaneous commands

`compile -boundary_optimization`
Description: Will optimize across all hierarchy levels.

```
report_area > REPORT_FILE
report_timing >> REPORT_FILE
report_constraint >> REPORT_FILE
report_design >> REPORT_FILE
report_reference >> REPORT_FILE
report_routability >> REPORT_FILE
report_port >> REPORT_FILE
```

```
report_clock_gating -hier -gating_elements >> REPORT_FILE
report_saif -flat -missing >> REPORT_FILE
report_power -analysis medium -hier -hier_level 3 >> REPORT_FILE
```
Description: Reports a lot of useful information and store it in `REPORT_FILE`.

```
change_names -rules vhdl -hierarchy
write -format vhdl -hierarchy -output "netlist/ALU_syntesized.vhd"
```
Description: Change names to match with current db and save the netlist as in vhdl-format. `change_names` is used since **DC** sometimes change names on nets and ports.
Warning: Will not fix all names for vhdl-format. Therefore, do not be surprised about all the warnings.

```
write_sdf -version 1.0 "ALU.sdf"
```
Description: Writes information about timing and delay between all pins in the design. This file is used to perform a more accurate simulation of the netlist.

```
set_register_type -exact -flip_flop HDSDFPQ1
```
Description: Replaces all registers with the specific register HDSFPQ1.

```
design_list = find (design, "*", -hierarchy)
design_list = design_list + current_design
foreach (de, design_list) {
   current_design de
   cell_list = find (cell,"*plus*")
   set_implementation "DW01_add/cla" cell_list
   remove_unconnected_ports cell_list
}
```
Description: Finds all adders in the elaborated design and set them to be implemented in carry-lookahead (CLA) style found in the **DesignWare** library DW01.

7.6.2 nc-verilog

```
read_lib_saif("umce13h210t3_tc_120V_25C.saif");
```
Description: Will read the timing information specific to the technology library that is used.

```
sdf_annotate("ALU.sdf", dut, ,"./sdf.log");
```
Description: Will read the timing information specific to the design **dut** that is to be simulated.

```
set_toggle_region(tb_ALU.dut);
```
Description: Specifies the top level in the design, in this case the **dut** module.

```
toggle_start();
```
Description: Specifies at what time toggle count will start, for example,

```
#50*CYCLE_TIME $toggle_start()
```

states that toggle count will begin after 50 clock cycles (CYCLE_TIME is a constant specified in the testbench).

`toggle_stop();`
Description: Specifies at what time toggle count will stop, for example,

```
#1000*CYCLE_TIME $toggle_stop()
```

states that toggle count will begin after 1000 clock cycles (CYCLE_TIME is a constant specified in the testbench).

`toggle_report("/backward.saif",1.0e-9,"tb_ALU");`
Description: Writes all information about switching activity for each pin and net in the design to the file backward.saif, the time scale is nano seconds. This file is used in **DC** to perform a more accurate power consumption estimate.

7.7 The ALU example

The power optimization scripts described in the previous section includes an example design, an ALU. Figure 7.17 shows the ALU schematics where A and B are inputs, Q is output, and ALU_CTRL are the control signals, see Chapter **??** for more details. What follows is a short description of the effects of running the scripts on the ALU design.

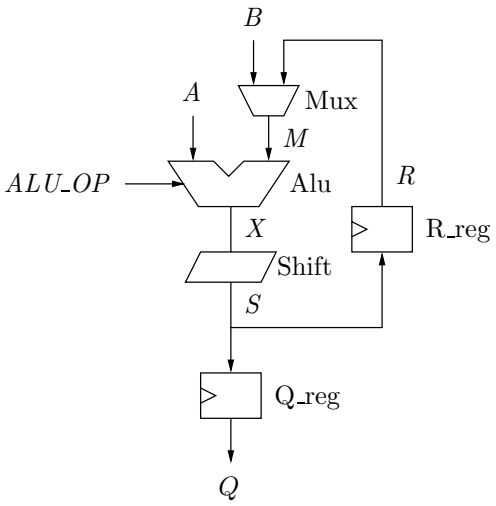


Figure 7.17: An Arithmetic Logical Unit (ALU).

The two enable flip-flops will be identified as candidates for clock gating and exchanged with a clock gate and basic flip-flops without enable input. The ALU will then be synthesized according to timing constraints, that is, the required clock frequency, and an estimate of the power consumption is performed based on the inputs in the stimuli file. Then, power constraints are specified and an incremental synthesis is performed. Finally, the new design is power estimated and the result is presented in a report file, the intermediate results from synthesis are also available in a different report file for comparison purposes.

Chapter 8

Formal Verification

With the growing complexity and reduced time-to-market the biggest bottleneck in digital hardware design is design verification. An insufficiently verified ASIC design may contain hardware bugs that can cause expensive project delays when they are discovered during software or system tests on the real hardware. The growing complexity of ASIC designs has caused the time of verification through simulation to increasing very much. Of this reason there is a huge demand to find verification methods that are faster without reducing the coverage of the verification.

One method that has matured and established it self in real life design flows is formal verification methods. The formal verification methods can be described as a group of mathematical methods used to prove or disprove properties of a design. It has been shown that for a design of 20 to 25 million gates the verification time was reduced to less than a tenth of the verification time using a simulation tool. In addition to being much faster than simulation, formal verification techniques are also inherently more robust. It is therefore not necessary for the designer to identify all corner cases and test each one with a set of test vectors. A formal verification tool will automatically find all cases where a design does not meet the specifications. This means that it will find the difficult corner case even if the designer has missed it.

An example that will make the difference between verification using simulation and formal verification visible is the verification of a 32 bit shifter. To fully verify the shifter block, 256 billion test vectors are needed where as with formal verification methods only a mathematical equivalence verification is needed.

8.1 Concepts within Formal Verification

In Figure 8.1 the most important concepts in formal verification, and how they relate to another is described.

Requirements and needs are more or less precisely formulated requirements and expectations of a system. By requirements formalization, one arrives at a formal specification, which gives precis requirements. The formal specification is validated to make sure that it correctly captures the requirements and needs.

The real system is the actual design. The system is modelled to obtain a mathematical description of how it works. The modelled description of the real

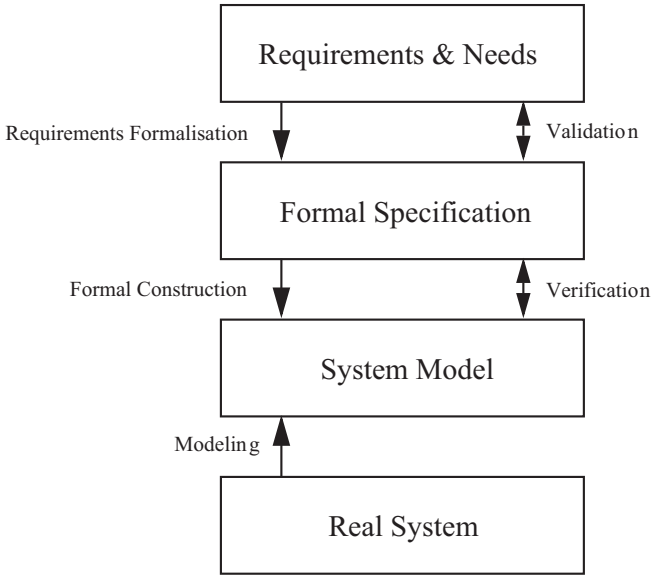


Figure 8.1: The concepts in formal verification.

system is called the system model.

By formal verification one can prove or disprove that the system model satisfies the requirements in the formal specification. Another possibility is to use formal construction to create a system model which with certainty satisfies the requirements.

8.1.1 Formal Specification

The formal specification gives an exact description of the system function. The function of the formal specification is therefore to precisely describe the concepts used when talking about the functionality of the system. It is also essential to make certain that the formal specification correctly describes the functionality of the system. This is done by validating the specification.

When a formal specification is generated by a tool it is important that the formal specification correctly describes the behavior of the design. Since this is an issue of the correctness of the tool used it is hard to have any control over it. To increase the certainty, some designers uses two formal verification tools from different vendors with the hope that the tools complement each other when it comes to correctness.

8.1.2 The System Model

Since the formal specification is a mathematical description of what the system should do, a mathematical description of what the system actually does is needed. This description is called the system model.

Also when the system model is generated by a tool the correctness is an issue and therefore the same procedure as for the tool generated formal specification is used.

8.1.3 Formal Verification

Formal verification means that using a mathematical proof it can be checked with certainty whether the system (as described by the system model) satisfies the requirements (as expressed by the formal specification).

Formal verification differs from testing in that it guarantees that the system acts correctly in every case. When relying on testing there is no guarantee that the system does not behave incorrectly in a case which has not been tested. A precondition for meaningful verification is, of course, that the system model and the formal specifications correctly capture the properties of the system and the requirements, respectively.

A successful formal verification does not imply that testing is superfluous since every step in the system development can not be treated formally. What the formal verification will substantially reduced is the amount of testing, and above all reduce the amount of debugging.

8.1.4 Formal System Construction

The feasibility of successful verification depends to a major extent on how the system is constructed. The principle of designing for testability is most certainly valid when formal methods are to be used. It is often difficult to verify an existing hardware system already designed and constructed using traditional methods.

Instead of developing the system with traditional methods and then verifying it, the system can be developed using a formal construction process (often referred to as synthesis of the system). In general, formal design and construction is therefore preferable to applying formal verification on top of a traditional system development process.

8.2 Areas of Use for Formal Verification Tools

There are two main kinds of usage for formal verification tools namely, model checking and equivalence checking, which is used in different stages in the design flow.

8.2.1 Model Checking

When the formal verification tool is used as a model checker design specifications are translated into a set of properties described in a formal language. The design is then mathematically verified for consistency with these properties. This kind of verification is often used to verify RTL code before synthesis, where the designer does not have time to wait for lengthy simulation runs during each design iteration.

8.2.2 Equivalence Checking

When the formal verification tool is used as an equivalence checker the tool compares two design descriptions and verifies that they have the same behavior. In this case the formal specification and the system model is automatically generated by the tool. This kind of verification is often used to compare two

versions of RTL code to ensure that an optimization has not changed behavior of the design, or it can compare two netlists to ensure that the insertion of a scan chain or other test logic has not ruined the original behavior of the design.

8.2.3 Limitations in Formal Verification Tools

With formal verification methods only functionality checks of a design can be done. To cover delay problems, timing problems and sequencing problems in a designspecial verificatin tools has to be used.

In formal verification tools as in synthesize tools the same class of algorithms is used. To reduce the risk of double error it is therefore important to use independent tools. Often secured by using tools from different vendors.

In an asynchronous circuit designs formal verification methods are not usable since formal verification methods can not handle non sequential circuit designs.

8.3 Other Methods to Decrease Simulation Time

8.3.1 Running the simulation in a cluster of several computers

The simulation is divided up on several computers in a network. To be able to do this the simulation tool used must have this possibility.

8.3.2 ASIC-emulators

The ASIC-emulators is mainly used to start software development earlier but can also be used as a hardware simulation tool. A feature in ASIC-emulators that makes hardware simulation easier is that the emulator can be compiled in such a way so that nodes in the design is visible to an desirable extent.

8.3.3 FPGA-Prototypes

As for the ASIC-emulator the FPGA-prototypes is mainly used for software development but can also be used for hardware simulation. The FPGA-prototypes has some limitation when it comes to imitate designs on chip. The penalty of these limitations for FPGA-prototypes is mainly that one has to adjust the design methodology to these limitations. Example of limitations is that FPGA's are developed for a certain type of applications which sets the architecture of the FPGA. Therefore when implementing an ASIC-design on an FPGA it is fairly often that the ASIC-architecture not agrees with the architecture of the FPGA. Limitations in development tools for FPGA's can also give some problems since they are not so advanced as development tools for ASIC's. Despite the limitations with FPGA's, it is one of the most preferred methods when simulating a hardware design.

8.4 Future

Reliability of formal verification tool has increased very much since the tools become commonly used for ASIC-verification. The reliability will properly in-

crease even more in the future and reduce the limitations.

A process in constant progress is to move hardware design to a higher abstraction level. The need of this comes from the desire to reduce time-to-market. Formal methods has shown that they apply very well on the next level of abstraction that is the system level. Hence, it can be expected that in new tools formal methods will be more common.

8.5 References

http://www.l4i.se/E_form.htm
http://www.edtnscandinavia.com/story/OEG20010403S0006
http://www.hardi.com/eda-tools/literature/Formell_verifiering_i_praktiken_1.pdf
http://www.hardi.se/eda-tools/why_formal_verification.htm
http://www.mentor.com/formalpro/css.html, click on Alcatel, Emmanuel Ligeon.

Chapter 9

Multiple Supply Voltages

Energy and power consumption are important factors in the design of integrated hardware. To make the battery in a hand-held device, such as a PDA or a cellular phone to last as long as possible, energy stored in the battery must be used efficiently. Furthermore, limiting the on-chip power consumption is important due to increased production costs for cooling arrangements.

For the past decades, the technology development has followed the predictions by Gordon Moore in "Moore's law". The original version of "Moore's law" was an observation that the number of components on integrated circuits doubles every 12 months [24]. Later, "Moore's law" has been revised to a factor two improvement every 24 months, and a prediction of doubled clock frequency every 24 months has also been added [25,26]. If the increased computing speed and packing density of modern technologies is to result in faster and more complex chips, new methods must be developed to handle power consumption. Power management has therefore gained much attention both in industry and in universities.

Active power consumption due to capacitive switching is traditionally regarded as the main source of power consumption in digital design. However, reaching deep submicron technologies also makes it necessary to take transistor leakage into account, since it causes an increasing power consumption in each technology generation [27]. Figure 9.1 shows the development in active and leakage power [27]. If the trend shown in Figure 9.1 continues, the active power consumption will soon be overtaken by the leakage power.

While the active power consumption varies with the workload, and can be locally eliminated using for instance clock gating, power consumption due to leakage remains, as long as the power supply voltage is switched on. For burst mode applications, such as mobile phones, a large part of the system is in idle mode most of the time. This makes the energy consumed by leakage a large contributor to the overall energy consumption and, thus, making simple saving schemes such as clock gating less attractive.

Reducing the power supply voltage is an efficient method to reduce all types of power consumption (due to switching, short circuit, and leakage) [28]. However, reducing the power supply voltage for an entire chip might not be possible due to performance constraints. Components of an ASIC or a microprocessor have different requirements on power supply voltage due to different critical paths. Therefore, running some components on a lower voltage can save power.

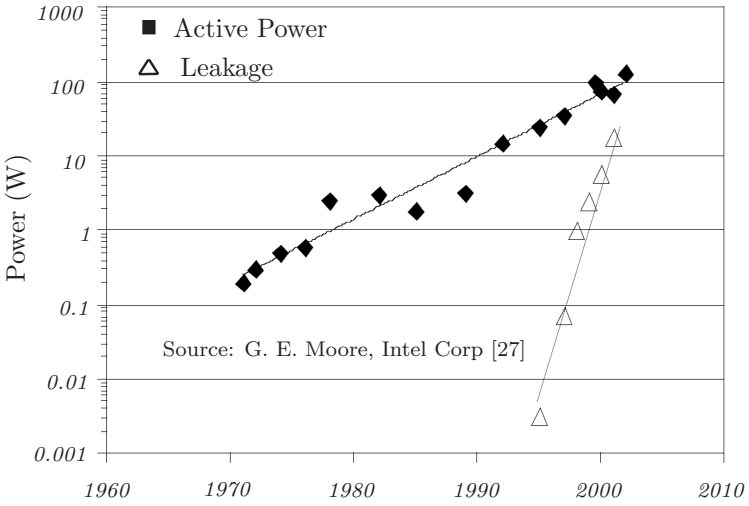


Figure 9.1: Processor power, active and leakage.

Increasing the number of available supply voltages from one to two often results in large power savings, while using more than two voltages only leads to minor additional savings. In [29], and [30], power savings in the range of 25-50 % are achieved for a number of design examples using dual supply voltages compared to a single voltage approach. For the designs in [29], an increase in the number of supply voltages from two to three results in less than 5 % additional power savings.

Supply voltage scaling is associated with an overhead due to DC/DC-converters for transforming the supply voltage down to lower voltage levels. The most frequently used DC/DC-converter for power savings by lowering the supply voltage is the Buck converter [28]. The Buck converter converts the main supply voltage down to a lower voltage using large transistor switches, as shown in Figure 9.2a. In the Buck converter, a control circuit is used for generating a square-wave with variable duty factor that controls the switches. The switched supply is filtered using a low-pass filter to minimize the ripple at V_{out}. Besides the power consumption in the control circuits for the Buck converter, there is always a power loss in the switches due to resistance of the transistors. In [31] and [32], there are examples of Buck converters reaching an efficiency as high as 90 %.

In a system with more than one supply voltage, signal level converters are needed to ensure correct functionality when going from a low voltage region to a higher voltage region. The signal level converter shown in Figure 9.2b is used in [30, 33, 34]. Furthermore, if each output bit is connected to an inverter supplied by a lower voltage, all buses and signals are run at the lower voltage. This leads to further power savings [35].

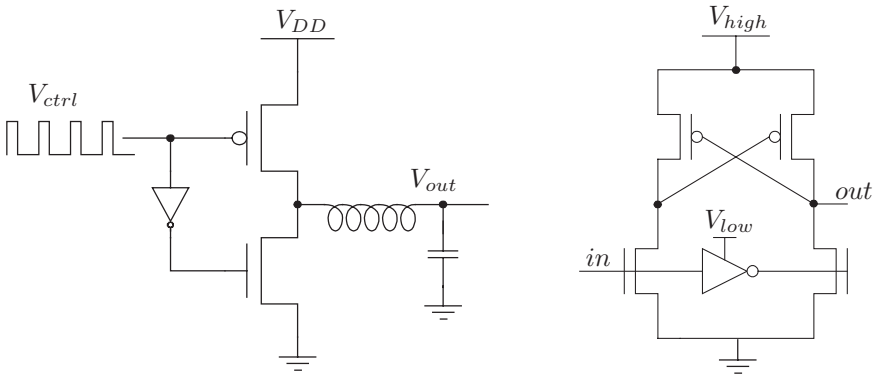


Figure 9.2: Overhead components associated with multiple supply voltages. Buck converter (a), and signal level converter (b).

9.1 Demonstration of Dual Supply Voltage in Silicon Ensemble

This part describes a (fairly) simple method to Place & Route a design with multiple supply voltages using the tool Silicon Ensemble. Advantages of this method are that no other tool than Silicon Ensemble is necessary, and that the designer has full control over the design procedure. The text is intended for anyone having knowledge on how to use scripts in Silicon Ensemble. The example is implemented using the UMC 0.13μm technology. A description of the design used in this demonstration is found in [Matthias paper]. Scripts and netlist files for the example are found in [**?**].

9.1.1 Editing Power Connections in the Netlist

The netlist contains information about which net to use for power and ground. Changing an automatically generated netlist is generally avoided. However, assignments of power nets are easily manipulated in a netlist written in Design Exchange Format (DEF).

It is likely that the design is initially described using a Verilog netlist (.v). To simplify edits in the netlist, it is converted to DEF format. The hierarchy of the design shall contain a top view where each block is intended to use a separate supply voltage. In this example, the top view contains the two blocks "conventional" and "improved".

Start Silicon Ensemble as described in ...Joachims manual.... To import supply pads use the DEF-file "supplyPads2Vdd.def", which contains an extra supply voltage pad with net name "VDP". The contents of the file "supplyPads2Vdd.def" is shown below. The added supply voltage pad is "pi_VDP", and it is connected to the net "VDP". The macro **"Verilog2DEF.mac"** can be executed in silicon ensemble to import and convert the design "dummy_top.v" to "dummy_dop.def".

```
DESIGN demoDesign ;

UNITS DISTANCE MICRONS 1000 ;

COMPONENTS 9 ;
- pi_VDDC WVVDD ;
- pi_VDP  WVVDD ;
- pi_VSSC WVVSS ;
- pi_VDDI WVV3IO ;
- pi_VSSI WVV0IO ;
- pi_CORNER_LT WCORNER ;
- pi_CORNER_RT WCORNER ;
- pi_CORNER_LB WCORNER ;
- pi_CORNER_RB WCORNER ;
END COMPONENTS

SPECIALNETS 5 ;
- V3IO ( pi_* V3IO ) + USE POWER ;
- V0IO ( pi_* V0IO ) + USE GROUND ;
- VDD ( pi_VDDC VDD ) + USE POWER ;
- VDP ( pi_VDP VDD ) + USE POWER ;
- VSS ( * VSS ) + USE GROUND ;
END SPECIALNETS

END DESIGN
```

Open the netlist “dummy_dop.def” in a text editor and skip down to the line “PINS 28 ;”. Change number of pins to 29 and add a supply voltage pin “VDP” as shown below.

```
PINS 29 ;
- V0IO + NET V0IO + DIRECTION INPUT + USE GROUND ;
- VSS + NET VSS + DIRECTION INPUT + USE GROUND ;
- V3IO + NET V3IO + DIRECTION INPUT + USE POWER ;
- VDD + NET VDD + DIRECTION INPUT + USE POWER ;
- VDP + NET VDP + DIRECTION INPUT + USE POWER ;
```

Skip down to the line “SPECIALNETS 5 ;”. The text between “SPECIALNETS” and “END SPECIALNETS” contains connections for power supply wires. As default, both blocks use power supply “VDD”. This is changed to “VDP” for the block named “conventional”.

Skip down to the first line that contains “conventional” and change the block to use “VDP” as shown below.

```
  ( improved/l_trellis_instance/l_butterfly_instance_0/U70 VDD )
  ( improved/l_trellis_instance/l_butterfly_instance_0/U69 VDD )
```

```
  ( pi_VDDC VDD ) + USE POWER ;
- VDP ( conventional/U16 VDD ) ( conventional/bm_sig_reg_0_0_3 VDD )
  ( conventional/bm_sig_reg_0_0_2 VDD ) ( conventional/bm_sig_reg_0_0_1 VDD )
  ( conventional/bm_sig_reg_0_0_0 VDD ) ( conventional/bm_sig_reg_0_1_3 VDD )
```

The line

```
  ( pi_VDDC VDD ) + USE POWER ;
```

sets the block "improved" to use the pad "pi_VDDC".

Skip down to the last assignment of VDD-pins for the block named "conventional" and change the block to use the pad "pi_VDP" as shown below.

```
  ( conventional/l_trellis_instance/l_butterfly_instance_0/U67 VDD )
  ( conventional/l_trellis_instance/l_butterfly_instance_0/U66 VDD )
  ( pi_VDP VDD ) + USE POWER ;
- VOIO ( U75 VOIO ) ( U74 VOIO ) ( U73 VOIO ) ( U72 VOIO ) ( U71 VOIO )
  ( U70 VOIO ) ( U69 VOIO ) ( U68 VOIO ) ( U67 VOIO ) ( U66 VOIO ) ( U65 VOIO )
  ( U64 VOIO ) ( U63 VOIO ) ( U62 VOIO ) ( U61 VOIO ) ( U60 VOIO ) ( U59 VOIO )
```

The line

```
- VDP ( pi_VDP VDD ) + USE POWER ;
```

which is found just before the line

```
  END SPECIALNETS
```

should be removed to avoid warnings.

The file "dummy.def" contains all the edits described in this section.

9.1.2 Floorplan and Placement

Import the design with **"1-import.mac"**. This script imports the netlist "dummy_top.def". Change this to "dummy.def" if you skipped the edits described in chapter 9.1.1.

Floorplan and I/O placement is performed with the script **"2-floorp_and_IO.mac"**. The I/O placement is set in the file "ioplace.ioc", which is the default placement except for the power pads.

Since the two parts of the design shall use separate supply voltages, they are placed as two separate blocks. This is controlled by adding "regions" and selecting one region for each block. Regions are added by running the script **"3-regions.mac"**, which is shown below.

```
ADD REGION NAME USEVDP CELL conventional* BBOX (-100000 -45000) (-50000 46000);
ADD REGION NAME USEVDD CELL improved* BBOX (50000 -45000) (100000 25000);
SET VAR DRAW.REGION.AT "On";
REFRESH
```

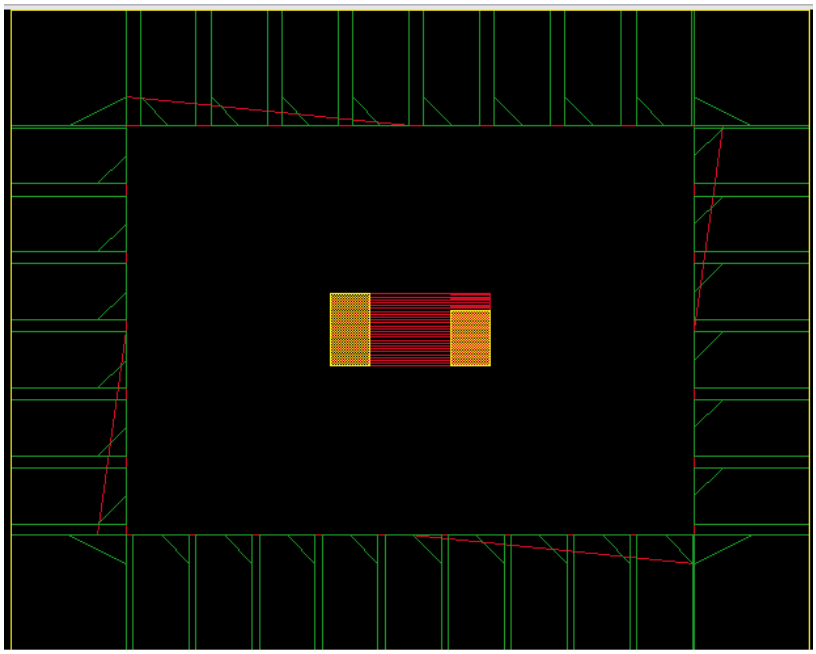


Figure 9.3: Floorplan with regions.

The command

```
ADD REGION NAME USEVDP CELL conventional* BBOX (-100000 -45000) (-50000 46000);
```

creates the region "USEVDP" at the specified coordinates, and sets the placement of all components at hierarchical level "conventional" and below to use the region "USEVDP". Deciding size and placement of the regions is an iterative process.

Figure 9.3 shows the floorplan with created regions.

Place cells by running the script **"4-placecells.mac"**. The cells are placed within the defined regions.

9.1.3 Power and Signal Routing

In this design, all components belong to one of the blocks that are placed within the two regions. Rows outside the regions can therefore be removed to give space for power routing. Running the script **"5-cutrows.mac"** (shown below) deletes rows outside the regions.

```
DELETE ROW AREA SPLIT (-50000,-50000) (50000,50000);
DELETE ROW AREA SPLIT (50000,25000) (100000,50000);
```

Initial power rings are created and then edited for this design. Run the script **"6-initialpower.mac"** to create three power rings (VDD, VDP, and VSS).

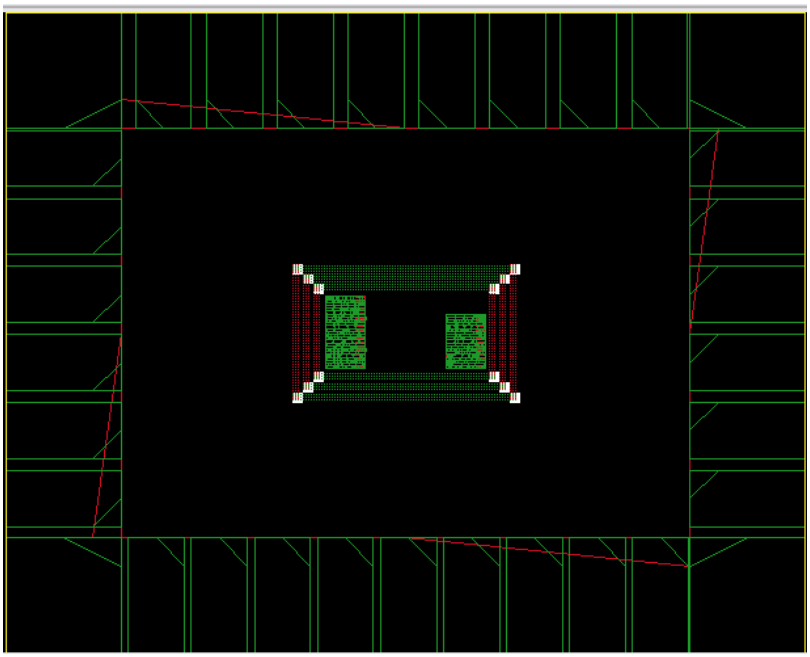


Figure 9.4: Initial power rings.

Figure 9.4 shows the initial power rings.

The script **"7-modifypower.mac"**, shown below contains two types of commands; First, two wires are added for the net "VSS", and then wires for the nets "VDP", and "VDD" are moved to optimize power routing. Adding and moving wires is easily done using the menu commands (Edit-Wire-Move/Add), and then saved as a script file.

```
ADD WIRE SPECIAL DRC NOVIASATCROSSOVER SHORTSCHECK NET VSS
LAYER MET2 WIDTH 10000 YFIRST (-41000, 55200) (-41000, -54800);
ADD WIRE SPECIAL DRC NOVIASATCROSSOVER SHORTSCHECK NET VSS
LAYER MET2 WIDTH 10000 YFIRST (40000, 55200) (40000, -54800);
MOVE WIRE NOVIASATCROSSOVER DRC SNAP NET VDP (120613,23800) (-28000, -22400);
MOVE WIRE NOVIASATCROSSOVER DRC SNAP NET VDD (-135842 25901) (27000, -14800);
MOVE WIRE NOVIASATCROSSOVER DRC SNAP NET VDD (62969,-80009) (62160, -68000);
MOVE WIRE NOVIASATCROSSOVER DRC SNAP NET VDD (62969,79721) (63120, 34800);
MOVE WIRE NOVIASATCROSSOVER DRC SNAP NET VDD (134658,16312) (124000,16312);
```

Power is routed by running the script **"8-routepower.mac"**, and finally, signals are routed by running the script **"9-wroute.mac"**.

There might be geometry violations on signal wires close to pads. This is due to a grid mismatch and can (at your own risk) be ignored.

9.1.4 Filler Cells

To fill out empty spaces between I/O-pads, run the script **"10-fillperi.mac"**.

Filler cells are also needed in empty spaces between standard cells. The two blocks must be handled separately since they have different names on power nets. The script **"11-fillcore.mac"** shown below adds filler cells to both blocks. For each command in **"11-fillcore.mac"**, cells are added in a part of the routing area, and connected to the power nets. The area "(20000 -75000) (133000 44000)" matches the block using supply voltage "VDD", and the area "(-130000 -75000) (-42000 55000)" matches the block using supply voltage "VDP".

Figure 9.5 shows the completed design with geometry violations.

```
#--Block using VDD

SROUTE ADDCELL MODEL HDFILL16 PREFIX FC16 NO FN SO FS
  SPIN VDD NET VDD SPIN VSS NET VSS
  AREA ( 20000 -75000 ) ( 133000 44000 ) ;

SROUTE ADDCELL MODEL HDFILL8 PREFIX FC8 NO FN SO FS
  SPIN VDD NET VDD SPIN VSS NET VSS
  AREA ( 20000 -75000 ) ( 133000 44000 ) ;

SROUTE ADDCELL MODEL HDFILL4 PREFIX FC4 NO FN SO FS
  SPIN VDD NET VDD SPIN VSS NET VSS
  AREA ( 20000 -75000 ) ( 133000 44000 ) ;

SROUTE ADDCELL MODEL HDFILL2 PREFIX FC2 NO FN SO FS
  SPIN VDD NET VDD SPIN VSS NET VSS
  AREA ( 20000 -75000 ) ( 133000 44000 ) ;

SROUTE ADDCELL MODEL HDFILL1 PREFIX FC1 NO FN SO FS
  SPIN VDD NET VDD SPIN VSS NET VSS
  AREA ( 20000 -75000 ) ( 133000 44000 ) ;


#--Block using VDP

SROUTE ADDCELL MODEL HDFILL16 PREFIX FC16 NO FN SO FS
  SPIN VDD NET VDP SPIN VSS NET VSS
  AREA ( -130000 -75000 ) ( -42000 55000 ) ;

SROUTE ADDCELL MODEL HDFILL8 PREFIX FC8 NO FN SO FS
  SPIN VDD NET VDP SPIN VSS NET VSS
  AREA ( -130000 -75000 ) ( -42000 55000 ) ;

SROUTE ADDCELL MODEL HDFILL4 PREFIX FC4 NO FN SO FS
  SPIN VDD NET VDP SPIN VSS NET VSS
  AREA ( -130000 -75000 ) ( -42000 55000 ) ;
```

```
SROUTE ADDCELL MODEL HDFILL2 PREFIX FC2 NO FN SO FS
  SPIN VDD NET VDP SPIN VSS NET VSS
  AREA ( -130000 -75000 ) ( -42000 55000 ) ;

SROUTE ADDCELL MODEL HDFILL1 PREFIX FC1 NO FN SO FS
  SPIN VDD NET VDP SPIN VSS NET VSS
  AREA ( -130000 -75000 ) ( -42000 55000 ) ;
```

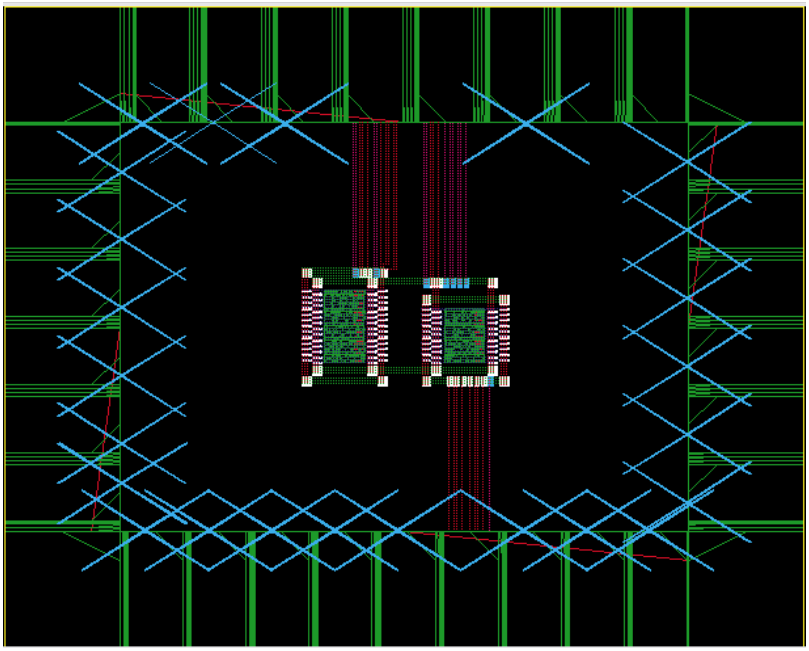


Figure 9.5: Completed design.

Bibliography

[1] J. Bhasker, *A VHDL Primer*, 3rd ed. Prentice Hall, 1999.

[2] P. J. Ashenden, *Designers Guide to VHDL*, 2nd ed. Morgan Kaufman, 2001.

[3] Jiri Gaisler, "High-level design methodology," http://www.gaisler.com/doc/vhdl2proc.pdf.

[4] F. Kristensen, P. Nilsson, and A. Olsson, "A flexible FFT processor," in *Proc. 20th NORCHIP Conference*, Copenhagen, Denmark, Nov. 2002.

[5] ——, "A generic transmitter for wireless OFDM systems," in *Proc. IEEE Conference on Personal, Indoor, and Mobile Radio Communication (PIMRC)*, vol. 3, Bejing, China, Sept. 2003, pp. 2234–2238.

[6] H. Jiang and V. wall, "FPGA implementation of controller-datapath pair in custom image processor design," in *Proc. IEEE Symposium on Circuits and Systems (ISCAS)*, Vancouver, Canada, May 2004.

[7] F. Catthoor, *Custom Memory Management Methodology*, 1st ed. Kluwer, 1998.

[8] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305–327, Feb. 2003.

[9] A. J. Bhavnagarwala, B. Austin, A. Kapoor, and J. D. Meindl, "CMOS System-on-a-Chip Voltage Scaling beyond 50nm," in *Proceedings of the 10th Great Lakes Symposium on VLSI, Chicago, Illinois, USA*, 2000, pp. 7–12.

[10] "Managing power in ultra deep submicron ASIC/IC design," Synopsys, Inc., Tech. Rep., 2002.

[11] K. K. Parhi, *VLSI Digital Signal Processing Systems*. New York, NY: Wiley, 1999.

[12] R. Gonzalez, B. Gordon, and M. Horowitz, "Supply and threshold voltage scaling for low power CMOS," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 8, pp. 1210–1216, Aug. 1997.

[13] Transmeta Corporation, "Transmeta LongRun Power Management," http://www.transmeta.com.

[14] V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez, "Dynamic power management for microprocessors: A case study," in *Proc. Tenth International Conference on VLSI Design*, Hyderabad, India, Jan. 1997, pp. 185–192.

[15] T. Karnik, S. Borkar, and V. De, "Sub-90 nm Technologies-Challenges and Opportunities for CAD," in *Proceedings of the 2002 IEEE International Conference on Computer Aided Design*, Nov. 2002, pp. 203–206.

[16] G. Sery, S. Borkar, and V. De, "Life is CMOS: why chase the life after?" in *Proceedings of the 39th Design Automation Conference, DAC'02*. ACM Press, June 10-14 2002, pp. 78–83.

[17] R. Kumar and C. P. Ravikumar, "Leakage Power Estimation for Deep Submicron Circuits in an ASIC Design Environment," in *Proceedings of the 7th Asia and South Pacific Design Automation Conference and the 15th International Conference on VLSI Design*, Jan. 2002, pp. 45–50.

[18] B. Chatterjee, M. Sachdev, S. Hsu, R. Krishnamurthy, and S. Borkar, "Effectiveness and Scaling Trends of Leakage Control Techniques for Sub-130 nm CMOS Technologies," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design, ISLPED'03*, Aug. 2003, pp. 122–127.

[19] S. Mukhopadhyay and K. Roy, "Modeling and Estimation of Total Leakage Current in Nano-scaled-CMOS Devices Considering the Effect of Parameter Variation," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design, ISLPED'03*, Aug. 2003, pp. 172–175.

[20] L. T. Clark, S. Demmons, N. Deutscher, and F. Ricci, "Standby Power Management for a 0.18μm Microprocessor," in *Proceedings of the 2002 International Symposium on Low Power Electronics and Design, ISLPED'02*, Aug. 2002, pp. 7–12.

[21] ???, "International Technology Roadmap for Semiconductors," http://public.itrs.net.

[22] "Switching activity interchange format (SAIF)," Synopsys, Inc., Tech. Rep., 2002.

[23] $UMC_LIB/../UMCE13H210D3_1.1/lib/umce13h210t3_tc_120V_25C.lib.

[24] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronocs*, vol. 38, no. 8, Apr. 1965. [Online]. Available: ftp://download.intel.com/research/silicon/moorespaper.pdf

[25] I. Toumi, "The Lives and Death of Moore's Law," *First Monday*, Nov. 2002. [Online]. Available: http://firstmonday.org/issues/issue7_11/tuomi/

[26] S. Borkar, "Obeying Moore's Law Beyond 0.18 Micron," in *Proceedings of the 13th Annual IEEE International ASIC/SOC Conference, Arlington, VA*, Sept. 2000, pp. 26–31.

[27] G. E. Moore, "No Exponential is Forever: But "Forever" Can Be Delayed!" in *Proceedings of the IEEE International Solid-State Circuits Conference, ISSCC'03*, Feb. 2003, pp. 20–23.

[28] A. P. Chandrakasan and R. W. Brodersen, "Minimizing Power Consumption in Digital CMOS Circuits," in *Proceedings of the IEEE*, vol. 83, no. 4, Apr. 1995, pp. 498–523.

[29] M. C. Johnson and K. Roy, "Optimal Selection of Supply Voltages and Level Conversions During Data Path Scheduling Under Resource Constraints," in *Proceedings of the 1996 IEEE International Conference on Computer Design*, Oct. 1996, pp. 72–77.

[30] V. Sundararajan and K. K. Parhi, "Synthesis of Low Power CMOS VLSI Circuits using Dual Supply Voltages," in *Proceedings of the 36th Design Automation Conference, DAC'99*. New Orleans, LA, USA: ACM Press, June 21-25 1999, pp. 72–75.

[31] G.-Y. Wei and M. Horowitz, "A Fully Digital, Energy-Efficient, Adaptive Power-Supply Regulator," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 4, pp. 520–528, Apr. 1999.

[32] T. Kuroda, K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, T. Sakurai, and T. Furuyama, "Variable Supply-Voltage Scheme for Low-Power High-Speed CMOS Digital Design," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 3, pp. 454–462, Mar. 1998.

[33] M. Johnson and K. Roy, "Scheduling and Optimal Voltage Selection for Low Power Multi-Voltage DSP Datapaths," in *Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS'97*, June 1997, pp. 2152–2155.

[34] T. Olsson, P. Åström, and P. Nilsson, "Dual Supply–Voltage Scaling for Reconfigurable SoCs," in *Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS 2001*, vol. 2, Sydney, Australia, May 6-9 2001, pp. 61–64.

[35] T. Nakagome, K. Itoh, M. Isoda, K. Takeuchi, and M. Aoki, "Sub-1-V Swing Internal Bus Architecture for Future Low-Power ULSI's," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 4, pp. 414–419, Apr. 1993.

Appendix A

std_logic_1164

```
PACKAGE std_logic_1164 IS

    ------------------------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------------------------
    TYPE std_ulogic IS ( 'U',  -- Uninitialized
                         'X',  -- Forcing  Unknown
                         '0',  -- Forcing  0
                         '1',  -- Forcing  1
                         'Z',  -- High Impedance
                         'W',  -- Weak     Unknown
                         'L',  -- Weak     0
                         'H',  -- Weak     1
                         '-'   -- Don't care
                       );
    ------------------------------------------------------------------
    -- unconstrained array of std_ulogic for use with the resolution function
    ------------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

    ------------------------------------------------------------------
    -- resolution function
    ------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;


    ------------------------------------------------------------------
    -- *** industry standard logic type ***
    ------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;

    ------------------------------------------------------------------
    -- unconstrained array of std_logic for use in declaring signal arrays
    ------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;

    ------------------------------------------------------------------
    -- common subtypes
    ------------------------------------------------------------------
    SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
    SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
    SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')
    SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')

    ------------------------------------------------------------------
    -- overloaded logical operators
    ------------------------------------------------------------------

    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;
```

```
-------------------------------------------------------------------
-- vectorized overloaded logical operators
-------------------------------------------------------------------
FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

function "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;

FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;

-------------------------------------------------------------------
-- conversion functions
-------------------------------------------------------------------
FUNCTION To_bit       ( s : std_ulogic;        xmap : BIT := '0') RETURN BIT;
FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0') RETURN BIT_VECTOR;
FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0') RETURN BIT_VECTOR;

FUNCTION To_StdULogic       ( b : BIT               ) RETURN std_ulogic;
FUNCTION To_StdLogicVector  ( b : BIT_VECTOR        ) RETURN std_logic_vector;
FUNCTION To_StdLogicVector  ( s : std_ulogic_vector ) RETURN std_logic_vector;
FUNCTION To_StdULogicVector ( b : BIT_VECTOR        ) RETURN std_ulogic_vector;
FUNCTION To_StdULogicVector ( s : std_logic_vector  ) RETURN std_ulogic_vector;

-------------------------------------------------------------------
-- strength strippers and type convertors
-------------------------------------------------------------------

FUNCTION To_X01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
FUNCTION To_X01  ( s : std_ulogic        ) RETURN  X01;
FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
FUNCTION To_X01  ( b : BIT               ) RETURN  X01;

FUNCTION To_X01Z ( s : std_logic_vector  ) RETURN  std_logic_vector;
FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
FUNCTION To_X01Z ( s : std_ulogic        ) RETURN  X01Z;
FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
FUNCTION To_X01Z ( b : BIT               ) RETURN  X01Z;

FUNCTION To_UX01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
FUNCTION To_UX01  ( s : std_ulogic        ) RETURN  UX01;
FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
FUNCTION To_UX01  ( b : BIT               ) RETURN  UX01;

-------------------------------------------------------------------
-- edge detection
-------------------------------------------------------------------
FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;

-------------------------------------------------------------------
-- object contains an unknown
-------------------------------------------------------------------
FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
```

```
    FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN;

END std_logic_1164;
```

std_logic_arith

```
--------------------------------------------------------------------------
--
-- -- Copyright (c) 1990,1991,1992 by Synopsys, Inc.  All rights
reserved. -- --
-- -- This source file may be used and distributed without
restriction     -- -- provided that this copyright statement is
not removed from the file  -- -- and that any derivative work
contains this copyrighy notice.          -- --
--
--------------------------------------------------------------------------

library IEEE; use IEEE.std_logic_1164.all;

package std_logic_arith is

    type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
    type SIGNED is array (NATURAL range <>) of STD_LOGIC;
    subtype SMALL_INT is INTEGER range 0 to 1;

    function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
    function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: INTEGER) return SIGNED;
    function "+"(L: INTEGER; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
    function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
    function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;

    function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;

    function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
    function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: INTEGER) return SIGNED;
    function "-"(L: INTEGER; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
    function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
    function "-"(L: STD_ULOGIC; R: SIGNED) return SIGNED;

    function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;

    function "+"(L: UNSIGNED) return UNSIGNED;
```

```
function "+"(L: SIGNED) return SIGNED;
function "-"(L: SIGNED) return SIGNED;
function "ABS"(L: SIGNED) return SIGNED;

function "+"(L: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED) return STD_LOGIC_VECTOR;
function "ABS"(L: SIGNED) return STD_LOGIC_VECTOR;

function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "*"(L: SIGNED; R: SIGNED) return SIGNED;
function "*"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED) return SIGNED;

function "*"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;

function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: SIGNED) return BOOLEAN;

function "<="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: SIGNED) return BOOLEAN;

function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: SIGNED) return BOOLEAN;

function ">="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: SIGNED) return BOOLEAN;

function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: SIGNED) return BOOLEAN;

function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: SIGNED) return BOOLEAN;

function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
```

```
    function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
    function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
    function SHR(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;

    function CONV_INTEGER(ARG: INTEGER) return INTEGER;
    function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
    function CONV_INTEGER(ARG: SIGNED) return INTEGER;
    function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;

    function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: SIGNED; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return UNSIGNED;

    function CONV_SIGNED(ARG: INTEGER; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: UNSIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return SIGNED;

    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER) return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED; SIZE: INTEGER) return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: SIGNED; SIZE: INTEGER) return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC; SIZE: INTEGER) return STD_LOGIC_VECTOR;

    -- zero extend STD_LOGIC_VECTOR (ARG) to SIZE,
    -- SIZE < 0 is same as SIZE = 0
    -- returns STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function EXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;

    -- sign extend STD_LOGIC_VECTOR (ARG) to SIZE,
    -- SIZE < 0 is same as SIZE = 0
    -- return STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function SXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;

end Std_logic_arith;
```

std_logic_unsigned

```
------------------------------------------------------------------------
--
-- -- Copyright (c) 1990, 1991, 1992 by Synopsys, Inc.
-- --                                            All rights
reserved.     -- --
-- -- This source file may be used and distributed without
restriction     -- -- provided that this copyright statement is
not removed from the file  -- -- and that any derivative work
contains this copyright notice.         -- --
--
------------------------------------------------------------------------

library IEEE; use IEEE.std_logic_1164.all; use
IEEE.std_logic_arith.all;

package STD_LOGIC_UNSIGNED is

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function SHL(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function SHR(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;

    -- remove this since it is already in std_logic_arith
    --    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER) return STD_LOGIC_VECTOR;

end STD_LOGIC_UNSIGNED;
```

std_logic_signed

```
------------------------------------------------------------------------
--
-- -- Copyright (c) 1990, 1991, 1992 by Synopsys, Inc.
-- --                                          All rights
reserved.     -- --
-- -- This source file may be used and distributed without
restriction     -- -- provided that this copyright statement is
not removed from the file  -- -- and that any derivative work
contains this copyright notice.          -- --
--
------------------------------------------------------------------------

library IEEE; use IEEE.std_logic_1164.all; use
IEEE.std_logic_arith.all;

package STD_LOGIC_SIGNED is

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "ABS"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;


    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function SHL(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function SHR(ARG:STD_LOGIC_VECTOR;COUNT: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;

-- remove this since it is already in std_logic_arith --
function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER) return
STD_LOGIC_VECTOR;

end STD_LOGIC_SIGNED;
```