NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

READ ONLY MEMORY (ROM)

Graduation Project
COM-400

Student: Ömer GÜLLÜOGLU (20033318)

Supervisor: MSc.,P. Eng. Mehmet Kadir ÖZAKMAN

Nicosia - 2008

ACKNOWLEDGEMENTS

"Firstly, I would like to thank to my supervisor Mcs.Mehmet Kemal ÖZAKMAN, Assoc. Prof Dr. Rahib ABiYEV ,Dr.Kaan UYAR, Mr.Okan DONANGİL,Assist. Prof Dr. Ali DENKER. for his great advise and recomendation for finishing my project properly also, teaching and guiding me in others lectures.

I am greatly indepted to myfamily for their endless support from my starting day in my educational life until today. I will never forget the things that my father Mr. İbrahim Halil Güllüoğlu did for me during my educational life, also I want to say thanks to my mother Mrs. Hafıza Güllüoğlu. I dedicate my project to them.

I thank all the staff of thefaculty of engineering for giving facilities to practise, teaching and solving problem in my complete undergraduation program

I thank my friends Murat KARAOGUL, Doğan BAYRAKTAR, Mehmet TAHTA, Can Emre ALTINKAYNAK, Turgut AYDIN for their help, they get tired with me, and they helped me and give morale evertime.I thank them with my all.

Finally, I promise to do my best in my life as an bachelor of engineer after finishing my undergraduate program"

ABSTRACT

Today is electronic design VHDL is behavioral language to describe design with out going to coınplex of the electronic circuit diagraın. We have done our design using VHDL and verify the design using siınulation tools. We wrote the VHDL code to the synthesis tools which generate detailed electronic circuit.

This Project is designing the ROM. ROM is a class of storage ınedia used in coınputer and other electronic devices.Because data stored ın ROM, it can not be ınodified , it is ınainly used to distribute fırınware This design include behavioral , RTL , logic and layout levels.

The aiın of this project is to develop and design the ROM and what are included in ROM. The program was prepared by using VHDL prograıınıning .

The project contains chips and inforınations about theın, how they work and inputs and outputs of the chips.

TABLE OF CONTENTS

CHAPTER 3

CHAPTER 7

7. Overview of Synthesis for Xilinx Devices

CHAPTERS

8. Coding Styles

CHAPTER9

9. My VHDL project

INTRODUCTION

This project is designing and implementing the ROM. This program was prepared by using VHDL language.

The subjects chapter by chapter so let us go through the overview the chapters in breif:

in the first Chapter VHDL language is described, its properties, components and some examples and application areas of VHDL.

in the Second Chapter I described signals and data types, to look at how signals can be assigned in VHDL, and to give an overview of the rules regarding the use of data types

Third Chapter is about VHDL Operators , how we introduce the main operators that allow us to perform operations on these signals.

The Fourth Chapter presents concurrent and sequentıal statements. we will look at the differences between concurrent statements that execute outside of a process and sequential statements that execute in sequence within a process.

The Fifth Chapter presents Sequential Statements, Our aim during this lesson is to explain further capabilities that a process can offer, we will show how a single process and then a multiple process execute, to build up a picture of a complete simulation in VHDL. Finally we will look at the wait statement and how it is affect the execution ofa process.

in the Sixth Chapter synthesis issues are described.

in the Seventh Chapter Overview of Synthesis for Xilinx Devices

The Eight Chapter describes coding styles. we are going to look at some of the features available in XILINX architectures. We will look at the coding styles which will make best use of these resources

The Ninth Chapter presents my VHDL project, codes of the project and design of the project.

CHAPTERl

WhatisVHDL

The "V" stands for VHSIC (Very High Speed Integrated Circuit), and the "HDL" stands for "Hardware Description Language'.

1.1 **Application** Areas

Let us now look at the stages involved in designing an electronic system, and see which are the main application areas of VHDL.

ELECTRONIC DESIGN PROCESS:

System Specification

Hardware/Software Partition

H/W Spec

S/W Spec

ASIC

PLD

Standard

Parts

Software

Boards of

The System

System Requirements

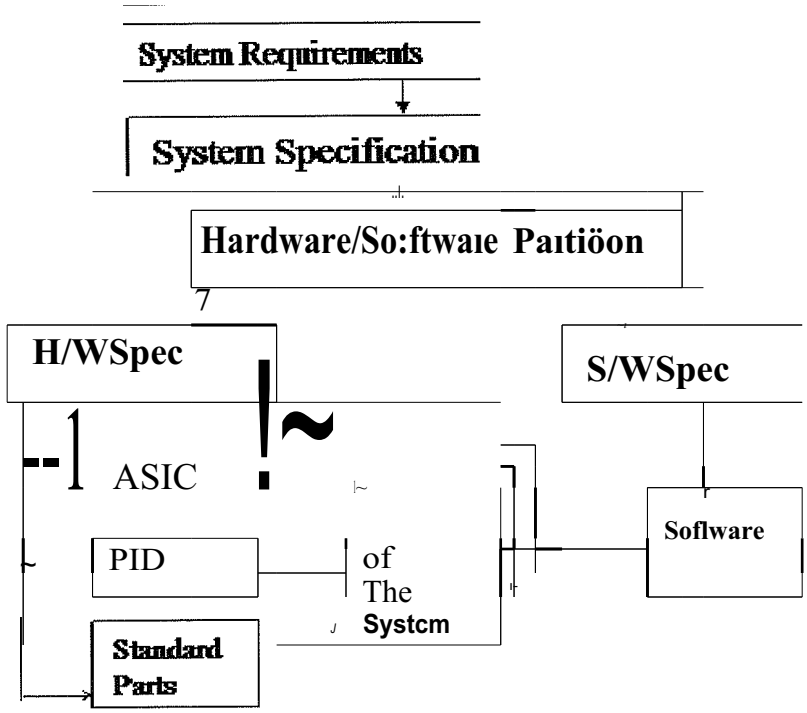


Figure 1- 1 Electronic Design Process

This Diagram shows the complete electronic design Process, from the requirements for the system, through hardware and software partitioning, down to the specification and implementation of the hardware and software parts of the completed system

HARDWARE IMPLEMENTATION:

In the early 1990s, VHDL was being used primarily for complex ASIC design, using synthesis tools to automatically create and optimize the implementation. Then the use of VHDL with synthesis has moved into the area of programmable logic design.

1.2 **Limitation** of VHDL

VHDL is primarily a digital design language. It currently has very limited capabilities in the analog area, and there is a lot of work going on to standardize an

analog version of the language. The 1076 standard defines a language and its syntax, without describing any styles of using it on a design project.

it is possible that VHDL code may need to be slightly modified before it can be used with different synthesis tool set than it was originally written for.

1.3 Levels of Abstraction

The different styles of writing VHDL code are to do with a concept known as abstraction. Abstraction defines how much detail about the design is specified in a particular description of it.

Let's look at the 4 main levels of abstraction to illustrate the principle.

Behavioral

RTL

Logic

Layout

Figure 1- 2 Levels of Abstraction

Layout Level:

The lowest level of abstraction is the layout level. This specifies information about the actual layout of the design on silicon, and may also specify detail timing information and analog effects.

Logic Level:

Above the layout level is the logic level, where we interconnect logic gates and registers. Layout information is ignored, and the design contains information about the function, architecture, technology and detailed timing.

REGISTER

REGISTER

Logic

Figure 1- 3 Register Transfer Level

Register Transfer Level:

At the Register Transfer Level we Use VHDL in a strict style that defines every register in the design, and the logic between them. The design still contains architecture information but not the details of the details of the technology. Absolute timing delays are not specified.

Behavioral;

Above the RTL, we have the behavioral level. This level uses VHDL to describe the function of a design, without specifying the architecture of registers. Behavioral code can contain as much timing information as the designer requires representing his function.

Function

only

Figure 1- 4 Behavioral

1.4 Behavioral Versus RTL

So we see that there are at least two distinct styles of using VHDL: Behavioral, and RTL.

At RTL style the designer has control over the architecture of the registers in his design.

Behavioral

RTL

Synthesis

Logic

Layout

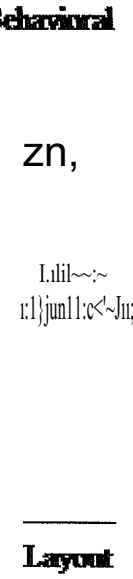
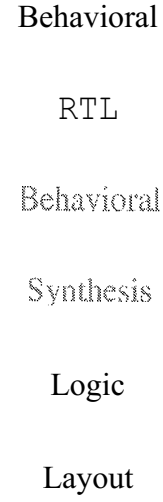


Figure 1- 5 Logic Synthesis

At Behavioral style the synthesis tools generates the architecture.



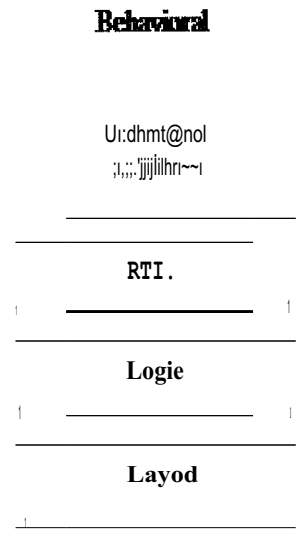


Figure 1- 6 Behavioral Synthesis

1.5 Exercise : Application of VHDL

Which stages in the electronic system design process can VHDL to be applied to?

Please select one of the answers:

A- Only ASIC and programmable logic design.

B- Only specifying the hardware part of the system.

C- Only specifying the system before it is partitioning into hardware and software.

D- All stages mentioned in choices A, B, and C.

1.6 Main Language Concepts

As the VHDL is a Language that allows us to describe hardware, it must be able to describe activities that happening in parallel. Such activities are said to be "concurrent"

1.7 Entity

We will start to look at some VHDL syntax, starting with the main building block of any VHDL design, the entity.

```
Entity HALFADD is

Port (A, B:  in bit;

      SUM,  CARRY:  out bit)

EndHALFADD;
```

HALFADD

SUM

CARRY

A

B

The entity in VHDL describes the interface to a hierarchical block, without defining its behavior. The entity is equivalent to a symbol in schematic based design. Let's look at the syntax for this example ofa HALF ADDER.

The syntax for an entity starts with the keyword "entity", followed by the name of the entity, and the keyword "is". The inputs and outputs are contained within the port statement: This lists the direction, and states that each signal is a single "bit". Finally we have the keyword "end ", end the name of the entity, followed by a semi-colon.

1.8 Architecture

Now we will take a look at how the behavior of an entity is described, using an architecture.

```
Architecture BEHAVE of HALFADD is

Begin

        SUM <= A xor B;

        CARRY <= A and B;

End  BEHAVE;
```

Here we have the architecture of our entity HALFADD that we saw previously. The syntax begins with the keyword architecture, followed by a user defined name of architecture itself, which is BEHAVE in this exarnple. We then have the keywords "is", and "begin". After the beginning of the architecture, we have the staternents defining the actual behavior. In this case, we have two signal assignrnents: the result of "A xor B" is assigned to the output SUM and the result of "A and B" is being assigned to the output "CARRY".

Entity Can Have Multiple Architecture

Entity

Architecture

X

Architecture

y

Architecture

Z

And finally it is important to note that an entity can have more than one architecture. üne usual application of this is when a design is described at several levels of abstraction: there may be behavioral, RTL and gate level descriptions of the same design.

1.9 Representing Hierarchy

The entity and architecture are the main building blocks from which hierarchy is build. Let's now look at how hierarchy is represented in VHDL, using the well known example of a full adder being build from two half adders, and a gate.

SUM

CARRY

A

B

CIN

H.A.a

H.A.a

OR

entity FULLADD is

port (A, B, CIN : in bit;

SUM, CARRY : out bit) ;

End FULLADD;

FULLADD

A

B

CIN

SUM

CARRY

The code is shown here for the entity of FULLADD. As you can see, all of the five ports are declared as single bits, in the same style as we saw for the "HALF ADDER".

SUM

CARRY

A

B

CIN

ul

u2

u3

12

13

Here we can see the architecture of FULLADD. Start by looking at the three lines of code between the 'begin' and the 'end' statements. Each of these statements creates an instance of another entity, similar to placing down a symbol in a schematic capture package.

Looking at the syntax, we have the instance name (ul), and following the colon we have the name of the entity we are making an instance of. Then we have the key words "port map", and then a list of the signals within FULLADD that are connected to the ports of our design "HALFADD"

Entity HALFADD is

Port (A, B: in bit ;

SUM, CARRY: out bit);

EndHALFADD;

ul: HALFADD port map (A, B, il, 12);

A

A

B

B

HALFADDDDD

SUM

CARRY

il

12

1.10 Local Declarations

```
ComponentDeclarations

Architecture STRUCT of FULLADD is

        signal il, 12, 13 : bit;

Component declarations

Begin

        ul: HALFADD port map (A, B, il, 12);

        u2: HALFADD port map (il, CiN, SUM, 13);

        u3: ORGATE  port map (13, 12, CARRY);

endSTRUCT;
```

Before we can make an instance of an entity such as 'HALFADD' it is necessary to declare the interface to the entity locally in the FULLADD architecture. This is done with what is called a component declaration.

```
Component HALFADD

        port (A, B : in bit:

                SUM, CARRY:  out bit);

End component;
```

The component declaration has the same style of syntax as the entity. in 99% of situations, you will want your component to have the same name and the same port list, as your entity. There is no getting around this: the component declaration must be present.

```
entity HALF ADD is
port (A, B : in bit;
        SUM, CARRY:  out bit)
End HALFADD;
```

Coınplete code for the FULLADD design:

```
Entity FULLADD is
Port (A, B, CJN: in bit;
        SUM, CARRY:  out bit);
End FULLADD;
Architecture STRUCT of FULLADD is
        Signal il, 12, I3 : bit;
Component HALFADD
Port (A, B : in bit;
        SUM, CARRY:  out bit;
End component;
Component ORGATE
Port (A, B : in bit;
        Z: out bit;
End component;
```

```
Begin

ul :HALFADD port map (A, B, 11, 12);

u2 :HALFADD port map (11, CIN, SUM, I3);


u3 : ORGATE port map (I3, 12, CARRY);


endSTRUCT;
```

The complete code for the FULLADD design is shown here. In an architecture, note that we declare the local signals and component before the "begin", and the description of the design' s function is between the "begin" and "end".

1.11 Configurations

The next VHDL object we will look at, is called configuration. Consider that a full design hierarchy consists of entities, each of which can have multiple architectures.

entity A

archX

arch Y

entity B

archX

arch Y

entity C

archX

arch Y

entity C

archX

arch Y

The configuration is like a parts list, specifying which architecture is to be used for each entity in the design.

entity	arch
A	y
B	X
C	X
D	y

Table 1- 1 Configuration

You are not required to give this 'parts' list information for all of your design hierarchy. lf no configuration information is specified for an entity, then the last architecture that was compiled by the simulator is the one that is used during the simulation.

it is common for complete design to only have on architecture for every entity and not require configuration.

Some simulators allow this, while others require at least a Default configuration". Always using a default configuration makes your code more portable.

Default Configuration;

entity A

archX

entity B

archX

entity C

archX

entity C

archX

1.12 Processes and Types

Process contains sequential statements:

The process is a region of VHDL code, inside which statements execute in sequence. A process exists inside an architecture, and multiple processes interact with each other concurrently.

```
entity ORGATE is
        port (A, B: in bit;
              Z: out bit);
EndORGATE;
Architecture BERAVE of ORGATE is
Begin
```

```
OR_FUNC:  process  (A,B)
begin
        if (A=' 1' or B=' 1') then
else

        Z<='O';
End if;
End process  OR_FUNC;
```

1.13 Packages

A package contains a collection of definitions that rnay be referenced by rnany designs at the sarne time. Placing definitions that are cornınon between designs in one location helps a design tearn to work rnore consistently.

```
package  PROJECT _X is
-----defini tions
end PROJECT_S;
```

1.14 Compilation and Libraries

The final concepts of the language that we will cover in this introduction section are how VHDL source code is cornpiled, and how designs are grouped together into a library.

Coınpilation or Analysis:

design.vhd

binary file

01011010100111000110101000110010 10

Compilation

(analysis)

Errors

1.15 The complete Picture

We will finish this lesson by looking at the complete picture of how they relate to each other in a typical design situation.

entity A

archX

entity B

archZ

entity C

arch Y

Config

CFG_A

Package Pl

Package P2

CHAPTER2

Signals and Data Types

2.1 What You Will Learn in This Lesson

The aim of this lesson is to look at how signals can be assigned in VHDL, and to give an overview of the rules regarding the use of data types.

First we will consider the concept of a data type, and look at the standard types that are defined as part of the language. Then we will describe the concept of array, which is like a bus within schematic base design. We will look at many different examples of how single bit and array signals can be manipulated within VHDL.

Then we will look at how you can define your own data type, e.g "enumerated type"

And finally we look at the definition ofa type known as standard for describing signals in VHDL.

2.2 The Concept ofa Type

First we review the concept of a data type, and look at where the data type of a signal is specified.

Type concept:

```
Architecture STRUCT of FULLADD is

Signal N_SUM: bit;

                =Other declarations

begin

                -- code
```

endSTRUCT;

2.3 StandardData Types

Pre-defined types:

Having reviewed the conceps of data type, we will now look at the standard types that pre-defined within the VHDL language. All of this types are specified as VHDL source code, and the language defines that they should be found in a package called STANDARD.

```
package STANDARD is

    type BOOLEAN is  ( FALSE, TRUE);

    type BiT is ('O', '1');


end STANDARD;
```

2.4 Signals And Drivers

We will look at the concepts behind an assignment to a signal.

```
Signal A, B, Z : bit;

Signal X_INT: integer;


Z <=A;
```

Signal assignment

Statement

Signal assignment statement:

A signal is assigned a new value with what is known asa 'signal assignment", as we have seen in the examples of half adder and full adder.

Multiple Drivers:

If more than one assignment is made to a signal, then that signal must be declared to be a special kind of type, known as resolved type. A resolved type has a function cali associated with it to resolve the final value of the signal. This only used in very special cases in hardware. Normally it is illegal to use it.

2.5 Arrays

We will look at how arrays are defined and manipulated.

Elements of same type:

'1'

'O'

'1'

'O'

Multiple objects

Each element of the same type

An array is a collection of objects, where each one is of the same type. The language defines two types of array: bit vector, and string.

Size When Declared:

The range of an array is defined when the array is declared.

Example:

Legal declaration:

Signal Z_BUS: bit_vector (3 downto 0);

Signal C_BUS: bit_vector (1 to 4);

This example shows two bit_vectors, where each one is four bits wide. The range may be declared using either the "to" or "downto" notation. However the appropriate keyword must be chosen.

illegal declaration:

Signal Z_BUS: bit_vector (0 downto 3);

Signal C_BUS: bit_vector (3 to 0);

Array Assignments:

Two array objects can be assigned to each other, as long as they are of the same type and same size: Note that assignment is by position, and tot by index number. There is no concept of a most significant bit defined with the language.

Signal Z_BUS: bit_vector (3 downto 0);

```
Signal C_BUS:  bit_vector  (1 to 4);
```

```
Z_BUS <= C_BUS:
```

Is the same as:

```
Z_BUS(3) <= C_BUS(l);

Z_BUS(2) <= C_BUS(2);

Z_BUS(l) <= C_BUS(3);

Z_BUS(O) <= C_BUS(4);
```

Array slices:

A "slice" of an array may be referenced, including a single element. The direction of the slice (i.e. to or downto) must match the direction in which the array is declared.

Legal:

```
Z_BUS (3 downto 2) <= "00";

C_BUS(2 to 4) <= Z_BUS (3 downto l);
```

illegal:

```
Z_BUS (2 to 1) <= "11"
```

2.6 Concatenationand Aggregates

Concatenation and aggregates are the two methods for associating signals together in order to assign them to array objects.

Concatenation:

```
signal Z_BUS : bit_vector (3 downto 0);

signal  A, B, C, D: bit;

signal BYTE : bit_vector (7 downto 0);

signal A_BUS : bit_vector (3 downto 0);

signal B_BUS : bit_vector (3 downto 0);



Z_BUS <= A & B & C & D;



BYTE <= A_BUS & B_BUS;
```

concatenation

operator

VHDL provides the ability to associate single bits and vectors together, to form array structures. This is known as concatenation, and uses the ampersand (&) operator. The examples here show that single bits, and bit vectors can be concatenated together to forni new vectors.

Aggregates:

```
signal Z_BUS : bit_vector (3 downto 0);

signal  A, B, C, D : bit;

signal BYTE : bit_vector (7 downto 0);

                Z_BUS <= ( A , B , C , D);
```

Aggregate

Equivalent to:

```
Z_BUS(3) <= A;

Z_BUS(2) <= B;

Z_BUS(l)  <= C;

Z_BUS(O)<= D;
```

Another rnethod of assigning to elernents of an array is known as aggregate. An aggregate is contained within round brackets, and assignrnent to each element are separated by cornrnas (,).

Specifying elements by name:

```
signal Z_BUS : bit_vector (3 downto 0);

signal  A, B, C, D:  bit;

signal BYTE : bit_vector (7 downto 0);



X <:± ( 3=>'1', 1 downto 0=>'1', 2=>B);
```

It is possible to specify the element of array to assign by name, as well as by position.

This example also shown that, a range of the array can be specified, as long as the same value is being assigned to each element in the range.

Others statement:

```
signal Z_BUS : bit_vector (3 downto 0);

signal A, B, C, D : bit;

signal BYTE: bit_vector (7 downto 0);


X <= ( 3 => '1', 1 => 'O', others => B);
```

Aggregates have the ability to use the "others" statement, which will assign a value to all other elements of the array that have not been specified. Finally, not all synthesis tools support the use of aggregates, so you may be required to use concatenation to perform array manipulation.

2.7 Type definition

Having looked at the standard types that are pre defined in the language, we will now look at how the user can define his own types in VHDL.

Enumerated type:

A user defined type in VHDL is known as an "enumerated" type. Types are most commonly defined inside a package, architecture or process.

Type declaration:

Here is the syntax for defining an enumerated type.

```
Type MY_STATE is
        (RESET, IDLE, RW_CYCLE, INT_CYCLE);
```

After specifying the type name, we have a list of values that an object of the type can contain, with each value separated by comma.

Signals of defined type:

```
Type MY_STATE is
        (RESET, IDLE, RW_CYCLE, INT_CYCLE);

signal STATE: MY_STATE ;
signal TWO_BIT: bit_vector (Oto l);

        STATE <= RESET;                 OK

        STATE <= 00;                    Not OK
        STATE <= TWO_BIT;               Not OK
```

Having defined the type, then we can define signals to be of that type.

Here we have declared the signal STATE to be of type MY_STATE. We have toobserve the strict rules when using tis signals: we cannot assign anything to signal state which is not of type MY_STATE.

Synthesis of enumerated types:

type MY_STATE is

(RESET, IDLE, RW_CYCLE, INT_CYCLE);

Encoding left to right in binary sequence

RESET "00"

IDLE "Ol"

RW_CYCLE "10"

INT_CYCLE "11"

Most synthesis tools can build logic from a signal which is of an enumerated type. Usually the signal has minimum number of bits required to represent the number of possible values. This example shows how each value is usually encoded in the hardware implementation.

2.8 Multi Valued Logic

2.9 Standard Logic

Standard_logic_1464 package:

Type definitions are contained within a package called standard_logic_l 164.

Package

Standard_logic_l 164

Standard logic type definitions are contained within a package called standard_logic_1 164.

We will see how to reference this package.

2.10 Using Standard Logic

'library' and 'use' clauses:

```
library IEEE;
use IEEE.Std_logic_1 164.all;
entity MYDESIGN is
        port (A,B, : in std_logic;
                Z  : out std_logic);
end MYDESIGN;
```

The package of definitions is contained within a library called IEEE. It is necessary to reference both the library and the package, as shown in the diagram. You will use these lines of code before every entity you write that uses Standard Logic.

CHAPTER3

VHDL Operators

3.1 What You Will Learn in This Lesson

Main operators:

Having learned about signals and their types, we now introduce the main operators that allow us to perform operations on these signals.

Logical:

We will cover the logical operators, such as "and" and "or".

Relational:

Then we will look at the relational operators that used to compare different values.

Arithmetic:

And finally we will look at the arithmetic operators for performing mathematics.

3.2 Logical Operators

And, **or, nand, nor, xor, not:**

And, or, nand, nor, and xor all have the same precedence, and execute from left to right across a statement. "not" has a higher precedence, and is therefore executed before the other operators in an expression.

Example:

Library IEEE:

```
Use IEE.Std_Logic_l   164.all;

Entity MYDESIGN  is

        Port (A, B, C : in std_logic;

                Z       : out std_logic);

        End MYDESIGN;

Architecture  MYD of MYDESIGN  is

Begin

        Z <= A and not (B or C);     -- to overcome the precedence

EndMYD;
```

3.3 **Relational Operators**

>, >=, <, <=, /=, =.

=	Equal to
>=	Greater than or equal to
<=	Less then or equal to
<	Less than
>	Greater then

Return a Boolean:

```
If A = B then

 Z<= '1";
```

Else Z <= 'O';

and if;

They each return a Boolean value, and are most often used within a if-then-else statement to control the flow of code depending on different conditions.

Operands of same type:

The rules for using relational operators are that the operands must be of the same type. For an array, the operands can be of different lengths: the operands are aligned to the left, and compared to the right.

Arrays have no numerical meaning:

No nuınerical meaning, associated with a vector: it is just a collection of objects of the same type.

3.4 Arithmetic Operators

+, * /, ** rem, mod, abs

+

Addition

******

Exponential

abs

Absolute value

ınod

modulus

rem

Remainder

Subtraction

\ *

Multiplication

/

Division

The arithmetic operators are listed here. They are pre-defined for types integer, real, (except for modulus and remainder), and type time. As vectors do nt represent a numerical value, the arithmetic operators do can not be used with types bit_vector, stdj1logic., vector, or std_logic_vector.

Generally operands of same type:

Signal A, B, Z : integer;

Z <=A+B;

Generally the operands need to be of the same type. in this example we are adding two integers and the result returned is also an integer.

CHAPTER4

Concurrent and **Sequential Statements**

in this lesson we will look at the differences between concurrent statements that execute outside of a process and sequential statements that execute in sequence within a process.

4.1 What You Will Learn in this Lesson

4.2 Concurrent Assignment Statements

Concurrent statements are those which can execute at the same time in parallel. This differs from sequential statements, which can only execute one at a time, in sequence.

Execute in parallel, order independent

X<=A+B;

Z<=C+X;

Z<=C+X;

X<=A+B;

The behavior of concurrent statement is independent of the order in which they are written. In this example, we can write either statement first, and the result is the same: the result of A+B is used as an input to the adder that defines Z.

Assign to self is OK for software:

X <=X+ Y;

Register X

Register Y

+

IN SOFTWARE

There are some things that you may have done when writing traditional software, they may not make much sense in describing the hardware.

in software, X and Yare register locations: We take the contents of X, add it to Y, and store the result in register X.

Not OK for hardware:

X<=X+ Y;

+

y

X

X

INHARDWARE

As a concurrent statement, the same line of code is describing an adder, with no implied storage registers. Hence we have described feedback around combinational logic. As you can see it is really important to think about the fact that you are describing hardware when you write VHDL.

4.3 The Case Statement

Considers possible values of object

case OBJECT is

```
When VALUE_1 =>
        -- statements
when VALUE_2 =>
--statements

when VALUE_3 =>
        --statements
    ……….

end case;
```

CHAPTER5

Sequential Statements

5.1 What You Will Learn in this Lesson

Our aim during this lesson is to explain further capabilities that a process can offer, we will show how a single process and then a multiple process execute, to build up a picture of a complete simulation in VHDL. Finally we will look at the wait statement and how it is affect the execution ofa process.

5.2 Concurrent Versus Sequential

First let' s review some of the concepts from previous lesson.

Process model:

Remember that our model of VHDL simulation consists of multiple processes executing sequentially, and interacting concurrently.

Process within architecture:

We said that this model maps to VHDL code by having any number of processes within architecture,

Concurrent in architecture:

and we said that the process is seen a single concurrent statement within the architecture.

Multiple Processes in architecture:

It is possible for architecture to contain any number of concurrent statements, and therefore any number of processes.

5.3 Signal Assignment in a Process

Our final review of the previous lesson is how signal values assigned within a process are actually updated.

Concurrent statements: multiple drivers

Remember that we looked at the example in section 4.4, where we have two assignments to the signal Z, made outside of the process. We saw that statements outside of a process are concurrent and so Z has two drivers and must be of of a resolved type, allowing the final value to be determined.

inside process one driver:

Then we looked at this similar example, where the two assignments to Z are inside a process, and we said that a process can only define one driver ona signal.

The language defines that within a process it is the last assignment made to a signal that takes affect, but only when the process suspends, which in this case is at the bottom of the process.

Different meaning

process (A, B, C, D)

begin

Z<=AandB;

Z<= C andD;

end process;

is not equal to

architecture X of multiple is

begin

Z<=AandB;

Z<= C andD;

endX;

Hence the same two statements have a very different meaning, depending on whether they are executed inside or outside ofa process.

5.4 Multiple Process Calls(l)

Having reviewed the basic concept of how signals are updated within a process, we will now move on to see how a process can potentially be executed several times before all signals are updated.

Example: Signal assigned then read

Example: process (A, B, M)

Begin

Y<=A;

M<=B;

Z <=M;

End process Example;

in this example we have a process with an assignment to a signal M, and the value of M is also read within the same process. Let us use this example to see how a process executes in more detail.

Event on B

Let' s assume that there has been an event on the signal B, which has just transitioned from 0 to 1. The process is called because B is in its sensitivity list.

M assigned when the process suspends

The value of Mis not updated when the line M< B; is executed, but when the process suspends at he bottom.

Z not updated

Hence when the line "Z<=M; is executed, the value of M is still 0, and Z is not updated with a 1 when the process suspends.

5.5 Multiple Process Calls(2)

Process suspends: M updated, Z is not

After the process suspends, M is updated with the new value of 1, but Z remains with the value of 0.

Process called again

The process is now called again, because there has just been an event on M, and Mis in sensitivity list.

Z to get 1 on the second cali

Now on the second call, the assignment of M to Z will cause Z to be updated with the value of 1 when the process suspends at the bottom.

Z is finally updated

So when the process suspends this time, Z is finally updated with a 1.

5.6 **Variables**

Let' s look at a new type of object in VHDL, called the variable.

```
Process (A, B, C)
        Variable M, N:  integer;
                Begin
                        M:=A;
                        N:=B;
                        Z<=M+N;
                        M:=C;
                        Y<=M+N;
End process;
```

Assigned immediately

The variable is an object in VHDL that can only be declared and used within a process. A variable is different to a signal in that it is assigned its value immediately, like in traditional software.

Use only in process declared

To make assignments easy to recognize, it is required to use the ":=" notation, rater than the "<=" notation used for signal assignments. A variable can only be used within the process that it is declared: this is known as the "scope" of the variable.

Retains value

A variable retains its value between calls to a process, and so is possible to imply the behavior of a memory element if required. To prevent any problems when describing combinational logic, always assign a value to your variable before you read its value, as we have done in this example.

Mix with signals: match types

And finally, it is possible to freely assign signals to variables, and variables to signals, as we have done in this example. However, note that the rules about matching data types still need to be observed.

5.7 Variable Usage Model

Having considered the rules about how a variable behaves in VHDL, let's now look at the situations where you are likely to use a variable in your design.

Intermediate values

Because a variable is updated immediately, it is commonly used to calculate an "intermediate" value within a process, which is then used later on in the process execution.

CHAPTER6

Synthesis Issues

6.1 What You will Learn in This Lesson

There are three main subject areas that this lesson will cover.

-What is RTL STYLE?

First we will look at ways of describing registers in VHDL, building up a definition of what is a register transfer level style.

-Vector arithmetic

You will recall from the VHDL operators lesson that it is not possible to perform arithmetic on vectors in VHDL by default. However there is a way around this, and the next part of the chapter covers this subject.

-Synthesis of arithmetic operators.

Finally we will look at how arithmetic operators can be synthresized in VHDL, and consider the effect this will have on how you should write your code for synthesis.

6.2 Specifying Registers in VHDL

So first let' s look at how to describe registers in VHDL.

A Clocked Process

```
entity FLOP is

port  (D, CLK: in stdJogic;

       Q : out std  Iogic);
```

```
endFLOP;


architecture  A of  FLOP  is

begin

        process

        begin

                Wait  until  CLK' event  and CLK='  1';

\               Q<=D;

        end process;

end A;
```

The basic style of describing registers is to have a process which only executes when there is a rising edge of clock. Here, Q is only updated with the value of D on a rising clock, and will not be updated if D changes at any other time. Hence it describes the function of a register in hardware.

Registers on all signals

The signals which are assigned to within the process are only assigned new values on a rising clock edge. Hence ALL signals assigned to are implemented as the outputs of registers in the synthesized hardware.

Clocked process

```
process

begin

        Wait until CLK'event  and CLK='l';
```

```
Q<=D;

end process;
```

6.3 Detecting a Rising Clock

```
process

begin

wait until

 Z<=  A+B;

Y <=  C+D;

end process
```

There are many ways to detect a rising edge of clock in VHDL, and some methods are supported by more synthesis tools than the others. Let's look at the different methods, and the issues in choosing which one to use.

Wait until forms

```
process

begin

        wait until CLK='1';

        Q<=D;

 end process;

 process

 begin

         wait until CLK' event and CLK ='1';
```

```
Q<=D;

end process;
```

These two forms use the "wait until" statement. They both wait for an event on the clock, however the second form uses the additional "CLK' event" syntax to double check for the clock event. This version is more commonly supported by synthesis tools, and is therefore recommended.

6.4 Clock Process Rules

```
 process

\ begin

wait until



=combinational logic

end process
```

Let' s look at some additional rules regarding the format of code within a clocked process.

```
Wait is first statement in process

Process

Begin

        Wait until CLK' event and CLK=' 1';

        -- statements

end process;
```

The general rule about using the wait statement to detect the clock edge is that it must be first, and only wait statement in the process. All other statements describe combinational logic that has registered outputs.

If form: all logic within if statement

```
Process (CLK)
Begin
        If ( CLK'event and CLK='1') then
        Q<=D;
\
end if;
        end process;
end process;
```

The rule regarding the "if" form of the clock process is that there should be no other statements outside of the if structure, and that the structure should have no else clause. All code inside the if statement describes combinational logic.

Asynchronous reset registers

```
process (CLK, RST)
Begin
        if (RST ='1') then
                Q<= 'O';
        elsif (CLK'event and CLK='1') then
                Q<= D;
        end if;
```

```
end process;
```

to specify registers with asynchronous resets, then it is necessary to use the code structure shown here. In this case, an event on RST causes the process to execute, and Q to be reset to '0'. Otherwise it is only an event on the clock which can cause Q to be updated with D.

Asynchronous reset rules

The rule for the asynchronous reset form of the clocked process is similar to that for the "if" form: No other statements outside of the if structure, and ther should be no else clause after the edge detection.

CHAPTER7

Overview of Synthesis for Xilinx Devices

7.1 Introduction

In this chapter we will be setting scene and giving a very general overview of some of the issues involved in using VHDL for the design of Xilinx devices.

7.2 What we will cover

Before we begin this chapter, let's first look at the topics we will cover.

PLD technologies:

PLD

ASIC

VERSUS

First we will discuss PLD architectures , how they differ from ASICs and how this effects design methodologies. Then we will architectures of two types of Xilinx PLD. These are FPGAs and CPLDs.

Xilinx Architectures:

Implementation in a Xilinx Device

Synthesis

VHDL Design Entry

7.3 Xilinx Device Architectures

We will give you an overview of the Xilinx device architectures.

7.4 FPGA Technologies

Xilinx FPGA technologies are based on static RAM building blocks which need to be downloaded with their logical function each time the system is powered up.

Configurable Logic Blocks (CLB):

Each building block in a given Xilinx FPGA device is known as Configurable Logic Block (CLB)

A CLB is a cell consisting of 8 or more inputs, 3 or more outputs, some combinational logic and two or more registers.

Connection by programmable switches:

The CPLDs are arranged into a fixed matrix and are connected by programmable switches. These form the required signal nets between CLBs.

CLB

CLB

CLB

CLB

7.5 CPLD Technologies

CPLD technologies are not based on static RAM building blocks and they don't need to be downloaded with their logical function each time the system is powered up. 'They have bigger physical size, and accommodate less function. CPLD technologies is mainly useful for smaller applications. CPLDs are based on a different type of building

block than used in FPGAs. In a CPLD, each building block is referred to as a Function Block (FB). Each FB is comprised of macro cells, each capable of implementing a combinational or registered function.

These function blocks are connected via a switch matrix.

7.6 Where PLD Specific Issues Occur

So, having looked at the architecture of PLD technologies, we can now move on to look at the main areas of interest from the coding style and synthesis point of view.

Efficient mapping to PLD architecture

The synthesis tool must be able to map the VHDL Code into an efficient utilization of CLBs.

However, the style in which you write your code can help the synthesis tool to obtain better results.

Good Coding Style

Good coding style means that the synthesis tool can identify constructs within your code that it can easily map to technology features.

7.7 How the PLD Specific Issues Are Handled

The key to using PLD resources efficiency is to write your VHDL so that it makes the best use of the architectures available within the target device.

Architecture independence

An important advantage of designing with VHDL is that the description can be independent of architecture, an can be re targeted to a new architecture if required. However the code that is purely generic may not make the most efficient use of architecture specific features.

Architecture independence versus efficiency

A trade off exists here. You might want to keep the code architecture independent, leaving the synthesis tool to infer the best resources for the target device, or you might use architecture specific constructs to exploit pre-optimized functions at he expense of architecture independence.

CHAPTER8

Coding Styles

Now we are going to look at some of the features available in XILINX architectures. We will look at the coding styles which will make best use of these resources.

8.1 Coding Styles Introduction

The terminology will be based on FPGAs however many of the issues are applicable to CPLDs. Where there are significant requirement differences we will point these out.

8.2 Using Modules

A very important factor for efficient resource usage is the question of utilizing modules. You will find that Xilinx have already identified the most commonly used complex functions and developed optimized modules that can perform these tasks.

This has two advantages for designers: they do not need to describe the behavior of the function in their code, and they don't need to worry about implementation optimization as this has already been done by Xilinx. All they need to do is instantiate these modules into the code.

8.3 LogiBOX Facility

Xilix provides a tool called LogiBLOX, which is a graphical interactive tool for creating modules, such as counters, shift registers and multiplexers. LogiBLOX includes both a library of generic modules and a set of tools for customizing them.

This means that you can create your own library of functions that are optimized towards the architecture and technology of your choice. You can instantiate these functions in your code as and when you need them.

LogiBLOX alsa generates simulateable VHDL modules, so you can simulate the function available in your target family series, and create architecture specific, optimized implementation of your chosen functions.

8.4 Synthesis of Arithmetic Operators

Sometimes small pieces of code can synthesize to large pieces of logic and this is particularly true of arithmetic functions. ldeally you should think carefully how to control the implementation of such functions.

Arithmetic functions might be best generated as a logiBLOX module.

8.5 Encoding for State Machines

Having discussed how to access modules from a VHDL based design, we are now going to look at how state machines can be implemented in Xilinx devices from VHDL.

Use **of enumerated types**

Most state machines written in VHDL use an enumerated type to define the state signal, as shown in the code here.

```
Architecture RTL of FSM is

    Type T_STATE is ( IDLE, RW_CYCLE, INT_CYCLE, DMA_CYCLE);

    Signal NEXT_STATE, STATE : T_STATE;

Begin
```

Default encoding

By default in VHDL, an enumerated type is usually encoded as a binary sequence, mapping from the value 0 throug the sequence of values which are listed in the type definition. However in the XILINX synthesis tool, the default encoding is a one-hot style.

Binary encoding

IDLE	00
RW_CYCLE	01
INT_CYCLE	10
DMA_CYCLE	11

One-hot **encoding**

IDLE	0001
RW_CYCLE	0010
INT_CYCLE	0100
DMA_CYCLE	1000

Changing the encoding

Within the Xilix tools you can choose between binary or one hot encoding from the user interface or specify a different encoding from from within the VHDL code.

En example of the VHDL syntax which can be used to change the encoding is shown here.

```
Type STATE_TYPE is (Sl, S2, S3, S4);

Attribute ENUM_ENCODING: STRING

Attribute ENUM_ENCODING of STATE_TYPE: type is "000100100100 1000"

Signal C_STATE, N_STATE : STATE _TYPE:
```

CHAPTER9

My **VHDL project**

My design is Read-only memory (usually known by its acronym, ROM) is a class of storage media used in computers and other electronic devices. Because data stored in ROM cannot be modified (at least not very quickly or easily), it is mainly used to distribute firmware (software that is very closely tied to specific hardware, and unlikely to require frequent updates).

Use of ROM for program storage. Computer requires the ROM to store the inital program that runs when the computer is powered .(a process known).

I design ROM an put in values using VHDL.

9.1 **This is my VHDL code**

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:

-- Create Date:    16:24:15 03/28/2008
-- Design Name:
-- Module Name:    ROM - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:

-- Dependencies:

-- Revision:
```

```
-- Revision  0.0l - File  Created
-- Additional  Comments:

----------------------------------------------------------      -----
library IEEE;
use IEEE.STD_LOGIC_l 164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ROM is
    Port ( Clock:  in  STD_LOGIC;
           Reset: in  STD_LOGIC;
           Enable:  in  STD_LOGIC;
            Read: in  STD_LOGIC;
           Address  : in  STD_LOGIC_VECTOR (4 downto 0);
           Data_out:  out  STD_LOGIC_VECTOR (7 downto 0));
endROM;

architecture Behav of ROM is

    type ROM_Array is array (Oto 31)

          of std_logic_vector(7 downto 0);

    constant Content: ROM_Array := (

        O=> "00000001",                -- Suppose ROM has

        l => "00000010",               -- prestored value
```

```
        2 => "00000011",                    -- like this table
        3 => "00000100",
        4 => "00000101",
        5 => "00000110",
        6 => "00000111",
        7 => "00001000",
        8 => "00001001",
        9 => "00001010",
        10 => "00001011",
        11 => "00001100",
        12 => "00001101",
        13 => "00001110",
        14 => "00001111",
        OTHERS => "11111111"
            );
begin
    process(Clock, Reset, Read, Address)
```

```
  begin

    if( Reset = '1' ) then
        Data_out <= "ZZZZZZZZ";
    elsif( Clock'event and Clock = '1' ) then
       if Enable = '1' then
            if( Read = '1') then
               Data_out <= Content(conv_integer(Address));
            else
           Data_out <= "ZZZZZZZZ";
            end if;
        end if;
    end if;
  end process;

endBehav;
```

9.2 **The ROM created by the synthesis**

```
Release 9.1i - xst J.30
Copyright (c) 1995-2007 Xilinx, ine.  All rights reserved.
--> Parameter TMPDIR set to ./xst/projnav.tmp
CPU : 0.00 / 1.67 s | Elapsed : 0.00 / 1.00 s

--> Parameter xsthdpdir set to ./xst
CPU:  0.00 /  1.70 s | Elapsed:  0.00 /  1.00 s

--> Reading design: ROM.prj

TABLE OF CONTENTS
  1) Synthesis Options Summary
```

2) HDL Compilation

3) Design Hierarchy Analysis

4) HDL Analysis

5) HDL Synthesis

5.1) HDL Synthesis Report

6) Advanced HDL Synthesis

6.1) Advanced HDL Synthesis Report

7) Low Level Synthesis

8) Partition Report

9) Fiiial Report

9.1) Device utilization summary

9.2) Partition Resource Summary

9.3) TIMING REPORT

---

* Synthesis Options Summary *

---- Source Parameters

Input File Name : "ROM.prj"

Input Format : mixed

Ignore Synthesis Constraint File : Nü

---- Target Parameters

Output File Name "ROM"

Output Format :NGC

Target Device : xc3s200-5-ft256

---- Source Options

Top Module Name :ROM

Automatic FSM Extraction :YES

FSM Encoding Algorithm : Auta

Safe Implementation :No
FSM Style : lut
RAM Extraction : Yes
RAM Style : Auto
ROM Extraction : Yes
Mux Style : Auto
Decoder Extraction :YES
Priority Encoder Extraction :YES
Shift Register Extraction :YES
Logical Shifter Extraction :YES
XOR Collapsing :YES
ROM Style : Auto
Mux Extraction :YES
Resource Sharing :YES
Asynchronous To Synchronous : Nü
Multiplier Style : auto
Automatic Register Balancing : No

---- Target Options
Add IO Buffers : YES
Global Maximum Fanout : 500
Add Generic Clock Buffer(BUFG) : 8
Register Duplication :YES
Slice Packing :YES
Optimize Instantiated Primitives : Nü
Use Clock Enable : Yes
Use Synchronous Set : Yes
Use Synchronous Reset : Yes
Pack IO Registers into IOBs : auto
Equivalent register Removal : YES

---- General Options
Optimization Goal : Speed
Optimization Effort : 1

```
Library Search Order            : ROM.lso
Keep Hierarchy                  : Nü
RTL Output                      : Yes
Global Optimization             : AllClockNets
Read Cores                      : YES
Write Timing Constraints        : NO
Cross Clock Analysis            : Nü
Hierarchy Separator             : /
Bus Delimiter                   : <>
Case Specifier                  : maintain
Slice Utilization Ratio         : 100
BRAM Utilization Ratio          : 100
Verilog 2001                    : YES
Auto BRAM Packing               : NO
Slice Utilization Ratio Delta   : 5

--------------------------------------------------      -----
----------

--------------------------------------------------
----------
*                  HDL Compilation                  *
---------------  --------------------------------      ------

Compiling vhdl file "C:/Xilinx9li/xilinx/myprojects/ROM/ROM.vhd"      in Library work.
Entity <ROM> compiled.
Entity <ROM> (Architecture «Behav») compiled.

-----------------------------------------~---------

*                Design Hierarchy Analysis          *
--------------------------------------------------      ------
```

Analyzing hierarchy for entity <ROM> in library <work> (architecture «Behav»).

==========

* HDL Analysis *

Analyzing Entity <ROM> in library <work> (Architecture <Behav>).
Entity <ROM> analyzed. Unit <ROM> generated.

==========

* HDL Synthesis *

Perforıning bidirectional port resolution...

Synthesizing Unit <ROM>.
Related source file is "C:/Xilinx9li/xilinx/ınyprojects/ROM/ROM. vhd".
Found 32x8-bit ROM for signal <Data_out$roın0000> created at line 92.
Found 8-bit tristate buffer for signal «Data out».
Found 8-bit register for signal <Mtridata_Data_out> created at line 88.
Pound l-bit register for signal <Mtrien_Data_out> created at line 88.
Suınınary:
inferred 1 ROM(s).
inferred 9 D-type flip-flopıs).
inferred 8 Tristate(s).
Unit <ROM> synthesized.

```
==========================------------------------------
==========
HDL Synthesis Report

Macro Statistics
#ROMs                          : 1
 32x8-bit ROM                  : 1
# Registers                    :2
 1-bit register                : 1
 8-bit register                : 1
# Tristates                    : 1
 8-bit tristate buffer         : 1
\

-------------------  -----  ------------ ----  ------
----------

---------------------------------  ---------------  ------
----------
*                Advanced HDL Synthesis                *
-------------------------------------------------------  ------
```

Loading device for application Rf_Device from file '3s200.nph' in environment C:\Xilinx91i.
INFO:Xst:2506 - Unit <ROM> : in order to maximize performance and save block RAM resources, the small ROM <Mrom_Data_out_romOOOO> will be implemented on LUT. If you want to force its implementation on block, use option/constraint rom_style.

```
-----------------------------  -----------------  ----
```

Advanced HDL Synthesis Report

Macro Statistics
#ROMs   : 1

32x8-bit ROM : 1

Registers :9

Flip-Flops :9

==========

==========

* Low Level Synthesis *

==========

INFO:Xst:2261 - The FF/Latch <Mtridata_Data_out_4> in Unit <ROM> is equivalent to the following 3 FFs/Latches, which will be removed : <Mtridata_Data_out_5> <Mtridata_Data_out_6> <Mtridata_Data_out_7>

WARNING:Xst:638 - in unit ROM Conflict on KEEP property on signal Mtridata_Data_out<4> and MtridataDataout-cfi> Mtridata_Data_out<5> signal will be lost.

WARNING:Xst:638 - in unit ROM Conflict on KEEP property on signal Mtridata_Data_out<4> and Mtridata_Data_out<6> Mtridata_Data_out<6> signal will be lost.

WARNING:Xst:638 - in unit ROM Conflict on KEEP property on signal Mtridata_Data_out<4> and Mtridata_Data_out<7> Mtridata_Data_out<7> signal will be lost.

Optimizing unit <ROM> ...

Mapping all equations...

Building and optimizing final netlist ...

Pound area constraint ratio of 100 (+ 5) on block ROM, actual ratio is 0.

FlipFlop Mtridata_Data_out_4 has been replicated 3 time(s) to handle iob=true attribute.

FlipFlop Mtrien_Data_out has been replicated 7 time(s) to handle iob=true attribute.

Final Macro Processing ...

Final Register Report

Macro Statistics

Registers : 16

Flip-Flops : 16

\

---

* Partition Report *

Partition Implementation Status

No Partitions were found in this design.

---

* Final Report *

Final Results

RTL Top Level Output File Name : ROM.ngr

Top Level Output File Name : ROM

Output Format :NGC

Optirnization Goal : Speed

Keep Hierarchy : Nü

Design Statistics

#IOs : 17

Cell Usage:

#BELS : 11

INV : 1

LUT3 :3

LUT4 :3

MUXF5 :3

VCC : 1

FlipFlops/Latches : 16

FDE :8

FDPE :8

Clock Buffers : 1

BUFGP : 1

10 Buffers : 16

IBUF :8

OBUFT :8

Device utilization summary:

Selected Device : 3s200ft256-5

Number of Slices:	4	out of 1920	0%
Number of Slice Flip Flops:	16	out of 3840	0%
Number of 4 input LUTs:	7	out of 3840	0%
Number of IOs:	17		
Number of bonded IOBs:	17	out of 173	9%

IOB Flip Flops: 16

Number of GCLKs: 1 out of 8 12%

Partition Resource Summary:

No Partitions were found in this design.

TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

Clock Signal	Clock buffer(FF name)	Load
Clock	IBUFGP	16

Asynchronous Control Signals Information:

Control Signal	Buffer(FF name)	[Load

Reset IIBUF 18

Timing Summary:

Speed Grade: -5

Minimum period: No patlı found

Minimum input arrival time before clock: 3.839ns

Maximum output required time after clock: 6.427ns

Maximum combinational patlı delay: No patlı found

Timing Detail:

All values displayed in nanoseconds (ns)

Timing constraint: Default OFFSET IN BEFORE for Clock 'Clock'

Total number of patlıs / destination ports: 64 / 32

Offset: 3.839ns (Levels of Logic = 3)

Source: Address-cz> (PAD)

Destination: Mtridata_Data_out_4 (FF)

Destination Clock: Clock rising

Data Patlı: Address-cz> to Mtridata_Data_out_4

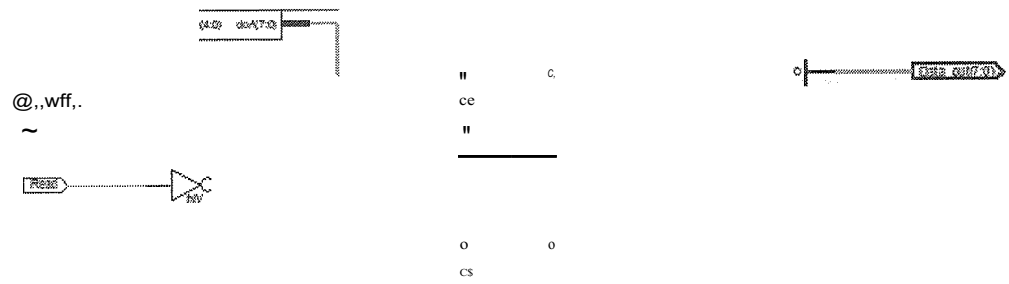
Cell.in-c-out	Gate fanout	Net Delay	Delay	Logical Name (Net Name)
IBUF:I->0	4	0.715	1.074	Address_2_IBUF (Address_2_IBUF)
LUT3:I0->0	2	0.479	0.915	Mrom_Data_out_rom0000411 (N21)
LUT3:Il->0	4	0.479	0.000	Mrom_Data_out_rom000051 (Mrom_Data_out_rom00004)

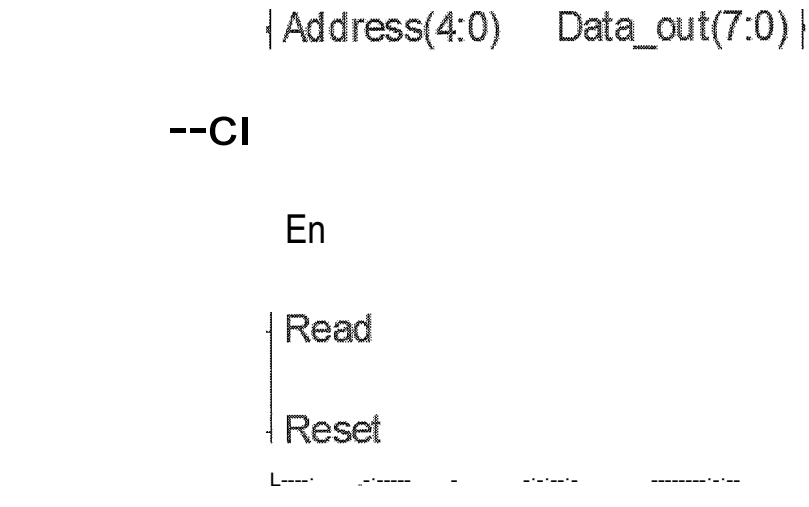
```
    FDE:D                  0.176          Mtridata_Data_out_  4

    Total                  3.839ns (l.849ns  logic,  1.990ns route)
                              (48.2%  logic,  51.8%  route)


==========
Timing constraint: Default OFFSET OUT AFTER for Clock 'Clock'
  Total number of paths / destination ports: 16 / 8

Offset:            6.427ns (Levels of Logic = 1)
  Source:          Mtrien_Data_out_1  (FF)
  Destination:     Data_out<7> (PAD)
  Source Clock:    Clock rising

  Data Patlı: Mtrien_Data_out_1  to Data_out<7>
                        Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)

    FDPE:C->Q             1   0.626   0.681  Mtrien_Data_out_1  (Mtrien_Data_out_O)
    OBUFT:T->0                    5.120         Data_out_7_0BUFT  (Data_out<7>)

    Total                  6.427ns (5.746ns logic, 0.681ns route)
                              (89.4% logic,  10.6% route)

-------------------------------------------------------------
==========
CPU:  8.16 /  10.03 s | Elapsed:  8.00 /  10.00 s

-->

Total memory usage is 140012 kilobytes

Number of errors       0 (   0 filtered)
```

Number of warnings : 3 (0 filtered)

Number of infos : 2 (0 filtered)

9.3 **This is the electronic circuit**





CONCLUSION

My desing Read Only Memory is a storage media used in computers an other electronic devices. Ones its program data stays in the ROM until its electrical erased.

Use of ROM for program storage. Computer requires the ROM to store the initial program that runs when the computer is powered .

I desing is ROM an put in values using VHDL.

We are seeing Read Only Memory the design language is VHDL.Read Only Memory is nonerasable storage in my project, I designed a ROM and we difened two dimensional array of 32 elements and each element 8 bit standart logic vector. We put data in to this ROM by using constant content rom_array at location 0.

REFERENCES

[1] Zwolinski Mark- Digital System Design with VHDL

[2] Yalamanchili Sudhakar- VHDL a Starter's Guide

[3] http://www.vhdl.org

[4] http://www.google.com

[5] http://www.wikipedia.org