

6502 emulator on FPGA

by

Khoo Eng How (1268)

Dissertation submitted in partial fulfilment of
the requirements for the
Bachelor of Engineering (Hons)
(Electrical and Electronic Engineering)

Jan 2004

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

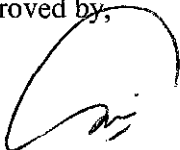
6502 emulator on FPGA

by

Khoo Eng How

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL & ELECTRONICS ENGINEERING)

Approved by,



(Mr. Abu Bakar Sayuti)

UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

May 2004

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.



Khoo Eng How

Abstract

6502 microprocessor was once used in almost all of the microcomputer in the 80s, including the Apple II lines of computer, the Commodore PET, the Commodore 64, the Atari 8-bit series and even on the Nintendo Entertainment System (NES) video game console.

The objective of this project is to emulate the once famous 6502 microprocessor onto a FPGA chip. The FPGA-based 6502 microprocessor had to emulate the functionality of a real 6502 microprocessor. Accurate pinouts emulation is desired but not a must. The 6502 assembly language is easy to learn and building a computer based on this microprocessor requires very few parts, thus making this project a great experiential learning process.

The scope of this project requires the student to have an in-depth understanding on computer system architecture, especially on 6502 architecture; Verilog to understand existing 6502 source code from Bird Computer and also FPGA development process (synthesis tools) to transfer the Verilog code to the FPGA chip.

Thus far, the resources and information on 6502 microprocessor looks promising. The student earlier scope was to come up with the 6502 code in Verilog HDL, but as there is available code from Bird Computer (State Machine coded) so the student had chanced his objectives to understand the existing code and implement it on FPGA only. But as along the way, problems occur on hardware implementation, focus had been switched again to simulate the existing code or ALU or simple processor to build up student understanding and for documentation for future project expansion. To test the functionality of the 6502 system, the student will either find existing application or come up with simple program to run using the FPGA-based 6502 system.

Acknowledgement

I would like to take this opportunity to express my greatest gratitude and thanks to several parties who have facilitated me at one stage or another throughout this project.

My family and friends, for giving me encouragement and bear with me during this rather difficult and time consuming project.

Mr. Abu Bakar Sayuti, my project supervisor, for sparing his time to supervise and guide me throughout the duration of the project.

EE Lab Technician, especially Mr. Musa for providing the necessary assistance during the several stages of the projects.

Universiti Teknologi PETRONAS, for providing me with the necessary foundation, resources and facility to embark this project.

Mr Polur Kir, an undergraduate from India for providing his time and advice on matter pertaining to FPGA designing.

Bird Computer for providing their 6502 source code for education purposes.

Others core designers, for sharing their their work, which allow me to learn more about hardware design.

Not forgetting my other course mate who working on with FPGA for sharing with me their problem faced and solution for it along the process.

Thank you all!

TABLE OF CONTENTS

Certification	i
ABSTRACT	iii
Acknowledgement	iv
CHAPTER 1: INTRODUCTION	1-4
1.1 Background of Study	1
1.2 Problem Statement	2
1.2.1 Problem Identification	2
1.2.2 Significant of the Project	2
1.3 Objectives and Scope of Study	3
1.3.1 Objectives	3
1.3.2 Scope of Study	3-4
CHAPTER 2: LITERATURE REVIEW	5-9
2.1 6502 Microprocessor/Computer Architecture.	5-7
2.2 6502 Machine Language	7-8
2.3 Hardware Description Language	8-16
2.3.1 Verilog vs VHDL	9-10
2.3.2 HDL for Synthesis	10-11
2.3.3 Introduction to Verilog	11-12
2.3.4 Verilog format	13-14
2.3.5 Verilog Data Type – Wire/Reg	14-15
2.3.6 Testbench	15-16
2.4 System-on-chip (SoC)	16
2.4 FPGA Design Stage	17-18
CHAPTER 3: METHODOLOGY/PROJECT WORK.	19-2
3.1 Procedure Identification	19-20

3.2 Tools Required	20
3.3 Project Work	21-23
CHAPTER 4: RESULTS AND DISCUSSION	24-38
4.1 Findings and Discussion	24-38
4.1.1 Verilog Model Examples	24-30
4.1.2 6502 source code	30
4.1.3 Verification	30-32
4.1.4 Arithmetic Logic Unit 74381.	32-35
4.1.5 FPGA Development Board	35
4.1.6 B-3Spartan2+ QuickStart Guide 2.0	35-36
4.1.7 Problem Faced	36-38
4.1.7.1 Verilog HDL	36-37
4.1.7.2 Equipment Condition	37-38
4.1.7.3 Incomplete Implementation	38
CHAPTER 5: CONCLUSION AND RECOMMENDATION	39-41
5.1 Conclusion	39-40
5.2 Recommendation.	40-41
5.2.1 Design Tools	40
5.2.2 Prototyping Tools	40
5.2.3 Proper Support	40-41
5.2.4 Soft Copy Submissions	41
REFERENCES	viii - ix
APPENDIXES	x

LIST OF FIGURES

- Figure 2.1 6502 Block Diagram
- Figure 2.2 Generic structure of a testbench and a design under test
- Figure 2.3 General FPGA Design Stages
- Figure 2.4 Synthesis Process
- Figure 3.1 Verilog Coding Digital Design Flow
- Figure 4.1 Timing Simulation for D type flip-flop
- Figure 4.2 Timing Simulation for D type flip-flop with asynchronous reset
- Figure 4.3 Timing Simulation for D type flip-flop with synchronous reset
- Figure 4.4 Timing Simulation for D type flip-flop with asynchronous reset and clock enable
- Figure 4.5 Timing Simulation for an ALU
- Figure 4.6 Timing Simulation for 74381 ALU

LIST OF TABLE

- Table 4.1 Functionality of 74381 ALU

CHAPTER 1

INTRODUCTION

1.1 Background of Study: 6502 Microprocessor

6502 was once found in almost every personal computer in the late 70's and early 80's including the Apple I, Apple II and Apple III, Commodore Pet and Atari 400 and Atari 800. 6502 microprocessor gained popularity mainly because of its low price. [B2]

6502 microprocessor is an 8 bit processor, this mean that it had an 8 bit data bus. As it instruction set consists of 8 bit operation, so for complex operation such as 16 bits or 32 bits arithmetic and memory transfer can only be performed by sequences of simpler operations. 6502 had a 16 bits address bus, meaning that the address space is only 64K bytes. This limitation was addressed by using memory banks. The original clock speed for 6502 was 1 MHz, but later version comes with better clock speed at 1.2MHz and 1.4MHz.

The 6502 is not really a register oriented microprocessor as its processing power comes from its addressing modes. An addressing mode is a method for generating the address (effective address) for a particular instruction value.

1.2 Problem Statement

1.2.1 Problem Identification

It is almost impossible to find a new 6502 microprocessor on the market now as it had stop its production for some time already. If a customer now wanted to use the 6502 microprocessor, he/she can either found it in the second hand computer shop or needs to order in bulk amount from the manufacturer which is a waste as normally only one or two units is/are needed for experiment or testing purposes. So it makes practical sense to emulate the 6502 microprocessor / system onto FPGA so that the processor can be fabricated as needed.

1.2.2 Significant of the Project

This project will be a great learning experience to learn the architecture and functionality of a microprocessor down to the machine language level. This will be a stepping stone to better understand other microprocessor as 6502 assembly language is relatively easy to learn compared to others microprocessor assembly language. Besides learning about microprocessor, the student will also learn how to program a FPGA chip according the needed specification and also on how to emulate a microprocessor using hardware. The student will also learn and be more proficient in Verilog Hardware Description Language (Verilog HDL) which is important in microprocessor prototyping design.

1.3 Objectives and Scope of Study

1.3.1 Objectives

- To gain a better understanding on microprocessor especially on 6502.
- To study an earlier computer system design.
- To emulate the 6502 microprocessor / system onto a FPGA chip.
- To have a better understanding on machine language, assembly, Verilog and also on FPGA development tools.

1.3.2 Scope of Study

The scope of this projects cover the basic understanding on 6502 microprocessor , its architecture, its operation in assembly language and also its physical layout. As ultimately the student needs to emulate 6502 microprocessor /system onto a FPGA, knowledge on one of the HDL is essential to transfer all function of the 6502 onto FPGA. A lot of time and effort will also be required to study similar works done by computer/electronic enthusiasts around the world on 6502 so that time and effort can be saved from starting this project from the scrap.

As the project progress, a complete 6502 System-on-Chip (SoC) source code (coded in hard coded state machine) in Verilog HDL for evaluation and education purposes is available from Bird Computer [8]. So the scope of the project now is to implement the 6502 SoC with an a dditional FIFO module and m odification o f UART module to utilize the FIFO on FPGA that will be connected to keyboard, mouse (for i nput) and monitor (for o utput) if the time is s till feasible. But the minimum is being able to write test benches to simulate ALU or a simple processor code. Being able to simulate the 6502 processor core to the check its functionality is an added bonus.

To test the real functionality of the 6502 system on FPGA, some application (in 6502 machine language) will be used. But this will be implemented by future student continuing this project on hardware implementation. This matter is discussed further in Chapter 4.

CHAPTER 2

LITERATURE REVIEW AND THEORY

Some literature review had been done on 6502 microprocessor including its architecture and its operation and also FPGA design techniques.

Basically the knowledge that should be picked up in order to finish this project are:

1. 6502 Microprocessor/Architecture
2. 6502 Machine Language
3. Hardware Description Language (HDL)
4. System-on-Chip (SoC)
5. FPGA Design Stage

Each of the above will be further discussed in the following sections.

2.1 6502 Microprocessor/Computer Architecture [6]

In every computer, there are three main parts, a Central Processing Unit (CPU), memory which can be divided into ROM and RAM and input and output devices such as keyboard, monitor, mouse and etc. In a microcomputer, all function of a CPU are contained in a microprocessor unit.

Early CPU had very limited processing power if compare with today's CPU. Earlier CPU like 6502 are only 8 bit processor. Other 8 bit processors are Z80, 8086, 6800 and etc. After that comes more powerful CPU with 16 bit processing power like the 68000 from Motorola. After 16 bit come 32 bit and the current processing power of a CPU in personal computer (PC) is 32 bit but 64 bit CPU are getting more and more common to the end user. 64 bit CPU for PC is now available in the market from both

Intel and AMD (the main rival in microprocessor market).

Although 8 bit CPU is now consider outdated system but it is a good starting point for student to have a better understanding on microprocessor as it is always good to learn from the basic. Although 8 bit CPU cannot be of much usage already in PC but 8 bit CPU is still powerful enough to be the core of a microcontroller to be put in electronic appliances like washing machine, rice cooker and etc. So there are actually still market and usages for 8 bits CPU.

Now let have a look at 6502 block diagram. Below is a block diagram of a 6502 microprocessor.

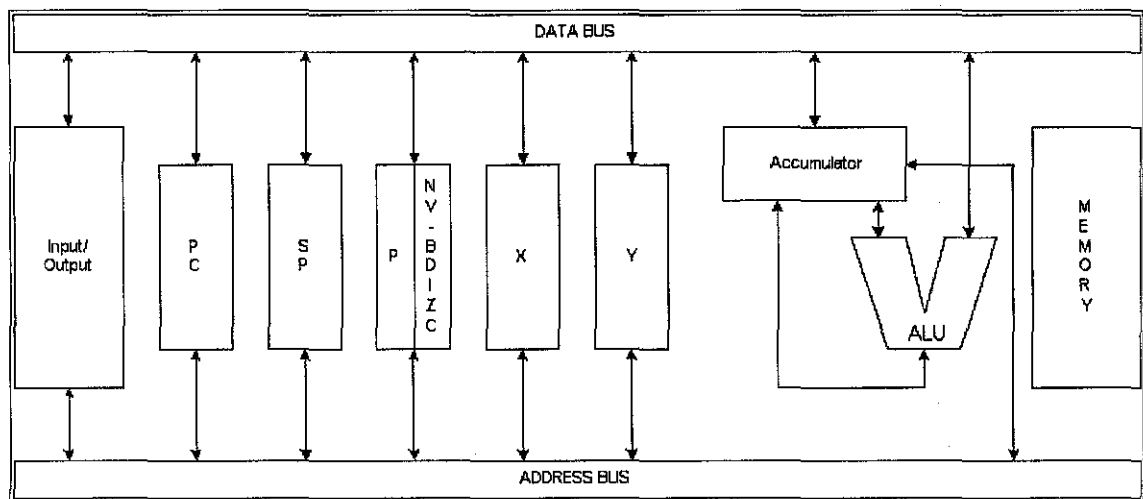


Figure 2.1 6502 Block Diagram [16]

As can be seen from Figure 2.1, the 6502 microprocessor contains seven main parts: an Arithmetic Logical Unit (ALU) and six addressable registers. Data is moved around in side the 6502 chip and between other components in the computer over transmission lines called buses. There are two kinds of buses in the 6502, an 8-bit data bus and a 16-bit address bus. The data bus is used for passing 8-bit data and instruction bytes from one 6502 register to another, and also for passing data and instructions back and forth between 6502 and memory (RAM). The address bus in the

mean time is used to keep track of CPU's 16-bit memory addresses. Memory addresses are the addresses that instructions and data are coming from, and the addresses that instructions and data are being sent to.

6502 have 16-bit address bus, thus the address is in the range of 0000000000000000 to 1111111111111111 in binary or \$0000 to \$FFFF in hex that can be accessed by the processor. Addressable memory is therefore 64 Kb. Addresses are stored by least significant byte first (Little Endian method). A 16 bit address needs to be stored in two consecutive bytes. A little endian processor will store the address \$458D as \$8D followed by the byte \$45.

Memory meanwhile is viewed as a set of 256 byte pages. The first page (\$0000 to \$00FF) is called the 'Zero page', and can be accessed by using a special addressing mode which enables shorter and therefore faster executing instructions. This makes it useful for storing tables of values or addresses that are going to be accessed frequently by your program. The second page (\$0100 to \$01FF) is used to hold the system stack. This is used to keep track of values, especially during subroutine calls. It cannot be moved.

2.2 6502 Machine Language

Machine language is the lowest possible level of programming. It's called machine language because it involves working directly with the computer hardware. Different computers use different hardware and so those differences are reflected in the machine language of each computer. Typically when people say "Machine Language" they are actually referring to Assembly Language. Machine language is just the electrical signals bouncing around on the circuit board of your computer. Because these electrical signals have two possible states, On or Off, we use binary numbers as an abstract representation of these electrical signals. Assembly language is a second level of abstraction from the same electrical signals. [I7]

To enable a person to program in Machine Language, knowledge and understanding on the machine instruction set and addressing mode is a must. For 6502 there are a total of 56 different instructions and 13 different addressing modes. Execution of each instruction will takes 2-7 cycles depending on the types of instructions. The time taken for each instruction can be referred to the table in Appendix E - 6502 Instruction Encoding. The execution of each instruction was always in order one after each other. For 6502, all opcodes are 1 byte in size while the operand may vary from 0 to 2 bytes. Implied, Accumulator, Immediate, Zero Page just to name a few are the different types of 6502's 13 addressing modes. Further information on 6502 Machine Language and its instruction set can be referred to Appendix C, D and E.

2.3 Hardware Description Language (HDL)

One of the major drawbacks of traditional design method is the manual translation of design description into a set of logical equations or a schematic. This step can be eliminated using HDL like Verilog and VHDL.

But what is HDL? HDL actually stands for hardware description language. It is used to describe the logic functionality of a circuit. HDL can also describe the behavioral aspects of a circuit function. It is also used sometime to show the netlist of a circuit. As mention above, there are two types of HDL, Verilog and VHDL (VHSIC HDL – Very High Speed Integrated Circuit Hardware Description Language). Beside Verilog and VHDL, there are actually others type of HDL language which is one of the latest like the C/C++ code but not widely accepted and Superlog which is very new and are still under research.

Before going further to discuss the different between Verilog and VHDL, let's have an understanding first on some background of Verilog and VHDL.

Verilog or Verilog HDL as it is called in full is a HDL developed in the 1984-1985 by Philip Moorby who needed a simple, intuitive and effective way of describing digital circuit for modelling, simulation and analysis purposes. The language later becomes the property of Gateway Design Automation, which was later acquired by Cadence Design Systems. From 1990 Cadence opened the language to the public, which led to the standardisation of the language by the IEEE in 1995. Do take note that Verilog is a registered trademark of Cadence Design Systems, Inc. and all information that the student obtain refers to Verilog HDL as defined by IEEE Standard 1364. [3] Verilog is easy to learn. It has syntax reminiscent of C (with some Pascal syntax thrown in for flavour). About half of commercial HDL work in the U.S. is done in Verilog, making it a compulsory requirement for digital hardware designer.

VHDL in the mean time is a U.S. Department of Defence (DOD), mandated language that is used primarily by defence contractors. Although most of the concepts in VHDL are not different from those in Verilog, VHDL is much harder to learn. It has a rigid and unforgiving syntax strongly influenced by Ada (which is an unpopular conventional programming language that the DOD mandated defence software contractors to use for many years before VHDL was developed). Although there are more academic papers published about VHDL than Verilog, only less than one-half of commercial HDL work in U.S. is done in VHDL as VHDL is more popular in Europe when compare to U.S. [B4 - pg 65]

2.3.1 Verilog vs VHDL

There always argument on which one is a better form of HDL. Actually both had its advantages and disadvantages. There is no clear cut on which is a better HDL. Let's start by looking at some characteristic on Verilog.

Verilog is easier to write, to read and to understand as it is similar to C. It is also easier to learn if compared to VHDL. For situation in Malaysia, all design centers

uses Verilog as their preferred HDL. This situation maybe due to many research and design centers are own by U.S company which prefer the usage of Verilog HDL.

VHDL in the mean while is more complicated and more difficult to learn when compared to Verilog. There are more coding rules to follow. Although there are more rules to follow, VHDL is in fact more flexible when compared to Verilog and it also can reflect the real design more efficiently.

So which one is better? A ctually whichever is more suitable to be used as the standard HDL depends largely on individual designer preference. As both HDL have its advantages and disadvantages. Furthermore most design tools in the market support both Verilog and HDL.

For this project, Verilog HDL was chosen simply because Verilog is used more widely in Malaysia and its similarity to C language.

2.3.2 HDL for Synthesis

A powerful and the most often used method of HDL. HDL for synthesis eases design by allowing the functionality of a circuit to be described. By describing the functionality of a circuit, HDL allows designers to design larger amounts of circuit functionality within a short period of time.

Previously, conventional design was done through schematic capture. With the amount of gates count in the numbers of hundred of thousand, manual drawing of transistors and logic gates and connecting them up becomes unmanageable as human error is unavoidable during the process thus making debugging a big issue. Furthermore large designs with hundred of thousands or even millions of

transistors take too long time to simulate using conventional SPICE type of simulation.

In the mean time, synthesizing of HDL for logic circuits are always misunderstood to be 2X or more complex when compared to conventional schematic capture which is NOT true. The outcome in area and performance of synthesis varies deeply with the coding style and synthesis optimization performed. In fact using different synthesising tools will produce different results in term of performance also. If a design is coded to be in a bad architecture, obviously the synthesized circuit will be huge. But efficient architecture, coding and synthesis can produce a very good area utilization for a design.

There are some important aspects of coding for synthesis that is often over-looked by designer. The designer must understand that coding for synthesis requires a different form of style than just to simulate it. Code that can simulate correctly does not necessarily can be synthesized to the logic that is required. One common mistake that many designers do is writing code that simulates accordingly but may generate “garbage” circuits.

Bad/inefficient coding will synthesize to inefficient circuits that have more logic gate than necessary. This causes larger than required die size for the design and therefore an increase in the cost of the design which is unnecessary. Inefficient coding also increases path timing delay, thereby reducing the performance ability of the synthesized circuit.

2.3.3 Introduction to Verilog

Verilog can be divided into 3 different types. Structural that is in netlist form. Behavioral that describes the behavioral of a circuit, mostly used for analog circuit design. The last type is RTL that describe the functionality of the circuit and is

synthesizable by any available synthesis tool. It is very important to have a good coding style when writing in RTL, so that most optimized synthesized logic can be obtain.

Let's start by having a inside into Behavioral type of Verilog or HDL in that sense. Behavioral HDL describes the behaviour of a "black box" circuit. The "black box" circuit can be any circuit (analog, digital or mixed signal). The advantages of using behavioral HDL is that simulation using HDL is much faster compared to conventional method of SPICE like simulations. It is also easier to integrate a "black box" with other designs. This allows the ease of system level simulation to check a system's functionality. With this, HDL can also be used to describe behaviour of an analog circuit. This would allow the combination of analog circuits and digital circuits in a fullchip level. This type of simulation is fast and accurate.

Structural HDL on the other hand is used as a netlist. It describes the components in a design with the interconnects between them. Commonly used as a netlist of a design being used by different tools (simulation, synthesis, layout), a bridging factor between different design tools.

Then finally is the RTL. RTL stands for register transfer level. This is the synthesizable code which becomes the golden model of a design. There are some design tools in the market that auto generates RTL based on a graphical mode of entry like flowcharts, truth table and state diagram. With good RTL coding style, timing and area can be greatly optimized. It is therefore essential that any new designs coded by inexperienced designer be checked by experienced Verilog designers.

2.3.4 Verilog format

Unlike VHDL that have entity, architecture and configuration, Verilog only have module declaration. The module declaration declares the name of the module being coded and have a list of all the interface signals (input, output or inout). This form of declaration is significantly different from VHDL since in VHDL an ENTITY is required to specify the interface signals.

All Verilog code starts with the keyword “module”. It is used to describe the name of a module together with the interface signals. Let’s have a look at the following example of a module named test with 3 input ports (inputA, inputB, inputC) and 2 output ports (outputA, outputB).

```
module test (inputA, inputB, inputC, outputA, outputB);  
    //enter your Verilog code  
    //more of your Verilog code  
endmodule
```

From the above example, we can see that all Verilog code ends with the keyword “endmodule”. It is the terminating keyword for a module declaration.

Now let’s look at module declaration.

```
module testmodule (inputA, inputB, inputC, outputA0;
```

The module declaration is then followed by the declaration of the direction of the interface signals: input inputA, inputB, inputC; output outputA;

Verilog also allows for bidirectional ports.

```
module testmodule (inputA, inputB, inputC, outputA0;  
    input inputA, inputB;  
    inout inputC;  
    output outputA;  
endmodule
```

Many designers always neglect to put comments in the code. Comments are an important form of documentation on the functionality/objective of the code. The

comment helps make the code readable to other designer. It can also serve as reminder when performing debugging on the code. Single line comments begins with the symbol // while multiple line comments begins with /* and ends with */.

Numbers meanwhile can be represented in Verilog as real numbers, integer number and base numbers (binary, octal, hex, decimal). Let's have a look at the following example.

```
integer A, B, C;  
A=4'b0101; // 4 bit binary  
B=5'o14; // 5 bit octal  
C=8'ha5; // 8 bit hex  
D=5'd14 // 5 bit decimal
```

2.3.5 Verilog Data Type – Wire/Reg

Interface signals are declared as either type reg or wire. Type reg means that it will be able to hold a value in the Verilog code. Type wire means that it will be assigned a value in the Verilog code. However please be noted that using reg does not necessary means that the signal is flopped or latched.

Another way to differentiate reg and wire is to view usage of wire as real physical wire connection between two gates, whereby the contents/value on the wire is consistently updated (continuous). A reg on the other hand can be viewed as a signal being assigned values during certain circuit conditions.

So the question now is, when to use wire and when to use reg? A common method to differentiate these two is the “assign” statement and “always” block. When using “assign” statement, always use “wire” declaration. When using assignment of values in an “always” block, use “reg” declaration.

Let's look at an example on the usage of wire. Using an AND gate as an example.

```

module ANDgate (inputA, inputB, outputA);
input inputA, inputB;
output outputA;
wire outputA;
assign outputA=inputA & inputB;
endmodule

```

From the example, the value of outputA is constantly updated with the value from inputA and inputB. The “&” symbol is used to represent AND Boolean.

Let’s look at an example on the usage of reg using the same AND gate example.

```

module ANDgate (inputA, inputB, outputA);
input inputA, inputB;
output outputA;
reg outputA;

always @(inputA or inputB)
begin
outputA=inputA & inputB;
end
endmodule

```

The code @(inputA or inputB) is known as sensitivity list. This example shows that the “always” block will be evaluated whenever there is a change in the signals listed in the sensitivity list. Sensitivity list is associated with an “always” block. Signals that will cause an evaluation of the “always” block must be included in the sensitivity list.

2.3.6 Testbench

When explaining about Verilog or HDL, we must also discuss about Testbench. So what is a testbench and what is its function? A testbench is an environment which surrounds a DUT (design under test) and forces stimulus into the DUT while monitoring the output ports of the DUT. In other words, a testbench is an

environment to verify the functionality of a design. It will be explain better with the following diagram.

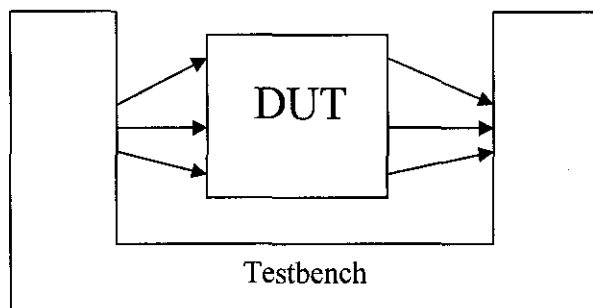


Figure 2.2 Generic structure of a testbench and a design under test

To get a better picture on how a testbench is, please refer to Appendix G for all the Testbenches written for the Verilog Model examples of D flip-flop in Chapter 4.

2.4 System-on-chip (SoC)

Advanced ASIC and FPGA technologies allow us to integrate complex systems on a single chip, embedding standard processor devices, dedicated processing blocks, interfaces to various peripherals, on-chip bus structures in a SOC, or even analog blocks in a mixed-signal device. Moving away from the use of traditional components towards SOC technology will help to satisfy the ever-increasing demands for high processing performance, while reducing mass and power consumption [110].

For 6502SoC, this design come with 6502 processor (6502 CPU) code that are compatible with all the 6502 instruction. It also includes a memory module with RAM and ROM, Universal Asynchronous Receiver Transmitter (UART) to connect the 6502 CPU with its Input/Output, and Video module to display simple text line. Thus instead of having just a 6502 core, now it is possible to have an Apple II-CPU within one FPGA chip. This design is referred to the code available from Bird Computer.

2.5 FPGA Design Stage

The general FPGA Design Stages is shown at Figure 2.2 below:

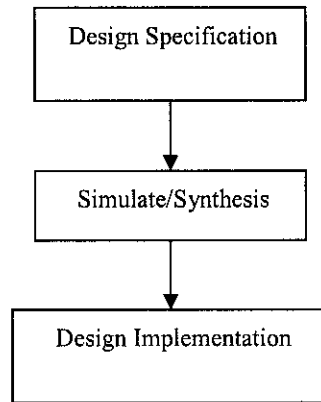


Figure 2.3: General FPGA Design Stages.

From Figure 2.2 above, the first step for FPGA Design Stage is design specification. Design specification had to be done in HDL, either in Verilog HDL or VHDL which is the more popular HDL used. The designer needs to be able to transfer their design specification to HDL. For some examples on Verilog HDL coding, please feel free to have a look at Chapter 4. There are Verilog Model Examples [19] for references or just to have a feel on how the coding is being done. For beginner it is always better to learn and see others implement the model or project.

After writing the code in HDL it is time now to check the functionality of the written code. One important thing to remember is that, what ever result in simulation that showed the design is functioning can only be proved or make sure when the deign is downloaded to FPGA. From experience, normally what working in simulation will not and normally would not be 100% working in real hardware implementation. Furthermore, the same code that can be simulated maybe not necessary can be synthesis without error. There are certain rules to follow when writing the code to ensure the code is synthesisable. For more information please refer to the HELP file in Project Navigator. For the simulation of codes, a test bench is needed to provide the

necessary input or stimulus to the code. The output of the code will then be showed as waveform in Waveform Editor from Aldec Active-HDL.

After the design has been successfully analyzed, the next step is to translate the design into gates and optimize it for the target architecture. This is the synthesis phase. XST (Xilinx Synthesis Technology) is a Xilinx tool that synthesizes HDL designs to create an NGC file. An XST flow project can contain *either* VHDL (XST VHDL) *or* Verilog (XST Verilog) modules, but not a mix of both. A functional VHDL model (XST VHDL) or Verilog model (XST Verilog) is created for schematics prior to synthesis. Process properties can be set to control XST synthesis. Actually after the code is synthesis, it is a good practice to simulate the code again to check for its functionality and finally timing diagram. Shown below is a diagram on synthesis [I9]

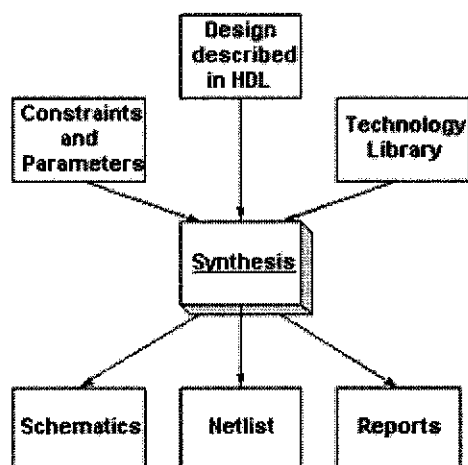


Figure 2.4 Synthesis process

The final stage is to implement the design on FPGA or sometime called downloading the design. This is accomplished by using iMPACT tool from Project Navigator. The steps needed to take will be shown in Chapter 4 and Appendix H.

CHAPTER 3

METHODOLOGY/PROJECT WORK

3.1 Procedure Identification

The project is being carried in 4 stages. The first stage is literature review and background study on 6502 microprocessor and its architecture. The second stage starts with understanding of Verilog HDL. This is then followed by familiarising with simple Verilog code or module and understanding others works on microprocessor. The third stage starts with understanding on 6502 code from Bird Computer and understanding on a simple processor. Work is being done on writing Testbenches on ALU and simple processor to demonstrate student understanding on the Verilog code. The fourth stage is to familiarise with FPGA development board and implement the design to FPGA. Unfortunately the final steps can only be implemented by future student as time had run out.

The following steps need to be taken as parts of the route to complete this project.

- Literature review of 6502 microprocessor, computer architecture, HDL (either VHDL or Verilog).
- Fully understand 6502 architecture and its instructions in assembly language.
- Understand how to code in either VHDL or Verilog.
- Understand the works of other computer/enthusiastic on 6502 microprocessor or other microprocessor, especially on the coding.
- Learn to write test bench.
- Simulate simple module like ALU.
- Understand the 6502 code from Bird Computer.
- Start to familiarise with FGPA development tools.
- Search for 6502 application.

- Search for information on Spartan-II and Virtex-II FPGA board.
- Download design to FGPA chip.
- Try out the FGPA chip using existing application or own program.
- Simulate the 6502 CPU code.
- Add in FIFO module for UART.

3.2 Tools Required

The tools required are:

- Simulation: Active-HDL, ModelSim XE
- Synthesising: Project Navigator (XST Verilog HDL)
- FPGA Board: Spartan-II Prototyping Board

Active-HDL software is needed to enable coding and examination of existing Verilog code. The said software can be obtained free for evaluation for 20 days from the company website or from the Computer System Research laboratory. All simulations are being implemented using this software.

Xilinx Integrated Software Environment (ISE) 6.1i Project Navigator software is needed to synthesis and simulates (ModelSim XE) the available code. ModelSim XE is a simulation tool that comes along with Project Navigator. As the university do not purchase the license for ModelSim, the student can only rely on Aldec Active-HDL software in the laboratory for simulation.

To download the design to FPGA, there are iMPACT from project Navigator that can be used. Steps on how to download a design to Spartan-II prototyping board is included in Chapter 4 and Appendix H.

6502 compiler is also needed create or modify existing 6502 application to test the functionality of the FPGA-based 6502.

3.3 Project Work

The design flow for this project is as below:

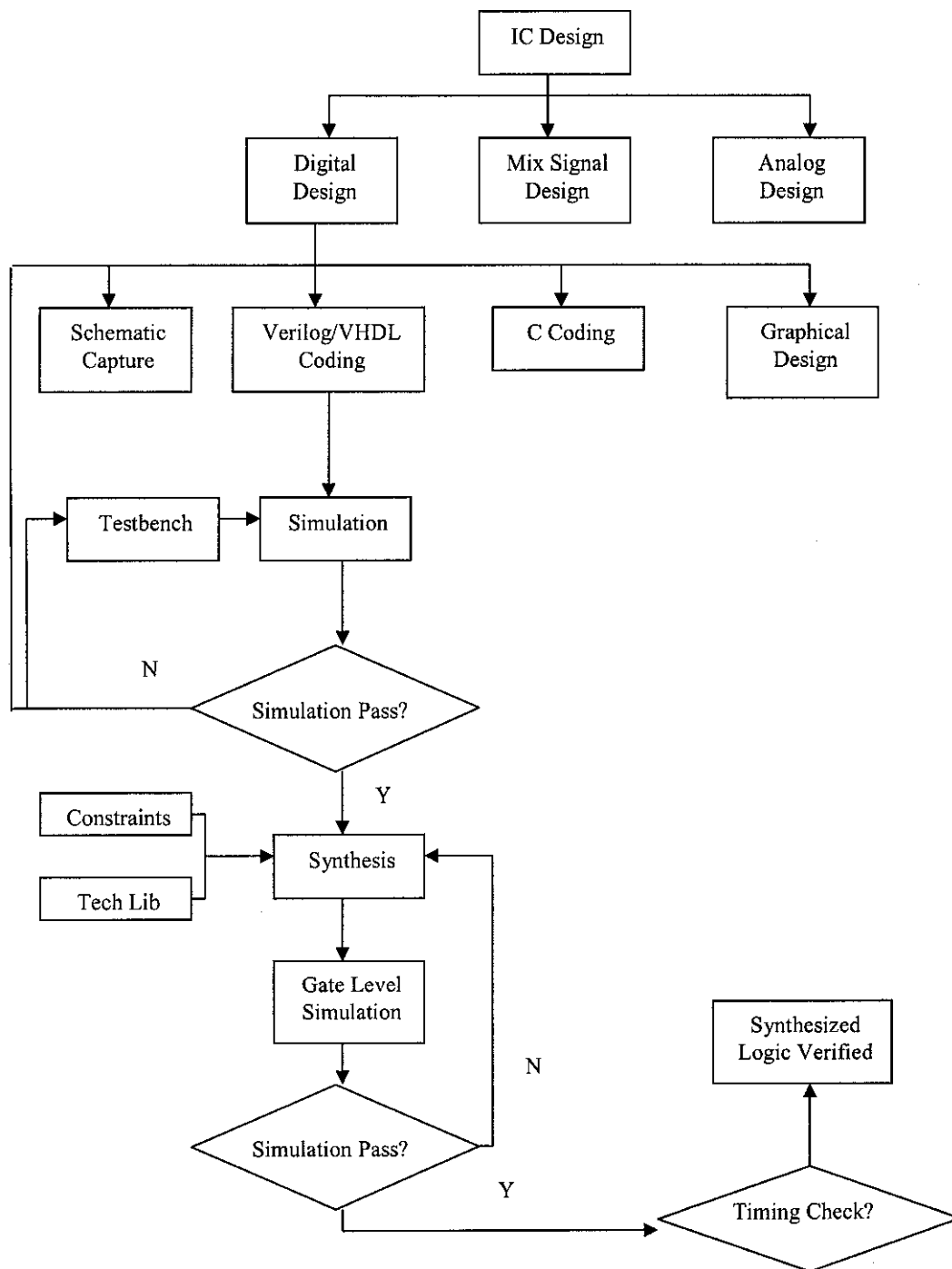


Figure 3.1 Verilog Coding Digital Design Flow

From Figure 3.1, in IC Digital Design, for coding using Verilog/VHDL, this is the path needed to be taken in order to have a design successfully implemented onto FPGA. After the code have been completed or for this project obtained from other source, the code needed to be simulated to check its functionality by providing stimulus thru testbench. If the code didn't pass the simulation, then either the code needs modification or the testbench needs to be redefined to suit the code. After the code had pass simulation then only it will be synthesized. Constraints and Tech Lib will be included for synthesis. After synthesising, then the synthesized code needs to be simulated again, but this time at gate level. After gate level simulation, the code also need to be check for its timing characteristic to make sure that the propagation delay in any path will not jeopardise the output results. Only after all this steps that we can say the code logically verified and are ready to be transfer to hardware implementation.

For this project, the student can only go until first stage of simulation, which is to check the functionality of the code using Active-HDL software. For gate level simulation, Active-HDL software is not suitable and instead ModelSim Xe should be use. But as the license in the laboratory for ModelSim is not complete so gate level simulation is not possible at this stage.

Meanwhile in the process of code writing, to have a better understanding on Verilog HDL tutorial on Verilog was being taken. To better understand on how a processor code in Verilog was being implemented, the code written for 6502 microprocessor by Bird Computer using hardcoded state machine (better optimisation) was being studied. But before this, a simple processor code was being studied from the reference book titled "Fundamentals of Digital Logic with Verilog Design". The ultimate goal is to fully understand the available 6502 CPU code, simulate the code to get its functionality and timing diagram and finally implement it on FPGA to make it a working 6502 SoC on FPGA. But the minimum is to simulate simple ALU code and the simple processor code using ModelSimXE or Aldec Active-HDL to demonstrate

student understanding on writing test benches. It had to be understood that before a test bench can be written, comprehensive understanding on the code written is a must in order to produce the correct timing simulation.

Knowledge on Xilinx Integrated Software Environment (ISE) 6.1i software especially on Project Navigator was very important as this is the only software available in the laboratory that can be used for both simulation (with ModelSimXe) and synthesising (XST) for Verilog HDL as the license for Active-HDL in the lab only enable simulation but not synthesis. This program will be used also to modify the existing UART module so that the FIFO module created using CORE GEN can be utilised.

After synthesising it is time to transfer the design to FPGA using iMPACT from Project Navigator. During these processes, experience on some troubleshooting technique needed to make sure the design really working will be acquired. In this project, the ultimate goal is to integration of some inputs and outputs to the FPGA core to test the functionality of the 6502 SoC implemented. Beside original ,modify 6502 application can also be used to be run on the FPGA-chip to ensure the chip is working as a 6502 processor based system.

Before implementing the design on FPGA, research had to be done on the available FPGA development board in the laboratory, like Spartan-II and Virtex-II to determine which board is a better choice. For this project, Spartan-II board will be used as the added I/O add-on board is an added advantage for future expansion purposed beside the available code is said being successfully implemented on this board.

As a lot of experience will be learned throughout this project, it is therefore very important that proper documentation is being done to enable future expansion of this project become easier and also served as a guideline for future student doing on FPGA to have a know how on the FPGA development stages, the possible problem faced and ways to solve it.

CHAPTER 4

RESULTS AND DISCUSSION

The main progress or work done for this project was on picking up the knowledge on Verilog HDL, understanding existing Verilog HDL code and implement simple circuit coding in Verilog and writing Testbenches to simulate all these codes.

So in this section, a lot of Verilog codes with their timing simulation result will be display to show student understanding on Verilog HDL and techniques on writing Testbenches for the codes.

Progress also had been made on documenting down the procedure needed to download the design to FPGA especially on Spartan-II development board.

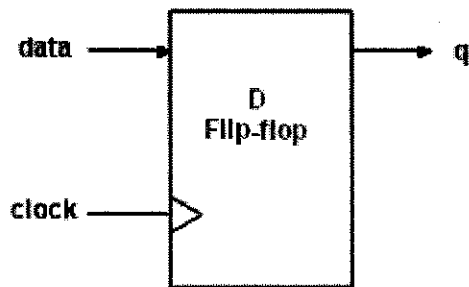
In this section, the student will also discuss on the problems faced during the duration of this project and ways and recommendation to overcome all arising issues. This will definitely be very useful for future student who wish to continue this project until implementation on FPGA.

4.1 Findings and Discussion

4.1.1 Verilog Model Examples [I9]

Below are different types of D flip-flop with its Verilog code and timing simulation of each code to show that different condition and specification set to the D flip-flop and the different it will make to the code to represent the circuit and its simulation result.

D Type Flip-flop:



A sample code is shown below.

```

module dff (data, clock, q);
  // port list
  input data, clock;
  output q;

  // reg / wire declaration for outputs / inouts
  reg q;

  // logic begins here
  always @(posedge clock)
    q <= data;
endmodule

```

The timing simulation of the above code is as shown below.

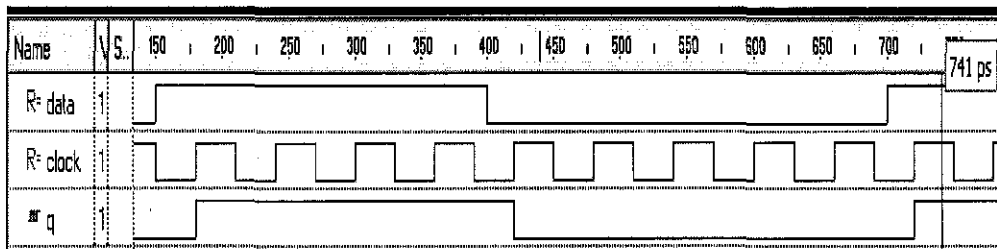
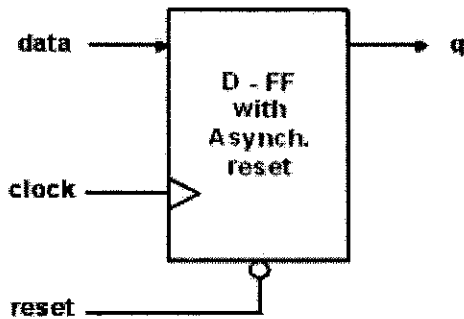


Figure 4.1 Timing Simulation for D type flip-flop

From the timing simulation we can know that the data input is only loaded to output q at the starting edge of positive clock cycle as indicated in the code.

D Type Flip-flop with asynchronous reset:



Example of D type Flip flop with asynchronous reset.

```

module dff_async (data, clock, reset, q);
  // port list
  input data, clock, reset;
  output q;

  // reg / wire declaration for outputs / inouts
  reg q;

  // reg / wire declaration for internal signals

  // logic begins here
  always @(posedge clock or negedge reset)
    if(reset == 1'b0)
      q <= 1'b0;
    else
      q <= data;
endmodule

```

The timing simulation of the above code is as shown below.

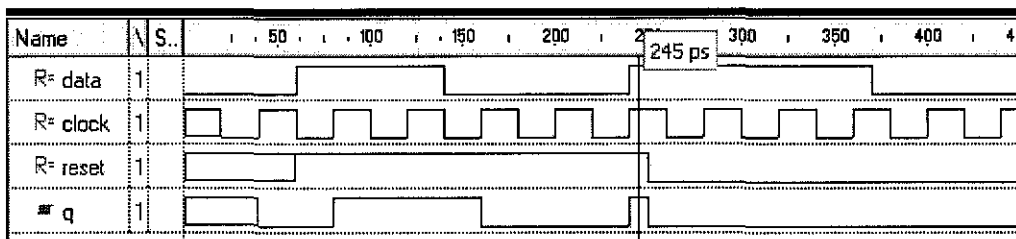
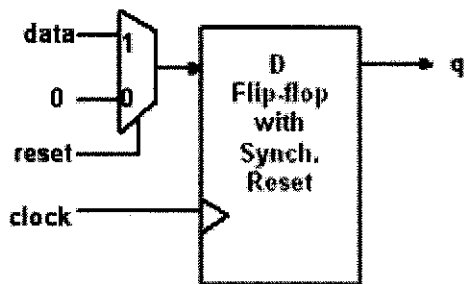


Figure 4.2 Timing Simulation for D type flip-flop with asynchronous reset

From the timing simulation in Figure 4.2, we can justify that the code is functioning as plan. With reset input, we can now set the output q to 0 by setting reset=0 regardless of the data input. From the simulation, after the reset input had been set to 0, the output q will only change to 0 at the falling edge of reset clock set by the always @(negedge reset) thus making this asynchronous reset as it need not wait for next positive clock cycle to change its output.

D Type Flip-flop with Synchronous reset:



Example of D type Flip flop with synchronous reset.

```

module dff_sync (data, clock, reset, q);
    // port list
    input  data, clock, reset;
    output q;

    // reg / wire declaration for outputs / inouts
    reg  q;

    // reg / wire declaration for internal signals

    // logic begins here
    always @(posedge clock)
        if(reset == 1'b0)
            q <= 1'b0;
        else
            q <= data;
endmodule

```

The timing simulation of the above code is as shown below.

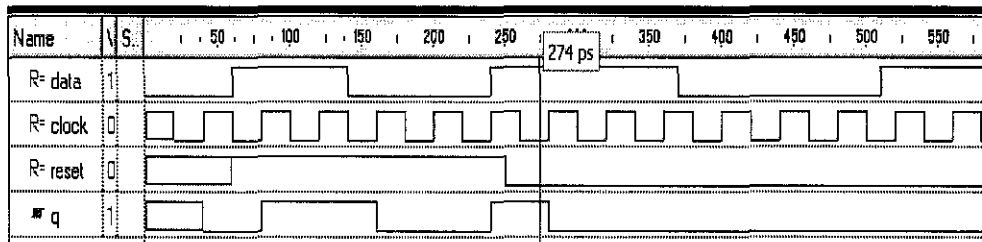
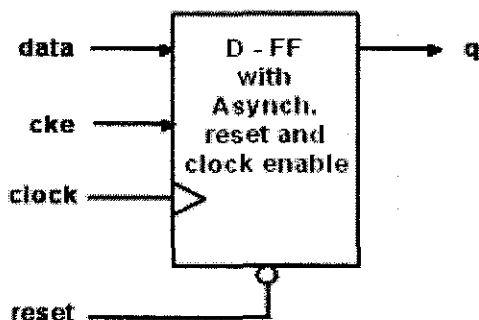


Figure 4.3 Timing Simulation for D type flip-flop with synchronous reset

The different between Timing Simulation in Figure 4.2 and Figure 4.3 shown the different in asynchronous and synchronous reset. In Figure 4.2, we can notice that the output q change to 0 when reset is set to 0. But with synchronous design, after reset had been set to 0, the output will only change to 0 at the next positive clock edge.

D Type Flip-flop with asynchronous reset and clock enable



Example of D type Flip flop with asynchronous reset and clock enable.

```

module dff_cke (data, clock, reset, cke, q);
  // port list
  input data, clock, reset, cke;
  output q;

  // reg / wire declaration for outputs / inouts
  reg q;

```

```

// logic begins here
always @(posedge clock or negedge reset)
  if (reset == 0)
    q <= 1'b0;
  else if (cke == 1'b1)
    q <= data;
endmodule

```

The timing simulation of the above code is as shown below.

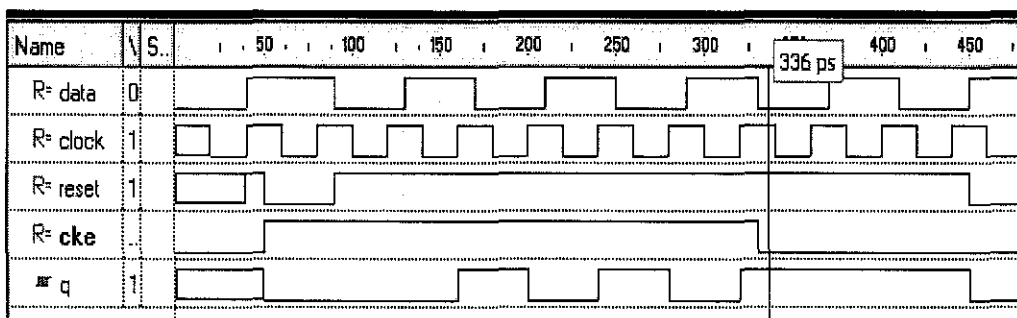


Figure 4.4 Timing Simulation for D type flip-flop with asynchronous reset and clock enable

From Figure 4.4, we can know that when the reset input is set to 0, the output q will also be set to 0. This is true as it had been set in the code to check for reset input first before checking for clock enable input. From simulation, we can see that only when reset is set to 1 that the clock enable input, cke will be of useful. If we notice clearly, from the timing simulation, we can see that the output q did not change its value when the data value change (when reset=1 and cke=1), this due to the condition set that the output will only change its value at the next positive clock cycle. When the cke input is set to 0, the output q value will not change and retain the value it obtains before cke is set to 0.

From the above D flip-flop Verilog model examples, it clearly shown that the understanding of the component functionality is very important. Without this knowledge, the code written will not represent the real function of the component. But for beginner, it is always recommended to learn from existing code to understand how others implement the model or module. The testbenches written

for all the above Verilog code which generate the timing simulation waveform will be attached together in the Appendix G for reference.

4.1.2 6502 source code

The complete 6502 compatible core source code was obtained from Bird Computer. The current version available is said to contain bugs and is provided free for evaluation and education purposes. This available code does not support undocumented instruction but this is acceptable as only one 6502 family of processor supported this.

As at this stage, the scope of this project had change to focus on simulation and not synthesising, the full code for this 6502 SoC from Bird Computer will not be attached along with this report as it takes up too many pages. The code can be obtained from the internet at www.birdcomputer.ca/index.html.

Some modification still needed to be done on the available code as a FIFO module is missing from the files accompanied. To synthesis and download the design to FPGA for a working 6502 SoC, the UART module needed to be slightly modified to utilise the FIFO module created using CORE GEN. But as time is running out, this will be put as future plan only.

4.1.3 Verification [B15]

There are general confusion between the term verification and testbench. Verification is not a testbench, nor is it a series of testbenches. Verification is a process used to demonstrate the functional correctness of a design.

The term “testbench”, in VHDL and Verilog, usually refers to the code used to create a pre-determined input sequence to a design so that the response of the code

can be observed. It is commonly implemented using either VHDL or Verilog, but may also include external data files or C routines.

The testbench provides inputs to the design (code) and monitors any expected outputs. The verification challenge is to determine what input patterns to supply to the design and what is the expected output of a properly working design to check the functionality of the HDL models.

Simulation of design had to be done before implementing it to prevent unnecessary troubleshooting. Simulators are the most common and familiar verification tools used. They are known as simulators as their role is only limited to approximating the reality. A simulation should never been the final goal of a project. Although for this project, the student will stop at simulation, but this didn't mean the project will stop at this stage as time is running out to continue the project until hardware implementation. Simulation lets the designers to interact with the design and correct the flaws if any before implementation on FPGA.

The testbench needs to provide a representation of the inputs observed by the design so that the simulator can emulate the design's responses based on its model description. But one thing need to remember is that simulators have no idea or knowledge on the functionality of the design. The simulator will not know if a design is being simulated correctly or not. Correctness is a value judgement on the outcome of a simulation that must be made by the designer. So it is very important for the designer to know how the design should be functioning, if not the design will not match the desire specification.

The most common verification tools used together with simulators are waveform viewers. Waveform viewers visualize the transitions of multiple signals over time and their relationship with each other transition. With such tool, you can zoom in and out over particular time sequences, measure time differences between tow

transitions, or display a collection of bits as bit strings, hexadecimal or as symbols values. The waveform showing the timing simulation of an ALU is shown in Figure 4.5 below.

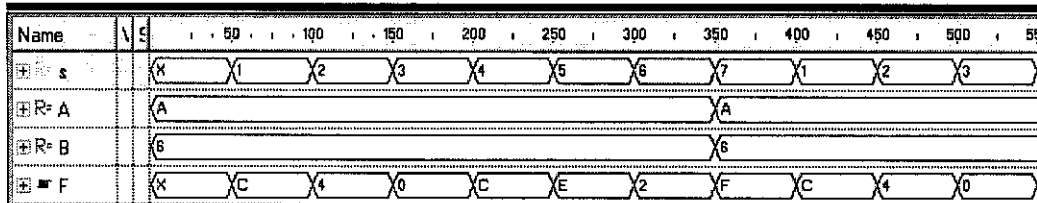


Figure 4.5 Timing Simulation for an ALU

With a viewer, a designer can inspect the output from the simulator to make sure that the code is behaving as expected. The timing diagram on Figure 4.5 is only a simple timing simulation of an ALU.

4.1.4 Arithmetic Logic Unit (ALU) 74381

An ALU is a logic circuit that performs various Boolean and arithmetic operations. One example of ALU chip is the 74381. Table 4.1 below specifies the functionality of this chip. It has 2 four-bit data inputs, A and B, a three-bit select input, S, and a four-bit output, F. As the table shows, F is defined by various arithmetic or Boolean operations on the inputs A and B. Each Boolean operation is done in a bitwise fashion. For example, $F = A \text{ AND } B$ produces the four-bit result $f_0 = a_0b_0$, $f_1 = a_1b_1$, $f_2 = a_2b_2$ and $f_3 = a_3b_3$.

Operation	Input			Output F
	S ₂	S ₁	S ₀	
Clear	0	0	0	0000
B - A	0	0	1	B - A
A - B	0	1	0	A - B
ADD	0	1	1	A + B
XOR	1	0	0	A XOR B
OR	1	0	1	A OR B
AND	1	1	0	A AND B
Preset	1	1	1	1111

Table 4.1 Functionality of 74381 ALU

The Verilog representation of this ALU is showed in below. It can be seen that this was rather an easy code to understand.

```

module alu (s,A,B,F);
  input [2:0] s;
  input [3:0] A,B;
  output [3:0] F;
  reg [3:0] F;

  always @(s or A or B)
    case (s)
      0:F=4'b0000;
      1:F=B-A;
      2:F=A-B;
      3:F=A+B;
      4:F=A^B;
      5:F=A|B;
      6:F=A&B;
      7:F=4'b1111;
    endcase

endmodule

```

The testbench to produce the timing simulation in Figure 4.6 is as shown below. It can be noticed that the testbench is trying to supplying the necessary input to the ALU so that the timing simulation of the waveform from the input and output can be compare. From the result obtained, it is certain that the code is functioning as expected. For example for the select input equal 2, the result from the timing simulation showing the output F=4 which is true for A (equal 10 in decimal) minus

6, the answer is 4. For the same select input equal 2, when A=B (equal 11 in decimal) and B=5, the output F=5 which is true also as $11-5=6$. So from here we can prove that the ALU code is performing as expected.

```
//-----  
//  
// Title      : alu_tb  
// Design     : ALU  
// Author     : KEH  
// Company    : UTP  
//-----  
//  
// File       : alu_TB.v  
// From       : alu_TB_settings.txt  
// By         : tb_verilog.pl ver. ver 1.2s  
//-----  
//  
// Description : Testbench for ALU  
//  
//-----  
  
`timescale 1ns / 1ns  
module alu_tb;  
  
//Internal signals declarations:  
reg [2:0]s;  
reg [3:0]A;  
reg [3:0]B;  
wire [3:0]F;  
  
// Unit Under Test port map  
alu UUT (  
    .s(s),  
    .A(A),  
    .B(B),  
    .F(F));  
  
always  
begin  
  
#50 s='b000;  
#50 s='b001;  
#50 s='b010;  
#50 s='b011;  
#50 s='b100;  
#50 s='b101;  
#50 s='b110;  
#50 s='b111;  
  
end
```

```

initial
begin
#450 A='b1010; B='b0110;
#450 A='b1011; B='b0101;
end
endmodule

```

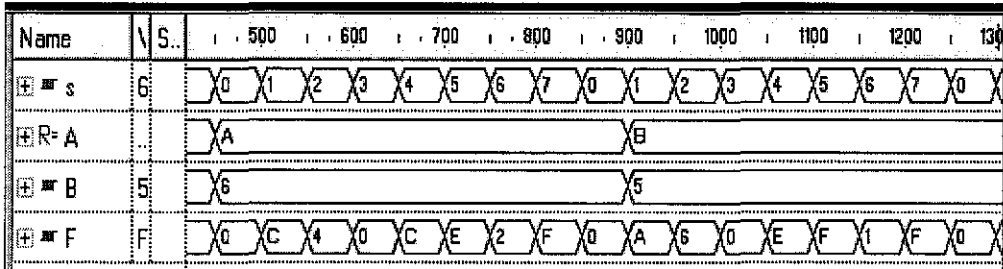


Figure 4.6 Timing Simulation for 74381 ALU

4.1.5 FPGA Development Board

There are currently two types of FPGA board in the laboratory. Spartan-II Prototyping Board and Virtex-II XC2V40/XC2V1000 Reference Board. After much consideration on the pros and cons of both boards, preference will be given to Spartan-II board as there is add-on board available like FPGA-CPU-IO board which will be very useful when expanding this project at later stage. Furthermore the code from Bird Computer had been claimed implemented on Spartan-II. So implementing this project on Spartan-II board will be more secured for successful implementation. For more information on Spartan-II FPGA family, do refer to Appendix F.

4.1.6 B3-SPARTAN2+ QuickStart Guide 2.0 [11,12]

For the purposes of documentation on how the FPGA designing process should be done, or better known as Best Known Method (BKM), a QuickStart guide [12] will be provided on how to use B3-Spartan2+ (200K and 300K) FPGA board. The used

guide available for B3-Spartan2+ board is absolute and need some modification especially on the Downloading of design section.

This quickstart guide takes you through the steps for creating and compiling a new project with the Xilinx WebPACK ISE Design software. This can also be a guide to make sure the Spartan-II board used is a working board. Shown in this part of report is just an overview of all the steps. For full documentation, please refer to Appendix H

The steps are as below:

1. Install the WebPACK_61_fcfull_i software
2. Create a new project
3. Assign the pinouts of the device
4. Create the bitfile (.rbt) for downloading
5. Download the .rbt file to your FPGA

If the LEDFLASH.rbt example file has been downloaded, the LED on the BED-SPARTAN2+ board will be flashing at a rate of about 1.4Hz.

4.1.7 Problem faced

4.1.7.1 Verilog HDL

Verilog HDL was chosen to be used in this project as it is the most commonly used language in the hardware designing world. Learning Verilog is not as easy as it may seem to understand it. For beginner who had no previous

experience on HDL, it is always recommended to start by understanding others work.

Understanding a simple logic operation like flip-flop or even ALU is relatively easy to understand the code written. But for an 8 bit 6502 processor, which had 13 addressing mode and 56 different instruction sets; this is a huge project for an amateur. Although some may argue that this is relatively a “simple” processor by today standard, but for someone who just started to learn Verilog, this is not a simple task all together.

So the student will instead of starting on the real work, will try to understand the functionality of a simple processor to have a feel of how the control signal and buses is being implemented in a processor. The student believes that by starting on something simpler, the understanding on the subject will be more. With the limited time, the student may just being able to demonstrate its ability to understand a simple processor code only by producing the relevant timing simulation.

4.1.7.2 Equipment Condition

All the equipment at the laboratory initially is not ready when the student needs to use it. Precious time has been wasted to set up the equipment. Time again was wasted when all the operating system had been upgraded to Windows XP system, the initial program used for downloading to the Spartan2 FPGA board encounter problem in the new environment. After some research on the internet and from the supplier, only did the student knew that the existing program used for download is now being obsolete and being replace by the iMPACT from Project Navigator.

There are also no proper user manuals on the Spartan2 Prototyping Board. The only manuals available are the circuit diagram of the board which is not much use to understand and the know how on how to use the board. Downloading the design into FPGA is one thing, but to test the programmed FPGA is another issue as the student will need to build its own hardware interface to the FPGA. This all will need to consume times and the student initially oversee this issue in the project planning stage as wrong impression had been given to the student believing all things are there to be utilize.

4.1.7.3 Incomplete Implementation

Due to constraint on resources and times, the student cannot implement the design to FPGA. Implementation on FPGA without proper hardware interfacing is like programmed a said working chip which cannot be verified. A logic analyzer can be used to monitor simple output from a chip, but with a processor, it is no longer a feasible equipment for testing already.

So the student will only implement the timing simulation of a simple processor to demonstrate student understand on the code and if possible produced the timing simulation on the existing 6502 CPU code. By proper documentation on a simple processor in Verilog code, the student hope future student continuing this project will have a better understanding on processor to start with and hopefully can complete this project within 2 semester time.

CHAPTER 5

CONCLUSSION AND RECOMMENDATION

5.1 Conclusion

In conclusion, this project looks promising with a lot of learning curve during the process. A lot of skills and experience will be picked up along the way in order to complete the project. After the completion of this project, the student will had a better standing on a microprocessor construction, its assembly language and also on how to emulate a microprocessor using a FPGA. On the way, the student knowledge on Verilog HDL language will also be improved. This will be a good project to undergone to learn about microprocessor as 6502 architecture and its assembly language as it was simpler and easier to learn.

During the project planning stage and along the way the project was being carried out, the student had actually did some mistake by making assumption that the facility in the laboratory is in good condition and functioning and will not encounter any unforeseen issue like downloading issue and no support from the supplier. The student should have taken this all into account when making decision on the project planning. With all these obstacles, the student can only do code functionality simulation as the code obtain from Bird Computer were not complete with one file missing which prevent it from being able to synthesis. The student really got not enough time to complete this project within the given timeline. The student can only provide the available 6502 processor code but not the whole 6502 SoC, which is not feasible to download the code to FPGA as there are no interfaces to test the functionality of the programmed FPGA. The student will only provide the necessary background study and proper way for future student to continue this project like how to do testbenches to test the functionality of the code. The student hopes with proper documentation on all works

done, it will make it easier for future student to continue this project and complete the initial goal of emulating a 6502 on FPGA.

5.2 Recommendation

5.2.1 Design Tools

The student would like to recommend that UTP obtain some proper design tools as soon as possible. The student would like to suggest UTP to obtain these tools from Open Source instead. Their advantages are first of foremost, it's free of charge and since it is Open Source, there will be more help and support from others designer worldwide that the student can source help from when faced with problems. UTP should also consider switching from windows based system to Linux or UNIX system as these systems are used widely in the industry. UTP management should also make sure that all licenses are obtained and training is provided to both staffs and student.

5.2.2 Prototyping Tools

The student would like to recommend to UTP to purchase better prototyping kits as most of our available prototyping kits are consider outdated and are of the low end type. The cost for better prototyping board is of course higher but they also have better support and easier to used and can get reference or assistance from the supplier. The FPGA prototype board should be purchased with accompanying I/O devices to allow integration and communication with external devices if there are none on the prototyping board like Spartan2 prototyping board.

5.2.3 Proper Support

The student would also like to suggest that more staff should be employ to take care of the laboratory and the management should give the trust to student to come to the

lab on weekend also as prototyping board is not cheap and most of the time the board is needed for testing and debugging of the written code. Without this, time on weekend will be wasted with no progress on the project work.

Proper training should also be given to the personal taking care of the laboratory as the personal now got no training on the design tools like Aldec Active-HDL or Project Navigator. Training should be given to all student involved if possible to make the student had a better start on the project. Equipment training also should be conducted on a regular basic to enable the staff and student to know how to use the available prototyping board in the laboratory. As far as student understand, there are some boards which until today had not been given demonstration by the supplier!

5.2.4 Soft Copy Submissions

The student would also suggest that the FYP submission of reports be done in PDF or DOC format instead of the present hard copy submission especially on logbook reports. So many papers are being generated. If UTP can move toward paperless submission, UTP can help to save the environment and also the hassle of submission. With the well establish UTP network, the management can establish a local server dedicated for FYP submission and distribution of FYP reports.

REFERENCES

Books / User Guide

1. 6502 Software Design by Leo SoanLon, Blackburg, 1980.
2. CS301 Lecture Notes by Glenn G. Chappell, U. of Alaska Fairbanks, Dec 2001.
3. Enhanced Verilog tutorial with Application by EVITA TM
4. Verilog Digital Computer—Algorithms to Hardware, Prentice Hall, by Mark Gordon Arnold, University of Wyoming, 1999.
5. VHDL Reference Manual, 096-0400-003, by Synario Design Automation, March 1997.
6. Verilog Digital System Design, Mc Graw Hill, by Zainalabedin Navabi, 1999.
7. Computer Design and Architecture, Marcel Dekker, by Sajjan G. Shiva, 2000.
8. Commodore 64 Programmer Reference Manual.
9. Spartan-II Demo Board User's Guide, Insight MEMEC, Version 1.1 Jan 2001.
10. Virtex-II XC2V40/XC2V1000 Reference Board User's Guide, Insight MEMEC, Version 1.1, July 2001.
11. B3-SPARTAN2+ Quickstart Guide 1.0
12. B5e-Super-Value-Pack Tutorial Updated 7 August 2002
13. Spartan-II 2.5V FPGA Family: Complete Data Sheet DS001 September 3, 2003
14. Fundamental of Digital Logic with Verilog Design, Mc Graw Hill by Stephen Brown and Zvonko Vranesic, 2003.
15. Writing Testbenches – Functional Verification of HDL Models, Kluwer Academic Publishers by Janick Bergeron, 2000.

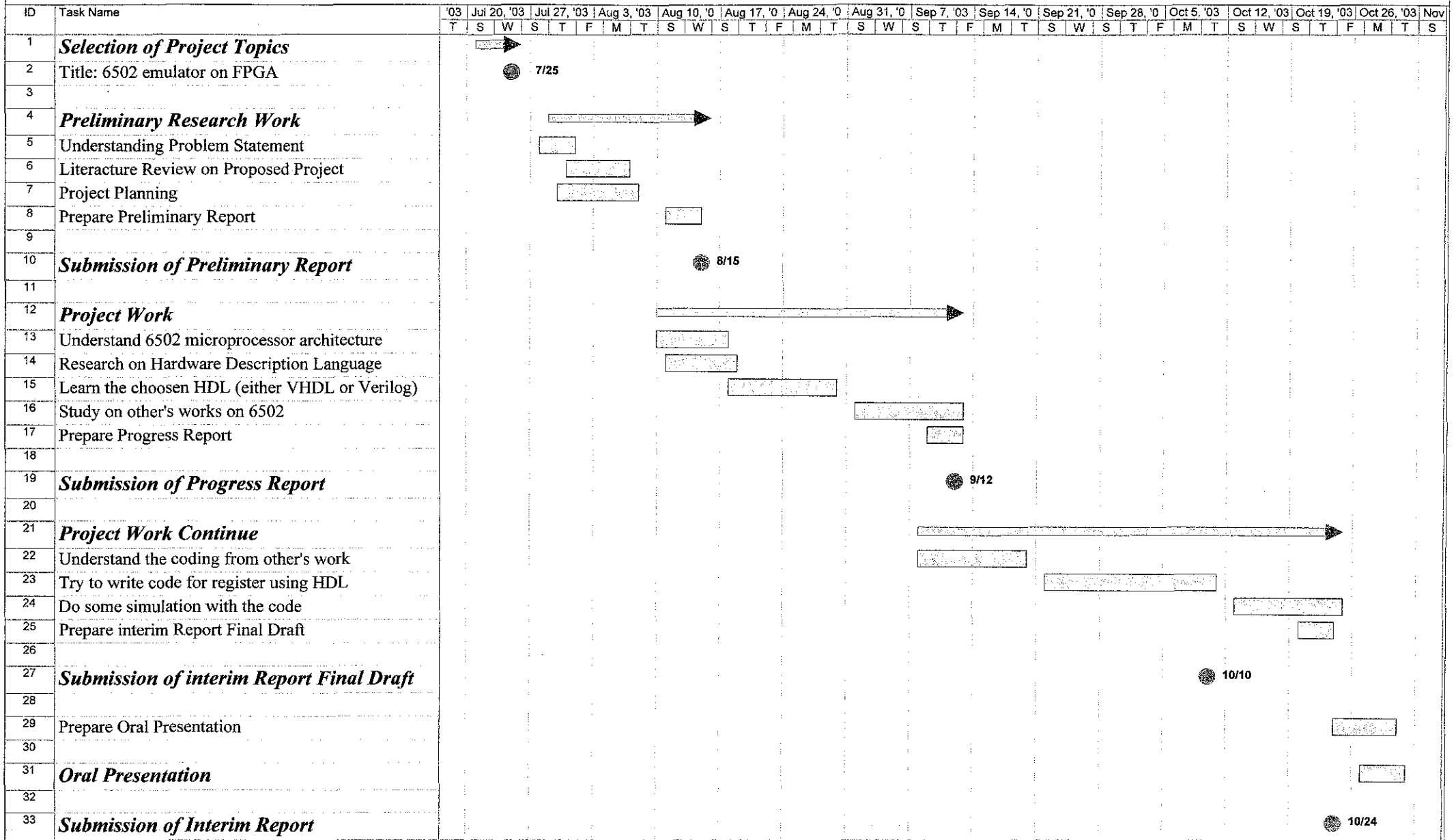
Internet Resources / WebPages

1. www.geocities.com/onecklruns/AssemblyInOneStep.htm (August 2003)
2. www.izabella.freeuk.com/6502CPU.htm (August 2003)
3. www.free-ip.com/6502/ (August 2003)
4. www.stanford.edu/~acylin/TheKarenA16BitRISCMicroprocessor.htm (September 2003)
5. www.cast-inc.com (October 2003)
6. www.artiarchieves.org (October 2003)
7. http://www.geocities.com/profdredd/cprogram/6502_ml.html (February 2004)
8. www.birdcomputer.ca/index.html (December 2003)
9. www.anglefire.com/in/verilogfaq/page3.html (March 2004)
10. <http://www.estec.esa.nl/wsmwww/core/soc.html> (March 2004)

Appendix A:

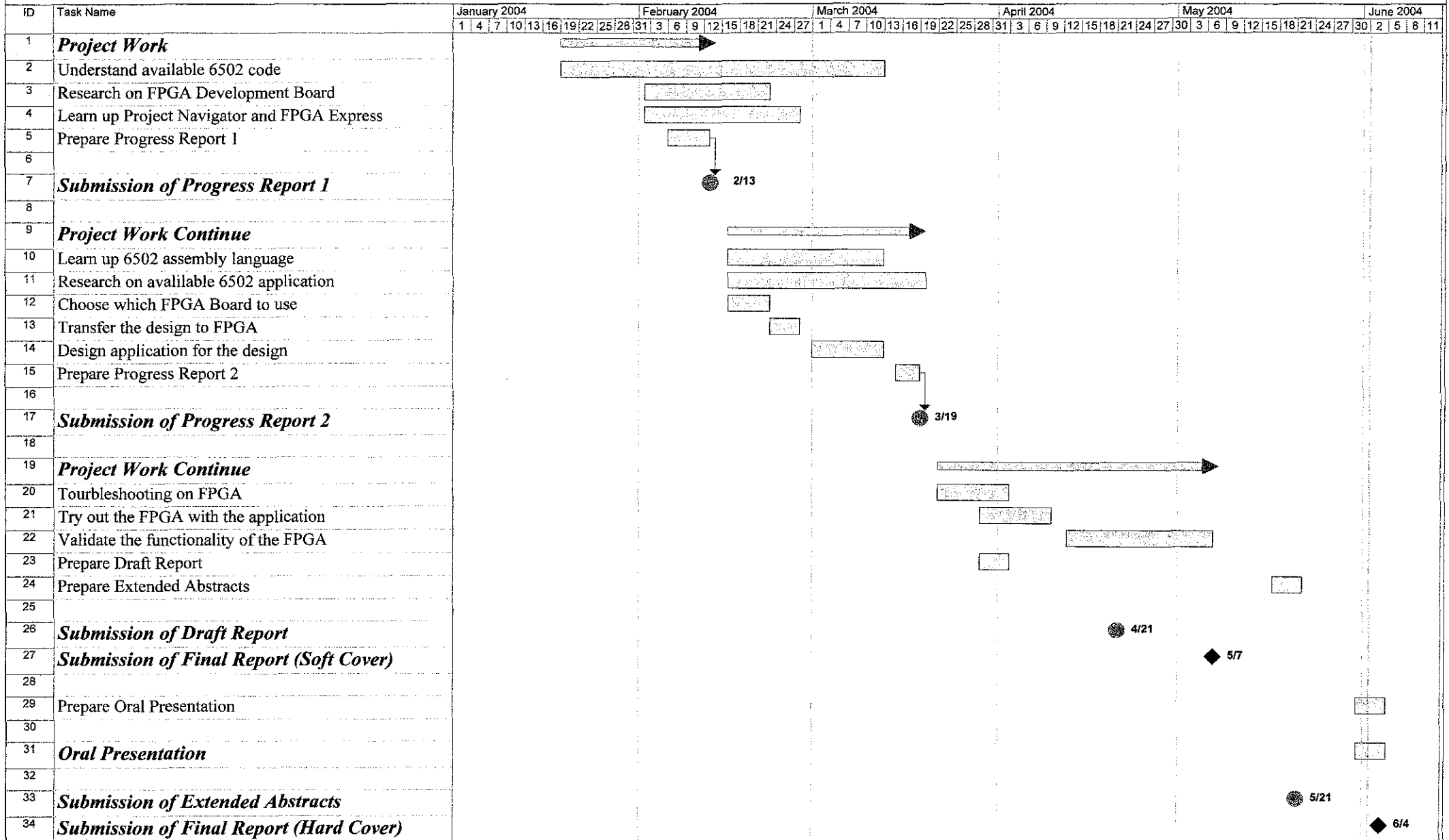
Gantt chart

5th Year 1st Semester Gantt Chart



Project: FYP 1st Semester Gantt Char Date: Wed 6/2/04	Task	Progress	Summary	External Tasks	Deadline	Split	Milestone	Project Summary	External Milestone
--	------	----------	---------	----------------	----------	-------	-----------	-----------------	--------------------

5th Year 2nd Semester Gantt Chart



Project: FYP 1st Semester Gantt Char Date: Wed 6/2/04	Task	Progress	Summary	External Tasks	Deadline
	Split	Milestone	Project Summary	External Milestone	

Appendix B

Overview on 6502 microprocessor [I2]

PIN ASSIGNMENT					
V_{SS}	□	1	40	□	\overline{RES}
\overline{RDY}	□	2	39	□	$\phi 2$ (OUT)
$\phi 1$ (OUT)	□	3	38	□	S.O.
\overline{IRQ}	□	4	37	□	$\phi 0$ (IN)
N.C.	□	5	36	□	N.C.
\overline{NMI}	□	6	35	□	N.C.
\overline{SYNC}	□	7	34	□	$\overline{R/W}$
V_{CC}	□	8	33	□	D0
A0	□	9	32	□	D1
A1	□	10	31	□	D2
A2	□	11	30	□	D3
A3	□	12	29	□	D4
A4	□	13	28	□	D5
A5	□	14	27	□	D6
A6	□	15	26	□	D7
A7	□	16	25	□	A15
A8	□	17	24	□	A14
A9	□	18	23	□	A13
A10	□	19	22	□	A12
A11	□	20	21	□	V_{SS}

Figure B1: 6502 Microprocessor Layouts

Pin layout

1	Ground
2	A negative transition halts the MPU. Allows for single step cycling etc.
3	CK 1 OUT Phase 1 clock output
4	/IRQ. If interrupt mask flag is not set, program counter jumps to FFFE & FFFF
5	n/c-
6	/NMI. Non-maskable interrupt requires low condition to jump to FFFA & FFFB
7	Sync Identifies the op-code fetch instructions
8	Vcc +5V
9 - 20	A0-11
22 - 25	A12-15 Address bus
21	Vss Ground
26 - 33	D0-7 Bi-directional / tristate data bus
34	R /W Read / Write line
35	n/c-
36	n/c-
37	CK in Single phase clock input
38	/SO Neg. input sets the overflow flag. Must be in sync. with CK1 trailing edge
39	CK 2 OUT Phase 2 clock output
40	/RES A low initialises the MPU and sets the program counter to FFFC & FFFD

Table B1: Pin Layout of 6502

Block Diagram of 6502

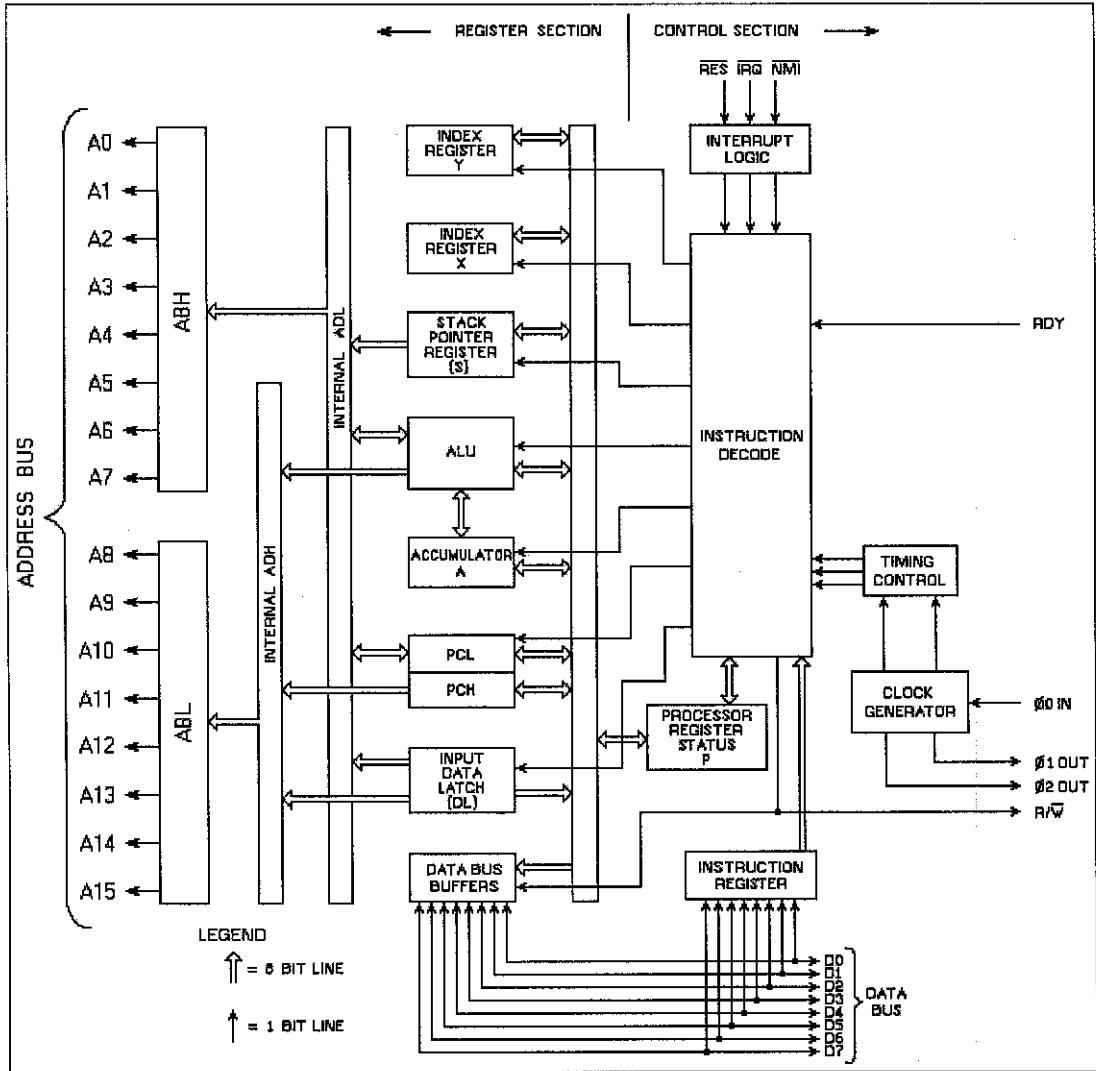


Figure B2: Block Diagram of 6502 [B8]

Appendix C

6502 Machine Language Overview [B2]

Instruction:

- 56 different instructions.
 - 13 different addressing modes.
 - Total of 151 legal 1-byte opcodes.
 - And one bug (indirect Jump at end of page).
- Execution of each instruction takes 2-7 cycles.
 - Every instruction has a base execution time.
 - Successful branch: +1.
 - Sometimes, +1 if indexing crosses page boundary.
 - We know exactly how long execution will take.
- Execution is always in-order.

Opcodes and Operands:

- All opcodes are 1 byte.
- Operands are 0, 1, or 2 bytes.
 - 1-byte operands
 - 8-bit constant
 - *Zero-page* address
 - Zero-page addresses were those that began 00h.
 - Zero-page addressing shorter & faster than normal.
 - Thus, zero-page space was in demand.
 - 1-byte offset from current address
 - Used in branch instructions.
 - 2-byte operands

- 16-bit memory location

Registers:

- 16-bit program counter and five 8-bit registers:
 - A (accumulator)
 - General purpose.
 - Stores arithmetic results.
 - X & Y (index registers)
 - Limited functionality.
 - Similar, but not identical in use.
 - S (stack pointer)
 - Prepend 01h to get the current top-of-stack address.
 - Can be accessed directly by transferring to X register.
 - 8 bits! So the stack is limited to 256 bytes.
 - P (processor status)
 - Holds seven 1-bit status flags.
 - Can be accessed directly by pushing on the stack.

Addressing Modes:

- 13 Addressing Modes:
 - Implied
 - No operand (example: register-to-register transfers)
 - Accumulator
 - No operand; instruction affects A register
 - Immediate
 - Operand: 1-byte constant.
 - Zero Page

- Operand: 1-byte zero page address.
- Relative
 - Operand: 1-byte offset from current address.
 - Used in branch instructions.
- Absolute
 - Operand: 2-byte address.
- Zero Page, X & Zero Page, Y
 - 1-byte zero page address + X [or Y, as appropriate].
- Absolute, X & Absolute, Y
 - 2-byte address + X [or Y, as appropriate].
- (Zero page indirect, X) & (Zero page indirect), Y
 - 1-byte zero-page address, contents of this & following address used as 2-byte address.
 - X added before dereferencing; Y added after.
- (Indirect)
 - 2-byte address, contents of this & following byte used as 2-byte address.
 - Available only in JuMP instruction.

Example of an assemble code:

START	LDY #\$05	Load Y: constant
LOOP	LDA STRING,Y	Get char to print
	JSR \$FDED	Print char in A
	DEY	Decrement Y
	BPL LOOP	If plus, continue
	RTS	Return
STRING	HEX 21 6F 6C	Reverse order
	HEX 6C 65 48	ASCII codes for ...

Appendix D

The 6502 Instruction Set [B8]

Load and Store Group

LDA	Load Accumulator	N,Z
LDX	Load X Register	N,Z
LDY	Load Y Register	N,Z
STA	Store Accumulator	
STX	Store X Register	
STY	Store Y Register	

Arithmetic Group

ADC	Add with Carry	N,V,Z,C
SBC	Subtract with Carry	N,V,Z,C

Increment and Decrement Group

INC	Increment a memory location	N,Z
INX	Increment the X register	N,Z
INY	Increment the Y register	N,Z
DEC	Decrement a memory location	N,Z
DEX	Decrement the X register	N,Z
DEY	Decrement the Y register	N,Z

Register Transfer Group

TAX	Transfer accumulator to X	N,Z
TAY	Transfer accumulator to Y	N,Z
TXA	Transfer X to accumulator	N,Z
TYA	Transfer Y to accumulator	N,Z

Logical Group

AND	Logical AND	N,Z
EOR	Exclusive OR	N,Z
ORA	Logical Inclusive OR	N,Z

Compare and Bit Test Group

CMP	Compare accumulator	N,Z,C
CPX	Compare X register	N,Z,C
CPY	Compare Y register	N,Z,C

BIT	Bit Test	N,V,Z
-----	----------	-------

Shift and Rotate Group

ASL	Arithmetic Shift Left	N,Z,C
LSR	Logical Shift Right	N,Z,C
ROL	Rotate Left	N,Z,C
ROR	Rotate Right	N,Z,C

Jump and Branch Group

JMP	Jump to another location
BCC	Branch if carry flag clear
BCS	Branch if carry flag set
BEQ	Branch if zero flag set
BMI	Branch if negative flag set
BNE	Branch if zero flag clear
BPL	Branch if negative flag clear
BVC	Branch if overflow flag clear
BVS	Branch if overflow flag set

Stack Group

TSX	Transfer stack pointer to X	N,Z
TXS	Transfer X to stack pointer	
PHA	Push accumulator on stack	
PHP	Push processor status on stack	
PLA	Pull accumulator from stack	N,Z
PLP	Pull processor status from stack	All

Status Flag Change Group

CLC	Clear carry flag	C
CLD	Clear decimal mode flag	D
CLI	Clear interrupt disable flag	I
CLV	Clear overflow flag	V
SEC	Set carry flag	C
SED	Set decimal mode flag	D
SEI	Set interrupt disable flag	I

Subroutine and Interrupt Group

JSR	Jump to a subroutine	
RTS	Return from subroutine	
BRK	Force an interrupt	B
RTI	Return from Interrupt	All
NOP	No Operation	

Mnemonic	Description	Flags
Load & Store Instructions		
LDA	load accumulator	NZ
LDX	load X index	NZ
LDY	load Y index	NZ
STA	store accumulator	-
STX	store X index	-
STY	store Y index	-
STZ	store zero	-
Stack Operations		
PHA	push accumulator	-
PHX	push X index	-
PHY	push Y index	-
PHP	push processor flags	-
PLA	pull (pop) accumulator	NZ
PLX	pull (pop) X index	NZ
PLY	pull (pop) Y index	NZ
PLP	pull (pop) processor flags	All
TSX	transfer stack pointer to X	NZ
TXS	transfer stack pointer to X	-
Increment & Decrement Operations		
INA	increment accumulator	NZ
INX	increment X index	NZ
INY	increment Y index	NZ
DEA	decrement accumulator	NZ
DEX	decrement X index	NZ
DEY	decrement Y index	NZ
INC	increment memory location	NZ
DEC	decrement memory location	NZ
Shift Operations		
ASL	arithmetic shift left, high bit into carry	NZC
LSR	logical shift right, low bit into carry	N=0 ZC
ROL	rotate left through carry	NZC
ROR	rotate right through carry	NZC
Logical Operations		
AND	and accumulator	NZ
ORA	or accumulator	NZ
EOR	exclusive-or accumulator	NZ
BIT	test bits against accumulator (1)	N=M7

		V=M6 Z
CMP	compare with accumulator	NZC
CPX	compare with X index	NZC
CPY	compare with Y index	NZC
TRB	test and reset bits	x
TSB	test and set bits	x
RMB	reset memory bit	x
SMB	reset memory bit	x
Math Operations		
ADC	add accumulator, with carry	NZCV
SBC	subtract accumulator, with borrow	NZCV
Flow Control Instructions		
JMP	unconditional jump	-
JSR	jump Subroutine	-
RTS	return from Subroutine	-
RTI	return from Interrupt	From Stack
BRA	branch Always	-
BEQ	branch on equal (zero set)	-
BNE	branch on not equal (zero clear)	-
BCC	branch on carry clear (2)	-
BCS	branch on carry set (2)	-
BVC	branch on overflow clear	-
BVS	branch on overflow set	-
BMI	branch on minus	-
BPL	branch on plus	-
BBR	branch on bit reset (zero)	-
BBS	branch on bit set (one)	-
Processor Status Instructions		
CLC	clear carry flag	C=0
CLD	clear decimal mode	D=0
CLI	clear interrupt disable bit	I=0
CLV	clear overflow flag	V=0
SEC	set carry flag	C=1
SED	set decimal mode	D=1
SEI	set interrupt disable bit	I=1
Transfer Instructions		
TAX	transfer accumulator to X index	NZ
TAY	transfer accumulator to Y index	NZ
TXA	transfer X index to accumulator	NZ

TYA	transfer Y index to accumulator	NZ
Misc Instructions		
NOP	no operation	-
BRK	force break	B=1

Table D1: 6502 Instruction Set

Notes:

1. The BIT instruction copies bit 6 to the V flag, and bit 7 to the N flag (except in immediate addressing mode where V & N are untouched.) The accumulator and the operand are ANDed and the Z flag is set appropriately.
2. The BCC & BCS instructions are sometimes known as BLT (branch less than) and BGE (branch greater or equal), respectively.

Appendix E

6502 Instruction Encoding [B8]

Mnemonic	Addressing mode	Form	Opcode	Size	Timing
ADC	Immediate	ADC #Oper	69	2	2
	Zero Page	ADC Zpg	65	2	3
	Zero Page,X	ADC Zpg,X	75	2	4
	Absolute	ADC Abs	6D	3	4
	Absolute,X	ADC Abs,X	7D	3	4
	Absolute,Y	ADC Abs,Y	79	3	4
	(Zero Page,X)	ADC (Zpg,X)	61	2	6
	(Zero Page),Y	ADC (Zpg),Y	71	2	5
	(Zero Page)	ADC (Zpg)	72	2	5
AND	Immediate	AND #Oper	29	2	2
	Zero Page	AND Zpg	25	2	3
	Zero Page,X	AND Zpg,X	35	2	4
	Absolute	AND Abs	2D	3	4
	Absolute,X	AND Abs,X	3D	3	4
	Absolute,Y	AND Abs,Y	39	3	4
	(Zero Page,X)	AND (Zpg,X)	21	2	6
	(Zero Page),Y	AND (Zpg),Y	31	2	5
	(Zero Page)	AND (Zpg)	32	2	5
ASL	Accumulator	ASL A	0A	1	2
	Zero Page	ASL Zpg	06	2	5
	Zero Page,X	ASL Zpg,X	16	2	6
	Absolute	ASL Abs	0E	3	6
	Absolute,X	ASL Abs,X	1E	3	7
BBR0	Relative	BBR0 Oper	0F	2	2
BBR1	Relative	BBR1 Oper	1F	2	2
BBR2	Relative	BBR2 Oper	2F	2	2
BBR3	Relative	BBR3 Oper	3F	2	2
BBR4	Relative	BBR4 Oper	4F	2	2
BBR5	Relative	BBR5 Oper	5F	2	2
BBR6	Relative	BBR6 Oper	6F	2	2
BBR7	Relative	BBR7 Oper	7F	2	2
BBS0	Relative	BBS0 Oper	8F	2	2

BBS1	Relative	BBS1 Oper	9F	2	2
BBS2	Relative	BBS2 Oper	AF	2	2
BBS3	Relative	BBS3 Oper	BF	2	2
BBS4	Relative	BBS4 Oper	CF	2	2
BBS5	Relative	BBS5 Oper	DF	2	2
BBS6	Relative	BBS6 Oper	EF	2	2
BBS7	Relative	BBS7 Oper	FF	2	2
BCC	Relative	BCC Oper	90	2	2
BCS	Relative	BCS Oper	B0	2	2
BEQ	Relative	BEQ Oper	F0	2	2
BIT	Immediate	BIT #Oper	89	2	2
	Zero Page	BIT Zpg	24	2	3
	Zero Page,X	BIT Zpg,X	34	2	4
	Absolute	BIT Abs	2C	3	4
	Absolute,X	BIT Abs,X	3C	3	4
BMI	Relative	BMI Oper	30	2	2
BNE	Relative	BNE Oper	D0	2	2
BPL	Relative	BPL Oper	10	2	2
BRA	Relative	BRA Oper	80	2	3
BRK	Implied	BRK	00	1	7
BVC	Relative	BVC Oper	50	2	2
BVS	Relative	BVS Oper	70	2	2
CLC	Implied	CLC	18	1	2
CLD	Implied	CLD	D8	1	2
CLI	Implied	CLI	58	1	2
CLV	Implied	CLV	B8	1	2
CMP	Immediate	CMP #Oper	C9	2	2
	Zero Page	CMP Zpg	C5	2	3
	Zero Page,X	CMP Zpg	D5	2	4
	Absolute	CMP Abs	CD	3	4
	Absolute,X	CMP Abs,X	DD	3	4
	Absolute,Y	CMP Abs,Y	D9	3	4
	(Zero Page,X)	CMP (Zpg,X)	C1	2	6
	(Zero Page),Y	CMP (Zpg),Y	D1	2	5
(Zero Page)	CMP (Zpg)	D2	2	5	
CPX	Immediate	CPX #Oper	E0	2	2
	Zero Page	CPX Zpg	E4	2	3
	Absolute	CPX Abs	EC	3	4
CPY	Immediate	CPY #Oper	C0	2	2

	Zero Page	CPY Zpg	C4	2	3
	Absolute	CPY Abs	CC	3	4
DEA	Accumulator	DEA	3A	1	2
DEC	Zero Page	DEC Zpg	C6	2	5
	Zero Page,X	DEC Zpg,X	D6	2	6
	Absolute	DEC Abs	CE	3	6
	Absolute,X	DEC Abs,X	DE	3	7
DEX	Implied	DEX	CA	1	2
DEY	Implied	DEY	88	1	2
EOR	Immediate	EOR #Oper	49	2	2
	Zero Page	EOR Zpg	45	2	3
	Zero Page,X	EOR Zpg,X	55	2	4
	Absolute	EOR Abs	4D	3	4
	Absolute,X	EOR Abs,X	5D	3	4
	Absolute,Y	EOR Abs,Y	59	3	4
	(Zero Page,X)	EOR (Zpg,X)	41	2	6
	(Zero Page),Y	EOR (Zpg),Y	51	2	5
	(Zero Page)	EOR (Zpg)	52	2	5
INA	Accumulator	INA	1A	1	2
INC	Zero Page	INC Zpg	E6	2	5
	Zero Page,X	INC Zpg,X	F6	2	6
	Absolute	INC Abs	EE	3	6
	Absolute,X	INC Abs,X	FE	3	7
INX	Implied	INX	E8	1	2
INY	Implied	INY	C8	1	2
JMP	Absolute	JMP Abs	4C	3	3
	(Absolute)	JMP (Abs)	6C	3	5
	(Absolute,X)	JMP (Abs,X)	7C	3	6
JSR	Absolute	JSR Abs	20	3	6
LDA	Immediate	LDA #Oper	A9	2	2
	Zero Page	LDA Zpg	A5	2	3
	Zero Page,X	LDA Zpg,X	B5	2	4
	Absolute	LDA Abs	AD	3	4
	Absolute,X	LDA Abs,X	BD	3	4
	Absolute,Y	LDA Abs,Y	B9	3	4
	(Zero Page,X)	LDA (Zpg,X)	A1	2	6
	(Zero Page),Y	LDA (Zpg),Y	B1	2	5
	(Zero Page)	LDA (Zpg)	B2	2	5
LDX	Immediate	LDX #Oper	A2	2	2

	Zero Page	LDX Zpg	A6	2	3
	Zero Page,Y	LDX Zpg,Y	B6	2	4
	Absolute	LDX Abs	AE	3	4
	Absolute,Y	LDX Abs,Y	BE	3	4
LDY	Immediate	LDY #Oper	A0	2	2
	Zero Page	LDY Zpg	A4	2	3
	Zero Page,Y	LDY Zpg,X	B4	2	4
	Absolute	LDY Abs	AC	3	4
	Absolute,Y	LDY Abs,X	BC	3	4
LSR	Accumulator	LSR A	4A	1	2
	Zero Page	LSR Zpg	46	2	5
	Zero Page,X	LSR Zpg,X	56	2	6
	Absolute	LSR Abs	4E	3	6
	Absolute,X	LSR Abs,X	5E	3	7
NOP	Implied	NOP	EA	1	2
ORA	Immediate	ORA #Oper	09	2	2
	Zero Page	ORA Zpg	05	2	3
	Zero Page,X	ORA Zpg,X	15	2	4
	Absolute	ORA Abs	0D	3	4
	Absolute,X	ORA Abs,X	1D	3	4
	Absolute,Y	ORA Abs,Y	19	3	4
	(Zero Page,X)	ORA (Zpg,X)	01	2	6
	(Zero Page),Y	ORA (Zpg),Y	11	2	5
	(Zero Page)	ORA (Zpg)	12	2	5
PHA	Implied	PHA	48	1	3
PHX	Implied	PHX	DA	1	3
PHY	Implied	PHY	5A	1	3
PLA	Implied	PLA	68	1	4
PLX	Implied	PLX	FA	1	4
PLY	Implied	PLY	7A	1	4
ROL	Accumulator	ROL A	2A	1	2
	Zero Page	ROL Zpg	26	2	5
	Zero Page,X	ROL Zpg,X	36	2	6
	Absolute	ROL Abs	2E	3	6
	Absolute,X	ROL Abs,X	3E	3	7
ROR	Accumulator	ROR A	6A	1	2
	Zero Page	ROR Zpg	66	2	5
	Zero Page,X	ROR Zpg,X	76	2	6
	Absolute	ROR Abs	6E	3	6

	Absolute,X	ROR Abs,X	7E	3	7
RTI	Implied	RTI	40	1	6
RTS	Implied	RTS	60	1	6
SBC	Immediate	SBC #Oper	E9	2	2
	Zero Page	SBC Zpg	E5	2	3
	Zero Page,X	SBC Zpg,X	F5	2	4
	Absolute	SBC Abs	ED	3	4
	Absolute,X	SBC Abs,X	FD	3	4
	Absolute,Y	SBC Abs,Y	F9	3	4
	(Zero Page,X)	SBC (Zpg,X)	E1	2	6
	(Zero Page),Y	SBC (Zpg),Y	F1	2	5
	(Zero Page)	SBC (Zpg)	F2	2	5
SEC	Implied	SEC	38	1	2
SED	Implied	SED	F8	1	2
SEI	Implied	SEI	78	1	2
STA	Zero Page	STA Zpg	85	2	3
	Zero Page,X	STA Zpg,X	95	2	4
	Absolute	STA Abs	8D	3	4
	Absolute,X	STA Abs,X	9D	3	5
	Absolute,Y	STA Abs,Y	99	3	5
	(Zero Page,X)	STA (Zpg,X)	81	2	6
	(Zero Page),Y	STA (Zpg),Y	91	2	6
	(Zero Page)	STA (Zpg)	92	2	5
STX	Zero Page	STX Zpg	86	2	3
	Zero Page,Y	STX Zpg,Y	96	2	4
	Absolute	STX Abs	8E	3	4
STY	Zero Page	STY Zpg	84	2	3
	Zero Page,X	STY Zpg,X	94	2	4
	Absolute	STY Abs	8C	3	4
STZ	Zero Page	STZ Zpg	64	2	3
	Zero Page,X	STZ Zpg,X	74	2	4
	Absolute	STZ Abs	9C	3	4
	Absolute,X	STZ Abs,X	9E	3	5
TAX	Implied	TAX	AA	1	2
TAY	Implied	TAY	A8	1	2
TRB	Zero Page	TRB Zpg	14	2	5
	Absolute	TRB Abs	1C	3	6
TSB	Zero Page	TSB Zpg	04	2	5

	Absolute	TSB Abs	0C	3	6
TSX	Implied	TSX	BA	1	2
TXA	Implied	TXA	8A	1	2
TXS	Implied	TXS	9A	1	2
TYA	Implied	TYA	98	1	2

Table E1: 6502 Instruction Encoding

Appendix F

Spartan-II FPGA Family [B13]

Introduction

The Spartan-II 2.5V Field-Programmable Gate Array family gives users high performance, abundant logic resources, and a rich feature set, all at an exceptionally low price. The six-member family offers densities ranging from 15,000 to 200,000 system gates, as shown in Table 1. System performance is supported up to 200 MHz. Spartan-II devices deliver more gates, I/Os, and features per dollar than other FPGAs by combining advanced process technology with a streamlined Virtex-based architecture. Features include block RAM (to 56K bits), distributed RAM (to 75,264 bits), 16 selectable I/O standards, and four DLLs. Fast, predictable interconnect means that successive design iterations continue to meet timing requirements. The Spartan-II family is a superior alternative to mask-programmed ASICs. The FPGA avoids the initial cost, lengthy development cycles, and inherent risk of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary (impossible with ASICs).

Features

- Second generation ASIC replacement technology
- Densities as high as 5,292 logic cells with up to 200,000 system gates
- Streamlined features based on Virtex architecture
- Unlimited reprogrammability
- Very low cost

- System level features
- SelectRAM+™ hierarchical memory:
 - 16 bits/LUT distributed RAM

- Configurable 4K bit block RAM
- Fast interfaces to external RAM
- Fully PCI compliant
- Low-power segmented routing architecture
- Full readback ability for verification/observability
- Dedicated carry logic for high-speed arithmetic
- Dedicated multiplier support
- Cascade chain for wide-input functions
- Abundant registers/latches with enable, set, reset
- Four dedicated DLLs for advanced clock control
- Four primary low-skew global clock distribution nets
- IEEE 1149.1 compatible boundary scan logic

- Versatile I/O and packaging
- Low cost packages available in all densities
- Family footprint compatibility in common packages
- 16 high-performance interface standards
- Hot swap Compact PCI friendly
- Zero hold time simplifies system timing

- Fully supported by powerful Xilinx development system
- Foundation ISE Series: Fully integrated software
- Alliance Series: For use with third-party tools
- Fully automatic mapping, placement, and routing

Table 1: Spartan-II FPGA Family Members

Device	Logic Cells	System Gates (Logic and RAM)	CLB Array (R x C)	Total CLBs	Maximum Available User I/O ⁽¹⁾	Total Distributed RAM Bits	Total Block RAM Bits
XC2S15	432	15,000	8 x 12	96	86	6,144	16K
XC2S30	972	30,000	12 x 18	216	132	13,824	24K
XC2S50	1,728	50,000	16 x 24	384	176	24,576	32K
XC2S100	2,700	100,000	20 x 30	600	196	38,400	40K
XC2S150	3,888	150,000	24 x 36	864	260	55,296	48K
XC2S200	5,292	200,000	28 x 42	1,176	284	75,264	56K

General Overview

The Spartan-II family of FPGAs have a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs), surrounded by a perimeter of programmable Input/Output Blocks (IOBs). There are four Delay-Locked Loops (DLLs), one at each corner of the die. Two columns of block RAM lie on opposite sides of the die, between the CLBs and the IOB columns. These functional elements are interconnected by a powerful hierarchy of versatile routing channels (see Figure B1).

Spartan-II FPGAs are customized by loading configuration data into internal static memory cells. Unlimited reprogramming cycles are possible with this approach. Stored values in these cells determine logic functions and interconnections implemented in the FPGA. Configuration data can be read from an external serial PROM (master serial mode), or written into the FPGA in slave serial, slave parallel, or Boundary Scan modes.

Spartan-II FPGAs are typically used in high-volume applications where the versatility of a fast programmable solution adds benefits. Spartan-II FPGAs are ideal for shortening product development cycles while offering a cost-effective solution for high volume production.

Spartan-II FPGAs achieve high-performance, low-cost operation through advanced architecture and semiconductor technology. Spartan-II devices provide system clock

rates up to 200 MHz. Spartan-II FPGAs offer the most cost-effective solution while maintaining leading edge performance. In addition to the conventional benefits of high-volume programmable logic solutions, Spartan-II FPGAs also offer on-chip synchronous single-port and dual-port RAM (block and distributed form), DLL clock drivers, programmable set and reset on all flip-flops, fast carry logic, and many other features.

The Xilinx XC17S00A PROM family is recommended for serial configuration of Spartan-II FPGAs. The In-System Programmable (ISP) XC18V00 PROM family is recommended for parallel or serial configuration.

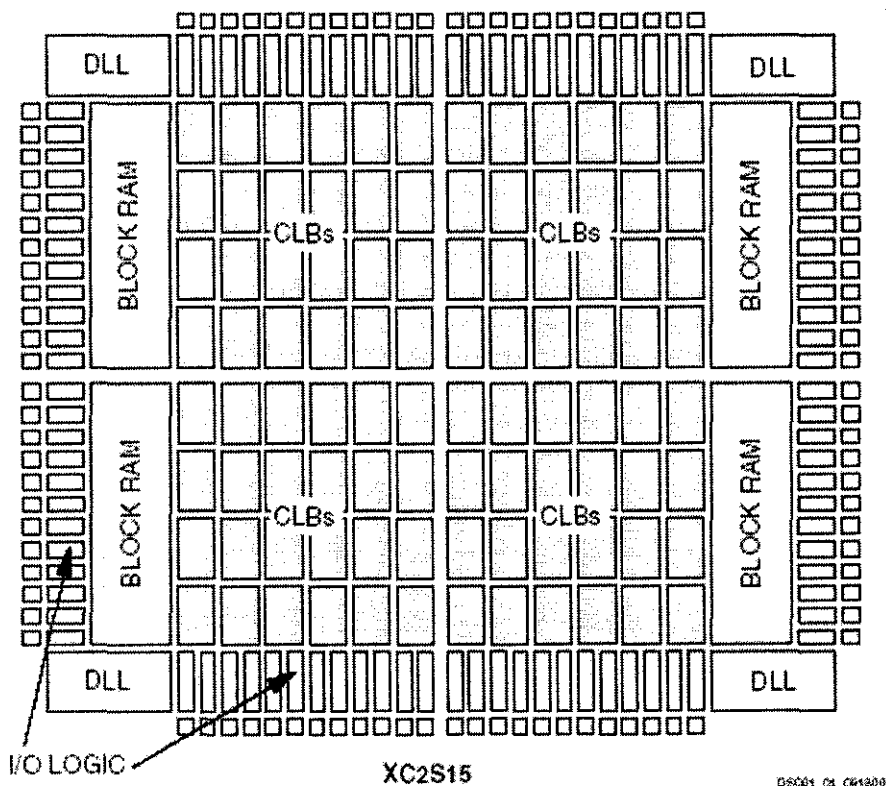


Figure F1: Basic Spartan-II Family FPGA Block Diagram

For more information on Spartan-II FPGA, please refer to Spartan-II 2.5V FPGA Family: Complete Data Sheet DS001 September 3, 2003. There is around 100 pages information on this device. The softcopy of this Data Sheet is easily available from Xilinx website.

Appendix G

D Flip-flop Verilog Model Testbenches

Testbench for Timing Simulation in Figure 4.1

```
//-----  
// Title   : dff_tb  
// Design  : DFlipFlop  
// Author   : KEH  
// Company  : UTP  
//-----  
// Description :  
//-----  
  
`timescale 1ps / 1ps  
module dff_tb;  
  
//Internal signals declarations:  
reg data;  
reg clock;  
wire q;  
  
// Unit Under Test port map  
dff UUT (  
    .data(data),  
    .clock(clock),  
    .q(q));  
  
always  
begin  
  
//Clock initial  
#30 clock <= 1'b0;  
#30 clock <= 1'b1;  
end  
  
initial  
begin  
  
    data = 1'b0;  
    #150 data = 1'b1;  
    #250 data = 1'b0;  
    #300 data = 1'b1;  
end  
  
endmodule
```

Testbench for Timing Simulation in Figure 4.2

```
//-----  
// Title   : dff_async_tb  
// Design  : DFlipFlop  
// Author  : KEH  
// Company : UTP  
//-----  
// Description :  
//-----  
  
`timescale 1ps / 1ps  
module dff_async_tb;  
  
//Internal signals declarations:  
reg data;  
reg clock;  
reg reset;  
wire q;  
  
// Unit Under Test port map  
dff_async UUT (  
    .data(data),  
    .clock(clock),  
    .reset(reset),  
    .q(q));  
  
always  
begin  
  
//Clock initial  
#20 clock <= 1'b0;  
#20 clock <= 1'b1;  
end  
  
initial  
begin  
  
    data = 1'b0;  
    #60 data = 1'b1;  
    reset = 1'b1;  
    #60 data = 1'b0;  
    #60 data = 1'b1;  
    #10 reset = 1'b0;  
    #60 data = 1'b0;  
    #60 data = 1'b1;  
  
end  
  
endmodule
```

Testbench for Timing Simulation in Figure 4.3

```
//-----  
// Title   : dff_sync_tb  
// Design  : DFlipFlop  
// Author  : 0  
// Company : 0  
//-----  
// Description :  
//-----  
  
`timescale 1ps / 1ps  
module dff_sync_tb;  
  
//Internal signals declarations:  
reg data;  
reg clock;  
reg reset;  
wire q;  
  
// Unit Under Test port map  
dff_sync UUT (  
    .data(data),  
    .clock(clock),  
    .reset(reset),  
    .q(q));  
  
always  
begin  
  
//Clock initial  
#20 clock <= 1'b0;  
#20 clock <= 1'b1;  
end  
  
initial  
begin  
  
    data = 1'b0;  
    #60 data = 1'b1;  
    reset = 1'b1;  
    #60 data = 1'b0;  
    #60 data = 1'b1;  
    #10 reset = 1'b0;  
    #60 data = 1'b0;  
    #60 data = 1'b1;  
  
end  
  
endmodule
```

Testbench for Timing Simulation in Figure 4.4

```
//-----  
// Title   : dff_cke_tb  
// Design  : DFlipFlop  
// Author  : KEH  
// Company : UTP  
//-----  
// Description :  
//-----  
  
`timescale 1ps / 1ps  
module dff_cke_tb;  
  
//Internal signals declarations:  
reg data;  
reg clock;  
reg reset;  
reg cke;  
wire q;  
  
// Unit Under Test port map  
dff_cke UUT (  
    .data(data),  
    .clock(clock),  
    .reset(reset),  
    .cke(cke),  
    .q(q));  
  
always  
begin  
  
    //Clock initial  
#20 clock <= 1'b0;  
#20 clock <= 1'b1;  
end  
  
initial  
begin  
    cke = 1'b0;  
    data = 1'b0;  
#40 data = 1'b1;  
    reset = 1'b1;  
#10 reset = 1'b0;  
    cke = 1'b1;  
#40 data = 1'b0;  
    reset = 1'b1;  
#40 data = 1'b1;  
#40 data = 1'b0;  
#40 data = 1'b1;  
#40 data = 1'b0;  
#40 data = 1'b1;  
#40 data = 1'b0;  
end
```



```
    cke = 1'b0;  
    #40 data = 1'b1;  
    #40 data = 1'b0;  
    #40 data = 1'b1;  
    reset = 1'b0;  
end  
  
endmodule
```

Appendix H

B3-SPARTAN2+ QuickStart Guide 2.0 [11,12]

1. Install the WebPACK_61_fcul_i software

Just follow all the graphical user interface instruction and select all component or devices.

2. Create a new project

Start:

WebPACK Project Navigator

Do:

File

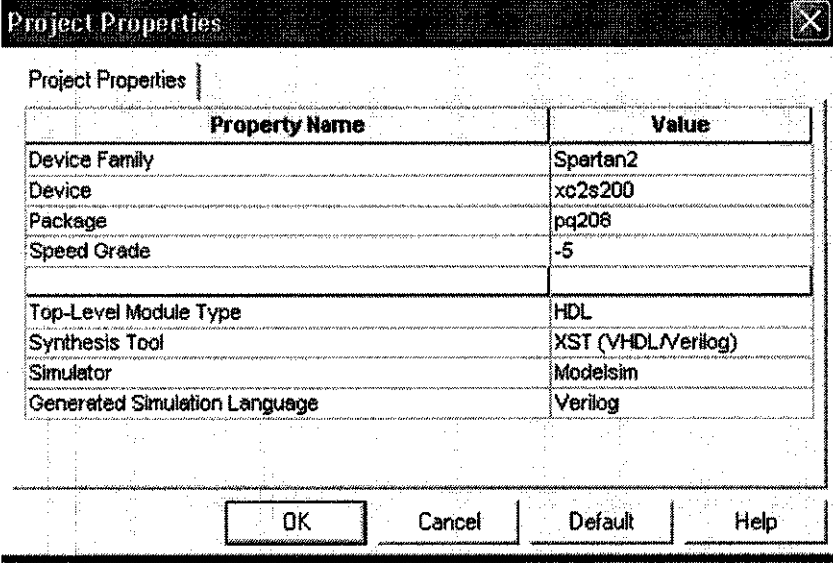
New Project

Type in a new Project name:

LEDFLASH

For the Top-Level Module Type, choose HDL.

Then fill in the Table as shown in Figure H.1 below:



Property Name	Value
Device Family	Spartan2
Device	xc2s200
Package	pg208
Speed Grade	-5
Top-Level Module Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim
Generated Simulation Language	Verilog

OK Cancel Default Help

Figure H.1 New Project

For the rest steps just press Next button until you will see the information as shown in Figure H.2 below:

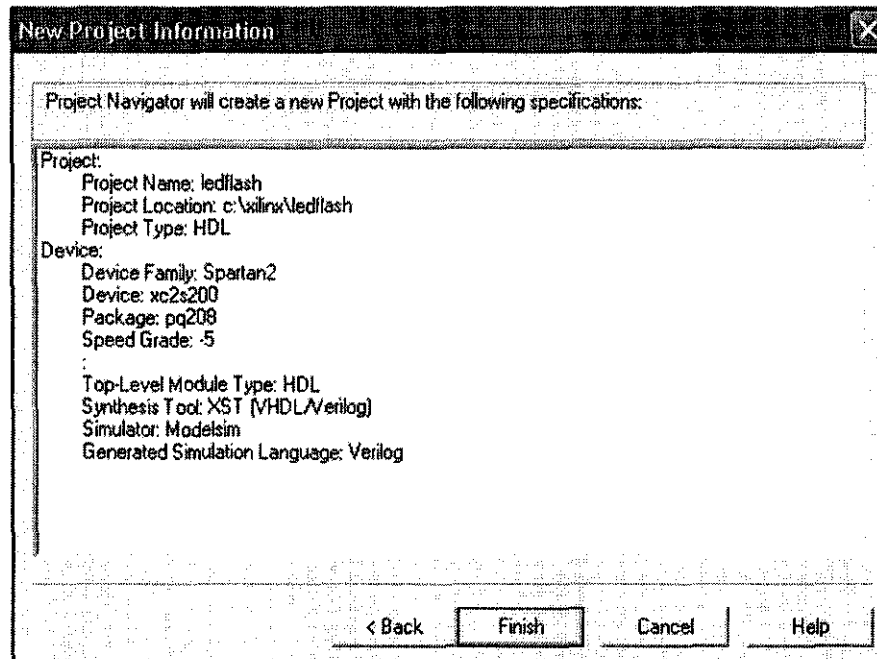


Figure H.2 New Project Information

Do:

Project

New Source

Then choose on VHDL Module.

File name= LEDFLASH

Fill in the table with

CLK in

LED out

Next > Finish

A window will pop up with the start of the new LEDFLASH.vhd code.
Modify it so that it looks like the code in Figure H.3.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  -- Uncomment the following lines to use the declarations that are
7  -- provided for instantiating Xilinx primitive components.
8  --library UNISIM;
9  --use UNISIM.VComponents.all;
10
11  entity ledflash is
12      Port ( CLK : in std_logic;
13            LED : out std_logic);
14  end ledflash;
15
16  architecture Behavioral of ledflash is
17      signal countL : std_logic_vector(23 downto 0);
18  begin
19      increment: process (CLK) begin
20          if (clk'event and clk = '1') then
21              countL <= countL + 1;
22          end if;
23      end process;
24      LED <= std_logic(countL(23));
25
26  end Behavioral;
27
```

Figure H.3 LEDFLASH VHDL code

3. Assign the pinouts of the device

You will now need to tell the compiler which pins you want associated with which signals in your design. The compiler reads the UCF (user constraints file) file to get this information. You can use the Constraints Editor process to enter the information that is written to the UCF file. In the following procedure, you will write the pinout / location information into the UCF file, using the Constraints Editor Process. Single click on the LEDFLASH.vhd item in the “Sources in Project” box, on the top left of the design environment. The “Processes for Source” box on the bottom left of your design environment now contains all of the processes that can operate on this design. Expand out the processes so that you can see the Constraints Editor

process. Click on Edit UCF file process. Have a look at the UCF file, and then close this window.

Double click on the Implement Design process. You should get green-ticks on all of the Translate, Map and Place-and-Route process items, as these processes complete. Your design has now been implemented. All that remains is the creation of the bitfile for downloading, and the downloading process itself.

4. Create the bitfile (.rbt) for downloading

Right click on the Create Programming File process item and select Properties. Make sure the Create ASCII Configuration File checkbox is checked (this tells the bitstream creator program to write out the .rbt rawbit file, which the iMPACT utility reads for downloading).

Double click on the Create Programming File process. A .rbt file is written out to your design folder.

5. Download the .rbt file to your FPGA

Connect the download pod board to the parallel port of the PC using the parallel port flat-cable. Connect the B3-SPARTAN2+ board to the download pod board using the 10-way flat-cable.

Make sure that your regulated DC power supply is +5V before you connect it to your SPARTAN2+ board Then connect the power supply to the SPARTAN2+ board.

In the “Processed for Source” box, double click on “Configure Device (iMPACT)” Then the figures below will appear, just follow the information in the Figure shown to download your design or test the FPGA board.

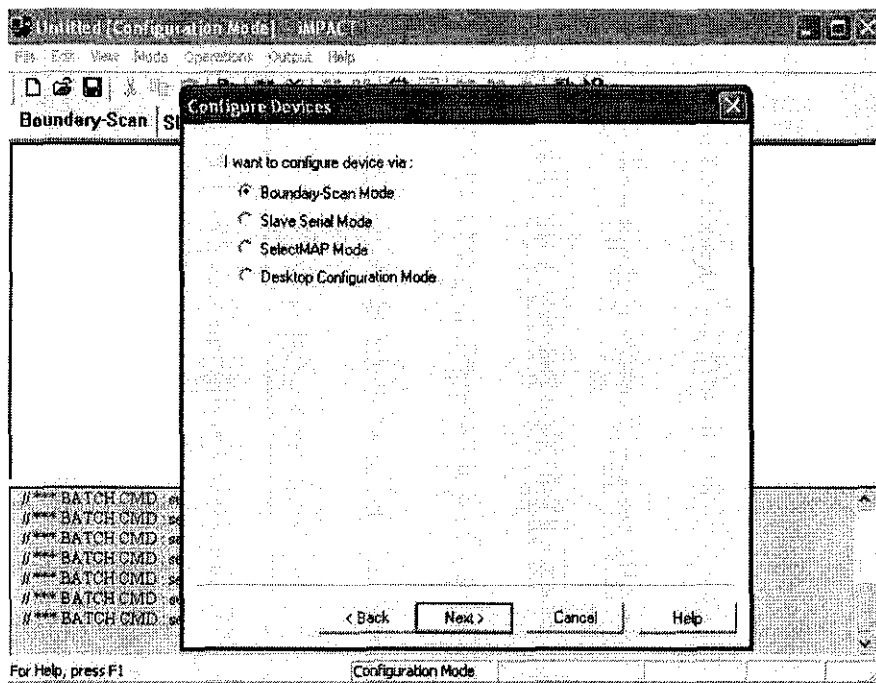


Figure H.4 iMPACT flow 1

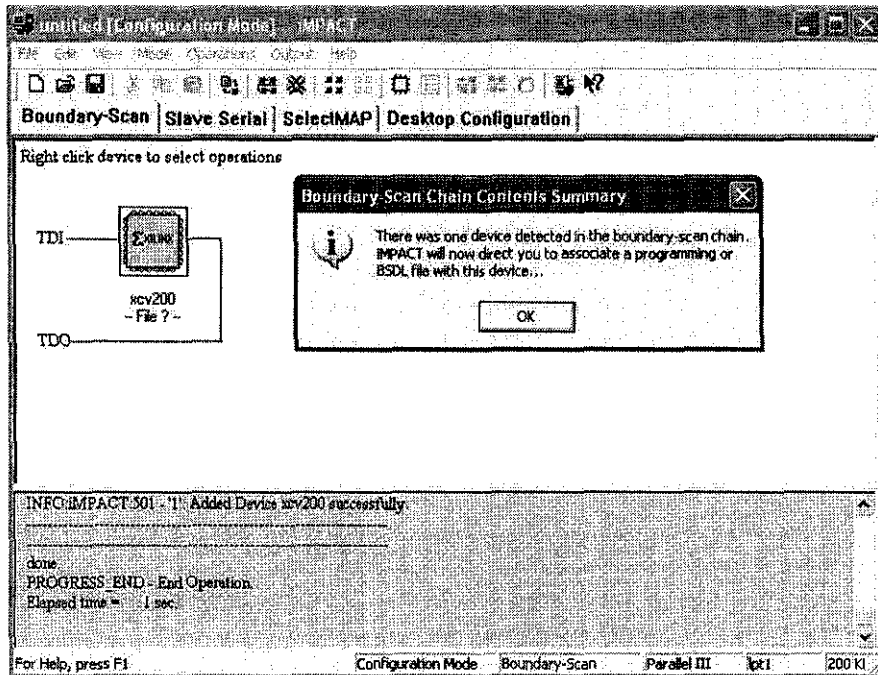


Figure H.5 iMPACT flow 2

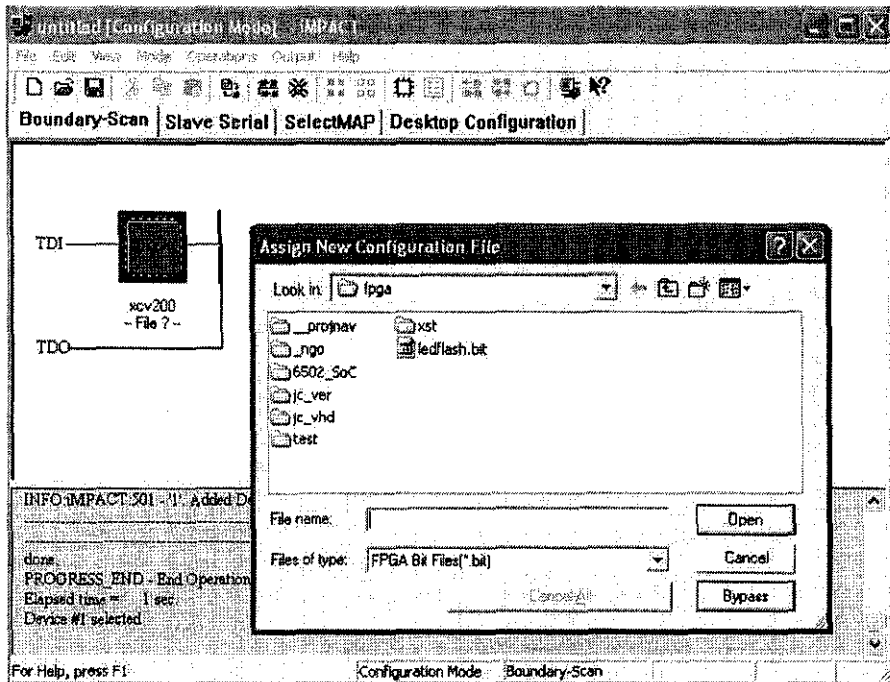


Figure H.6 iMPACT flow 3

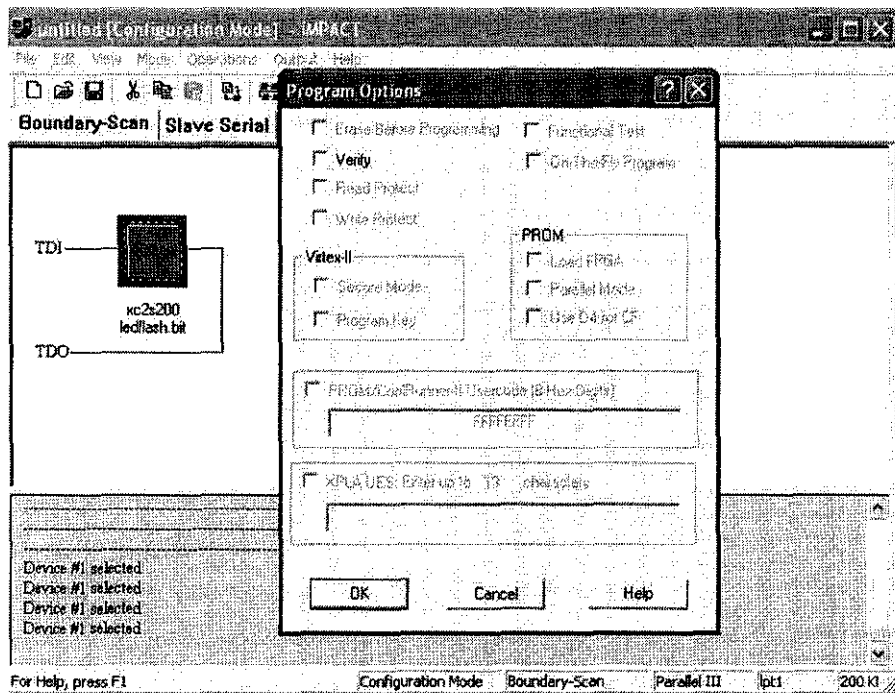


Figure H.7 iMPACT flow 4

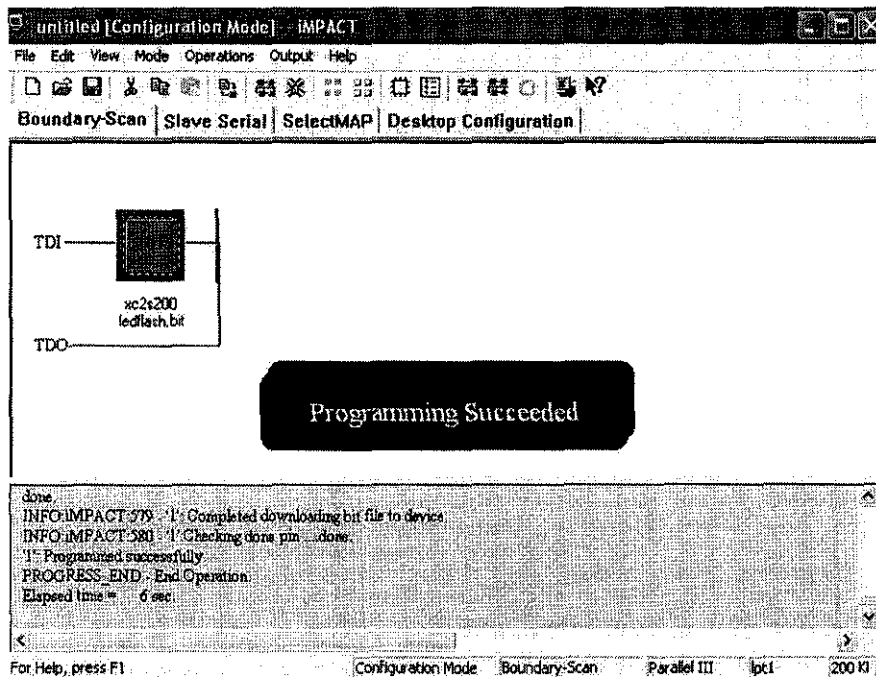


Figure H.8 iMPACT flow 5

If you have downloaded the LEDFLASH.rbt example file, the LED on your BED-SPARTAN2+ board will be flashing at a rate of about 1.4Hz.