Development of Stack Based Central Processing Unit for a FORTH Computer Using FPGA

By

KENNETH WONG FATT KONG

FINAL YEAR PROJECT REPORT

Final Dissertation

Submitted to the Electrical & Electronics Engineering Program
in Partial Fulfillment of the Requirements for the Degree
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

DECEMBER 2009

Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

Development of Stack Based Central Processing Unit for a FORTH Computer Using FPGA

by

Kenneth Wong Fatt Kong

A project Final Dissertation submitted to the
Electrical & Electronics Engineering Program
Universiti Teknologi PETRONAS
in partial fulfillment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved:

_____	_____
Dr. Yap Vooi Voon	Mr. Patrick Sebastian
Project Supervisor	Project Co-Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

December 2009

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.

_____

Kenneth Wong Fatt Kong

ABSTRACT

This is the Final Dissertation for Electrical & Electronics Engineering Bachelor Degree Final Year Project (FYP). The title for this FYP is “Development of Stack Based Central Processing Unit for a FORTH Computer Using FPGA”. This project is based on the design by a previous FYP student, Aaron Tang Shen Lee with his title, “Development of a Stack-Based Centre Processing Unit (CPU) using TTL Logic”. Using the same stack architecture and FORTH programming language, this CPU is to be implemented using FPGAs instead of fully TTL. Besides, this project will make reference to the FORTH computer, Mark 1 built by Andrew Holme, just as the previous project did. This Final Dissertation will contain the progress on the implementation of the stack-based CPU into FPGA. The achievements and obstacles arise while completing this project will be recorded in this report.

TABLE OF CONTENT

LIST OF FIGURE

LIST OF TABLES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
AMD	Advance Micro Device
ATX	Advance Technology eXtended
CISC	Complex Instruction Set Computers
CPLD	Complex Programmable Logic Device
CPU	Centre Processing Unit
FIG	Forth Interest Group
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
I/O	Input / Output
ISA	Instruction Set Architecture
ISR	Interrupt Sub-Routine
LIFO	Last in First Out
RAM	Random Access Memory
RISC	Reduced Instruction Set Computers
ROM	Read-Only Memory
VLSI	Very Large Scale Integration
VM	Virtual machine

CHAPTER 1
INTRODUCTION

1.1 Background of Study

Computer architecture is the selecting, interfacing and interconnection between hardware and software of a system. It is generally consisting of the Centre Processing Unit (CPU), the I/O part and Memory [1]. **Figure 1** shows the interconnection between the major parts of computer architecture.

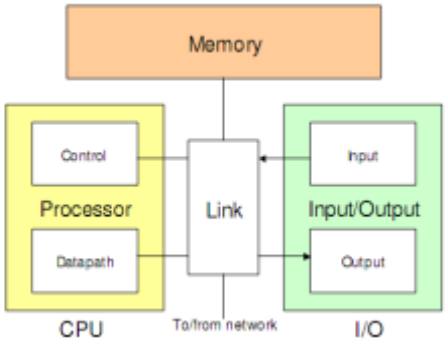


Figure 1 : Key parts of digital computer architecture (figure from [1] page 44)

A CPU design and architecture distinguish one from another. This project explores a different type of CPU architecture from the widely used register-based CPU designs – the stack-based CPU. Stack-based CPU is not as popular as registry-based CPU in primary data handling for the reason many find stack a little complex to handle. However, stacks have advantages of their own.

Programming language especially assembly level language are majority processor based. Hence syntax of one assembly language maybe specific to a particular CPU designs.

The stack-based CPU that this project explores has architecture oriented for implementing a stack-oriented, reflective programming language – FORTH language. FORTH is a structured stack-based programming environment and the language itself is type check free. Hence it is said to be reflective where one can expand the language itself.

On top of that, this project is based on a FYP project by Aaron Tang, which studied the same FORTH Computer architecture, Mark 1 by Andrew Holme but implemented using fully TTL chips – the Mark 1 Clone. The objective of this project is to study the feasibility of implementing the Mark 1 Clone using FPGA. At the end of this project, a hybrid version of Mark 1 using FPGA and TTL is developed for FORTH Computer– Mark 1 FPGA.

1.2 Problem Statement

FORTH language or stacked-based computer system is an alternative type of computer architecture. It has advantages of its own. However, there are engineers who are not familiar with the advantages, design and implementation of such computer architecture. It is a bigger challenge to implement a stack-based computer using FPGA.

1.3 Objective

It is envisaged that the following is expected to be achieved.

- Implementation of FORTH computer in FPGA form
- Study the feasibility of CPU design using FPGA and its interaction with TTL device.

1.4 Outline of Report

This report will consist of the following chapters.

Chapter 2 Literature Review And Theory contains the literature review of computer system architecture, stack based machines and also FORTH language. It provides an overview of the mentioned topics.

In Chapter 3 Methodology, an outline of the method used to implement this project will be discussed. The list of tools used in completing this project will also be described as well as providing some description on the work completed during the duration of this project.

As for Chapter 4 Results And Discussion, the achievement of this project will be reported and discussed. The obstacles and challenges faced throughout the duration of this project will also be discussed.

Lastly Chapter 5 Conclusion And Recommendation will provide some recommendation for future improvement and development and also a recap of this project.

CHAPTER 2
LITERATURE REVIEW AND THEORY

2.1 Computer System Architecture

The architecture of computer system can be divided into a few operation structures. These operation structures rely on one another to perform a required computational task. This sub-chapter will discuss each the typical structure of a computer system.

2.1.1 Data Path

Data path is the part of a computer system which manipulates and control the data flow in the system according to the instruction's definition. A simple computer will have a simple linear data path which is controlled by the control path for execution of an instruction set for the path. However, a more complicated computer will have multiple data path branch out and perhaps interconnect with each other to perform an operation or instruction. [1]

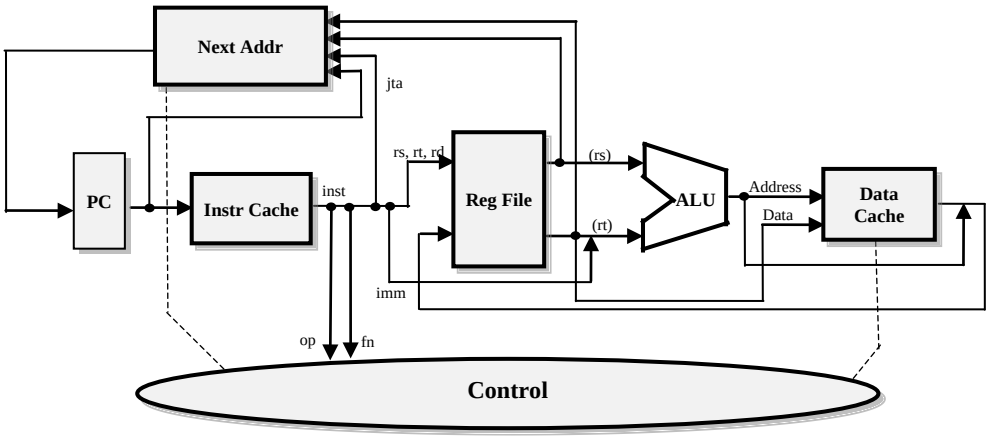


Figure 2 : Example of data path (figure from [1], page 246)

Figure 2 show an example of data path, which is the instruction execution unit of the MicroMIPS architecture [1]. A simple single-cycle data path is basically consisting of a program counter, instruction files, register files, ALU and data cache or memory. The data cache forms the data buses of the data path while the remaining components form the functional units of the data path. [1]

2.1.2 Control Path

The control path is responsible for controlling the data flow according to given instruction. Control of data flow is needed so that the system execution could flow from a completed instruction to the next instruction sequence. Control path does not have memory. It forms the control signals as a function according to certain bit from the instruction code. [1]

The control signals can be executed easily for single-cycle implementation. However, most of the time the system instructions will requires a certain resource to be used for more than once. Such instructions required a more complex execution and implementation called multi-cycle implementation. [1]

2.1.3 Instruction Set Architecture (ISA)

Instructions are words of the machine language used by a machine. Instruction sets are the language and word vocabulary. Instruction set architecture is the vocabulary of words together with parts of the machine which functions to provide guidance to both data path and control path to perform a task. [1]

Understanding of ISA is important for a computer system engineer and programmer to produce fast, compact and correct program for a machine. [1] This is because the operation of data path and control path of a given system architecture rely on a given instruction.

Some example of instructions used in basic operation includes arithmetic, logic instruction, data instruction and control flow instruction. These instructions are in a

specific set of operational codes (opcodes) which means machine language, the native commands implementation by the computer design. [2]

However, ISA is distinguished from the term micro-architecture which simply means the data path and control path of a CPU design. Reason being a different CPU design may share a common instruction set. Example could be taken from the Intel Pentium and AMD Athlon. Both of the CPU architecture is different but sharing the same ISA, which is the x86 32-bit instruction set. [2]

2.2 FORTH

FORTH is a structured, stack-based computer language and programming environment. [3] It can be implemented on a virtual machine like Java VM. It is normally implemented using indirect threaded code, which is a form of programming code. This makes a compiled FORTH extremely compact. FORTH has a compiler and command-line interpreter besides supporting structured programming. Besides having compiler that is free to use (for example Fig-FORTH), FORTH is simple, elegant and compact. [4]

FORTH is a procedural, stack-oriented and reflective programming language without type check. It features both interactive execution of command and the ability to compile sequences of commands for later execution. Earlier versions of FORTH compile threaded codes but later versions compiler generate optimized machine code. [3]

FORTH language is said to be a reflective programming language because of the ability to extend the language as a whole. Reason being the core language of FORTH has virtually no syntax. As you extend the language, you are actually defining your own syntax. [4]

As FORTH is a stack-oriented program language, most data or parameters passing are done completely on the stack. Therefore, there is no need to define a lot of variables. However, it is recommended to comment the stack effect of every

FORTH word, because in FORTH, you are actually naming the actions and not the data itself. [4]

FORTH is originally developed for small embedded control miniature computers. FORTH is implemented on many major processors manufactured. [5] However, FORTH lack the support from large industry for it is unique and the acceptance of it has to be done from basic fundamental. Hence, it is not as popular as other programming language such as C, C++ as well as other similar level languages. [6] Nevertheless, it is still being used in some embedded system especially in space application and also boot loader such as Open Firmware. [3]

2.3 Stack Machine

Stack-based hardware supports Last in First out (LIFO) stacks is being used on computer since the late 1950s. Stack is originally designed to increase the execution efficiency of high level languages such as ALGOL. However, this approach has not gain popularity and in favor of designers and hence is being used only as secondary data handling structure in most computers. Many designers prefer to use register-based machine for their primary data handling due to the reason some finds stack rather dismay compared to registers. [7]

Emergence of VLSI processors brings forward the question on conventional methods of computer designs. CISC and RISC instruction sets evolves to incorporate the advancement of VLSI processors. With this, stack machines are being considered as an alternative design style. VLSI allows new stack computers to attain impressive combination of speed, flexibility and simplicity with their features. [7]

With VLSI, stack machines could offer lower processor complexity than CISC machines and lower overall system complexity than either RISC or CISC machines. These good performances are achieved without complicated compilers or cache control hardware. The first successful application is in the area of real-time embedded control environment, where they outperformed other system design. [7]

Stack machine uses lower raw resources but produce superior performance for a given price in most of the programming environment. It shows great performance while executing logic programming language such as Prolog, functional programming language such as Miranda and Scheme, and artificial intelligence research language such as OPS-5 and Lisp. [7]

2.3.1 What is Stack?

Stack is also known as LIFO stacks or “push down” stacks. It is conceptually the simplest way to save information in a temporary storage location for common computer operations such as mathematical expression evaluation and recursive subroutine callings. [7]

LIFO can best be described using cafeteria tray example. Consider a spring-loaded tray dispenser. Assuming each try has number on it and is being loaded in from the top one after another. Each of the loaded tray will rest on the already loaded trays with the spring is being compressed to make room for more trays. **Figure 3** illustrates the loading of tray with number 42, 23, 2 and 9, with 42 loaded first and 9 loaded last. [7]

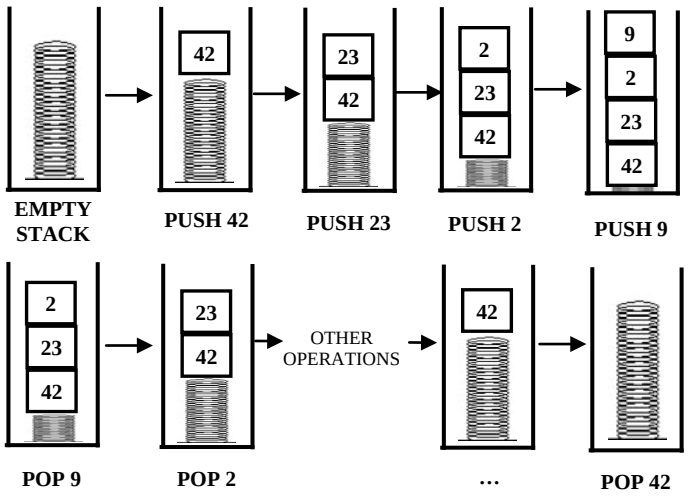


Figure 3 : Example of LIFO stacks operation (from Philip Koorman, section 1.2 [7],)

As illustrated in **Figure 3**, the ‘Last In’ tray is number as 9 and the ‘First Out’ tray is also tray numbered 9. The next tray that will be removed after 9 will be 2 and so on. However, if more trays were to be added at this point, they had to be removed from the stack before the very first tray, tray 42 could be removed. Any pushes and pops on the top will retain tray 42 in illustration on the bottom. The stack would only be empty again after the tray 42 is being popped from the top of the stack. [7]

2.3.2 Advantages of Stack-Based Machine

Stack machines are more efficient in running certain type of program than register-based machines, in particular modularized program. Stack machines are also simpler than other machine besides providing good computational power with little hardware. Real time embedded control application favor the use of stack machines. This is because it requires a combination of small size, high processing speed and excellent support for interrupt handling that can only be achieved with stack machines. [7] Following are some highlight of stack machines from the point of view of someone who had made a living with stack machines. [8]

- Stack processors do not need to pipeline ALU and operands because operands are immediately available in the top of stack buffer registers.
- Only about 16 deep on-chip stack buffers are needed and spilling can be done by stack overflow interrupts hence reduce cost for interrupt-driven overflow.
- Context switching for interrupts needs only zero time, whereby no registers is needed to be saved. ISR values are placed on the top of the stack and are being clean off when done.
- Program size makes a lot of difference in embedded control. Stack computer small program size can be achieved with compact opcodes, reuse of short code segment and implicit argument passing without subroutines.

2.3.3 Important of Stack-Based Machine

Theoretically, stacks are important because they are the most basic and natural tool that can be used in processing a well structured code. LIFO stacks machine are required to compile computer languages and maybe the translation of natural

languages. A computer with support for stack structure will probably execute application requiring stacks more efficiently than other machine. [7]

Compilers for stack machines are easier to be written simply because they have fewer exceptional cases to complicate a compiler. This made some people says that programming stack machines is easier than conventional machine and stack machine program run more reliably than other programs. However, running compiler require certain percentage of machine resources. Therefore, building a machine with efficient compiler is important. [7]

2.4 Stack-Based Machine and FORTH

As mentioned, FORTH is a structured, imperative, stack-based programming language, which runs on a stack-based computer. LIFO stacks, which is also known as “push down” stacks is the key element for a stack machine and also FORTH. Combining the advantages of FORTH programming language and the stack-based machines, a high performance embedded system could be developed and perhaps achieve low cost with different type of CPU construction technology.

2.5 Chapter Summary

This chapter provides a review on computer system architecture, stack machines and FORTH language. We could generally divide a computer system into three (3) major components, namely data path, control path and instruction set architecture. A stack based machine uses LIFO stack for data handling. FORTH language on the other hand, is a structured, stack-based computer programming language, which required a stack machine for implementation.

CHAPTER 3
METHODOLOGY

3.1 Procedure Identification

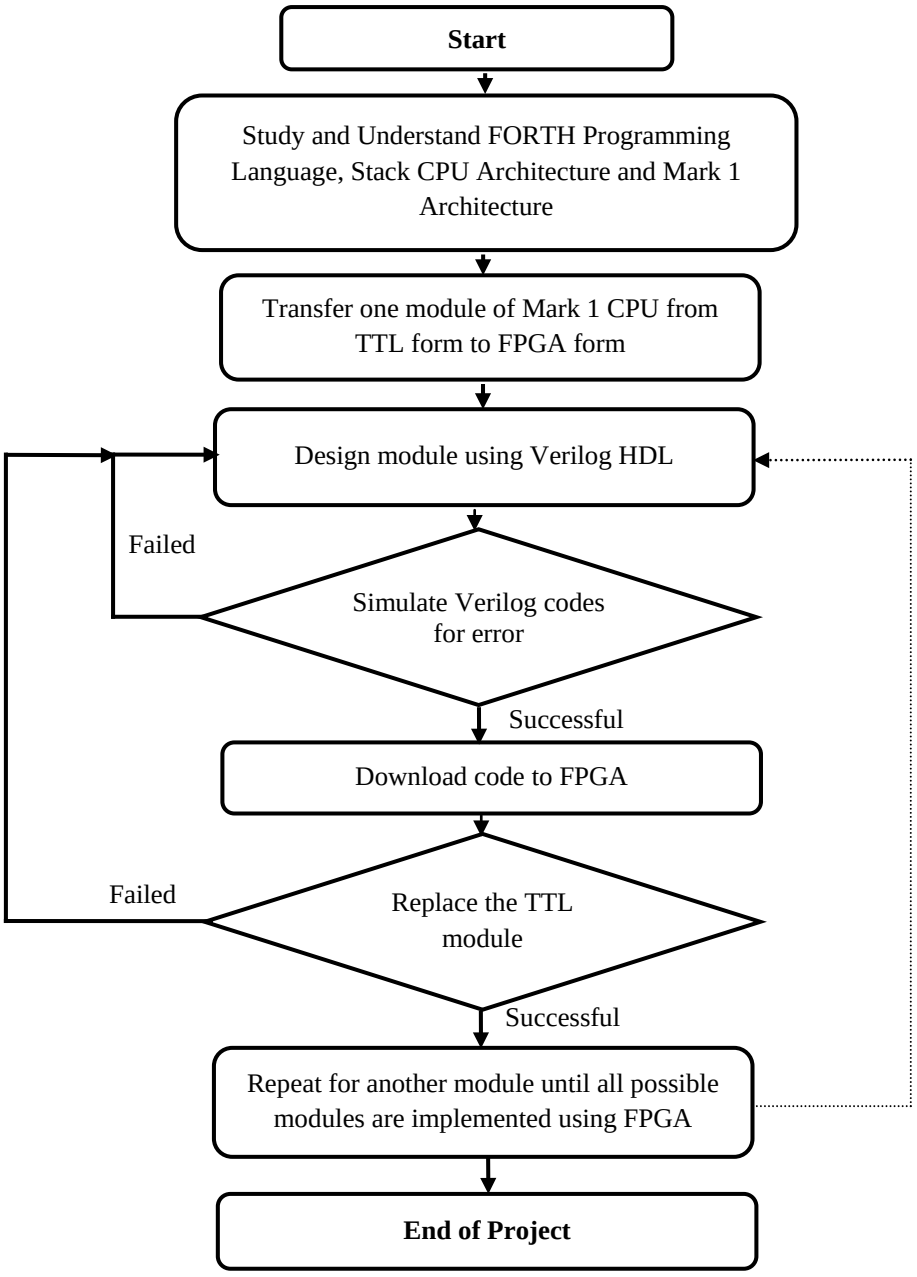


Figure 4 : Flow Chart of Project

The objective of this project is to implement Mark 1 Clone, a fully TTL version FORTH computer using FPGA and study the interfacing capability of FPGA and TTL. Hence, a modular approach is used in this project.

Implementation is done one module at a time as outlined by the flow chart in **Figure 4**. Proper study of module design is done before design is made with Verilog code. Verilog code is then downloaded into the FPGA and tested for compatibility with TTL modules before another module is designed in Verilog. Module which could not be fitted into the FPGA will be implemented using TTL, especially module involves memory chips.

3.2 Tools

The main tool in this project would be the FPGA chip that the CPU will be build into. Altera University Program 2 (UP2) board and other electronics components and sockets are among some essential tools. Verilog HDL programming language will be the main programming language used to program the FPGAs.

3.2.1 Hardware

Computer with connection cable to the FPGA chip used is the hardware involved at the implementation stage. Altera UP2 board that houses the FPGA chip will be the hardware required. The MAX7000 CPLD and FLEX10K FPGA on the UP2 board will be used for this project.

Euro-cards and IDC connectors are among the electronic components that are needed for interfacing and connecting the UP2 board to external components, especially to the existing TTL CPU.

Besides, instrumentation tools such as oscilloscope, logic analyzer and digital multi-meter are also used to aid analysis procedure. The tools eases troubleshooting procedure in this project.

3.2.2 Software

The software that will be used for current stage of implementation would be Altera ModelSim and also Altera Quartus II for HLD simulation, compilation and also programming the compiled codes into the FPGA.

3.3 Work Completed

There are nine (9) modules build on euro cards that forms the Mark 1 Clone FORTH CPU. **Table 1** show the modules that forms the Mark 1 Clone CPU which will be attempted to be implemented using FPGA.

Table 1 : Modules in Mark 1 Clone

1.	Instruction Decoder
2.	Diode ROM
3.	Instruction Pointer Index
4.	Address Pointer Index
5.	ALU
6.	Stacks
7.	Memory
8.	Microcode Sequencer and Power ON reset
9.	I/O

Several steps are taken to examine the influential factors that could affect the success of implementation. After ensuring all these factors were taken care of, the implementation of TTL modules in FPGA forms started with the sequence of listing

as of above. The success of first module implementation will serves to guide the implementation of the remaining modules.

3.3.1 Testing of UP2 Board

The first step of implementation of the FORTH Computer System in Verilog form is to perform hardware check, testing and verification for error. Therefore, the first task upon recipient of the UP2 board is to write a test program to be run on the UP2 board. This is to ensure that the board is functional before the actual program is loaded into it. The test program code is attached on Appendix IV . Result on testing of the UP2 board will be discussed in the next section, Chapter 4.1 .

3.3.2 The Power Supply

Besides this, there is a need to build a power supply for the existing stack-CPU built by the previous FYP student. Two different power supplies are being built for the Mark 1 Clone. One of the power supplies was modification of a variable 12VDC power supply and another is from the desktop computer ATX power box. However, only the ATX power box is capable of supplying sufficient current for the operation of Mark 1 Clone. Detail of design is discussed in Chapter 4.2 .

3.3.3 The Expansion Card

An expansion card is made using a Euro card to interface the Mark 1 Clone with UP2 board for design verification on the Verilog codes functionality. In addition, this board also functions to allow the FPGA connect to the existing TTL modules during the implementation.

In order to make the expansion board flexible and universally connectable, more work had been done in adding inter-changeable connectors on the board. This allows the assigned FPGA I/O pins to be connected to any of the pins on the Mark 1 Euro card backbone. Sub-section “Expansion card” under Appendix V contains the photo of expansion card mentioned.

Besides, an expansion card Version 2 also being built on later stage to accommodate more I/O pins from the UP2 board. Reason being one (1) expansion port from UP2 FLEX10K FPGA provides only forty-two (42) programmable I/O pins. Hence, to allow connection from the FPGA to sixty-two (62) I/O pins of the Mark 1 Clone back panel, two (2) expansion ports is needed.

3.3.4 TTL Module in HDL Design

The next task after hardware examination would be designing the Mark 1 modules in Verilog form for implementing in the FPGA. The sequence of implementation would be according to the sequence of **Table 1**. The modules are being redesign and coded into Verilog HDL according to sequence. The Verilog designs of the modules are attached in Appendix III . A top module is used in to combine the individual modules before interfacing to the TTL modules during the testing of multiple modules.

3.3.5 Interfacing and Replacing TTL Module with FPGA

After simulating and verifying the designed module using Altera ModelSim and Altera Quartus II software, the respective TTL module is ready to be replaced. Interfacing is done using the expansion card that is tested with error free.

During the replacing process, the respective TTL module will be removed from back panel and the Verilog design of the respective module will be downloaded into the FPGA. After connecting the FPGA to the back panel using the expansion card, the Mark 1 Clone is power up to verify if the system is being replaced correctly.

However, the success of implementation was not as expected every time the design is being implemented. Several analyses and troubleshooting are done in the process of implementation. Chapter 4.4 and 4.5 records the analysis and discussion of the troubleshooting result.

3.4 Chapter Summary

This chapter discusses the methodology used to implement this project. A flow chart (**Figure 4**) is used to illustrate the process of implementation. Besides, the achievement of this project is recorded under Chapter 3.3 "Work Completed".

CHAPTER 4
RESULTS AND DISCUSSION

4.1 Test Result of the UP2 Board

Several versions of Verilog code is designed and programmed to test the UP2 board. The Verilog codes used to test the board is attached in Appendix IV .

Two UP2 board were used but only one was tested working properly with the test code on the MAX7000 CPLD chip. The FLEX10K FPGA chip did not respond properly to the test. However, when the FLEX10K chip is revisited, it responded to a test code which uses input driven by the MAX7000 chip.

After carrying out more testing with the FLEX10K FPGA, using test codes attached, it is concluded that the FPGA chip is functional. Further test shows that the on-board oscillator is accessible but there are problem deriving clock signals from FPGA. This will be further discussed in Chapter 4.3.1 , 4.5.1 and Appendix VI .

Besides, under Appendix V "Testing stage" contains pictures taken during testing of the FPGA with various test code attached in Appendix IV .

4.2 Design of the Power Supply

Two different power supplies were utilized for Mark 1 Clone.

The first is a DC power supply that could supply power varying from 10VDC to 15VDC. Therefore, a voltage regulator circuit is made to limit the supply current to 5VDC. However, it is not able to supply sufficient power required by the TTL Mark 1 Clone and not being used in this project.

The second is the computer power supply unit, an ATX power box. This power box could provide DC power of 12V and 5V. A simple circuit is made to switch the ATX box ON and OFF while tapping the 5VDC supply for the Mark 1 Clone. Mark 1 Clone is tested working properly with the ATX power box and hence it shall remain the power supply for this project.

4.3 Design and Simulation

Each module design in Verilog code went through two level of simulation. The code is firstly designed and compiled using Altera ModelSim software to simulate and verify the functionality of the code. After the code is verified to be functioning correctly, it is transferred to Altera Quartus II software for simulation, download configuration and timing analysis.

4.3.1 Instruction Decoder Module

The function of Instruction Decoder module is to decode the microcode of the system to various control signals. The Verilog code of Instruction Decoder is compiled in ModelSim and functional simulation is performed on it to study and verify the codes functionality before being transfer to Quartus II for implementation into the UP2 board.

First attempt of simulation fails as some of the signal used in the design shows error. After several corrections, the code shows that the Instruction Decoder written is working well and all output signals are responding to changes in input. **Figure 5** shows the successful result from functional simulation on the written code using ModelSim.

This functional simulation waveform is cross-checked with the schematics diagram of the Instruction Decoder. This is done by checking the output signal in accordance to some sets of input signals. After cross-checking, the output waveform does show that the Verilog version of the Instruction Decoder performs the necessary

decoding as per the TTL version. This simulation waveform serves as the base of comparison for the timing simulation from Quartus II later for performance check.

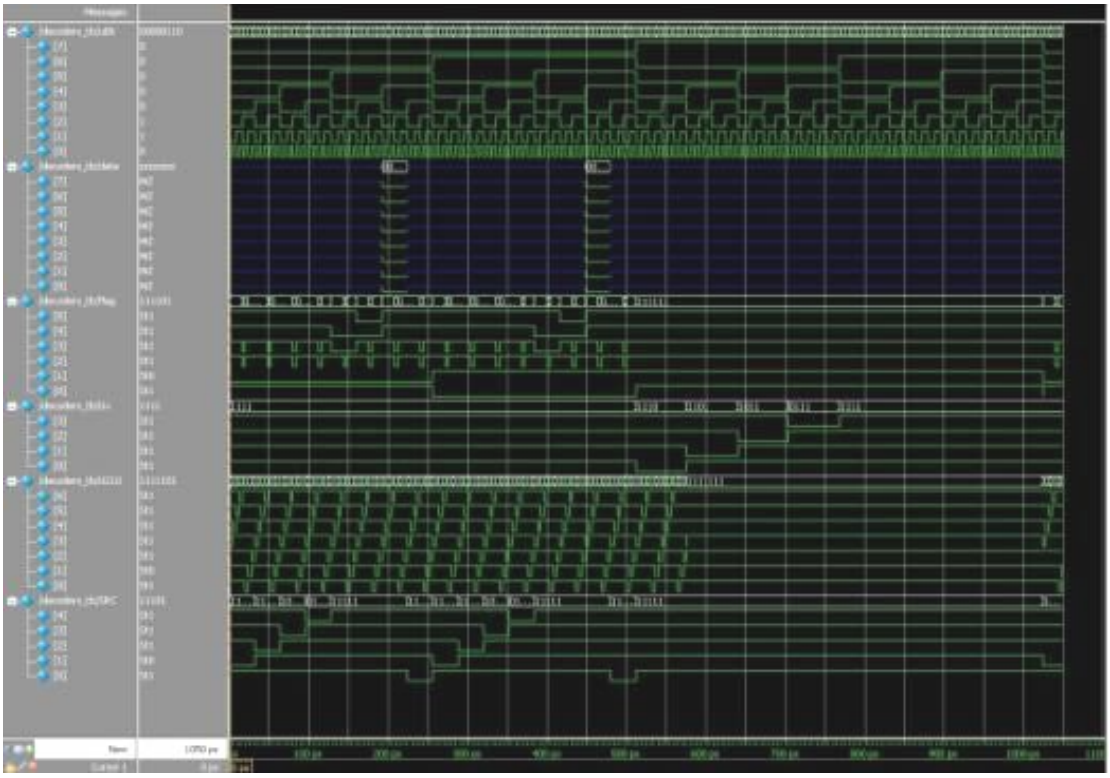


Figure 5 : Functional simulation waveforms for Instruction Decoder module

However, when the code is being simulated in Quartus II on timing simulation, the clock signal did not produce the expected result. The code written is verified that it could not be implemented in real world. Attached code on Appendix III shows the modified code without clock signals, which the final code that works.

Further investigation confirmed that FPGA is not capable of deriving clock signals for a digital system. Hence, the original clock design using TTL will be used to provide the clock signal for the system. More discussion can be found in Chapter 4.4.1 and “Clock Signal Derivation” section under Appendix VI

Figure 6 shows the timing simulation result using Quartus II. Verification on timing simulation result shows that the written code could function on the chosen

FLEX10K FPGA chip. After assigning each I/O pin, the program is downloaded to the chip and tested with a test unit designed for the instruction decoder module.

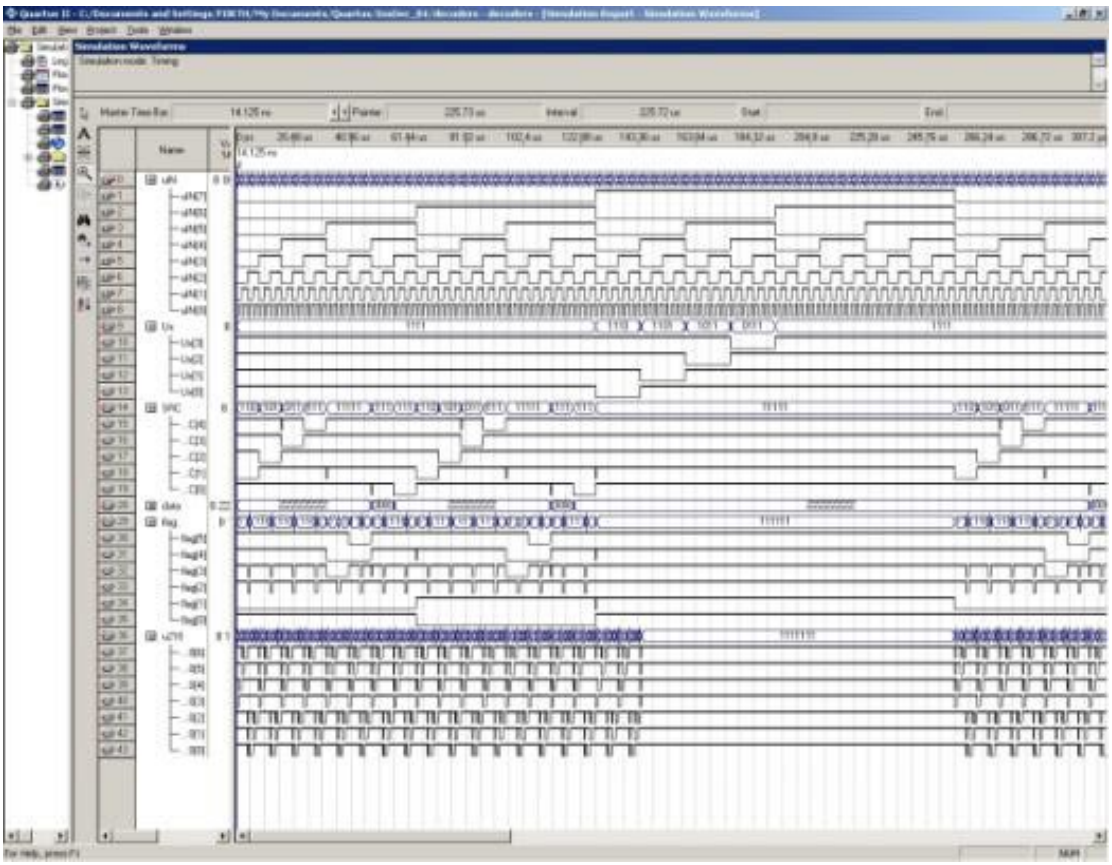


Figure 6 : Timing simulation waveform for Instruction Decoder module

After satisfied with the code verification on the FLEX10K FPGA, the FPGA is then connected to the expansion card, which had been wired the designated I/O pins of FPGA to the correct Mark 1 signal buses. The Verilog version of Instruction Decoder is then put into full test.

4.3.2 Diode ROM Module

The Diode ROM module basically functions to decode the function selection signals for ALU. Hence, the design of the module is similar to a decoder. This made this module an easier one to be designed.

After designing the module in code, the code is simulated using ModelSim. **Figure 7** shows the functional simulation waveform for Diode ROM module. The simulation result is satisfying and hence the code is loaded into Quartus II for next step of implementation.

Figure 7 : Functional simulation waveform for Diode ROM module

In Quartus II timing simulation is carried out on the codes. Besides, it also tested for implementation feasibility with the place and route function of Quartus II. **Figure 8** shows the timing simulation waveform for Diode ROM module.

Both simulation waveforms suggest that the module is functioning. Waveform patterns are compared with the module design for further verification and confirmation.

After completing the test and verification, the code is downloaded into the FPGA chip and put into full test.

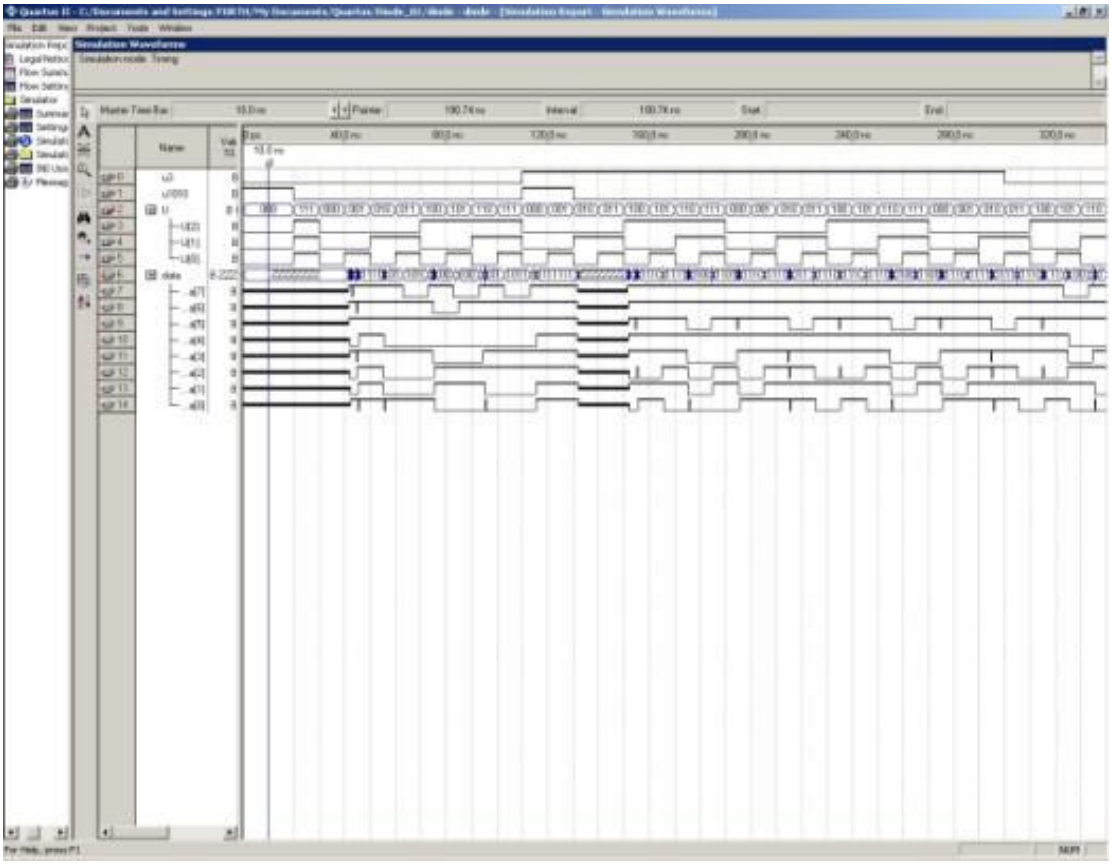


Figure 8 : Timing simulation waveform for Diode ROM module

4.3.3 Index Pointer Module

Both Instruction Pointer Index and Address Pointer Index are of the same design. The difference between them is on some of the input pins, namely u210, SRC

and flag signals. They function to index and point the address location of current instruction execution for both memory addresses and instructions location.

The Index module is designed with such flexibility, where by the different input pins connection is made general. This could ensure that the module can be used for both modules without the need to modify the code but just a little change with the connection assignment.

As this module involves INOUT ports, functional simulation could not be done using ModelSim without separating the INOUT ports to individual input port and output port. Hence, the simulation result for this module could not accurately shows that the module designed is working perfectly.

However, the codes are downloaded into the FPGA after successful compilation and simulation in Quartus II. The codes is then put into full test after the expansion card is wired correctly for replacing the respective module, be it Instruction Pointer Index or Address Pointer Index.

4.4 Interfacing Troubleshoot and Discussion

Troubleshooting on interfacing is done during the implementation of the first module, the Instruction Decoder module. The unsuccessful implementation prompted the need to re-examine some of the factors that could and may affect the implementation results.

As the Mark 1 Clone did not respond when the FPGA version of Instruction Decoder is implemented, investigation was carried out:

- Short circuit test on the expansion card, as the possibility of occurrence is very high with the number of micro wires soldered on it
- Voltage drop on the system and each module to verify the system is running normal
- Signal waveform check using oscillator to check for faulty output

Before the mentioned troubleshooting steps are taken, the system clock has to be taken care off first. This is necessary to ensure that the correct clock signal is supplied to the system.

4.4.1 System Clock Derivation

System clock affects the performance of the entire system. Hence it is needed to ensure that the system is receiving the required clock waves. As clock signals could not be derived from FPGA, the system clock is tapped from the original TTL version of Instruction Decoder. Minor rewiring is done to use the clock signal from the TTL module. This is only needed when testing the Instruction Decoder module.

4.4.2 Short Circuit Test

Short circuit test is being carried out on full-scale on the expansion card. Each pins and wire are tested with all adjacent pins. All points with possibility to short circuit are tested. After rectifying short circuit points on the card, it could be concluded that the system is not affected by short circuit and signal sending on the wrong bus.

4.4.3 Voltage Drop Test

Voltage drop on each module of the TTL version system is measured to assist troubleshooting. **Table 2** shows the figure of measurement on voltage drop for investigation.

Table 2 : Voltage drop measurement

Module	**Voltage Drop (V)****
Instruction Decoder	0.04
Memory	0.03
Index M@W/A	0.08
Index M@IP/PC	0.08
I/O	0.03
Stacks	0.05
ALU	0.06
Diode ROM	0.00

***Note: approximation of ±0.01V applies*

A total of 0.37V drop is measure on the TTL system with a supply of 5.00V. The measured voltage from VCC to GND reads 4.57V. This voltage level is just sufficient for the operation of TTL as their minimum operating voltage is 4.5V.

The current consumed by the TTL Mark 1 Clone is 1.10A, which is rather high. This explains the reason for the overload of the 12VDC converted power supply. The added voltage regulator could not regulate current above 1.0A for 5VDC.

Studying the pattern of voltage readings with and without interfacing card as well as the FPGA module, short circuit within the system is not likely the reason for the system not being able to run.

However, this serves to aid the observation for reducing power consumption of the system with the use of FPGA. Power consumption of a system is a crucial part of design as technology is moving towards low power and efficient system.

4.4.4 Signal Waveform Test

It is hypothesized that the timing waveform and signal level of signals from TTL to FPGA and FPGA to TTL leads to the investigation to test the signal waveforms. This test is done with the aim to verify that there is no signal timing issue between FPGA and TTL.

The signal waveforms are observed using oscilloscope and the signal pattern is observed using Logic Analyzer. However, as there are no consistent signal waveforms, the test could not be properly verified and concluded.

On the other hand, observation on the critical path and propagation delay shows that there are sufficient setup times for the signal. The worse case propagation delay for FPGA would be around 40ns (nano-seconds) while the clock period is 1us (micro-seconds). Hence signal violation is not likely to happen.

More observation will be made from time to time to examine the signal waveforms between FPGA and TTL for future studies.

4.4.5 Other Test

In addition, a test circuit to investigate the correctness of the module function is build to cross-check FPGA version with the TTL version. This test circuit is built based on the functionality of the Instruction Decoder module.

Test result from the circuit shows that the FPGA version functions just as the TTL version. Both of the versions decode incoming signals correctly and identically. This study shows that FPGA can be designed to performs and function just as TTL does. Besides, this test could verify the functionality of FPGA module design on independent basis.

4.5 Implementation Results and Discussions

Various problem faced when the modules are implemented using FPGA. As the interfacing method had been tested feasible with no errors, troubleshoot to overcome failure of implementation lies on the module design as well as pin assignment and connection.

4.5.1 Instruction Decoder Module

Appendix VI records some also findings on design notes and application using Verilog HDL and also FPGA. However, the studies are based on Altera product architecture and software. Slight different may appears with product from other company such as Xilinx etc.

After several isolate and test studies and implementing the clock derivation section using TTL, the design module is finally working properly. Some fine tuning and repeated testing is done with the design to ensure that the system is not operating on intermittent state.

The success of implementing this module marks another steps of possibility and feasibility in implementing TTL design using FPGA.

4.5.2 Diode ROM Module

The Diode ROM design and structure is less complex compared to the other modules. Hence, the module was successfully interfaced with TTLs without much complication.

4.5.3 Index Pointer Module

Index Pointer module is mainly form by counters. Hence the counter design is the key success of implementing this module using FPGA. Besides, this module also involves INOUT ports. The finding on design using INOUT ports is recorded in Appendix VI under section “INOUT Port Implementation in Verilog HDL”.

Several intermittent issues arise during the implementation and testing stage but it was taken care of using proper buffers design. Codes in Appendix III records the final working design of the FPGA version of this module.

4.6 Design Limitation

There are several limitations in implementing some modules of Mark 1 Clone using FPGA. This section discusses the difficulties faced.

4.6.1 Memory Capacity

Memory is the data storage location for a system. Three (3) of Mark 1 Clone design is memory based, namely Stack, Memory and Microcode Sequencer.

Stack module forms the stack data memory of Mark 1 Clone hence requires a large number of RAM space. Memory module contains the boot data for Mark 1 Clone and serves as the primary data storage of the system hence requires a great capacity of RAM and ROM. Microcode Sequence stores the microcode of the system in ROM.

Table 3 shows the memory capacity requirement of each of the mentioned modules.

Table 3 : Memory requirement of Mark 1 Clone modules

Module	**Memory Chip Required**
Stack	2 x 2KB RAM (6116)
Memory	3 x 8KB RAM (6264) 1 x 8KB ROM (2764)
Microcode Sequencer	1 x 8KB ROM (2764)

The FPGA used in this project, FLEX10K is having only a memory capacity of 18,432-bits, which is insufficient for each of the modules. The total memory required by the three mentioned modules is close to 44-kilo-bytes or 44 x 8-kilo-bits.

Hence, Stack module, Memory module and Microcode Sequencer module are not implemented fully using FPGA. Instead, TTL chips are used to implement them.

4.6.2 Complexity of Customization

ALU module performs the execution of arithmetic and logical operation of an instruction. This module could not be implemented using FPGA successfully for he complexity of the module design to be customized using FPGA. Hence, this module is being implemented using TTL chips.

The I/O module functions to provide interfacing between Mark 1 Clone with Personal Desktop Computer through serial communication. Windows Operating System in Personal Desktop Computer communication with Mark 1 Clone serially using the application ‘Hyper Terminal’. Serial interfacing using FPGA is another field of application. Hence this module is implemented using TTL chips for this project.

4.7 Chapter Summary

This chapter discusses the troubleshooting and FPGA modules implementation results in detailed. In summary, there are several modules of Mark 1 Clone could not be implemented using FPGA, due to memory and complexity limitation. However, there are also modules of Mark 1 Clone that is implemented successfully using FPGA. Troubleshooting methodology and approach is essential in determining the fault and errors in a design.

CHAPTER 5
CONCLUSION AND RECOMMENDATION

5.1 Recommendation for Future Studies

In this project, there are some portions of Mark 1 Clone design that could not be implemented using FPGA successfully. More study on Verilog code design techniques and real-time feasible design strategy could help resolve complex design required by Mark 1 Clone.

Besides, in-depth understanding of CPU design modules and functionality could help compact and simplify current FPGA designs. Reduced complexity would reduce the number of logical elements required and hence power consumed.

However, on the mean time, the FORTH Computer developed from this project shall remain as a hybrid of FPGA form and TTL form. A new board is used to reduce the size of CPU design with the elimination of back panel.

5.2 Conclusion

The objective of this project is to study the feasibility of implementing a stack-based CPU using FPGA. FORTH Computer is the stack machine that this project explores. The FORTH Computer is based on Mark 1 design.

There are nine (9) modules in the CPU design. Four (4) modules had been successfully implemented using FPGA, namely the Instruction Decoder, Diode ROM and Index (address index and instruction pointer index) module.

The success implementation of said modules using FPGA shows that TTL design can be implemented using FPGA. However, the key of success lies on the design strategy and also signal interfacing. The success of interfacing TTLs with

FPGA design serves an advantage in this project as some parts of CPU especially memory are easier and more feasible to be implemented using TTL.

Besides, the design of I/O Ports in FPGA will have impact on the overall system functionality. Buffers shall be placed at the OUTPUT port of FPGA when ever applicable to provide protection for FPGA on sink current. Multiplexing signals design on the INOUT port of FPGA is also critical as data signals going in and out of the FPGA will have great impact on the system performance and stability.

In a nut shell, a hybrid CPU system using FPGA and TTL is a feasible design approach. This is proven by the success interfacing of FPGA modules with TTL modules in this project.

REFERENCES

[1] B. Parhami, *Computer Architecture - from microprocessors to supercomputers*. New York, USA: OXFORD University Press, 2005.

[2] Wikipedia. Instruction Set. http://en.wikipedia.org/wiki/Instruction_set

[3] Wikipedia. Forth (programming language). http://en.wikipedia.org/wiki/Forth_(programming_language)

[4] A. Holme. What is FORTH?. http://www.holmea.demon.co.uk/FORTH.htm

[5] J. Philip J. Koopman. A Brief Introcution to Forth. http://www.ece.cmu.edu/~koopman/forth/hopl.html

[6] Introduction to the Forth Programming Language. http://www.angelfire.com/in/zydenbos/WhatisForth.html

[7] J. Philip J. Koopman, *Stack Computer: the new wave*, World Wide Web ed. USA: Ellis Horwood, 1989.

[8] J. Philip J. Koopman. Why Stack Machines. http://www.ece.cmu.edu/~koopman/forth/whystack.html

[9] A. Holme. Mark 1 FORTH Computer. http://www.holmea.demon.co.uk/Mk1/Architecture.htm

[10] S. L. Aaron Tang, "Development of a Stack-Based Central Processing Unit (CPU) Using TTL Logic," Dissertation, December 2006.

[11] L. Brodie, *Starting FORTH*. Prentice Hall, 1981.

[12] The Forth Programming Language - Why YOU should learn it. http://www.hcsw.org/reading/forth.txt

APPENDICES

Appendix I Mark 1 FPGA Specification

(The content of this appendix made reference to [9], [10]*)*

Specification

Technology	Hybrid of FPGA and TTL
Clock	1 MHz
Data Bus	8-Bit
Address Bus	16-Bit
Software	fig-FORTH

System Overview

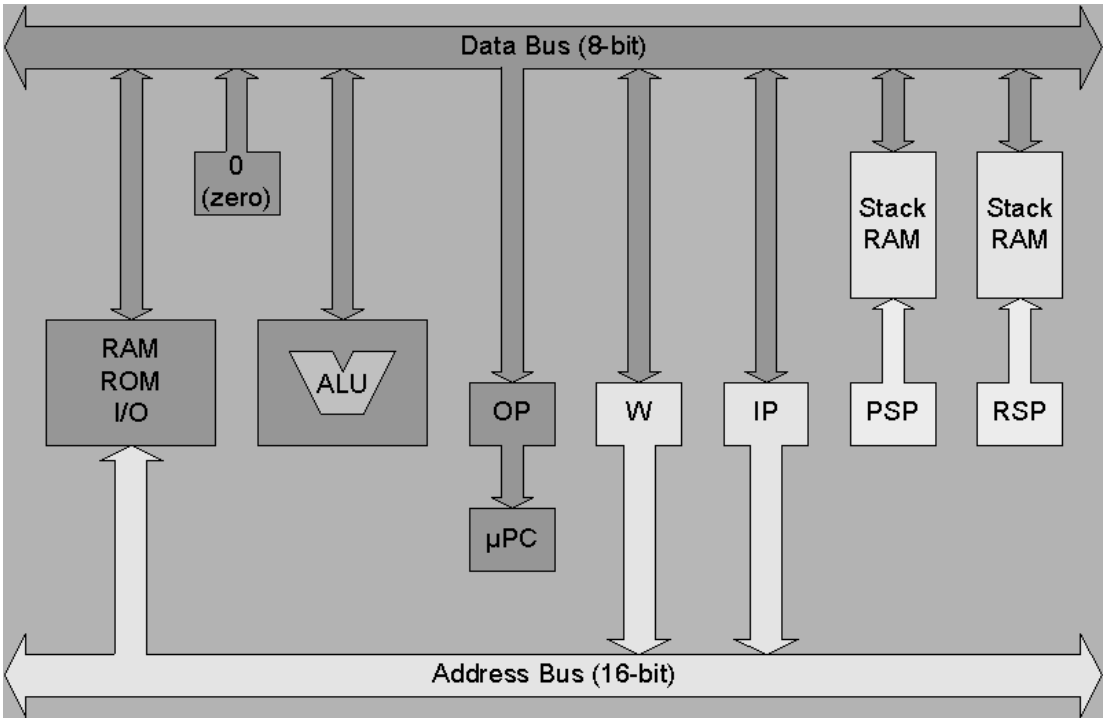


Figure: Mark 1 Computer Architecture [9]

Table: Description of Mark 1 Computer Components

Module	Width (bits)	Description	Comments
ALU	8	Arithmetic and logic unit	The ALU data path is a bottleneck. It takes four clock cycles to load the inputs, set the ALU function, and read the result. This is the least satisfactory aspect of the whole design.
OP	8	Operand register	OP is loaded into the uppermost 8 bits of µPC. The lower 4 bits are reset to zero.
µPC	12	Microcode program counter	
W	16	FORTH Working register	The 16-bit index registers, IP and W, support increment, decrement, and can address memory.
IP	16	FORTH Instruction Pointer	
PSP	8	FORTH Parameter stack pointer	The stack pointers, RSP and PSP, are 8-bit up/down counters feeding the A1-A9 address inputs of the stack RAMs. The least significant address input (A0) selects the upper or lower byte. Logically, the stacks are 16-bits wide by 256 words deep. The FORTH word length is 16 bits.
RSP	8	FORTH Return stack pointer	
Stack RAM	16	Dedicated stack RAM	
0	8	Force 00H on data bus	

µ-Instruction Format

Mark 1 is a micro-programmed machine with ‘vertical’ encoded microcode. The microcode instruction (µ) is only 8-bit wide. This is in contrast with normally used ‘horizontal’ encoded microcode which is wider in bit width and less encoding. Hence, it makes Mark 1 resembles a RISC processor. Following table shows the encoding of Mark 1 µ-instruction.

Table: Mark 1 8-bit µ-instruction (µ) encoding

	µ7	µ6	µ5	µ4	µ3	µ2	µ1	µ0
Move LSB	0	0	Source			Destination		
Move MSB	0	1	Source			Destination		
Decrement	1	0	0	0	0	0	Register	
Disable IRQ	1	0	0	0	0	1	x	x
Increment	1	0	0	0	1	0	Register	
Enable IRQ	1	0	0	0	1	1	x	x
Jump Direct (zero page)	1	0	0	1	Address			
Set ALU function	1	0	1	0	Function			
Jump Indirect (µPC←OP*16)	1	0	1	1	x	x	x	x
Conditional skip	1	1	Test		Distance			

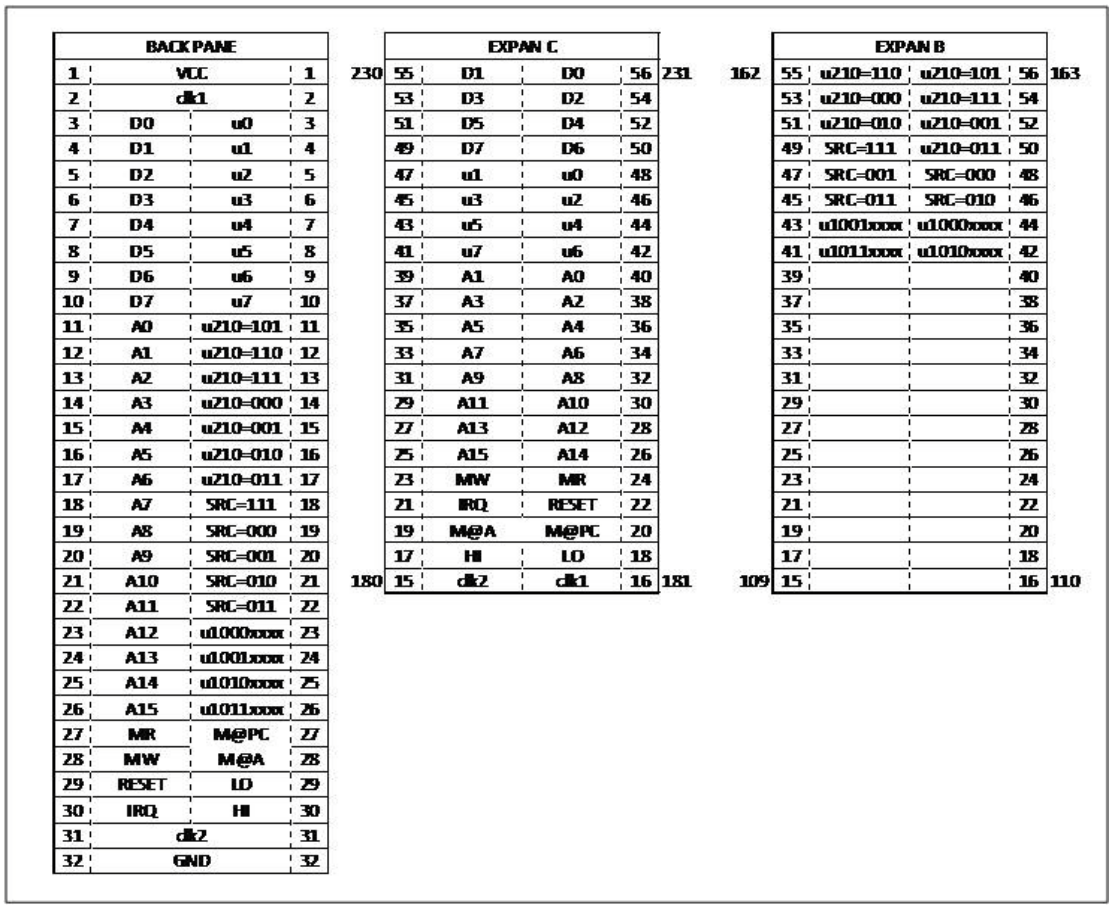
The source and destination fields of the move instructions are coded as follows:

	Destination	Source
000	W	W
001	IP	IP
010	TOS	TOS
011	R	R
100	Memory[W]	Memory[W]
101	OP	Memory[IP]
110	ALU input A	Zero
111	ALU input B	ALU output

TOS = Top of parameter stack; R = Top of return stack

FPGA I/O Pins and Back Pane Connection

Pin connection from 2 ports of FPGA (Expansion Port B and C) I/O pins to back panel.



BACK PANE			
1	VCC		1
2	clk1		2
3	D0	u0	3
4	D1	u1	4
5	D2	u2	5
6	D3	u3	6
7	D4	u4	7
8	D5	u5	8
9	D6	u6	9
10	D7	u7	10
11	A0	u210=101	11
12	A1	u210=110	12
13	A2	u210=111	13
14	A3	u210=000	14
15	A4	u210=001	15
16	A5	u210=010	16
17	A6	u210=011	17
18	A7	SRC=111	18
19	A8	SRC=000	19
20	A9	SRC=001	20
21	A10	SRC=010	21
22	A11	SRC=011	22
23	A12	u1000xxxx	23
24	A13	u1001xxxx	24
25	A14	u1010xxxx	25
26	A15	u1011xxxx	26
27	MR	M@PC	27
28	MW	M@A	28
29	RESET	LO	29
30	IRQ	HI	30
31	clk2		31
32	GND		32

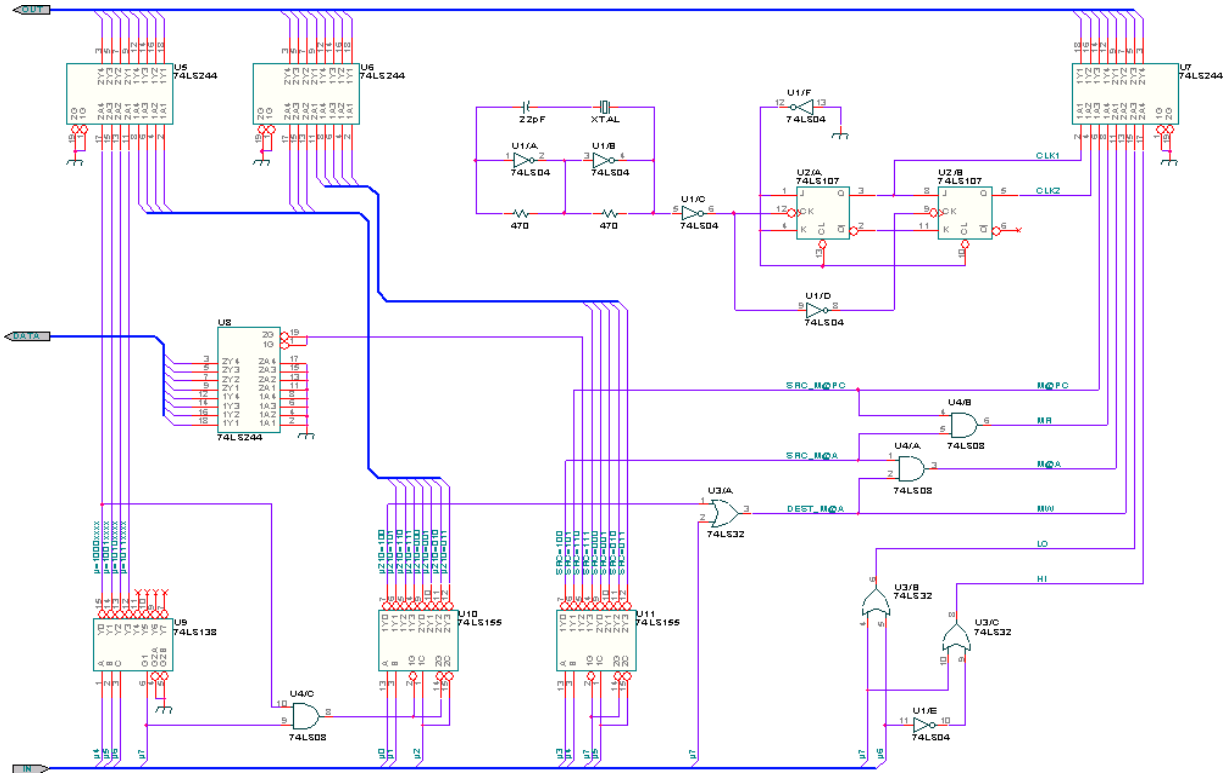
EXPAN C				
230	55	D1	D0	56 231
	53	D3	D2	54
	51	D5	D4	52
	49	D7	D6	50
	47	u1	u0	48
	45	u3	u2	46
	43	u5	u4	44
	41	u7	u6	42
	39	A1	A0	40
	37	A3	A2	38
	35	A5	A4	36
	33	A7	A6	34
	31	A9	A8	32
	29	A11	A10	30
	27	A13	A12	28
	25	A15	A14	26
	23	MW	MR	24
	21	IRQ	RESET	22
	19	M@A	M@PC	20
	17	HI	LO	18
180	15	clk2	clk1	16 181

EXPAN B				
162	55	u210=110	u210=101	56 163
	53	u210=000	u210=111	54
	51	u210=010	u210=001	52
	49	SRC=111	u210=011	50
	47	SRC=001	SRC=000	48
	45	SRC=011	SRC=010	46
	43	u1001xxxx	u1000xxxx	44
	41	u1011xxxx	u1010xxxx	42
	39			40
	37			38
	35			36
	33			34
	31			32
	29			30
	27			28
	25			26
	23			24
	21			22
	19			20
	17			18
109	15			16 110

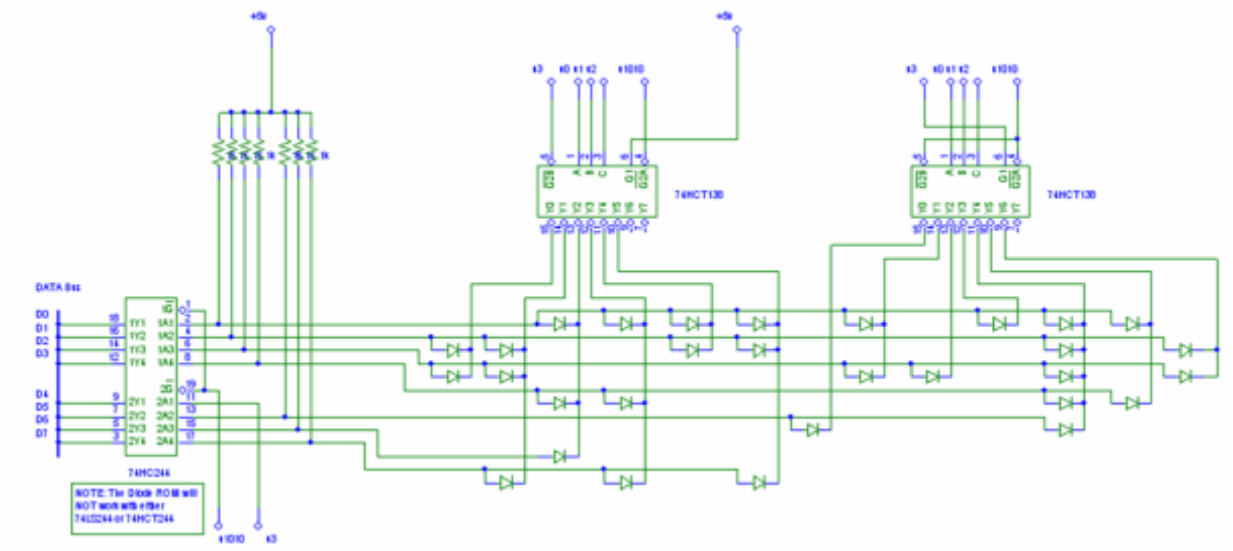
Figure: Pin relation between FPGA and Back Panel

Appendix II Mark 1 Design Schematics

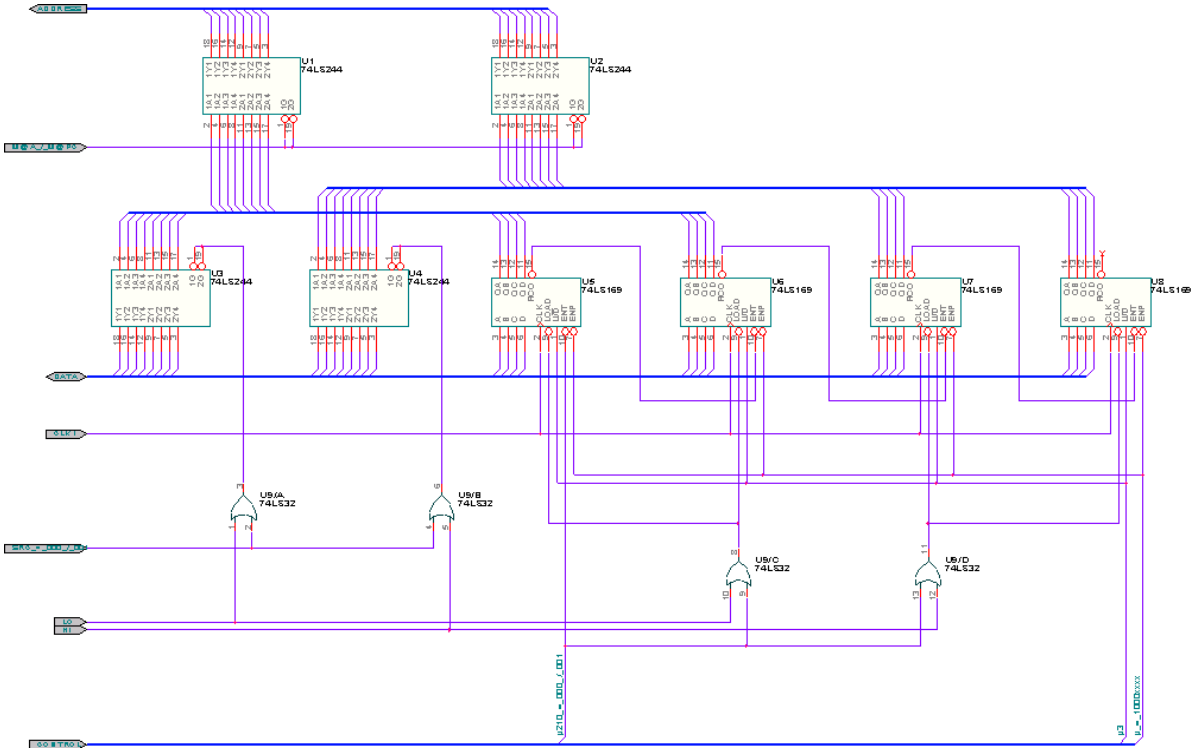
Instruction Decoder Module (reference made to [10]***)***



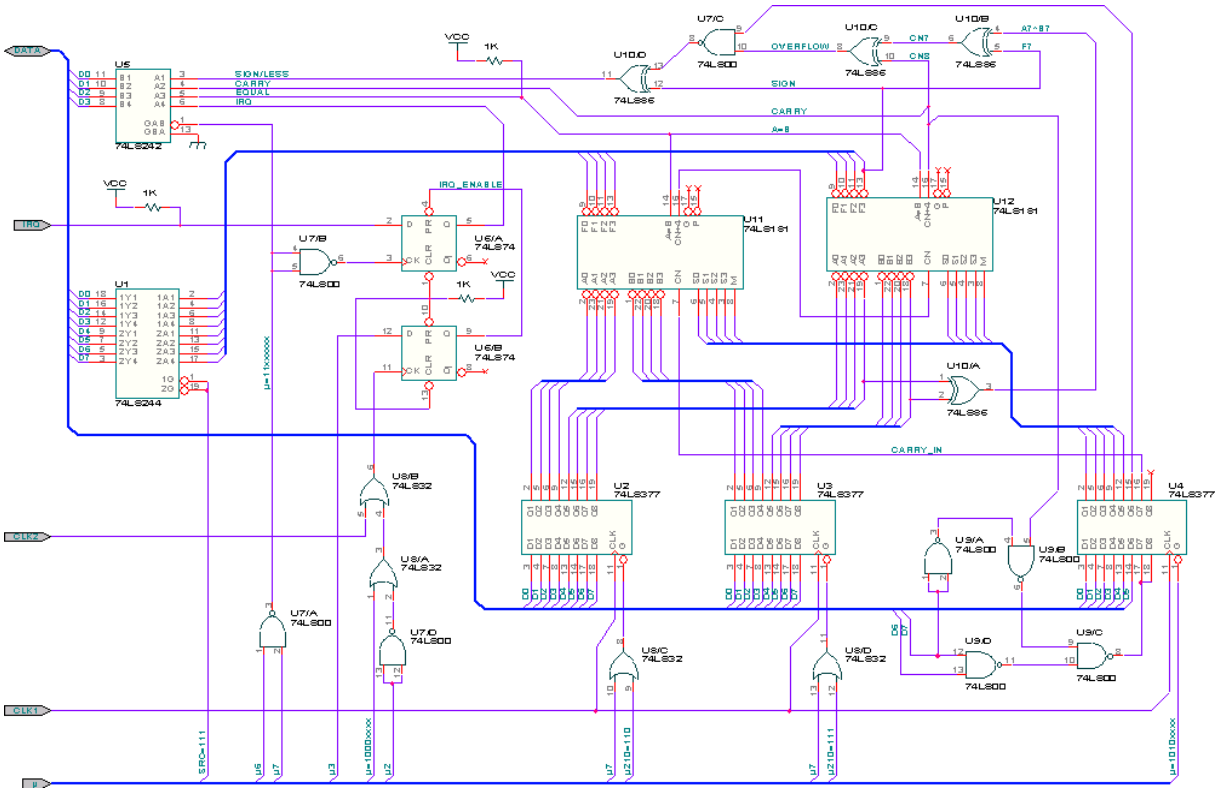
Diode ROM Module (reference made to [10]



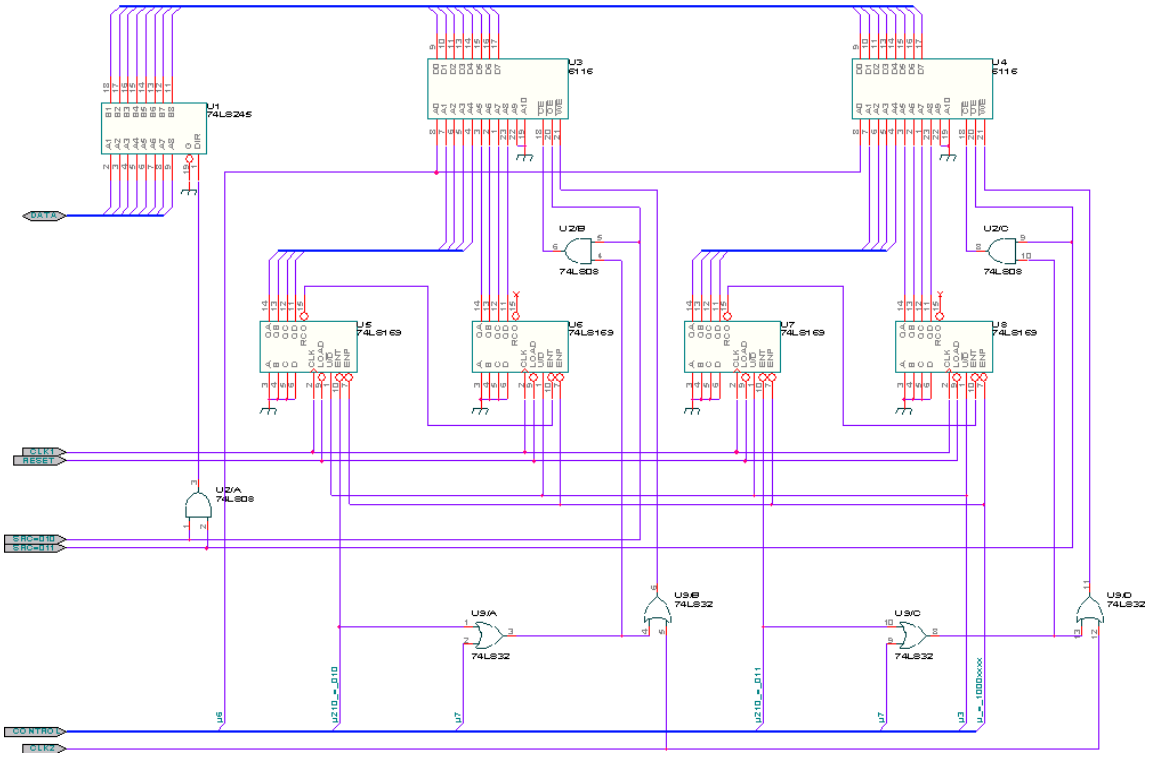
Index Module (reference made to [10]***)***



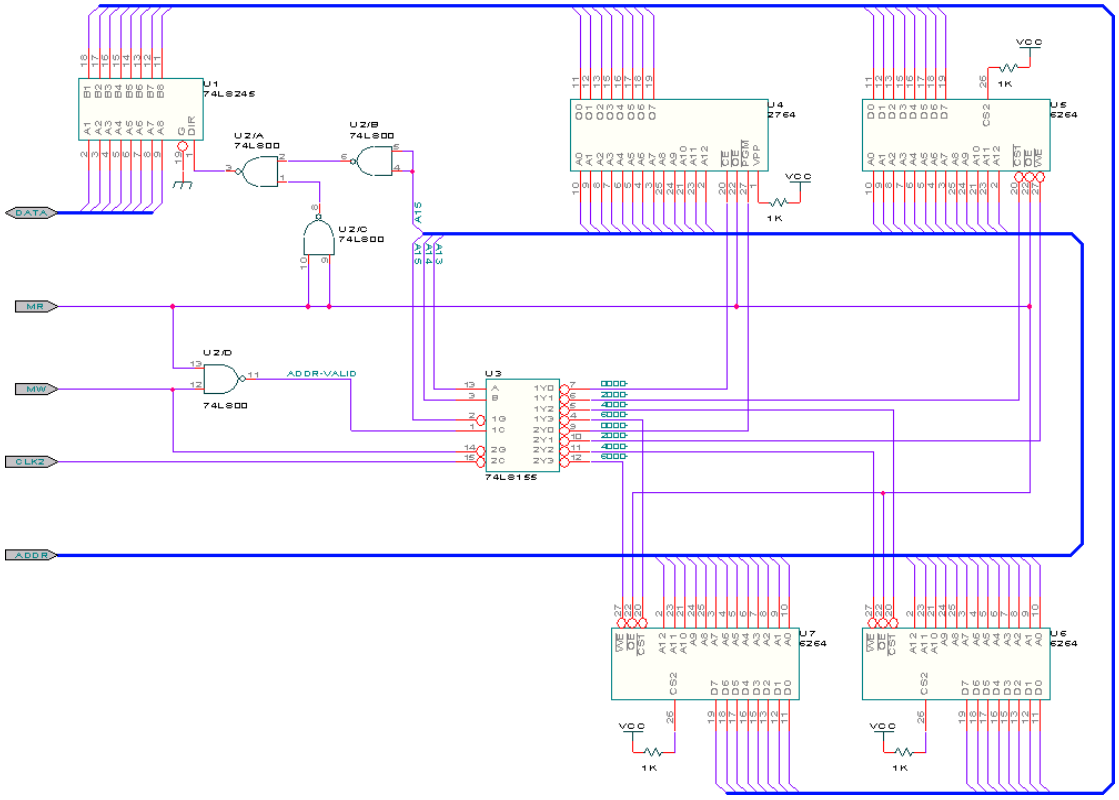
ALU Module (reference made to [10]***)***



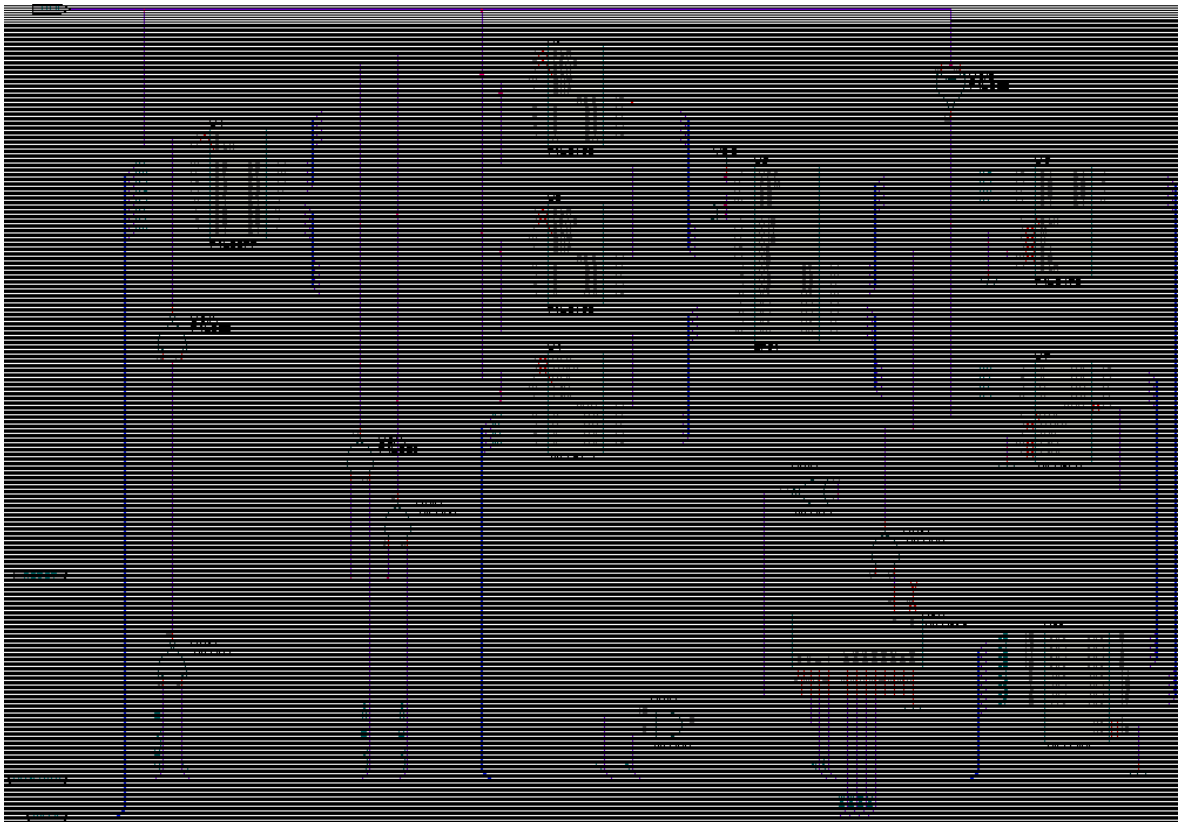
Stacks Module (reference made to [10]***)***



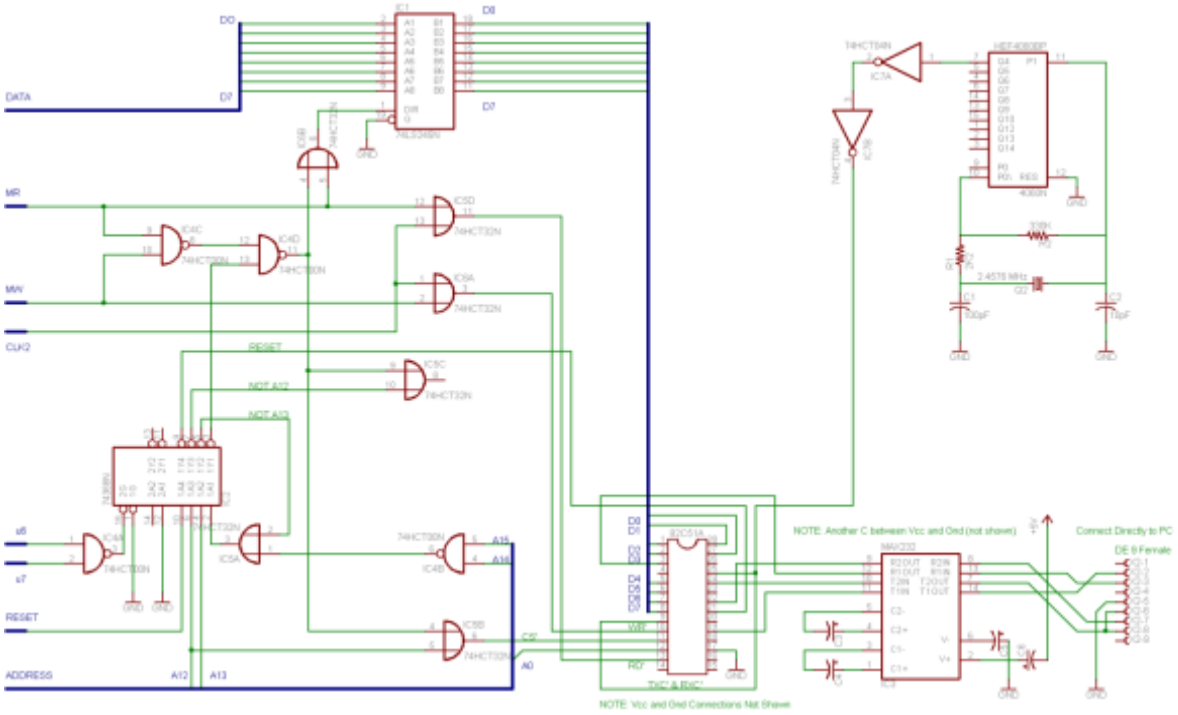
Memory Module (reference made to [10]***)***



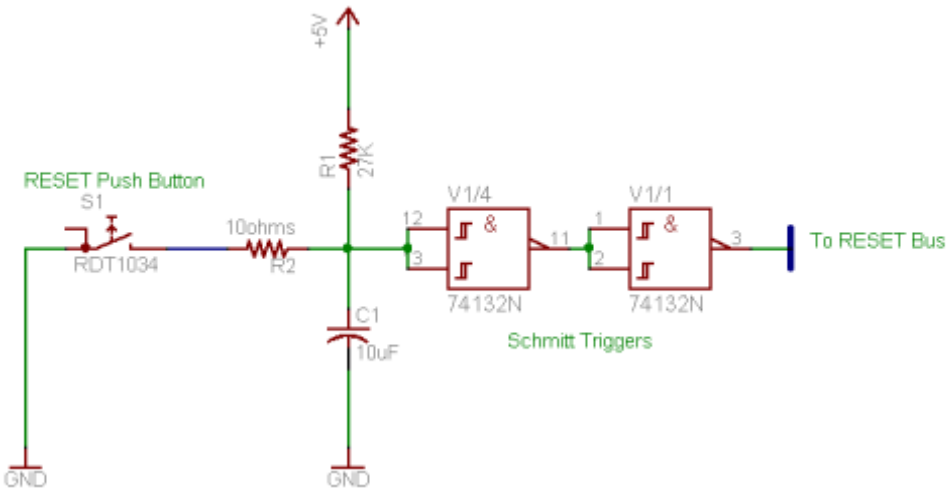
Microcode Sequencer Module (reference made to [10]***)***



I/O Connection Module (reference made to [10]***)***



Power ON Reset Module (reference made to [10]***)***



Appendix III Mark 1 FPGA Module Design in Verilog Codes

```
/* TOP MODULE MARK 1 */
// merge of Index and Diode ROM

// Mark 1 module
module Mark1 (data, addr, clk1, u, u210, SRC, uX, M);
    input [3:0] u;                  //u signal input (7~0)
    input clk1;                     //clock 1
    input [1:0] uX;                 //u1010xxxx,u1000xxxx
    input [1:0] u210;               //u210=001, u210=000
    input [1:0] SRC;                //SRC=001, SPC=000
    input [3:0] M;                  //flags HI, LO, M@PC, M@W
    inout [7:0] data;               //data signals
    output [15:0] addr;             //address signals

    //FLAG ASSIGNMENT
    wire MPC, MW, LO, HI;
    assign MW = M[0];
    assign MPC = M[1];
    assign LO = M[2];
    assign HI = M[3];

    //WIRE DECLARATION
    wire [7:0] d_in;                //data in from INOUT
    wire [7:0] Dout_Index;          //data out from index module
    wire [7:0] Dout_PC, Dout_W;
    wire [7:0] Dout_ROM;            //data out from Diode ROM
    wire [7:0] d_out;               //data out for INOUT

    /* DIODE ROM */
    //input: u[0],u[1],u[2],u[3],u1010xxxx
    //output: data[7:0]
    wire [2:0] U;
    assign U = {u[2],u[1],u[0]};
    diode D1 (Dout_ROM, U, u[3]);

    /* Index data input control */
    wire u210_sel;
    and B0 (u210_sel,u210[0],u210[1]);
    Buff8_244 B1 (d_in,data,u210_sel);

    /* Index Pointer (M@W / M@A) */
    //input: clk1, u[3], M@W, SRC=000, u210=000, LO, HI, u1000xxxx
    //inout: data[7:0]
    index M_W (addr, MW, d_in, Dout_W, clk1, SRC[0], LO, HI, u210[0], u[3], uX[0]);

    /* Index Pointer (M@IP / M@PC) */
    //input: clk1, u[3], M@IP, SRC=001, u210=001, LO, HI, u1000xxxx
    //inout: data[7:0]
    index M_PC (addr, MPC, d_in, Dout_PC, clk1, SRC[1], LO, HI, u210[1], u[3], uX[0]);
```

```
    /* DATA output control */
    wire Index_sel, En;
    and N1 (Index_sel,SRC[0],SRC[1]);
    //SRC control data output from Index modules
    assign Dout_Index = (~SRC[0])?Dout_W:Dout_PC;
    //uX[1] control data output from Diode ROM
    assign d_out = (~uX[1])?Dout_ROM:Dout_Index;
    and N2 (En,Index_sel,uX[1]);
    bufif0 (data[0],d_out[0],En);
    bufif0 (data[1],d_out[1],En);
    bufif0 (data[2],d_out[2],En);
    bufif0 (data[3],d_out[3],En);
    bufif0 (data[4],d_out[4],En);
    bufif0 (data[5],d_out[5],En);
    bufif0 (data[6],d_out[6],En);
    bufif0 (data[7],d_out[7],En);

endmodule


/* Components of Index Module */

module Buff8_244 (Outp, Inp, En);
    input [7:0] Inp;
    input En;
    output [7:0] Outp;

    bufif0 (Outp[0],Inp[0],En);
    bufif0 (Outp[1],Inp[1],En);
    bufif0 (Outp[2],Inp[2],En);
    bufif0 (Outp[3],Inp[3],En);
    bufif0 (Outp[4],Inp[4],En);
    bufif0 (Outp[5],Inp[5],En);
    bufif0 (Outp[6],Inp[6],En);
    bufif0 (Outp[7],Inp[7],En);

endmodule


module Count_169 (In, Out, clk, load, UD, ENT,
ENP, RCO);
    input [7:0] In;
    input clk,load,UD,ENT,ENP;
    output [7:0] Out;
    output RCO;
    reg [7:0] Out;
    reg RCO;

    always @ (posedge clk)
    begin
       if (~load)
         begin
            Out <= In;
         end
       else if (~ENT && ~ENP)
         begin
            if (UD)
               Out <= Out + 1;
               else
               Out <= Out - 1;
            end
          else
            begin
               Out <= Out;
            end
    end

    always @ (ENT,Out,UD)
    begin
       if (~ENT)
       begin
          if (UD)
          begin
             if (Out==8'b11111111)
             RCO = 0;
             else
             RCO = 1;
          end
          else
          begin
             if (Out==8'b00000000)
             RCO = 0;
             else
             RCO = 1;
          end
       end
       else
       begin
          RCO = 1;
       end
    end

endmodule
```

```
/*INDEX module*/
module index (Addr, MA, D_in, D_out, clk,
    SRC, LO, HI, u210, u3, uX);
  input clk;
  input MA, SRC, LO, HI, u210, u3, uX;
  input [7:0] D_in;
  output [7:0] D_out;
  output [15:0] Addr;
  reg [7:0] D_out;

  //counter for lower address (u5, U6)
  wire [7:0] add1;
  wire load1, rco1;
  or U9c (load1, u210, LO);
  Count_169 U56 (D_in, add1, clk, load1, u3,
      u210, uX, rco1);

  //counter for upper address (U7, U8)
  wire [7:0] add2;
  wire load2, rco2;
  or U9d (load2, u210, HI);
  Count_169 U78 (D_in, add2, clk, load2, u3,
      rco1, uX, rco2);

  //address line buffer enable (U1, U2)
  wire [7:0] addr1, addr2;
  Buff8_244 U1 (addr1,add1,MA);
  Buff8_244 U2 (addr2,add2,MA);
  assign Addr = {addr2,addr1};

  always @ (LO, HI, add1, add2)
  begin
    if (~LO)
      D_out = add1;
    else if (~HI)
      D_out = add2;
    else
      D_out = 8'b0000_0000;
  end

endmodule
```

```
/* Diode ROM Module */
module diode (data, U, u3);
  input [2:0] U;
  input u3;       //, u1010;
  output [7:0] data;
  reg [7:0] data;

  always @ (U, u3)
  begin
    if (~u3)
      begin
      case(U)
        4'b000: data<=8'b1110_1001;
        4'b001: data<=8'b0110_1001;
        4'b010: data<=8'b1010_0110;
        4'b011: data<=8'b0110_0110;
        4'b100: data<=8'b1110_1100;
        4'b101: data<=8'b0110_1100;
        default: data<=8'b1111_1111;
      endcase
    end
    else
    begin
      case(U)
        4'b000: data<=8'b1101_1111;
        4'b001: data<=8'b1111_1010;
        4'b010: data<=8'b1111_1011;
        4'b011: data<=8'b1111_1110;
        4'b100: data<=8'b1101_0000;
        4'b101: data<=8'b1111_0110;
        4'b110: data<=8'b1111_1001;
        default: data<=8'b1111_1111;
      endcase
    end
  end

endmodule
```

```
/* TOP MODULE MARK 1 */
// merge of Diode ROM and Instruction Decoder

module Mark1 (data, u, u210, SRC, flag, uX);
    input [7:0] u;                  //u signal input (7~0)
    output [3:0] uX;                //u1011xxxx,u1010xxxx,u1001xxxx,u1000xxxx
    output [6:0] u210;              //u210=011, u210=010, u210=001, u210=000
                                    //u210=111, u210=110, u210=101
    output [4:0] SRC;               //SRC=011, SRC=010, SRC=001, SRC=000, SRC=111
    output [5:0] flag;              //flags M@PC, MR, MA, MW, LO, HI
    inout [7:0] data;               //data signals

    wire [7:0] d_out;


    /* DIODE ROM */
    //input: u[0],u[1],u[2],u[3],u1010xxxx
    //output: data[7:0]
    wire [2:0] U;
    wire [7:0] Dout_ROM;
    assign U = {u[2],u[1],u[0]};
    diode D1 (Dout_ROM, U, u[3]);


    /* Instruction Decoder */
    //input: u[7:0]
    //output: data, uX[3:0], u210[6:0], SRC[4:0], M@PC, MR, M@A, MW, LO, HI
    wire D_en;      //data enable for InsDec (SRC=110)
    decoders InDe (u, D_en, flag, uX, u210, SRC);


    // data signal selection
    wire En;
    assign d_out = (D_en)?Dout_ROM:8'b0000_0000;
    and N0 (En,D_en,uX[2]);
    bufif0 (data[0],d_out[0],En);
    bufif0 (data[1],d_out[1],En);
    bufif0 (data[2],d_out[2],En);
    bufif0 (data[3],d_out[3],En);
    bufif0 (data[4],d_out[4],En);
    bufif0 (data[5],d_out[5],En);
    bufif0 (data[6],d_out[6],En);
    bufif0 (data[7],d_out[7],En);

endmodule


/* Components of Instruction Decoder */

//demux 74138 (modified)
module Uxxxx (uOUT, EN, sel);
    input [2:0] sel;   //input u6,u5,u4
    input EN;          //input u7
    output [3:0] uOUT;     //output u1011xxxx,
     u1010xxxx, u1001xxxx, u1000xxxx
    reg [3:0] out;    //as buffer for output

    always @ (sel)
    begin
        case (sel)
            3'b000: out = 4'b1110;
            3'b001: out = 4'b1101;
            3'b010: out = 4'b1011;
            3'b011: out = 4'b0111;
            3'b100: out = 4'b1111;
            3'b101: out = 4'b1111;
            3'b110: out = 4'b1111;
            3'b111: out = 4'b1111;
            default: out = 4'b1111;
        endcase
    end
```

```
    assign uOUT = (EN)?out:4'b1111;

endmodule

//demux 74155
module mux155 (outA, outB, enA, enB, add);
    input [1:0] enA,enB;      //enable select (G,C)
    input [1:0] add;          //select line (address)
    output [3:0] outA;        //output A
    output [3:0] outB;        //output B
    reg [3:0] out_a;
    reg [3:0] out_b;

    always @ (add, enA)     //enabled with 01
    begin
       if (enA[0])
       begin
          case (add)
             2'b00: out_a = 4'b1110;
             2'b01: out_a = 4'b1101;
             2'b10: out_a = 4'b1011;
             2'b11: out_a = 4'b0111;
             default: out_a = 4'b1111;
          endcase
       end
       else
       begin
          out_a = 4'b1111;
       end
    end
    assign outA = (~enA[1])?out_a:4'b1111;

    always @ (add, enB)     //enabled with 01
    begin
       if (~enB[0])
       begin
          case (add)
             2'b00: out_b = 4'b1110;
             2'b01: out_b = 4'b1101;
             2'b10: out_b = 4'b1011;
             2'b11: out_b = 4'b0111;
             default: out_b = 4'b1111;
          endcase
       end
       else
       begin
          out_b = 4'b1111;
       end
    end
    assign outB = (~enB[1])?out_b:4'b1111;

endmodule
```

```
/* Instruction Decoder module [without clock signals] */
module decoders (uIN, En, flag, Ux, u210, SRC);
    input [7:0] uIN;          //u signal input (7~0)
    output En;
    output [5:0] flag;
    //flag signals (M@PC, MR, M@A, MW, LO, HI)
    output [3:0] Ux;
    //u1011xxxx, u1010xxxx, u1001xxxx, u1000xxxx
    output [6:0] u210;
    //u210=011, u210=010, u210=001, u210=000
    //u210=111, u210=110, u210=101
    output [4:0] SRC;
    //SRC=011, SRC=010, SRC=001, SRC=000, SRC=111

    //define supply for HI and LOW signals
    supply0 zero;
    supply1 one;

    //u=____xxxx decoder (U9 in schematic)
    wire [2:0] Ux_in;
    //input to uXXXX decoder
    wire [3:0] Ux_out;
    assign Ux_in = {uIN[6],uIN[5],uIN[4]};
    Uxxxx u9 (Ux_out, uIN[7], Ux_in);
    //output buffer for u=____xxxx decoder
    buf u9a (Ux[0],Ux_out[0]);
    buf u9b (Ux[1],Ux_out[1]);
    buf u9c (Ux[2],Ux_out[2]);
    buf u9d (Ux[3],Ux_out[3]);

    //u210 signal decoder (u10 in schematic)
    wire [3:0] outA, outB;
    wire [1:0] enAB;
    wire [1:0] add1;
    wire n_out;
    assign add1 = {uIN[1],uIN[0]};
    and u4c (n_out,Ux[0],uIN[7]);
    assign enAB = {n_out,uIN[2]};
    mux155 u10 (outA, outB, enAB, enAB, add1);
    //output buffer for u210 signal decoder
    buf u10a (u210[0],outA[1]);
    buf u10b (u210[1],outA[2]);
    buf u10c (u210[2],outA[3]);
    buf u10d (u210[3],outB[0]);
    buf u10e (u210[4],outB[1]);
    buf u10f (u210[5],outB[2]);
    buf u10g (u210[6],outB[3]);

    //SRC signal decoder (u11 in schematic)
    wire [3:0] outC, outD;
    wire [1:0] enCD;
    wire [1:0] add2;
    assign add2 = {uIN[4],uIN[3]};
    assign enCD = {uIN[7],uIN[5]};
    mux155 u11 (outC, outD, enCD, enCD, add2);
    //output buffer for SRC signal decoder
```

```
    buf u11a (SRC[0],outC[3]);
    buf u11b (SRC[1],outD[0]);
    buf u11c (SRC[2],outD[1]);
    buf u11d (SRC[3],outD[2]);
    buf u11e (SRC[4],outD[3]);

    //flag signals (u7 in schematic)
    wire Hi, Lo, MW, MA, MR, MPC;
    //DECT_M@A = MW
    wire uIn6;
    or u3b (Lo, uIN[7], uIN[6]);
    not u1e (uIn6, uIN[6]);
    or u3c (Hi, uIN[7], uIn6);
    or u3a (MW, uIN[7], outA[0]);
    and u4a (MA, MW, outC[0]);
    //outC[0] = SRC_M@A
    and u4b (MR, outC[0], outC[1]);
    //outC[1] = SRC_M@PC
    assign MPC = outC[1];
    //output buffer for flag signals
    buf u7a (flag[0],Hi);
    buf u7b (flag[1],Lo);
    buf u7c (flag[2],MW);
    buf u7d (flag[3],MA);
    buf u7e (flag[4],MR);
    buf u7f (flag[5],MPC);

    //data signals control (u8 in shcematic)
    assign En = outC[2];

endmodule
```

```
/* Diode ROM Module */
module diode (data, U, u3);
   input [2:0] U;
   input u3;     //, u1010;
   output [7:0] data;
   reg [7:0] data;

   always @ (U, u3)
   begin
      if (~u3)
         begin
         case(U)
            4'b000: data<=8'b1110_1001;
            4'b001: data<=8'b0110_1001;
            4'b010: data<=8'b1010_0110;
            4'b011: data<=8'b0110_0110;
            4'b100: data<=8'b1110_1100;
            4'b101: data<=8'b0110_1100;
            default: data<=8'b1111_1111;
         endcase
      end
      else
      begin
         case(U)
            4'b000: data<=8'b1101_1111;
            4'b001: data<=8'b1111_1010;
            4'b010: data<=8'b1111_1011;
            4'b011: data<=8'b1111_1110;
            4'b100: data<=8'b1101_0000;
            4'b101: data<=8'b1111_0110;
            4'b110: data<=8'b1111_1001;
            default: data<=8'b1111_1111;
         endcase
      end
   end

endmodule
```

Appendix IV Test Codes for UP2 Board

```
//TEST MODULE FOR SENDING SIGNALS FROM FPGA TO TTL
//[INPUT FROM FPGA IS PROCESSED BY 74LS181 (ALU)]
//[RESULT RETURN TO FPGA FOR DECODE TO 7-SEGMENT]

module core_1 (inA, inB, outA, outB, inSel, outSel, BCD, LCD);
  input [2:0] inA, inB;
  input [3:0] BCD;
  input [1:0] inSel;
  output [2:0] outA, outB;
  output [5:0] outSel;
  output [7:0] LCD;
  reg [5:0] outSel;
  reg [7:0] LCD;

  //on board DIP switch to ALU input
  assign outA = inA;
  assign outB = inB;

  //ALU function select (active high in,out)
  always @ (inSel)
  begin
    //active low
    case (inSel)
      2'b00: outSel <= 6'b100101; //add
      2'b01: outSel <= 6'b011000; //sub
      2'b10: outSel <= 6'b111011; //or
      2'b11: outSel <= 6'b101111; //and
      default: outSel <= 6'b111110;
    endcase
  end

  //BCD for 7-segment display (ALU result)
  always @ (BCD)
  begin
    //active low
    case (BCD)
      4'b0000: LCD = 8'b0000_0011; //0
      4'b0001: LCD = 8'b1001_1111; //1
      4'b0010: LCD = 8'b0010_0101; //2
      4'b0011: LCD = 8'b0000_1101; //3
      4'b0100: LCD = 8'b1001_1001; //4
      4'b0101: LCD = 8'b0100_1001; //5
      4'b0110: LCD = 8'b0100_0001; //6
      4'b0111: LCD = 8'b0001_1111; //7
      4'b1000: LCD = 8'b0000_0001; //8
      4'b1001: LCD = 8'b0001_1001; //9
      4'b1010: LCD = 8'b0001_0001; //A
      4'b1011: LCD = 8'b1100_0001; //b
      4'b1100: LCD = 8'b0110_0011; //C
      4'b1101: LCD = 8'b1000_0101; //d
      4'b1110: LCD = 8'b0110_0001; //E
      4'b1111: LCD = 8'b0111_0001; //F
      default: LCD = 8'b1111_1110; //.
    endcase
  end

endmodule
```

```
//TEST MODULE FOR FPGA AND MEMORY
    CHIP (RAM)
//[FPGA READ DATA FROM RAM AND
    DECODE TO 7-SEGMENT]
//[FPGA WRITE DATA TO RAM AND
    CONTROL ADDRESS]

`timescale 1us/1ps

module core_2 (dip, data, addr, ctrl, LCD);
   input [7:0] dip;
   inout [3:0] data;
   output [1:0] addr;
   output [2:0] ctrl;
   output [7:0] LCD;
   reg [2:0] ctrl;
   reg [7:0] LCD;

   //write data
   assign data = (dip[7:6]==2'b01) ? dip[3:0] :
    4'bz;

   //sending location address
   assign addr = dip[5:4];

   //read and write control
   always @ (dip[7:6])
   begin
      case (dip[7:6])
         2'b00: begin ctrl=3'b011; #5 ctrl=3'b001;
                    end  //read
         2'b01: begin ctrl=3'b011; #5 ctrl=3'b010;
                    end  //write
         default ctrl=3'b100; //disable
      endcase
   end

   //decode for display
   reg [3:0] BCD;
   always @ (ctrl,data)
   begin
      BCD = (ctrl==3'b001)?data:BCD;
       //load data to display at read mode
   end
   always @ (BCD)
   begin
      //active low
      case (BCD)
         4'b0000: LCD = 8'b0000_0011; //0
         4'b0001: LCD = 8'b1001_1111; //1
         4'b0010: LCD = 8'b0010_0101; //2
         4'b0011: LCD = 8'b0000_1101; //3
         4'b0100: LCD = 8'b1001_1001; //4
         4'b0101: LCD = 8'b0100_1001; //5
         4'b0110: LCD = 8'b0100_0001; //6
         4'b0111: LCD = 8'b0001_1111; //7
         4'b1000: LCD = 8'b0000_0001; //8
         4'b1001: LCD = 8'b0001_1001; //9
         4'b1010: LCD = 8'b0001_0001; //A
         4'b1011: LCD = 8'b1100_0001; //b
         4'b1100: LCD = 8'b0110_0011; //C
         4'b1101: LCD = 8'b1000_0101; //d
         4'b1110: LCD = 8'b0110_0001; //E
         4'b1111: LCD = 8'b0111_0001; //F
         default: LCD = 8'b1111_1110; //.
      endcase
   end

endmodule
```

```
//TEST MODULE FOR FPGA FUNCTION WITH EXTERNAL CLOCK
//[READ CLOCK SIGNAL FROM MARK1 AND PERFORM COUNT]

module core_3 (clk, rst, LCD);
   input clk, rst;
   output [7:0] LCD;
   reg [7:0] LCD;
   reg [3:0] BCD;

   //clock down from 1MHz to 1Hz
   //count up from 0 to F
   reg [25:0] temp;
   always @ (posedge clk)
   begin
      if (rst)
      begin
         temp = 0;
         BCD = 4'b0;
      end
      else
      begin
         temp = temp + 1;
         BCD = (temp == 1000000) ? BCD+1 :
                  BCD;
      end
   end

   //decode binary to 7-segment
   always @ (BCD)
   begin
      //active low
      case (BCD)
         4'b0000: LCD = 8'b0000_0011; //0
         4'b0001: LCD = 8'b1001_1111; //1
         4'b0010: LCD = 8'b0010_0101; //2
         4'b0011: LCD = 8'b0000_1101; //3
         4'b0100: LCD = 8'b1001_1001; //4
         4'b0101: LCD = 8'b0100_1001; //5
         4'b0110: LCD = 8'b0100_0001; //6
         4'b0111: LCD = 8'b0001_1111; //7
         4'b1000: LCD = 8'b0000_0001; //8
         4'b1001: LCD = 8'b0001_1001; //9
         4'b1010: LCD = 8'b0001_0001; //A
         4'b1011: LCD = 8'b1100_0001; //b
         4'b1100: LCD = 8'b0110_0011; //C
         4'b1101: LCD = 8'b1000_0101; //d
         4'b1110: LCD = 8'b0110_0001; //E
         4'b1111: LCD = 8'b0111_0001; //F
         default: LCD = 8'b1111_1110; //.
      endcase
   end

endmodule
```

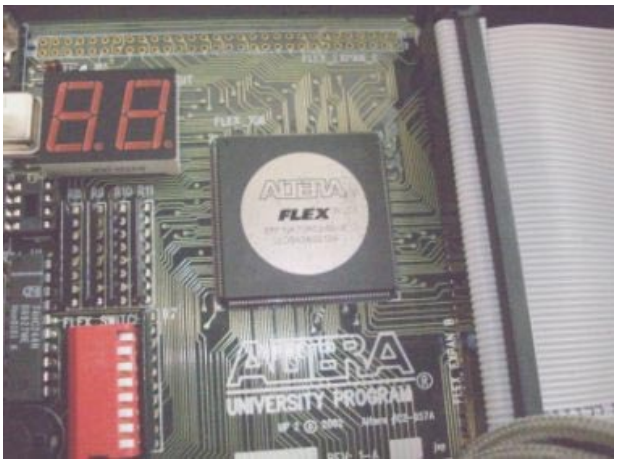
```
//TEST MODULE FOR QUADRATURE CLOCK SIGNAL
//[DERIVE CLOCK FROM CRYSTAL]

module core_4 (clk, rst, LCD, clk_1, clk_2);
   input clk, rst;
   output [7:0] LCD;
   output clk_1, clk_2;
   reg [7:0] LCD;
   reg [3:0] BCD;

   //clock down from 4MHz to 1Hz and count up
   reg [21:0] temp;
   reg [1:0] cnt;
   wire clk_o;
   always @ (posedge clk)
   begin
      if (rst)
      begin
         temp = 0;
         BCD = 4'b0;
         cnt = 0;
      end
      else
      begin
         temp = temp + 1;
         BCD = (temp == 4000000) ? BCD+1 :
                  BCD;
         cnt = cnt + 1;
      end
   end
   assign clk_o = cnt[1];
   assign clk_1 = ~clk_o;
   assign clk_2 = ~clk_1;

   //decode binary to 7-segment
   always @ (BCD)
   begin
      //active low
      case (BCD)
         4'b0000: LCD = 8'b0000_0011; //0
         4'b0001: LCD = 8'b1001_1111; //1
         4'b0010: LCD = 8'b0010_0101; //2
         4'b0011: LCD = 8'b0000_1101; //3
         4'b0100: LCD = 8'b1001_1001; //4
         4'b0101: LCD = 8'b0100_1001; //5
         4'b0110: LCD = 8'b0100_0001; //6
         4'b0111: LCD = 8'b0001_1111; //7
         4'b1000: LCD = 8'b0000_0001; //8
         4'b1001: LCD = 8'b0001_1001; //9
         4'b1010: LCD = 8'b0001_0001; //A
         4'b1011: LCD = 8'b1100_0001; //b
         4'b1100: LCD = 8'b0110_0011; //C
         4'b1101: LCD = 8'b1000_0101; //d
         4'b1110: LCD = 8'b0110_0001; //E
         4'b1111: LCD = 8'b0111_0001; //F
         default: LCD = 8'b1111_1110; //.
      endcase
endmodule
```

```
//TEST MODULE FOR INDEX COUNTER
//[ MIMIC INDEX MODULE OF MARK 1 ]

//counter (mimic 74LS169)
module Counter (In, Out, clk, load, RCO);
   input [3:0] In;
   input clk,load;
   output [3:0] Out;
   output RCO;
   reg [3:0] Out;
   reg RCO;

   always @ (posedge clk)
   begin
      if (~load)
        begin
          Out = In;
        end
      else
        begin
          Out = Out + 1;
        end
   end

   always @ (Out)
   begin
      if (Out==4'b1111)
        RCO = 0;
      else
        RCO = 1;
   end

endmodule


//display module
module disp (BCD, LCD);
   input [3:0] BCD;
   output [7:0] LCD;
   reg [7:0] LCD;

   always @ (BCD)
   begin
      //active low
      case (BCD)
         4'b0000: LCD = 8'b0000_0011; //0
         4'b0001: LCD = 8'b1001_1111; //1
         4'b0010: LCD = 8'b0010_0101; //2
         4'b0011: LCD = 8'b0000_1101; //3
         4'b0100: LCD = 8'b1001_1001; //4
         4'b0101: LCD = 8'b0100_1001; //5
         4'b0110: LCD = 8'b0100_0001; //6
         4'b0111: LCD = 8'b0001_1111; //7
         4'b1000: LCD = 8'b0000_0001; //8
         4'b1001: LCD = 8'b0001_1001; //9
         4'b1010: LCD = 8'b0001_0001; //A
         4'b1011: LCD = 8'b1100_0001; //b
         4'b1100: LCD = 8'b0110_0011; //C
         4'b1101: LCD = 8'b1000_0101; //d
         4'b1110: LCD = 8'b0110_0001; //E
         4'b1111: LCD = 8'b0111_0001; //F
         default: LCD = 8'b1111_1110; //.
      endcase
   end

endmodule


//main module
module indexing (clk, rst, LCD1, LCD2);
   input clk, rst;
   output [7:0] LCD1, LCD2;

   //clock down 1MHz to 1Hz
   reg [19:0] temp;
   reg clck;
   always @ (posedge clk)
   begin
      if (~rst)
      begin
         temp = 0;
         clck = 0;
      end
      else
      begin
         temp = temp + 1;
         clck = (temp == 1000000)?~clck:clck;
      end
   end

   //counter
   wire [3:0] dat;
   wire [3:0] c_out1, c_out2;
   wire RCO1, RCO2;
   assign dat = 4'b0;
   Counter C1 (dat, c_out1, clck, rst, RCO1);
   Counter C2 (dat, c_out2, RCO1, rst, RCO2);

   //display output
   disp D1 (c_out1, LCD1);
   disp D2 (c_out2, LCD2);

endmodule
```

Appendix V Photo Collection of The Project

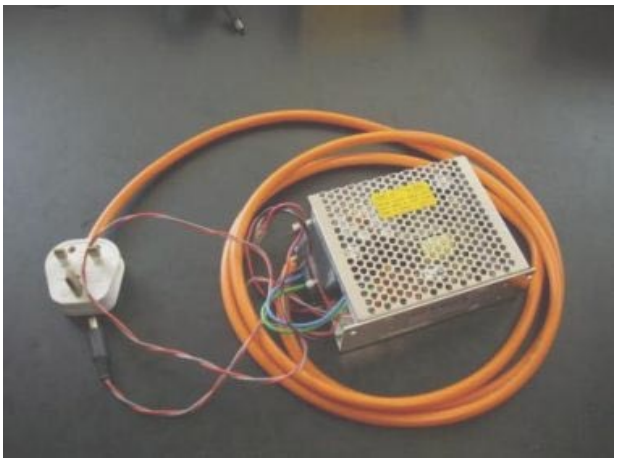
This section provides a collection of photo taken throughout the project, for illustration and also record purposes.

FLEX10K70 FPGA by Altera

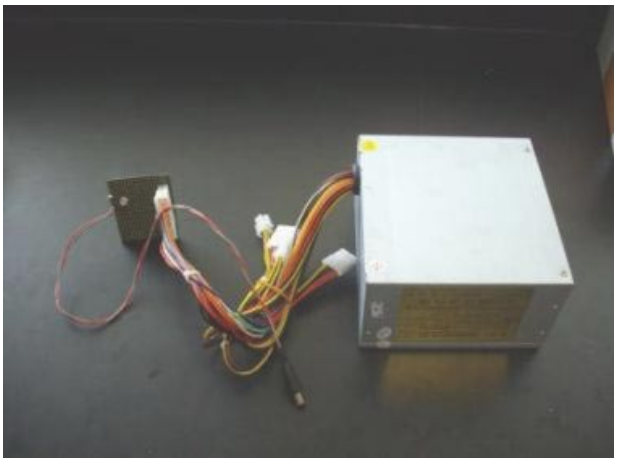


FLEX10K FPGA used, on UP2 board

Power Supply

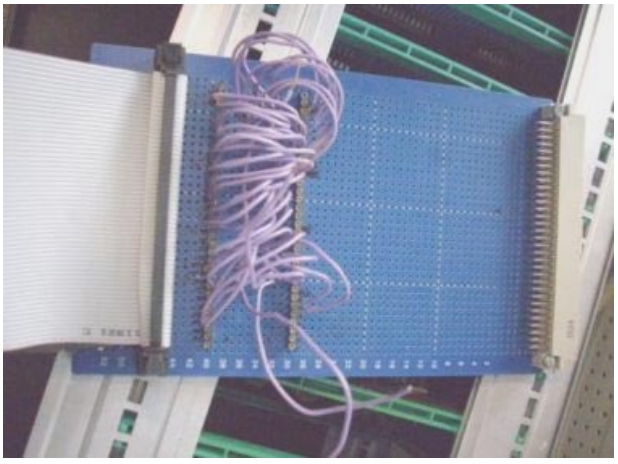


12VDC Power Supply

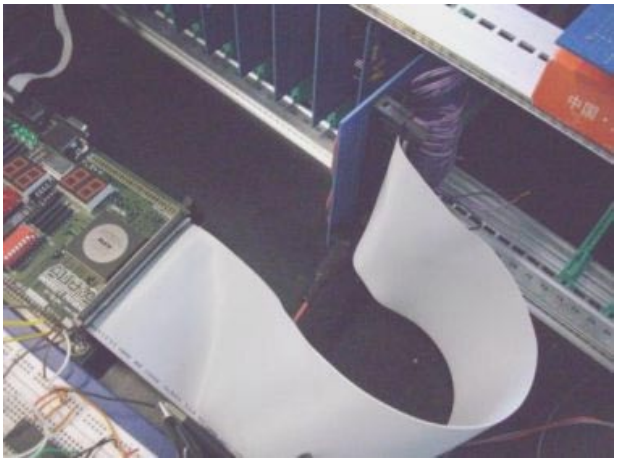


ATX Power Supply

Expansion card

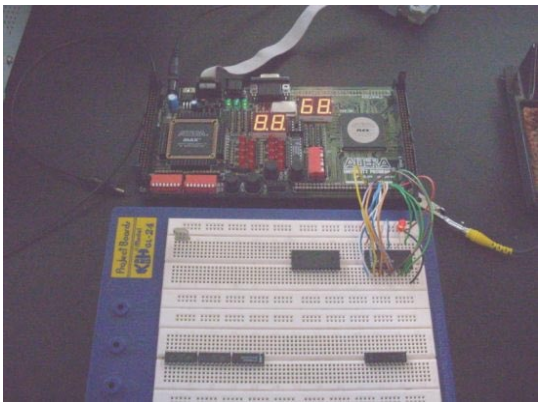


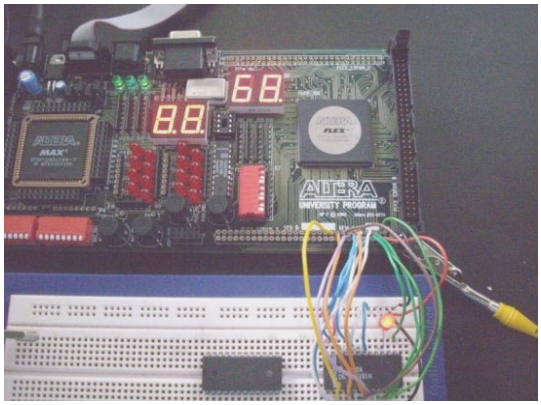
Front view of Expansion Card Version 1 - flexible connection wires

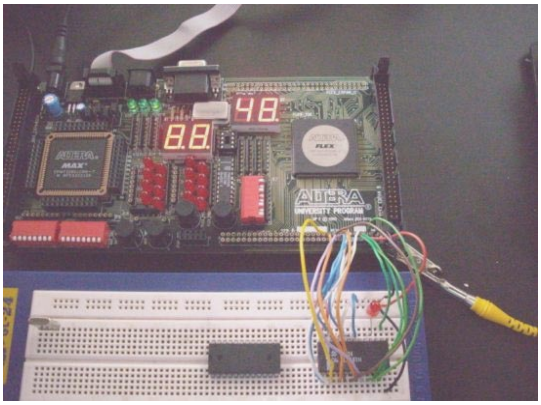


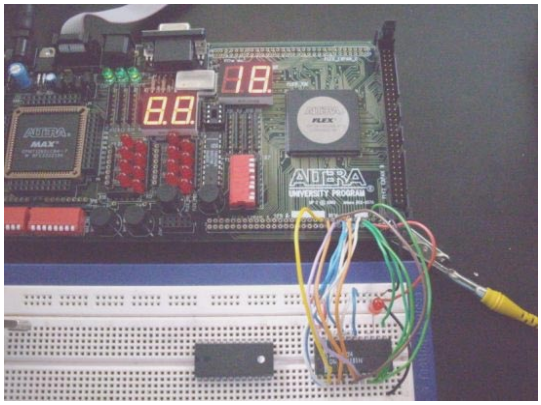
Connection between Expansion Card and FPGA

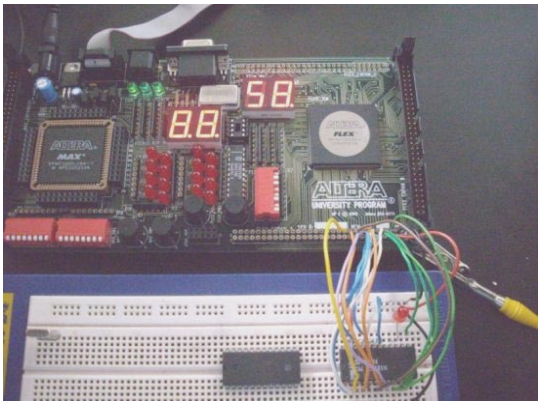
Testing stage



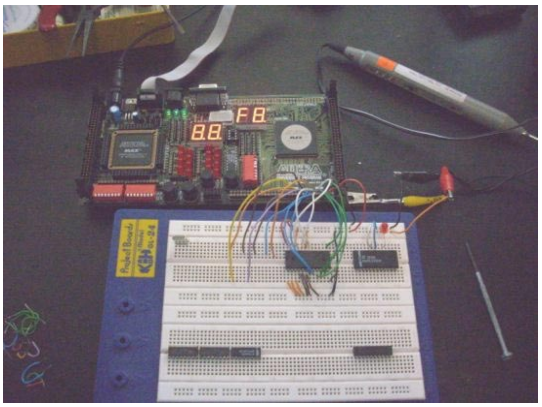


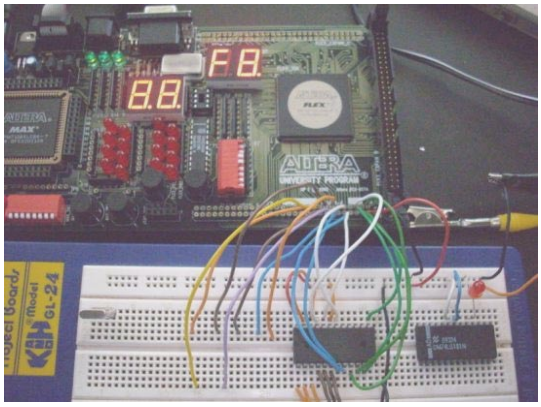


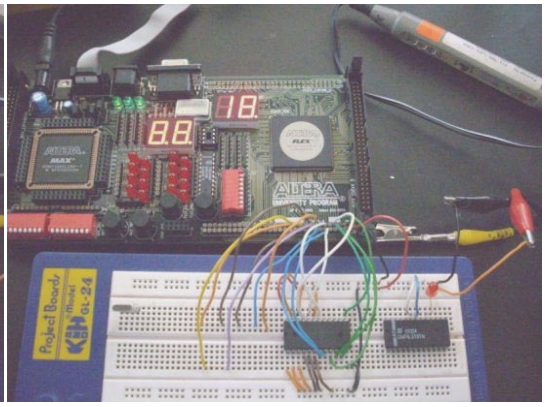


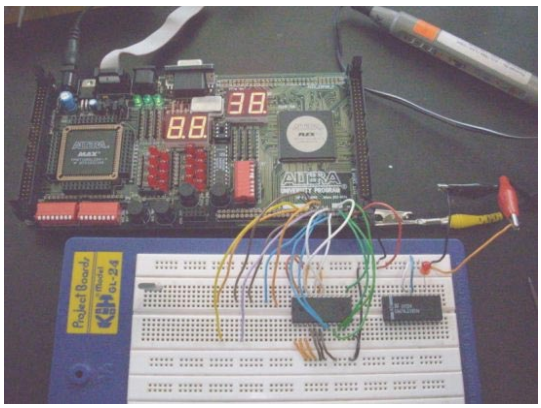


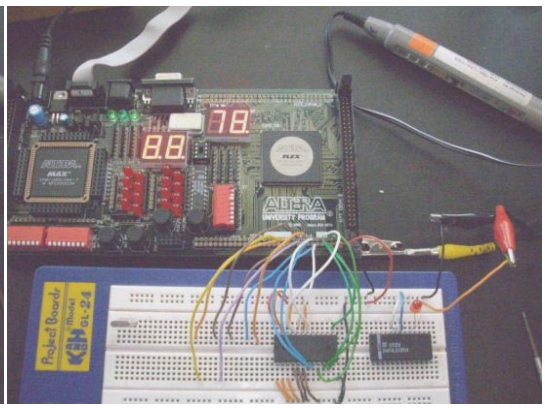
FPGA interface test with 74LS181 ALU chip



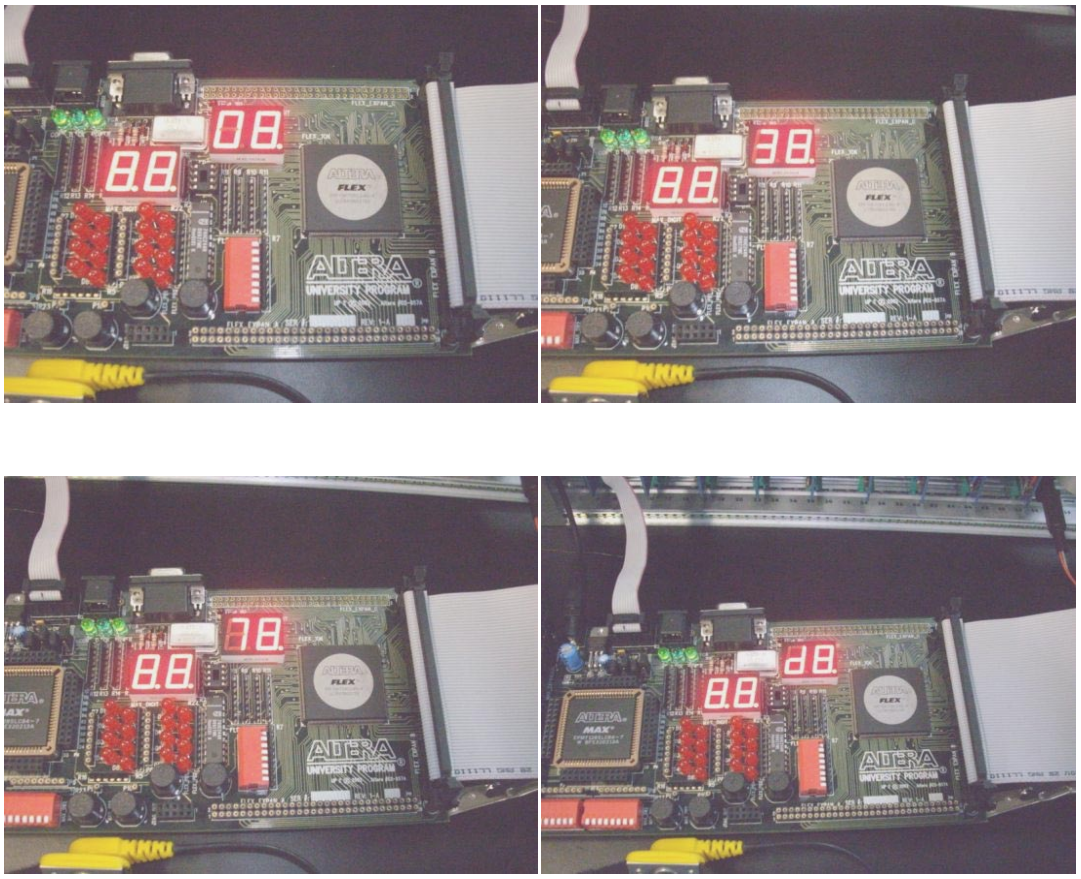








FPGA interfacing with 6116 2KB SRAM



FPGA performing count-up using clock signal from Mark 1

Appendix VI Application Notes on FPGA Design

This section records the experience encountered while carrying out this project with Altera FPGA device. The finding recorded could serve as reference for FPGA design and application in the future.

Below is the list of discussion recorded in this section.

1. TTL Compatibility with FPGA
2. Clock Signal Derivation
3. INOUT Port Implementation in Verilog HDL
4. Use of Buffers for OUTPUT Port

TTL Compatibility with FPGA

Modern FPGA such as the FLEX10K from Altera or later models are TTL compatible. In the configuration program provided by Altera, there is an option to switch the I/O pins of FPGA to be TTL compatible at 5V level 3.3V level.

Besides, testing of FPGA with simple TTL design also confirmed that FPGA and TTL signals are compatible. On the other hand, the success of implementing Mark 1 Clone with FPGA in this project clearly proves that both FPGA and TTL are compatible.

However, attention shall be given to the propagation time and setup time for both TTL and FPGA signal when high speed application is required. Reason being it was observed from oscilloscope that the setup time for TTL signals is slightly longer than that of FPGA. This could affect the data signal especially for system involves memory access.

Clock Signal Derivation

Several attempts had been made throughout this project to include the Mark 1 Clone system clock derivation into the FPGA but fails. There are 2 clock system for the system namely ‘clock 1’ and ‘clock 2’. Both the clock signals are quadrature of each other. Hence, a JK flop-flip is used in the design to derive the required quadrature clock signal for the system.

After a few trials, a quadrature clock signal of 1MHz could be output from FPGA I/O pin with clock edge identification and count-down from a 4MHz crystal oscillator. However, the other portion of FPGA program, namely the Instruction Decoder codes could not function correctly.

This finding shows that FPGA is not suitable for deriving clock signal using frequency division, JK flip-flop or edge detection. The solution is to supply the clock signal for digital system using physical components such as TTL chips.

INOUT Port Implementation in Verilog HDL

INOUT port in Verilog HDL code is implemented using 2 tri-state buffers with a common enable. The tri-state buffers are enabled with the opposite signal level. Hence, the usage of INOUT port in Verilog design has to be accompanied by a signal that will determine the direction of data flow through the INOUT port.

The design of INOUT port control has a critical impact towards the design of this project. Signal flow direction of a data bus is not much of concern in TTL design as it is protected by the input or output of TTL chips. However, the case is rather different in FPGA design. The wrong signal flow could put the FPGA system in intermittent stage.

This issue is first encountered during the design for Index pointer module. The port label “Data” is an INOUT port. The success control of data flow through this port is the key of successfully in implementing the module using FPGA.

On the other hand, there is difficulty performing simulation for INOUT port. Error is encountered when functional simulation is performed on the design. The INOUT port had to be separated into individual INPUT and OUTPUT before the simulation in ModelSim could be carried out properly. As for timing simulation using Quartus II, there is no need to separate the port but high-impedance signal has to be supplied to the INOUT port while it is functioning as output.

Use of Buffers for OUTPUT Port

The use of tri-state buffers before a data signal is send to the OUTPUT port can help protect the FPGA I/O port and provide a stable signal to the system that is receiving the signal.

This is particular when the OUTPUT port needs to provide high-impedance signals. The use of tri-state buffer for each OUTPUT pin that needs to provide high-impedance signal could help the FPGA in providing a stable signal.

This method should be used in place of equating the particular net or port with high impedance, “z” signal. The output from a port which is supplied with “z” signal is not as stable as the “z” signal is initiated by a tri-state buffer.

This issue is encountered during the design of Instruction Decoder and Index module as both the module involves high-impedance signal output.