EDUCATIONAL PROCESSOR

HAFIZUL HASNI BIN MANAB

ELECTRICAL & ELECTRONICS ENGINEERING
UNIVERSITI TEKNOLOGI PETRONAS
JUNE 2010

EDUCATIONAL PROCESSOR

By

HAFIZUL HASNI BIN MANAB

FINAL PROJECT REPORT

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

Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

EDUCATIONAL PROCESSOR

by

Hafizul Hasni Bin Manab

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:

____
(Mr. Patrick Sebastian)
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

June 2010

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.

Hafizul Hasni Bin Manab

ABSTRACT

This report discusses the overview of the chosen project, which is an Educational Processor. The objective of this project is to develop a simple processor with TTL logic for educational purpose. This processor will be used as a learning tool for Computer System Architecture class. To complete this project, the scope of study will cover the computer system architecture and Central Processing Unit (CPU). The CPU datapath design and hardware circuit design is based on the MIPS single-cycle processor. The methodologies that will be involved in this project are design and validation phase, constructing the hardware and then interfacing phase through serial communication between CPU and a graphic user interface using microcontroller. The prototype would be used as a learning tool in Computer System Architecture class and to assist student in understanding the computer architecture.

ACKNOWLEDGEMENTS

Special thanks to project supervisor Mr. Patrick Sebastian, Lab Technicians Ms. Siti Fatimah, Ms. Siti Hawa and Mr. Isnani who help me a lot in finishing this project.

Not to forget, greatest appreciation to my family who have supported throughout the development of this project as well as friends who have given a lot of helps.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

RISC	Reduced Instruction Set Computing
CISC	Complex instruction set computing
CPU	Central Processing Unit
RAM	Random access memory
EPROM	Erasable Programmable Read-only memory
PC	Program Counter
CSA	Computer System Architecture
TTL	Transistor-transistor Logic
OPCODE	Operation Code
PCB	Printed Circuit Board
ASM	Assembly Code
GUI	Graphic User Interface

CHAPTER 1
INTRODUCTION

This chapter discusses the introduction to this project. It covers the background of study that discuss the background knowledge involve in this project. The problem statement and objectives that lead to implementation of this project are also discussed.

1.1 Background of Study

This project is about a development of a simple processor for learning purpose, which will be used in Computer System Architecture class. The main objective is to provide an opportunity for the student to examine at the gate level on how a processor executes an instruction. This educational processor will be a great learning tool for computer system student to learn computer system architecture. Therefore, the knowledge required in this project is application knowledge of digital electronics as well as computer system architecture. This project also required knowledge in microcontroller since this processor would be interfaced to the computer by using microcontroller via serial communication. After that, all the information and operation involved during execution of an instruction by the processor will be shown in a graphic user interface.

1.2 Problem Statement

The processor is an essential part of a computer system. The development of the processor has involved over the years. In 1945, a mathematician John Von Neumann outlined the design of most modern CPUs [3]. Most of the processor designs now are very sophisticated and complex.

The Electrical and Electronics Engineering student in UTP who are majoring in Computer System have the opportunity to learn about computer system through Computer System Architecture course. The current Computer System Architecture course exposes the student to the course with lecture as well as hands on lab assignment.

Nevertheless, there is no main focus on any specific computer architecture. Most of the time, the overall CPU datapath design that they are exposed to be just the high-level functional unit block that explains the CPU datapath.

Therefore, this project would give an opportunity the Computer System Architecture student to explore and examine at the gate level of CPU datapath, which means the student can observe how each logic device interact with each other to complete a CPU instruction.

1.3 Objective and Scope of Study

The main objectives of this project are:

- To develop a simple processor as a learning tool in computer system class
- To construct the PCB and validate the prototype
- To develop a graphic user interface to program the designed CPU

The scope of work for this project covers the planning and design phase, developing the prototype phase, validating phase and the last is future improvement phase. In the planning and design phase, the scope of work will be focus on processor instruction set architecture and datapath. After that, it will followed by prototype developing phase where the data path hardware is implemented using the TTL logic implemented on PCB. In the validating phase, the datapath hardware will be interfaced to the computer with a graphic user interface by using microcontroller.

CHAPTER 2
LITERATURE REVIEW

This chapter discusses the theory and paperwork review related to this project. Details on the CPU architecture and datapath design would also be discussed here.

2.1 Procedure Identification

There are two ways of introducing the processor. One is to explain how a computer works of its internal information flow by describing the way in which information is transmitted between registers and the internal units and showing how an instruction is decoded and interpreted. The other approach is to introduce the native language, or machine code, of a computer and demonstrate what computer instructions can do [1].

2.2 Instruction Sets Architecture

Beginning with the hardware and looking at very primitive operations hides the "big picture". So, beginning with the explanation of an instruction set architecture would give reader the whole picture of a processor and therefore the detail hardware level of how an instruction is translated and executed could be easily understand.

An instruction set architecture (ISA) is an abstract model of a computer that describes what it does, rather than how it does it (functional definition). So, it can be said that the instruction set architecture and the instructions available in the processor determine the processor capabilities and performance [1].

The instruction set architecture varies from machine to machine. Instructions are classified by format and the number of operands they take. The three basics instruction types are data movement which copies data from one location to another, data processing which operates on data, and flow control which modifies the order in which instructions are executed.

Instruction formats can take zero, one, two or three operands. It depends on how much bit is used to represent the whole instructions. The instruction sets architecture can be distinguished by two classes which are the Complex Instruction Set Computing (CISC) and the Reduced Instruction Set Computing (RISC).

2.2.1 Complex Instruction Sets Computing

The CISC employs complex instruction which usually their instruction width (in bits) could vary depending on the type of instruction (data movement, data processing and flow control). For example there would be an instruction that consists of only the opcode (instruction identifier in bits) where it does not require any operation on the operand e.g.: Return from Subroutine (RTS).

Beside the variety of instruction width, each instruction could possibly be very complex in a way that it could perform operation with complex addressing. Complex addressing requires extra decoding and operation cycle. This is achieved usually through the usage of microcoding.

With variety of instruction width and complexity, it would take variable total of clock cycles to execute each instruction. It is because each instruction's opcode would be decoded firstly in the earlier cycle. The rest of the cycle would depend on the type of the instruction, which would make each instruction take at least two clock cycles to be executed.

2.2.2 Reduced Instruction Sets Computing

One of the important characteristic of a RISC is that having the single-instruction format in contrast to the variable-width (length) instruction of CISC. This reduces the complexity of the decoding logic itself and thus could be used to easily educate new student in learning computer architecture.

Typically a RISC instruction format would consist of Opcode + Registers addresses. RISC is designed to contain only the register-to-register operation while for the memory access operation, RISC introduce a special instruction which is Load/Store. Thus, addressing modes in RISC processor are not as sophisticated as CISCs'.

RISC processors aim to execute on average one instruction per clock cycle. This goal imposes a limit on the maximum complexity of instruction and so to the hardware design of a RISC processor.

The *table 1* below summarizes the differences between RISC and CISC processor.

Table 1 Comparison between RISC and CISC processor

	CISC	**RISC**
Instruction width	Variable instruction width	Fix instruction width
Instruction cycle	Multiple clock-cycle	Single clock-cycle
Addressing modes	Complex addressing mode for memory access	Register-to-register data transfer with special instruction for memory access

2.2.3 Instruction usage

From the introduction of the microprocessor in the mid 1970s to the mid 1980s there was an almost unbroken trend toward more and more complex architectures [1]. With the advancement in the chip fabrication process, it allows designers to add more to the microprocessor's central core, which leads to cumbersome architectures and inefficient instruction sets but has tremendous commercial advantage for the end user. Intel's 8086 illustrates this trend particularly well, because Intel took their original 16-bit processor and added more features in each successive generation [1].

Although processors were advancing in terms architectural sophistication in the late 1970s, a high price was being paid for this progress in terms of efficiency [1]. Complex instructions required complex decoders and a lot of circuitry to implement while there was no guarantee that these instructions would be used in actual programs.

Computer scientist carried out extensive research over a decade or more in the late 1970s into the way in which computers execute programs [1]. Theirs studies demonstrated that there is no uniform frequency in which different type of instructions are executed. Some types of instructions are executed far more frequently than others.

Fairclough divided machine-level instructions into eight groups according to type and compiled the statistics described by Table 2. The mean value represents the result averaged over both program types and computer architecture.

Table 2 Frequency of instruction usage

Instruction Group	Mean Value (%)
Data movement	45.28
Program modification (branch, call, return)	28.73
Arithmetic	10.75
Compare	5.92
Logical	3.91
Shift	2.93
Bit manipulation	2.05
Input/output and miscellaneous	0.44

2.2.4 RISC instruction format

One of the characteristics of RISC architectures is that it has a single instruction format. By providing a single instruction format, the decoding of an instruction into its component fields can be performed by a minimum level of decoding logic. A RISC's instruction length should be sufficient to accommodate the operation code field (opcode) and one or more operand fields [1]. Consequently, a RISC processor may not utilize memory space as efficiently as a conventional CISC microprocessor.

Figure 1 describes the format of a Berkeley RISC instruction, one of the first RISC processor that came from the University of California at Berkeley.

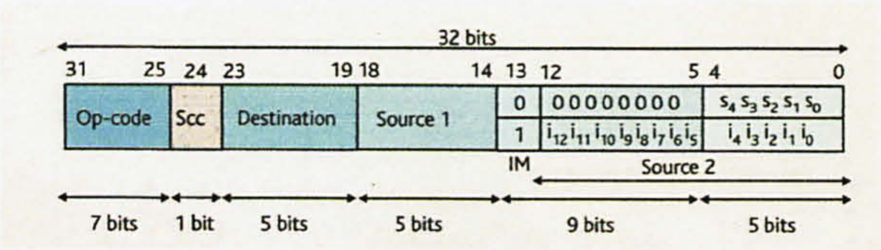


Figure 1 RISC Berkeley's instruction format

The op-code field is the Operation code field that indicates the code for each instruction. Each instruction has its own unique Op-code. Scc field whether the condition code bits are updated after the execution of an instruction. Destination and Source 1 fields determine the address of register of which the result would be written into and the first source for an instruction's operand, respectively. The IM field determines the source for another instruction's operand. If it is 0, the source is the 5-bit address of the registers while if it is 1, the source is the 13-bit immediate number.

Because of 5-bits are allocated to each operand field, it follows that this RISC can access up to $2^5 = 32$ internal registers at a time.

2.2.5 Addressing modes

Addressing mode is the method by which the location of an operand is specific within an instruction. Some of addressing modes most commonly used are describe as follows.

1. *Immediate addressing*. Operand is given in the instruction itself. Usually the second source of instruction's operand is supplied as part of the instruction.

2. *Address registers indirect addressing*. Operand is taken from, or result placed in, a pointer register. RISC processors allow any registers to act as a pointer.

3. *Base addressing*. Operand is in memory and its location is computed by adding of offset to the content of a specified base register. In RISC processor, this type of addressing mode is used by the Load and Store instruction to access RAM. The computed result would be asserted to the address of the RAM, and then the data would be stored into that location or the data is loaded into the destination register from that location.

4. *PC-relative addressing*. Same as base addressing, but with the register always being the program counter.

2.3 CPU Functional Units

Before looking into the details of how a CPU works, it is important to understand the relationship between the CPU, the memory and the program. The program contains list of instructions to be executed by the processor, for example the applications and software that available nowadays. The memory temporarily stores the list of instruction of the program and also the data of the program during CPU execution. The CPU read one-by-one list of instruction of program from the memory and perform the required execution on the data and probably store back the data in the memory.

In this project, the focus is on how this list of instructions in fetched from the memory, decoding the instructions producing the appropriate control signal, perform instruction-specific execution on the data and probably store the result of execution back into the memory. This process is shown in the project by the combinational logic circuits that make up CPU internal units which do specific job to complete one instruction. CPU internal units include are as follows:

2.3.1 Program Counter (PC)

Program counter contain the next instruction address to be executed. This address will be input the program RAM to access a specific line of instructions. Normally, PC would be increased after every instruction executed to point to the next address except if flow control instructions is executed which modify the bits contain in the PC.

2.3.2 Program/Instructions Memory

Program memory contains the list of instruction to be executed. In Von Neumann architecture machine, program memory and data memory use the common RAM, while in Harvard architecture machine, program memory and data memory use separate RAMs.

2.3.3 Instruction Register

Instruction register contains the current instruction. It stores the current register temporarily and connects to various other logic devices such as control logic, and register files. When the next instruction is executed, it will overwrite the content of this instruction register.

2.3.4 Register File (General Purpose Registers)

In RISC machine, register files are the important characteristic. It serves as the general purpose register to store temporary data that is executed by specific instruction. Register files are pretty similar to the RAM except that it doesn't have as much capacity as RAM and thus reduce the cost. Typically, registers are faster than RAM that makes execution of register-register instruction could be faster.

2.3.5 Arithmetic and Logic Unit

ALU is the unit that does the manipulation to the data such as addition, subtraction, logical AND, logical OR and many more.

2.3.6 Data Memory

Data memory is the storage device that store data from the program executed. It could be the constants, variables, address etc. Normally, data that are stores here are not a frequently used data as accessing the memory is slow thus make the program execution slower.

2.3.7 Control Logic

Control logic is among most important modules that make up a processor. It controls the sequence and datapath flow of an instruction. When an instruction is executed, it fetch and decode the opcode of that instruction and output the control logic signals to the appropriate modules such as register files, ALU and memory.

2.3.8 Address Bus and Data Bus

Bus is used to simplify the movement of data from point to point in a computer. Bus is analogous to a highway and the devices are analogous to junctions that connect to this highway. By having both address bus and data bus, it is possible to reduce the number of wires that interconnect within a computer but, it introduces a complexity. In a bused system, only one communication from point to point could happen at a time. Thus a careful synchronization needs to be taken care of and each bus access time has to be long enough for the safe reception in a communication.

2.4 Brief introduction to MIPS processor

MIPS processor is designed in 1984 by researchers at Stanford University. MIPS is part of RISC processor family.

Like the other processor in RISC family, MIPS employs load-store architecture. This means that there are two instructions for accessing memory, a Load instruction to load data from memory, and a Store instruction to write data into memory. It also means that none of the instruction can access memory directly. To do operation on data, the data has to be loaded into registers and the operation is performed on the data in the register. As most of the instruction operations are between registers, they allows faster execution and simpler circuit design.

MIPS processor executes instruction in a single clock cycle because of the nature of a RISC processor which is single instruction format. This fact allows the MIPS instructions to be split into stages for implementing pipelining. The stages are:

1. **IF** – *Instruction fetch.* Fetch the next instruction from memory using the address in Program Counter register and stores the instruction in Instruction Register.

2. **ID** – Instruction decode. Decode the instruction in the Instruction Register, calculate the next Program Counter, and read any operand required from the register files.

3. **EX** – Execution stage. Perform arithmetic and logic operation.

4. **MA** – Memory access. Perform any memory access required by the current instruction.

5. **WB** – Register write back. For any instruction with destination register specified, it writes back the result into the destination register.

By splitting instructions into different stages, it results in 5-clock cycle execution. But, with pipeline implementation, this technique would attempt to execute instructions approximately in one clock-cycle. Table 3 below shows the pipeline implementation in MIPS processor.

Table 3 MIPS pipeline architecture

CYCLE	IF	ID	EX	MA	WB
1	i				
2	i+1	i			
3	i+2	i+1	i		
4	i+3	i+2	i+1	i	
5	i+4	i+3	i+2	i+1	i
6	i+5	i+4	i+3	i+2	i+1
7	i+6	i+5	i+4	i+3	i+2

Examining the table above, it can be clearly seen that from the fifth cycle, the first instruction is completed. Then, at the next cycle, the i+1 instruction (next instruction) is completed. This goes on the same towards further cycle. Although it takes five cycles to complete an instruction, but approximately the instructions are executed in one cycle. This is explained before, from the fifth cycle onwards, each instruction execution is completed.

The words "approximately" from the second last statement do carry an important meaning. Theoretically, the single-cycle approximation could be achieved based on the explanation before. But in reality, there are some dependencies of an instruction to another instruction. It means that some instruction stage could not be executed before it gets the valid data from instruction before. This hazard introduces "waiting delay", which makes the single-cycle approximation could not be achieved.

The MIPS instruction has three basic formats. Figure 2 below illustrates the of the MIPS instructions.

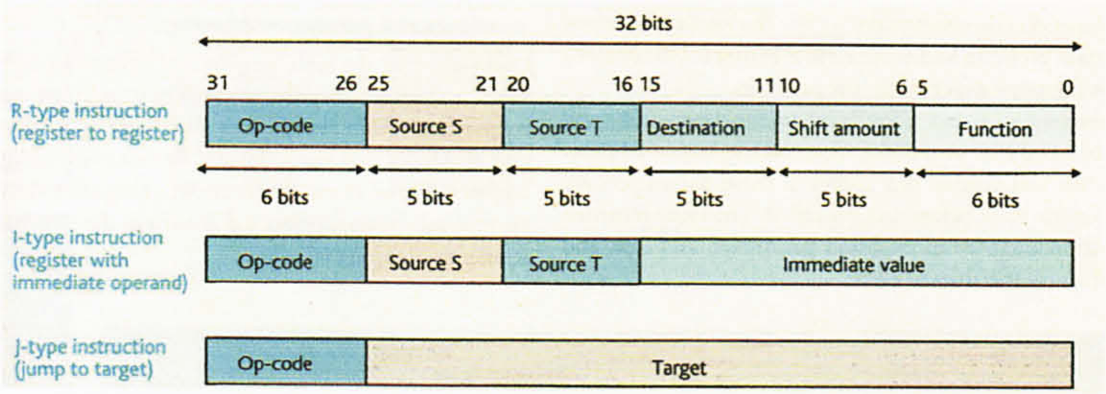


Figure 2 MIPS instruction formats

R-type instruction is a register-to-register format for all data processing instructions. I-type instruction is immediate format for either data processing instructions with a literal or load/store instructions with an offset. While J-type instruction is the format for branch/jump instruction with a 26-bit literal that is concatenated with the six most-significant bits of the program counter to create 32-bit address.

CHAPTER 3
METHODOLOGY

This chapter discusses how the project is carried out. It includes the method of research, tools and software involved.

3.1 Project Flowchart

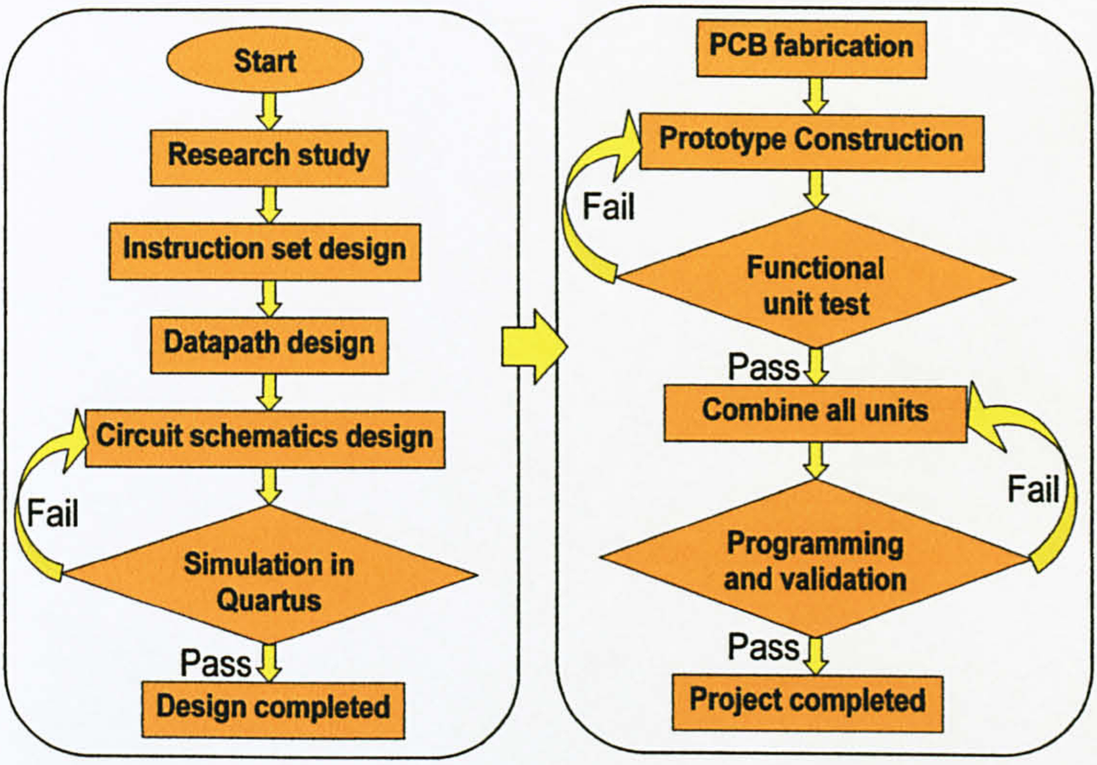


Figure 3 Project flowchart

3.1.1 Research Study

In research study phase, the theory behind the CPU design and CPU architecture is studied. This includes the study on the CISC and RISC architecture design of a CPU. The research mainly focuses on the decision between these architectures that would best educate student.

After decision is made, the research continues on the details of RISC architecture. To understand the CPU architecture, the knowledge on these theories is important, which include instruction set architecture, CPU functional units and CPU data path. These theories are explained in the Literature Review chapter before.

The sources of research include from the books, websites, and journals. The author's participation in Computer System Architecture classes has also contributed to the research study.

3.1.2 Instruction set design

In this stage, instruction sets architecture is designed. This defines the whole identity of the processor itself. Since the processor would have a very limited instruction set, thus the choice of instructions have been made according to research that shows the most commonly used instruction in a program.

Design starts with the format of the instruction design. The instruction format defines the width of the instruction, op-code field and operand fields. Figure 4 below illustrates how the instructions format is design.

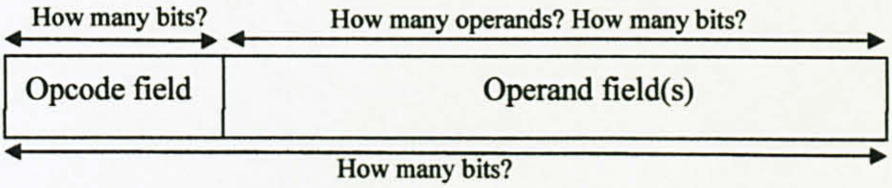


Figure 4 Instruction format design

Concurrently, the selection of instructions to be included in the CPU is done. As explained before, the selection is done based on the most commonly used instructions in a program. Each of the selected instruction is then assigned with specific operation code (opcode). Their operands are then fitted accordingly. This means that each instruction would have different operands type as well as number of operands to be fitted with the designed instruction format. Figure below illustrates how the operand field for R-type instruction is fitted.

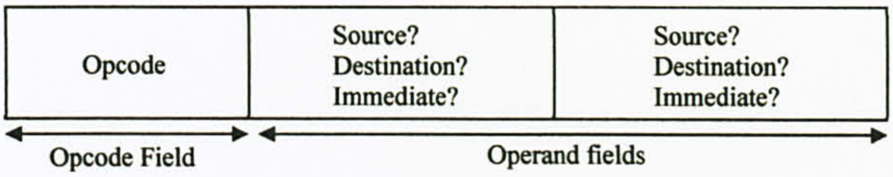


Figure 5 Operand fields design for R-type instructions

3.1.3 Datapath design

In this stage, the datapath of each chosen instructions are design. This is the last stage of CPU design. Datapath determines the connectivity between each CPU's functional units with each other. It translates an instruction into the hardware that does the execution to complete the instruction. The path for data movement from the start at the instruction fetch from memory towards the end, data write back into memory, is constructed. Hence, it is called "datapath".

To design a datapath, the formats of instructions are examined. For this project, there are 3 types of instruction formats that categorized all the instructions. This is discussed later in the next chapter. The purpose of identifying the formats of instructions is because instructions with the same format would have the same datapath.

Next is examining CPU functional units operations during the execution of a certain instruction format. For example, what does the Program Counter do? Does it increment to next instruction or it fetch address for branch instruction? Does register files do read operation only or both read and write operation? Does ALU is executing on the data or it does nothing? Does data memory access is needed or not? All of these factors determine the datapath of an instruction.

Knowing the operation of each CPU functional units, datapath for each instruction format is then designed. The final step would be combining those datapath for each format together to form the whole CPU datapath. The datapath designed for this project would be discussed in the next chapter, which is result and discussion chapter.

3.1.4 Circuit schematic design

In this stage, the datapath designed earlier is translated into combinational logics circuit. Each CPU functional unit logic circuit is constructed.

The schematic is designed using the Quartus software. Verilog HDL codes are written to simulate TTL devices such as registers, multiplexers and so on. Then, the CPU datapath logic circuit is constructed from these block diagrams of TTL devices emulations.

The purpose of designing the schematic using the Quartus is to allow simulation of the CPU. In fact, this schematic could be directly downloaded into an FPGA.

3.1.5 Simulation

In this stage, simulation of the designed schematic is done. The purpose of simulation is as the first stage of error debugging.

Using this simulation, it provides the timing waveform of signals. These signals are examined whether it behaves as it should be in the datapath designed earlier. This fact prevents major debugging to the circuit later, because if there is any modification to the datapath required, it could be done in software rather than hardware which are tedious and costly.

3.1.6 Prototype construction and module/unit test

In this stage, the prototype is developed according to the schematic designed before. Development are done phase by phase according to CPU functional unit (PC, register files, etc).

After developing each CPU functional unit, it is put into a test. The test is done by invoking all possible inputs to the unit and verifies the output signals produce. The error is expected to be just the wrong connections, pins not connected or the TTL devices not functioning, if any.

3.1.7 Combine modules (CPU functional units) and test

After completing the entire module, the modules are combined together producing the whole processor. It is then put into a test again. The test procedure is done by loading a program into the processor and then executes.

In the design, the clock circuit is built in such a way that it can produce a single clock-cycle at a time. Thus, the program could be executed instruction by instruction.

For each instruction, the signals from each device each checked to ensure that all are functioning accordingly. After completing a program, the test is repeated again by loading the same program. This is to ensure the consistency of the circuit behavior.

CHAPTER 4
RESULT AND DISCUSSION

This chapter discusses the result from the design phase and the simulation phase as well as the construction phase. The problem arise along those phases would also be discussed here.

4.1 Design phase results

4.1.1 Instructions designed

Instruction sets that are designed are as follows:

Data movement instruction:

LDR rd, rs - load data from memory location pointed by register *rs* into register *rd.*

STR rd,rs - store data from register *rd* to memory location pointed by register *rs.*

MOV rd,rs - move (copy) data from register *rs* into register *rd.*

MOVI rd,Imm - move immediate (literal) value data into register *rd.*

Data processing instruction:

ADD rd,rs - Add content of register *rs* to content of register *rd* and store the result into register *rd.*

ADDI rd,Imm - Add immediate (literal) value data to content of register *rd* and store result into register *rd.*

SUB rd,rs - Subtract content of register *rs* from content of register *rd* and store result into register *rd.*

SUBI rd,Imm - Subtract immediate value data from content of register *rd* and store result into register *rd.*

CMP rd,rs - Compare content of register *rd* and content of register *rs* and set the condition code register (status register)

CMPI rd,IMM - Compare content of register *rd* with immediate value and set the condition code register (status register).

Flow control instruction:

BNE rt,Imm - Check the CCR for zero flag, if not set, change PC to point next address pointed by content of register *rt* + immediate data.

BLT rt,Imm - Check the CCR for negative flag, if set change PC to point next address pointed by content of register *rt* + immediate data.

JMP rt,Imm - Unconditional jump (subroutine/function call) to address pointed by content of register *rt* + immediate data.

RTS - Return from subroutine. Restore the pc with next address from *stack pointer*.

END - Halt or stop the cpu operation. Terminate the program/end of line.

4.1.2 Instruction set formats

Instruction set architecture designed is using 8-bit word. Four most significant bits are the Operation code (opcode) field while the rest four-bits are the Operand field. The instructions could be divided into three formats according to MIPS processor which include:

1. R-type instructions (register-to-register instruction)

7 4	3 2	1 0
OPCODE	Source 1 register Destination register	Source 2 register

Figure 6 R-type instruction format

2. I-type instructions (immediate operand)

7 4	3 2	1 0
OPCODE	Source 1 register Destination register	immediate data

Figure 7 I-type instruction format

3. J-type instructions (branch/jump instruction)

7 4	3 2	1 0
OPCODE	target/pointer register	immediate offset

Figure 8 J-type instruction format

4.1.3 Datapath design result

Datapath is designed based on the three formats above explained in the chapter 3. The first datapath is for the R-type instructions. The result is as follows:

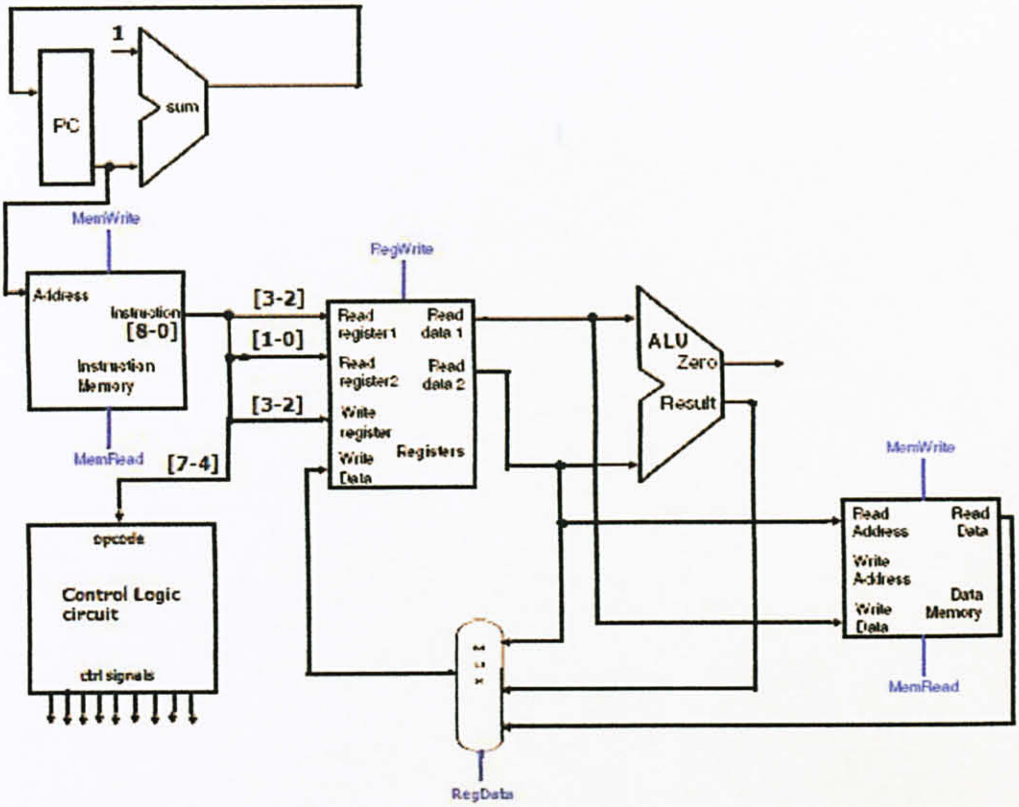


Figure 9 R-type instruction datapath

This datapath is the same for all instructions as follows:

- ADD
- SUB
- MOV
- CMP
- LDR
- STR

As shown in figure 9, the PC is incremented by 1 only. This is because these instructions do not affect the program sequence. Bit 7-4 is asserted to the control logic opcode address to produce appropriate control signals. Bits 3-2 is asserted for address Read register 1 and also address write register. This provides the first operand for the instruction as well as the destination register to be written into. Bit 1-0 is asserted for Read register 2 providing the second operand for the instruction. The multiplexer is there to select source for data to be written into destination register. This is because instruction like MOV instruction does not require ALU operation, instruction like ADD instruction requires ALU operation and LDR instruction that take the data from the data memory.

The second datapath is for the I-type instruction. The result is as follows:

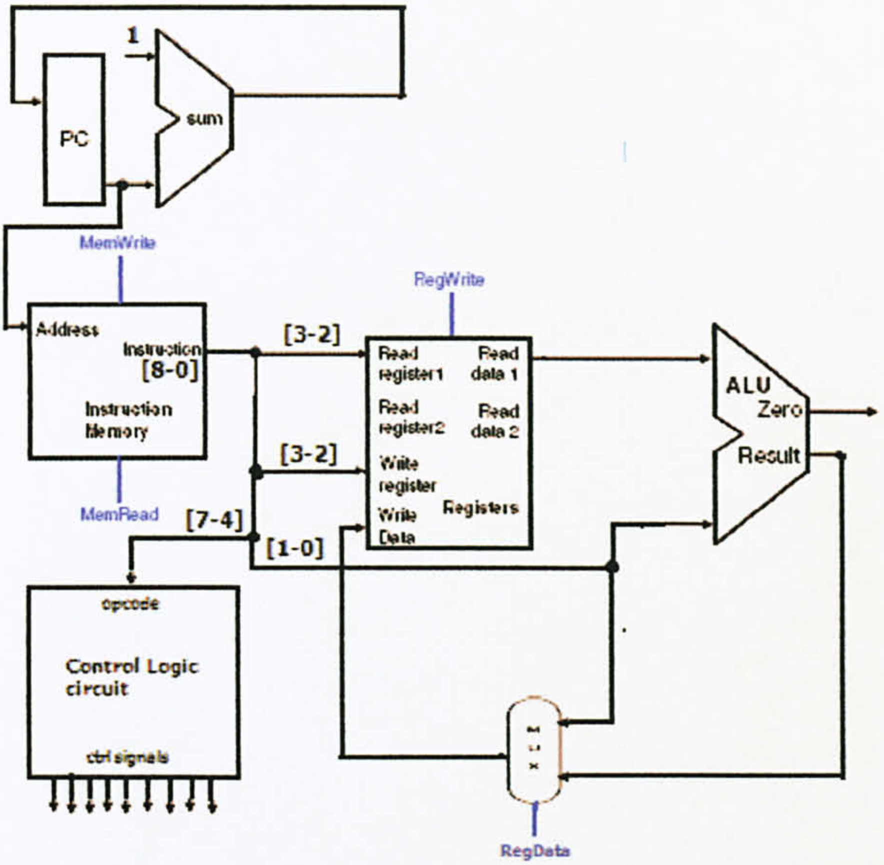


Figure 10 I-type instruction datapath

This datapath is the same for all instructions as follows:

- ADDi
- SUBi
- MOVi
- CMPi

As shown in figure 10, the PC is also incremented by 1 only. This is because these instructions do not affect the program sequence. Bit 7-4 is asserted to the control logic opcode address to produce appropriate control signals. Bits 3-2 is asserted for address Read register 1 and also address write register. This provides the first operand for the instruction as well as the destination register to be written into. The difference between R-type datapath and I-type datapath is the bit 1-0 the two bits source for second operand of the instruction. The multiplexer is there to select source for data to be written into destination register. This is because instruction like MOV instruction does not require ALU operation while instruction like ADD instruction requires ALU operation. Another difference from the R-type instruction is that these instructions do not involve data memory access.

The third datapath is for the J-type instruction. The result is as follows:

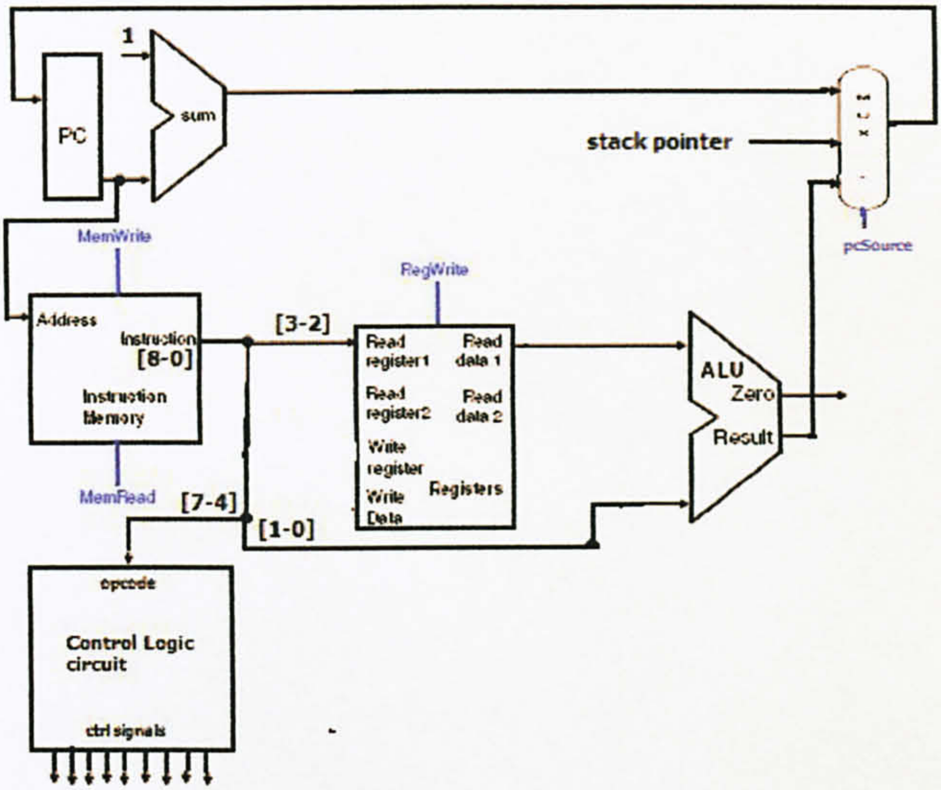


Figure 11 J-type instructions datapath

This datapath is the same for all instructions as follows:

- BNE
- BLT
- JMP
- RTS

This format of instruction differs from the two formats before because it modifies the program sequence. PC is updated with either from three sources which are increment by 1, stack pointer data, or ALU results. PC updates from source increment by 1 if the branch condition is not met, thus resulting no branch operation is executed. PC updates from source stack pointer data for RTS instruction. This instruction restores PC with the content of the return address when the JMP instruction is executed before. PC updates from source ALU results if the branch conditions is met, thus resulting in branch operation. The address for the branch subroutine is given by the operand 1 plus with the two bits immediate data.

The combined datapath is then constructed. The result is as follows:

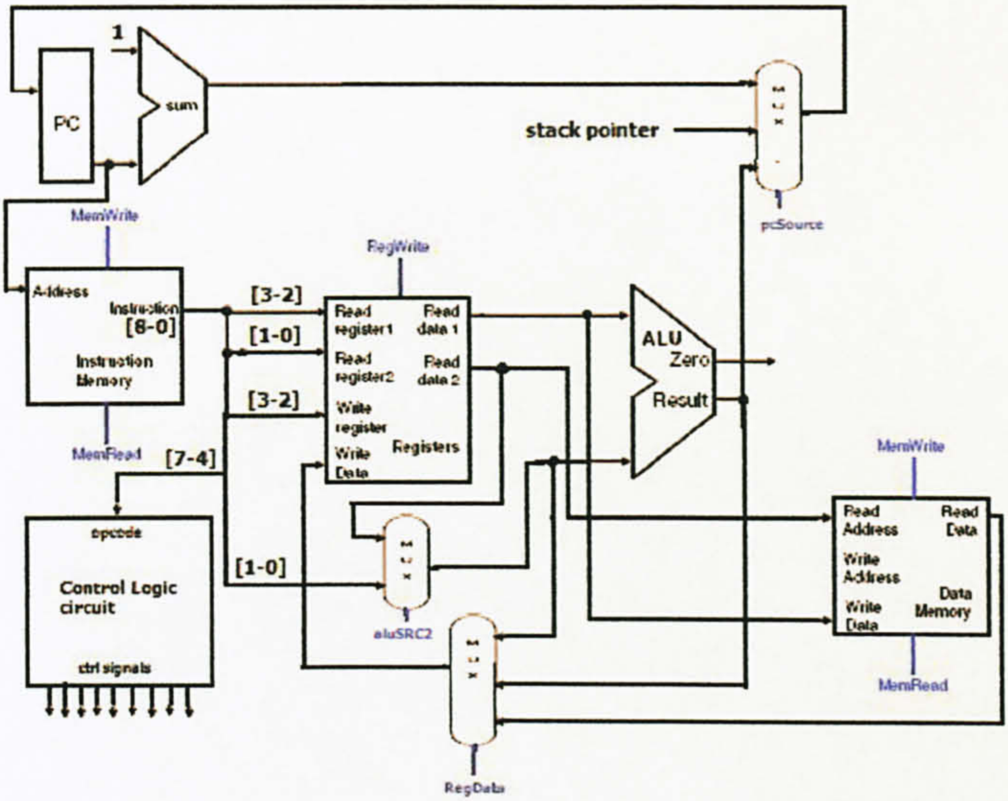


Figure 12 Combined datapath

As shown in the figure 12, all the datapath designed are then combined together. To execute certain instruction in a certain instruction format group, control signals would have to play this role. It controls the CPU functional units so that only the affected functional units are "activated".

4.1.4 Circuit schematic design

The designed schematics are provided in the appendix. Those schematics include:

- Program Counter circuit schematic
- Instruction RAM and programming circuit schematic
- Control Logic circuit schematic
- Registers file circuit schematic
- ALU unit circuit schematic
- Clock circuit schematic
- Data RAM and busses circuit schematic
- LED driver circuit schematic
- Input output circuit schematic

4.1.5 CPU characteristic

4.1.5.1 Programming the processor

As the Instruction sets include the jump (JMP) instruction, two separate RAM is used; one for the instruction memory and the other one for data memory. To program the processor, the program will be written into the instruction memory manually by using the switch to write data into the memory.

4.1.5.2 Processor control

To allow the greater flexibility of the processor, microcontroller is used to store control signals instead of using control matrix to drive the control signal to each functional units. This also can reduce defect probability as using lots of gate in the control matrix.

4.1.5.3 Arithmetic and Logic Unit

Based on the reference book [1], there is only ADDER IC that is used in the processor. This adder can be modified to perform subtraction too. But then, this limit the capability of the whole processor. Thus, the author has used 74181 chip to replace the former one. This allows the greater option of instructions to be included in the processor. The appropriate flag generation has also been designed for the instruction that require the comparison between two operands.

4.1.5.4 Registers and data RAM

Four general purpose registers is used in the processor. This allows the flexibility of such instruction as ADD and SUBTRACT. 2Kx8 RAM has been selected to be used for storing the data of the instruction's result.

4.1.5.5 Summary of CPU operation

All of the instruction will be executed in single-cycle. Half cycle of the operation is used to decode the instruction and send the control signal, data processing would also happen during this time. Another half cycle is for register writeback operation.

4.1.5.6 PCB Implementation

After the design of the CPU is simulated using the quartus software, the next step is to implement it into PCB. To design the PCB, the schematics for all CPU's functional units is drew using EagleCAD software. After that, the schematics are netlisted to board files. Here the actual PCB board layout is designed before it will be printed as shown in Appendix D.

4.2 CPU Functional Unit Control Signals

The control logic signal is configured to all CPU functional units for each instruction. The control signal will control what action the CPU functional units should takes. Table below describe the control signal of the respective CPU functional units.

Table 4 Control signal

Asm	OpCode				Program Counter					ALU								Data RAM						I.O	Mux1	Mux2	
	7	6	5	4	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	C3	C2	C1	C0	B0	B1	B2	B3	B4	B5	B6	B7	C4	C5	C6	C7	D0	D1	D2	D3	D4	D5	D6	D7	E0	E1	E2
LDR	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	0	1	0	0	1	1	1
STR	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1	1	1	0	1	0	0	0	0	1	1	0
MOV	0	0	1	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	0	0	0	1	0	0
MOVI	0	0	1	1	1	0	0	1	0	0	0	0	1	0	0	1	1	0	1	1	1	1	1	0	0	1	0
ADD	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	1	1	0
ADDI	0	1	0	1	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	0	1	0
SUB	0	1	1	0	1	0	0	1	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	0	1	1	0
SUBI	0	1	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	1	0
CMP	1	0	0	0	1	0	0	1	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	0	1	0	0

CMPI	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1	0	0	0	0
BNE	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1
BLT	1	0	1	1	0	1	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1
JMP	1	1	0	0	0	1	0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1
RTS	1	1	0	1	0	1	1	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	0	0	1	0
END	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1

4.3 CPU Hardware Fabrication

The results of the CPU fabrication phase are the overall circuit built using the TTL logic on Printed Circuit Board (PCB). Figure below illustrates the fabrication phase.

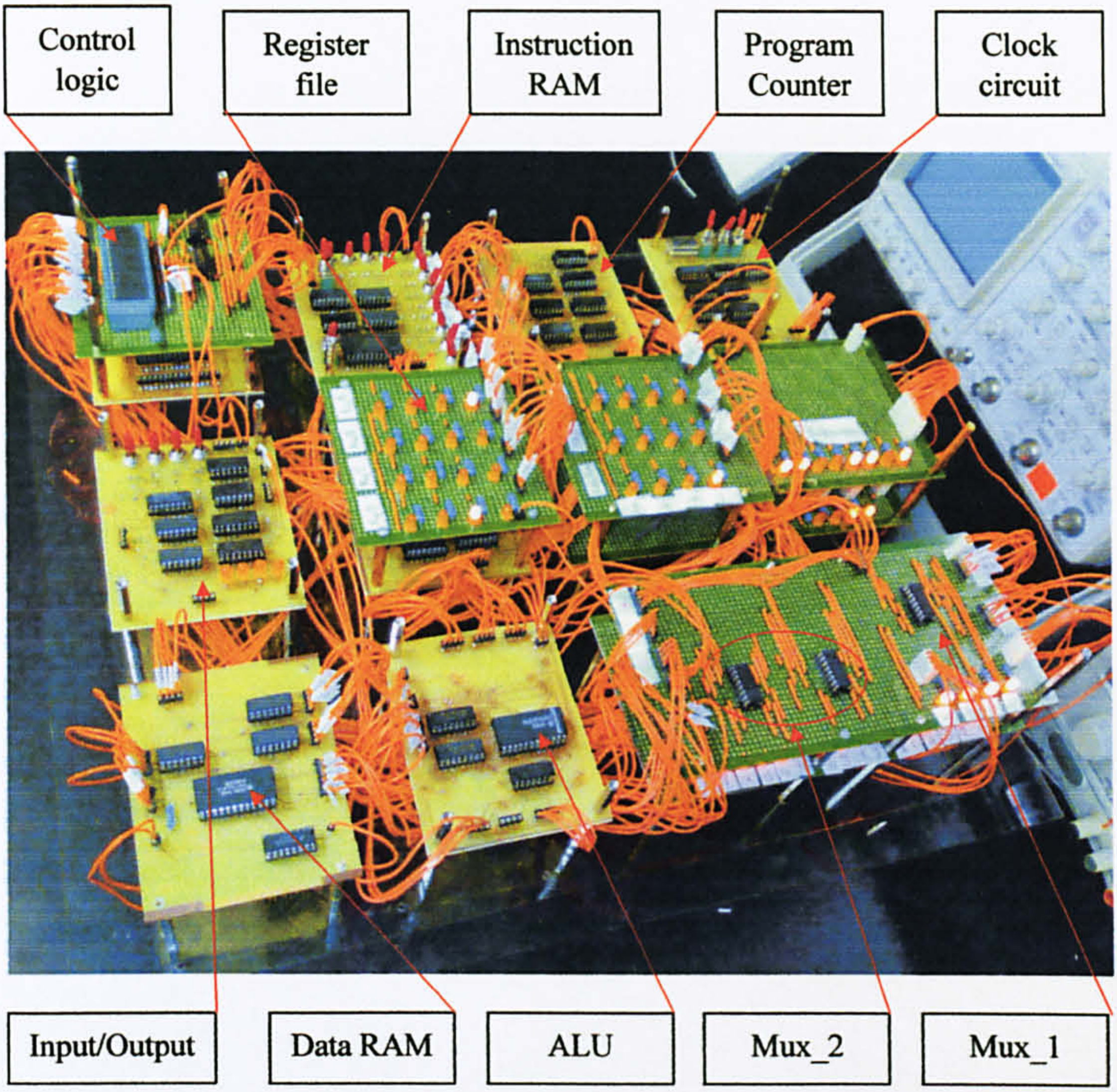


Figure 13 CPU fabricated

4.4 Graphic User Interface

The result of the graphic user interface is a Windows application that was programmed using Visual Basic to send 8-bit instruction one by one to the CPU. User needs to program the CPU by writing the instructions from this GUI and then the instructions will be written to the CPU's instruction RAM. Figure below shows the programmed graphic user interface.

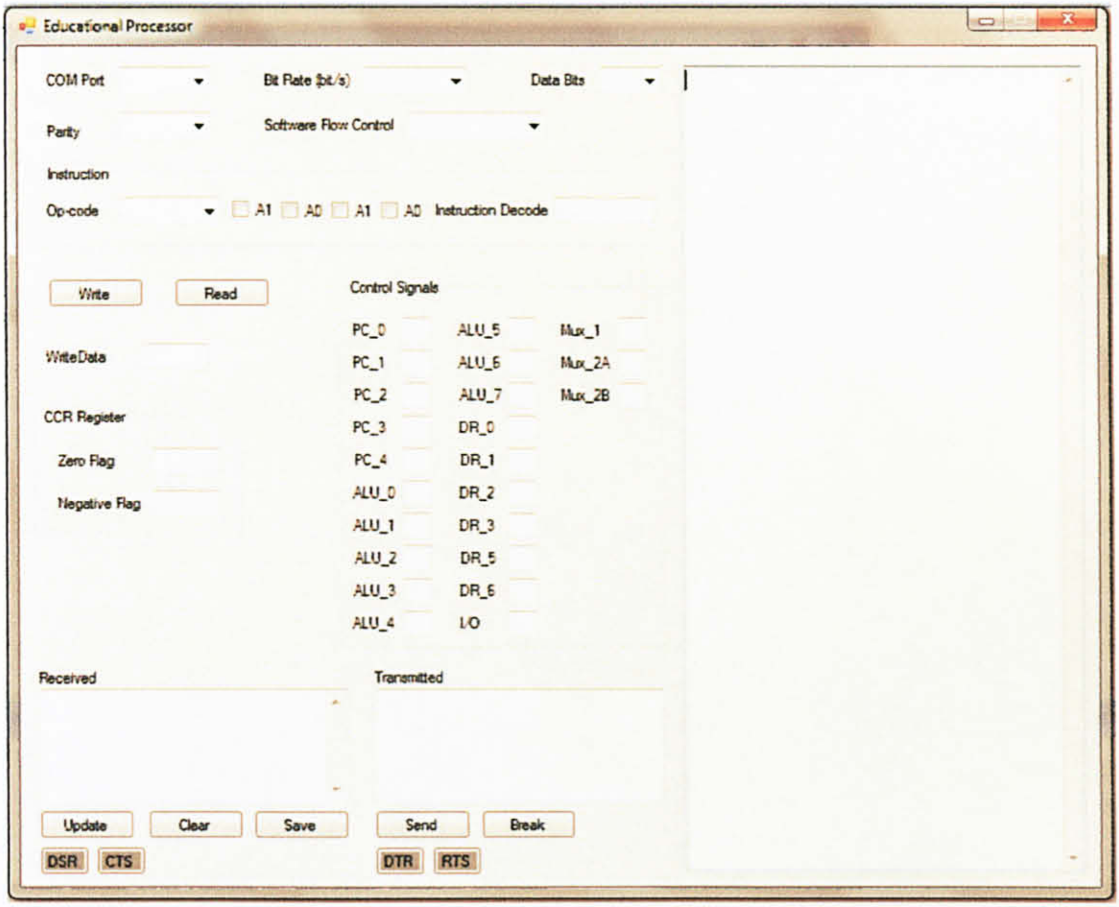


Figure 14 CPU graphic user interface

4.5 CPU-GUI Interfacing

The result for processor and computer interfacing is the serial communication between microcontroller and serial port of the computer. All instructions are sent from the GUI to the CPU through this serial communication. Figure below illustrates the full system of the designed educational processor.

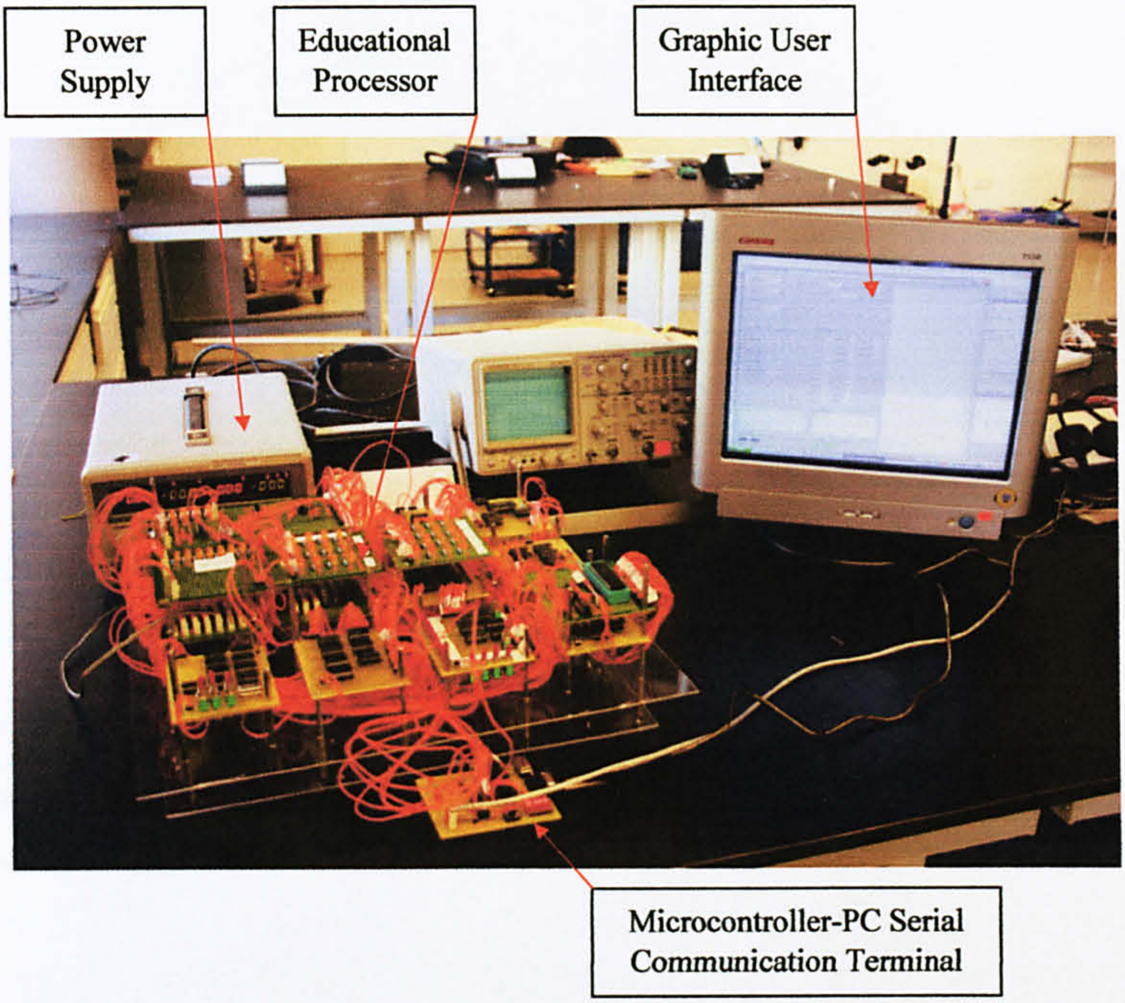


Figure 15 Full system Educational Processor

4.6 Testing Procedure and Validation Result

For validation of the designed processor, the hardware is tested by running some test programs and the results are observed by comparing the outputs with the expected result.

Five simple test programs are used to test all the instruction including data movement, data processing (ADDi,SUBi) and flow control (BRANCH).

Table 5 First Test program

Address	Instruction	Op-code	Operand1	Operand2
0001	ADDi A,#3	0101	00	11
0010	SUBi A,#1	0111	00	01
0011	CMPi A,#0	1001	00	00
0100	BNE B,#2	1010	01	10
0101	END	1111		

This program will test the BNE instruction that does the looping until Register A content is subtracted until 0. To run the test program, below are the testing procedures.

1. The Educational Processor GUI application is opened and the GUI's serial communication setting are configured set as below:

Table 6 GUI's serial communication configuration

COM Port	COM4
Bit Rate (bit/s)	9600
Data Bit	8
Parity	None
Software Flow Control	None

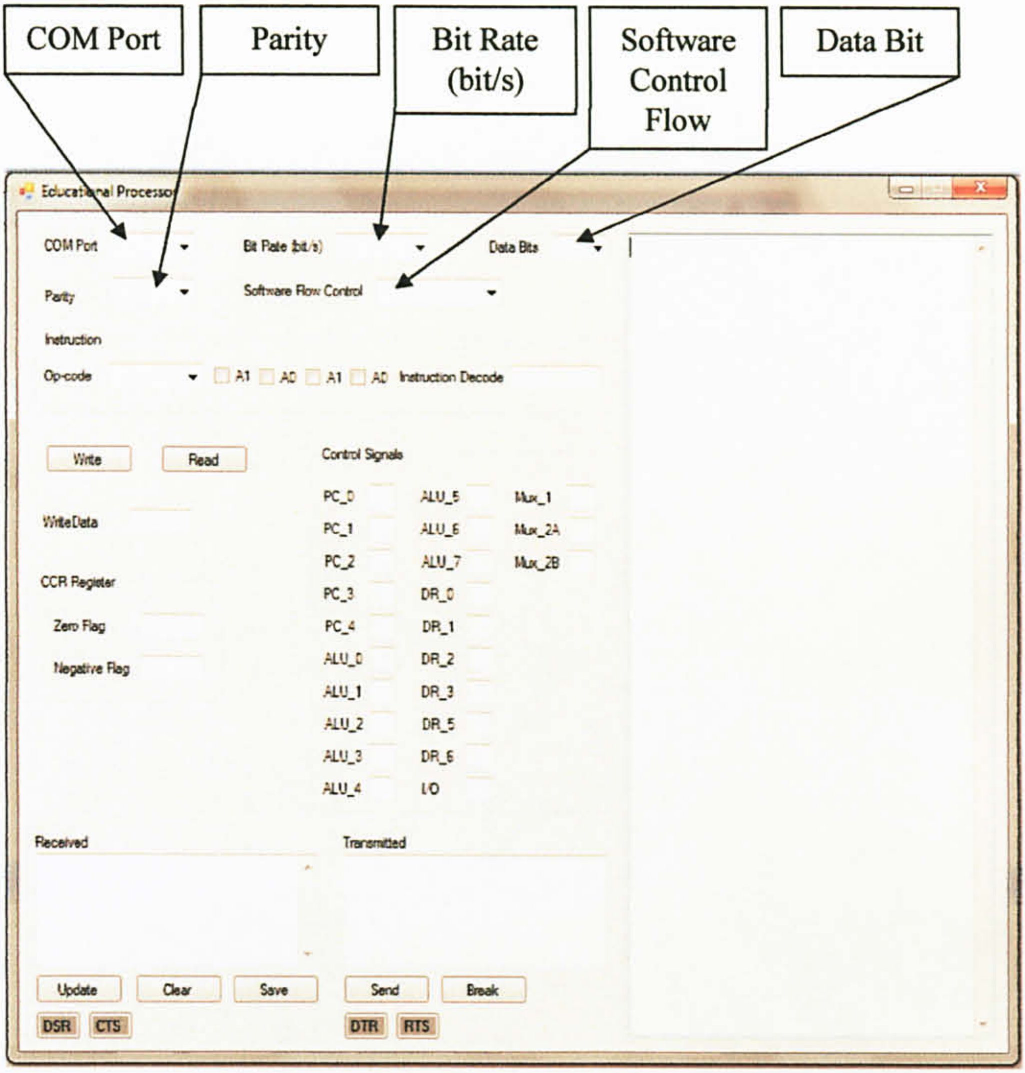


Figure 16 GUI's serial communication configuration

2. To write the first instruction in the instruction RAM, the first address is assigned manually from the hardware by changing the state of address switches as below.

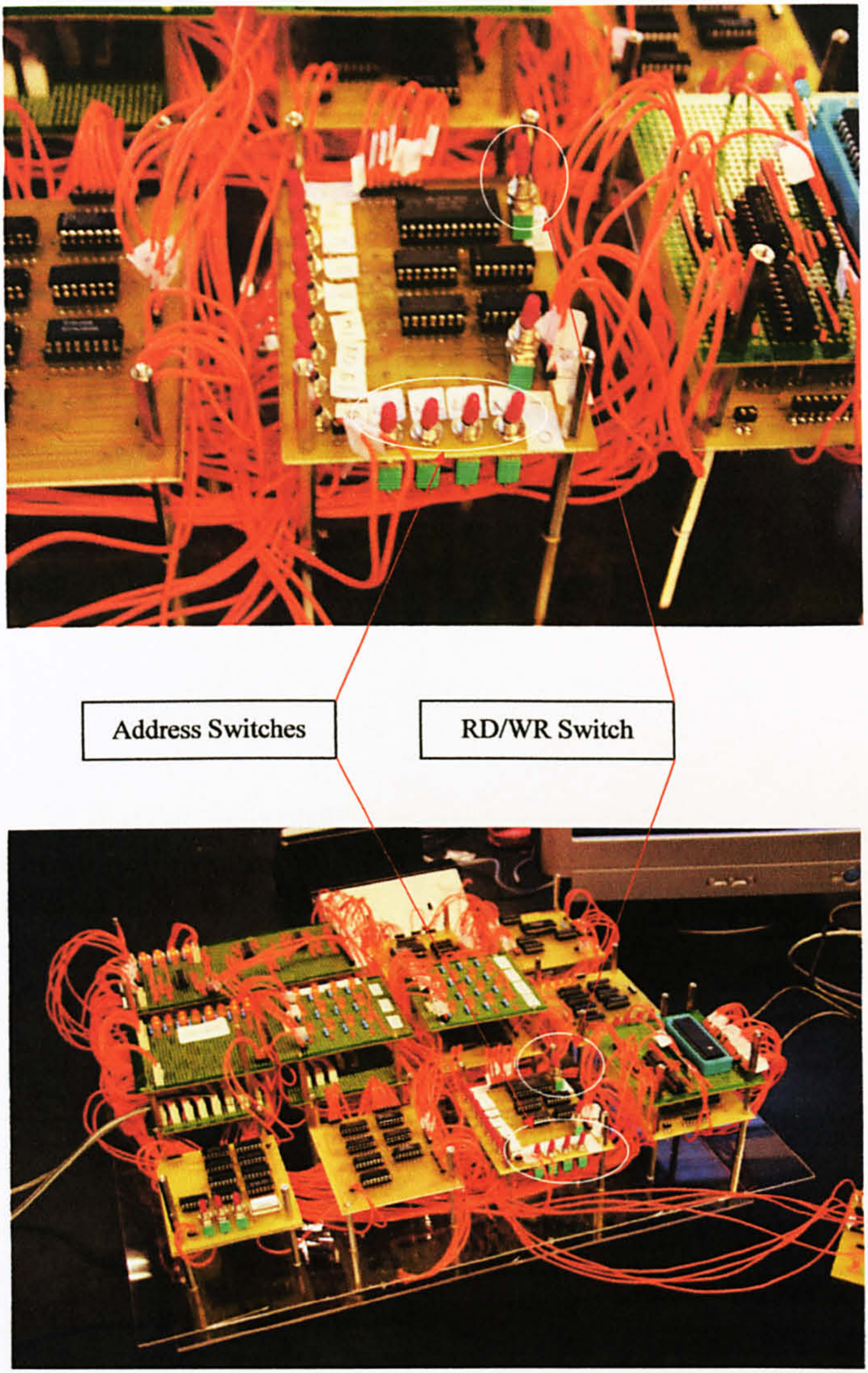


Figure 17 Address switches and RD/WR switch

3. After the address of the first instruction is ready, which is equal to 0001, the first op-code and the respective operands are selected as shown below. The

instruction that is to be written is shown in the Instruction textbox which is equal to 01010011. Now the first instruction is ready to be sent.

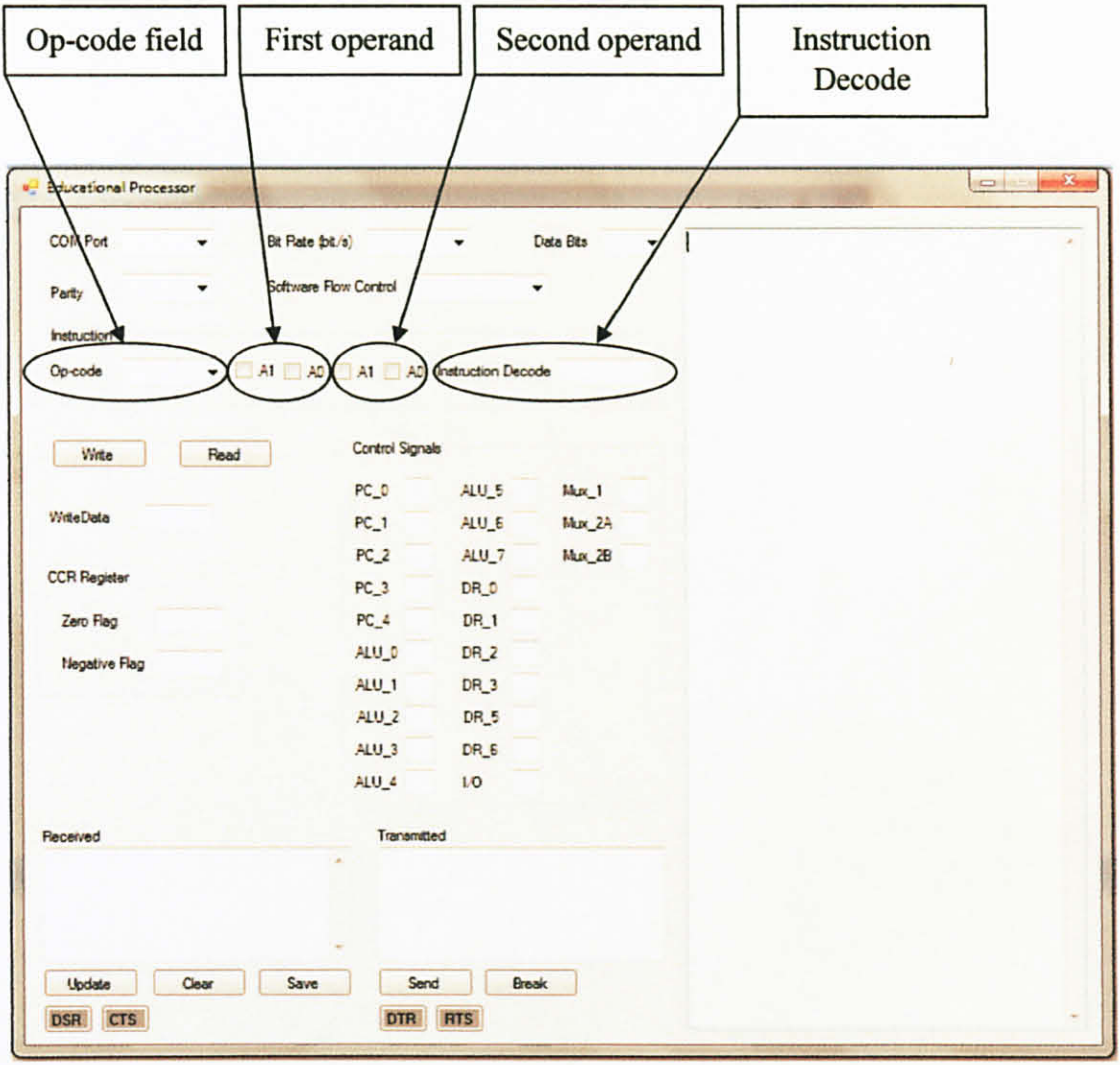
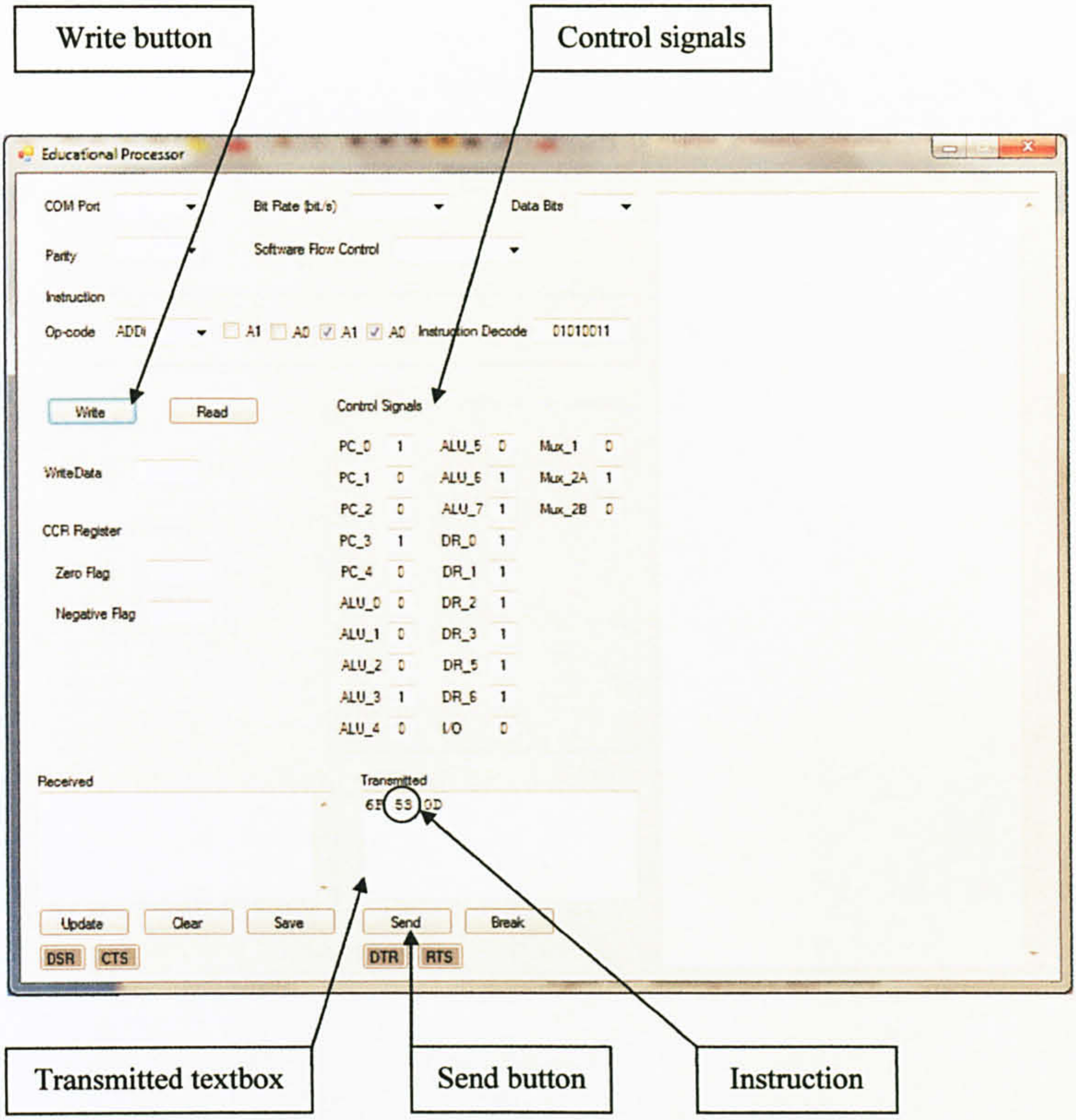


Figure 18   Writing a CPU Instruction

4. To send the first instruction, the “Write” button is clicked and the instruction is transferred to the “Transmitted textbox”. Here the instruction is prepared in order to be transmitted to the CPU hardware through microcontroller via 8-bit serial communication. Figure below illustrates this step. To transmit the content in the “Transmitted textbox”, “Send” button is clicked. The instruction is framed with 6F and 0D which is refer to character “o” as output and carriage return respectively. The microcontroller will receive three characters which are 6F, instruction and also 0D, and then the microcontroller will give

an 8-bit output as instruction to the hardware. At the same time, the appropriate control signals are shown to ensure that the op-code of the instruction will decoded to the right control signal.



Write button

Control signals

Instruction

Op-code ADDi □ A1 □ A0 ☑ A1 ☑ A0 Instruction Decode 01010011

Control Signals

PC_0	1	ALU_5	0	Mux_1	0
PC_1	0	ALU_6	1	Mux_2A	1
PC_2	0	ALU_7	1	Mux_2B	0
PC_3	1	DR_0	1		
PC_4	0	DR_1	1		
ALU_0	0	DR_2	1		
ALU_1	0	DR_3	1		
ALU_2	0	DR_5	1		
ALU_3	1	DR_6	1		
ALU_4	0	I/O	0		

Transmitted

6E 53 0D

Transmitted textbox

Send button

Instruction

Figure 19 Sending a CPU Instruction

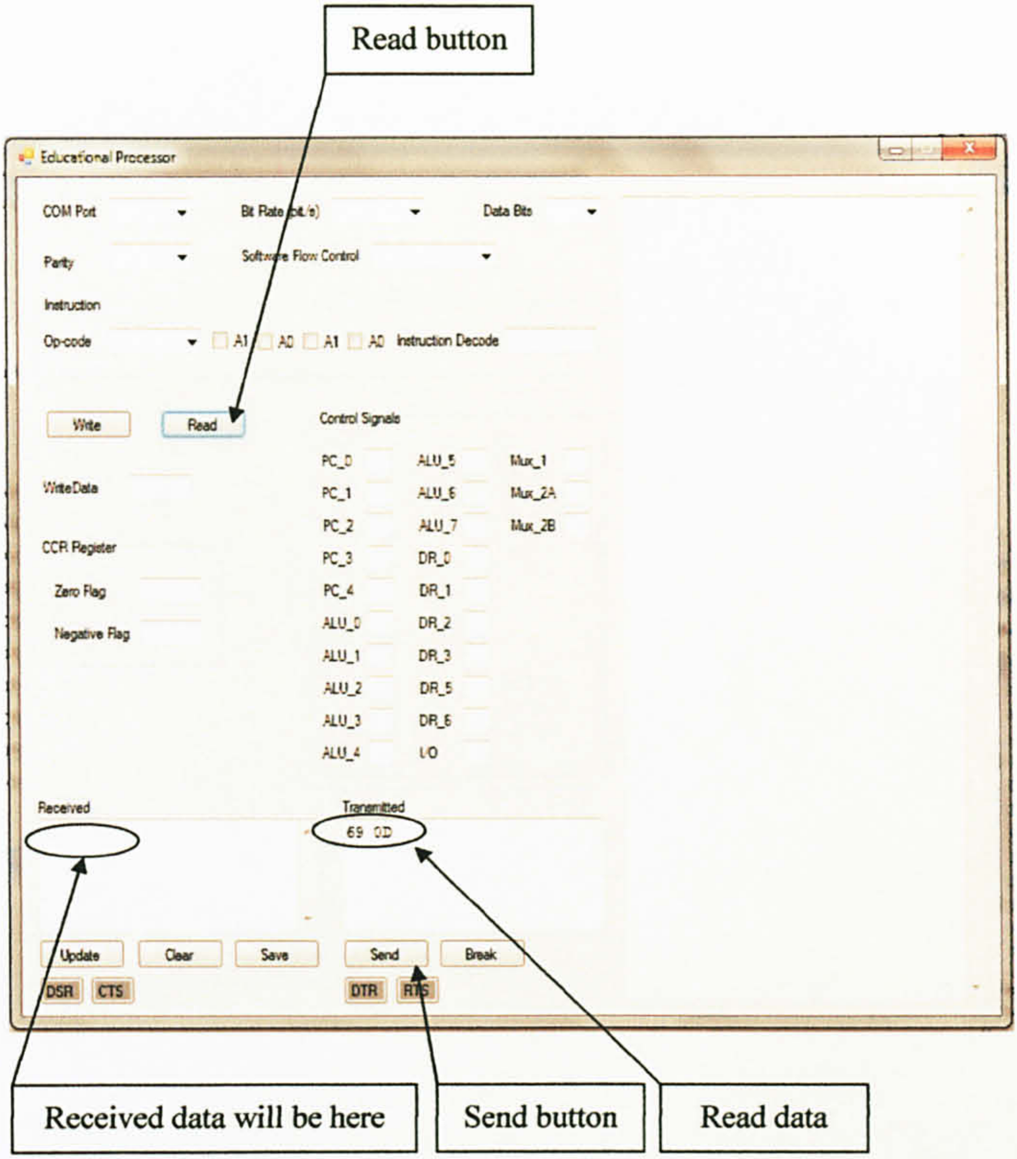
5. Now the hardware is received the instruction and is ready to store the instruction into instruction RAM. Since this CPU architecture is following Harvard architecture, the instruction and data is stored into separate memory. To write the received instruction, RD/WR switch is turned to the WR state. Now the first instruction, 01010011 is already stored in memory location 0001 of instruction RAM.

6. For the next instructions, the same step 2 to step 6 are repeated where the next instructions are stored into memory location 0010, 0011 and so on.

7. After all instructions of the test program are written, the CPU is ready to run all the instructions. To execute the all instructions, clock signal will be supplied in single step operation manually from the hardware itself. Figure below shows the clock switch. To generate to first clock signal, the clock switch is turned to HIGH state. Once the clock signal is changing its state for the first time, the program counter will start fetching the first instruction that was stored in memory location 0001. The CPU will execute the first instruction when the second clock signals is positive edged and this step will continue for the next instructions. When the CPU executes an instruction, appropriate control signal is generated by the microcontroller which is acting as ROM. The microcontroller is programmed to decode each op-code to its appropriate control signal. The control signals will control all the CPU functional units to what action they should take to execute an instruction.

8. After the CPU finished executing all the instructions, the hardware result can be observed through graphic user interface. Now, the expected result and the actual hardware result are validated by comparing the content of register files, ALU outputs and the flags. To read the output of the hardware, "Read" button is clicked to request the data from the hardware. The microcontroller again is used the collect the required data which are data sum as well as the flags from the condition code register.

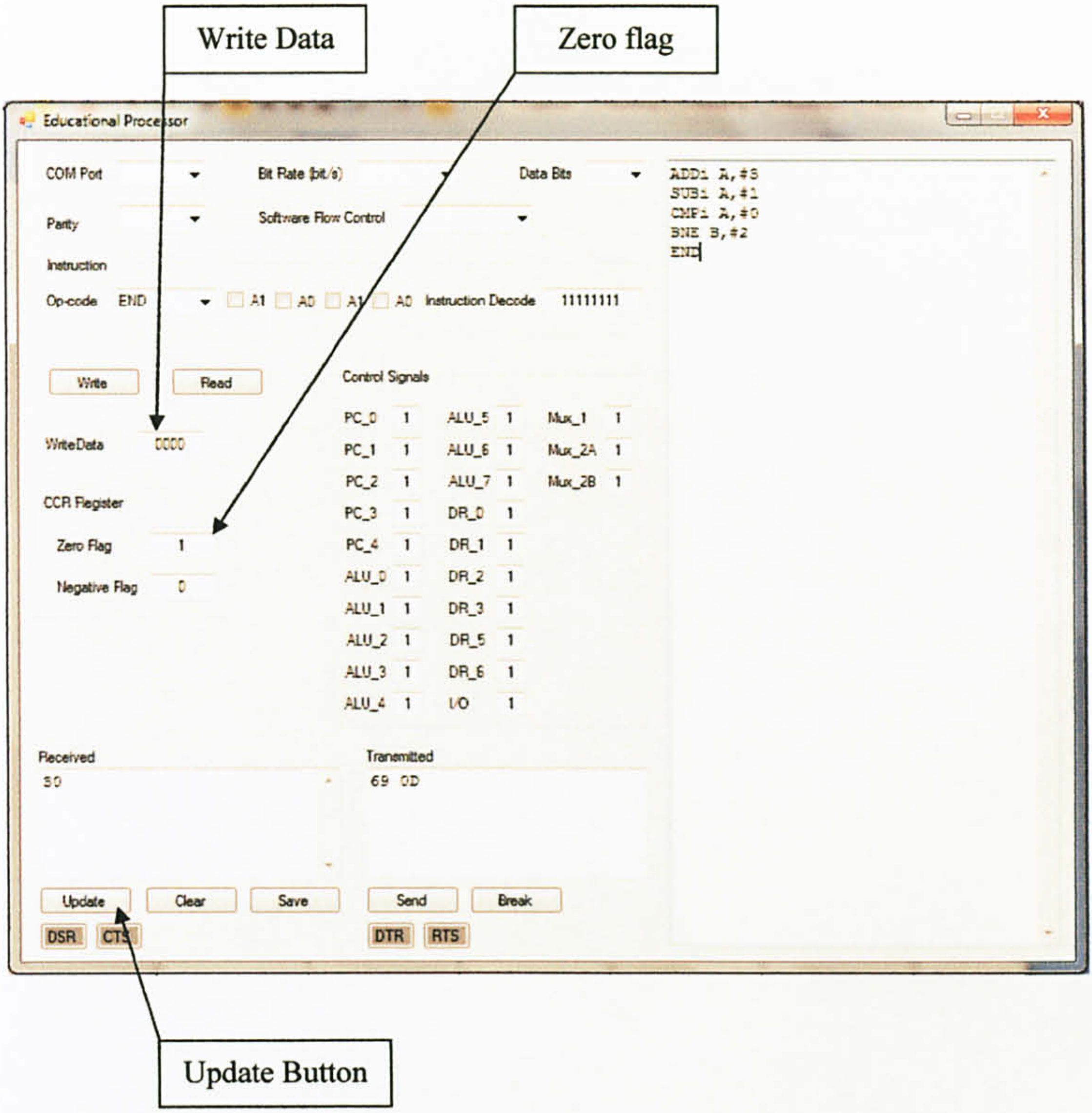
9. As shown the figure below, after the "Read" button is clicked, then "Send" button is clicked for the GUI to transmit 69 0D to microcontroller to request the output data from the hardware. Character "69" (char "i" in ASCII) will ask the microcontroller the read input and send to the GUI while "0D" is referred as carriage return.



Transmitted: 69 0D

Figure 20 Requesting hardware result

10. The received 8-bit data from microcontroller to the GUI is located in the "Received" textbox. From here, the received data is analyzed to the respective fields which are data sum and flags. The other fields of the GUI are analyzed automatically by the GUI itself. The final output is shown below.



COM Port
Bit Rate (bit/s)
Data Bits
Parity
Software Flow Control
Instruction
Op-code END □ A1 □ A0 □ A1 □ A0 Instruction Decode 11111111

WriteData 0000

CCR Register
Zero Flag 1
Negative Flag 0

Control Signals

PC_0	1	ALU_5	1	Mux_1	1
PC_1	1	ALU_6	1	Mux_2A	1
PC_2	1	ALU_7	1	Mux_2B	1
PC_3	1	DR_0	1		
PC_4	1	DR_1	1		
ALU_0	1	DR_2	1		
ALU_1	1	DR_3	1		
ALU_2	1	DR_5	1		
ALU_3	1	DR_6	1		
ALU_4	1	I/O	1		

Received
30

Transmitted
69 0D

```
ADDi A,#3
SUBi A,#1
CMPi A,#0
BNE B,#2
END
```

Figure 21 First test program final result

For the above first test program, the validation result is shown in the table below.

Table 7 First test program validation result

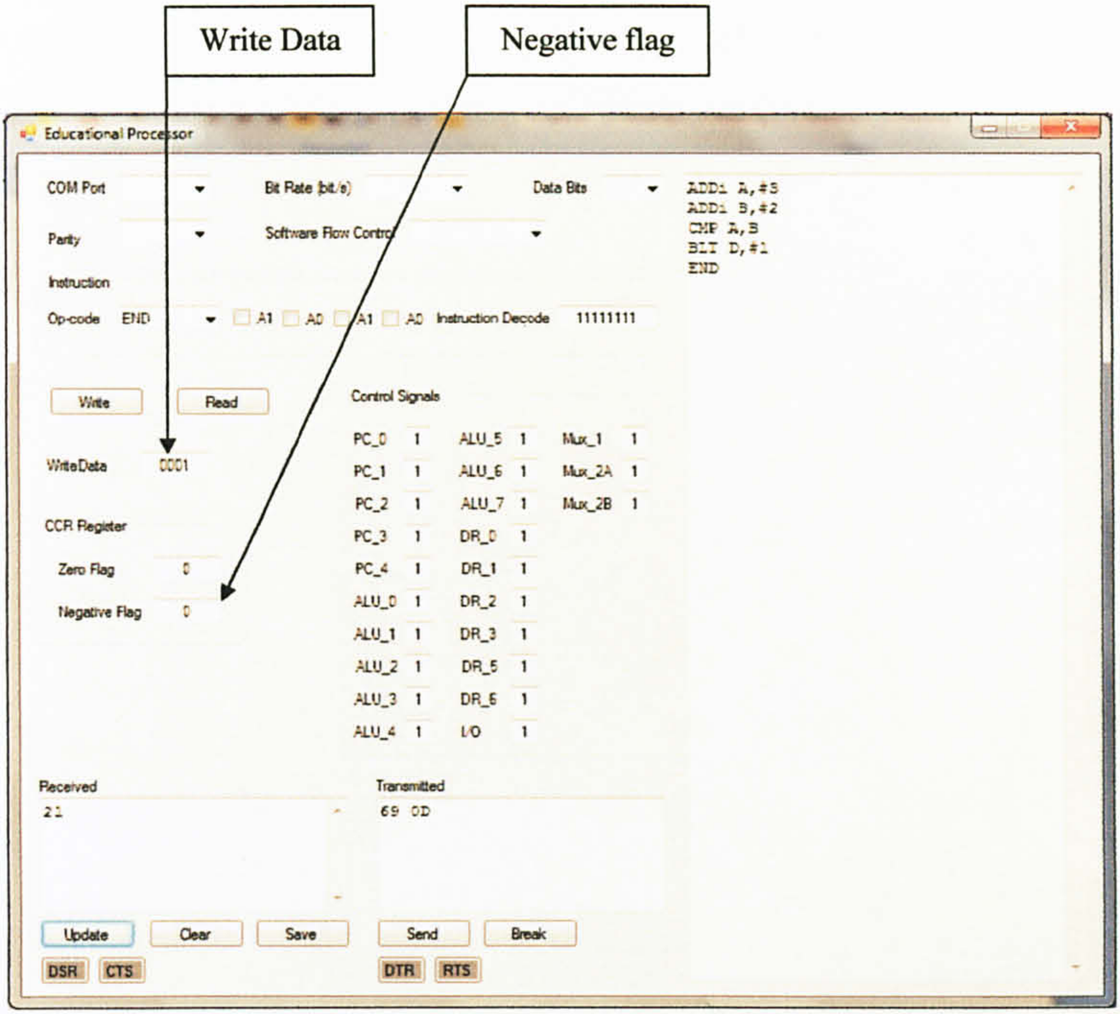
Address	Instruction	PCcount	Instruction Decode	RegA	RegB	Data Sum	Zero Flag
0000		0000	00000000	0000	0000	0000	0
0001	ADDi A,#3	0001	01010011	0000	0000	0011	0

0010	SUBi A,#1	0010	01110001	0011	0000	0011	0
0011	CMPi A,#0	0011	10010000	0010	0000	0010	0
0100	BNE B,#2	0100	10100110	0010	0010	0010	0
0010	SUBi A,#1	0010	01110001	0010	0000	0010	0
0011	CMPi A,#0	0011	10010000	0001	0000	0001	0
0100	BNE B,#2	0100	10100110	0010	0010	0010	0
0010	SUBi A,#1	0010	01110001	0001	0000	0001	0
0011	CMPi A,#0	0011	10010000	0000	0000	0000	1
0100	BNE B,#2	0100	10100110	0010	0010	0010	1
0101	END	0101	11111111	0000	0000	0000	1

Second test program is also executed to test the BLT instruction where the flow of program will be changed if A is less than B. Here, for CMP instruction, the ALU is always doing subtraction operation to check the negative flag by BLT instruction and there is no write back operation of ALU result for CMP operation.

Table 8 Second test program

Address	Instruction	Op-code	Operand1	Operand2
0001	ADDi A,#3	0101	00	10
0010	ADDi B,#2	0101	01	11
0011	CMP A,B	1000	00	01
0100	BLT D,#1	1011	11	01
0101	END	1111		



```
ADDi A,#3
ADDi B,#2
CMP A,B
BLT D,#1
END
```

Instruction

Op-code END □ A1 □ A0 □ A1 □ A0 Instruction Decode 11111111

WriteData 0001

CCR Register

Zero Flag 0

Negative Flag 0

Control Signals

PC_0	1	ALU_5	1	Mux_1	1
PC_1	1	ALU_6	1	Mux_2A	1
PC_2	1	ALU_7	1	Mux_2B	1
PC_3	1	DR_0	1		
PC_4	1	DR_1	1		
ALU_0	1	DR_2	1		
ALU_1	1	DR_3	1		
ALU_2	1	DR_5	1		
ALU_3	1	DR_6	1		
ALU_4	1	I/O	1		

Received

```
21
```

Transmitted

```
69 0D
```

Figure 22 Second test program final result

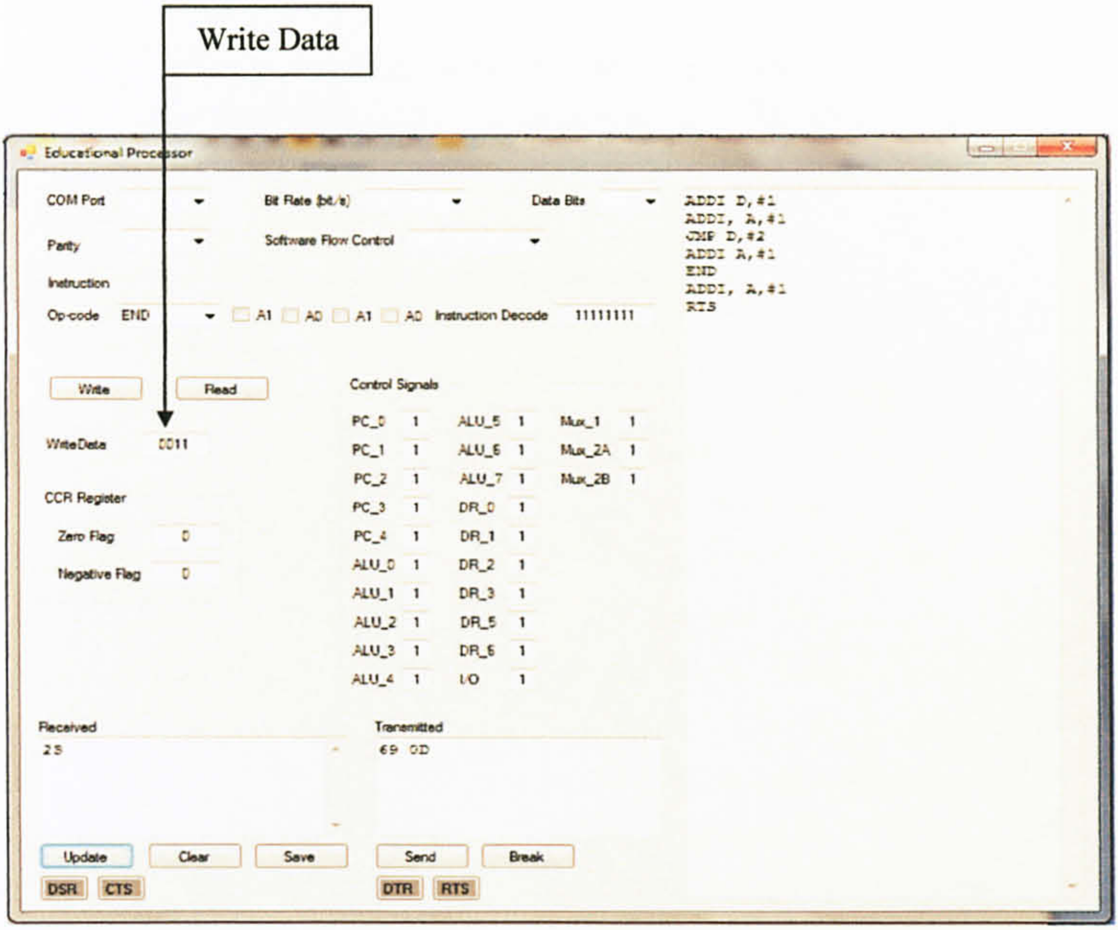
Table 9 Second test program validation result

Address	Instruction	PCcount	Instruction Decode	RegA	RegB	Data Sum	N Flag
0000		0000	00000000	0000	0000	0000	0
0001	ADDi A,#3	0001	01010011	0000	0000	0011	0
0010	ADDi B,#2	0010	01010110	0011	0000	0010	0
0011	CMP A,B	0011	10000001	0011	0010	0001	0
0100	BLT D,#1	0100	10111101	0011	0010	0001	0
0101	END	0101	1111111	0011	0010	0001	0

For the third test program, JMP instruction will be tested. It calls the subroutine and return back to the main after finish executing the subroutine instructions.

Table 10 Third test program

Address	Instruction	Op-code	Operand1	Operand2
0000		0000	00	00
0001	ADDI D,#1	0101	11	01
0010	ADDI, A,#1	0101	00	01
0011	JMP D,#2	1100	11	10
0100	ADDI A,#1	0101	00	01
0101	END	1111		
0110	ADDI, A,#1	0101	00	01
0111	RTS	1101		



Op-code END □ A1 □ A0 □ A1 □ A0 Instruction Decode 11111111

WriteData 0011

CCR Register

Zero Flag 0

Negative Flag 0

Control Signals

PC_0	1	ALU_5	1	Mux_1	1
PC_1	1	ALU_6	1	Mux_2A	1
PC_2	1	ALU_7	1	Mux_2B	1
PC_3	1	DR_0	1		
PC_4	1	DR_1	1		
ALU_0	1	DR_2	1		
ALU_1	1	DR_3	1		
ALU_2	1	DR_5	1		
ALU_3	1	DR_6	1		
ALU_4	1	I/O	1		

Received

```
25
```

Transmitted

```
69 0D
```

```
ADDI D,#1
ADDI, A,#1
JMP D,#2
ADDI A,#1
END
ADDI, A,#1
RTS
```

Figure 23 Third test program final result

Table 11 Third test program validation result

Address	Instruction	PCcount	Instruction Decode	RegA	RegD	Data Sum
0000		0000	00000000	0000	0000	0000
0001	ADDI D,#1	0001	01011101	0000	0000	0001
0010	ADDI, A,#1	0010	01010001	0000	0001	0001
0011	JMP D,#2	0011	11001110	0001	0001	0011
0100	ADDI A,#1	0100	01010001	0010	0001	0011
0101	END	0101	11111111	0011	0001	0011
0110	ADDI, A,#1	0110	01010001	0001	0001	0010
0111	RTS	0111	1101			

For the fourth test program, load and store instruction will be tested by writing data into the data RAM and load it back for the next instruction. The pseudo code of this test program is as below:

Table 12 Fourth program pseudo code

Instruction	Pseudo code	Description

ADDi A,#3	A = 3
	PC = PC + 1
		Add immediate value 3 to register A.
		PC increments by 1.

ADDi B,#3	B = 3
	PC = PC + 1
		Add immediate value 3 to register B.
		PC increments by 1.

ADD A,B	A = A + B
	PC = PC + 1
		Add content of register B to the content of register A; write back the ALU result into register A.
		PC increments by 1.

ADDi C,#1	C = 1
	PC = PC + 1
		Add immediate value 1 to register C. Preparing the address to for STR.
		PC increments by 1.

STR A,C	A --> Mem[C] PC = PC + 1	Store content of register A to memory location 0001. PC increments by 1.

LDR D,C	D <-- Mem[C]
	PC = PC + 1
		Load data from memory location 0001 and write back the data into register D.
		PC increments by 1.

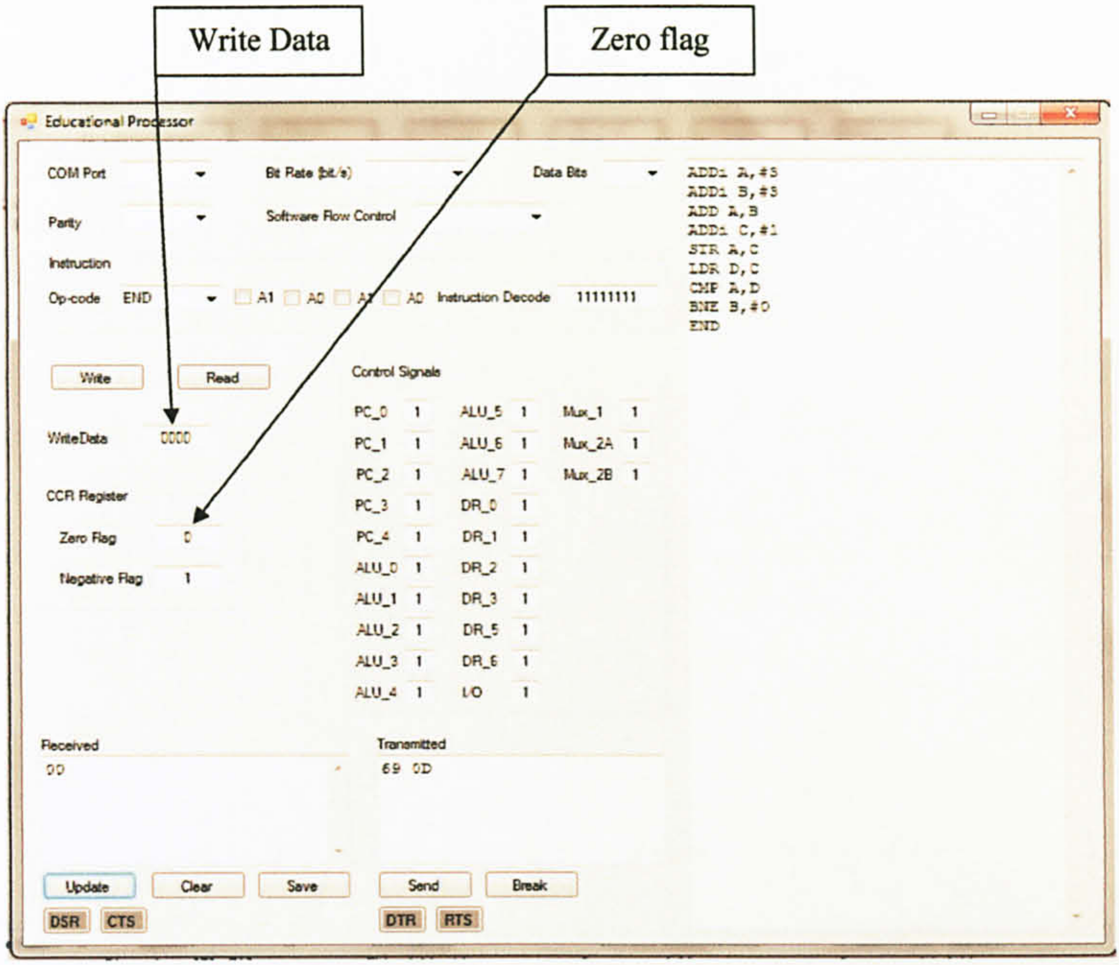
CMP A,D	A - D	Compare content of register A and the content of register D

BNE B,#0	If A – D = 0
	PC = PC + 1
	Else
	PC = B[]
		If A = D, PC increments by 1, else, if content of register A is not equal to the content of register D, PC = 3.

END		End of CPU execution.

Table 13 Fourth test program instruction decode

Address	Instruction	Op-code	Operand1	Operand2
0000		0000	00	00
0001	ADDi A,#3	0101	00	10
0010	ADDi B,#3	0101	01	11
0011	ADD A,B	0100	00	01
0100	ADDi C,#1	0101	11	01
0101	STR A,C	0001	00	10
0110	LDR D,C	0000	11	10
0111	CMP A,D	1000	00	11
1000	BLT B,#0	1011	01	00
1001	END	1111		



Op-code END ☐ A1 ☐ A0 ☐ A1 ☐ A0 Instruction Decode 11111111

```
ADDi A,#3
ADDi B,#3
ADD A,B
ADDi C,#1
STR A,C
LDR D,C
CMP A,D
BNE B,#0
END
```

WriteData 0000

CCR Register

Zero Flag 0

Negative Flag 1

Control Signals

PC_0	1	ALU_5	1	Mux_1	1
PC_1	1	ALU_6	1	Mux_2A	1
PC_2	1	ALU_7	1	Mux_2B	1
PC_3	1	DR_0	1		
PC_4	1	DR_1	1		
ALU_0	1	DR_2	1		
ALU_1	1	DR_3	1		
ALU_2	1	DR_5	1		
ALU_3	1	DR_6	1		
ALU_4	1	I/O	1		

Received

```
0D
```

Transmitted

```
69 0D
```

Figure 24 Fourth test program final result

Table 14 Fourth test program validation result

Addr	Instr.	PC Cnt.	Instruction Decode	Reg A	Reg B	Reg C	Reg D	Data Sum	N flag
0000		0000	00000000	0000	0000	0000	0000	0000	0
0001	ADDi A,#3	0001	01010011	0000	0000	0000	0000	0000	0
0010	ADDi B,#3	0010	01010100	0011	0000	0000	0000	0000	0
0011	ADD A,B	0011	01001101	0011	0011	0000	0000	0110	0
0100	ADDi C,#1	0100	01011001	0110	0011	0000	000	0001	0
0101	STR A,C	0101	00010010	0110	0011	0001	0000	0001	0
0110	LDR D,C	0110	00001110	0110	0011	0001	0000	0110	0

0111	CMP A,D	0111	10000011	0110	0011	0001	0110	0000	1
1000	BNE B,#0	1000	10110100	0110	0011	0001	0110	0000	1
1001	END	1001	11111111	0110	0011	0001	0110	0000	1

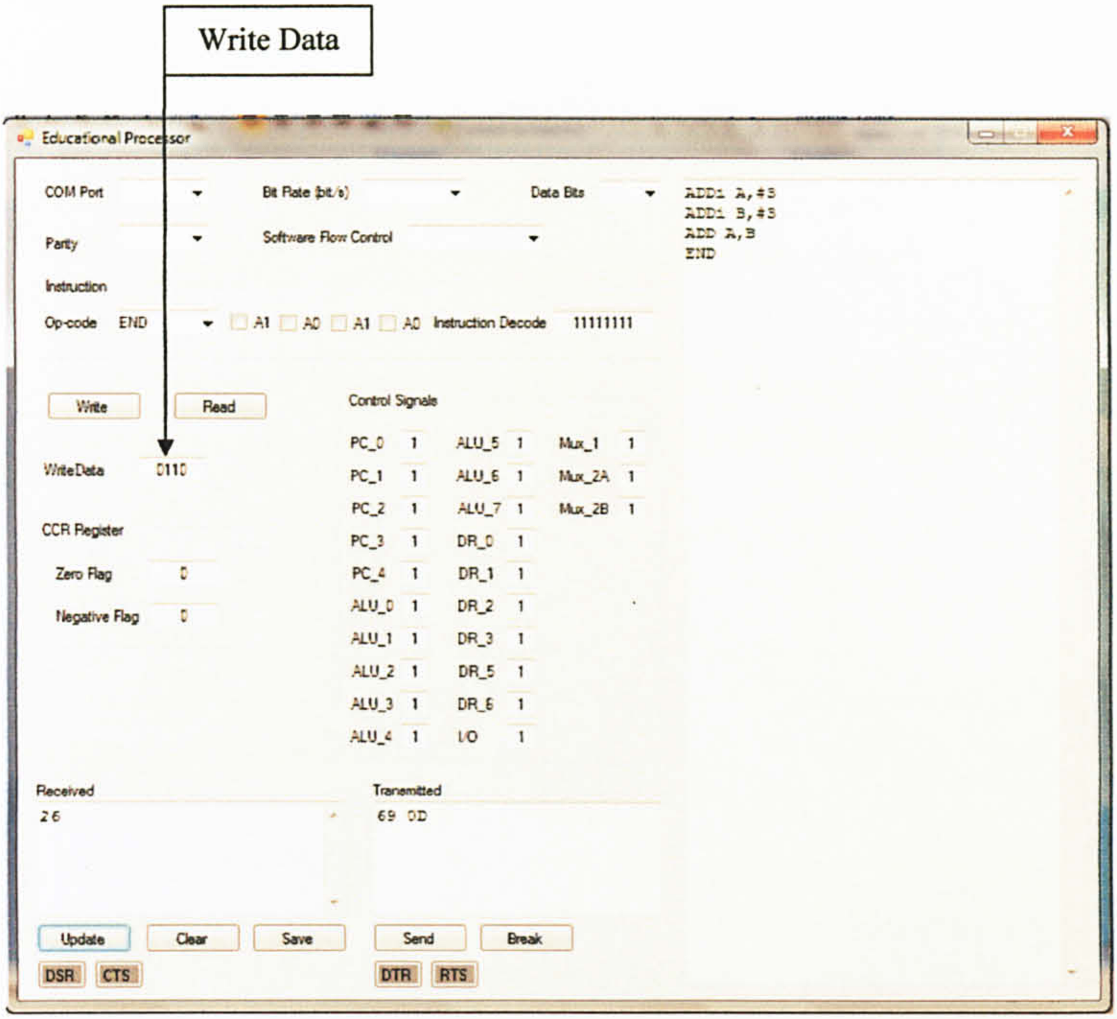
For the fifth test program, the arithmetic operation will be tested. The CPU will execute the addition operation to add two numbers from the registers.

Table 15 Fifth test program

Address	Instruction	Op-code	Operand1	Operand2
0000		0000	00	00
0001	ADDI A,#3	0101	00	11
0010	ADDI B,#3	0101	00	11
0011	ADD A,B	0100	00	01
0100	END	1111		

Table 16 Fifth test program validation result

PCcount	Address	Instruction	Instruction Decode	RegA	RegB	Write Data	Data Sum
0000	0000		00000000	0000	0000	0011	0000
0001	0001	ADDI A,#3	01010011	0011	0000	0011	0000
0010	0010	ADDI B,#3	01010111	0011	0011	0011	0000
0011	0011	ADD A,B	01000001	0011	0011	0011	0000
0100	0100	END	11111111	0110	0011	0110	0110



Write Data

```
ADD1 A,#3
ADD1 B,#3
ADD A,B
END
```

Instruction

Op-code END ☐ A1 ☐ A0 ☐ A1 ☐ A0 Instruction Decode 11111111

WriteData 0110

CCR Register

Zero Flag 0

Negative Flag 0

Control Signals

PC_0	1	ALU_5	1	Mux_1	1
PC_1	1	ALU_6	1	Mux_2A	1
PC_2	1	ALU_7	1	Mux_2B	1
PC_3	1	DR_0	1		
PC_4	1	DR_1	1		
ALU_0	1	DR_2	1		
ALU_1	1	DR_3	1		
ALU_2	1	DR_5	1		
ALU_3	1	DR_6	1		
ALU_4	1	I/O	1		

Received

```
26
```

Transmitted

```
69 0D
```

Figure 25 Fifth test program final result

CHAPTER 5
CONCLUSION AND RECOMMENDATION

This chapter discuss the conclusion arrive after completing this project and recommendation or the future work that can be done to improve the project.

5.1 Conclusion

As mentioned in the chapter one, the objective of this project is to provide a new learning environment for Computer System Architecture class for student to learn computer system architecture at gate level on how a CPU execute an instruction. The CPU functional unit is designed and fabricated part by part and then they are combined together to get a completed full run working CPU with interfacing to a Graphic User Interface via serial communication between computer and microcontroller.

5.2 Recommendation

For future work, there are a lot of improvements that can be done to improve this project. Such improvements include but not limited to:

- Full working CPU with capability of handling an Operating System. With a complete working CPU, there would be much more areas that this project can educate student when such areas like Operating System, assembler and compiler design are included.

REFERENCES

[1] Alan Clements, 2006 fourth edition, "Principle of Computer Hardware"

[2] Behrooz Pahrami, 2005, "Computer Architecture"

[3] Albert P. Malvino & Jerald A Brown, 3rd edition, "Digital Computer Electronics", "*SAP Processor*"

[4] David A. Patterson & John L. Hennessy, 3rd edition, "Computer organization and Design"

[5] 4bit CISC CPU constructed with TTL logics, http://www.galacticelectronics.com/4BitCPU_ALU.HTML

[6] MIPS Processor architecture, https://www.cs.tcd.ie/Jeremy.Jones/vivio/dlx/dlxtutorial.htm

APPENDICES

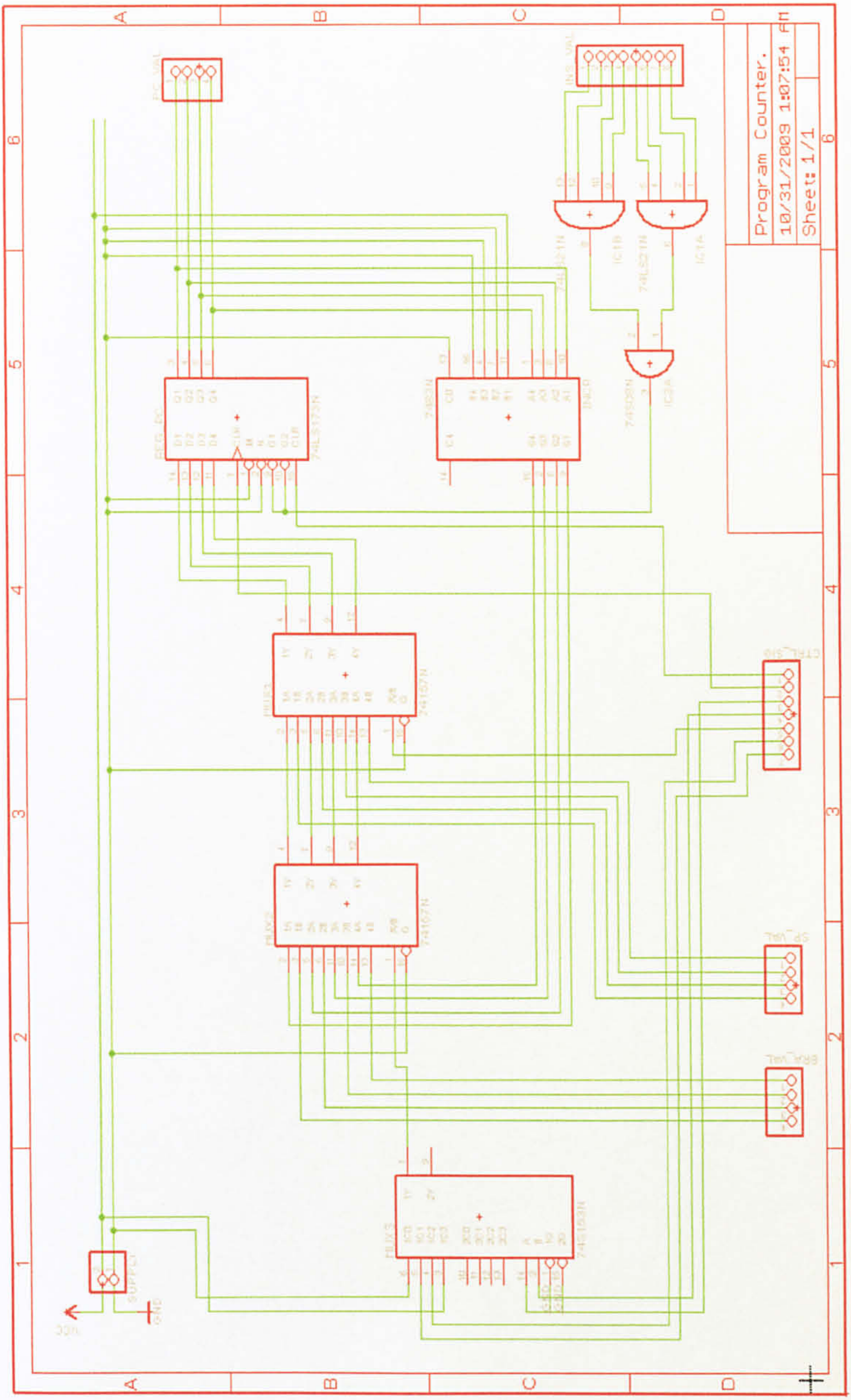
APPENDIX A

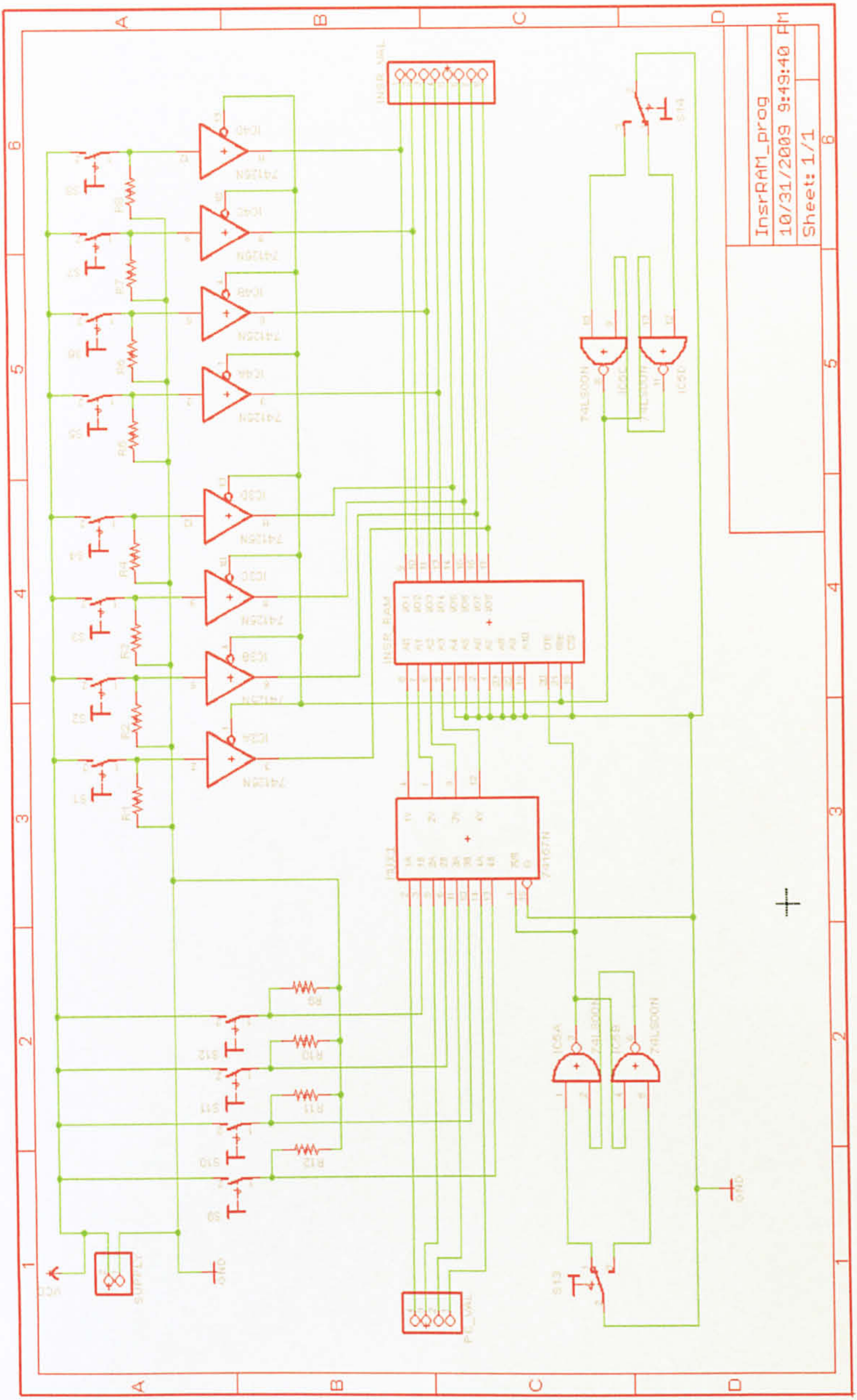
PROJECT GA NTT CHART

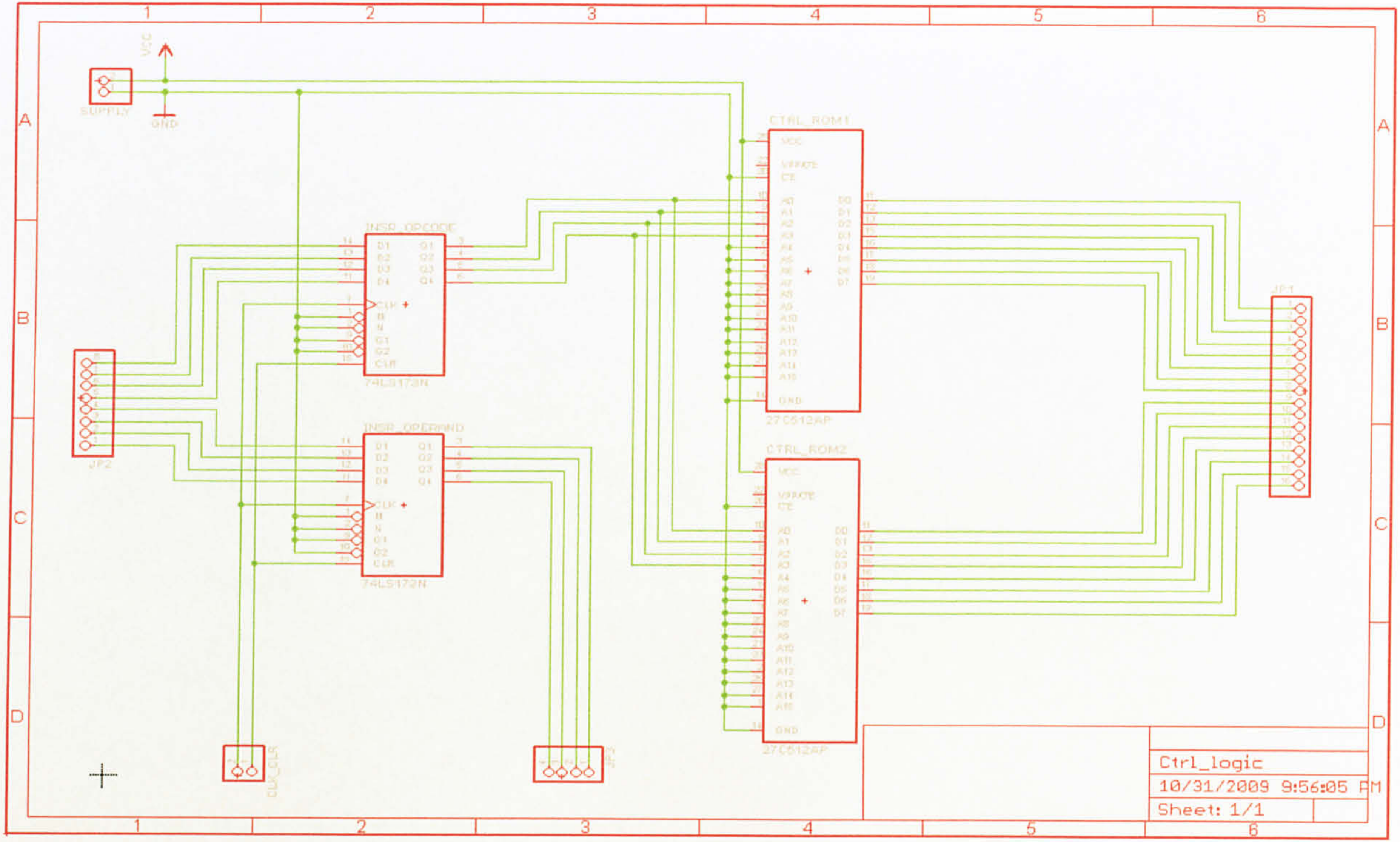
No.	Description	Week_1	Week_2	Week_3	Week_4	Week_5	Week_6	Week_7	Week_8	Week_9	Week_10	Week_11	Week_12	Week_13	Week_14
	Semester July 2009														
1	Selection of topic	🟩													
2	Preliminary Research Work		🟩	🟩											
3	Submission of Preliminary Report			🟩											
4	Instruction Set Design			🟩	🟩										
5	Instruction Set Design			🟩	🟩	🟩									
6	Datapath Design				🟩	🟩	🟩								
7	Datapath Design						🟩	🟩	🟩						
8	Datapath Design								🟩	🟩	🟩				
9	Circuit Schematics Design										🟩				
10	Circuit Schematics Design										🟩	🟩			
11	Circuit Schematics Design											🟩	🟩		
12	Circuit Schematics Design												🟩		
13	Simulation and Debugging												🟩	🟩	🟩
14	Submission of Interim Report and Oral Presentation														🟩
	Semester Jan 2010														
No.															
1	PCB Layout Design	🟩													
2	PCB Layout Design		🟩												
3	PCB Layout Design			🟩											
4	PCB Layout Design			🟩	🟩										
5	PCB Layout Design				🟩	🟩	🟩								
6	Assembling and Validation						🟩								
7	Assembling and Validation							🟩	🟩	🟩					
8	Assembling and Validation								🟩	🟩					
9	Assembling and Validation									🟩					
10	Assembling and Validation										🟩				
11	Assembling and Validation										🟩	🟩	🟩		
12	Assembling and Validation												🟩		
13	Interfacing													🟩	🟩
14	Interfacing													🟩	🟩

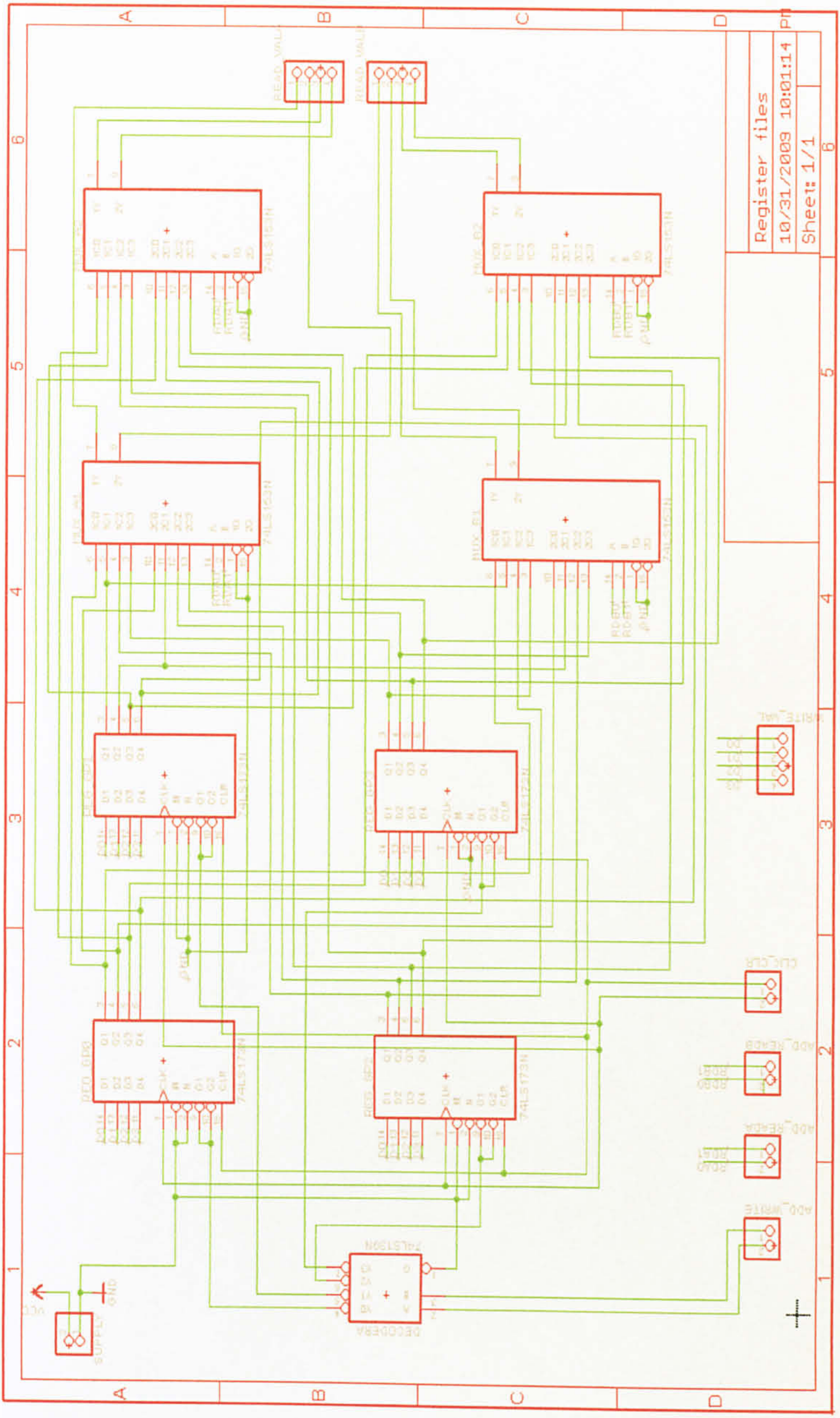
APPENDIX B

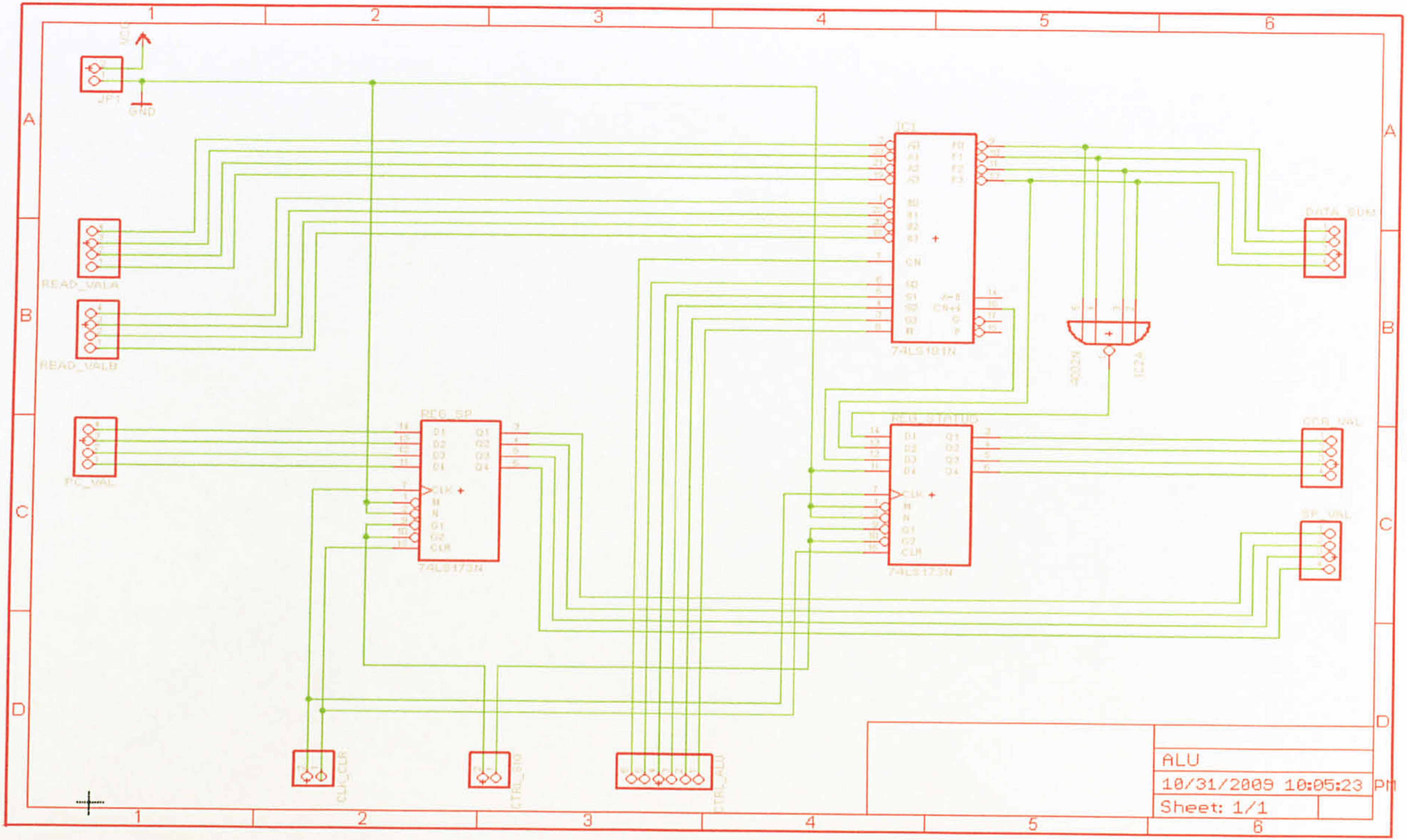
CIRCUIT SCHEMATICS

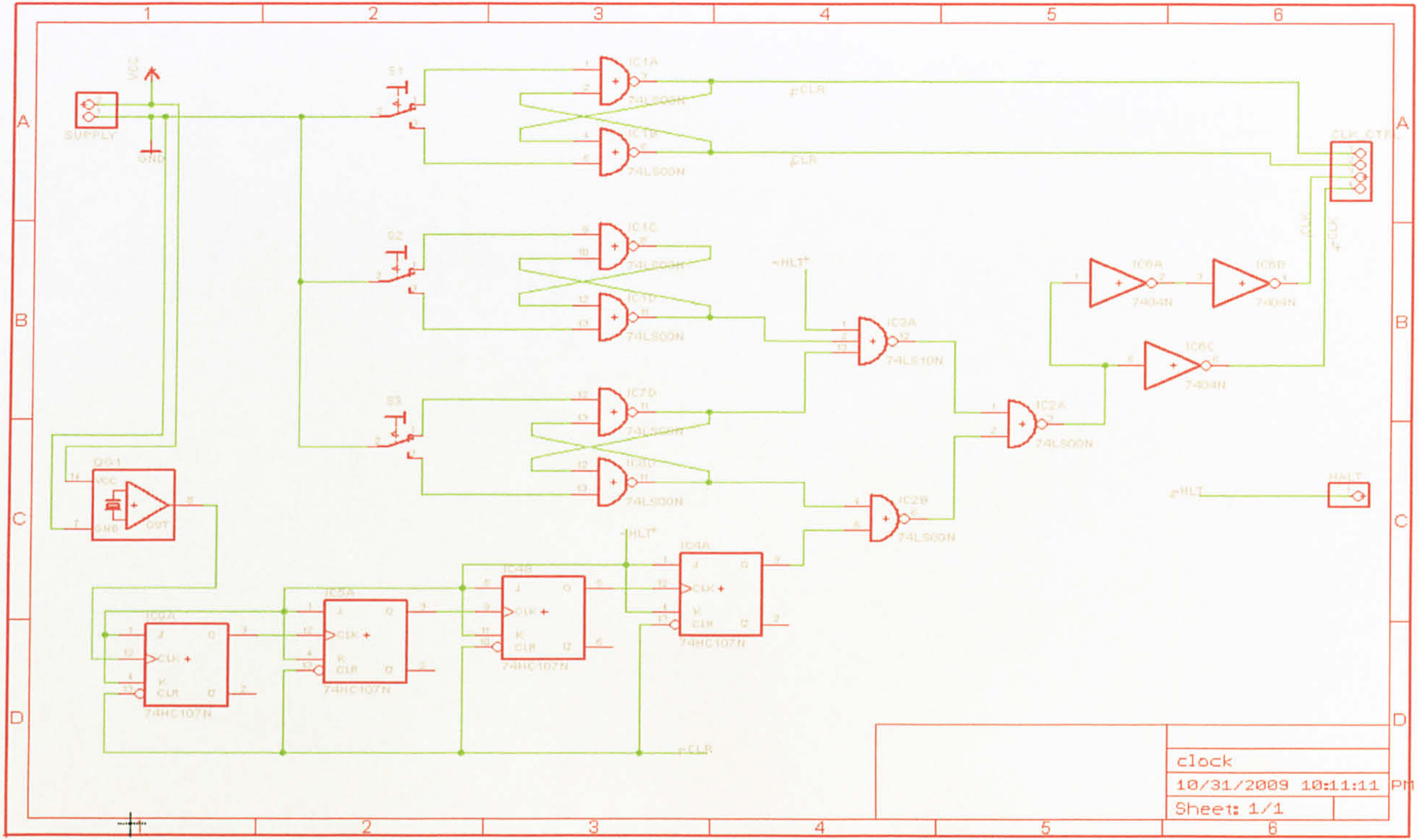

Program Counter.
10/31/2009 1:07:54 AM
Sheet: 1/1
PC_VAL
INS_VAL
REG_PC
74LS173N
7483N
INLP
74LS21N
IC1B
IC1A
7432N
IC2A
MUX1
74157N
MUX2
74157N
MUX3
74S153N
CTRL_SIG
SP_VAL
BRA_VAL
SUPPLY
VCC
GND
A
B
C
D
1
2
3
4
5
6

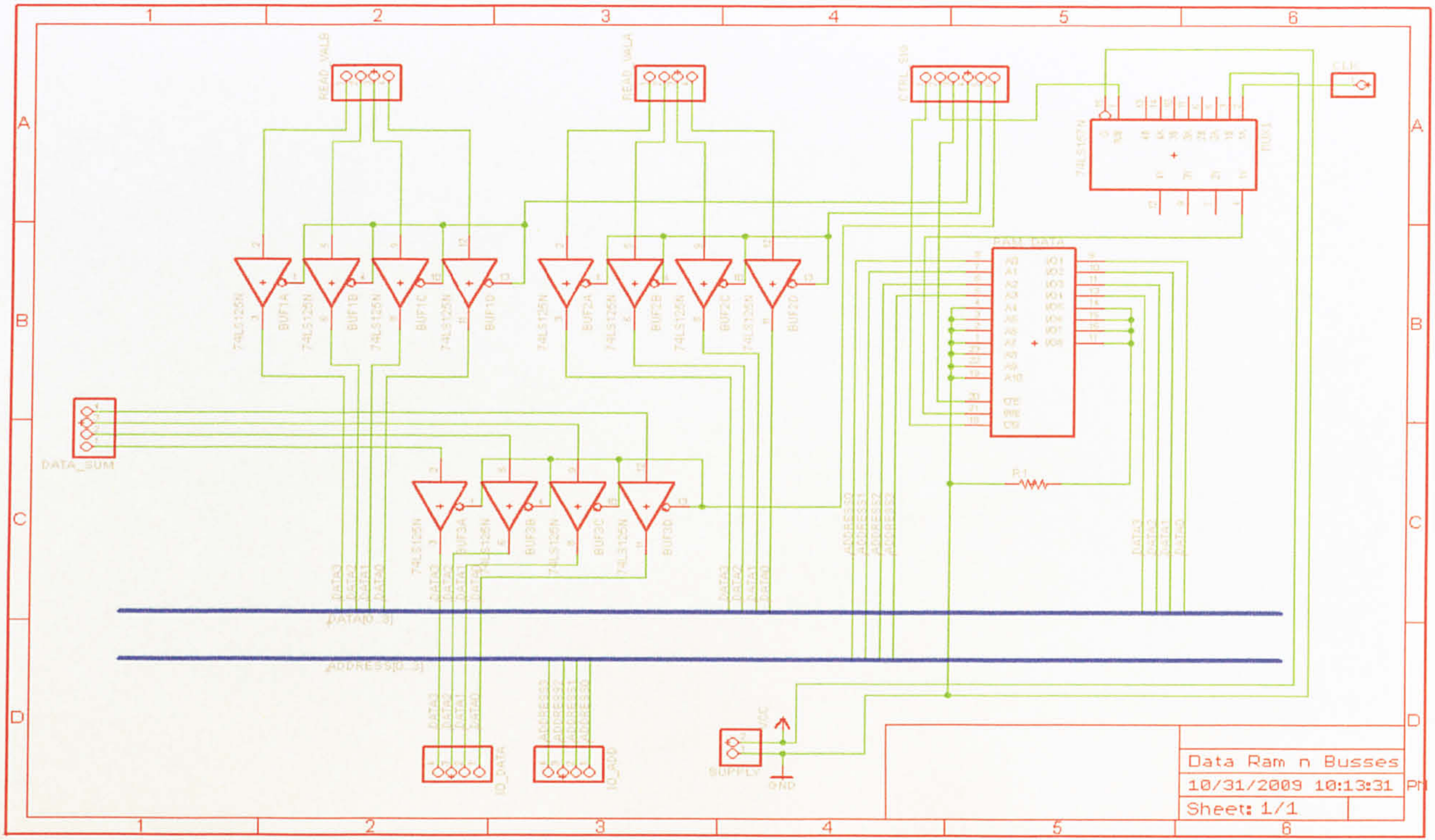

InsrRAM_prog
10/31/2009 9:49:40 AM
Sheet: 1/1
INSR_VAL
INSR_RAM
MUX1
74157N
PC_VAL
SUPPLY
VCC
GND
74125N
IC1A
IC1B
IC1C
IC1D
IC4A
IC4B
IC4C
IC4D
74LS00N
IC5A
IC5B
IC5C
IC5D
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
S13
S14
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
A
B
C
D
1
2
3
4
5
6

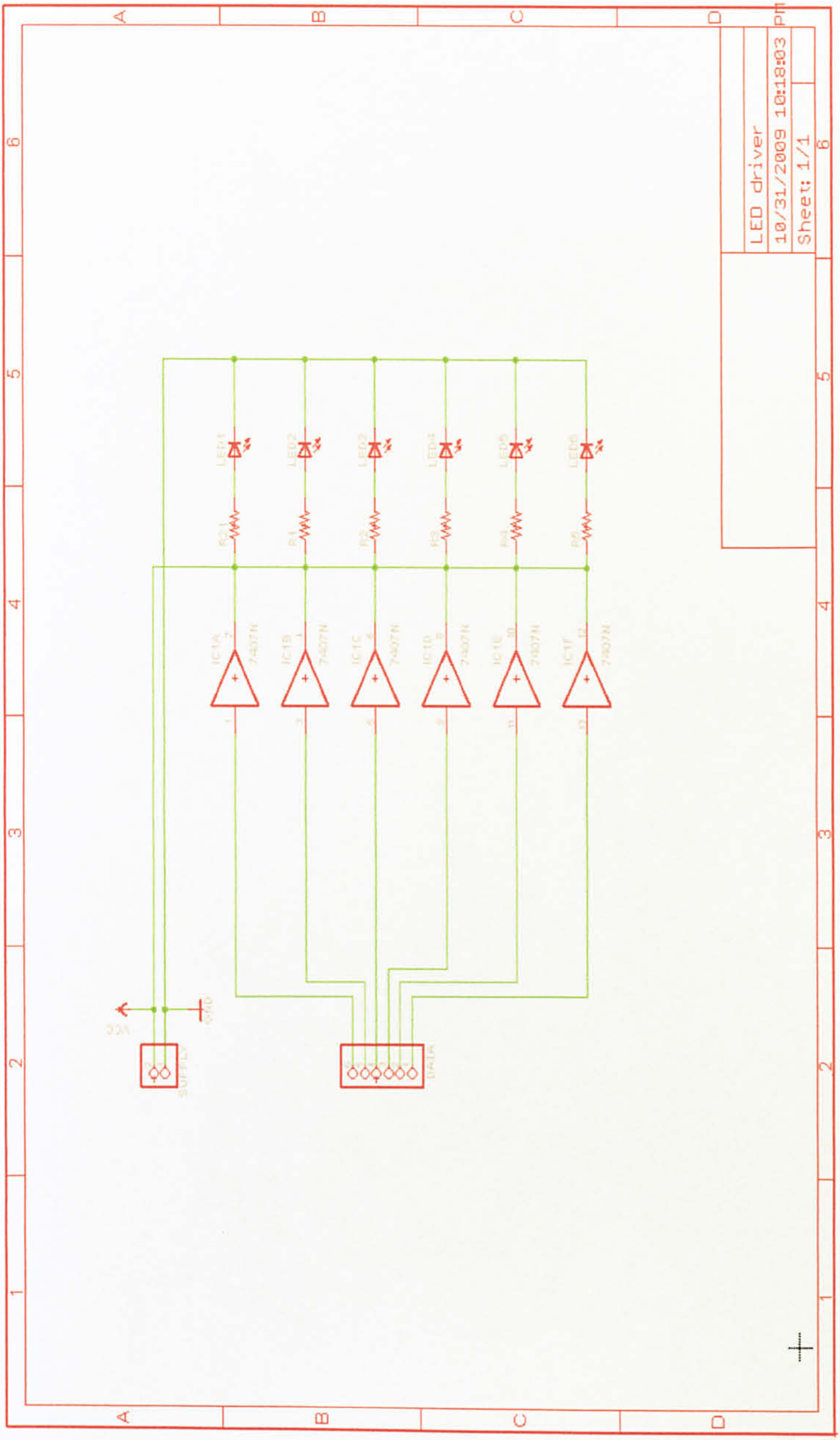

VCC
GND
SUPPLY
INSR_OPCODE
D1
D2
D3
D4
Q1
Q2
Q3
Q4
CLK
M
N
G1
G2
CLR
74LS173N
INSR_OPERAND
74LS173N
JP2
CLK_CLR
JP3
CTRL_ROM1
VCC
VPP/OE
CE
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
A15
GND
D0
D1
D2
D3
D4
D5
D6
D7
27C512AP
CTRL_ROM2
27C512AP
JP1
Ctrl_logic
10/31/2009 9:56:05 PM
Sheet: 1/1

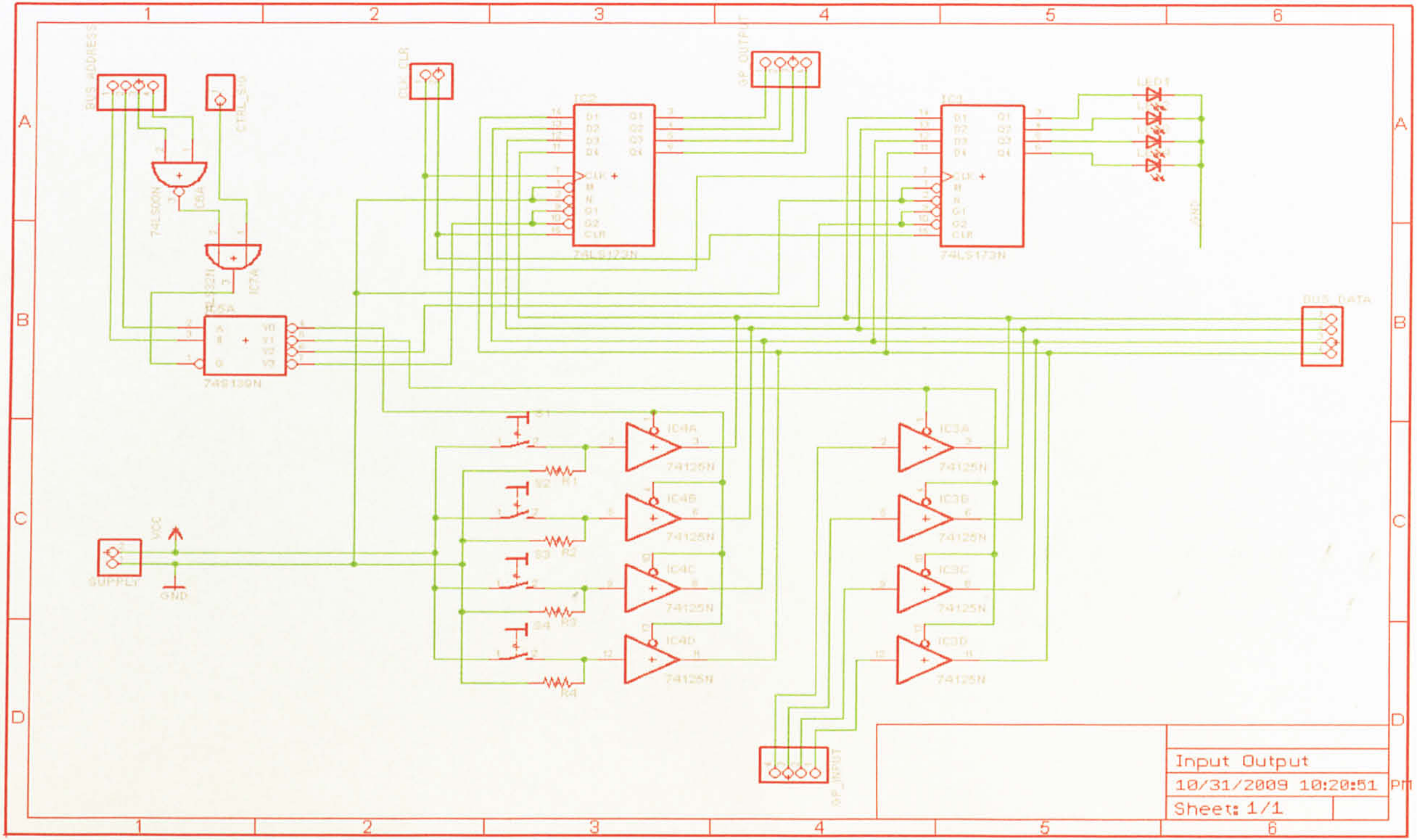

Register files
10/31/2003 10:01:14 pm
Sheet: 1/1
74LS153N
74LS173N
74LS139N
DECODERA
SUPPLY
VCC
GND
WRITE_VAL
CLK_CLR
ADD_READB
ADD_READA
ADD_WRITE
READ_VALA
READ_VALB
A
B
C
D
1
2
3
4
5
6


VCC
JP1
GND
READ_VALA
READ_VALB
PC_VAL
IC1
A0
A1
A2
A3
B0
B1
B2
B3
CN
S0
S1
S2
S3
M
F0
F1
F2
F3
A=B
CN+4
G
P
74LS181N
IC2A
4002N
DATA_SUM
REG_SP
D1
D2
D3
D4
CLK
M
N
G1
G2
CLR
Q1
Q2
Q3
Q4
74LS173N
REG_STATUS
74LS173N
CTR_VAL
SP_VAL
CLK_CLR
CTRL_SIG
CTRL_ALU
ALU
10/31/2009 10:05:23 PM
Sheet: 1/1

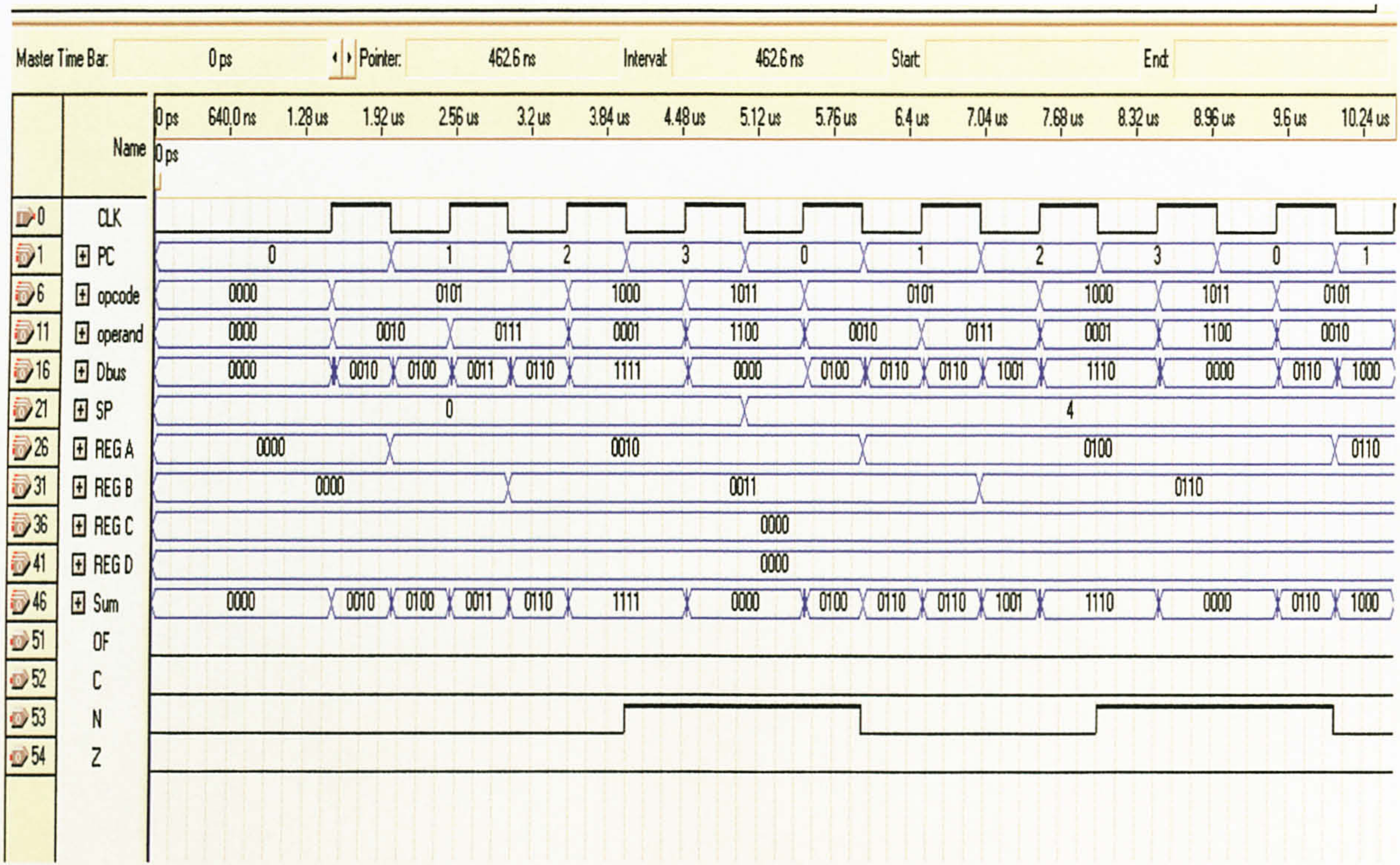

SUPPLY
VCC
GND
S1
S2
S3
IC1A
IC1B
IC1C
IC1D
IC7D
IC8D
74LS00N
/CLR
CLR
CLK CTRL
CLK
/CLK
IC6A
IC6B
IC6C
7404N
IC3A
74LS10N
IC2A
IC2B
/HLT
HLT
HALT
QG1
VCC
GND
OUT
IC9A
IC5A
IC4B
IC4A
J
Q
CLK
K
CLR
74HC107N
/CLR
clock
10/31/2009 10:11:11 PM
Sheet: 1/1


READ_VALB
READ_VALA
CTRL_SIG
CLK
74LS157N
MUX1
RAM_DATA
R1
BUF1A
BUF1B
BUF1C
BUF1D
BUF2A
BUF2B
BUF2C
BUF2D
BUF3A
BUF3B
BUF3C
BUF3D
74LS125N
DATA_SUM
DATA0
DATA1
DATA2
DATA3
ADDRESS0
ADDRESS1
ADDRESS2
ADDRESS3
DATA[0..3]
ADDRESS[0..3]
IO_DATA
IO_ADD
SUPPLY
VCC
GND
Data Ram n Busses
10/31/2009 10:13:31 PM
Sheet: 1/1


LED driver
10/31/2009 10:18:03 PM
Sheet: 1/1
A
B
C
D
1
2
3
4
5
6
LED1
LED2
LED3
LED4
LED5
LED6
R21
R1
R2
R3
R4
R5
IC1A
IC1B
IC1C
IC1D
IC1E
IC1F
7407N
VCC
GND
SUPPLY
DATA


BUS_ADDRESS
CTRL_SW
CLK_CLR
GP_OUTPUT
IC2
74LS173N
IC1
74LS173N
LED1
GND
74LS00N
IC6A
74LS32N
IC7A
IC5A
74S139N
BUS_DATA
S1
S2
S3
S4
R1
R2
R3
R4
IC4A
IC4B
IC4C
IC4D
IC3A
IC3B
IC3C
IC3D
74125N
VCC
SUPPLY
GP_INPUT
Input Output
10/31/2009 10:20:51 PM
Sheet: 1/1

APPENDIX C
SIMULATION RESULTS

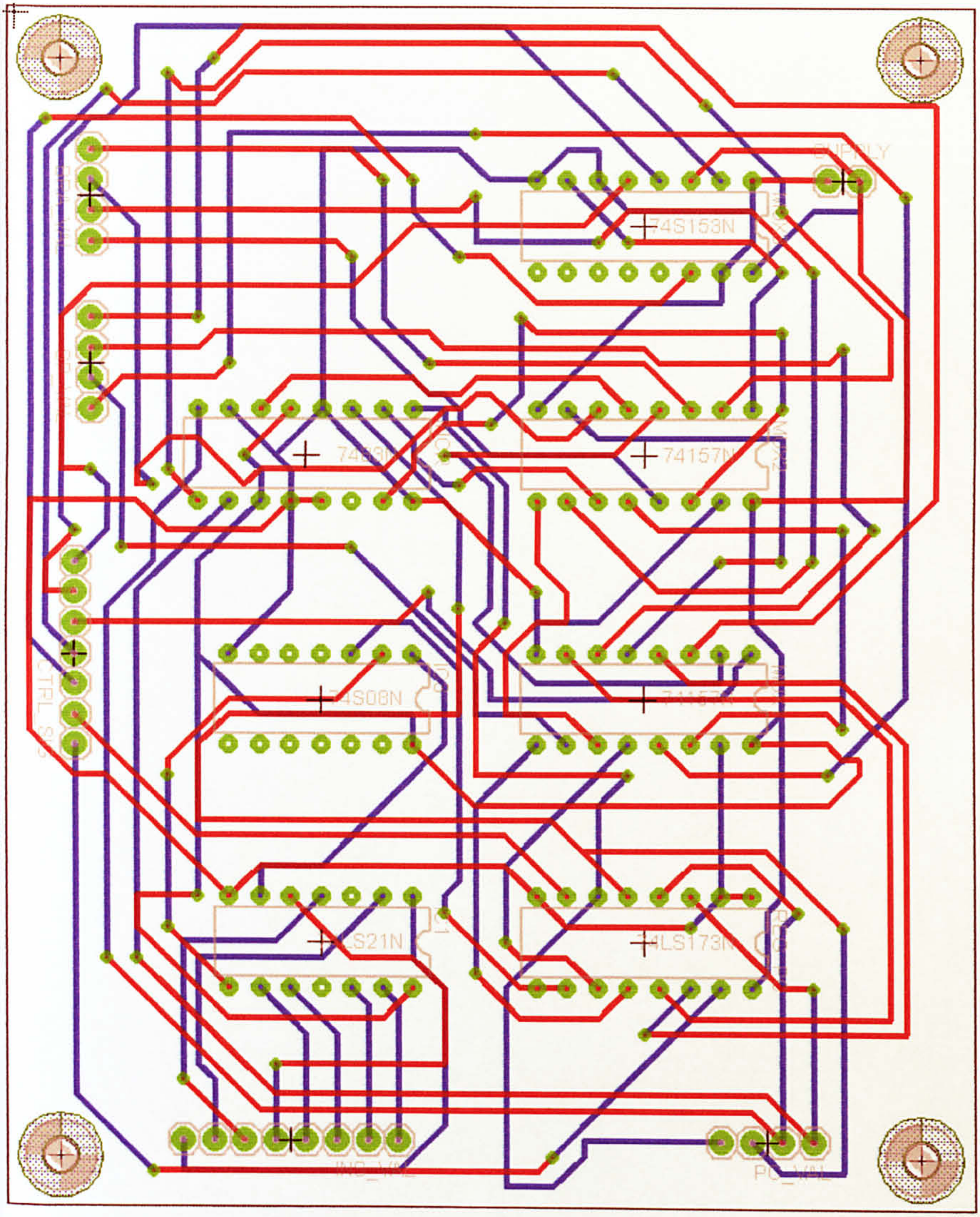

Master Time Bar: 0 ps
Pointer: 462.6 ns
Interval: 462.6 ns
Start:
End:
Name
0 ps 640.0 ns 1.28 us 1.92 us 2.56 us 3.2 us 3.84 us 4.48 us 5.12 us 5.76 us 6.4 us 7.04 us 7.68 us 8.32 us 8.96 us 9.6 us 10.24 us
0 ps
0 CLK
1 PC 0 1 2 3 0 1 2 3 0 1
6 opcode 0000 0101 1000 1011 0101 1000 1011 0101
11 operand 0000 0010 0111 0001 1100 0010 0111 0001 1100 0010
16 Dbus 0000 0010 0100 0011 0110 1111 0000 0100 0110 0110 1001 1110 0000 0110 1000
21 SP 0 4
26 REG A 0000 0010 0100 0110
31 REG B 0000 0011 0110
36 REG C 0000
41 REG D 0000
46 Sum 0000 0010 0100 0011 0110 1111 0000 0100 0110 0110 1001 1110 0000 0110 1000
51 OF
52 C
53 N
54 Z

Second Program

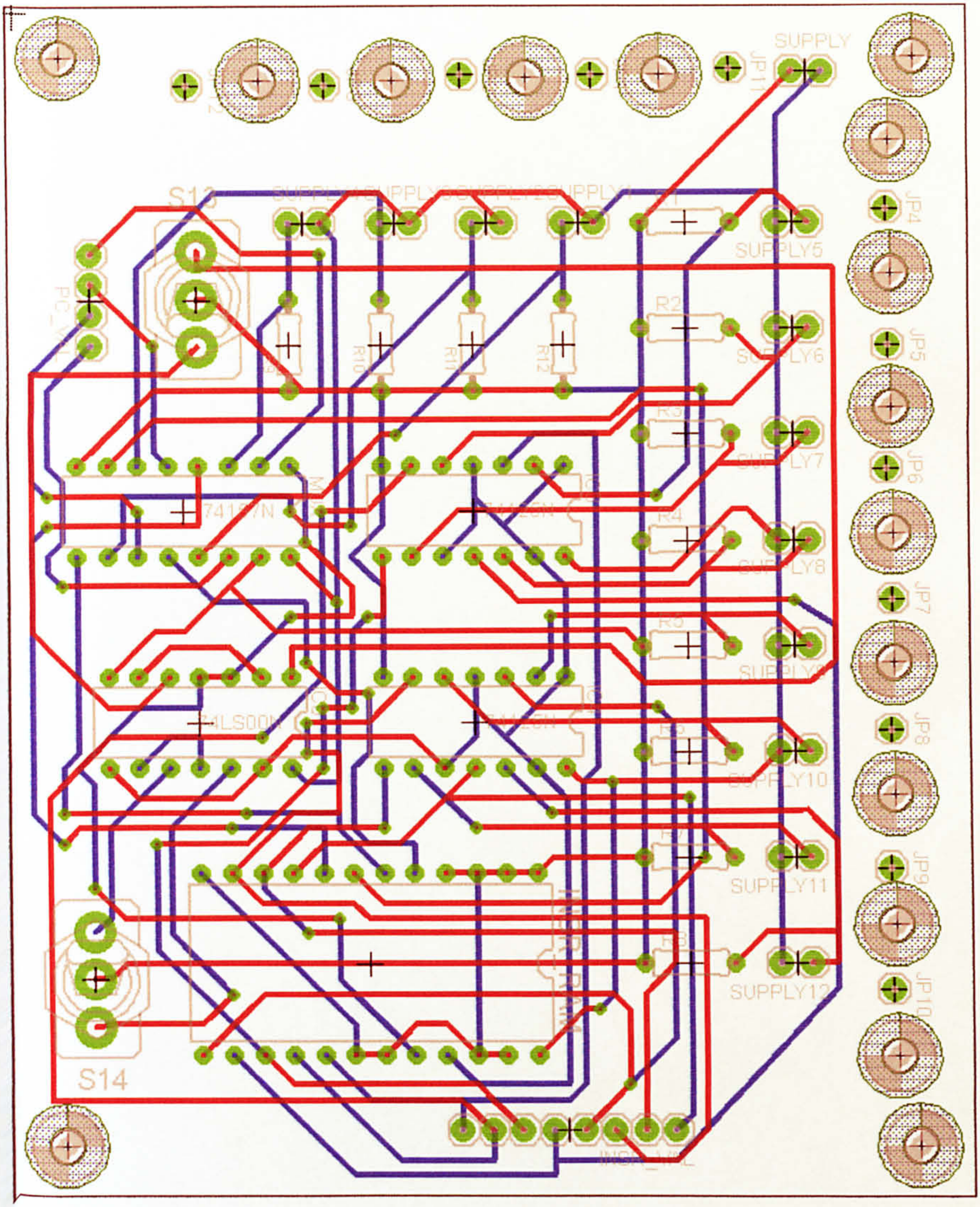
APPENDIX D

PCB LAYOUTS

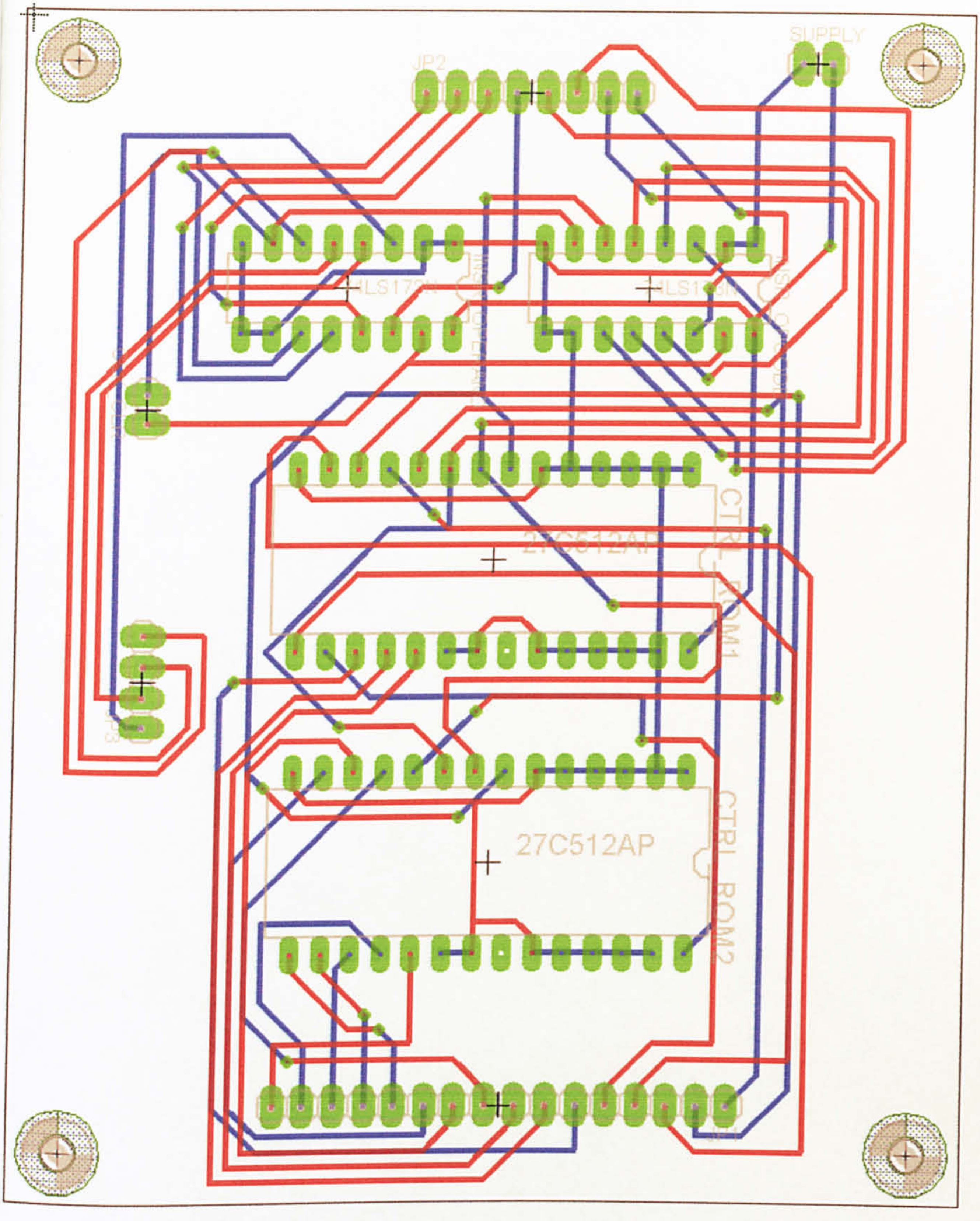
Program Counter



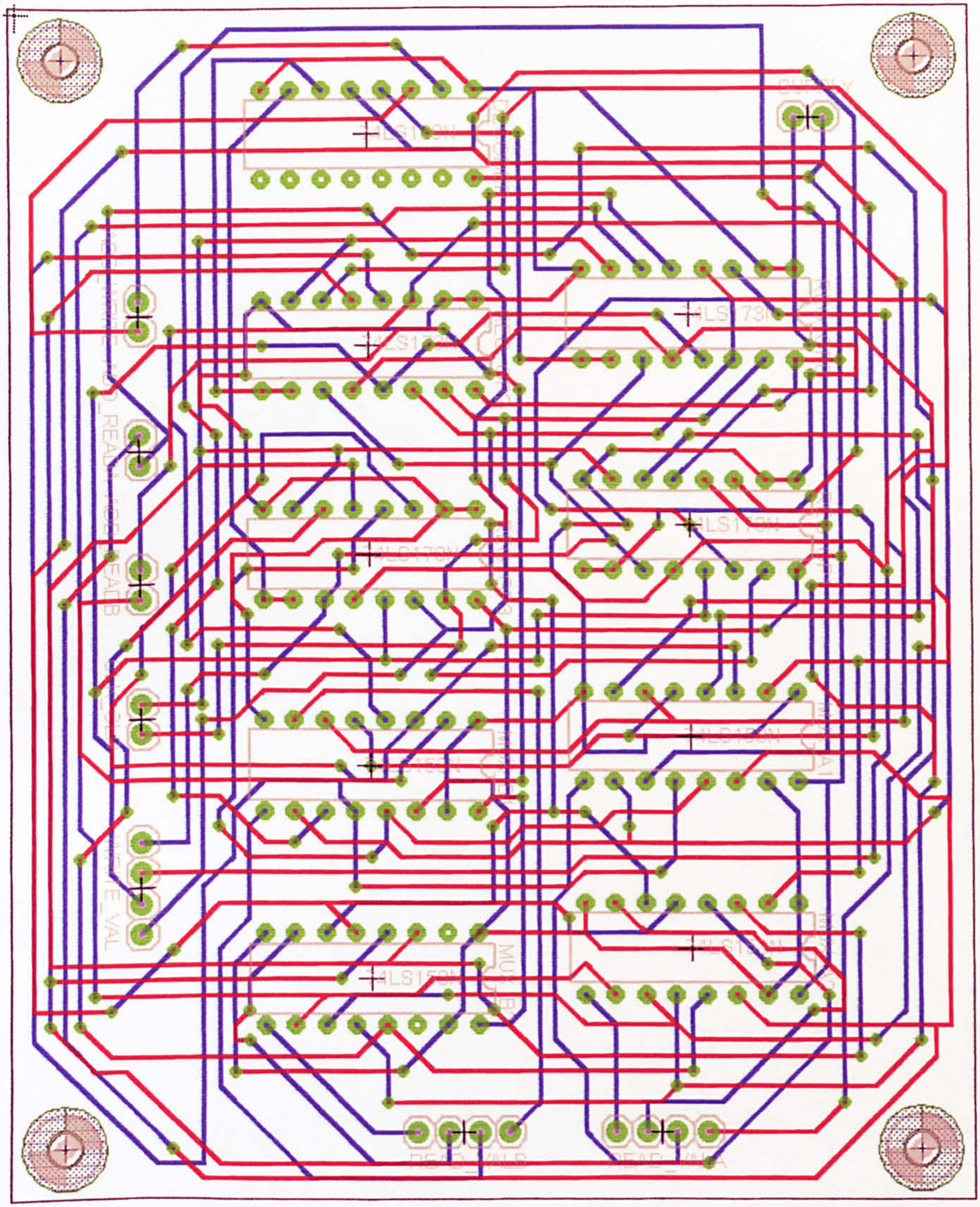
Instruction RAM



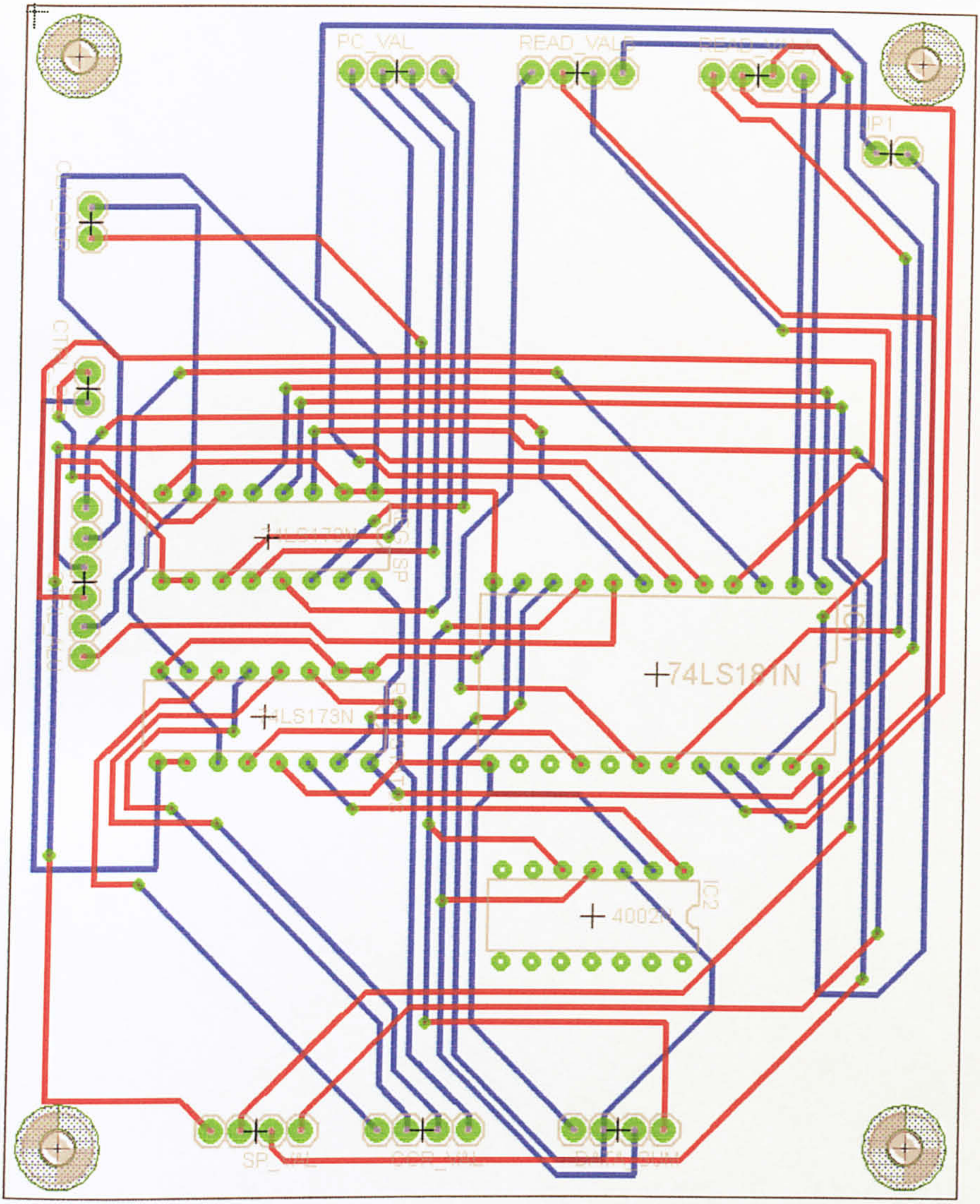
Control Logic



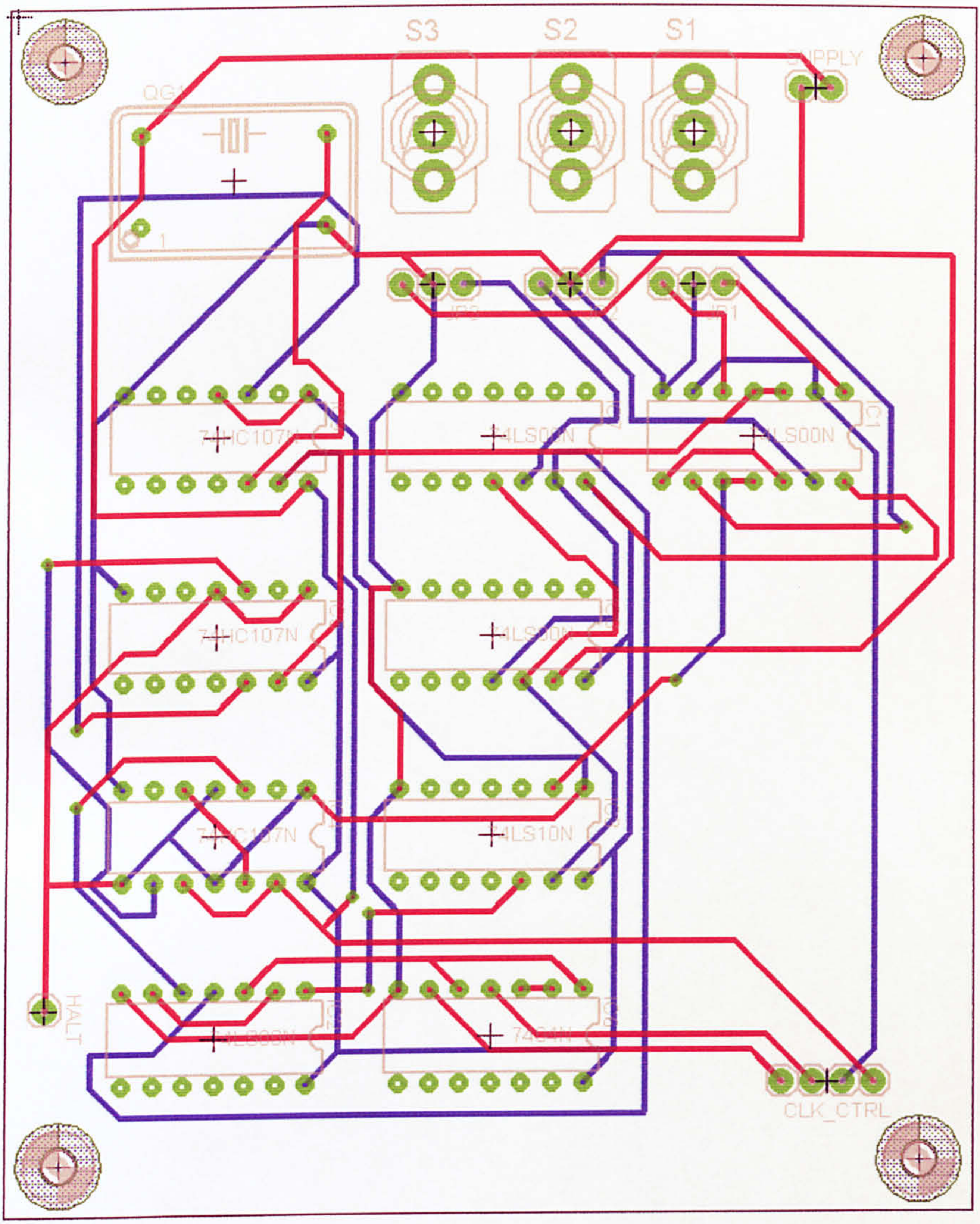
Register File



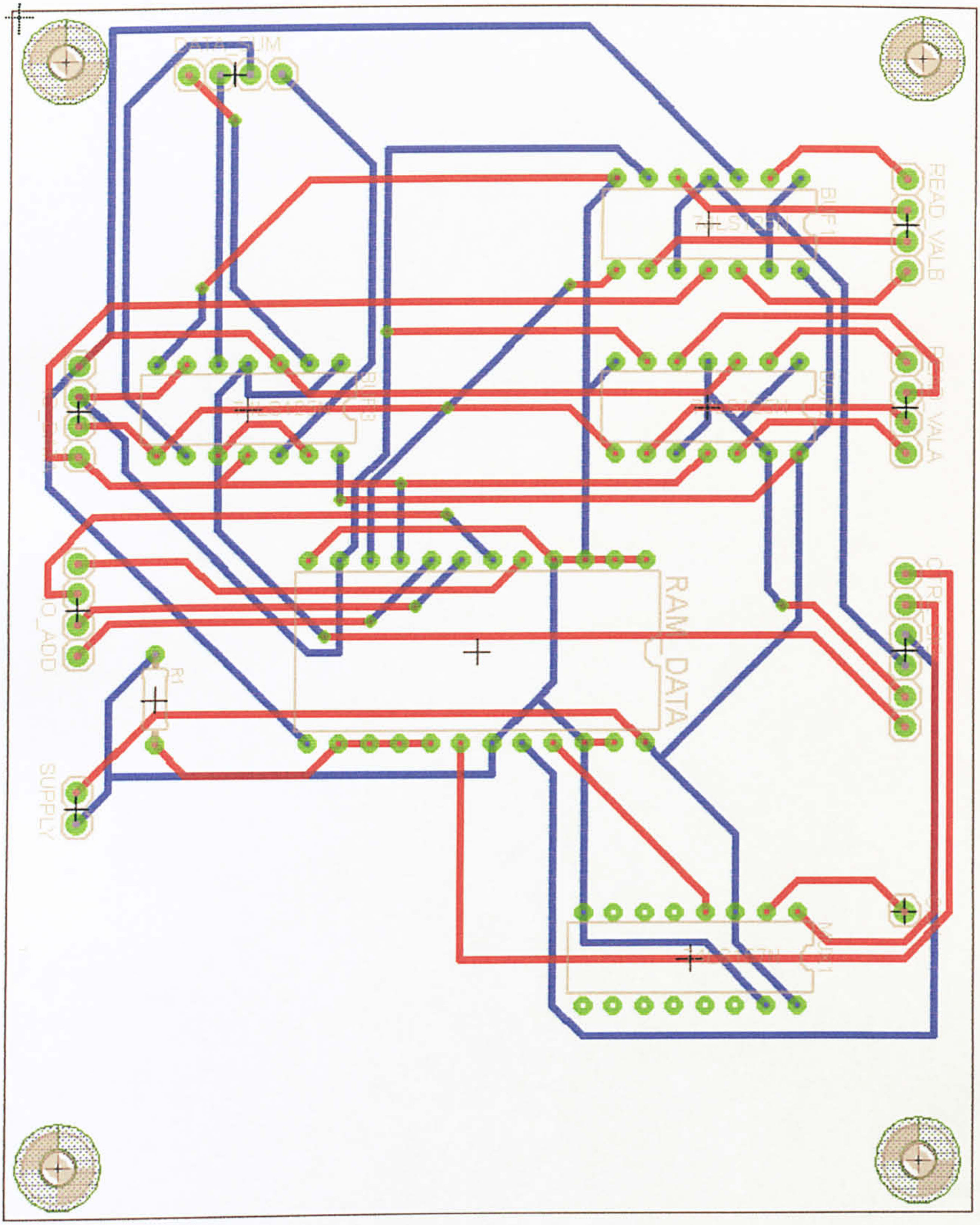
ALU

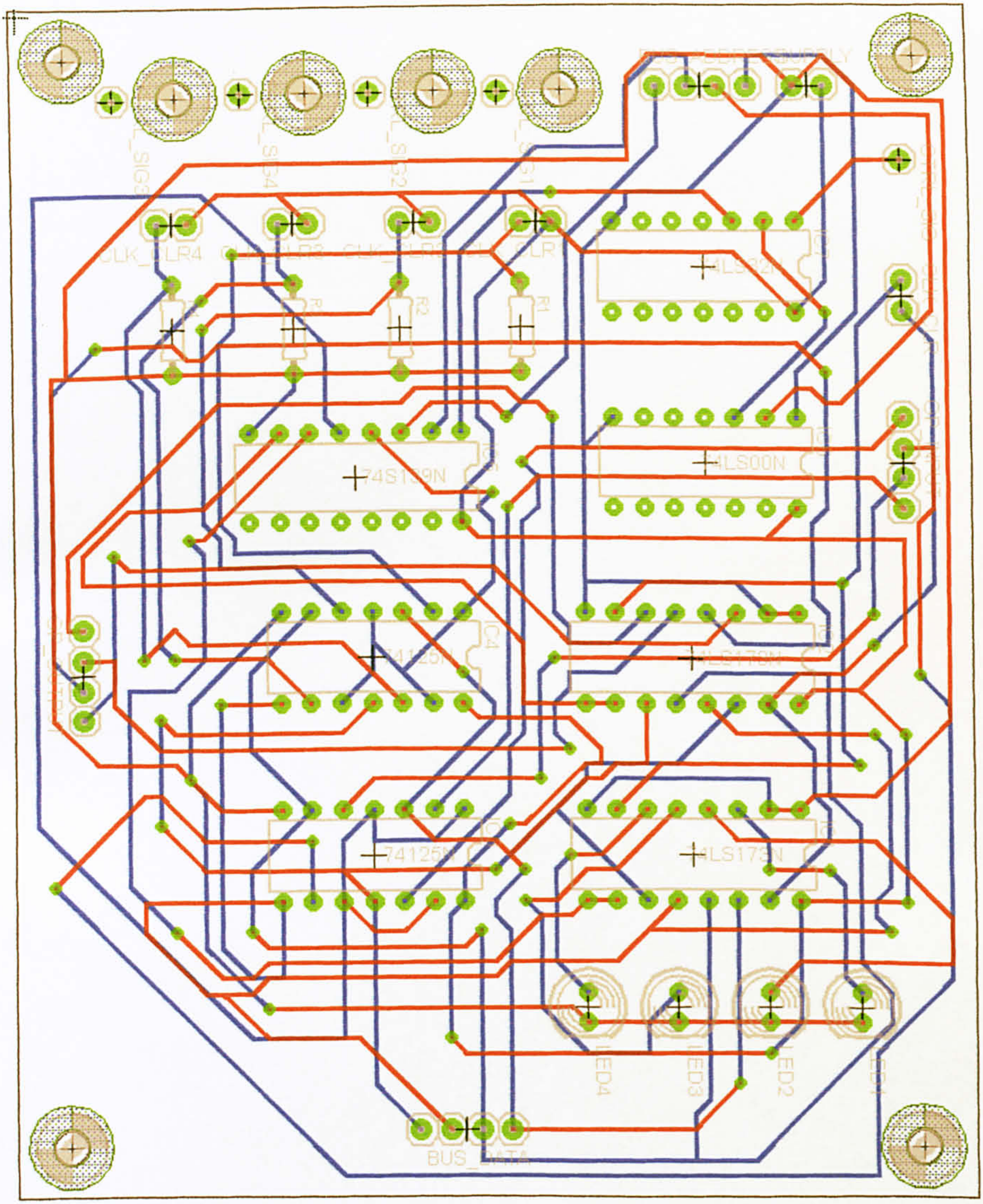

PC_VAL
READ_VAL
JP1
74LS173N
IC3
SP
74LS181N
IC1
74LS173N
4002N
IC2
SP_VAL
DATA_BUM

Clock


S3
S2
S1
SUPPLY
QG1
1
JP3
JP2
JP1
74HC107N
74LS00N
IC1
74HC107N
74LS00N
74HC107N
74LS10N
74LS00N
7404N
HALT
CLK_CTRL

Data RAM and Busses




L_SIG3
L_SIG4
L_SIG2
L_SIG1
CLK_CLR4
CLK_CLR3
CLK_CLR2
CLK_CLR1
R2
R1
74LS32N
74LS00N
74S139N
74125N
74LS175N
74125N
74LS173N
LED4
LED3
LED2
LED1
BUS_DATA

APPENDIX E

SOURCE CODE FOR CONTROL LOGIC

```
#include<18f452.h>
#use delay(clock=4000000)// clock=4MHz
#fuses XT,NOWDT,NOPROTECT,NOLVP

unsigned int1 op_code_3, op_code_2, op_code_1, op_code_0;

main()
{


   set_tris_c(0x0f);//set 1st 4 pins of portC as input n 2nd 4 pins
as output
   set_tris_b(0x00);//set all portB as output
   set_tris_d(0x00);//set all portD as output
   set_tris_e(0x00);//set 1st 3 pins of portE as output

   do{

   op_code_3 = input(PIN_C3);
   op_code_2 = input(PIN_C2);
   op_code_1 = input(PIN_C1);
   op_code_0 = input(PIN_C0);

      if ((input(PIN_C3) == 0) && (input(PIN_C2) == 0) &&
(input(PIN_C1) == 0) && (input(PIN_C0) == 0))  // op_code = 0000
(LDR)
      {//LDR
         output_b(0x09);    // 0000 1001
         output_c(0x90);    // 1001 xxxx
         output_d(0x2f);    // 0010 1111
         output_e(0x07);    // xxxx x111
      }

      else if ((input(PIN_C3) == 0) && (input(PIN_C2) == 0) &&
(input(PIN_C1) == 0) && (input(PIN_C0) == 1))  // op_code = 0001
(STR)
      {//STR
         output_b(0x09);    // 0000 1001
         output_c(0x90);    // 1001 xxxx
         output_d(0x1b);    // 0001 1011
         output_e(0x03);    // xxxx x011
      }

      else if ((input(PIN_C3) == 0) && (input(PIN_C2) == 0) &&
(input(PIN_C1) == 1) && (input(PIN_C0) == 0))  // op_code = 0010
(MOV)
      {//MOV
         output_b(0x09);    // 0000 1001
         output_c(0x90);    // 1001 xxxx
         output_d(0x9f);    // 1001 1111
         output_e(0x01);    // xxxx x001
      }

      else if ((input(PIN_C3) == 0) && (input(PIN_C2) == 0) &&
(input(PIN_C1) == 1) && (input(PIN_C0) == 1))  // op_code = 0011
(MOVi)
      {//MOVi
         output_b(0x09);    // 0000 1001
         output_c(0x90);    // 1111 xxxx
         output_d(0xff);    // 1111 1111
         output_e(0x02);    // xxxx x010
      }
      else if ((input(PIN_C3) == 0) && (input(PIN_C2) == 1) &&
(input(PIN_C1) == 0) && (input(PIN_C0) == 0))  // op_code = 0100
(add)
      {//ADD
         output_b(0x09);    // 0000 1001
         output_c(0x90);    // 1001 xxxx
         output_d(0xff);    // 1111 1111
         output_e(0x03);    // xxxx x011
      }

      else if ((input(PIN_C3) == 0) && (input(PIN_C2) == 1) &&
(input(PIN_C1) == 0) && (input(PIN_C0) == 1))  // op_code = 0101
(addi)
      {//ADDi
         output_b(0x09);    // 0000 1001
         output_c(0x90);    // 1001 xxxx
         output_d(0xff);    // 1111 1111
         output_e(0x02);    // xxxx x010
      }

      else if ((input(PIN_C3) == 0) && (input(PIN_C2) == 1) &&
(input(PIN_C1) == 1) && (input(PIN_C0) == 0))  // op_code = 0110
(sub)
      {//SUB
         output_b(0x89);    // 1000 1001
         output_c(0x60);    // 0110 xxxx
         output_d(0xff);    // 1111 1111
         output_e(0x03);    // xxxx x011
      }

      else if ((input(PIN_C3) == 0) && (input(PIN_C2) == 1) &&
(input(PIN_C1) == 1) && (input(PIN_C0) == 1))  // op_code = 0111
(subi)
      {//SUBi
         output_b(0x09);    // 0000 1001
         output_c(0x60);    // 0110 xxxx
         output_d(0xfe);    // 1111 1110
         output_e(0x02);    // xxxx x010
      }

      else if ((input(PIN_C3) == 1) && (input(PIN_C2) == 0) &&
(input(PIN_C1) == 0) && (input(PIN_C0) == 0))  // op_code = 1000
(CMP)
      {//CMP
         output_b(0x89);    // 1000 1001
         output_c(0x60);    // 0110 xxxx
         output_d(0xff);    // 1111 1111
         output_e(0x01);    // xxxx x001
      }

      else if ((input(PIN_C3) == 1) && (input(PIN_C2) == 0) &&
(input(PIN_C1) == 0) && (input(PIN_C0) == 1))  // op_code = 1001
(CMPi)
      {//CMPi
         output_b(0x09);    // 0000 1001
         output_c(0x60);    // 0110 xxxx
         output_d(0xfe);    // 1111 1110
         output_e(0x00);    // xxxx x000
      }
```