1-31-1993

Partial VLSI implementation of the architecture for reusable components (ARC)

Deepak Kakadasam
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

Partial VLSI Implementation of the Architecture for Reusable Components (ARC)

by

Deepak S. Kakadasam

This work describes a novel VLSI implementation of the Architecture for Reusable Components (ARC) processor, using Hardware Description Language (HDL). The main goal here is to achieve efficient execution of reusable software through proper hardware support. This involves the hard wired implementation of each instruction designed for the ARC processor.

Instructions are broken down into their logical functions, then modeled and simulated through the hierarchical design methods that HDL offers. The structural model of the processor has been developed and simulated. The purpose here has been to begin work on the design and implementation of the ARC processor.

The instructions were built using HDL modules, and then simulated using a logic simulator. The effect of internal propagation delays in the execution of the logic modules have been investigated. Changes in delay parameters have been applied to obtain correct logic transfer operations. The redundancy in the logic transfer operations have also been investigated to see parallelism at the instruction execution level.

PARTIAL VLSI IMPLEMENTATION OF THE ARCHITECTURE FOR REUSABLE COMPONENTS (ARC)

by

Deepak S. Kakadasam

A Thesis
Submitted to the Faculty of the
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science

Department of Electrical and Computer Engineering
January, 1993

APPROVAL PAGE

Partial VLSI Implementation of the Architecture for Reuseable Components (ARC)

Deepak S. Kakadasam

Dr. Durga Misra, Thesis Advisor
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Lonnie R. Welch, Committee Member
Assistant Professor of Computer and Information Science, NJIT

Dr. Walter F. Kosonocky, Committee Member
NJIT Foundation Chair for Optoelectronics and Solid State Circuits, NJIT

BIOGRAPHICAL SKETCH

Author: Deepak S. Kakadasam

Degree: Master of Science in Electrical Engineering

Date: January, 1993

Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1993

- Bachelor of Electrical Engineering,
  Bangalore University, Bangalore, India, 1990

This thesis is dedicated to

my parents

ACKNOWLEDGMENT

I take this opportunity to express my gratitude to Dr. Durga Misra, Assistant Professor, Electrical and Computer Engineering Department of NJIT and to Dr. Lonnie R. Welch, Assistant Professor, Computer and Information Science Department of NJIT for their encouragement and valuable guidance throughout the course of this thesis. Their helpful hints, suggestions and patience were of immense help.

Special thanks are due to Dr. Walter F. Kosonocky for serving as member of the committee.

I also wish to thank my friends and colleagues Elie I. Mourad, Subramanyam Ayyagari, Sudesh J. Tekpat, Christine I. Mourad, and Nathaniel McCaffrey for their patience, advice and the help they extended to me.

I would like to extend my thanks to the various people on the net, who responded with invaluable suggestions to my queries and to all those in the Real Time Operating Systems Lab, NJIT for their help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A Processor called ARC (an acronym for Architecture for Reusable Components) is being developed to which is a virtual machine designed for executing programs that use abstract data types (ADTs). The ADT mechanism is provided by many modern programming languages and is often employed during system development to promote modularity and reuse. The major contribution of ARC is that it supports Asynchronous Remote Procedure Call (ARPC), a model of parallel execution that works well for programs developed by layering ADTs.

Research indicates that a computer designed in conjunction with a programming language is more effective than one designed for use by programs written in diverse languages. The ARC is unique in the sense that it is being implemented to support RESOLVE [1] (REusable SOftware Language with Verifiability and Efficiency), a language currently under the final stages of development at New Jersey Institute of Technology. RESOLVE provides for the ADT construct. The ARC processor was designed to address the potential inefficiencies of reusable software.

1.1 Statement of Design Flow

This thesis is part of a design strategy whose objective is to implement the ARC as an integrated circuit. The ARC may be considered as an Application Specific Integrated Circuit (ASIC). Its application being the efficient

implementation and execution of RESOLVE.

Gate-Level design is practically dead for large systems. In todays world, hardware complexity has increased beyond schematic comprehension. Integrated circuits (ICs) are getting so complex that schematics show only a web of connectivity and modern day engineers are therefore moving toward hardware description languages (HDLs). HDL coupled with logic synthesis is the future for IC design.

The ARC has been broken down to its lowest level, i.e. the instructions themselves and a part of the processor have been structurally modeled using HDL. The HDL used here is MHDL [1]. Each individual instruction has been designed at the logic level, modeled structurally using MHDL and simulated on a logic simulator (Lsim). The purpose of this thesis has been to start the design process of the ARC as an IC. It is beyond the scope of this thesis to construct and implement the entire processor.

ARC is an Instruction Set Architecture and the idea here has been to implement each instruction as a single entity. This approach gives us an understanding of instruction execution and provides details of the number of Clock Cycles per Instruction (CPI). This data can be used to modify the instruction set such that most instructions take the same number of CPI to execute.

---

[1] MHDL is a trademark of Mentor Graphics Corporation

1.2 Overview of the Chapters

Chapter 2 gives a brief overview of the Reduced Instruction Set Computer (RISC) Architectures for VLSI implementation with a perspective towards implementing Instruction Set Architectures. It also briefly talks about Very high speed integrated circuit Hardware Description Language (VHDL) and the advantages it offers in the area of digital system design.

Chapter 3 gives an explanation of the breakup involved in the design and development of the architecture of ARC.

Chapter 4 deals with the system modeling and the hierarchy exhibited in the structural modeling.

Chapter 5 presents results of the simulation work and also deals with the architectural implementation and the problems faced by a design engineer. It also discusses the delays associated with the logical elements and its impact on the execution time of the instructions.

Chapter 6 discusses conclusions drawn from the simulation. A note on the direction for future research is also included.

Appendix A contains a listing of the programs developed.

Appendix B contains the internal operations of the T-states.

Appendix C contains some of the control circuits with the optimization for parallelism exhibited in instruction execution sequence.

CHAPTER 2

LITERATURE REVIEW

2.1 Overview of RISC Architectures

Reduced Instructions Set Computers (RISC) aim for both simplicity in hardware and synergy between architectures and compilers [17]. Reduced instruction sets simplify compilers, alleviate software crisis and provide for improving architecture quality. Increasing the size or complexity of a digital circuit may either enhance or impair the overall system performance, depending on how judiciously the added complexity is chosen [4]. The RISC project was started at U. C. Berkeley in 1980.

Simpler instruction sets help drastically reduce control logic thus freeing silicon area, which could be used for on-chip registers or memory. RISC architectures are register oriented. Instructions operate on the contents of two registers, or one register and the immediate field included in the instruction. The result of this instruction is either written into a third register, or is used as effective address for memory access. All instructions follow about three stages in their execution process. They are Instruction Fetch, Instruction Decode and Instruction Execute.

RISC architectures use instruction pipelining. Essentially, pipeline architecture is a way of exploiting inherent parallelism or providing additional resources to create necessary parallelism. A simple pipeline can

be built from inherent concurrency between the fetching of one instruction and the execution of the previously fetched instruction.

In other words, when one instruction is being executed, the next instruction is being fetched. If the next instruction is already available for execution at the end of the current instruction, the processor overlaps the fetch cycle with the execution cycle. This is a typical instruction pipeline. To discuss the RISC architecture concept in its entirety is beyond the scope of this chapter. Exhaustive references will be provided thus minimizing the learning cycle for new workers in this area and providing a solid base for future research.

The ARC can be classified as a RISC processor. The ARC supports ARPC by performing data synchronization, automatic parameter restoration, and dynamic load balancing. RISC machines provide primitive instruction sets, and are designed by examining the way compiled code uses the instruction sets of computers and then providing instructions that will be used frequently.

The ARC has been developed on these very same principles. The usefulness of RISC processors as nodes in parallel computers has been investigated in recent years. The ARC is developed on the idea that it would be used in a multi processor environment.

2.2 VHDL and Digital System Design

In the design of large digital systems, more time is spent on changing formats for using various design aids and simulators. Computer Aided Design (CAD) tools have significantly contributed in reducing ASIC development time. Until recently, design engineers have had to rely on schematic capture tools which used a mouse and special software to draw the schematics on a CAD screen. With the increasing complexity of ASICs, schematic entry is becoming impractical. A hierarchical design schematic tool also demands that a design engineer enter dozens of schematics. The engineer then has to use schematic capture packages to create a netlist that describes the interconnection of all the inputs and outputs of the logic gates in the ASIC.

This led to the development of Hardware Description Languages (HDL) to ease the data entry tasks of ASIC designers. HDL is used to describe hardware for the purpose of simulation, modeling, testing, design and documentation of digital systems. VHDL has now become a standard HDL. These languages provide a convenient and compact format for hierarchical representation of functional and wiring details of digital systems. Some HDLs are a simple set of symbols and notations for replacing schematic diagrams of digital circuits. Other HDLs are more formally defined and can be used for representation of hardware in one or more levels of abstraction.

HDLs also need to have a simulator and a hardware compiler program. A simulator can be used for design verification, while the compiler is used

for automatic hardware generation. VHDL specifically has constructs ranging from behavioral level to gate interconnection level. There are many HDLs and they only differ in the sense that they offer more than just the standard environment for simulation purposes. The government and the industry standard is based on VHDL, which is the VHDL-IEEE 1076 language. A merger of tools developed by vendors with user friendly front end programs together with VHDL code, has only created many types of HDLs. MHDL is one such kind which is merged with GDT [2] (Graphic Design Tools). GDT offers the merger of HDL together with their interactive simulator (Lsim) to provide for a complete design and simulation environment.

2.2.1 Levels of Abstraction in HDL

HDLs offer various levels of abstraction to describe the hardware. In MHDL particularly, four levels of abstraction have been defined; Behavioral, Register transfer, Logic, and Device levels. Figure 2-1 shows the levels of abstraction and also includes information about their applicability.

A Behavioral description is the most abstract. It represents the design only in terms of its top-level chip architecture. Only input and output behavior is specified. The internal structural details are omitted. The Behavioral level is the most appropriate for fast simulation of complex hardware units,

[2] GDT and Lsim are trademarks of Mentor Graphics Coproration

verification and functional simulation for design ideas, modeling standard components, and documentation.

Register transfer level models describe a design in terms of blocks. Input/Output behavior, including bus oriented data is specified. Timing delays may also be included. This is sometimes also referred to as structural modeling.

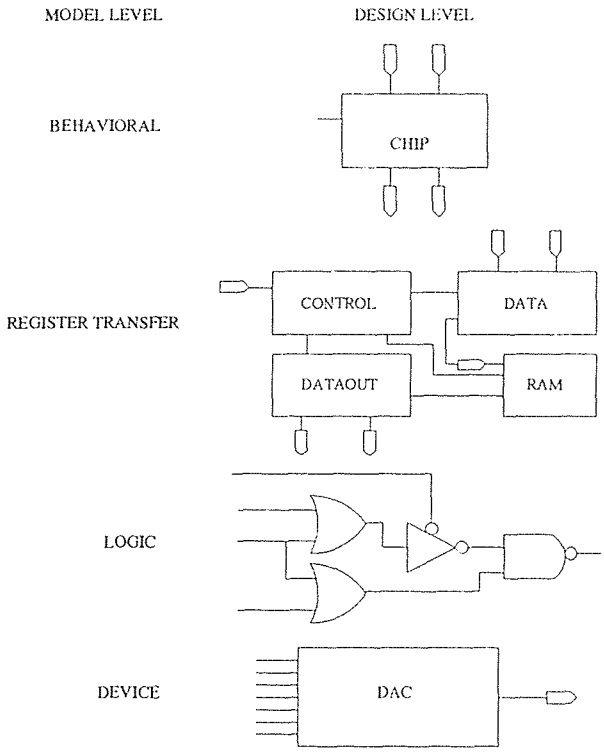


Figure 2-1. Levels of Abstraction.

Logic level models describe a design in terms of its logical elements, such as AND and OR gates, latches and registers. Bit oriented data is specified at this level for control lines. Timing delays may be specified in terms of

rise and fall time delays, and parasitics estimated on net output capacitance.

Device level models describe a design in terms of its device level characteristics and possible analog behavior. This level is useful for modeling op-amps and comparators.

In standard VHDL three levels of abstraction are defined; Behavioral, Dataflow and Structural. MHDL offers the very same levels of abstraction with the only addition being the device level.

2.2.2 Basic Concepts of HDL

HDL used in conjunction with logic simulation and synthesis tools make it a powerful tool. Any hardware component can be described, simulated and tested. The description of a particular component in HDL consists in creating a software module where the inputs and the outputs are first stated. The next step is a build section where instances of components are created and netted. The last stage is to create test vectors for the above module and simulate the design. This format is the same for all levels of abstraction. See Figure 2-2.

The first line declares the name of the module. The module could also contain arguments whose value could be passed at simulation time. The Input/Output declaration defines the input and output terminals of the module. A bus type declaration could also be made by defining a particular terminal as an array of n-bits. The Build section of the module is that section

that is executed only at circuit build time. All information that includes circuit connectivity must be included in this section. An initialize section could also be added to initialize all terminals prior to the simulate section. The initialize and the simulate section together could be included in separate files called the initialization file and the testvector file. The module is then compiled and it is now ready for the simulation stage.

```
MODULE name( )

{
        IN                  //Declare Inputs
        OUT                 //Declare Outputs

        BUILD{              //Start to build

                INSTANCE    //Call Instances of primitive logic
                  ..
                  ..
                  ..
                NET         //Net all instances of primitive logic
                  ..
                  ..
                  ..

                }

        SIMULATE{           //Simulate section
                        ..
                        ..
                        ..  //A test vector file couldbe used
                        ..  //instead of this section
                    }

}
```

Figure 2-2. Component Description.

2.2.3 Designing with HDL

Expressing designs in HDL can provide several benefits. An HDL description can be used as a specification of the design. There are many HDLs. Some of the more important ones are Verilog [3], and MHDL. Hardware description languages allow for easy text processing, whereas binary schematics usually require a graphics editor. HDL offers the advantage of simulation which can uncover design errors that would otherwise be detected only when the hardware is built. It also provides logic synthesis. There are synthesis tools which can take an HDL description of a design and generate a gate level implementation with library components.

These tools help optimize the design with respect to speed, circuit size, or some other cost function. The other advantage is, HDLs are more like the C programming language. It is very easy to learn. Finally, HDL is the best way to document a design. A well commented HDL description can give a better and more concise documentation than a set of schematics that show gate level details.

---

[3] Verilog is trademark of Cadence Design Systems

CHAPTER 3

SYSTEM ARCHITECTURE

3.1 Introduction to Architecture Development

Integrated circuit technology has made possible the production of chips with hundreds of thousands of transistors. Systems of such complexity remain difficult to design. The computer architect faces problems in the areas of system partitioning with subglobal specification, subsystems interface specification and verification, and overall system integration.

In the design of any processor architecture, the designers should make sure of certain facts: its effectiveness in supporting high level languages, and the base it provides for system level functions. Both the cost and the performance of implementation should be taken into consideration to scan the effectiveness of the architecture.

ARC is proposed to be a 32-bit processor. Its implementation in VLSI would attempt to compromise between performance and functionality. It all depends on the instruction set designer to carefully consider both the usefulness of the instruction set for encoding programs and the performance of implementation of that instruction set.

Most programs are written in high level languages (HLL) and the role of the architecture as a host for programs depends on its ability to express the code generated by compilers for the high level languages. A large

instruction set architecture will require microcode to implement the instruction set. In VLSI, silicon area limitations often force the use of microcode for all but the smallest and simplest of instruction sets. An additional level of translation is required in microcoded processors. This can be avoided in processors with simpler instruction sets by implementing the control logic through hardwired means.

The Architecture and its strength as a compiler target determine much of the performance at the architectural level. The organization of hardware for an architecture can dramatically affect quantitative measures of architectural performance. Since the architecture imposes implementation requirements on the hardware, performance measurements made on the architecture that are implementation dependent may not yeild realistic measures of the performance of an actual implementation of the architecture.

The architecture affects the performance of the hardware primarily at the organizational level. Smaller effects occur at the implementation level where technology becomes relevant. The key goal in implementation is to provide the fastest hardware possible. In other words this is to minimize the overall clock speed, to reduce the overhead on instructions as well as organizing the hardware to minimize the delays in each clock cycle.

3.2 Architecture of ARC

The ARC processor has the following principle elements: control unit, data-path, and memory and an operation processor. The control unit has all the control logic to manage data transfer between the datapath, memory and the external interface.

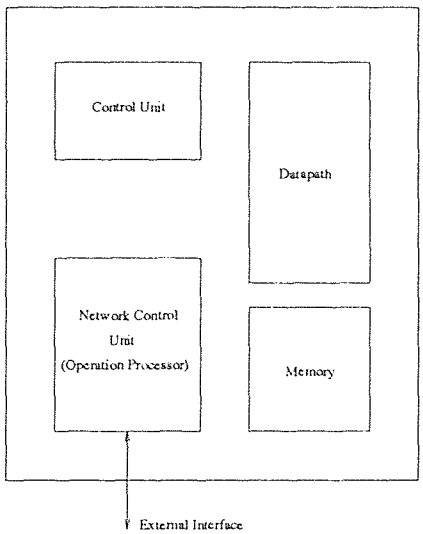


Figure 3-1. Constructs of the ARC.

3.2.1 The Control Unit of the ARC

The design of a controller depends on the datapath requirements of the system. The control unit loads instructions from instruction memory (IM) into the instruction register (IR) and uses a program counter (PC) to hold the address of the next instruction to fetch from memory. The architecture is based on the instruction set, The instruction set is listed in Figure 3-2. The instruction set is basically divided into Data Movement instructions,

Arithmetic and Logic instructions, Control Flow instructions, and Miscellaneous instructions.

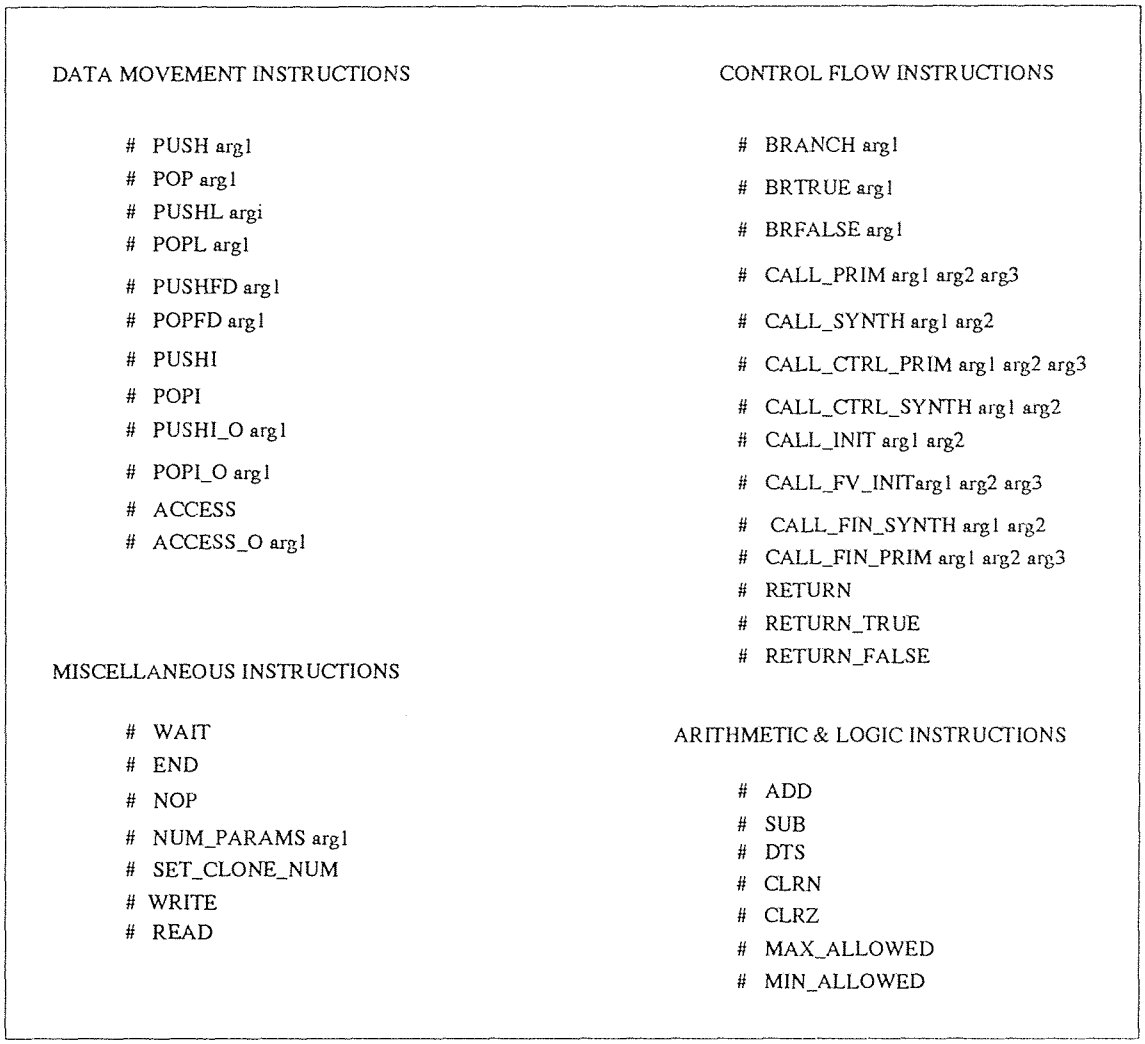


Figure 3-2. Instruction Set of the ARC.

ARC supports programming language constructs that facilitate software reuse. The Control Flow instructions have a need for a separate hardware unit called a Network Control Unit (NCU), which would facilitate data transfer to handle ARPC. These instructions have not been included in this

preliminary design. The instructions that have been designed are those that are included in APPENDIX B.

Each instruction follows its own unique steps to complete its execution. The execution of each instruction is accomplished by the execution of operations of a combination of T-state. Each T-state has certain operations associated with it. These T-states generate the required control signals within the control unit. These control signals are fed to the datapath where they operate on the elementary units of the datapath. These signals perform the operation of gating data from a register to a bus, gating data from a bus to a register, control of the Arithmetic and Logic Unit (ALU) operation, etc.

Table 3-1. Major Stages and Their Functions

Stage	Mnemonic Task
Instruction Fetch (T1)	Send out the PC to the MAR, Increment PC
Instruction Fetch (T2)	Get memory contents of location Pointed to by the MAR in InsM.
Instruction Decode & execute (T3)	Perform required operations as demanded by the instruction.

The T-state diagram is shown in Figure 3-3. The execution of a particular instruction is done by a unique combination of T-states. The operations occuring within the T-states are included in APPENDIX B.

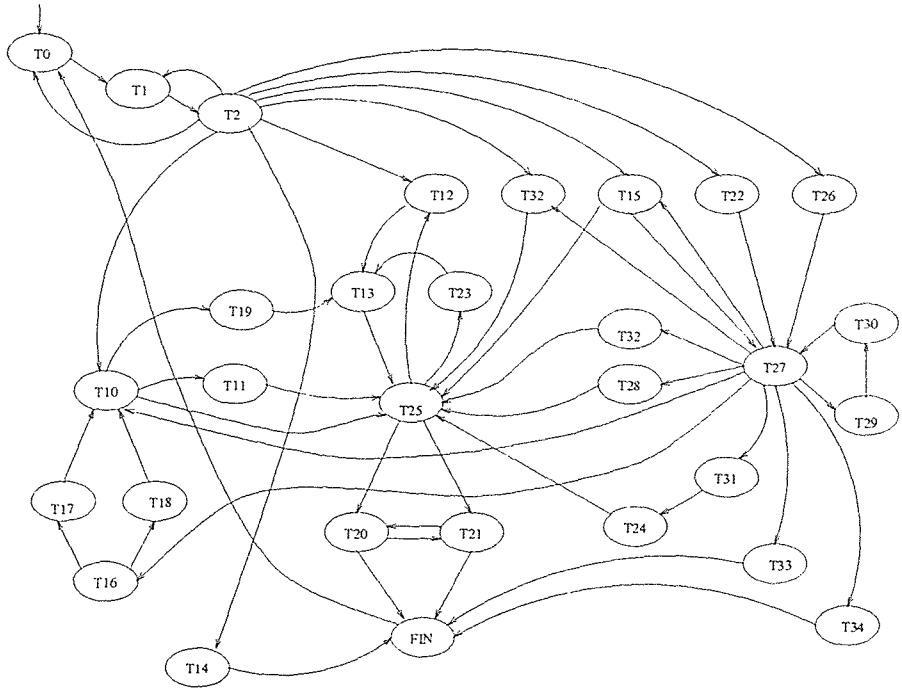


Figure 3-3. T-state Diagram of the ARC.

The T-states for some instructions are shown in Figure 3-4.

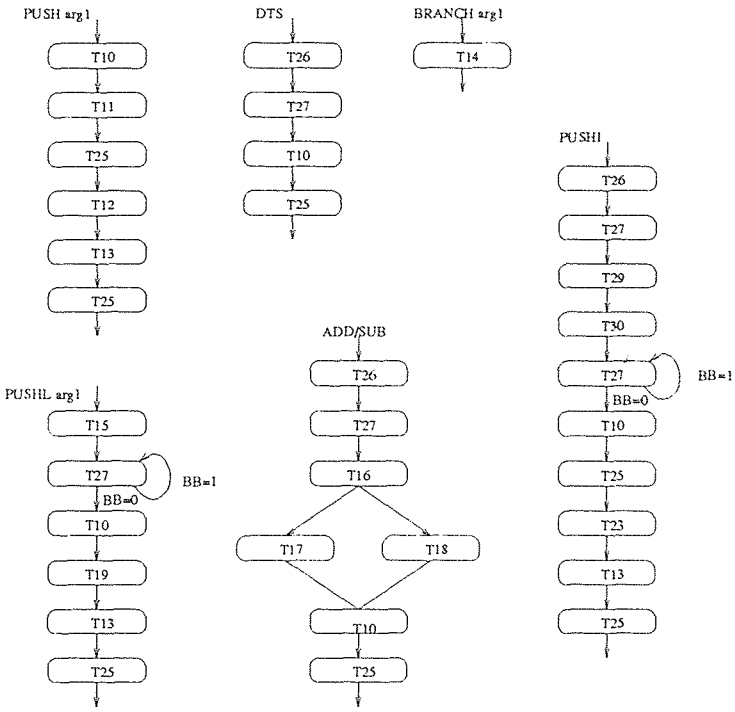


Figure 3-4. T-state Diagram of Some Instructions.

The Control Unit is designed in such a way that it exhibits hierarchy in its implementation. The control unit is split into two smaller units: Cu1 and Cu2. There are some instructions that do not go beyond the T2 state. For this the entire unit waits until it gets a start signal and jumps from state T1 to T2 immaterial of what the instruction is. Then if the decoded instruction needs execution on state T3, the control is transferred to the next unit and Cu1 waits until Cu2 has generated a finish signal. Then Cu1, jumps back in control and relieves Cu2 of its duties. See Figure 3-5 for the structure of the basic control unit.

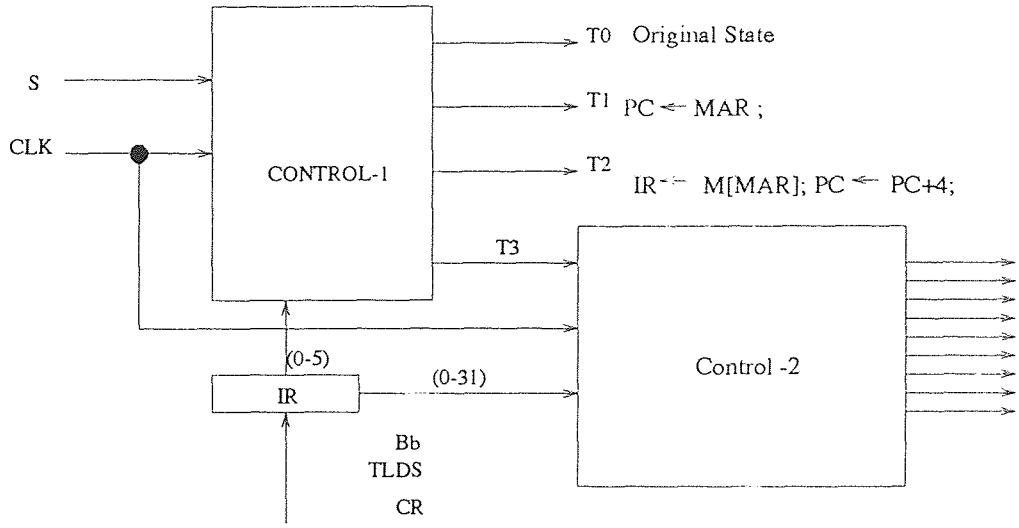


Figure 3-5. Structure of the Basic Control Unit

The Cu2 stage of the control unit is modeled such that each individual instruction execution can be observed. The instructions that have been implemented are those that give a vivid idea of the processor. The ones that have not been implemented are for later development. The control units for the individual instructions have been modeled. These are basically counter units with combinational logic inputs.

The instructions have been grouped on the basis of the number of T-states that they take to execute. This is shown in Table 3-2. The number of instructions used by the ARC at this time is about 40. To represent these 40 instructions, we need ($2^n = 40$) n = 6 bits. Thus in the Instruction register, the bits 0 through 5 represent the instruction. The remaining 26 bits are used as immediate data. Depending upon instruction usage or design, the remaining bits of the instruction register could be used as necessary.

Table 3-2. Number of T-states Taken by the Instruction.

Instructions	# of T-states
NOP, WAIT, END, BTRUE BFALSE	2
BRANCH arg1, BRTRUE, BFALSE	3
CLRN, CLRZ	5
DTS	6
PUSH arg1, ADD, SUB, PUSHL arg1,	
POPL arg1, POPFD arg1, POPI arg1	8
PUSGFD arg1, POPI, PUSHI_O arg1,	
PUSHFD arg1	9
PUSHI	12

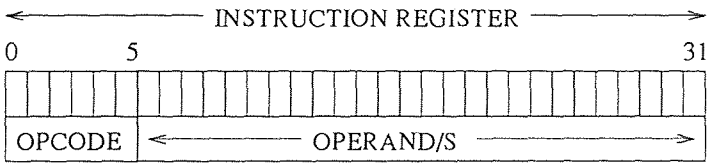


Figure 3-6. Contents of the Instruction Register.

The Cu2 control unit gives a start signal to a particular counter depending upon the instruction and the counter in turn drives the decoder unit which generates the necessary T-states. In actuality each T-state pulse is anded together with a $4-\Phi$ clock to generate the actual control signals

which are gated to the datapath.

Let us consider an instruction and deal with its implementation in the control unit. The ADD/SUB would be one such instruction. These instructions take about eight T-states to execute. The main control unit Cu1 takes care of the top two T-states and we are left with generating the remaining six T-states. So once the Control Unit has decoded that the instruction is an ADD/SUB instruction, control is passed over to the Cu2 stage which enables the necessary counter by driving that particular counter which further drives a decoder to produce the necessary T-state signals. More of this is explained in Chapter 4, where examples of some instructions have been discussed.

3.2.2 Datapath Components of the ARC

Arithmetic operations in the critical path require careful logic and circuit design. Care should be taken to see that there is minimal loading on the adder. The ARC has a 32-bit ALU. See Figure 3-7 for a description of the datapath. Its basic parts, namely latches, functional unit, and busses are the following:

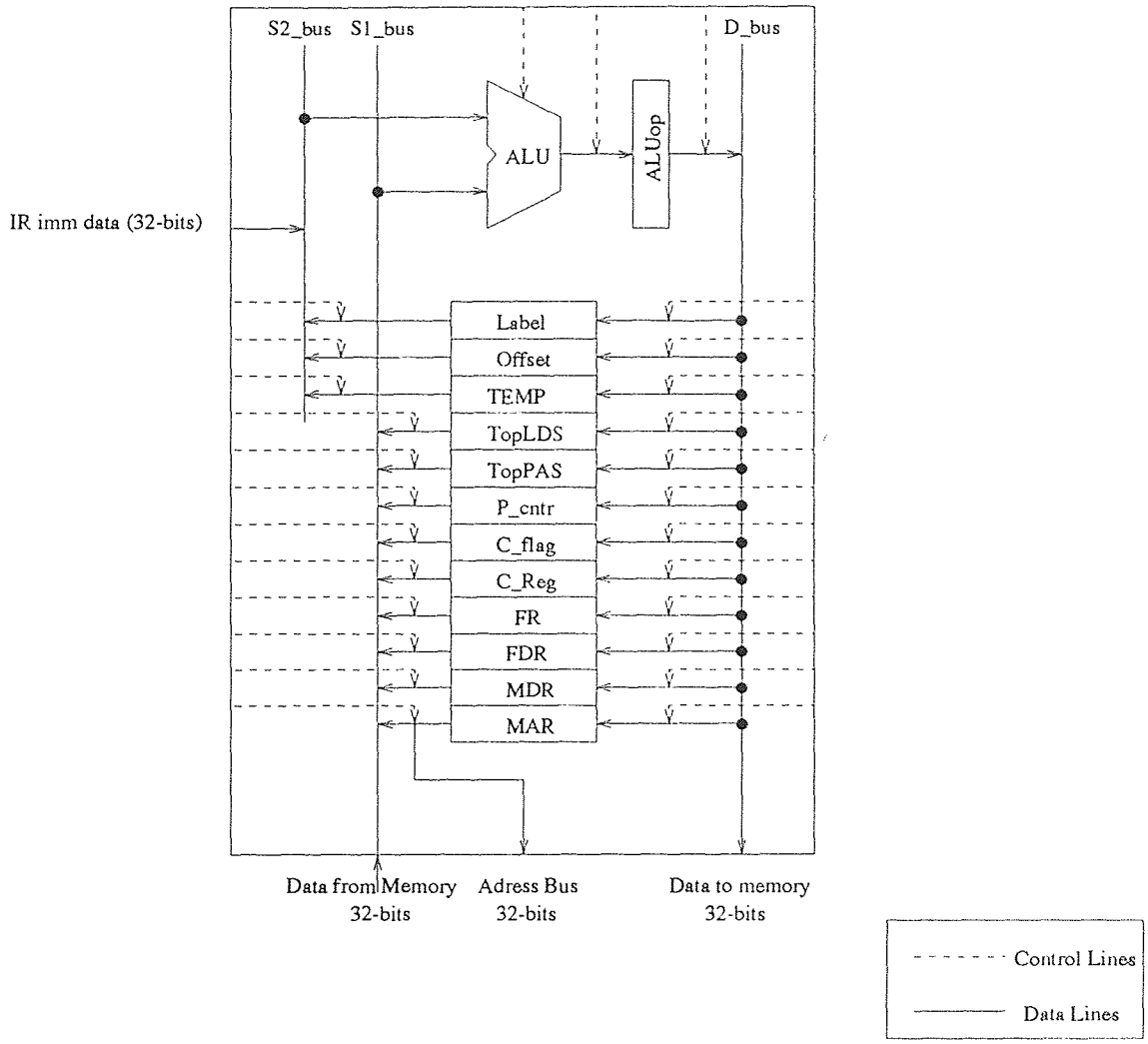


Figure 3-7. Components of the Datapath.

- Register file: Which includes the following registers.
  - •TOP of LDS pointer to the Local Data Stack
  - •TOP of PAS pointer to the Parameter Address Stack
  - •TOP of LDS
  - •PC Program Counter (Address of the next instruction)
  - •C_Flag Control Flag (Indicate Zero or Negative)

- C_reg Holds Clone number
- FR Facility Register (contains an offset)
- FDR Facility Data Register (contains an address)
- MDR Memory Data Register
- MAR Memory Address Register
- Label (2-bit unique location identifier)
- Offset (address offset for PC incrementation)
- TEMP (Temporary Storage area)

- Arithmetic and Logic Unit (ALU)

The ALU is a 32-bit carry look-ahead adder which is capable of addition and subtraction. The subtraction is performed by 2's complement addition. The structure of the ALU is shown in Figure 3-8. When the control signal is logic 1 (assuming logic 1 = HIGH level), the ALU performs addition. Two's complement addition (subtraction) is performed when the ALU control is logic 0. It is to be noted that the ARC does not directly support floating point arithmetic. For intensive applications a numeric co-processor could be used.

3.2.3 Memory Components of the ARC

There are four kinds of memory in the ARC. They are the Local Data Stack (LDS), Instruction memory (InsM), Indexed memory (IM), and the Facility memory (FM). The data memory components are the LDS and the IM. FM

contains a table for each facility used in a program. The LDS is intended to be used for storing activation records for operation invocations. Each entry is typically the address of a variables representation IM memory. The LDS has a word stack because the activation records are stacked and are created and destroyed in a last-in-first-out fashion. The top pointer of the LDS is incremented (or decremented) when an item is pushed (or popped). The LDS is not a true stack since entries can also be accessed randomly. Values on the LDS can be copied, swapped with values in IM, destructively read and written.

Indexed memory contains static module data and the representations of local variables. IM entries are addressed by specifying a base and an offset. Values can be copied, swapped with values in LDS, destructively read and written.

Facility memory (FM) holds tables used by CALL instructions. These tables are called run-time facility records (RFRs). The facility register (FR) is also used by CALL instructions and points to the RFR of the current facility.

The ARC also contains the Parameter Address Stack (PAS) to automatically restore parameters. The PAS contains a record of addresses of all items pushed onto the LDS, since the LDS is used for passing parameters to operations.

ARC is designed to be a word addressed machine, providing several advantages over byte addressed architecture. Word addressed architecture simplifies memory interface since insertion and extraction hardware is not required. This is particularly important since instruction and data fetch and data store are in the critical path. Word addressing also makes computation more efficient.

The number of address bits is 32, which gives us about 2^{32} = four gigabytes of physical address space. This amount of memory is not intended for use with the ARC in the immediate future, but later developments and applications may demand this amount of memory. At this time the ARC memory is split up with the LDS and the InsM having about 4K address space, the IM and the FM having about 64K address space.

Modern processors face the problem that the sum of memory access time and the memory mapping time is too great to allow the processor to run at its full speed. There are several constraints on implementing memory on chip and off chip memory leads to the above mentioned delay. The present ARC can hold all of its memory on chip given that the memory is limited as mentioned previously. But this is not the case as the ARC is further developed. There may be a need to implement a cache memory. In some processors increased bandwidth was achieved by implementing an instruction cache [14]. All these considerations should be dealt with in the future.

CHAPTER 4

4 SYSTEM MODELLING AND SIMULATION

This chapter provides a description of the actual modelling and the procedure adopted in the modelling process. It discusses the inherent problems associated with the modelling process itself.

4.1 Fundamentals of Modelling

The idea was to break up the instruction set and to simulate them individually. The datapath remains the same for all the instructions. Most of the control unit also remains the same. There are a few differences as to producing the right number of T-states for each instruction. We shall start first with the design of the datapath and then go on to talk about the control unit followed by a discussion on memory implementation..

Modern day IC designers use just behavioral models of their circuits to build their designs. These people in the commercial world have software that can convert their circuit from behavioral descriptions to structural models and also easily create a netlist. Not many designers start with a structural model description. Structural modelling has been used here so that work on the processor can continue at the university level and future researchers in this area can actually implement certain blocks of the ARC, due to the fact that the internal details of the components are available in the structural model. Behavioral models treat the circuit black-box.

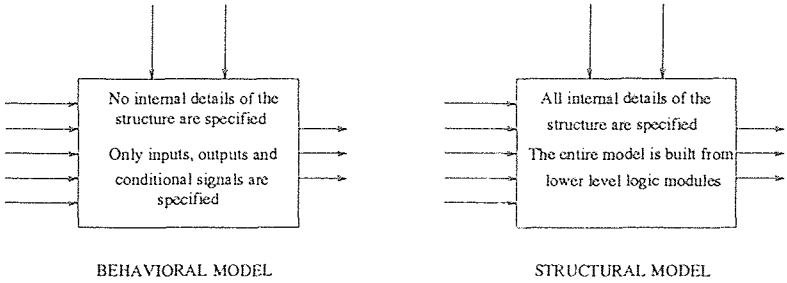


Figure 4-1. Behavioral and Structural models.

4.2 Design of Datapath

4.2.1 Register Set Design

The datapath was previously introduced in Chapter 3 and its diagram is shown in Figure 3-7. The datapath consists of a register set, ALU and busses. These resources are interconnected through a pair of source busses and one destination bus. As in most datapath designs, the convention followed here is that, there is only one unique path to traverse through the datapath. All data originates from a register, is gated to one of the source busses, onwards into the ALU, then into the destination bus and back into a register. The source and destination registers are determined by the instruction.

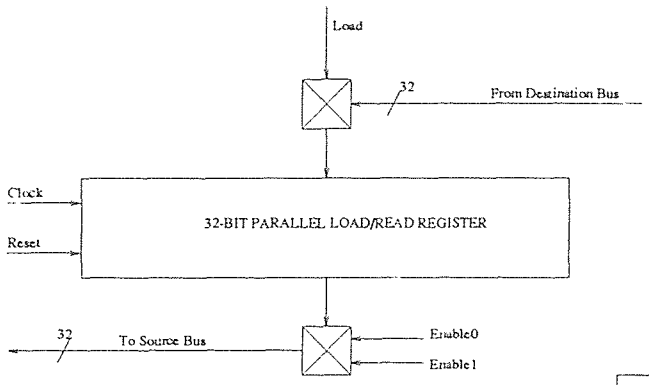
The instruction operations were observed. It was noticed that only three registers need use of a second source bus, to input data into the ALU. This bus has been provided so that future design changes in the instruction set can be accomodated. The S2_bus is local or internal to the datapath. The S1_bus and the D_bus are global. It is to be noted that at the interface with memory

the D_bus and the S1_bus are multiplexed and the datapath has only two global busses and they are the Address bus and the Data bus. The external interface to the other Processing Elements (PEs) are provided by the same global address and data busses.

The registers are implemented using basic D flip-flops. See Figure 4-2 for the implementation of a single register. These are parallel load and parallel read. When the load input is 1 (logic high state), the I inputs are transferred to the register on the next clock pulse. When the load input is 0 (logic low state), the load inputs are inhibited and the D flip-flops are reloaded with their present value, thus maintaining the content of the register. This is necessary because a D flip-flop does not have a "no-change" input condition.

The other signals are the enable and the clear signal. The clear signal resets the register to zero state. This signal can be tied to a reset pin. The enable signal is used to drive a tri-state buffer whose outputs are tied to the source bus. This enables data read from a register. See Figure 4-3.

The lowest level is the logic level. The first task was to construct D flip-flops and this was done using the primitive logic level constructs. The D flip-flops were clubbed together with the other logic circuitry to obtain one 32-bit parallel read/load register. This now becomes a higher level. To create a bunch of registers now, all we have to do is call the register module using a for loop that creates as many registers as is necessary.



Signal	Status	Operation
Load	High	Data is transferred from destination bus to register
	Low	No Data transfer
Enable0 & Enable1	High	Data is transferred from register to Source Bus
	Low	No Data transfer
Reset	High	Normal operation
	Low	Register reset to Zero

Figure 4-2. Register with Parallel Load/Read using D-flipflop.

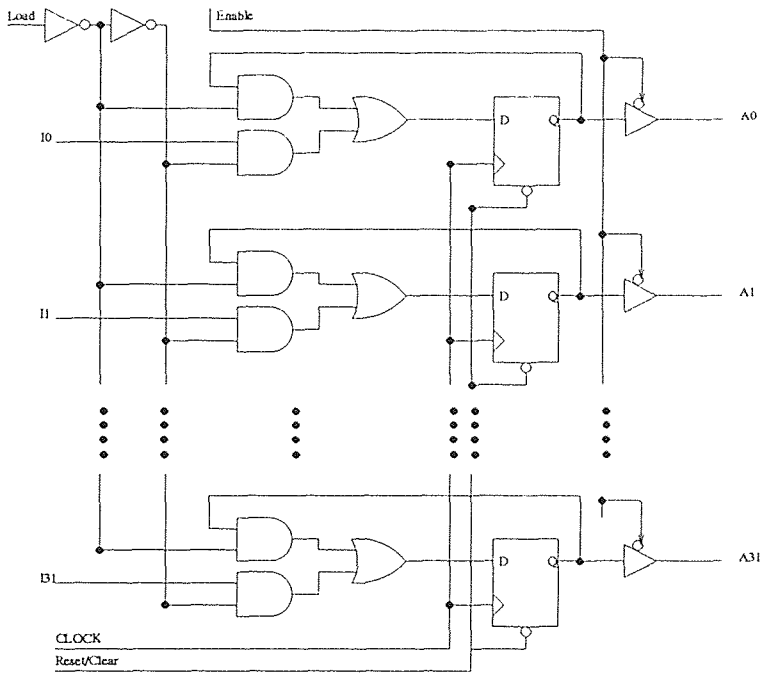


Figure 4-3. 32 Bit Register with Parallel Load/Read.

4.2.2 Bussing Systems

The initial consideration to be given when designing busses is how fast should data transfer take place. It also depends on how much a designer wishes to pay for higher speed logic circuitry. The speed of data transfer within the internal data bus is related directly to the number of bits that are transferred in parallel. Since full word transfers take place on the ARC, maximum tranfer rate is only dependent on the propagation delay of the bus. The ARC uses a multiple bus structure. It is to be noted that within the datapath the busses are unidirectional. To create them structurally and implement them using realistic propagation delays, the busses have been modeled as shown in Figure 4-4.

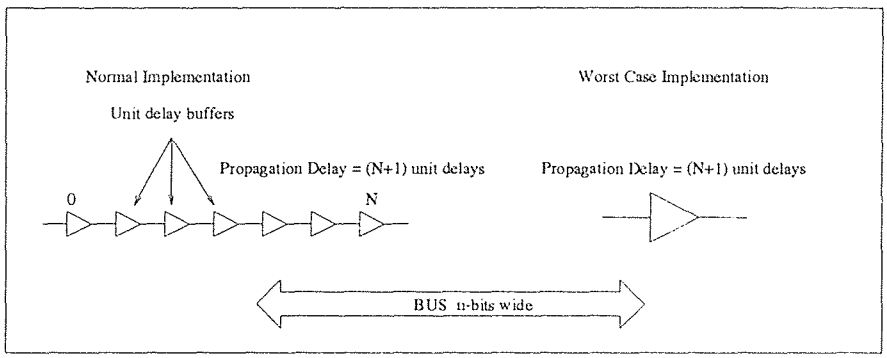


Figure 4-4. Bus Implementation within the Datapath.

Observe Figure 4-4, and it can be seen that in case of implementing a bus, the propagation delay is of utmost importance. In any normal situation of implementing a bus, the propagation delays would differ from point to point on the bus. This essentially means that the point at which one register transfers data onto a bus is different from the point at which another register

tansfers data onto the bus. In ICs this difference in the distance between, one point of transfer to another will constitute different delays.

In the ARC the busses have been modeled with worst case delay. This ensures that data injected to the bus at any point along the bus takes the maximum delay possible. This maximum delay can be changed to any arbitrary number of unit delays. This helps model the structure with a worst case delay.

At the external interface of the datapath to the memory components, the S1_bus and the D_bus are both multiplexed using bi-directional tri-state buffers whose enable signal signify a read/write operation. See Figure 4-5 for details of the interface circuit. If the control signal is logic low then a write operation occurs and a read operation occurs if the control signal is logic high.

The remaining issues to be dealt with are the control signal lines. There are a lot of control signals and these basically perform the following operations: Drives data from a register to the source bus. Drives data from ALU output buffer to the Data bus. Also loads registers with data from the Data bus. The control lines are single unidirectional buffered busses, which must be designed for high speed operation. The effect of the propagation delay in the control line should not affect the overall delay to a very large extent.

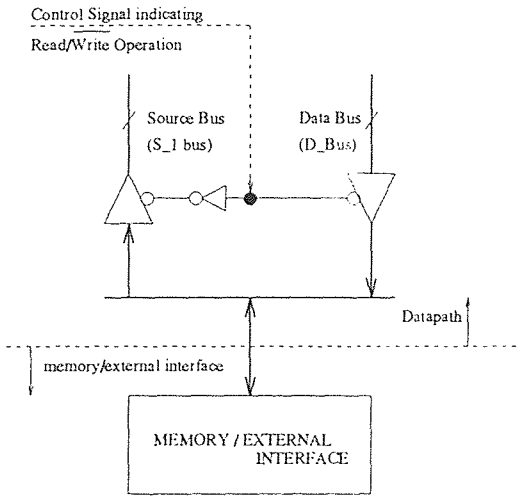


Figure 4-5. Interface/multiplexing of Busses in the Datapath.

4.2.3 Arithmetic and Logic Unit

A ripple carry adder has been used to implement the ALU of the ARC. The data stabilizes after about 98 unit delays at the output. This means that the propagation delay in the adder is 98 unit delays. A faster implementation would be to use a carry look ahead adder. The carry look ahead adder has a higher gate count but is faster. See Figure 4-6 for a description of the ALU of the ARC.

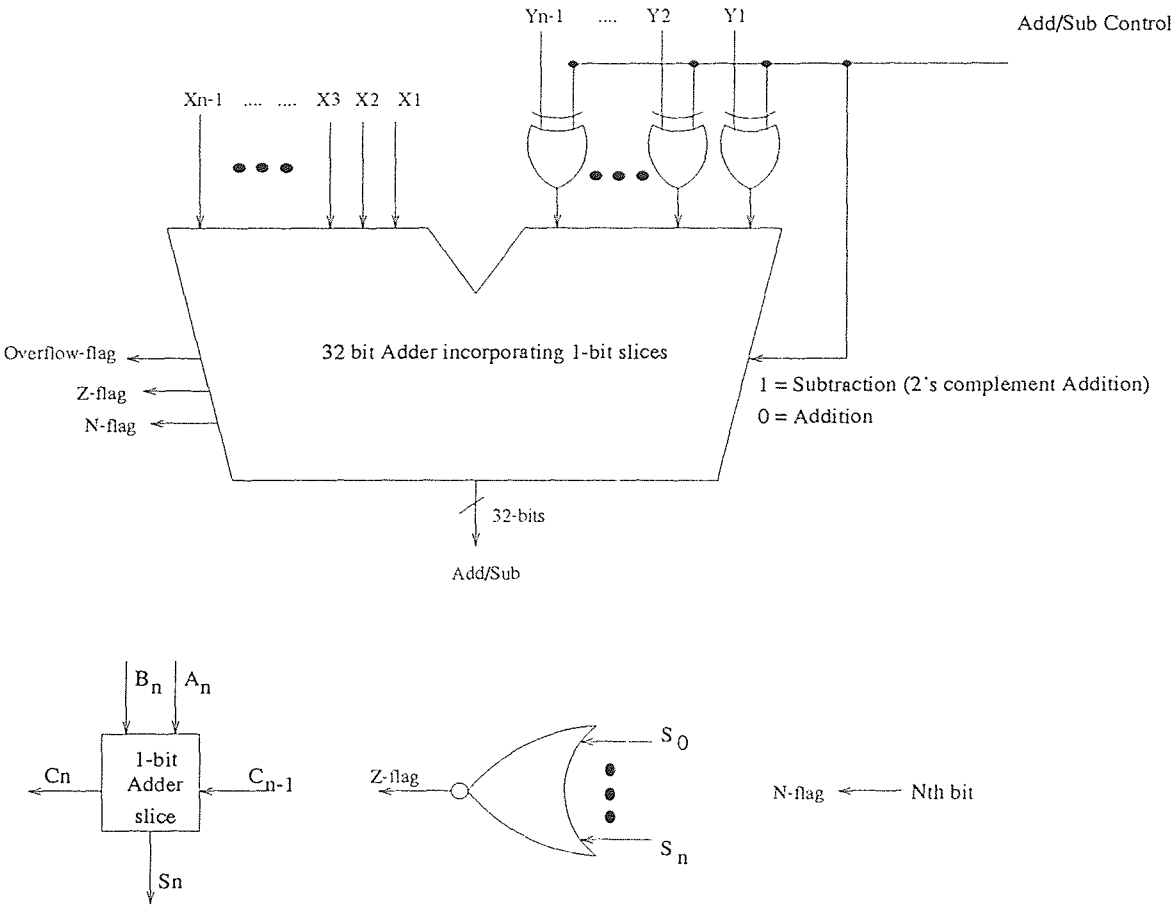


Figure 4-6. Arithmetic and Logic Unit.

4.3 Design of Control Unit

The control unit for the ARC, introduced earlier is based on simulating the instructions individually. Most processors use microcode for their control units. But in this situation, hardwired logic implementation of the control has been proposed. The number of T-states taken by the instructions were discussed earlier in Table 3-2. Some instructions have repeated states depending on the busy-bit. Some may avoid a particular state depending on the logic level of a certain bit.

The state table and diagram for the main control unit is shown in Table 4-1. The control inputs are the ones specified in the input condition column. The state diagram shows the transitions from one state to another.

Present State		Next State		Input Conditions	Multiplexer inputs	
G1	G2	G1	G2		Mux1	Mux2
0	0	0	0	$\overline{S}$	0	S
0	0	0	1	S		
0	1	1	0	—	1	0
1	0	0	1	$\overline{IR\#0\text{-}4}$ • IR#5		
1	0	0	0	$\overline{IR\#0\text{-}4}$ • IR#5		$\overline{IR\#0\text{-}4}$ • IR#5
1	0	1	1	IR#0-4	IR#0-4	+ IR#0-4
1	1	1	1	$\overline{FIN}$	$\overline{FIN}$	$\overline{FIN}$ + FIN
1	1	0	1	FIN		

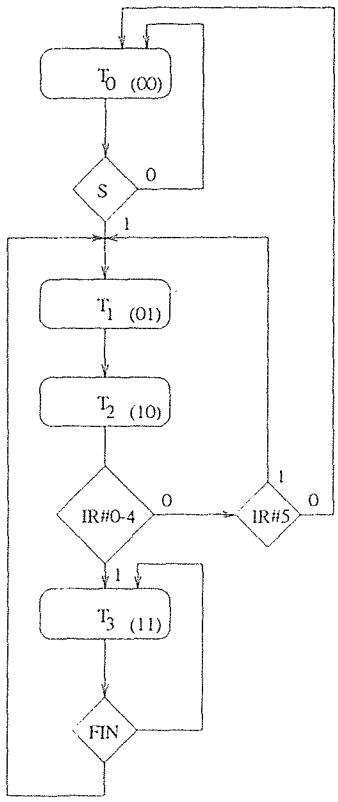


Figure 4-7. State Table and Diagram of the Main Control Unit.

This state table diagram yields the circuit diagram in Figure 4-7. The circuit produces the T-states, T_0, T_1, T_2, T_3. The 'S' control input is the start signal to the processor. The control unit is in state T_0 until the 'S' signal goes high. When this happens, the unit changes state on the next clock pulse. It jumps to T_1 state. It remains in T_1 state for one clock cycle and then changes to T_2 state in the next clock cycle. Now if the instruction demands that it go into T_3 state, then the processor goes into T_3 state and

remains in T_3 state until the unit has received the 'FIN' signal. This 'FIN' signal is the finish signal.

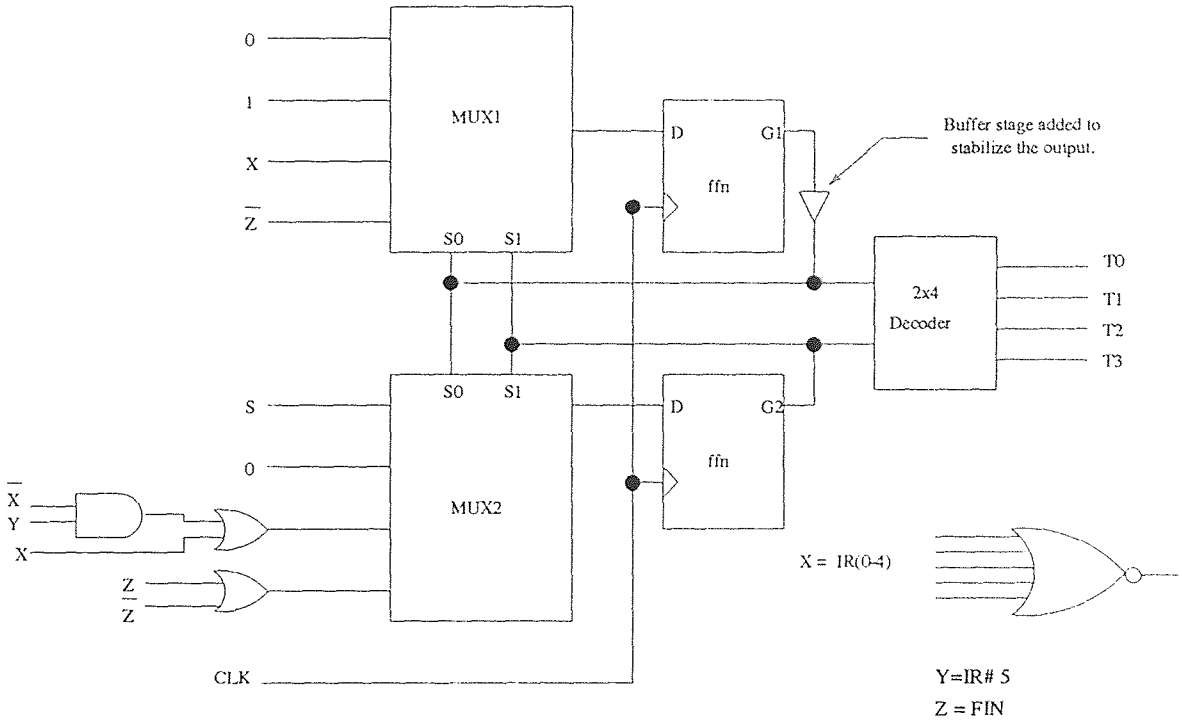


Figure 4-8. Diagram of the Main Control Unit.

The instructions NOP, END, and WAIT are the only instructions that do not require the control unit to go into T_3 state. All other instructions go into T_3 state. The control is transferred over to the next stage of the control unit. The instruction is recognized by a decoder in this case and a particular signal is driven high. This signal starts the individual instruction control units.

These control units are based on the number of T-states that are required to complete the execution of the instruction. Some instructions just follow a certain arbitrary number of T-states, while others depend upon certain condition codes that decides the number of T-states they take to execute. The

T-state diagrams of all the instructions that were implemented are included in APPENDIX B.

We shall have to take a particular instruction to show how the implementation works. We shall take the ADD/SUB instruction. This instruction takes about 6 more T-states after it has gained control from the previous stage. To traverse through these 6 T-states, we use an 8-bit counter that feeds a 3-8 decoder which generates the necessary T-states with two unused states. See Figure 4-8 for block diagram description of the control unit. A single control unit can be used for both the ADD and the SUB instruction due to the fact that they are almost identical.

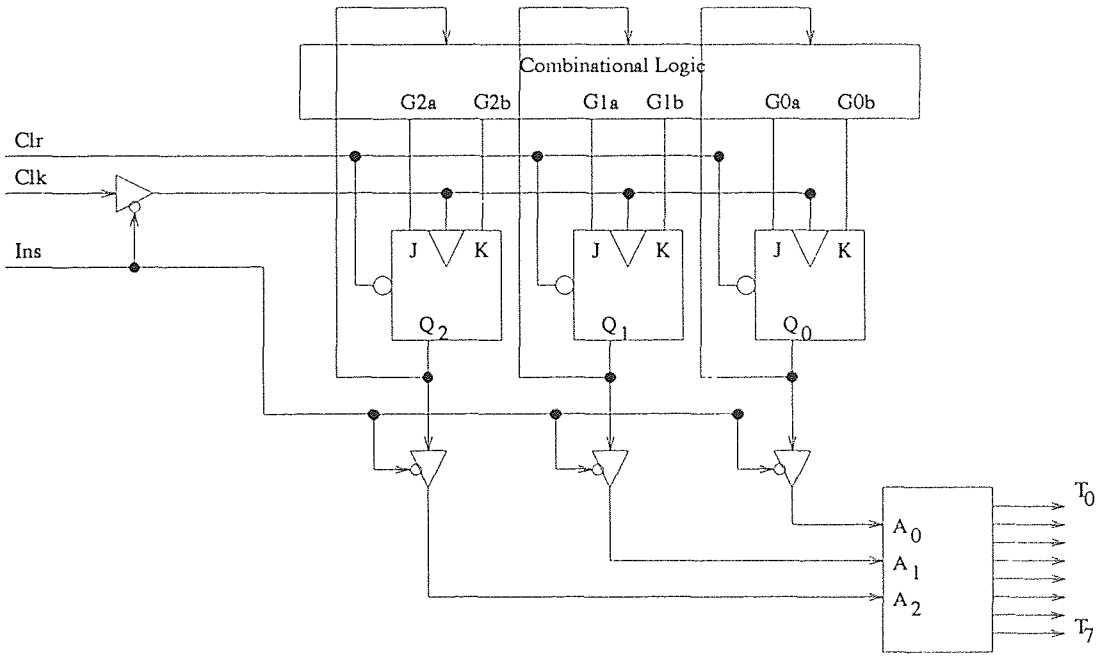


Figure 4-9. Control Subsystem for ADD/SUB Instruction.

The control subsystem which includes the counter uses a JK flip-flop for its counting operation. The JK flip-flops operate in the master-slave mode

and are edge-triggered. They are driven by a combinational logic block. The design of the elements of this combinational logic block was done by using state flow diagrams and Karnaugh map reduction.

The clock signal is fed only if the 'Ins' signal occurs. When this occurs, the counter is reset to 0 by the 'CLR' signal which signifies clear. This starts the counter operation and at the same time the counter output is fed to the decoder. The decoder gives the necessary T-states. These signals are the ones that connect to the datapath unit control lines. They determine the flow of data within the datapath or the processor. See Figure 4-9 for the control signals produced by the subsystem.

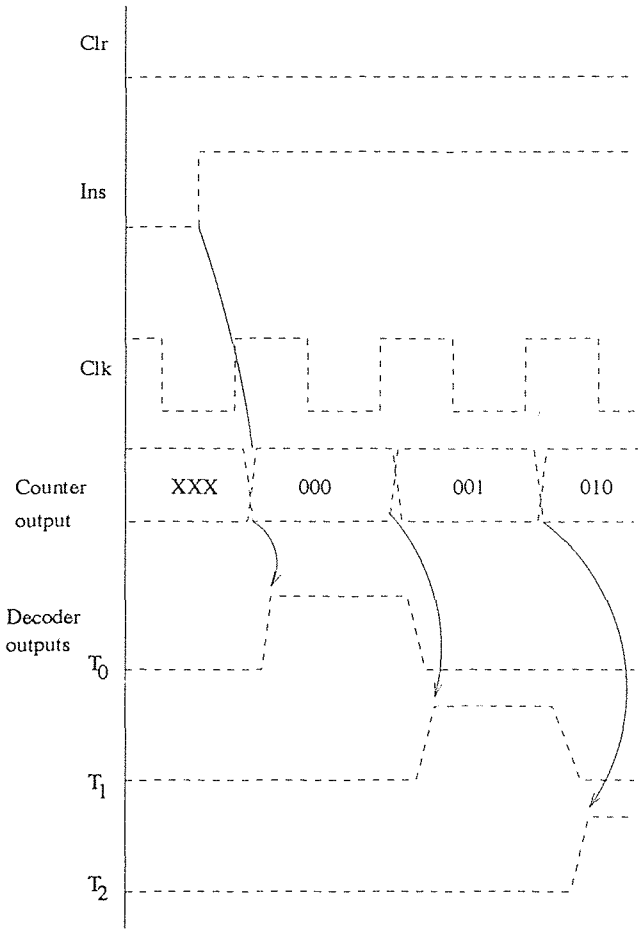


Figure 4-10. Control Signals Produced by the Subsystem.

Every state line is logically anded with a $4-\phi$ clock to break up the T-state pulse into 4 different signal components. Each of these signal components have a unique function. See Figure 4-10 for details. The internal operations for two states has been shown. Most other T-state operations follow in the same manner.

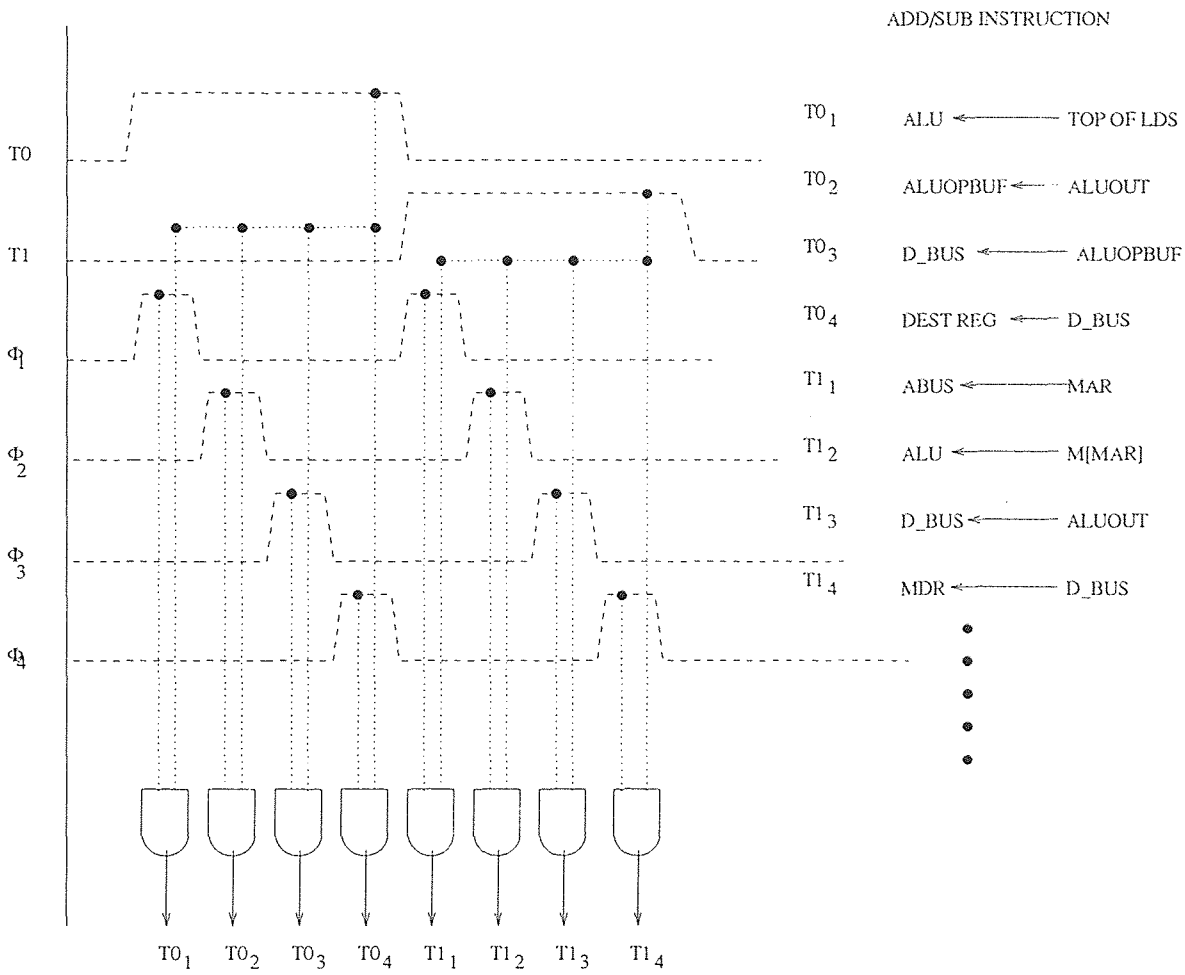


Figure 4-11. T-state Internal Operations.

4.4 Design of Memory

The memory can be modeled in many ways in HDL. It could be incorporated as a large bank of registers. This is most practical when the models are implemented structurally in HDL. One of the other ways of implementing memory is the behavioral level. but it does not serve the purpose of the simulation here. Any memory implementation also has associated decoding circuitry. The size of this decoding circuitry depends on the size of the memory

implementation. Since in the ARC the data-transfer through busses is being modeled with a worst case delay, we are assuming that the time for a memory access is the same immaterial of the distance it has to traverse.

Based on the register implementation of memory, the register module has to be used to build the memory. The memory module consists of a databus, an address bus, and a read/write signal. It also consists of decoding logic which uses the address bus as input to point to a particular location in memory.

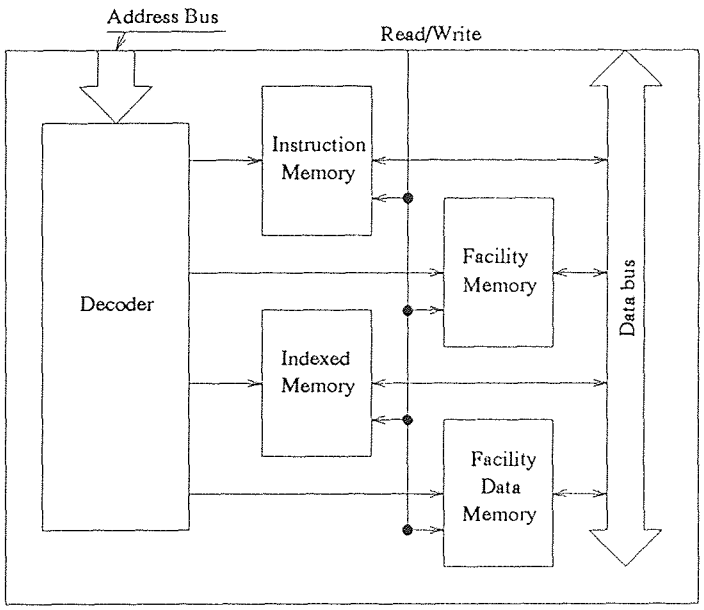


Figure 4-12. Architecture of the Memory.

4.5 Hierarchy in the Design Environment

The basic steps in the design process are to create the lower level modules. As we all know the any digital architecture is built from a collection of basic gates. These basically are Inverters, And gates, Or gates, Nand and Nor

gates etc. The advantage in VHDL is that these gates do not have to be built everytime an instance of it is called. The basic gates can be called in as primitive logic. Once a module is built from these basic gates, then that module can be called over and over again, while maintaining the same characteristics at the higher level.

Figure 4-13 shows the basic level of hierarchy involved in the design process. The lower level modules are used to build the next higher level modules and so on. This helps in faster simulation and higher accuracy.

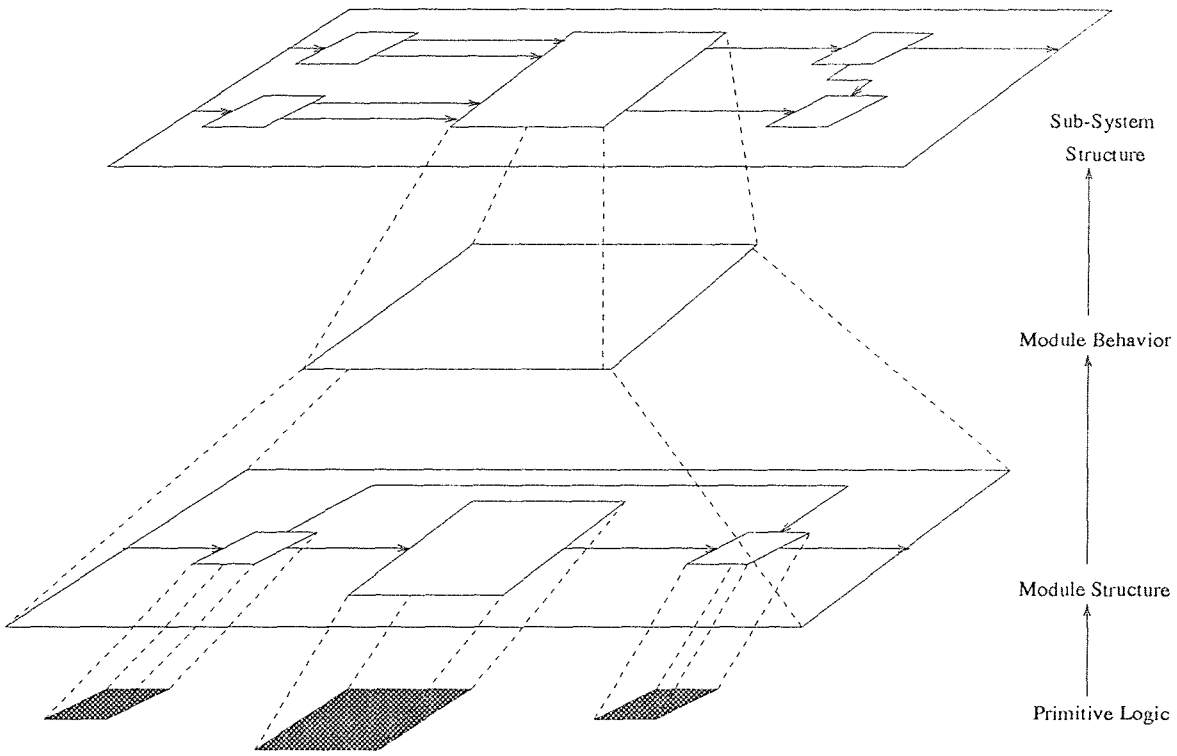


Figure 4-13. Design Hierarchy.

4.5.1 Timing Considerations in a Hierarchial Environment

As the design evolves around building higher level blocks from lower level modules, it is to be noted that the timing characteristics are passed on to the higher level by the lower level module. If the lower level modules are assumed to be of zero-delay (i.e. 0_{NS} for both the rise-time and the fall-time of the input and output), then the Logic simulators give the modules a unit delay when used at a higher level. It is also the same when we assume unit delays for every lower level module.

In a particular example, the positive-edge-triggered D-flip-flop with a clear pin uses 3 nand gates with 3 inputs and 3 nand gates with 2 inputs. Each of the lower level modules are instantiated with a unit delay at the higher level module. The propagation delays are added up when the D-flip-flop module is simulated. This gives us an understanding of the approximate delays to expect when this module is used at a higher level. When these modules are instantiated and used, the delay of the lower modules is summed up to arrive at the propagation delays for the higher level modules. If the modules are netted in series then the propagation delays are automatically added up at simulation time. If they are netted such that they are in parallel, then the propagation delays are not summed up. This can be observed in Figure 4-14 and Figure 4-15.

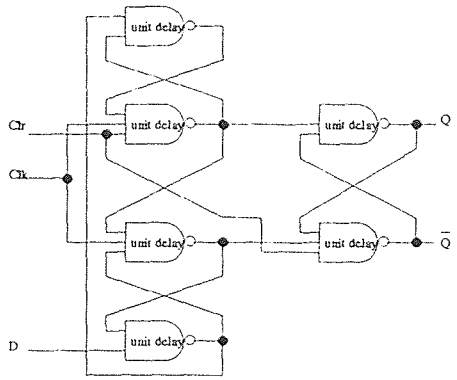


Figure 4-14. Propagation Delay in a Module.

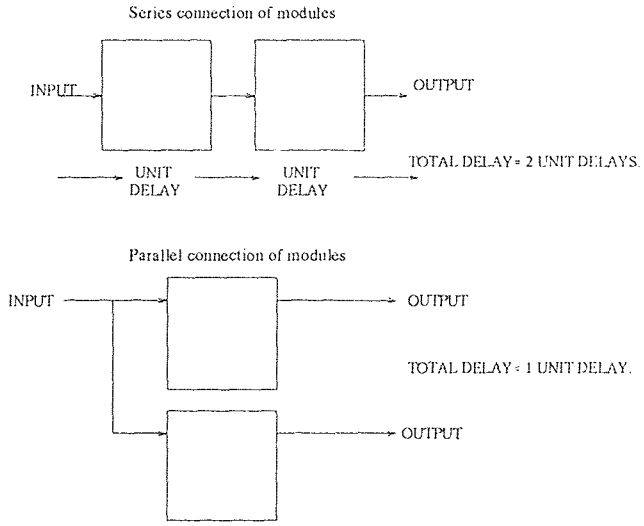


Figure 4-15. Propagation Delay in Series and Parallel Modules.

This propagation delay through the modules give us an approximation of when the output stabilizes. Knowing such parameters, the design of control units to generate control signals becomes much easier. The control signals are fed to the datapath and other units only after the data has stabilized. If all lower level modules are simulated and their delay parameters are known, then the design of the control unit is rendered easier.

The concept of unit delay at the lowest level is a big advantage in the design of digital systems using HDL. Certain parts of a circuit may not behave as expected due to incorrect data transfer. These faults can be verified by adding delays in the circuit by including buffers with specific delays. These buffers act as delay elements and create the necessary delay, which accounts for output stability.

CHAPTER 5

SIMULATION RESULTS AND DISCUSSION

This chapter provides a step by step discussion of the simulation focusing on particular instances involving transfer of data. It also discusses the problems encountered in the architecture of the different elements of the processor.

The architecture of the ARC has been broken down to be able to comprehend the simulation results. Every instruction, involves the use of the elements in the datapath, and memory. The data is transferred from one point to another. Our concern is to look at the lowest level. The simulation of any instruction involves transfer of data at the lower level. Some instances of data transfer are: from registers to busses, from busses to ALU, from ALU to output buffers, from memory to datapath, and bit-data transfer from datapath to control. The other kind of data-transfer is that of the control circuitry. The control circuit produces signals which drive certain elements within the datapath and memory. All these are discussed individually.

5.1 Data Handling by Registers

The registers in the datapath are constructed from basic D-flip-flops. This have been shown in Figure 4-2 and Figure 4-3. The input to a particular register is applied at the 'I' inputs. The register inputs are driven from the destination bus through a tri-state buffer unit. The buffer unit passes the

available at the bus to the register only if it is enabled. This enable signal is available as a control line. It has to be noted that all the enable signals of the input buffers are control lines. The output of the register is also fed-back to the input so as to maintain the internal data by refreshing it every clock cycle if no input is present. See Figure 5-1 for input and output data of a register.

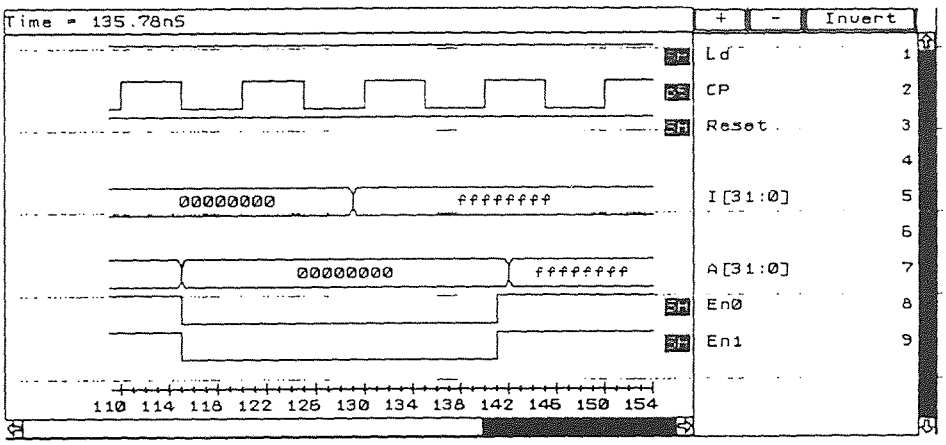


Figure 5-1. Data Handling by Registers.

5.2 Data Transfer to Busses

The data transfer to busses are either from the register set or from memory. In case of transfer from the register set, the register outputs are enabled and the data at the outputs of the registers are gated to the bus. In case of data transfer from memory, the difference is in the amount of time taken for the

data to propagate to the ALU, which is usually the destination, is higher.

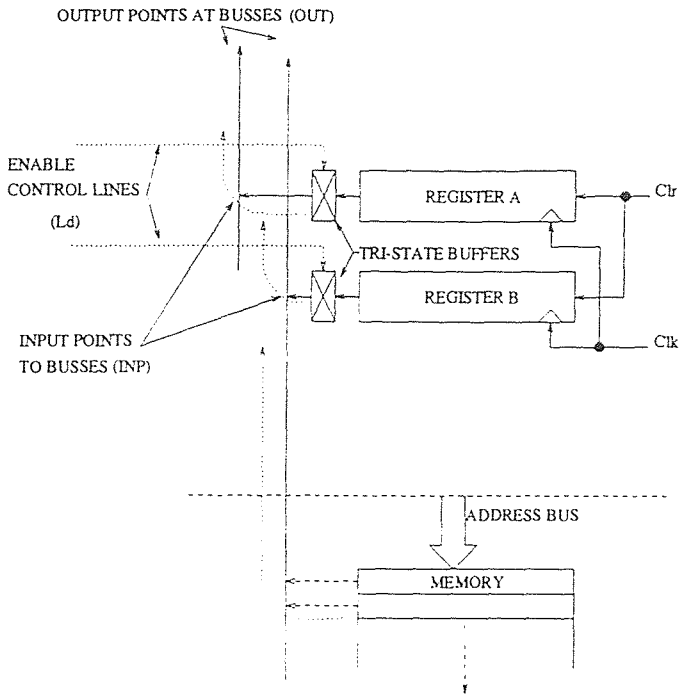


Figure 5-2. Simulated Structure for Data Transfer to Busses.

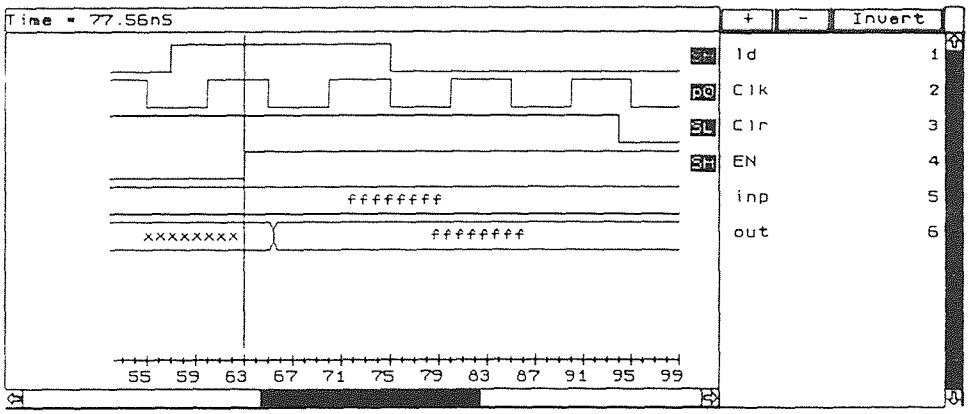


Figure 5-3. Data Transfer to Busses.

Figure 5-2 gives on overview of data transfer to busses. The transfer of data to busses is achieved when a register's output terminals are enabled. All registers are connected to the bus via a tri-state buffer. When this tri-state buffer is enabled, the data in the register is transposed onto the databus. Whatever previous data that existed on the bus is overwritten.

When simulating M-HDL modules on an Lsim simulator, it is necessary that each module be declared with input and output ports. Therefore for the purpose of simulation input and output ports have been declared and the same is shown in Figure 5-3. Arbitrary names have been given to the points at which the data has been probed.

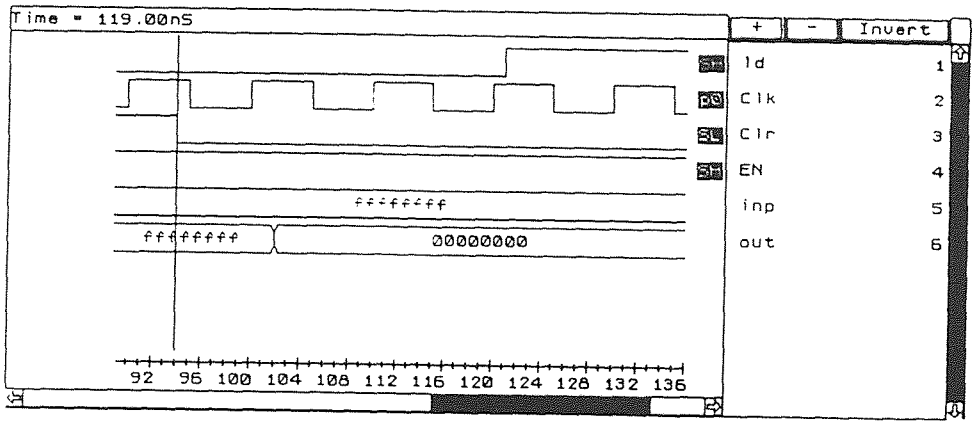


Figure 5-4. Effect of a Clear Operation.

Also observe in Figure 5-4 where the effect of a clear register operation is reflected on the bus due to the fact that the particular register has been enabled. In the case of a memory access, the same principle holds except that the propagation time for data transfer is longer. The effect of a clear operation loads the bus with zeroes. This operation is used in computations where the second operand needs to be zero.

5.3 Data Propagation in Busses

The structure of the bus, already introduced in Chapter 4, see Figure 4-4, is modeled as a delay buffer. In actual VLSI implementation of a bus, the inherent capacitance and resistance offered by the metal line give rise to propagation delays. No primitive instances of a bus are available to model a bus in HDL. The best modeling method would be to simulate the bus as a series of delay buffers. The bus could also be modeled using a single delay buffer with a known delay parameter.

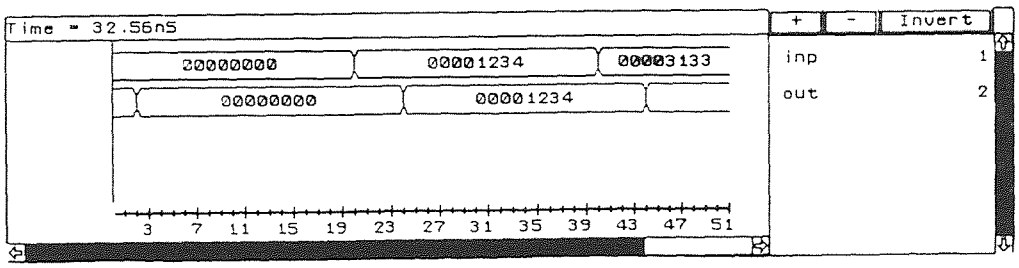


Figure 5-5. Propagation Delay in Bussing Systems.

5.4 Operation of The ALU

The Arithmetic and Logic Unit in the ARC is a basic ripple carry adder. It also has a provision for 2's complement addition (subtraction). The ARC does not support on chip floating point arithmetic. For such computations a co-processor would have to be used. The ALU shown in Figure 4-6 is modeled for 32 bit capacity. It has an overflow bit which is the carryout from the last stage.

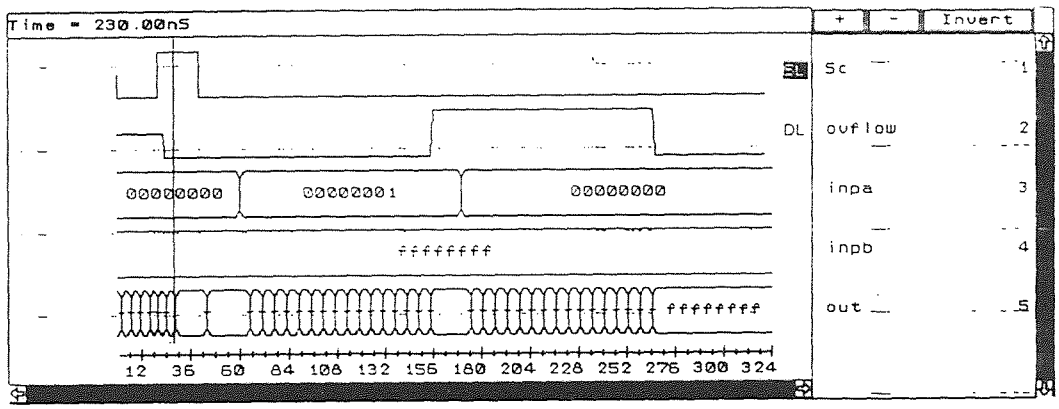


Figure 5-6. Operation of the ALU.

The ALU is a concatenation of 32 single bit adders. The carry is propagated from the least significant bit to the most significant bit. The ALU in its present implementation takes about 100 delay units to stabilize at the output. This delay is because of the time taken for the carry to propagate

through the 32 stages of the ALU. The carry out from the 32nd stage is the overflow indicator. This signal has been made available at the outside and could be used to generate interrupts for future design implementations.

It can be seen in Figure 5-6 that data stabilizes at the output of the ALU approximately about 100 unit delays after input is applied. The control signals to drive the ALU output buffer should therefore be delayed by an amount greater than the propagation delay of the ALU. This is a slow implementation of the ALU. A much faster implementation of the ALU could be achieved by using the carry look-ahead principle. But this has a very large gate count. This principle could be applied in future imlementations of the ALU.

Figure 5-7 illustrates a part of the datapath unit. This when expanded to include more registers gives us almost the entire datapath. In this particular case two registers have been modeled together with the source and destination busses and the ALU to provide an idea of the working of the datapath unit. There are control lines which enable tri-state buffers which gate data onto a bus from the registers or vice-versa.

In this particular example the registers are preloaded with certain data, and they are then gated to the source bus. This in turn is fed to the ALU which performs the operation as indicated by its control signal and the resulting output is stored in an ALU output buffer register. This data is then fed to the destination bus from where it is gated into one of the registers. In

totality this operation involves all of the other operations discussed above.

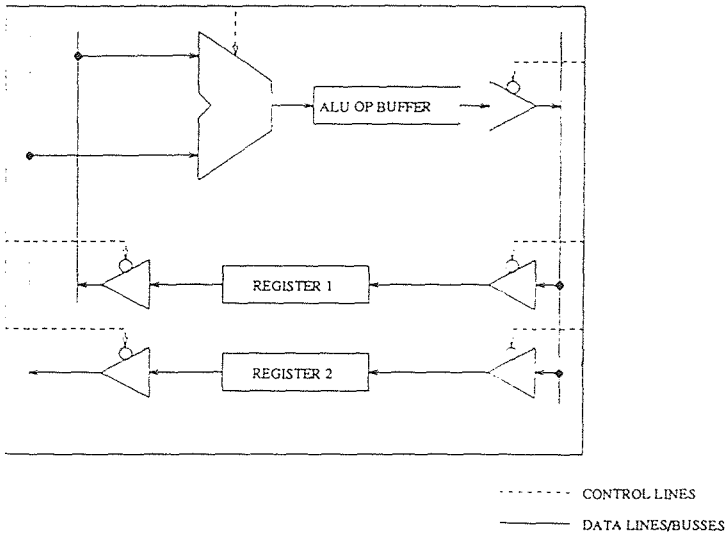


Figure 5-7. Simulated Structure of the Datapath

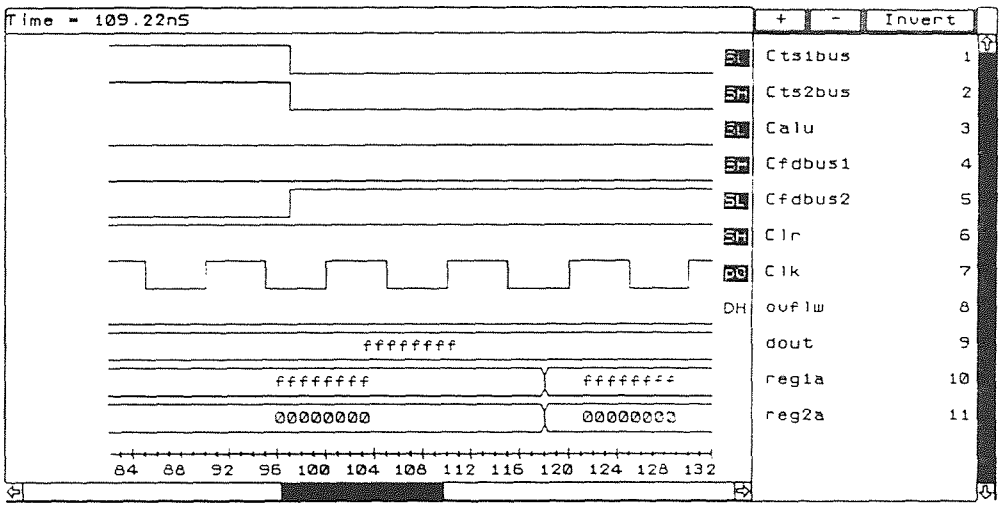


Figure 5-8. Simulated Operation of the Datapath

5.5 Memory Read/Write Operation

A memory fetch operation usually involves the transfer of data to the to the Memory Data Register (MDR) from the memory location pointed to by the Memory Address Register (MAR). The MAR is gated to the address bus,

which in turn feeds the address decoder. This address decoder points to a location and either enables the memory register output or the input for a read or write operation respectively. The above procedure involves all of the above mentioned operations. The MAR is loaded with the contents of the destination bus which has the address of the location in memory.

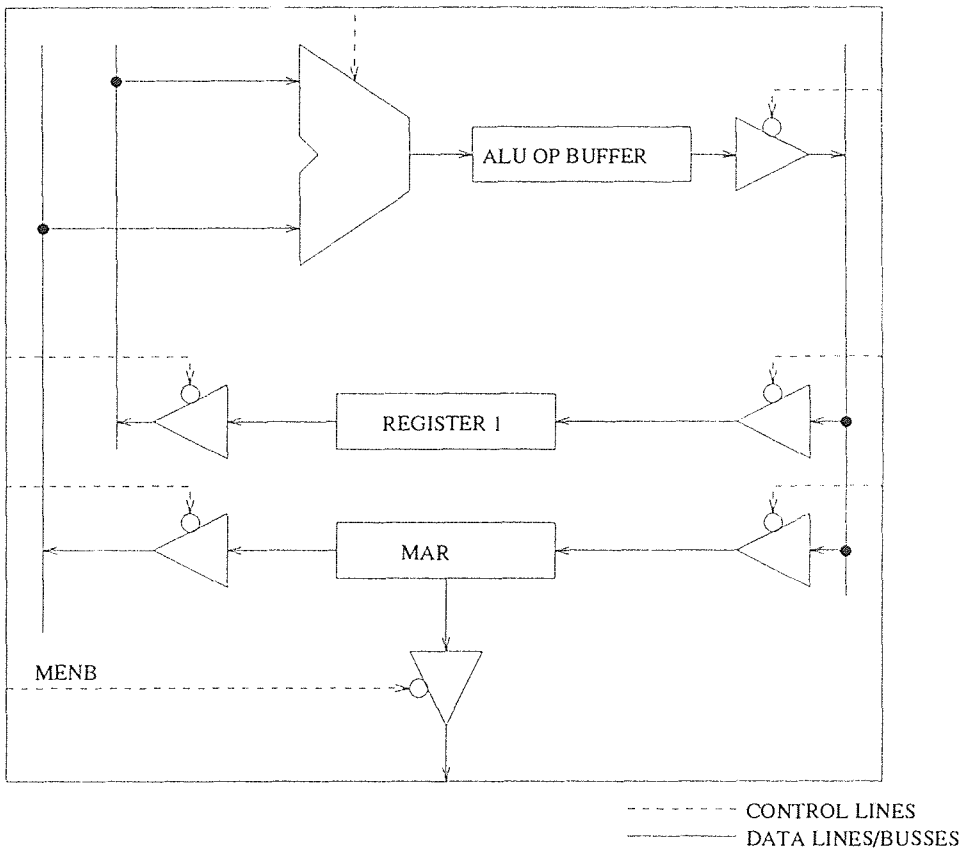


Figure 5-9. Reference Circuit for Memory Access

The MAR in addition to being connected to one of the source busses is also connected to the address bus external to the datapath. This interface when enabled points to a location in memory. Since the memory in the ARC is specifically split up into four categories, the first 2-bits could be used to

reference the particular memory component. See Figure 5-8 for a load operation on the MAR. Also refer to Figure 5-7 where one of the registers may be assumed as the MAR.

The Maddr component in Figure 5-8 is the memory address bus. The Reg2a component is the MAR. The Menb component is the memory enable signal.

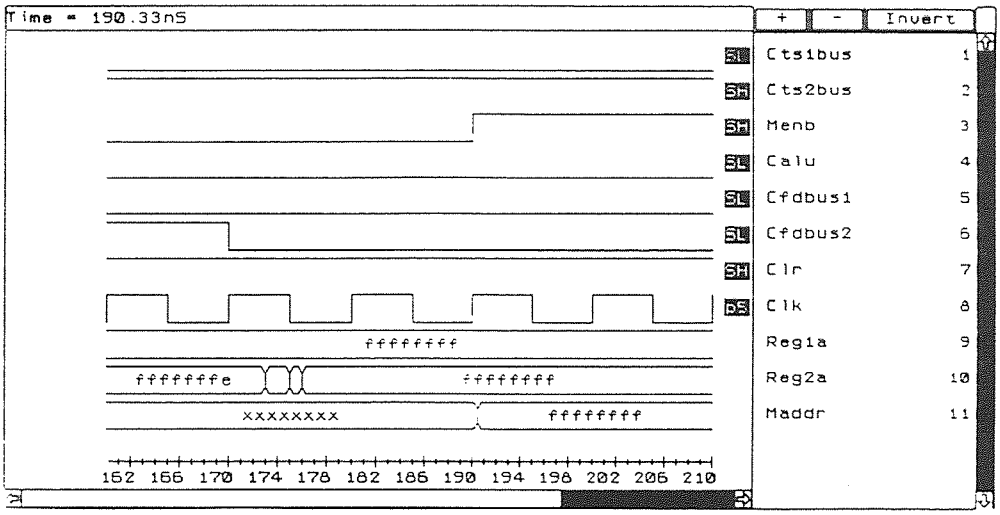


Figure 5-10. Simulated Operation of a Memory Access

5.6 Bit Data Access

There is a need to access some bits of information from within the datapath. One of them, the Busy-Bit is the most significant bit (MSB) of data stored in memory. This bit is accessed only after the data from a memory access has

reached the datapath. If this bit signifies the busy state, then the procedure of memory access is repeated. This goes on until the Busy-Bit is in the not busy state. The other bit data, which is the control-flag is needed with instructions like CLRZ and CLRN.

If any of the instructions need to execute with a check on the Busy-bit, then the following happens. The address of a location is loaded into the MAR, and the contents of the location pointed to by the MAR are brought to the MDR. The most significant bit of the MDR can be led to the outside of the datapath to check its value. This is also the case in the control flag bit. The other point of discussion is that the ARC does not support bit data manipulation in the present version of the instruction set.

5.7 Simulation of Control Unit Components

The control unit previously described in Figure 4-8, provides the main T-state signals. It is designed to work through states T_0 to T_2, immaterial of what the instruction might be. It is during these two T-states that operations like loading of instruction register from memory, incementing the program counter take place. When the main control unit was first designed, some instabilities at the output were noticed. This was due to improper swiching. Extra delay parameters have been added an ensure smoother transition. See Figure 5-11.

The control sub-units are basically constructed from counter and decoder circuits. Figure 5-12 here shows the working of the counter circuit. These counters are built using JK flip-flops. The outputs of these control units drive the inputs of the decoder units.

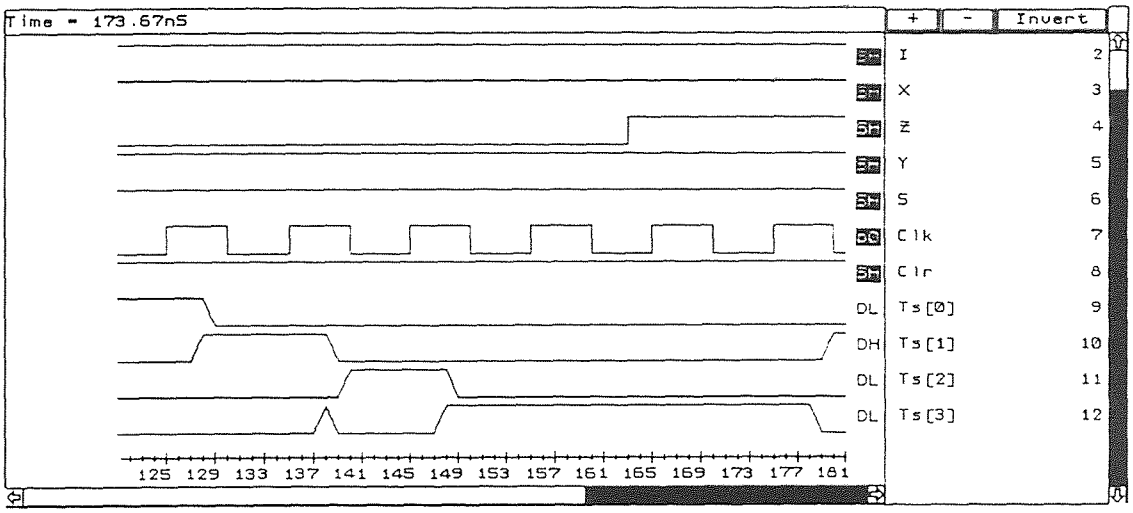


Figure 5-11. Control Unit Simulation

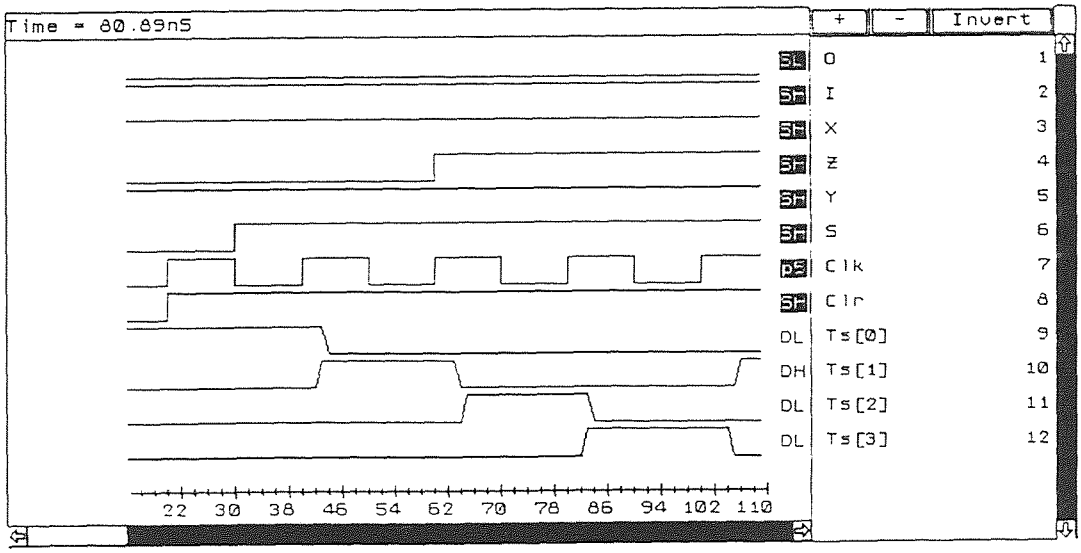


Figure 5-12. Corrected Control Unit Simulation

The corrected module produced the desired results as can be seen in Figure 5-12 and Figure 5-14.

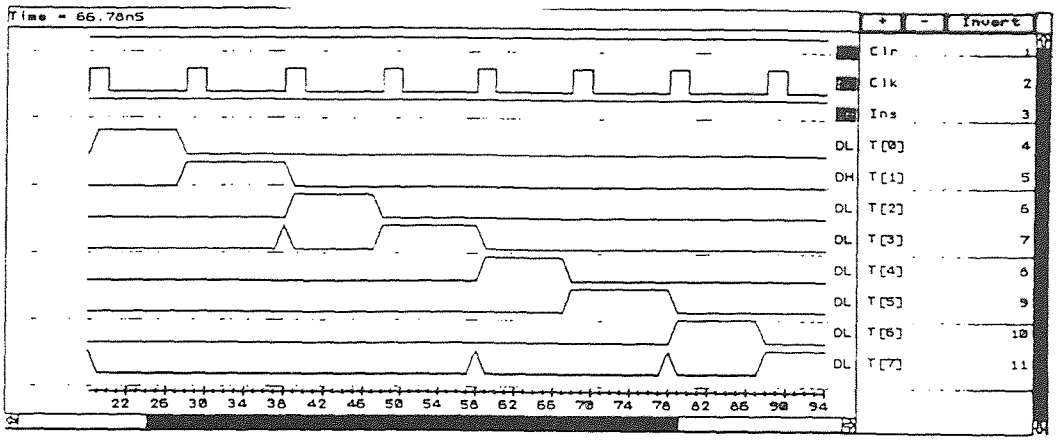


Figure 5-13. Unstable Counter Operation

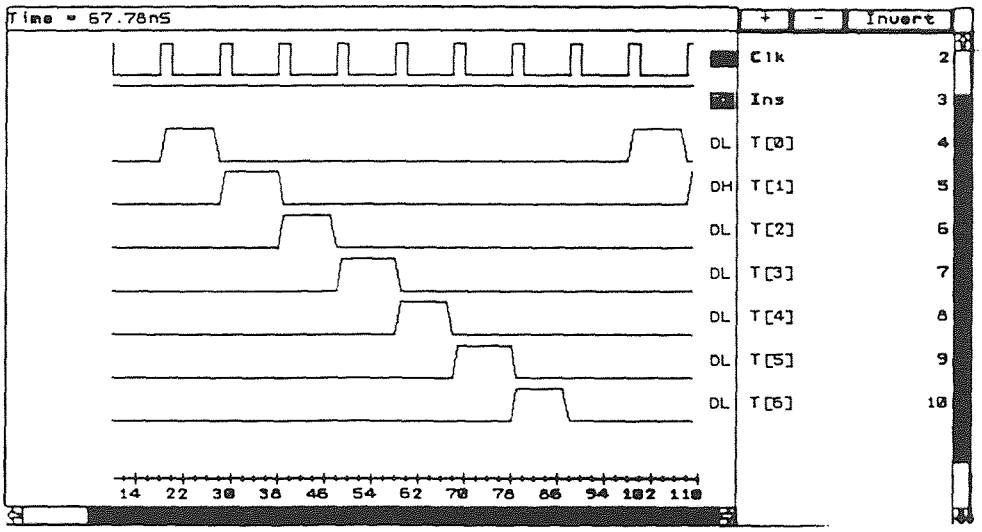


Figure 5-14. Stable Counter Operation

5.8 Sequence of Instruction Execution

Every instruction that executes has a certain number of T-states. It is within these T-states, that certain operations are performed. These operations within the T-states are a combination of the above mentioned operations. The execution of any instruction at the logic level involves operations such as: transfer of data to a bus, an alu operation which may be add or subtract, transfer of data from a bus to a register, gating of data to the memory address bus, receiving data from the memory data bus into the datapath source bus.

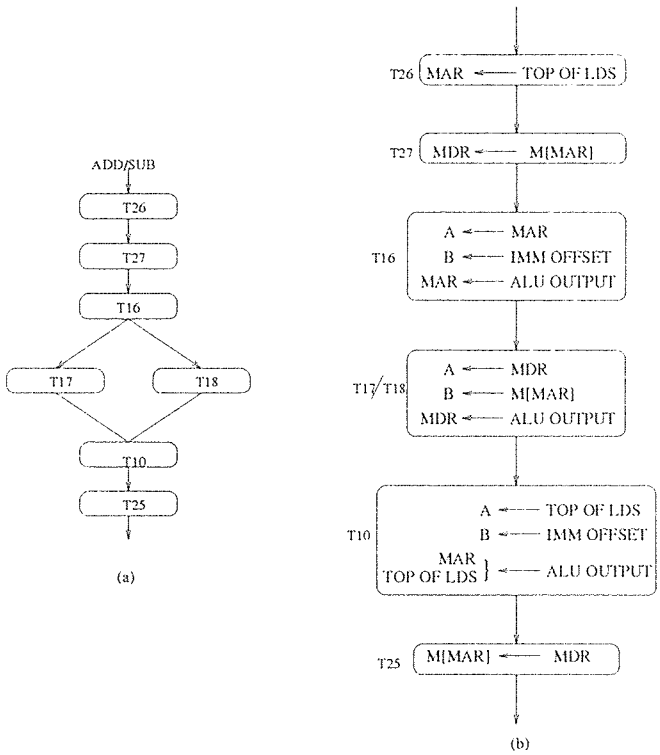


Figure 5-15. Instruction Flow Sequence of ADD/SUB

As we discussed, to understand the execution of a particular instruction at the logic level it is necessary to traverse through the different elements of the ARC. Looking back at Figure 3-4 where we have presented the T-state

diagrams of some of the instructions, we shall constrict our discussion to a particular instruction. All T-state internal operations are all included in APPENDIX B. We shall discuss the instruction with references to simulation results. The T-state internal operations for the ADD/SUB instruction is shown in Figure 5-15.

The operations that occur within the T-states of the ADD/SUB instruction can be seen in Figure 5-15. Figure 5-15a describes the instruction just by specifying the T-states involved. Figure 5-15b gives a more descriptive view of the operations involved. Now each operation occuring within the T-states are a sum of repeated processes discussed above.

Figure 5-16 shows the data flow within a particular T-state. It is to be understood that at the logic level every operation is just a sequential flow of data. As can be seen in Figure 5-16 the first T-state operation of the ADD/SUB instruction is the movement of data from a register to another register. This sequence can be easily comprehended from the figure. It therefore follows that most other T-states follow the same internal operations more or less. The difference may occur if the T-state operation is a memory access operation.

In case of a memory access operation, the difference is in the propagation delay. This is explained in Figure 5-17. The propagation time delays are more in this case and therefore it adds up to the total delay. The T-states T_{27} and T_{25} are examples of such operations. T_{27} is a memory read and T_{25} is a memory write operation. These are the only two T-states which access memory. All the other T-states are just a variation of the T-state discussed in Figure 5-16.

The memory in the ARC has been modeled as a bank of registers. This is the only way memory can be structurally represented. In a behavioral model memory can also be modeled as a read only memory (ROM). To model a ROM in structural representation, it is only necessary to not provide any input lines to the memory bank and to preload the memory bank individually at initialization time.

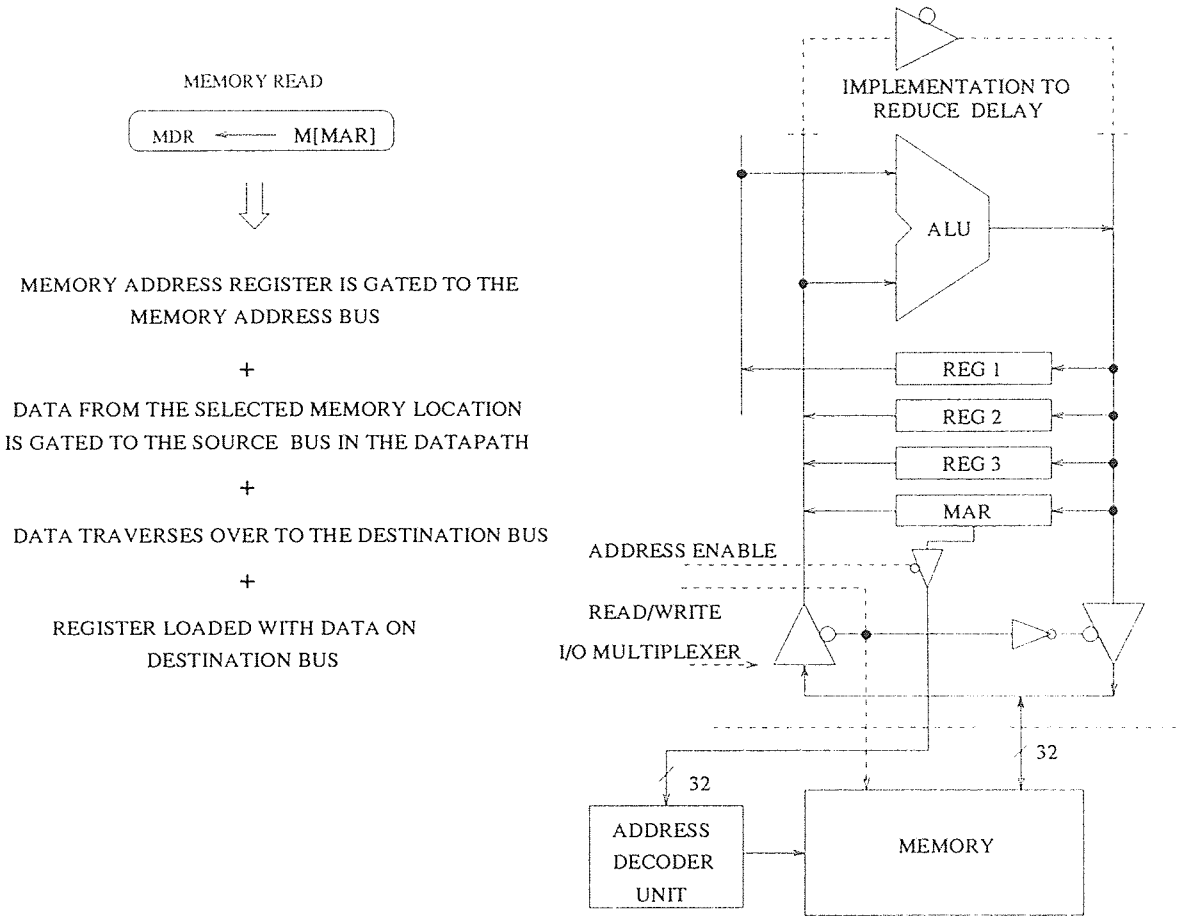


Figure 5-17. Memory Read Operation

The propagation delay could be reduced by avoiding the path through the ALU. For a memory read operation, this path through the ALU could be avoided by gating the data to the destination bus through a tri-state buffer. This path would reduce the delay. In case of a memory write operation which is shown in Figure 5-18, the exact opposite of a memory read operation takes place. The T_{25} State is a memory write operation.

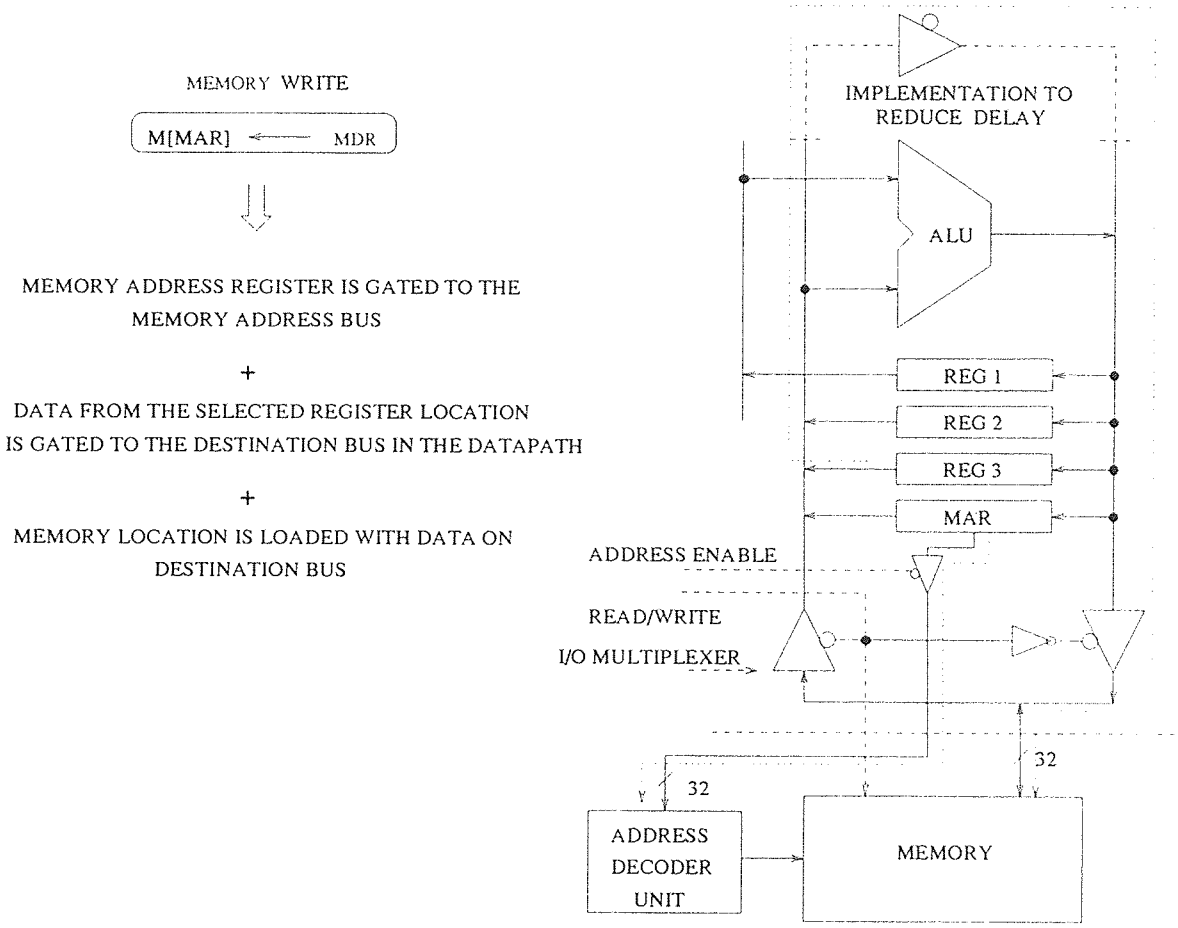


Figure 5-18. Memory Write Operation

5.9 Instruction Execution Time

The time taken for an instruction to execute T_{total} is the sum of the time taken for all the individual T-states. It is therefore necessary to compute beforehand the time taken by every individual T-state. When this has been done, the time taken by every instruction could be stated with the best approximation. The processor is being modeled with the worst case delay parameters. So any timing approximation done here is with respect to the

worst case delay parameters. An approximation of the time taken by each T-state has been provided in Appendix B. Using this data, the time for each instruction can be computed.

In computing the time taken by the individual T-states, we have considered only the delays associated with data transfer. We have also to consider delays in the control circuit, because the delays in the control circuit are not negligible compared to the delays in the datapath and memory components.

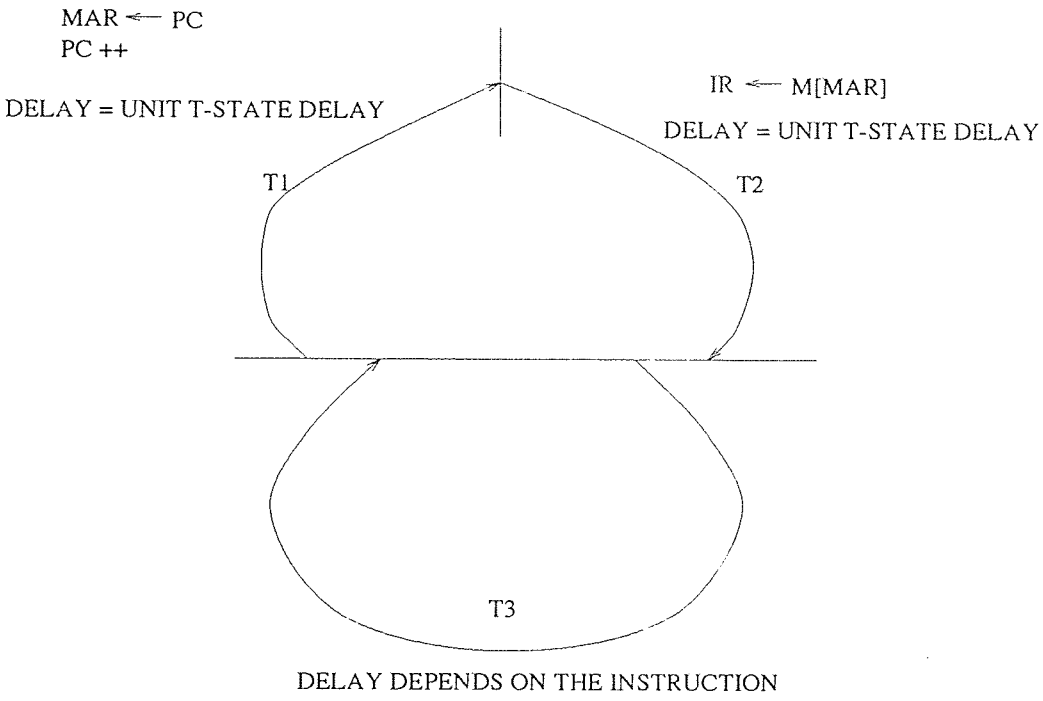


Figure 5-19. Main Timing Diagram

Observe Figure 5-19, where the instruction execution time is graphically represented as to the number of T-state delays involved. The instructions that operate only in the T_1 and T_2, states are the NOP, WAIT, and END. Most

other implemented instructions go into the T_3 state, and the execution time then depends on the number of sub T-states within T_3 state for a particular instruction.

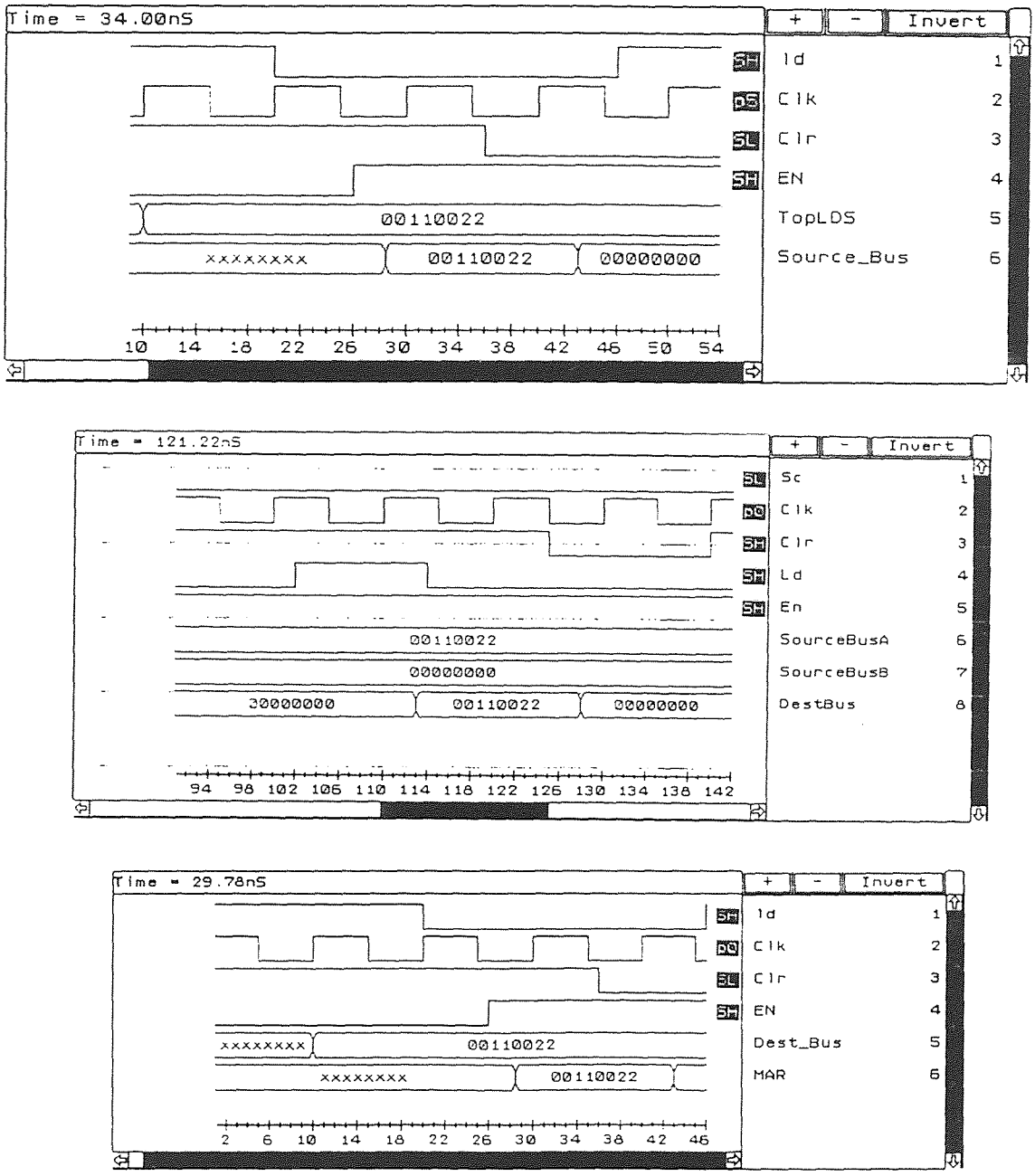


Figure 5-20. Data Transfer Operation

The data transfer operations in some of the T-states is shown in Figure 5-20. This is the T_1 state in the ADD instruction where the top pointer of the local data stack gets transferred to the MAR. The operations involved are: data transfer from register (TOPLDS) to source bus, data transfer from source bus to destination bus and the data transfer from destination bus to the MAR. It is also during the same T-state that the program counter (PC) gets incremented. This has been shown in Figure 5-21. Also refer to the block diagram of the datapath for this discussion.

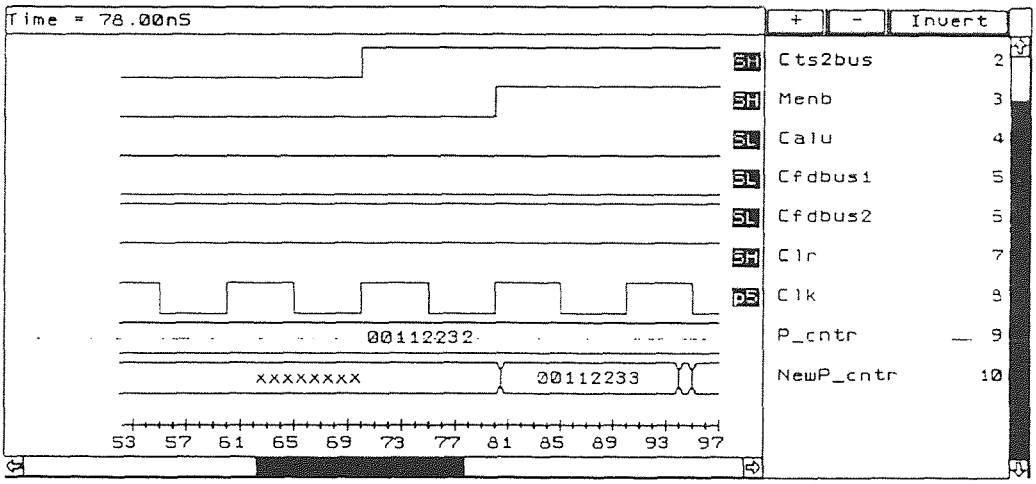


Figure 5-21. Incrementation of Program Counter

Therefore the execution of an instruction at logic level amounts to data transfer through the different elements of the processor. It is to be

understood that for an instruction to execute, the above mentioned operations get repeated as many times as the instruction demands. The only difference would be in the fact that the registers involved at every step of the operation would be different.

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH

6.1 Conclusions

A start has been made on the architecture design of ARC processor. HDL, a very powerful tool for Integrated Circuit design, has been used to build and simulate the different elements of the processor. The flow of data through the different elements of the processor have been observed and the timing analysis has been done. The delays involved in the propagation of data have been studied. The stability of the control circuits have been achieved by adding delay elements.

This work provides a stepping stone for future work towards the completion of the ARC processor. The circuit modules built so far could be further extended to include all the instructions. Work could be started on building the circuit schematics using an interactive graphics editor. This could also be supplemented by floorplanning of the architecture. Once the major components within the architecture have been built, an autoroute package for layout placing and route packing could be employed.

The completion of the ARC would also require a Network Control Unit (NCU). This NCU would assist in the control of operations, particularly with respect to the CALL instructions of the ARC. The CALL instructions are basically remote procedure calls, where multiple ARC processors would be involved. The NCU would have to be able to keep

track of other processors in the same environment. It should be more of an Asynchronous Interface Unit. This NCU of the ARC would have communicate with its counterparts on the other ARCs.

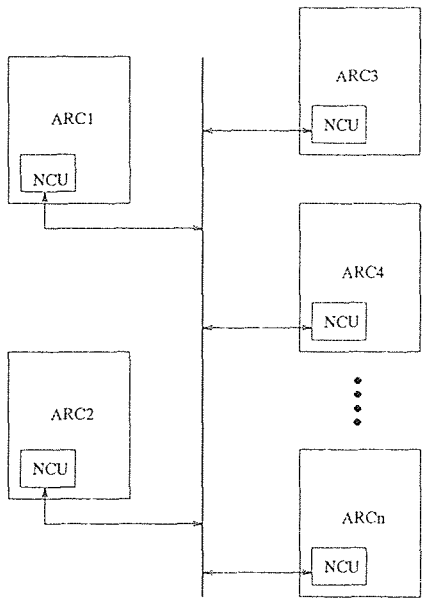


Figure 6-1. NCU and the ARC in a Multiprocessor Environment .

The NCU would be more of an extension of the control unit of the ARC. It would have active participation only in the execution of the control instructions. It would not only have an interface with the other processors in a multiprocessor environment, but also have internal connectivity with the different elements of the processor. As the NCU would handle requests for remote procedure calls, it would in a way be a dedicated control unit for the CALL instructions of the ARC.

Observation of the CALL instructions leads us to understand the different elements that the NCU needs to communicate with. For a better understanding of the operation of the interface that these instructions need, it is

necessary to refer [1-3], which provide details about the architectural support needed for these instructions, the different elements necessary for the NCU to communicate with thus providing the internal and the external interface can be realized. This can be seen in Figure 6-2.

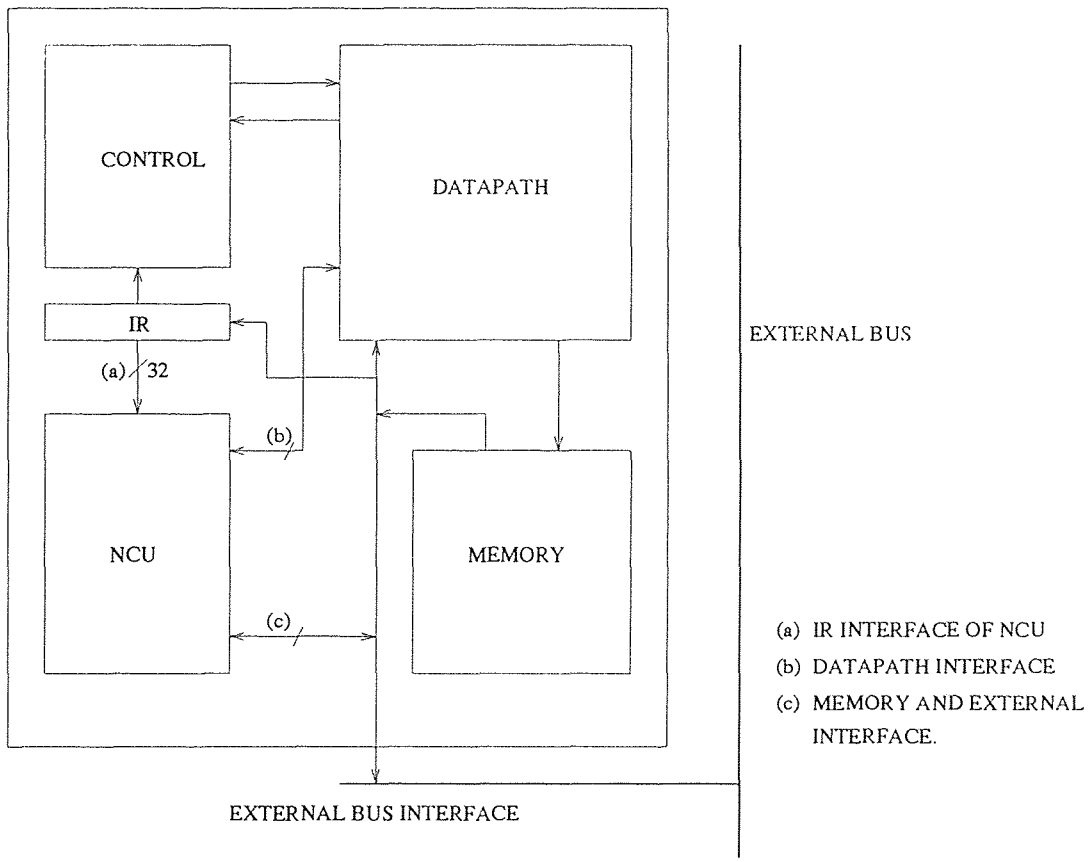


Figure 6-2. Internal Interface of the NCU

It is necessary to talk about the interface in greater detail. The ARC is a processor that is intended to be used in a multi-processor environment. It therefore needs to have an excellent handshaking capability with the other processors in the system. The ARCs operate on a stand-alone basis until interrupted by a CALL routine from another processor. This CALL is

serviced with a higher priority and the processor goes back to its original state.

Most CALL instructions access data from the Facility Register (FR) and the Facility Data Register (FDR) and the Facility Data Memory (FDM). The FR and the FDR are registers that hold addresses and are located in the Data-path. The Facility Data Memory holds data accessed by these instructions. This suffices the need for interface between the NCU and the other elements discussed. The NCU would definitely need to read the entire contents of the IR since for the CALL operation it is necessary to distinguish between the many CALL instructions and the immediate arguments included within. The NCU would also need an interface with the memory because it needs access to the FDM.

The External Interface of the NCU could be multiplexed together with the external interface bus. There is an address and data bus interface that already exists. This external interface could be accomodated to include the NCU interface. The CALL instructions have immediate arguments in them. Some have two and others have three immediate arguments. The NCU should also be able to handle the RETURN (RETURN from CALL) instructions. The NCU can be designed to accomodate the READ and WRITE instructions. For this operation the NCU could send some control signals to the Datapath unit and to the external interface to either read or write data. The source bus and destination bus interface has been shown in Figure 4-5.

The same concept could be applied at the external interface. In this case we would have only one bus for read and write respectively.

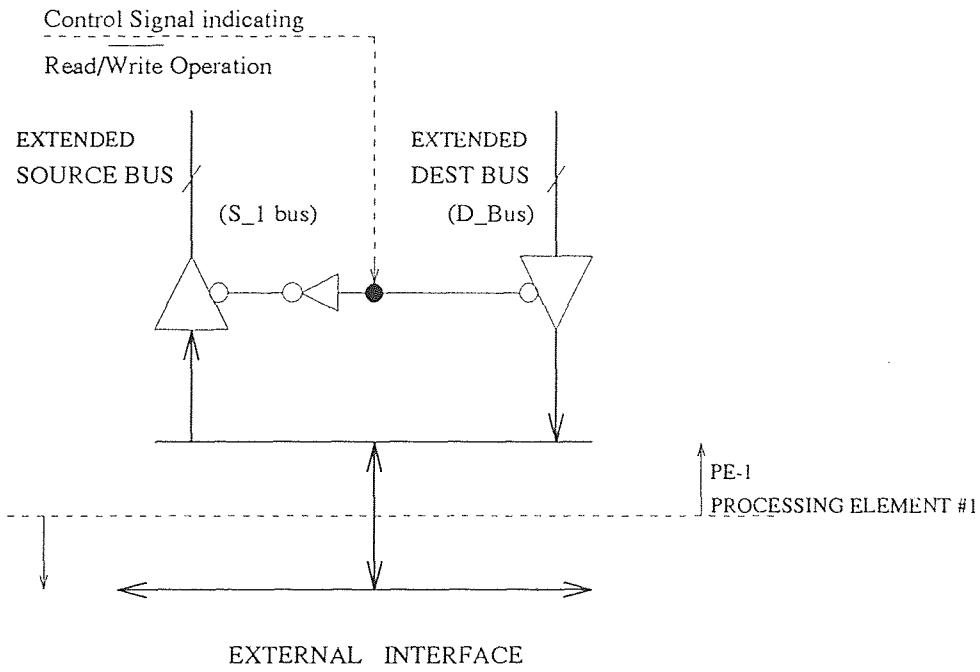


Figure 6-3. External Interface of the NCU .

6.2 Future Research

Future work on the ARC needs to be split up to provide for rapid implementation. The ARC could be implemented as the summation of the work of various smaller projects representing modules in the instruction set. The suggested partition for future work is as indicated:

- To break the ARC into smaller modules such as those implemented
- This break-up would enable work on the ARC to progress at a classroom level, wherein the smaller modules could be implemented as one semester

projects.

- Work from the modeling environment would have to proceed to the next level, where the modules could be implemented using tools such as schematic editors, and layout tools to realize the modules.
- Work would have to be commenced on the design of the NCU taking into considerations the interface requirements in case of a multi-processor environment.
- Parallelism in the instructions at the logic level needs to be investigated. This would lead to reduction in redundant control logic circuitry. Some optimization circuits have been included in Appendix B. On the same lines, the T-state diagrams for the CALL instructions would have to be studied and instructions with redundant operations can be clubbed together with optimization circuits.
- When all instructions have been implemented, the state machine implementation of the control logic for the ARC would have to be designed and implemented.

The break-up of projects:

- One major project would be to first implement the Datapath unit. Once the mask layout and extraction characteristics have been obtained, the actual internal propagation delays would be known. This information would be needed to design the control logic with necessary delay, allowing for the

data to stabilize. The design of the datapath is vital. This is because all the register transfer delays, the bus delays and other control line delays within the datapath have to be known in order to build the other control logic circuits.

- To design and implement a circuit to perform the operations indicated. The operations would be the ones indicated by the T-state diagrams. Design control logic circuitry to produce the necessary T-states. Care should be taken to see that the T-states are active high long enough considering the delays in the datapath. This means that the control logic circuitry should wait for data to stabilize in the datapath. Use the datapath is given in the figure as a model for your design.
- Implement the above design to obtain mask layouts using mask layout tools. Extract the parameters using extraction techniques and give a feedback to the design section to use these extracted parameters in their design simulations.
- Another design project would be to implement the external interface cicuitry. The project should be designed on the idea from the Figure 6-3. The interface should have the control signals to read data into the processing element and to write data out of the processing element to the external bus.
- Merge the structures so developed for the individual modules. The control logic designed by using the state machine techniques could replace all the

individual modules here. Simulate this using a logic simulator and accomodate for delays.

For example the design of the control unit for an instruction requiring a maximum of 8 T-states with a control bit to specify repeated T-states has the structure indicated in Figure 6-4. It is necessary to design the combinational logic to drive the counter. This counter drives the decoder which produces the T-states. Given data-transfer delays for the datapath unit, design the circuit such that the control unit produces the necessary signals at required time.

The logic structure of the control unit has been designed. It is necessary to implement the mask layout for the structure and obtain extraction parameters. These extraction parameters could be used to run a circuit simulation on the structure using a circuit simulation package. The delays then observed in the simulation output files could be used to modify the structure to alter the delay characteristics of the control logic structure.

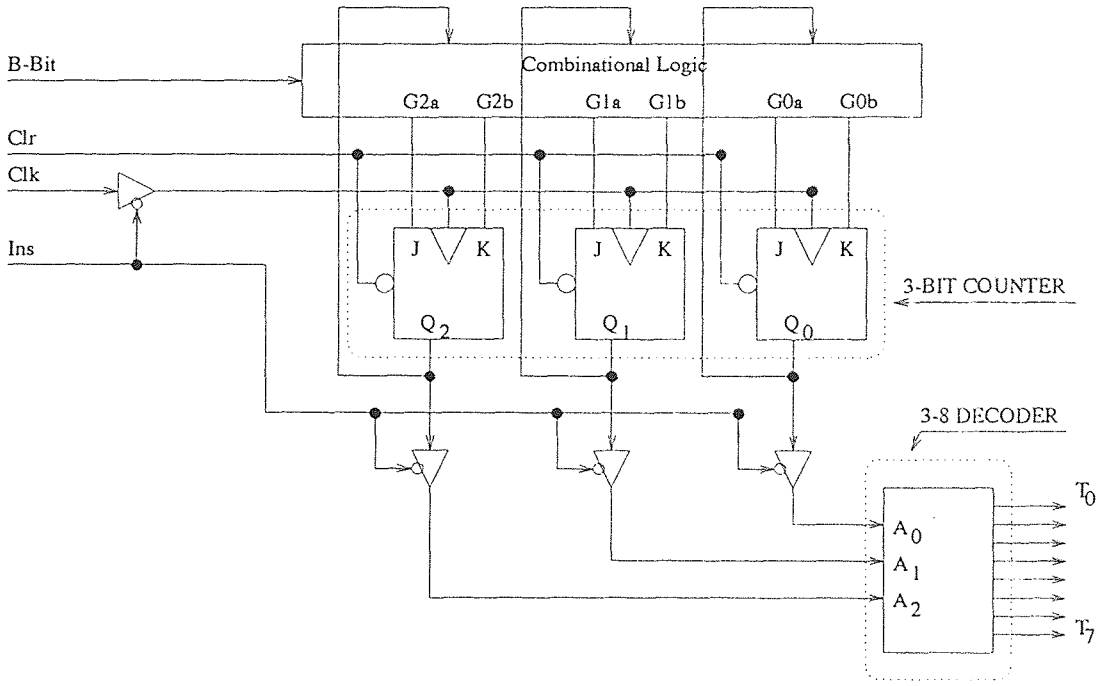


Figure 6-4. Design Example for a Project.

- List of Implementation projects:

Given Data should include specifications of the datapath, Bus propagation delays, memory access delays, and register transfer delays.

- Project 1: Given the parameters of the datapath of the ARC, design a control logic circuitry for the instruction specified using HDL. Use the delay parameters of the datapath for designing the control logic. Use a schematic editor to create lower instances of the logic modules. Use these lower level modules to build the layout cells. Capture the electrical connectivity information. Write out the logic simulator (Lsim) netlist and use this information to verify the design.

- Project 2: Given the parameters of the datapath of the ARC, design a Network Control Unit (NCU). The NCU is based on the CALL instruction. It receives the opcode information from the Instruction register (IR). Design the T-states required to perform the specified CALL instruction and together with the parameters specified for the datapath, arrive at the control logic that will enable a CALL instruction to execute.
- Project 3: The NCU must also be able to handle RETURN instructions. Design the T-states required to perform the specified RETURN instruction and together with the parameters specified for the datapath, arrive at the control logic that will enable a RETURN instruction to execute.

These are all the different ways in which the projects could be split up to slowly realize the ARC. A freeze on the project would be necessary at some stage to realize a primitive structure. It is to be noted that to build a processor of commercial practicality is not a one-man job. It takes a team and a lot of man-hours to build a processor from scratch. What has been done so far, has been a part of that design process utilizing new Computer Aided Design and Engineering tools to begin the realization of the ARC.

APPENDIX A

This chapter contains some of the source files that were developed. It also includes some of the testvector and initialization files.

- SOURCE FILE FOR THE DATAPATH UNIT:

```
//The module of the datapath which includes such elements such as the
//register set, the source buses, the destination bus, the various
//control signals which control the operation of these elements, the ALU
//and the interface of the Memory Address register MAR, to the memory.

MODULE datapath()

{

    IN Cts1_bus[9], Cts2_bus[3], C_tdbus, C_alu, C_fdbus[12];
    IN D_IRin[32], CtALUbuf, Clock, Reset[13];
    OUT Maddr[32], DMEMout[32];
    OUT Ovflow;

    //The number of registers in the datapath are 13. They are
    //all connected in such a way as to receive data from the
    //destination bus 'd_bus' and to place data onto the Source
```

```
//bus 's_bus'. The dataflow around the datapath is unique.
//The data from the registers have only one way of reaching
//their intended destination and that is through the ALU and
//the d_bus. This can be visualized by looking at the block
//diagram of the datapath.

//      The functions of the control signals follow.
//
//Cts1_bus: The control signal to gate data from the registers to
//        the source bus 1.
//Cts2_bus: The control signal to gate data from the registers to
//        the source bus 2.
//Ct_dbus:  The control signal to gate data from the ALU output
//        buffer to the databus
//C_alu:    The ALU control signal.
//C_fdbus:  The control signal to load registers from databus
//


BUILD{          int i;          //The registers that are linked
only to s1_bus//
```

```
        INSTANCE(reg32buf, TopLDS); // Register pointer to LDS//
        INSTANCE(reg32buf, TopPAS); // Register pointer to PAS//
        INSTANCE(reg32buf, MDR);    // Memory Data Register//
        INSTANCE(reg32buf, C_Reg);  // Clone # register//
        INSTANCE(reg32buf, P_cntr); // Program Counter//
        INSTANCE(reg32buf, FDR);    // Facility Data Register//
        INSTANCE(reg32buf, FR);     // Facility Register//
        INSTANCE(reg32buf, C_flag); // Control Flag//
        INSTANCE(reg32buf, MAR);    // Memory Address Register//

//The ALU output buffer which holds ALU output data//
        INSTANCE(reg32buf, ALUop);  // ALU output Buffer//
//The registers that are linked only to s2_bus//
        INSTANCE(reg32buf, TEMP);   // Temporary register//
        INSTANCE(reg32buf, Offset); // For address computation//
        INSTANCE(reg32buf, Label);  // Label register//
// Create instances of Busses within the datapath//
        INSTANCE(Bus, s1_bus);    // Create Source bus 1//
        INSTANCE(Bus, s2_bus);    // Create Source bus 2//
        INSTANCE(Bus, d_bus);     // Create Destination bus//
// Create instance of an ALU within the datapath//
```

```
INSTANCE(ad_sb4, alu);    // Create Instance of an ALU//

// Create the links between the various elements in the datapath //

for(i=0;i<=31;i++){       // Register To Bus Link//
NET(s1_bus.inp[i], TopLDS.A[i]);
NET(s1_bus.inp[i], TopPAS.A[i]);
NET(s1_bus.inp[i], MDR.A[i]);
NET(s1_bus.inp[i], MAR.A[i]);
NET(s1_bus.inp[i], C_Reg.A[i]);
NET(s1_bus.inp[i], P_cntr.A[i]);
NET(s1_bus.inp[i], FDR.A[i]);
NET(s1_bus.inp[i], FR.A[i]);
NET(s1_bus.inp[i], C_flag.A[i]);
NET(s2_bus.inp[i], Offset.A[i]);
NET(s2_bus.inp[i], Label.A[i]);
NET(s2_bus.inp[i], TEMP.A[i]);
NET(alu.A[i], s1_bus.out[i]);  // ALU To BUS Link//
NET(alu.B[i], s2_bus.out[i]);
NET(ALUop.I[i], alu.Qo[i]);
NET(ALUop.A[i], d_bus.inp[i]);
```

```
// Bus to Register Link//
                NET(TopLDS.I[i], d_bus.out[i]);
                NET(TopPAS.I[i], d_bus.out[i]);
                NET(MDR.I[i], d_bus.out[i]);
                NET(MAR.I[i], d_bus.out[i]);
                NET(C_Reg.I[i], d_bus.out[i]);
                NET(P_cntr.I[i], d_bus.out[i]);
                NET(FDR.I[i], d_bus.out[i]);
                NET(FR.I[i], d_bus.out[i]);
                NET(C_flag.I[i], d_bus.out[i]);
                NET(Label.I[i], d_bus.out[i]);
                NET(TEMP.I[i], d_bus.out[i]);
                NET(Offset.I[i], d_bus.out[i]);
                NET(Maddr[i], MAR.A[i]);
                NET(DMEMout[i], d_bus.out[i]);
                NET(s2_bus.inp[i], D_IRin[i]);
                }

// The link from main reset to individual resets of the registers//

                NET(TopLDS.Reset, Reset[0]);
                NET(TopPAS.Reset, Reset[1]);
```

```
NET(MDR.Reset, Reset[2]);
NET(MAR.Reset, Reset[3]);
NET(C_Reg.Reset, Reset[4]);
NET(P_cntr.Reset, Reset[5]);
NET(FDR.Reset, Reset[6]);
NET(FR.Reset, Reset[7]);
NET(C_flag.Reset, Reset[8]);
NET(Label.Reset, Reset[9]);
NET(TEMP.Reset, Reset[10]);
NET(Offset.Reset, Reset[11]);
NET(ALUop.Reset, Reset[12]);
```

// The link from main Clock to individual Clocks of the registers//

```
NET(TopLDS.CP, Clock);
NET(TopPAS.CP, Clock);
NET(MDR.CP, Clock);
NET(MAR.CP, Clock);
NET(C_Reg.CP, Clock);
NET(P_cntr.CP, Clock);
NET(FDR.CP, Clock);
NET(FR.CP, Clock);
```

```
NET(C_flag.CP, Clock);
NET(ALUop.CP, Clock);
NET(TEMP.CP, Clock);
NET(Label.CP, Clock);
NET(Offset.CP, Clock);

// The link from main Enable to individual Enable signals of the registers//
NET(TopLDS.En0, Cts1_bus[0]);
NET(TopLDS.En1, Cts1_bus[0]);
NET(TopPAS.En0, Cts1_bus[1]);
NET(TopPAS.En1, Cts1_bus[1]);
NET(MDR.En0, Cts1_bus[2]);
NET(MDR.En1, Cts1_bus[2]);
NET(MAR.En0, Cts1_bus[3]);
NET(MAR.En1, Cts1_bus[3]);
NET(C_Reg.En0, Cts1_bus[4]);
NET(C_Reg.En1, Cts1_bus[4]);
NET(P_cntr.En0, Cts1_bus[5]);
NET(P_cntr.En1, Cts1_bus[5]);
NET(FDR.En0, Cts1_bus[6]);
NET(FDR.En1, Cts1_bus[6]);
NET(FR.En0, Cts1_bus[7]);
```

```
NET(FR.En1, Cts1_bus[7]);
NET(C_flag.En0, Cts1_bus[8]);
NET(C_flag.En1, Cts1_bus[8]);
NET(Label.En0, Cts2_bus[0]);
NET(Label.En1, Cts2_bus[0]);
NET(TEMP.En0, Cts2_bus[1]);
NET(TEMP.En1, Cts2_bus[1]);
NET(Offset.En0, Cts2_bus[2]);
NET(Offset.En1, Cts2_bus[2]);
```

// The link from main Load to individual Load signals of the registers//

```
NET(TopLDS.Ld, C_fdbus[0]);
NET(TopPAS.Ld, C_fdbus[1]);
NET(MDR.Ld, C_fdbus[2]);
NET(MAR.Ld, C_fdbus[3]);
NET(C_Reg.Ld, C_fdbus[4]);
NET(P_cntr.Ld, C_fdbus[5]);
NET(FDR.Ld, C_fdbus[6]);
NET(FR.Ld, C_fdbus[7]);
NET(C_flag.Ld, C_fdbus[8]);
NET(Label.Ld, C_fdbus[9]);
```

```
        NET(TEMP.Ld, C_fdbus[10]);
        NET(Offset.Ld, C_fdbus[11]);

// The control signals relative to the ALU output buffer //

        NET(ALUop.Ld, CtALUbuf);
        NET(ALUop.En0, C_tdbus);
        NET(ALUop.En1, C_tdbus);
        NET(alu.Sc, C_alu);
        NET(Ovflow, alu.Qo[32]);

        }

}
```

- INITIALIZATION AND SIMULATION FILE FOR THE DATAPATH

```
# initialization file
bus -w1 S1_bus[31:0] x //Create a bus of width 31 bits and radix hex.
bus -w1 S2_bus[31:0] x
bus -w1 D_bus[31:0] x
bus -w1 reset[12:0] x
Rename S1_bus[31:0] S1_bus // Rename the busses with specific names
and
Rename S2_bus[31:0] S2_bus // also to view all the 32 bits in a single
Rename D_bus[31:0] D_bus   // bus waveform.
Rename reset[12:0] reset
low Reset S1_bus S2_bus D_bus
lpulse -w1 Clock Low High 0 0 0 6 10
# LPULSE: S1=SL31 S2=SH31 Td=0 Tr=0 Tf=0 Pw=5 Per=10
simulate 4
33221100 S1_bus[31:0]  // Give certain test inputs to the bus.
33220000 S2_bus[31:0]
high C_alu CtALUbuf C_tdbus // Operate the control signals
probe D_bus S2_bus S1_bus   // To probe a particular bus or signal
simulate 10    // Simulate for a 10 unit time.
```

- SOURCE FILE FOR THE ALU: (Refer Figure 4-6)

```
//Module of a 32 bit ALU which can perform addition/subtraction
//operations. The lower level module is the 1-bit adder.
//
MODULE ad_sb4()
{
   IN A[32],B[32],Sc;
   OUT Qo[33];
   BUILD{

   int i;
   for (i=0;i<=31;i++){
   INSTANCE(add_sub, ads[i]);
   NET(ads[i].A,A[i]);
   NET(ads[i].B,B[i]);
   NET(ads[i].Sb,Sc);
   NET(ads[i].SUM,Qo[i]);
   }
   for (i=1;i<=31;i++){
   NET(ads[i].C,ads[i-1].CARRY);
   }
```

```
    NET(ads[31].CARRY,Qo[32]);
    NET(ads[0].C,Sc);
    }
}
```

- SOURCE FILE FOR THE 32 BIT PARALLEL READ/LOAD REGISTER: (Refer Figure 4-3)

```
//module of a 32 bit parallel load/read register using the 1-bit
//register module. This register can be loaded in parallel & can
//be read in parallel.

MODULE reg32buf()

{
    IN Ld, I[32], CP, Reset, En0, En1;
    OUT A[32];

    BUILD{

    int i;
```

```
    for(i=0;i<=31;i++){
    INSTANCE(reg1b, reg[i]);
    INSTANCE(prim_tribuf, tribuf[i], 1.0, 1.0, 2, 2, 0);

    NET(tribuf[i].enable0, En0);
    NET(tribuf[i].enable1, En1);

    NET(reg[i].L1, Ld);
    NET(reg[i].CP, CP);
    NET(reg[i].R, Reset);
    NET(reg[i].I, I[i]);
    NET(tribuf[i].in, reg[i].Q);
    NET(A[i], tribuf[i].out);
    }
        }
}
```

- SOURCE FILES FOR SOME BASIC FLIP-FLOPS

  D Flip-Flop

```
MODULE dff() // module of a dfip_flop
```

```
{
   IN CP,D,R; // declaration of the inputs
   OUT Q;   // declaration of the output terminals.

   BUILD{
      //INSTANCE(type, name, inputs, rt, ft, fout);
   INSTANCE(prim_nand, nand0, 2, 1.0, 1.0, 2, 2, 0);//using
   INSTANCE(prim_nand, nand1, 3, 1.0, 1.0, 2, 2, 0);//instances
   INSTANCE(prim_nand, nand2, 2, 1.0, 1.0, 2, 2, 0);//of
   INSTANCE(prim_nand, nand3, 2, 1.0, 1.0, 2, 2, 0);//primitive
   INSTANCE(prim_inv, inv0, 1, 1.0, 2, 2, 0);       //logic

   NET(nand0.in[0],nand2.out);
   NET(nand0.in[1],nand1.out);

   NET(nand1.in[0],nand3.out);
   NET(nand1.in[1],nand0.out);
   NET(nand1.in[2],R);

   NET(nand2.in[0],CP);
   NET(nand2.in[1],D);
```

```
    NET(nand3.in[0],inv0.out);
    NET(nand3.in[1],CP);

    NET(inv0.in,D);
    NET(Q,nand0.out);
    }
}
    JK Master Slave Flip-Flop

MODULE jkff()
{
    IN J,K,CP,Clr;
    OUT Q;

    BUILD{
    int i;

    for (i=0;i<=3;i++){
    INSTANCE(prim_nand, nand3[i], 3 ,1.0 ,1.0 , 2, 2, 0);
    }
    for (i=0;i<=3;i++){
    INSTANCE(prim_nand, nand2[i], 2, 1.0 ,1.0 , 2, 2, 0);
```

```
}
INSTANCE(prim_inv, inv, 1.0,1.0, 2, 2, 0);

NET(nand3[0].in[0],nand2[3].out);
NET(nand3[0].in[1],J);
NET(nand3[0].in[2],CP);
NET(nand3[1].in[0],nand3[3].out);
NET(nand3[1].in[1],K);
NET(nand3[1].in[2],CP);

NET(nand3[2].in[0],nand3[0].out);
NET(nand3[2].in[1],nand2[0].out);
NET(nand3[2].in[2],Clr);

NET(nand3[3].in[0],nand2[1].out);
NET(nand3[3].in[1],nand2[3].out);
NET(nand3[3].in[2],Clr);

NET(nand2[0].in[0],nand3[2].out);
NET(nand2[0].in[1],nand3[1].out);

NET(nand2[1].in[0],nand3[2].out);
```

```
NET(nand2[1].in[1],inv.out);

NET(nand2[2].in[0],inv.out);
NET(nand2[2].in[1],nand2[0].out);

NET(nand2[3].in[0],nand3[3].out);
NET(nand2[3].in[1],nand2[2].out);

NET(Q,nand3[3].out);
NET(inv.in,CP);
    }
 }
```

- SOURCE FILE FOR THE MAIN CONTROL UNIT (figure 4-8):

```
MODULE Ctrlmain1()

{
  IN O, I, X, Z, Y, S, Clk, Clr;
  OUT Ts[4];

  BUILD{
  int i;

  INSTANCE(prim_and, and, 2, 1.0, 1.0, 2, 2, 0);
  INSTANCE(prim_buf, buffer, 1.0, 1.0, 2, 2, 0);
  INSTANCE(Dec2_4, dec);

  for(i=0;i<=1;i++){
  INSTANCE(MUX4_1, Mux[i]);
  INSTANCE(deff, deff[i]);
  INSTANCE(prim_or, or[i], 2, 1.0, 1.0, 2, 2, 0);
  INSTANCE(prim_inv, inv[i], 1.0, 1.0, 2, 2, 0);
  }
```

```
NET(inv[0].in, X);
NET(inv[1].in, Z);
NET(and.in[0], inv[0].out);
NET(and.in[1], Y);
NET(or[0].in[0], and.out);
NET(or[0].in[1], X);
NET(or[1].in[0], Z);
NET(or[1].in[1], inv[1].out);

NET(Mux[0].I[0], O);
NET(Mux[0].I[1], I);
NET(Mux[0].I[2], X);
NET(Mux[0].I[3], inv[1].out);

NET(Mux[1].I[0], S);
NET(Mux[1].I[1], O);
NET(Mux[1].I[2], or[0].out);
NET(Mux[1].I[3], or[1].out);

NET(deff[0].D, Mux[0].out);
NET(deff[1].D, Mux[1].out);
```

```
    NET(deff[0].Clr, Clr);
    NET(deff[1].Clr, Clr);
        NET(deff[0].Clk, Clk);
    NET(deff[1].Clk, Clk);

    NET(deff[0].Q, buffer.in);
    NET(dec.A, buffer.out);
    NET(deff[1].Q, dec.B);

    NET(Mux[0].S[0], deff[1].Q);
    NET(Mux[1].S[0], deff[1].Q);

    NET(Mux[0].S[1], buffer.out);
    NET(Mux[1].S[1], buffer.out);

    for(i=0;i<=3;i++){
    NET(Ts[i], dec.T[i]);
    }
        }
}
```

APPENDIX B

T-STATE OPERATIONS

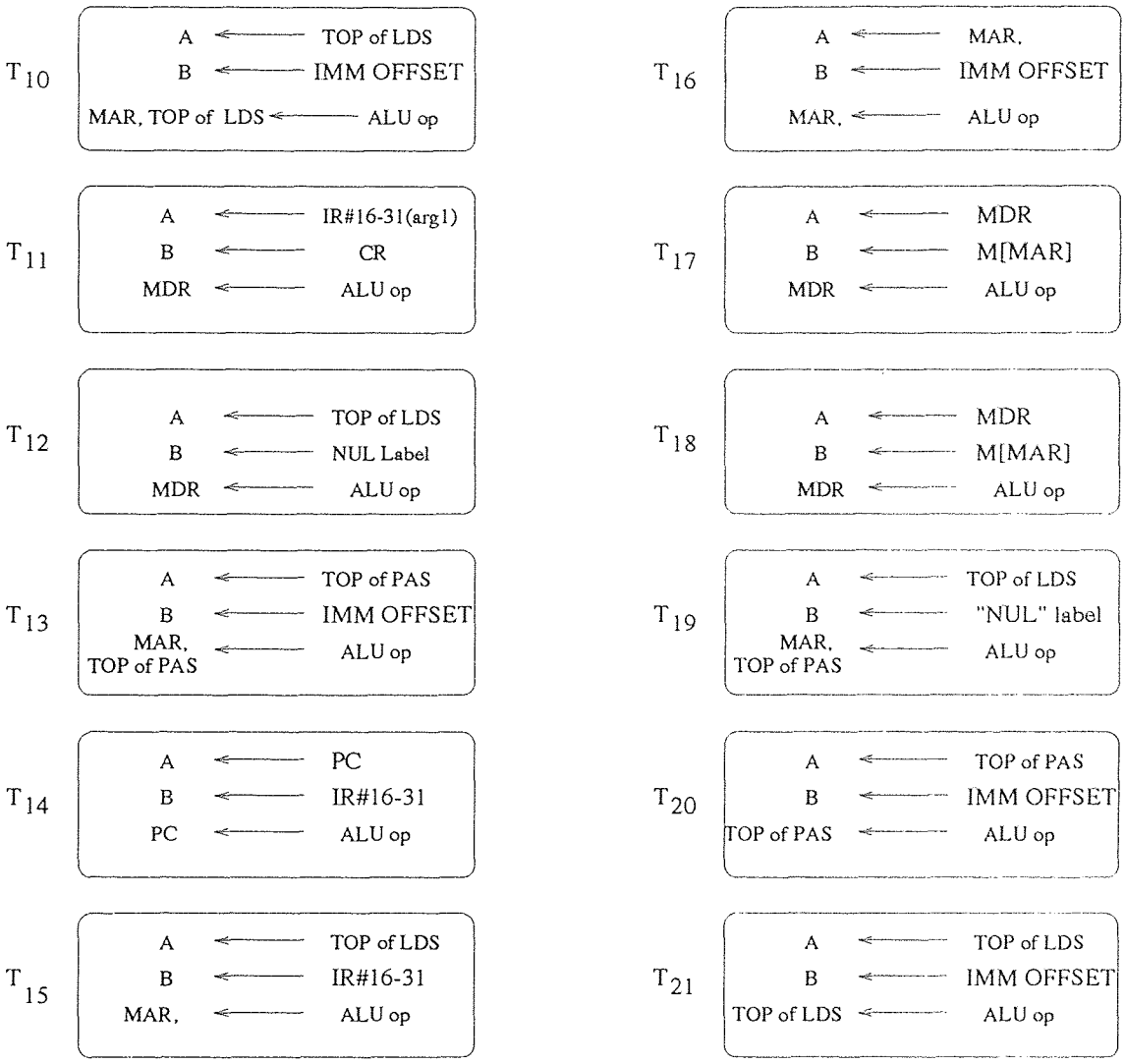


Figure B-1 Internal T-state Operations.

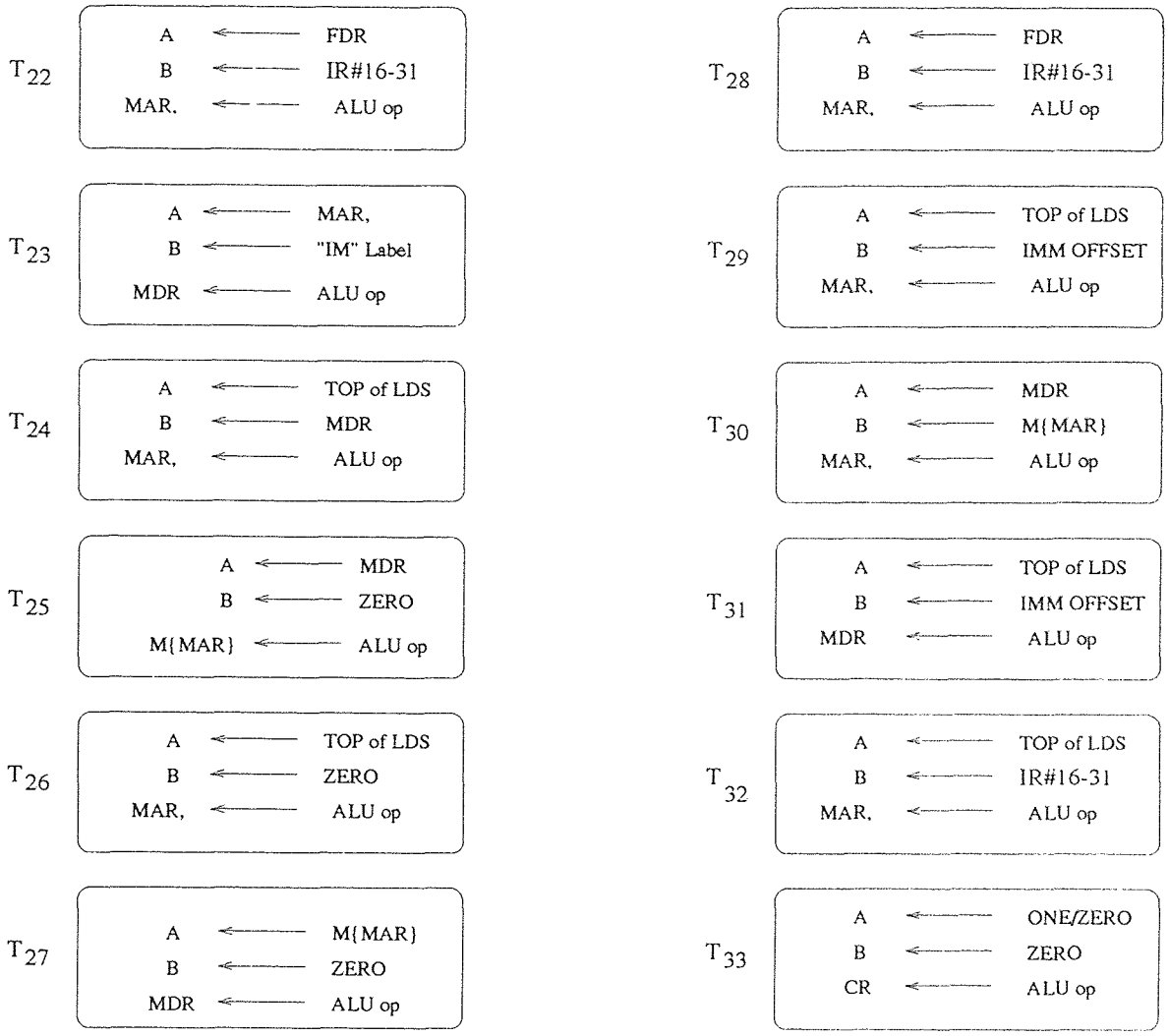


Figure B-1 Internal T-state Operations.

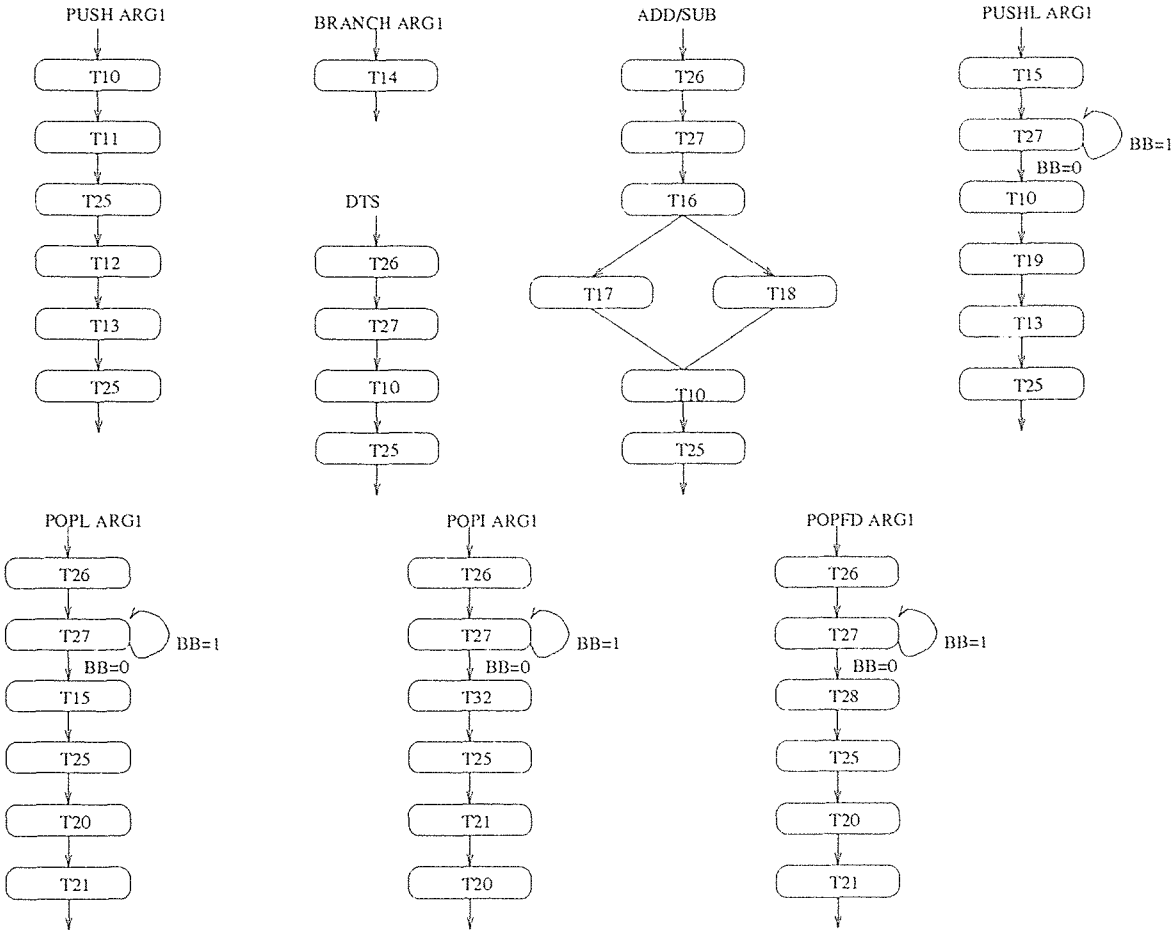


Figure B-2 T-state Sequence of Instructions.

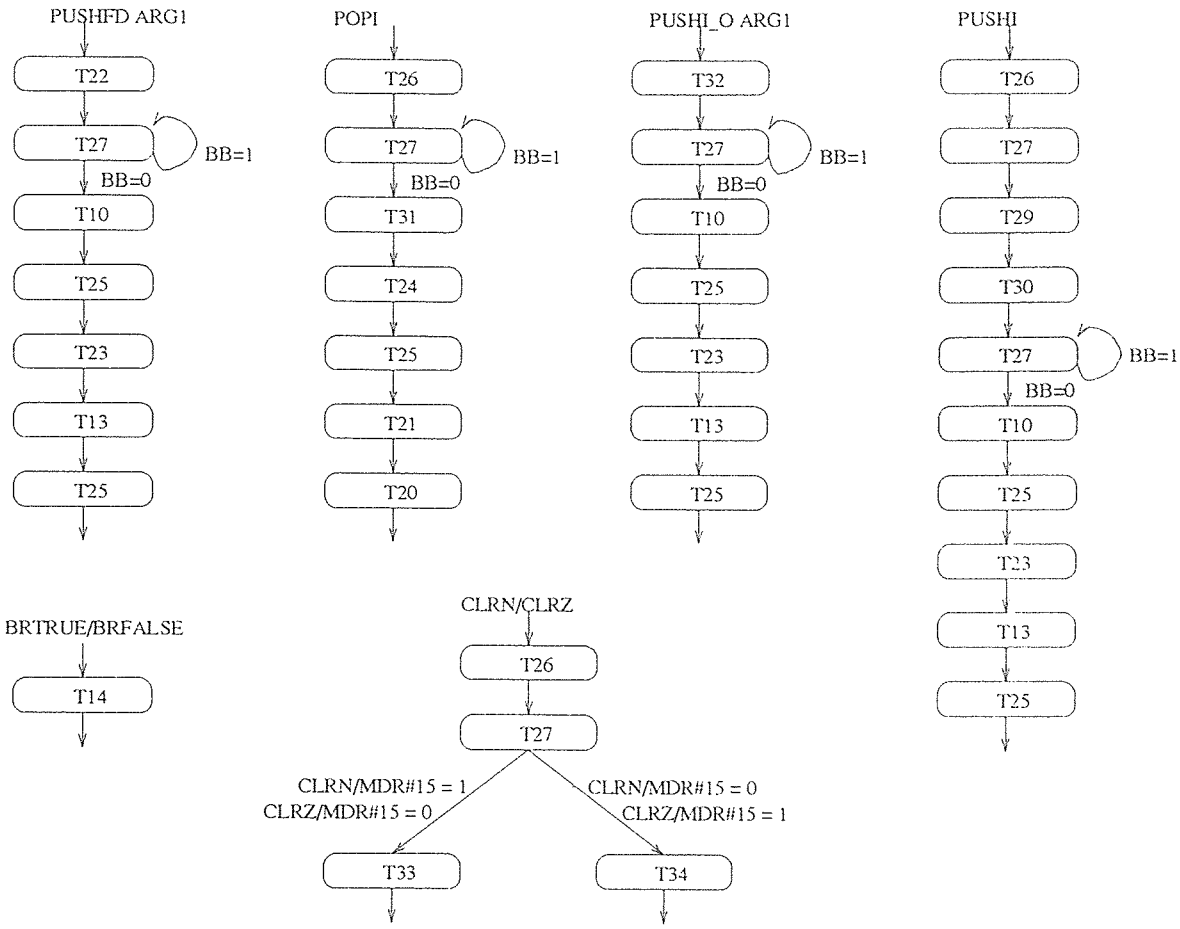


Figure B-2 T-state Sequence of Instructions.

APPENDIX C

OPTIMIZATION CIRCUITS

Some instructions are very similar at the logic transfer level. These have been identified and some of them are presented here with logic optimization for their control circuitry. The extra logic has been avoided by clubbing the instructions together with simple extra logic to supplement for the execution of both instructions with just one control circuit. See Figure C-1.

CR	BRTRUE	BRFALSE
0	FIN	T14
1	T14	FIN

IF BRTRUE LET INSTRUCTION SIGNAL PRODUCED = 0
IF BRFALSE LET INSTRUCTION SIGNAL PRODUCED = 1

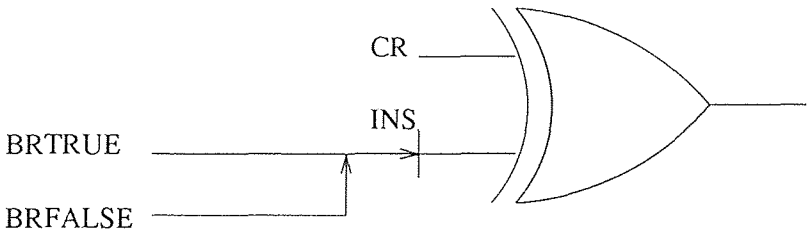


Figure C-1. Control Logic Optimization for Branch instruction.

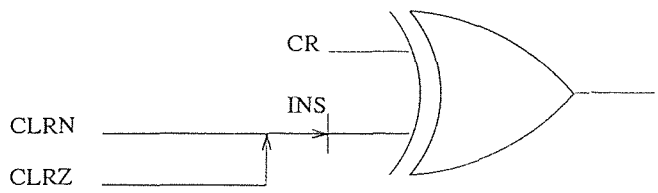
Observe the figure where the table shows the parallelism exhibited by the BRANCH instruction. A simple 2-input XOR gate would suffice for the operation of both the instructions with one control logic. After the instruction signal is obtained from the respective instructions, they are used to drive the INS signal at one of the XOR inputs. This implementation is one case of optimization.

In the ADD or SUB instruction the same T-states are followed except for one state. This can be observed in the T-state diagrams in Appendix B. This case could be exploited as another case of parallelism and optimization of the control logic could be performed.

In case of the CLRN and CLRZ parallelism is again observed. The T-state flow of these instructions are almost similar except for the order. This can be seen in Appendix B. The logic optimization in this case is shown in Figure C-2.

FLAG	CLRN	CLRZ
0	T34	T33
1	T33	T34

IF CLRN LET INSTRUCTION SIGNAL PRODUCED = 0
IF CLRZ LET INSTRUCTION SIGNAL PRODUCED = 1



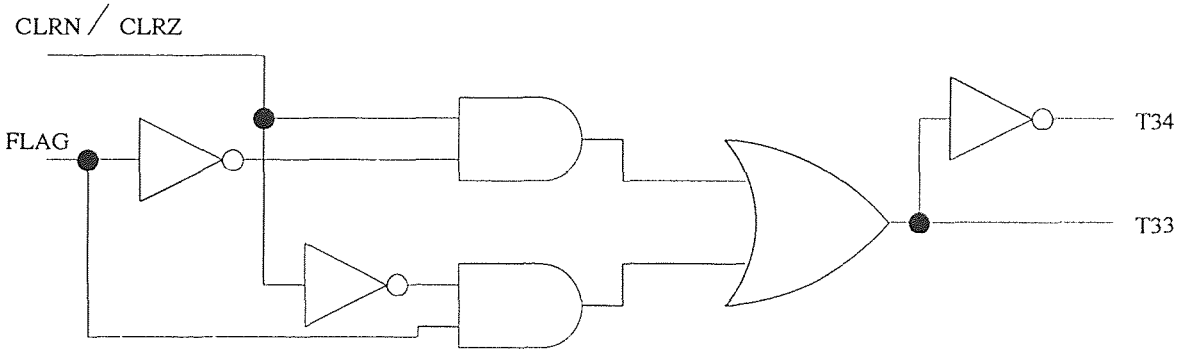


Figure C-2. Control Logic Optimization for CLRN/CLRZ instruction.

REFERENCES

1. Welch, L.R. "Architectural Support for, and Parallel Execution of, Programs Constructed from Reusable Software Components", PhD dissertation, The Ohio State University, December 1990.

2. Welch L.R. "Generic Modules: Efficient Compilation ans Execution", Research report: CIS-91-22, NJIT, Newark, N.J, 07102.

3. Welch L.R. "A Parallel Virtual Machine for Programs Composed of Abstract Data Types", Paper submitted to the IEEE transactions on Computers.

4. Manolis G.H. Katevenis *Reduced Instruction Set Computer Architectures for VLSI*, The MIT Press, (1984)

5. Sternhiem E., R. Singh and Y. Trivedi. *Digital Design with Verilog HDL*, Automata Publishing Co, (1990)

6. Geiger L.R., P.E. Allen and N.R. Strader. *VLSI Design Techniques for Analog and Digital Circuits*, McGraw-Hill Publishing Co, (1990)

7. Mano M.M., *Digital Design*, Prentice Hall Inc (1984)

8. Weste N., K. Eshraghian, *Principles of CMOS VLSI Design*, Addison Wesley (1985)

9. Hennessey, J.L. and D.A. Patterson. *"Computer Architecture - A Quantitative Approach*, Morgan Kaufman Publishers, CA, (1990)

10. Hamacher, C.V., Z.G. Vranesic and S.G. Zaky. *Computer Organization*, Mc-Graw Hill Publishing Co, (1984)

11. Navabi Z. *VHDL: Analysis and Modeling of Digital Systems* McGraw-Hill College Division, (1992)

12. M-language users guide, Mentor Graphics.

13. Explorer Lsim users guide, Mentor Graphics

14. Prezbylski S.A. et al. "Organization and VLSI implementation of MIPS", Journal of VLSI and Computer systems, Vol 1, number 2

15. Ditzel, D.R., H.R. Mclellan and A.D. Berenbaum. "The Hardware Architecture of the CRISP microprocessor", transactions of the ACM, 1987

16. Patterson. D. A. and C.H. Sequin. "RISC-I: A Reduced Instruction Set VLSI Computer," Proc. of the Eighth Annual Symposium on Computer Architecture, Minneapolis, Minn., May 1981

17. Triebel, W.A. and A. Singh. *The 68000 Microprocessor, Architecture, Software, and Interfacing Techniques* Prentice Hall, (1986)

18. Harman, T.L. *The Motorola MC68020 and 68030 Microprocessor, Assembly Language, Interfacing and Design." Prentice Hall, (1989)*

19. *Wakerly, J.F. Microcomputer Architecture and Programming, The 68000 Family* John Wiley and Sons Inc, (1989)