A stack-based RISC architecture for control applications

by Timothy Dale Thompson

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering
Montana State University

Abstract:
Automatic control systems are assuming an increasingly important role in the advancement of modern civilization and technology. This paper focuses on the development of a very simple stack-based computer architecture tailored to execute a core subset of the Forth programming language, with consideration toward use in real-time control applications.

The instruction set developed is composed of 27 instructions, which operate on a processor with a dual-stack architecture that has a datapath width of 16 bits. Instruction op-codes are 16 bits wide. The limitations of this instruction set prevent the processor from being used in applications when extensive floating point arithmetic calculations are necessary, but the instruction set is adequate for processor applications such as direct memory access control.

A STACK-BASED RISC ARCHITECTURE FOR CONTROL APPLICATIONS

by

Timothy Dale Thompson

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 1991

APPROVAL

of a thesis submitted by

Timothy Dale Thompson

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

11/18/91
Date

Chairperson, Graduate Committee

Approved for the Major Department

11/26/91
Date

D. L. Price for V. Lorenz
Head, Major Department

Approved for the College of Graduate Studies

December 17, 1991
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his/her absence, by the Dean of Libraries when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Timothy D. Thompson

Date 11/26/91

ACKNOWLEDGEMENTS

I would like to thank the members of my committee: Kel Winters, Roy Johnson and Harley Leach, for their guidance during my graduate work. I would also like to thank Bob Wall and Diane Mathews for their help in developing this thesis. Finally, thanks to Jaye Mathisen for his help with UNIX and the computer system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Automatic control systems are assuming an increasingly important role in the advancement of modern civilization and technology. This paper focuses on the development of a very simple stack-based computer architecture tailored to execute a core subset of the Forth programming language, with consideration toward use in real-time control applications.

The instruction set developed is composed of 27 instructions, which operate on a processor with a dual-stack architecture that has a datapath width of 16 bits. Instruction op-codes are 16 bits wide. The limitations of this instruction set prevent the processor from being used in applications when extensive floating point arithmetic calculations are necessary, but the instruction set is adequate for processor applications such as direct memory access control.

CHAPTER 1

INTRODUCTION

Automatic control systems are assuming an increasingly important role in the advancement of modern civilization and technology. The phrase automatic control system is self explanatory: the word system implies not just one component but a number of components which work together to achieve a particular goal; that goal is the control of some physical quantity; and the control is to be accomplished in an automatic fashion, without the aid of human supervision.

Practically every aspect of everyone's day-to-day activities is affected by some type of control system. For example, automatic controls in heating and air-conditioning systems regulate the temperature of office buildings and homes. There are control systems in robotics, power systems, weapon systems, aircraft, automobiles, washers and dryers, toasters, and many other common items. Control systems are an integral component of any industrial society, and are necessary for the production of goods required by the people of the world.

Types of Control Systems

Control systems can be separated into two classes: open-loop control systems, and closed-loop control systems. An

open-loop control system is characterized by the fact that the output quantity has no effect upon the input quantity. An example of an open-loop control system is the toaster. In this case the input quantity is the timer which controls the length of time that heat is applied. The output quantity, which is the darkness of the toast, can in no way alter the length of time that heat is applied. Open-loop control systems have practical use in numerous situations, due to their simplicity and economy.

Although open-loop control systems have many applications, in a large number of systems there is a need for more accurate and adaptive control. A necessary factor in this case is the feedback of the output of the system to the input of the system, where it is compared to the desired output value and adjusted accordingly. This type of control system, where the output has an effect on the input quantity, is known as a closed-loop control system. The thermostat in a home or office building is a common example of a closed-loop control system. A coil in the thermostat is affected by the desired temperature setting and the actual room temperature. If the room temperature is less than the chosen temperature, the coil changes shape, which in turn activates a relay, which causes the furnace to produce heat in the room. When the room reaches the desired temperature, the coil again changes shape, deactivating the relay, which turns off the furnace. Most industrial control systems are closed-loop control systems,

since the quality of a manufactured product is dependent on the accuracy of the systems which are used to fabricate it.[D'Azzo88]

A control system can be implemented as an analog system, or as a digital system. An analog signal is a continuous and time varying signal which describes most physical quantities such as light or sound. In a control system, these quantities are usually represented by a time varying voltage. A digital system represents continuous values, to a given accuracy, as digital values. A digital system uses binary numbers, or a base 2 counting system, to represent digital quantities, since the binary states 1 and 0 are easily represented by digital electronic devices.

Analog electronic systems are typically made up of discrete transistors, analog integrated circuits, and resistors, capacitors, and inductors, interconnected to perform a specific function. Analog systems are often used when high-frequency signals must be processed, or when simple functions such as signal amplification, comparison, or filtering are required.

Digital systems can be separated into two classes: dedicated digital systems, and computer-based systems. Dedicated digital systems are often used when rapid response to inputs is necessary, but the processing of analog signals is not necessary. Digital logic elements, such as combinational logic gates and storage elements, and the way

they are interconnected define the function of a dedicated digital system. These systems are high speed, but the application must be simple, and function of the system must not be expected to change, since changing the function of the system would require that the system be redesigned. Often a digital system is composed of discrete components, but for high performance applications, custom VLSI (very large scale integrated) circuits are usually required.

Although the preceding two types of systems are characterized by high performance, they lack flexibility. Computer-based control systems have the advantage that they can be designed for a general case, and then programmed for a specific task. This allows mass production of these systems, which reduces the cost of the systems, and makes them more available for use by system designers. Programs can be easily revised during the development cycle of the system, and later, while the system is in use, the program can be modified to adjust to a changing environment. Another advantage of computer-based systems is that they are compatible with both analog and digital inputs and outputs. The flexibility in a computer-based control system does not come without some functional cost. Operation of this type of system is slower than that of hard wired analog and dedicated digital systems. In many cases, however, the speed of a computer-based system is sufficient, and the number of applications continues to increase. Other factors in the increased use of this type of

system include speed improvements in microprocessors, special purpose architectures, cost reductions in conjunction with increased functionality, ease of testing, and the growing number of designers in industry who are familiar with microcomputer technology.[Lawrence87]

Real-Time Control Systems

An important class of computer-based control systems is known as real-time control systems. A real-time system is defined as any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period.[Young82] The lag time between an input signal and the necessary output changes greatly as applications vary. In some systems, a failure to respond to an input signal can be tolerated, but in others, a failure to respond can be as bad as an incorrect response. This leads to the conclusion that the correctness of the response of a real time system depends not only on the result of processing the input data, but also on the time at which the result is produced. To distinguish between systems in which a failure to respond can be accepted and those in which a failure to respond may be disastrous, real-time systems may be classified as hard real-time systems or soft real-time systems. Hard real-time systems are those in which a response within a given time window is critical, while soft real-time systems are those in which response times are important, but the system can still function if a response time deadline is missed.

Another important requirement for a real time system is that it must have a high degree of reliability. Since many computer-based systems operate without human supervision, it is important that if a failure occurs, the failure does not have destructive effects on the system being controlled. The system should detect the failure and execute a controlled recovery.[Burns90]

System Hardware and Software

A real-time computer-based control system is a combination of hardware and software. There is little to distinguish real-time computer system hardware from that of a general purpose computer system, but in addition to the computer hardware there are other hardware components necessary for interaction with the world in a real-time system. Input signals to the system are furnished by sensors, which are devices that produce an output signal, usually an analog voltage level, which is a function of the change in a physical variable being represented.

Once a change is sensed and represented as an analog signal, it may be necessary to modify the signal so that it may be sampled and interfaced with the processor. Signal conditioning includes the matching of sensor output voltages to voltages acceptable by input interfaces, filtering of unwanted high frequency or power supply noise in the signal, conversion of signal forms, such as the conversion of an analog signal into a series of pulses for transmission in

which the frequency or duration of the pulses is proportional to the sensor voltage level, and electrical isolation to prevent noise and voltage fluctuations which may damage the system.

The next stage, if one is following a signal produced by an event through the control system from the time it is sensed until a correcting signal is produced, is the computer input stage. There are two types of computer interfaces: analog input interfaces, and digital input interfaces. In an analog interface, the analog signal is passed through an analog-to-digital (A/D) converter. An A/D converter samples the analog input voltage at one instant of time and converts it into an equivalent digital value. The accuracy of the digital value is dependent upon the number of binary digits (bits) used. For example, an A/D converter which has a 16 bit resolution can represent 2^{16} = 65,536 different values between the minimum and maximum input voltages allowed. A digital input interface is much simpler. Since the input data is already in binary form, it can simply be transferred directly into the computer when the information is needed.

The most important part of a real-time control system is the processor. In the broadest sense, the processor acts on the information that it receives from the input interfaces. This involves fetching the instructions of the controller program from memory and executing them, fetching data from memory and input interfaces, writing data to memory and output

interfaces, and responding to external requests for action.

Output interfaces also can be separated into two categories. Analog output interfaces take a digital value, and by means of an digital-to-analog (D/A) converter, generate an analog output signal. A 16-bit D/A converter with output voltage range of 0 - 15 Volts has 65,536 possible levels of output voltage in that range, which gives a resolution of approximately 2.3×10^{-4} volts. Digital output interfaces are again very simple. The output data is written to a storage register, which drives the output signal lines, and the data remains on those lines until new data is written to the register.

It may be necessary to perform some conditioning on the output signals, which is similar to the conditioning done on input signals, before they reach the devices that the signals are controlling. Often the devices that are controlled are actuators. An actuator is any device which produces a motion. Electric actuators include relays, motors, and solenoids. Non-electric actuators, such as hydraulic rams or pneumatic devices, must have some sort of electrical interface in order for the devices to be computer-controlled.[Savitzky85]

The other part of a real-time computer-based system is the system software. The basic requirement for real-time software is that it must execute quickly and not take up much storage space. Computer languages can be divided into two classes: assembly languages, and high-level languages. The

most optimal software for real-time systems is written in the assembly language of the processor which is the core of the system. Assembly language is a method of programming a processor in which the instructions, the actual binary input bit patterns, that the processor will execute directly are given alphabetic names. Using assembly language allows the programmer to take advantage of every time-saving trick that the machine allows, and this means that the software will execute quicker than if it is programmed in another language. Assembly language programs execute approximately twice as fast as programs written in high-level languages.[Lawrence87] The main problem with assembly language is that programming is time consuming and difficult, and requires a programmer who is familiar with the particular processor being used to produce quality programs.

High-level languages are programming languages for which the statements do not have a simple, direct mapping into the internal binary representation of the processor's instructions, but are instead closer to the terms in which the user thinks of a problem. It is much easier to write programs in a high-level language, since a high-level language will support a variety of control and data structures without consideration of the processor on which the program will be executed. This allows the program development cycle to be shortened. Programs written in a high-level language are converted into assembly language by other programs known as

compilers or interpreters. A compiler converts the whole program into an executable assembly language program, while an interpreter decodes a single instruction, executes it, and then continues on to the next instruction. For this reason, interpreted languages are generally slow, but are very good for program development. The sophistication of the conversion program will drastically affect the size and efficiency of the assembly language generated, and hence impact the speed that the program will execute. Unfortunately, the speed that the program will execute is very important in real-time systems.

Often, real-time systems take advantage of both high-level and assembly languages. Time critical portions of the software are written in assembly code, while non-critical portions are written in high-level language and compiled into assembly code. This allows software to be developed rapidly, while still meeting timing requirements. The main problem with this approach is integrating the code written in assembly language with that created by the compiler.

Other than the speed and size of the software in a real-time system, other important characteristics of a real-time computer language include the ability to directly control external devices and custom hardware, efficient interrupt handling, and support for system testing.

Examples of compiled languages used in real time systems include BASIC, Fortran, PL/M, Pascal, Modula-2, Ada, and C. BASIC has the advantages that it is easy to learn, simple to

use, and is inexpensive, but it also has the disadvantages that it supports unstructured programming techniques, but does not easily support the direct control of external devices.

Fortran was developed in the mid-1950s as a scientific and engineering applications programming language. It suffers from the same problems as BASIC when applied to real-time systems, but if the real-time system requires extensive numerical processing capability, the disadvantages of Fortran may be overcome by its excellent data processing capabilities.

PL/M is a language developed by Intel for real-time systems using Intel processors. It allows direct access to the input/output of Intel processors, and also has capabilities which improve interrupt handling. PL/M has the disadvantage that it does not support non-Intel processors.

Pascal was developed as a language for teaching structured programming to students. It has little application as a real-time language because it does not allow direct control of the computer hardware.

Modula-2 was created by the designer of Pascal for use as a real-time programming language, and it corrects for the deficiencies that Pascal has in these applications while maintaining the advantage of the strong structure of Pascal.

Ada was developed with the support of the Department of Defense as a language for all its embedded computer systems. An embedded computer system is a computer system which is an integral part of a larger system. Ada has all of the

capabilities required of a real-time computer language, but it has the problem that it is very large and complex, which makes it difficult to learn.

C is a language developed at Bell Labs, and is probably the most popular language in real time control and many other applications at this time. It has the advantages that it allows direct access to the computer hardware, and is usually very efficient when compiled.

The only interpreted language that is used extensively in real time control is Forth. Forth has all the capabilities of a good real-time language, and application programs are generally very compact. Forth's biggest advantage is that since it is a stack-based language, it provides fast context switching. Although interpreted languages tend to be slow, Forth runs quite efficiently, while also providing the program development advantages of other interpreted languages. After program development is finished, the Forth program can be compiled into a assembly language program which can be burned into a read-only memory (ROM) just like compiled languages. Other advantages of Forth are that it supports development of the software on the computer system that will be used in the real-time system, and that software-development packages are inexpensive. The biggest disadvantages of Forth are that the programs developed tend to be cryptic and hard to maintain, and that postfix notation, which is the syntax of the language, tends to be harder to understand than standard

syntax.[Lawrence87]

In a real-time system, there must be tradeoffs made between software and hardware, depending on costs, computation speed required, development time, and other factors. In low-speed applications, functions which are usually performed in hardware in time critical systems are performed in software, and the performance of the software is also sacrificed for budget reasons. Medium-speed applications require system software to be time efficient, with prioritized event handling, but some hardware functions may still be performed in software. In high-speed applications, the required response time is nearly equal to the computer's capacity. All hardware speed improvements are used, and the software must be optimized so that no processing capacity is wasted. High-speed systems are usually exceptionally simple, due to the fact that there is not time to execute anything complicated.

One approach to improving the marriage of hardware and software in a computer system is to tailor the architecture of a processor to a particular high-level language. This allows the programmer to take advantage of ease of programming in a high-level language, while increasing the efficiency of mapping the high-level language instructions into the assembly code of the processor. This has been realized by several design teams, but the results tend to be more complex than is necessary for control applications.[Fraeman86][Golden85] [Hayes87][Hayes86][Jones87][Koopman88][Williams86]

The rest of this document addresses the development of a very simple hardware architecture tailored to execute a core subset of the Forth programming language, with consideration toward use in real-time control applications.

The first three chapters give background information concerning control systems, the Forth programming language, and stack-based architectures. The fourth chapter focuses on the instruction set and architecture developed. The fifth chapter discusses the architectural decisions which were made, and the register transfer level modelling of the system. The final chapter discusses work yet to be accomplished and some packaging considerations and performance trade-offs.

CHAPTER 2

THE FORTH PROGRAMMING LANGUAGE

A Brief History of Forth

Unlike most other languages, Forth is not the creation of a committee or design team, but that of a single person, Charles Moore. During the 1960s, he became frustrated with the time and effort required to write programs in the programming languages which existed at the time, so he created Forth. In doing so, he disregarded most of the conventions of other programming languages and included capabilities which he believed were needed for productive programming. The most important of these is the extensibility of Forth, so that it may be tailored to solving the problem at hand.

In 1971, Moore was hired by the National Radio Astronomy Observatory to program a data-acquisition system for a radio telescope, which he wrote in Forth. This led to the acceptance of Forth as a major applications language by the community of astronomers. By 1973, the demand for Forth had increased such that Moore, in partnership with Elizabeth Rather, formed the company FORTH, Inc., with the goal of expanding Forth applications to various mainframe, mini- and microcomputers.

The popularity of Forth also has spread due to the

availability of public-domain versions of the language. These have been distributed by the Forth Interest Group (FIG), and are available for most computers. FIG also publishes FORTH Dimensions, a bimonthly magazine which addresses the modification and extension of Forth as well as its application to programming problems, and sponsors a conference called the Forth Modification Laboratory, the purpose of which is to allow users and system developers to meet and discuss the development of the language. A counterpart to this conference is the Rochester Forth Application Conference, sponsored by the Institute for Applied Forth Research, Inc.

Forth applications are wide and varied. To name a few, it is used in data acquisition and analysis, expert systems, graphics, medicine, process control, and robotics.

Basic Structure of Forth

The core of Forth is a dictionary or list of approximately 300 operations known as words. These words are essentially subroutines, and can be combined to define other operations, which can then be added to the dictionary and used in more complicated definitions. These definitions can be combined until a single word completely describes and executes a given task. Forth is highly structured, in that every program is a list of these words, each of which is defined in terms of other words, and so on until at the bottom level, everything is defined in terms of the core operations of Forth. This type of code is known as threaded code, and is

efficient in terms of speed and memory.

The use of Forth complements the practice of a top-down design methodology. A problem is broken down into functional blocks, which in turn are broken down further until each function is described in basic steps. Forth allows each of these basic steps to be defined as words, and then combined into functional blocks until the solution of the problem is complete.[McCabe83][Brodie87]

Forth Difficulties

Forth's largest pitfall is the fact that programs written in the language are difficult to read and comprehend, and it often requires special effort and documentation for the program to be understood by anyone, including the original programmer. The main problem concerning the readability of Forth code is that the language is stack-based, which requires that the syntax of the language be postfix, similar to the notation used on Hewlett-Packard calculators.

Forth programs are stored in a non-standard form. Information is stored in mass-storage as numbered blocks of 1,024 characters, which are arranged into 16 lines of 64 characters each. When a program is larger than a block long, special procedures are required to load and execute it. This format was reasonable when Forth was invented, since computer memory was limited and expensive, but these problems do not exist to the scale that they did originally.

Forth also lacks several basic functions which are

standard in other high-level languages. Error checking is virtually nonexistent, and there is no standard way of manipulating strings of data, working with files of data, or using graphics.

The counter-argument to these complaints is that most commercial versions of Forth provide functions which overcome these limitations, and that Forth allows a programmer to easily extend the dictionary of the system to include words which correct for limitations as they are discovered. [Lawrence87]

Forth also uses scaled-integer arithmetic instead of the more standard floating-point arithmetic, and if floating point functions are needed, then those functions need to be written and integrated into the system. Scaled-integer arithmetic is a method of storing numbers in memory without having to keep track of each number's decimal point. All numbers are treated as integers, which are all of the same scale. Examples of scaled-integer and floating-point representations are shown in Table 1.

Table 1. Scaled-Integer Arithmetic

Real-World Value	Scaled-Integer Representation	Floating-Point Representation
1.23	123	123×10^{-2}
10.98	1098	1098×10^{-2}
100.00	10000	1×10^{2}
58.60	5860	586×10^{-1}

The arguments for the use of scaled-integer arithmetic are based on the concept that it is much faster to execute a

calculation in scaled-integer arithmetic than it is in floating-point, and the fact that Forth supports special operators which improve the ease of using scaled-integer math.[Brodie87]

Although all these arguments, both positive and negative, bring up valid points, it must be considered that the ability of Forth to be tailored toward a particular problem, the speed of the programs written, and the compact size of the code may be the deciding factors when choosing a language to write real-time software.

CHAPTER 3

STACK-BASED ARCHITECTURES

Most conventional processors are optimized for office automation systems or computer-aided design applications. The memory management and general-purpose data processing requirements of these applications require design complexity in the processor which is usually not necessary in control applications. Microcontrollers, on the other hand, have been developed for specific applications and do not require the complexity of general-purpose processors, but tend to have much slower instruction execution rates than general-purpose processors, which hinders their use in real-time systems.

High-performance stack-based architectures have been proposed to improve the performance of microcontrollers in real-time applications, citing advantages in algebraic problems and internal addressing schemes. This idea is not new, since stack-based architectures were proposed as early as 1962, when the KDF.9 computer system was designed by the English Electric Company.[Haley62]

Stacks

A stack, also known as a push-down stack, is a one dimensional array used for temporary storage of data. Items are entered and removed from stacks one at a time, such that

the last item placed on the stack is the first one to be removed. This method of storing and removing data on the stack is generally known as a last in, first out (LIFO) method. The process of adding an item to a stack is called a push, and the act of removing an item from the stack is called a pop. A common example of the operation of a stack is that of a rifle magazine, where a spring is used to keep the ammunition at the top, and the last shell to be pushed into the magazine is the first to leave the magazine, just like the data in a LIFO stack.

Advantages

One advantage of a stack-based architecture is that the addresses of operands are implicit in the stack, which minimizes control overhead. A stack uses only a single pointer register to keep track of accessible data. All arithmetic operations execute on data from the stack and store the result of the operation on the stack. This means that no addresses are required for arithmetic operations, since they are implied by the function.

Stack-based architectures use postfix notation, more commonly known as reverse Polish notation, which is the notation used in Hewlett-Packard calculators. When using postfix notation, the operator is specified only after the operands have been placed on the stack. Stack operations do not depend on the manner in which the operands reach the stack. Operands may be placed on the stack explicitly, or

they may be the result of an earlier operation. The stack serves as a common location where data may be passed between operations or manipulated. Examples of postfix notation are shown in Table 2, where they are compared to the more common algebraic or infix notation.

Table 2. Algebraic and Postfix Notation

Algebraic Notation	Postfix Notation
4 + 8	4 8 +
21 / 7	21 7 /
5 * (2 + 7)	5 2 7 + *
(5 - (6 * 7))/8	5 6 7 * - 8 /

Keeping track of the order and magnitude of stack contents requires that the programmer pay attention to what is happening in the program, but overall, postfix notation is often easier to work with than infix notation.

Compilers also use postfix notation to express high-level language calculations as an intermediate step in translation to machine language, so the use of a stack-based architecture removes some complexity from the compiler.[Kelly86]

A stack-based architecture also has an important advantage in the event of an interrupt. In a conventional processor, when an interrupt occurs, the program counter and the contents of all registers in the processor must be saved before the interrupt can be serviced. In a stack-based processor, all registers are already saved on the stack, so all that has to be preserved is the program counter before entering the interrupt service routine.

There must be some consideration taken with respect to how the program counter is saved in the occurrence of an interrupt and also on subroutine calls. In a conventional processor the program counter is saved on a memory stack and then popped back into the program counter register on the return from subroutine. Stack-based processors take the same approach, but the program counter is saved on a second hardware stack usually called the return stack, which is used for saving the return address for subroutines and interrupts and for temporary storage of data in normal processing operations.

Special Requirements

The use of a stack-based architecture and postfix notation require some special data manipulation instructions which have no equivalent counterparts in other systems. In a calculation it may be necessary to reverse the order of the top two locations in the stack, to duplicate the top location in the stack, or to discard the top location in the stack, and instructions must be defined to accomplish these operations.[Haley62]

Other Stack-Based Designs

Stack-based designs generally focus on tailoring their architecture to the high-level language Forth, since it is stack-based and therefore directly is supported by the architecture. Recent examples of stack-based architectures include the Johns Hopkins University FRISC machines

[Fraeman86][Hayes87][Hayes86][Williams86], the Novix machines[Golden85][Ting86], and the RTX machines, which use the FORCE architecture[Jones87][Koopman88]. All of these machines support a very fast subroutine call and return, since almost every Forth word is a subroutine. Of these machines the Novix is the simplest, with only the datapath and control on the chip, with stacks, program, and data memory off-chip. The Novix processor allows only 256 word stacks, and expects the programmer to watch for overflow problems. One feature in the Novix processor which improves the throughput of the system is the return from subroutine. A flag bit in the op-code of the last instruction in a subroutine indicates the return, and the return is executed concurrently with the instruction, effectively reducing overhead to zero.

The FRISC (Forth Reduced Instruction Set Computer) machines are set apart from the other architectures by the stack caching implemented in the chip. The caching algorithm implemented in FRISC 3 uses a ring buffer for the on-chip stack cache. A stack pointer is used to indicate position in the ring, and two sliding pointers mark the overflow and underflow points on the ring. If the stack pointer reaches the overflow or underflow point, then data is written to or read from external memory, and the overflow or underflow point is adjusted accordingly. The problem with caching in a real-time system is that the cache introduces uncertainty in critical timing paths. If the cache has to be serviced during

a critical operation it may cause the system to fail.

The RTX (Real-Time Express) machines use an architecture which was called the FORCE (Forth Optimized RISC Computing Engine) architecture when it was designed, but the name was changed to avoid confusion with the FORCE computer company. These processors use a building block approach to systems. They supply the FORCE processor core and a library of macrocells which can be interfaced on a single VLSI chip to create a system. This library includes timers, interrupt controllers, clock generators, I/O controllers, stack controllers, RAM, and a 16 X 16 bit multiplier. This approach allows each system to be adapted for a particular application. The RTX machines also support the same type of return from subroutine as the Novix processors, which results in zero overhead for returns.

Each of these systems has its advantages, but they appear to be more complex than is necessary for high-speed control applications.

CHAPTER 4

THE FTCP

As was mentioned in previous chapters, the key to high-speed controllers is often extreme simplicity. With this in mind, a limited instruction set which is a core subset of the Forth programming language was chosen as the executable instructions for a very simple architecture, with a few modifications to support the needs of control applications.

The FTCP Instruction Set

A large portion of the FTCP instruction set was taken from the instruction set of the Bridger SIMD processor array[Winters88]. Additions to that instruction set include the greater than, less than, and equal comparison instructions, the interrupt control instructions, and the external data storage instructions. The instruction set, which is composed of 27 instructions, is shown in Figure 1.

Figure 1. The FTCP Instruction Set
(continued on next page)

FTCP Instruction Set
Revision 6
11/5/91

Stack Functions		
DUP	(n - - n n)	Duplicates the top word of the data stack.
DROP	(n - -)	Discards the top word of the data stack.
SWAP	(n1 n2 - - n2 n1)	Reverses the order of the top two stack words.

Figure 1. (cont.) The FTCP Instruction Set

(continued on next page)

>R	(n - -)(- - r)	Pops the top word of the data stack onto the return stack.
R>	(r - -)(- - n)	Pops the top word of the return stack onto the data stack.
Arithmetic Functions		
+	(n1 n2 - - n1+n2)	Adds the top two words of the data stack.
-	(n1 n2 - - n1-n2)	Subtracts the top word of the data stack from the word below it in the data stack.
2*	(n - - 2*n)	Multiplies the top word of the data stack by 2. (Note: The least significant bit of the word is replaced with a 0)
2/	(n - - n/2)	Divides the top word of the data stack by 2. (Note: The most significant bit of the word is fed back into the register so that the sign of the number is retained)
SHIFTR	(n - - n)	Shifts the top word of the data stack right one bit. (Note: The leftmost bit of the word is replaced with a 0)
Boolean Functions		
NOT	(n - - n')	Performs one's complement on the top word of the data stack.
NAND	(n1 n2 - - n)	Performs bitwise NAND of the top two words of the data stack.
XOR	(n1 n2 - - n)	Performs bitwise XOR of the top two words of the data stack.
Comparisons		
>	(n1 n2 - - n)	Pushes -1 on the top of the data stack if n1>n2. If the condition is false a 0 is pushed on the top of the data stack.
<	(n1 n2 - - n)	Pushes -1 on the top of the data stack if n1<n2. If the condition is false a 0 is pushed on the top of the data stack.
=	(n1 n2 - - n)	Pushes -1 on the top of the data stack if n1=n2. If the condition is false a 0 is pushed on the top of the data stack.
Control Functions		
CALL	(- - r)	Subroutine call performed by pushing the program counter on the top of the return stack and placing the subroutine address in the program counter.

Figure 1. (cont.) The FTCP Instruction Set

IF	(n - -)	Examines the word on the top of the data stack and if it is a 0 (False), performs a relative jump to the address of the instruction which begins the next program module. If the top word of the data stack is non-zero (true) then the instructions following the IF instruction are executed. (Note: the IF instruction must be followed by a NOP instruction due to the instruction pipeline.)
RETURN	(r - -)	Return from subroutine by popping the top word of the return stack into the program counter register and resuming execution at that address
DO	(r - - r)	Begin a loop, using the top word of the return stack to count the number of times the loop has been executed.
LOOP	(- -)	Performs a relative jump to the address of the beginning of the loop if the loop counter is non-zero. (Note: The LOOP instruction must be followed by a NOP instruction due to the instruction pipeline.)
EI	(- -)	Enable Interrupts
DI	(- -)	Disable Interrupts
NOP	(- -)	No operation.
Load, Store and I/O Operations		
!	(n - -)	Pop the top word off the data stack and write it to random access memory or to I/O device. (2 clock cycles)
@	(- - n)	Fetch a word from random access memory or I/O device and push it onto the data stack. (2 clock cycles)
ENTER	(- - n)	Pushes data from the instruction bus onto the data stack. (2 clock cycles)

The instruction set shown in Figure 1 has the following format: instruction mnemonic, stack operations, and instruction description. The stack operations are shown in the form (contents of stack -- results of operation). All instructions execute in a single clock cycle, with the following exceptions: !, @, ENTER. The reason that the !

(Store) instruction and the @ (Fetch) instruction require two clock cycles to execute is due to the common address bus used to address program and data memory. The common address bus was introduced to reduce the number of necessary I/O pins when the design is fabricated. If separate address busses are introduced, the ! and @ instructions could execute in a single clock cycle. The ! and @ instructions are also used to write and read data from I/O devices, since these devices are intended to be memory mapped. The ENTER instruction requires two clock cycles because the instruction tells the processor that the next word in program memory is a constant which needs to be pushed onto the stack.

Since most Forth words are subroutines, it is critical that a fast subroutine call be supported. In the FTCP, subroutine calls require only one clock cycle to execute, because the subroutine address is embedded in the call instruction in the manner shown in Figure 2. The most significant bit in the instruction indicates whether the instruction is a subroutine call or any other instruction. If the most significant bit is a 1, then the instruction is a subroutine call, and the rest of the bits in the instruction are the address of the subroutine. If the most significant bit of the instruction is a 0, then the instruction is not a subroutine call.

Figure 2. FTCP Call Instruction

1AAA AAAA AAAA AAAA

One of the problems with embedding the subroutine address in the op-code of the instruction is the fact that this cuts the address space of the processor in half. In the FTCP, by indicating subroutines with the most significant bit of the op-code, the main program must reside in the lower 32 kilobytes (K) of program memory, while subroutines are restricted to the upper 32K of program memory.

The return from subroutine is a simple matter of popping the return address off the return stack and into the program counter. In other processors of this type, this process is accomplished concurrently with the final instruction of the subroutine, but in the FTCP, this process requires one clock cycle.

The data memory address for the ! and @ instructions is embedded in the instructions themselves. This limits the amount of data addressable to 8K. The form of the ! and @ op-codes is shown in Figure 3. The most significant bit of the instruction is a 0, indicating that it is not a subroutine call, the next bit is a 1, indicating a memory access, the third bit indicates whether the instruction is ! or @, and the remaining 13 bits contain the address.

Figure 3. ! and @ Instructions

! Instruction

010A AAAA AAAA AAAA

@ Instruction

011A AAAA AAAA AAAA

The IF and LOOP instruction use the concept of program counter-relative branches. This type of branch is used because often the branch target is close to the current instruction, and it would require less bits to specify the next address relative to the program counter than specifying the address explicitly. In a study of three representative programs, TeX, Spice and GCC, it was found that 8 offset bits accommodated 94% of all branches, 9 offset bits accommodated 97% of all branches, 10 offset bits accommodated 99% of all branches, and 11 offset bits accommodated all branches.[Hennessy90] With this in mind, the IF and LOOP op-code formats (Figure 4) were developed. The first three bits of the op-code are zero, indicating that the op-code is not a subroutine call, and that there is also no memory access. The fourth bit in the op-code is a 1, indicating that the instruction is an IF or a LOOP, and the fifth bit indicates the exact instruction. The remaining 11 bits are the offset address.

The IF and LOOP instructions must be followed by a NOP instruction due to the instruction pipeline. The problem here is that the instruction which will follow the IF or LOOP is not known until the IF or LOOP is executed. In the meantime, the instruction directly following the IF or LOOP in the program memory is being decoded in the instruction pipeline. To prevent errors, this instruction must be a NOP.

Figure 4. IF and LOOP Instructions

IF Instruction

0001 1JJJ JJJJ JJJJ

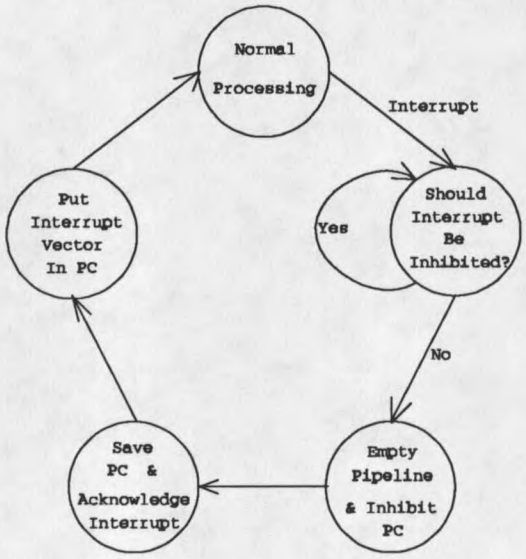
LOOP Instruction

0001 0JJJ JJJJ JJJJ

Interrupt Handling

An important part of computer-based control systems is the manner in which interrupts are handled. The way that external devices indicate to the processor that they need service is to generate an interrupt, and in a real-time system, the lag time to the interrupt service routine is critical.

Figure 5. Interrupt Finite State Automata



To handle interrupts in the FTCP, a small finite state machine was designed. The finite state automata for the interrupt handler is shown in Figure 5. The normal processing mode at the top center is the state in which the processor will reside a large percentage of the time. In the event of an interrupt, the processor will move clockwise to the next state, which determines if the interrupt handler will be allowed to progress to the next state. The reason for this state is that the interrupt handler uses the offset circuitry, and if the next instruction in the pipeline is an IF or a LOOP there will be a conflict. If this is the case, the interrupt handler will wait one clock cycle and then proceed. The next step in the automata is a state which inhibits the program counter, so that a valid return address can be saved on the return stack, and which empties the instruction pipeline. The fourth state saves the program counter on the return stack and acknowledges the interrupt, and the fifth state loads a vector which is the address of the interrupt service routine into the program counter and then the processor returns to normal processing. The return from the interrupt service routine is exactly like a return from subroutine.

The timing diagrams for a non-inhibited interrupt and for an inhibited interrupt are shown in Figures 6 and 7 respectively.

Figure 6. Uninhibited Interrupt Timing

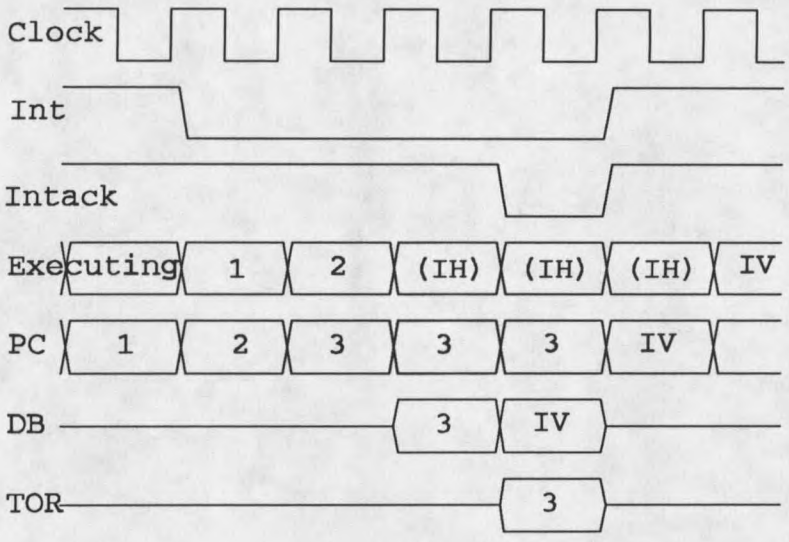
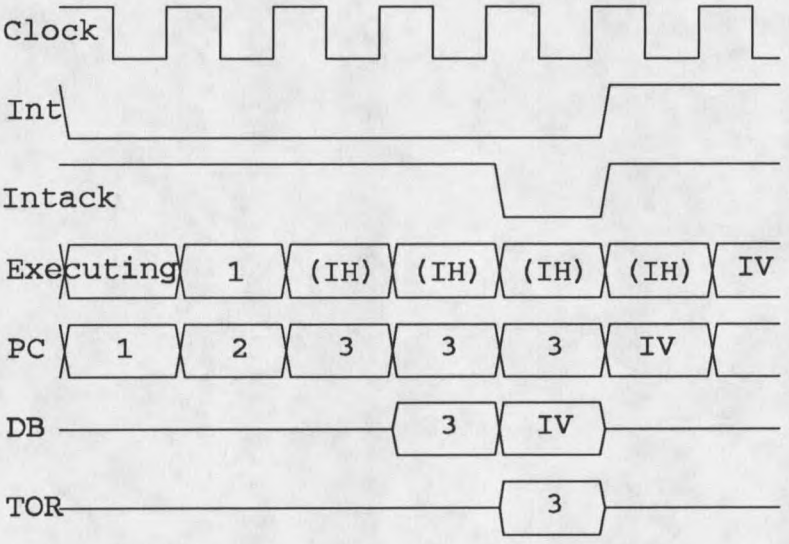


Figure 7. Inhibited Interrupt Timing



In Figure 6 it can be seen that the interrupt line goes active (low) while instruction 1 is executing. It is also determined during that clock cycle that the next instruction in the pipeline, instruction 2 is not an IF or LOOP instruction, so the process is not inhibited, and on the next

clock cycle instruction 2 is executed and the program counter is inhibited. It can be seen that the program counter contains the address of instruction 3 until the interrupt vector is latched into it. The processor then enters the interrupt handler, which pushes the program counter, still containing the address of instruction 3, onto the return stack, acknowledges the interrupt and latches the interrupt vector into the program counter, at which time normal processing begins again.

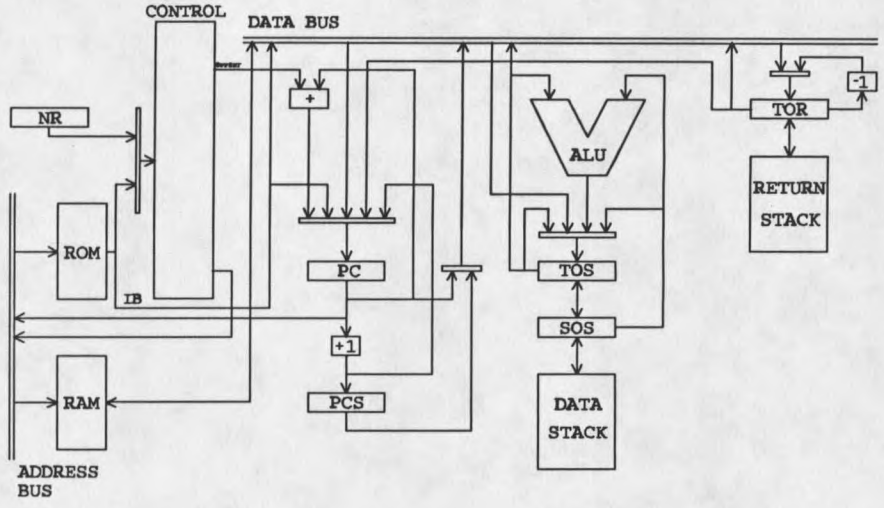
Figure 7 shows the timing diagram for the case in which the interrupt handler must be inhibited. The interrupt line goes active, and it is determined that instruction 1 is an IF or a LOOP instruction. Instruction 1 is executed and then the interrupt handler takes over and waits one clock cycle before taking action. If one looks closely at the diagram in Figure 7 it is apparent that instruction 2 is not executed or saved, but appears to be lost. This is true, but has no effect on the operation, because it has already been defined that IF and LOOP instructions have to be followed by a NOP, which is an instruction which does nothing but increment the program counter.

The FTCP Processor Architecture

The processor architecture of the FTCP is very simple. It is designed so that the results of any operation are latched into registers on the rising edge of the system clock. This fact should increase the ease of programming and the

speed of the processor. A block diagram of the architecture of the FTCP is shown in Figure 8.

Figure 8. The FTCP Processor Architecture



The outputs of the control block are latched into a register on the rising edge of the system clock. All control to the rest of the system comes from this control register, with the following exceptions: the input to the program counter is controlled by the logic which feeds the control register when the instruction in the control logic is either a subroutine call or a return from subroutine. This scheme allows the processor to be pipelined one stage and still be able to execute subroutine calls and returns from subroutines in a single clock cycle. The way that this works is that the control logic senses that the instruction in the logic is a subroutine call or a return, and asynchronously adjusts the program counter input accordingly.

The control logic extracts the data memory address from the ! and @ instructions and connects that address to the address bus when these instructions are executed. The control logic also extracts the offset address from the IF and LOOP instructions and provides it to the logic which calculates the next program counter when these instructions are executed. This offset would be calculated by the compiler and encoded in two's complement form. The control logic will sign extend the address to the width of the program counter to ensure proper branching.

There is a common address bus which is used to address program memory, data memory and all I/O devices. The ROM and RAM shown in the block diagram are part of the system but would probably be external to the chip. The RAM memory which would make up the two stacks would also be external to the chip, depending on the size of the stacks. The top two registers of the stack (TOS and SOS) and the top register of the return stack (TOR) would reside on the chip, and pointer registers controlling each stack would also be present. If the depth of the stacks was limited, it might be possible to put the RAM memory for the rest of the stack on the same die as the processor, which would increase the performance of the system.

The NR register shown feeding the control logic is a register which contains the op-code for a NOP, and it is used with the !, @ and ENTER instructions to control the second

clock cycle of those instructions.

The next program counter (PC) can come from one of five sources. It can be fed by an incrementer, the top of the return stack, the data bus, the instruction bus and the offset logic. Normal program flow causes the PC to be fed by the incrementer, since the processor is just stepping through program memory. The PC is fed by the top of the return stack on return from subroutines. The data bus selection is used when an interrupt vector is coming in from an external device. Subroutine calls cause the instruction bus to be selected, and the offset logic option is used in conjunction with IF and LOOP instructions.

The PCS register is a register which contains a copy of the current program counter. There are two cases in which this register does not contain a copy of the program counter: a subroutine call and an interrupt. In the event of a subroutine call, the PCS register provides the return address to be pushed on the top of the return stack since the program counter contains the subroutine address. A subroutine call uses the data bus to connect the PCS register and the top of the return stack. When the program counter is inhibited in the interrupt handler, the PCS register contains the one more than the program counter. In this case, since the program counter contains the proper return address, it is connected to the data bus and pushed on the top of the return stack.

The main part of the datapath consists of the arithmetic

and logic unit (ALU) and the top two registers of the data stack. The ALU operates on the data in the top two registers of the data stack and stores the result in the top register of the data stack. All arithmetic instructions are performed in twos complement arithmetic, and since the datapath is 16 bits wide, the range of data is limited from -32768 to 32767. The top of the data stack can also be fed by the data bus, which allows external input to the stack and input from the top of the return stack. The top of the data stack may be fed back for the DUP instruction, or the second word of the data stack may be input on execution of the SWAP instruction.

The main function of the return stack is the storage of return addresses for subroutines, but it is also used for temporary data storage and as a down-counter in the LOOP instruction. The advantage of using the top of the return stack as a loop counter is that in the event of an interrupt, the position in the loop is automatically saved, and the processor can easily pick up where it left off after servicing the subroutine.

Although this instruction set and architecture may seem very simple, often the simplest solution is the best solution.

CHAPTER 5

DESIGN METHODOLOGY

When developing a complex system of any sort, it is important to follow a good design methodology. The FTCP was developed using a fairly structured approach which focused on top-down design. The first step in this procedure was to research the type of problems which were going to be addressed and the approaches to these problems which were taken by others. The problems, in this case, were computer-based real-time control systems. This research helped to define the operations which are required by real-time control systems, and it also defined the operations which are not necessary in this type of system.

Once the necessary operations were defined, an instruction set and an architecture were developed in parallel which would satisfy the defined requirements, while complementing each other to improve performance.

Architectural Decisions

Presently, a major portion of the embedded systems market is held by processors with an 8 bit datapath width. Other processors which are trying to break in on this market tend to have a 32 bit wide datapath. Looking at this seems to indicate that there is a market for a processor for

applications which require more power than the 8 bit processors can provide, but do not need the processing power that the 32 bit machines can provide. After considering these facts, in conjunction with the fact that reasonably priced analog-to-digital converters are usually limited to 16 bits, a datapath width for the FTCP of 16 bits was chosen.

Caching was not included in the design of the FTCP. The reasons for this include the fact that the introduction of caching introduces uncertainty in critical timing paths, and that the hardware necessary to support caching would result in a significant increase in the size of the system, given the simplicity of the system.

Register Transfer Level Modelling

After defining the architecture and the instruction set, register transfer level (RTL) modelling of each instruction was performed. This description of the operations is not exceptionally detailed, but is an adequate interface to a hardware description language. The RTL description of the operation of the FTCP is shown in Figure 9.

Figure 9. The FTCP RTL Description
(continued on next page)

FTCP RTL Description
Revision 4
11/6/91

Description

This file contains a Register Transfer Language (RTL) description of each instruction in the FTCP instruction set, and will be the basis of the BDS description of the control logic of the FTCP.

Notes

The outputs of the control logic are latched into a register on the rising edge of the system clock, and all control to the rest of the chip comes from this control register, with the following exceptions: Subroutine calls are accomplished by routing the instruction bus directly to the program counter multiplexer and latching

Figure 9. (cont.) The FTCP RTL Description (continued on next page

the subroutine address into the program counter at the same time as the subroutine call reaches the control register, and return from subroutines are accomplished by selecting the top of return stack on the program counter multiplexer and latching the return address into the program counter at the same time as the return reaches the control register.

It is assumed that all other transfers into registers occur on the rising edge of the system clock. This should increase the speed and the ease of programming of the system.

Unless otherwise indicated in the RTL description, the program counter is incremented on each rising edge of the system clock. There is also a program counter save register, which contains a copy of the program counter, and is used in subroutine calls, because it contains a copy of the return address to be pushed onto the return stack when a subroutine call is executed. It is necessary due to the asynchronous way that the subroutine address is routed to the program counter. There is a program counter offset which comes from the control logic and is used to calculate the next program counter in IF and LOOP instructions. This offset is calculated by the compiler and encoded in two's complement form. The offset is embedded in the instruction, and is sign extended in the logic when it is extracted from the instruction.

There are pointers for control of the data stack (SP) and the return stack (RSP). These registers are used to sense overflow and underflow of the stacks. When there is a data stack or return stack push, these registers are incremented, and when there is a data stack or return stack pop, these registers are decremented.

The address bus is used to address both program and data memory to save pins on the chip.

The data memory address for store and fetch is embedded in the op-code for the instruction. The address is separated from the instruction in the control logic and is routed to the data memory when the instruction is executed. The reason that these instructions require 2 clock cycles to execute is due to the common address bus. The store and fetch instructions are also used to access I/O devices, since they will be memory mapped.

It is required that the IF and LOOP instructions be followed by a NOP instruction due to the instruction pipeline and the fact that the next instruction is not known until the IF or LOOP instruction is executed.

There are several control input/output pins included in this RTL description of the FTCP. The logic state of the pin is included in parentheses following the pin name, ie. PIN(state), with the state indicated in the following manner: 1 - high, and 0 - low. If the state of these pins is not specified, assume that they are in an inactive state. The output pins come from the control register and change state on the rising edge of the system clock.

Interrupts are basically a subroutine call in which the subroutine address is placed on the data bus by the interrupting device, and then the address on the data bus is latched into the program counter. To handle this, a simple state machine was designed, internal to the control logic, which controls the processor from the time when the interrupt input goes active, until the processor enters the interrupt service routine. This state machine ensures that IF and LOOP instructions finish executing before allowing the interrupt service process to begin. Failure to do this will result in an incorrect return address, and program failure. There is an interrupt status flipflop in the control logic, which indicates whether interrupts are enabled or disabled.

Notation Dictionary

TOS	Top of Data Stack
SOS	Second Word of Data Stack
TOR	Top of Return Stack
SP	Stack Pointer
RSP	Return Stack Pointer
DB	Data Bus
IB	Instruction Bus
ADRB	Memory Address Bus
CL	Control Logic
PC	Program Counter
PCS	Program Counter Save

Figure 9. (cont.) The FTCP RTL Description
(continued on next page

OS	Program Counter Offset (encoded in 2's complement)
NR	No Operation (NOP) Register
IS	Interrupt Status Flipflop (active high)
INT	Interrupt Input (active low)
INTACK	Interrupt Acknowledge (active low)
STATE	State of the Interrupt State Machine (normal processing is 000)
RD/WR'	Read/Write pin (read active high, write active low)
MEMRQ	Memory Request (active low)
RESET	Reset Input (active low)
ROM	Program Memory
RAM	Data Memory
↑S	Data Stack Pop
↓S	Data Stack Push
↑R	Return Stack Pop
↓R	Return Stack Push
→	Transfer
$X_{subscript}$	Indication of Specific Bits
Stack Functions	
DUP	TOS → SOS; TOS → TOS; ↓S
DROP	TOS → DB; ↑S
SWAP	TOS → SOS; SOS → TOS
>R	TOS → TOR; ↑S; ↓R
R>	TOR → TOS; ↓S; ↑R
Arithmetic Functions	
+	TOS + SOS → TOS; ↑S
-	SOS - TOS → TOS; ↑S
2*	$TOS_{15} \rightarrow TOS_{15}$; $TOS_{13-0} \rightarrow TOS_{14-1}$; $0 \rightarrow TOS_0$
2/	$TOS_{15} \rightarrow TOS_{15}$; $TOS_{15-1} \rightarrow TOS_{14-0}$
SHIFTR	$TOS_{15-1} \rightarrow TOS_{14-0}$; $0 \rightarrow TOS_{15}$
Boolean Functions	
NOT	TOS' → TOS
NAND	TOS nand SOS → TOS; ↑S
XOR	TOS xor SOS → TOS; ↑S
Comparisons	
>	SOS > TOS True: -1 → TOS; ↑S False: 0 → TOS; ↑S
<	SOS < TOS True: -1 → TOS; ↑S False: 0 → TOS; ↑S

Figure 9. (cont.) The FTCP RTL Description

=	SOS = TOS True: -1 → TOS; ↑S False: 0 → TOS; ↑
Control Functions	
IF	TOS = -1 PC + 1 → PC; ↑S TOS = 0 PC + OS → PC; ↑S
CALL	PCS → TOR; ↓R; IB → PC
RETURN	TOR → PC; ↑R
DO	TOS → TOR; ↑S; ↓R
LOOP	If TOR = 0 PC + 1 → PC; ↑R If TOR > 0 PC - OS → PC; TOR - 1 → TOR
EI	1 → IS
DI	0 → IS
NOP	PC + 1 → PC
Load,Store and I/O Operations	
!	Cycle 1: NR → CL; TOS → DB; OS = 0; PC + OS → PC; ↑S; MEMRQ(0); RD/WR'(0) Cycle 2: IB → CL
@	Cycle 1: NR → CL; DB → TOS; OS = 0; PC + OS → PC; ↓S; MEMRQ(0); RD/WR'(1) Cycle 2: IB → CL
ENTER	Cycle 1: NR → CL; IB → DB; DB → TOS; S↓; Cycle 2: IB → CL
Interrupt Service	
INT(1)	No Action
INT(0)	If IS = 0 No Action If IS = 1 Cycle 1: 001 → STATE Cycle 2: If IF or LOOP instruction in CL, wait one cycle 001 → STATE Else 010 → STATE Cycle 3: OS = 0; PC + OS → PC; 011 → STATE Cycle 4: PC → TOR; R↓; INTACK(0); 111 → STATE Cycle 5: DB → PC; INTACK(1); 000 → STATE
Reset	
RESET(1)	Normal Operation
RESET(0)	0 → PC; 0 → PCS; 0 → SP; 0 → RSP; 0 → TOS; 0 → SOS; 0 → TOR

The RTL description shown in Figure 9 begins with a notes section which gives information about overall system specifications, instructions which require clarification as to their effects on the operation of the hardware and some conventions followed. This is followed by a section which defines all the notation used to describe different parts of the system and actions of the system. The final section is the actual RTL description of each instruction in the instruction set, the interrupt handler and the effects of a reset. To better understand the RTL description of an instruction, consider the RTL description of the 2* instruction shown in Figure 10.

Figure 10. RTL for the 2* Instruction

2* $\quad TOS_{15} \rightarrow TOS_{15};\ TOS_{13-0} \rightarrow TOS_{14-1};$
$0 \rightarrow TOS_0$

The RTL description of the 2* instruction specifies exactly how the top of the data stack is affected by the instruction. The most significant bit (TOS_{15}) is fed back into itself, so that the sign of the number does not change, the 14 lowest bits are shifted up one place, and a zero is put into the least significant bit (TOS_0).

The next step in the implementation of the design would be to describe the control of the system in a hardware description language like BDS, which is the hardware description language in the Berkeley OCT VLSI toolset. Once

this description is accomplished, the logic necessary to accomplish the control can be generated and tested. The culmination of this process would be a completed VLSI standard-cell implementation of the control logic, which would probably be combined with a custom VLSI datapath. Finally, the inputs and outputs of the core of the chip would be routed to a padring and the design process would be complete. The completed design would then be sent away for fabrication, and tested on return.

CHAPTER 6

CONCLUSIONS

The design presented here is much simpler, both in instruction set and architecture, than other designs in this field, but it still can accomplish the necessary tasks in a real-time control system. An economical solution should not go beyond the scope of the problem, which is what other designs appear to have done. A statement to this effect was made by Charles Moore in his introduction to Leo Brodie's Starting Forth[Brodie87].

> One principle that guided the evolution of Forth, and continues to guide its application, is bluntly: Keep It Simple. A simple solution has elegance. It is the result of exacting effort to understand the real problem and is recognized by its compelling sense of rightness. I stress this point because it contradicts the conventional view that power increases with complexity. Simplicity provides confidence, reliability, compactness, and speed.

Future Work

There is much work left to be accomplished in the implementation and testing stages of the design. The op-code assignment for each instruction needs to be optimized, although an initial op-code assignment has been done and is shown in Appendix A.

The RTL description of the instruction set needs to be ported into a hardware description language to generate the

control logic for the processor. The basic stack controller necessary for this project was designed and tested by Mark Major and Steve Gaub[Major91]. The BDS description of this design is shown in Appendix B. These files will be modified slightly so that the exact requirements of the FTCP can be met, and then they will be included in the control logic module. A datapath will also need to be generated for testing purposes, but the final datapath would likely be a custom VLSI implementation.

After the logic was generated, timing constraints for the processor would have to be investigated to determine the maximum clock speed. Analysis of other standard cell libraries may be necessary to provide logic which meets performance and size constraints.

A final step in the design cycle would be to have a prototype processor fabricated and test the physical part in the environment in which it would be operating.

Potential Problems

One of the problems which would have to be addressed before having a prototype fabricated is the fact that although the architecture is very simple, it requires many I/O pins and would probably be pad limited. A large amount of space on the die would be wasted because the padring would require a large perimeter die, but it is unlikely that the control logic and datapath of the processor would require a large portion of that die. One possible solution to this problem would be to

put both the data stack and the return stack RAM on the chip, which would eliminate the pins necessary to address the RAMs and the data pins for each stack. Depending on the amount of space available on the die, it is likely that a balance between the I/O pins and the functionality of the processor could be achieved.

Another issue which could be addressed if the decision to place the stacks on the chip is made is the scheme used to address data memory and I/O. Presently the design has an address bus which is shared between program memory, data memory and I/O. This causes the ! and @ instructions to require two clock cycles to execute. It may be possible to introduce a second address bus for these operations, which would allow them to execute in a single clock cycle.

Performance

Initial performance estimates made from instruction execution frequency data given for the FRISC machines [Hayes87], give a throughput rate of 1.35 instructions per clock cycle. This rate was determined by a 35% occurrence of two clock cycle instructions in the execution frequency data and a 65% occurrence of single clock cycle instructions in the execution frequency data. If a second address bus is introduced for data memory, the occurrence of two clock cycle instructions drops to 17% so the throughput rate increases to 1.17 instructions per clock cycle. These estimates are conservative, since some instructions given in the execution

frequency data correspond to multiple single clock cycle instructions on the FTCP, which alters the percentage of two clock cycle instructions executed in the code analyzed.

REFERENCES CITED

Brodie, L., Starting Forth, Prentice-Hall, 1987.

Burns, A., and Wellings, A., Real-Time Systems and Their Programming Languages, Addison-Wesley,1990.

D'Azzo, J., and Houpis, C., Linear Control System Analysis & Design, McGraw-Hill, 1988.

Fraeman, M., Hayes, J., Williams, R. and Zaremba, T., "A 32 Bit Architecture For Direct Execution of Forth", Proc. of the Eighth FORML Conference, 1986.

Golden, J., Moore, C. and Brodie, L., "Fast processor chip takes its instructions directly from Forth", Electronic Design, March 21, 1985, pp127-138.

Haley, A., "The KDF.9 Computer System", Proc. AFIPS FJCC, vol22, pp108-120, 1962.

Hayes, J., Fraeman, M., Williams, R., and Zaremba, T., "An Architecture for the Direct Execution of the Forth Programming Language", Proc.of ASPLOS-II, Palo-Alto, CA, October 1987.

Hayes, J., "An Interpreter and Object Code Optimizer for a 32 Bit Forth Chip", Proc.of the Eighth FORML Conference, 1986.

Hennessy, J., and Patterson, D., Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990.

Jones, T., Malinowski C., and Zepp, S., "Standard-cell CPU toolkit crafts potent processors", Electronic Design, May 14, 1987, pp93-100.

Kelly, M., and Spies, N., Forth: A Text And Reference, Prentice-Hall, 1986.

Koopman, P., "Writable Instruction Set, Stack Oriented Computers: The WISC Concept", Journal of Forth Application and Research, vol5(1), pp49-71.

Lawrence, P., and Mauch, K., Real-Time Microcomputer System Design: An Introduction, McGraw-Hill, 1987.

Major, M., and Gaub, S., EE 321, Montana State University, Spring 1991.

McCabe, C., Forth Fundamentals, Dilithium Press, 1983.

Savitzky, S., Real-Time Microprocessor Systems, Van Nostrand Reinhold Company, 1985.

Ting, C., Footsteps In An Empty Valley, Offete Enterprises Inc., 1986.

Williams, R., Fraeman, M., Hayes J., and Zaremba, T., "The Development of a VLSI Forth Microprocessor", Proc. of the Eighth FORML Conference, 1986.

Winters, K., "A Mesh-Structured Processor Architecture Based on a Stack-Oriented Instruction Set", Proc. 1988 Rochester Forth Conference on Programming Environments, Rochester, NY, June, 1988.

Young, S., Real Time Languages: Design and Development, Ellis Horwood, 1982.

APPENDICES

APPENDIX A - A SAMPLE OP-CODE ASSIGNMENT

FTCP Op-Code Assignment
Revision 1
11/5/91

Notes

In this document the following designations are used to describe the logic state of the 16 bits used in the op-code assignment for each instruction.

1	Logic High State
0	Logic Low State
A	Address, Either 1 or 0
J	Offset, Either 1 or 0

The op-codes for each instruction are specified in the following manner:

$B_{15}B_{14}B_{13}B_{12}$ $B_{11}B_{10}B_9B_8$ $B_7B_6B_5B_4$ $B_3B_2B_1B_0$

The way that the CALL instruction is designated limits subroutines to the upper 32K of the program memory address space, and the main program to the lower 32K of the program memory address space.

All external I/O devices are memory mapped, and are accessed with the ! and @ instructions. The address for memory operations is embedded in the instruction, and limits data memory to 8K.

The program counter offset is embedded in the instructions which use it to execute a relative jump. 11 bits are allowed for this offset, which according to Hennessy and Patterson covered 100% of relative jumps in the programs that were studied.

Stack Functions

DUP	0000 0000 0000 0001
DROP	0000 0000 0000 0010
SWAP	0000 0000 0000 0011
>R	0000 0000 0000 0100
R>	0000 0000 0000 0101

Arithmetic Functions

+	0000 0000 0000 0111
-	0000 0000 0000 1000
2*	0000 0000 0000 1001
2/	0000 0000 0000 1010
SHIFTR	0000 0000 0000 1011

Boolean Functions

NOT	0000 0000 0000 1100
NAND	0000 0000 0000 1101
XOR	0000 0000 0000 1110

Comparisons

>	0000 0000 0000 1111
<	0000 0000 0001 0000
=	0000 0000 0001 0001

Control Functions

IF	0001 1JJJ JJJJ JJJJ
CALL	1AAA AAAA AAAA AAAA
RETURN	0000 0000 0001 0010
DO	0000 0000 0001 0011
LOOP	0001 0JJJ JJJJ JJJJ
EI	0000 0000 0001 0100
DI	0000 0000 0001 0101
NOP	0000 0000 0000 0000

Load, Store, and I/O Operations

!	010A AAAA AAAA AAAA
@	011A AAAA AAAA AAAA
ENTER	0000 0100 0000 0000

APPENDIX B - BDS DESCRIPTION OF A STACK CONTROLLER

```
!  BDS description of a Stack Controller
!  Mark Major
!  Steve Gaub
!  EE 321 Spring 1991

MODEL  stack

!     DEFINITION OF OUTPUTS

writed<0>,readd<0>,overflowd<0>,underflowd<0>,
resetout<0>,b<15:0>,bufferenable<0>

=

!     DEFINITION OF INPUTS

push<0>,pop<0>,reset<0>,enable<0>,addressq<15:0>,read<0>,write<0>;


!*************************************************************************
!     THIS IS THE SUBROUTINE NOP

ROUTINE  NOP();

!     See if the last clock cycle was a pop, indicating that the
!     address is not at the top of the stack.  If so, get it to the top.

IF read<0> EQL 1 THEN
      BEGIN
      IF addressq<15:0> NEQ 0000#16 THEN  !Make sure stack is not empty
            b<15:0> = FFFF#16;           !Decrement the address
      END;

ENDROUTINE; !END OF THE SUBROUTINE NOP


!*************************************************************************
!     THIS IS THE SUBROUTINE NOT_TOP_OF_STACK
!     This routine will be called to perform a push, pop, or reset if
!     we know that the current address is not pointing to the top of
!     the stack

ROUTINE NOT_TOP_OF_STACK();

!     See if operation is push, pop, or reset and assert correct lines

IF reset<0> EQL 1 THEN
      resetout<0> = 0   ! Reset the address register

ELSE BEGIN
     IF push<0> EQL 1 THEN
      BEGIN
      writed<0> = 1;           ! Write enable the ram
      bufferenable<0> = 1;     ! Connect the address to bus
      END

ELSE BEGIN
     IF pop<0> EQL 1 THEN
      BEGIN
      IF addressq<15:0> EQL 0000#16 THEN  ! Check for empty stack
            underflowd<0> = 1 ! Assert underflow, do nothing
```

```
      ELSE  BEGIN
            bufferenable<0> = 1;     ! Connect that address to bus
            readd = 1;               ! Assert read for the ram
            b<15:0> = FFFF#16;       ! Get address to top of stack
            END;
      END;
    END;
    END;
ENDROUTINE; ! END OF THE SUBROUTINE NOT_TOP_OF_STACK


!*************************************************************************
!     THIS IS THE SUBROUTINE TOP_OF_STACK
!     This routine will be called to perform a push, pop, or reset if
!     we know that the current address is pointing to the top of
!     the stack

ROUTINE TOP_OF_STACK();

IF reset<0> EQL 1 THEN          !Check for a reset condition
      resetout<0> = 0    ! Reset the address register

ELSE BEGIN
     IF push<0> EQL 1 THEN
      BEGIN
      IF addressq<15:0> EQL FFFF#16 THEN !Check for full stack
            overflowd<0> = 1  !Set overflow, do nothing

      ELSE
      BEGIN
      writed<0> = 1;                  !Write enable the ram
      bufferenable<0> = 1;            !Connect address to bus

      IF addressq<15:0> NEQ 0000#16 THEN
            b<15:0> = 0001#16 !Inc address, stack not empty
      ELSE                    !Stack is empty
            IF write<0> EQL 1 THEN
                  b<15:0> = 0001#16;  !Need to fill 00h
      END;
      END

ELSE BEGIN
     IF pop<0> EQL 1 THEN
      BEGIN
      IF addressq<15:0> EQL 0000#16 THEN !Check for empty stack
            underflowd<0> = 1 !Set underflow, do nothing

      ELSE  BEGIN
            bufferenable<0> = 1;     !Connect address to bus
            readd<0> = 1;            !Read enable the ram
            END;
        END;
     END;
     END;
ENDROUTINE; ! END OF THE SUBROUTINE TOP_OF_STACK


!*************************************************************************
!     THIS IS THE MAIN ROUNTINE FOR STACK

ROUTINE  MAIN();
```

```
!     Setting the default values

writed<0>=0; readd<0>=0; overflowd<0>=0; underflowd<0>=0;
resetout<0>=1; b<15:0>=0; bufferenable<0>=0;

!     Check to see if enable is low. If so, do an effective NOP.

IF enable<0> EQL 0 THEN
      NOP()

!     Otherwise, we must perform an operation.  We need to
!     know if the present address is at the top of the stack
!     by checking to see if read is high. Then we will call
!     the appropriate subroutine.

ELSE  IF read<0> EQL 1 THEN
            NOT_TOP_OF_STACK()
      ELSE  TOP_OF_STACK();

ENDROUTINE; !  END OF MAIN ROUTINE

ENDMODEL;   !  END OF BDS FILE
```