The Next Generation Computer, Compiler, Language, and Emulator

Kevin Schaefer
kevin.l.schaefer1394@gmail.com

August 23, 2014

Preface

Thanks for taking time to look at my project. The basic idea behind this project is to do something that I think is pretty cool. I basically want to integrate aspects of Computer Science, Computer Architecture, and Electrical Engineering all together in one project. I have no motivation to do this project except my own desire to do something that I think is cool and fascinating. Please note that this is a work in progress and will be changing as time progresses. This document also serves as my notes to myself, so it is possible that portions of it will not be very clear; sorry in advance for that.

To anybody who is looking for help on one of the topics I touch on, good luck and I hope my project helps.

To anybody who is looking to replicate what I have done, I am flattered and I hope you are successful. My goal is to provide as much documentation as you should need to build it yourself. I hope this project helps your interest in these areas and I would love to hear your feedback.

To anybody who is looking to get out of a homework assignment by copying my work, you won't get very far in life.

To anybody who thinks I am crazy, I don't really care; I'm doing this anyway.

To everyone else, thanks for any interest you have in my project. I'm not sure how much my project will benefit you, but you're welcome to spend as much time reading through any documents or files I have provided or to fork this project and somehow build on it further.

Contents

Chapter 1

Introduction

This paper documents the system being developed by Kevin Schaefer as a learning exercise in computer science and electrical engineering. It consists of three components:

1. Compiler
2. Computer
3. Emulator

The compiler, and associated programming language, is an exercise in writing a complete language and compiler capable of compiling it into machine code suitable for running on the computer or the emulator. To make it more fun, the language will be themed after Star Trek. The high level programming language that will be developed will be similar to the C or C++ programming languages, but will incorporate specific elements for the computer being developed.

The computer is an exercise in computer or electrical engineering. The computer will be developed with as many low-level electrical components as are feasible. This means that the computer will consist of a large number of transistors and possibly logic ICs to prevent the computer from becoming too large. This basically comes down to the fact that building an XOR gate from transistors is pretty straight forward (and also somewhat busy work), but using ICs that implement XOR gates helps eliminate some of the busy work. On the other hand, certain parts of the computer, namely long-term storage, are going to require something other than basic components.

Finally, the emulator will provide a method of testing the compiler and language without having to copy the output onto the already built computer. This allows for faster development, and is also pretty cool.

Depending on how all of this goes, a converter tool might also be developed to convert code written in the custom language to C++, or vise-versa.

Chapter 2

TNG Architecture

2.1 Introduction

This chapter will provide an in-depth explanation of the architecture of the system being developed [1].

This computer uses 4 bit opcodes, with 12 bit operands, for a total word size of 16 bits. Addresses are represented with 12 bits, resulting in $2^12 = 4096$ addressable locations.

The computer maintains a system stack in memory, with the SP pointing to the top of the stack, and an input queue, with the IQP pointing to the front of the queue. All keyboard input is stored sequentially in the input queue, and the system stack is used for general purpose operations.

The computer also maintains an IRQ table in memory which maps interrupts to addresses, specifying the location of the subprocedure to call when various interrupts are fired.

2.2 CPU Registers

This computer is slightly unique in that it is designed with as few registers as possible. This is to make it easier and cheaper to build. Of course the CPU has a few important registers, namely a PC, SP, IR, MCR, IQP, and MBR.

2.2.1 PC

The PC, or Program Counter, stores the address of the currently executing instruction in memory. It is 12 bits wide.

2.2.2 SP

The SP, or Stack Pointer, stores the address of the top of the system stack. It is 12 bits wide.

2.2.3 IR

The IR, or Instruction Register, stores the entire instruction and operand of the currently executing instruction. It is 16 bits wide (4 bit opcode + 12 bit operand).

[1] The instruction set was designed to be able to do a wide range of things, as well as to be easy to implement in circuitry

2.2.4 MCR

The MCR, or Memory Control Register[2] , stores the address of the location in memory to be accessed and the operation to be carried out. The most significant bit represents the memory operation, 0 represents a read and 1 represents a write, and the next 12 bits represent the address. The MCR is 13 bits wide.

2.2.5 IQP

The IQP, or Input Queue Pointer, stores the address of the front of the queue where keyboard input to the system is stored. The IQP is 12 bits wide.

2.2.6 MBR

The MBR, or Memory Buffer Register, stores any data received from memory. The MBR is 16 bits wide.

2.3 Instruction set

This section outlines the instruction set of the computer.

Opcode	Hex	Description
AND	0x1	bitwise AND
OR	0x2	bitwise OR
XOR	0x3	bitwise XOR
INV	0x4	bitwise invert (not)
POP	0x5	pop value off stack
PUSH	0x6	push value onto stack
ADD	0x7	Add top two values on the stack, pushing the result back on the stack
SUB	0x8	Same as ADD, except it subtracts...
MLT	0x9	
DIV	0xA	
JPZ	0xB	Jump if top of stack is zero
JPN	0xC	Jump if top of stack is negative
PRT	0xD	Print message, operand is pointer
RD	0xE	Read data from input buffer
CPU	0xF	CPU command

2.3.1 AND

Usage: AND <addr>

The AND command performs a simple bitwise AND with the top element on the stack and the word pointed to by the operand. The result is pushed onto the top of the stack.

2.3.2 OR

Usage: OR <addr>

The OR command performs a simple bitwise OR with the top element on the stack and the word pointed to by the operand. The result is pushed onto the top of the stack.

[2]It would ordinarily be called the Memory Address Register, but I needed a good way to also send the operation (read or write), so I figured a name change to the Memory Control Register would suffice.

2.3.3 XOR

Usage: XOR <addr>

The XOR command performs a simple bitwise XOR with the top element on the stack and the word pointed to by the operand. The result is pushed onto the top of the stack.

2.3.4 INV

Usage: INV

The INV command performs a simple bitwise NOT with the top element on the stack and pushes the result back onto the stack.

2.3.5 POP

Usage: POP <addr>

The POP command removes the top element from the stack and stores it into the specified address location.

2.3.6 PUSH

Usage: PUSH <addr>

The PUSH command loads a word from the specified memory location and pushes it onto the top of the stack.

2.3.7 ADD

Usage: ADD <addr>

The ADD command loads a word from the specified memory location, adds it to the top element of the stack, and pushes the result onto the stack.

2.3.8 SUB

Usage: SUB <addr>

The SUB command loads a word from the specified memory location, subtracts it from the top element of the stack, and pushes the result onto the stack.

2.3.9 MLT

Usage: MLT <addr>

The MLT command loads a word from the specified memory location, multiplies it with the top element of the stack, and pushes the result onto the stack.

2.3.10 DIV

Usage: DIV <addr>

The DIV command loads a word from the specified memory location, divides it by the top element of the stack, and pushes the result onto the stack.

2.3.11 JPZ

Usage: JPZ <addr>[3]

The JPZ command sets the Program Counter to the specified address if the top element of the stack is 0x0000.

2.3.12 JPN

Usage: JPN <addr>

The JPN command sets the Program Counter to the specified address if the top element of the stack is negative.

2.3.13 PRT

Usage: PRT <addr>

The PRT command writes data, starting at the specified address until it encounters 0x0000, to the display buffer.

2.3.14 RD

Usage: RD <addr>

The RD command reads bytes from the system input buffer, pushing them onto the stack, then storing the count into the address specified as an operand.

2.3.15 CPU

Usage: CPU <cpu_instruction>

The CPU command is used to access special extended CPU instructions.

2.4 Extended CPU operations

This section describes the various extended operations the CPU can perform. These mainly are designed to access special features of the CPU, not something a standard user-space program would need access to. Here are the Extended CPU operations:

Full Opcode	**Description**
0xF000	Set the address of the IHR Register to the top element on the stack
0xF001	Store the contents of the IHR onto the top of the stack
0xF010	Disables system interrupts
0xF011	Enables system interrupts
0xF012	Reads the system interrupt enable flag, and stores either 0xFFFF onto the top of the stack, or 0x0000 onto the top of the stack
0xF100	not implemented yet...

2.5 Keyboard Input

A computer without any means of keyboard input would be rather boring, so we are designing it to work with a standard keyboard. Every key typed will be stored in the Input Queue. Each keypress generates a system interrupt, so the inputted data can be processed.

[3] A note on going to a specific address, you will have to force JPZ or JPN to jump in order to achieve the same result

2.6 System Interrupts

System interrupts are used to signal special events to the operating system. Interrupts are signaled by bringing one of four available interrupt request (IRQ0 - IRQ3) lines to the active state. When the processor finishes a complete instruction cycle, it polls the four IRQ lines. If one or more lines are active, it invokes the First Level Interrupt Handler[4]. When this occurs, the CPU performs a context switch, and loading the address of the FLIH via the IHR Register. The FLIH's job is simply to queue the interrupt request and to notify the requestor that its request has been received. The FLIH, which will be defined by the OS, pushes the IRQ onto the Interrupt Request Queue in main memory. It then signals an IRR (Interrupt Request Received) to the appropriate requestor. The requestor must then bring its associated interrupt request line to the inactive state as it waits to be serviced.

After the FLIH saves off the interrupt request to memory, the CPU can switch back to another process. At a later time, the OS can schedule a Second Level Interrupt Handler to service a waiting interrupt request from the Interrupt Request Queue.

When the OS schedules a Second Level Interrupt Handler, it loads the IRQ number (0-3) from the queue, and then looks up the address of the appropriate interrupt handler in the interrupt vector table. The specified interrupt request is popped off the front of the Interrupt Request Queue, and the interrupt handler is invoked. The interrupt handler is then responsible for alerting the appropriate requestor that its interrupt request is being serviced. The requestor can then send the appropriate data via the system bus to the CPU for processing by the interrupt handler.

IRQ #	**Description**
0x0	Exception
0x1	System Timer
0x2	Keyboard key received
0x3	Reserved

2.7 Input and Output

2.7.1 Keyboard

2.7.2 Video output

[4]See IHR Register

Chapter 3

TNG Operating System

Under construction

3.1 Introduction

3.2 Privileged Execution

3.3 Interrupts

3.3.1 Interrupt Vector Table

3.3.2 First Level Interrupt Handler

3.3.3 Second Level Interrupt Handlers

3.4 System Timer

3.5 Processes

3.5.1 Process Control Block

3.5.2 Process Scheduling

3.5.3 Process Execution

Chapter 4

TNG Language Specification

4.1 Introduction

Extension	Description
.tng	Source code file
.st	Header file

4.2 Basic File Structure

4.2.1 Credits File Section

4.2.2 Code File Section

4.2.3 Other File Sections

4.3 Data Types

4.3.1 Basic Data Types

4.3.2 Custom Data Types

4.4 Operators

4.4.1 Overview

4.4.2 Operator Precedence

4.4.3 Operator Associativity

4.5 Functions

4.5.1 Parameters

4.5.2 Return Types

4.5.3 Function Calling

4.6 Classes

4.6.1 Constructors

4.6.2 Destructors

4.6.3 Properties

4.6.4 Methods

4.6.5 Access Control

Private

Public

4.7 Flow Control

4.7.1 Branching Statements

If Statement

If-else Statement

If-else-if Statement

4.7.2 Looping Statements

For loop

Do while loop

While loop

4.7.3 Expressions

4.7.4 Variables

Variable Scope

4.7.5 Literals

Character

```
**Section: Credits
  Author: Kevin Schaefer
  Date: 8-20-2014
  Package: Samples
  Email: kevin.l.schaefer1394@gmail.com
**EndSection: Credits
**Section: Code

* This is the main code section in the file

* Define a sub procedure with a parameter of type int, which returns nothing
episode printHello(int param1) space
{
  * Print a message to the screen
  write("Welcome, the parameter is ", param1);

  * Return nothing
  return space;
}

* The main entry point of the program
episode enterprise() space
{
  * Call a sub procedure
  warp printHello(1);

  * Return nothing
  return space;
}

**EndSection: Code
```

Chapter 5

TNG Language Compiler

Under construction

Chapter 6

System Emulator

Under construction