An Interactive Gate-Level Simulator of a Classical Von Neumann Architecture, as an Educational Aid for Introducing Novices to the Fundamentals of Computer Organization

Gabriel Robins

Computer Science Department
University of California, Los Angeles
Los Angeles, California 90025

1. Abstract

I have developed an interactive tool for the simulation of a classical Von Neumann computer architecture. The simulation takes place at the register, bus, and gate level. The simulated system consists of 9 registers, 4 buses, 40 gates, an adder, a memory, a micro-programmed control subsystem, a 3-phase clock, a "scratch" register, logical inverters, a bi-directional shift register, several constant registers, and zero-detect logic. A friendly user interface was also implemented, featuring an assembler, a microcode interpreter, and a terminal-independent full-screen display facility. My simulator prototype could effectively be used as an educational tool for the introduction of novices to the fundamentals of computer organization. Alternatively, the construction of such a simulator may in itself constitute a good term project for an upper division hardware course.

Keywords: Computer organization, simulation, learning tools, computer hardware, educational aids, user training systems.

2. Introduction

We have developed an interactive tool for the simulation of a classical von Neumann computer architecture. The simulation takes place at the register, bus, and gate level. The components of our system include 9 registers, 4 buses, 40 gates, 1 adder, a memory, a micro-programmed control subsystem, a 3-phase clock, an extra "scratch" register, logical inverters, a bi-directional shift register, several constant registers, and zero-detect logic. In addition, we have constructed a friendly user interface, featuring an assembler, a microcode interpreter, and a terminal-independent full-screen display facility.

There exists a distinct lack of software tools to aid and enhance the teaching of computer science at the undergraduate level. We believe that our interactive simulator prototype constitutes an extremely useful educational tool for the introduction of novices to the fundamentals of computer organization. The architecture we consider is based on the one discussed in [Tanenbaum].

3. Overview

This simulator requires 3 specification: the micro-code, the assembly instruction set, and the user program. When the simulator starts running, it loads the micro-program into the micro-store; next, it reads and assembles the user program into machine language, according to the instruction set specified (or else the default assembly instruction set). The resulting machine program is loaded into the main memory of the simulator. The simulator then begins to execute the micro-program; the micro-program, in turn, fetches, decodes, and executes instructions of the machine-language program.

By programming the simulator in micro-code, the user may thus create new and novel "instruction sets" for the "machine." For example, suppose the user wanted to add an assembly instruction "sqrt" which takes the integer square-root of the ACC register and leaves the result in the ACC register. The user will then need to add a new opcode called "sqrt" (and a corresponding machine-instruction code) to the assembly instruction set of the machine (by updating that file), and next modify the micro-program to perform the square root operation on the ACC register whenever the new instruction is encountered.

The organization of the rest of this paper is as follows: section 4 describes the details of the simulated hardware, section 5 describes the assembler and the assembly language, section 6 describes the microcode interpreter and its language, section 7 discusses the user interface, and section 8 summarizes the implementation and explains how to obtain the source code.

4. The Hardware

The computer system we chose to simulate is a simplified von Neumann-type single-processor micro-program controlled machine. The schematic organization of this system is given in Appendix II. A detailed description of the components and topology of the system follows. Unless otherwise specified, when two registers/buses with different numbers of bits are connected, say m and n where m > n, the connection consists of bits 0 through n-1 of the first register/bus being connected to bits 0 through n-1 of the second register/bus. The rest of the m-n connections are connected to logical low (0).

4.1. Registers

IC - a 10-bit used as the instruction counter for the user's program.
IX - a 10-bit register used as the index register by user programs for array-type addressing.
SP - a 10-bit register used as a stack pointer for call/return instructions as well as for arbitrary push/pop operations.
X - an 18-bit register used as a scratch register in various micro-instructions, and is invisible to the assembly -language program.
ACC - an 18-bit register which serves as an "accumulator" in the user's program.
MAR - a 10-bit register used primarily to store the address of where main memory is going to be written into or read from. It may also be used as a scratch register by the micro-program.
MBR - an 18-bit used to store the data involved in all memory read/write operations.
OC - a 6-bit register used to store the op-code of the currently executed macro-instruction.
II - a 2-bit register used to store the indexing/indirection flags of the currently executing assembly instruction.

4.2. Buses

Data bus - an 18-bit bi-directional bus that is connected to the various registers and to the adder output lines. Most movement of data between registers takes place via the data bus.
Address bus - a 10-bit bi-directional bus that is connected to the MAR register and to the adder output bus. This bus is used to supply the MAR register with the address of memory locations in read/write operations.
Left adder bus - an 18-bit bus that connects the various registers and several constant registers with the left input to the adder module.
Right adder bus - an 18-bit bus that connects the various registers and several constant registers with the right input to the adder module.

4.3. Gates

There are 40 distinct gates, each, when open, initiates a micro-operation. Any number of gates may be open at the same time, but some combinations of gates are mutually exclusive (ex: left-shift and right-shift). The hardware diagram in Appendix II specifies which gates open what hardware connections.

4.4. Memory

The main memory consists of 1024 words of 18 bits each. The memory locations have addresses in the range 0 through 1023, inclusive. Each word has its bits numbered 0 through 17, inclusive, where bit 0 is considered to be the least significant when numeric values are represented.

4.5. Inverters

There is a single logical inverter between each of the adder left and right buses, and the adder. These may invert none, one, or both arguments to the adder, depending on whether neither, one, or both are enabled.

4.6. Adder

There is an 18-bit adder whose inputs are the outputs of the inverters. At each clock cycle, the adder (which consists of solid-state combinatorial logic) sums its inputs and outputs the answer to its output.

4.7. Shifter

There is a single bi-directional shift register between the adder output and the data and address buses. It may shift the adder output by one bit to either left or right, depending on whether it is enabled.

4.8. Zero-detect logic

After each addition operation of the adder, the zero-detect logic resets or presets a bit that can be later tested for branching purposes. The zero-detect logic is set to '1' if the last addition resulted in a zero answer, and to '0' if the last addition resulted in a non-zero answer.

4.9. The Control Subsystem

The micro-programmed control subsystem of the machine is implemented by a control store micro-memory, a CSAR (control store address register) and CSBR (control store data register) registers, and hard-wired micro control logic. This entire subsystem is invisible to the assembly-language user.

4.9.1. The Micro-memory

The micro-memory consists of 512 words of storage, each of which contains 41 bits. The micro-memory words are numbered 0 through 511, inclusive, while the bits in each micro word are numbered 0 through 40, inclusive.

4.9.2. Micro-registers

CSAR - this is a 9-bit register that is used to address the micro-memory. It is similar in function to the MAR register for the main memory. CSAR is an acronym for "Control Store Address Register".

CSBR - this is a 41-bit register that contains the current micro instruction being executed. This register is directly in control of the hard-wired control logic and supervises the opening and closing of control gates (i.e., the generation of control signals) by virtue of the values contained in its bits. CSBR is an acronym for "Control Store Buffer Register".

4.9.3. Control Logic

The control logic for the micro programmed control subsystem is hard-wired (in this simulation it is written in C). It supervises the loading of instructions from the micro-memory, incrementing the CSAR register, and generating the control signals from the value of the CSBR and the clock pulses.

4.9.4. Start Toggle

The start toggle is a single bit register that allows the system to commence execution (when high) or causes the entire operation of the system to be suspended (when low). This is used to halt execution of the simulation, so that the user may inspect the contents of various registers/buses.

4.9.5. Clock

The operation of the control subsystem is governed by a three-phase clock. The phases of the clock are numbered P0, P1, and P2. The set of 40 system gates is partitioned into 3 distinct non-empty disjoint subsets, each of which contains gates that can be open ONLY during a unique clock phase. These sets are:

phase 1 gates = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 37, 38 }
phase 1 gates = { 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 39, 40 }
phase 3 gates = { 34, 35, 36 }

This partition exists in order to eliminate certain nasty ambiguities that arise when several inputs are allowed to to enter into the same register simultaneously, thereby rendering its contents undefined.

4.9.6. Micro-Instruction Format

Each micro instruction has one of the two formats specified in Appendix III. In the first format, the only operations that can occur are gates being opened according to which of bits 1 through 40 of the instruction are set high, during the appropriate clock phases. In the second format, a certain bit (specified by bits 10 through 14) of a register (specified by bits 1 through 9) is examined and compared with bit 15 of that instruction. If the comparison was successful (i.e., they were equal), then micro-control is transferred to the micro-location specified by bits 16 through 25 of the instruction. Both the "bit num" and the "address" fields are encoded in binary; the rest of the fields are linearly encoded, and only one of the bits of all of these fields must be set high (the rest being set low) in order for the instruction to logically make sense.

Having the system be micro-programmed

makes it very powerful with respect to non-micro-programmed systems. This is because new user-level assembly instruction sets can be easily implemented, and only by changing the micro-program, not having to touch the hardware at all. In fact, users may write their own micro-programs, thereby taking advantage of higher machine efficiency to suit their particular applications.

5. The Assembly Language

This section describes the assembler for the machine. The purpose of the assembler is to "compile" the user's assembly language into machine language and to place the resulting object code into the main memory so that it may be later executed. The reason for having an assembler, is to make the task of programming less tedious for the user; otherwise, the user would have had to program directly in the hardware's binary machine language.

A reasonable instruction set has already been written (and is the default instruction set) in order to accommodate users who do not wish to go through the tedium of writing their own micro-programs. Sensible mnemonics were also assigned to the various operations. It should be noted that the assembler is written in a general manner. The opcode mnemonics are read from an external file, and thus subject to modification by the user. The rest of the functions of the assembler remain unchanged from language to language. In fact, the only difference between two assembly languages here is between their two respective opcode mnemonic sets. Appendix IV gives the the mnemonics and their respective opcodes for the default assembly instruction set.

5.1. Stack

As can be easily seen, this language has a built-in stack facility for calling functions and for pushing values onto a stack. This makes the language posses substantial versatility. In the micro-program, the stack is rooted at the top of memory (location 1023) and grows toward smaller memory locations.

5.2. Instruction Format

The instruction format for this language calls for each instruction to be one word in length, in the format specified in Appendix V.

5.3. Assembly Syntax

This assembler recognizes an assembly language that is in a standard format, where each line of code is composed of one to three fields: label, opcode, and address. The address field may be immediately preceded by the character '*' which signified indirection, and succeeded by the two characters '()' which signify indexing. In addition to the default opcodes described earlier, there are three additional pseudo-opcode: the '<u>equ</u>' opcode, which is used to associate a label with a number/address, the '<u>con</u>' pseudo-operator, which is used to store data/constants into memory locations during the assembly process, and the '<u>org</u>' pseudo-operator, used to assemble code into several separate memory regions. A sample assembly program is given in Appendix I.

6. The Microcode Interpreter

This section describes the microcode interpreter. The function of the microcode interpreter is to convert the microcode from the symbolic form it is written in, to the form that can be placed into the micro-memory. Alternatively, the microcode would have been coded in binary by the user, which makes for a very tedious and error-prone task.

6.1. Microcode Syntax

The micro-program in symbolic form is composed of as many occurrences of the following 40 strings as desired: alu-right=ic, alu-left=ic, alu-right=ix, alu-left=ix, alu-right=sp, alu-left=sp, alu-right=x, alu-left=x, alu-right=acc, alu-left=acc, alu-right=-1, alu-left=0, alu-right=0, alu-right=1, alu-right=sign, mar=mbr, oc=mbr, ii=mbr, alu-left=mbr, left-shift, right-shift, data-bus=alu-output, address-bus=alu-output, data-bus=mbr, sp=data-bus, x=data-bus, x=18, acc=data-bus, mar=ic, ic=data-bus, mar=address-bus, mbr=data-bus, ix=data-bus, mbr=mem(mar), mem(mar)=mbr, start=off, invert-left-alu, invert-right-alu, x=10, data-bus=mar.

Each set of micro-operations that are specified on ONE input line, will be executed during ONE clock cycle (but maybe in different clock phases). The character ';' is used as a separator and should follow each one of the strings. Labels may be used, and comments are placed between curly brackets. In addition to the micro operations specified above, two more micro-instructions may be specified: the 'if' and the 'goto'. The 'if' has the following syntax:

if(**reg**.**bit**)=**cmp** then goto **label**;

where 'reg' is one of the strings { ic, ix, sp, x, acc, mbr, mar, oc, ii, zero-detect }, 'bit' is a decimal number that represents the bit to be tested, 'cmp' is either 0 or 1 (the value to be tested against), and 'label' is a valid label in the micro-program to be branched to if the test is successful (i.e. reg(bit)=cmp). The 'goto' micro-instruction is much simpler:

goto **label**;

This micro-instruction unconditionally transfers micro-control to the micro-location specified by 'label'.

7. The User Interface

7.1. Screen format

The simulator updates the terminal display in a screen-oriented fashion. Direct cursor control is exercised through a library package which is intelligent enough to look up the terminal type in the appropriate UNIX system file. The most current values of the various registers and buses are displayed on the screen at all times, unless the user specified to the simulator to run in the 'quiet' mode. This display makes possible for the user to trace only the specific system components of his/her choice, while possibly ignoring the rest, with minimal cognitive overhead. While the system is running, the display appears as in Appendix VI.

7.2. The Interaction With the User

All commands are one letter long, which in all cases is the first letter of the word describing the command. A short menu is present at the bottom of the display at all times, summarizing the commands. A help facility makes it possible to review the functions of the commands at any given time. Some commands generate a sub-menu, which contains subcommands appropriate for the original command only.

The various commands that are available at the top-level are: Pause - pauses between clock cycles (or phases), and wait for a new command, Continue - negates the last pause command, Stop - halts the machine, and creates a final memory dump, Quiet - does all things silently without updating the display, Trace - negates the 'quiet' command, Redraw - clears the screen and redraw the display, Values - allows the user to change the contents of registers and buses, Microcode - lists the interpreted microcode, Object - lists the object code of the assembled program, Examine - lists the contents of the entire main memory, Help - print this summary.

7.3. Error Handling

The microcode interpreter, as well as the assembler, may produce various diagnostic messages during normal operation. This usually occurs when the user fails to comply with the syntax rules built into the simulator. All such error messages are meant to be self-explanatory. The line number on which the error occurred is included in the error-message, when appropriate. When the microcode contains errors, assembly will not be attempted. When the source program contains errors, execution will not be attempted.

8. The Implementation

The hard-wired part of the control subsystem is written directly in the C language (after all, the simulation has to *end* somewhere). Execution of the microcode is done here and here only. Execution of the microcode commences at micro location 0 and proceeds logically unless "goto" instructions alter the logic flow. The Microcode is assumed to have been assembled and placed into the micro-memory. Execution of the microcode halts only after the microcode instruction 'start=off' has been executed. Each microcode instruction is fetched from the micro-memory, placed into the CSBR register, and combined with the clock pulses to generate control signals that will open various system gates.

As the microcode executes, it will fetch and interpret individual assembly/machine instructions from the user's program in main memory. Appropriate gates will open and close, and the desired effect will be achieved by having the corresponding micro operations take place. The types and effects of the various micro operations are described in earlier sections. To obtain the annotated C-sources constituting the simulator, please contact the author: Gabriel Robins, P.O. Box 8369, Van Nuys, California, 91409-8369, U.S.A.

9. Summary

I have developed an interactive tool for the simulation of a classical Von Neumann computer architecture. The simulation takes place at the register, bus, and gate level, and features a friendly user interface, an assembler, a microcode interpreter, and a terminal-independent full-screen display facility.

There exists a distinct lack of software tools to aid the teaching of computer science at the undergraduate level. I believe that my interactive simulator prototype, or other similar tools, will prove to be useful educational tools for the introduction of novices to the fundamentals of computer organization. Indeed, the construction of such a simulator will in itself constitute a good term project for an upper division hardware course.

10. Bibliography

Tanenbaum, S., Structured Computer Organization, Englewood Cliffs, New Jersey, Prentice Hall, 1976.

11. Appendix I: Usage Examples

11.1. Sample Micro-program

This is part of the default microcode for the simulated machine:

```
{ initialize the instruction counter and stack pointer to 0 }
alu-left=0 ; alu-right=0 ; data-bus=alu-output ; ic=data-bus; sp=data-bus;
        { fetch a macro-instruction from the main memory }
fetch:  mar=ic; mbr=mem(mar);
        { transfer the opcode and the indexing and indirection flags and
          increment the instruction counter }
oc=mbr; ii=mbr; mar=mbr; alu-left=ic; alu-right=1; data-bus=alu-output;  $
        ic=data-bus;
{ the following section is a giant 'switch' construct, that decodes the 64
possible opcodes and branches to the appropriate place for the execution
of the corresponding machine instruction }
0-to-63: if bit(oc,5)=1 then goto 32-to-63;
0-to-31: if bit(oc,4)=1 then goto 16-to-31;
0-to-15: if bit(oc,3)=1 then goto 8-to-15;
0-to-7: if bit(oc,2)=1 then goto 4-to-7;
0-to-3: if bit(oc,1)=1 then goto 2-to-3;
0-to-1: if bit(oc,0)=1 then goto 1-to-1;
                { nop - no operation }
0-to-0: goto fetch;
{---------------------------------------------------------------------}
                     { add - add memory to register }
{---------------------------------------------------------------------}
{ see if this instruction requires indexing }
1-to-1: if bit(ii,0)=0 then goto 1-to-1-no-indexing;
        { preform the indexing }
        data-bus=mar; x=data-bus;
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-bus;
        { see if this instruction requires indirection }
1-to-1-no-indexing: if bit(ii,1)=0 then goto 1-to-1-no-indirection;
        { perform the indirection }
        mbr=mem(mar);
        mar=mbr;
        { fetch the data from memory }
1-to-1-no-indirection: mbr=mem(mar);
 alu-left=mbr; alu-right=acc; data-bus=alu-output; acc=data-bus;
         goto fetch;
2-to-3: if bit(oc,0)=1 then goto 3-to-3;
```

```
{----------------------------------------------------------------------}
                { sub - subtract memory from register }
{----------------------------------------------------------------------}
        { see if this instruction requires indexing }
2-to-2:  if bit(ii,0)=0 then goto 2-to-2-no-indexing;
        { preform the indexing }
        data-bus=mar; x=data-bus;
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-bus;
        { see if this instruction requires indirection }
2-to-2-no-indexing: if bit(ii,1)=0 then goto 2-to-2-no-indirection;
        { perform the indirection }
        mbr=mem(mar);
        mar=mbr;
        { fetch the data from memory }
2-to-2-no-indirection: mbr=mem(mar);
alu-left=mbr; alu-right=0; invert-left-alu; data-bus=alu-output; x=data-bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=x; alu-right=acc; data-bus=alu-output; acc=data-bus;
        goto fetch;
{ Most of the micro-program is omitted here for space considerations...}
{----------------------------------------------------------------------}
                { hlt - halt the machine }
{----------------------------------------------------------------------}
63-to-63: start=off;
        goto fetch;
end
```

11.2. Sample Assembly Program

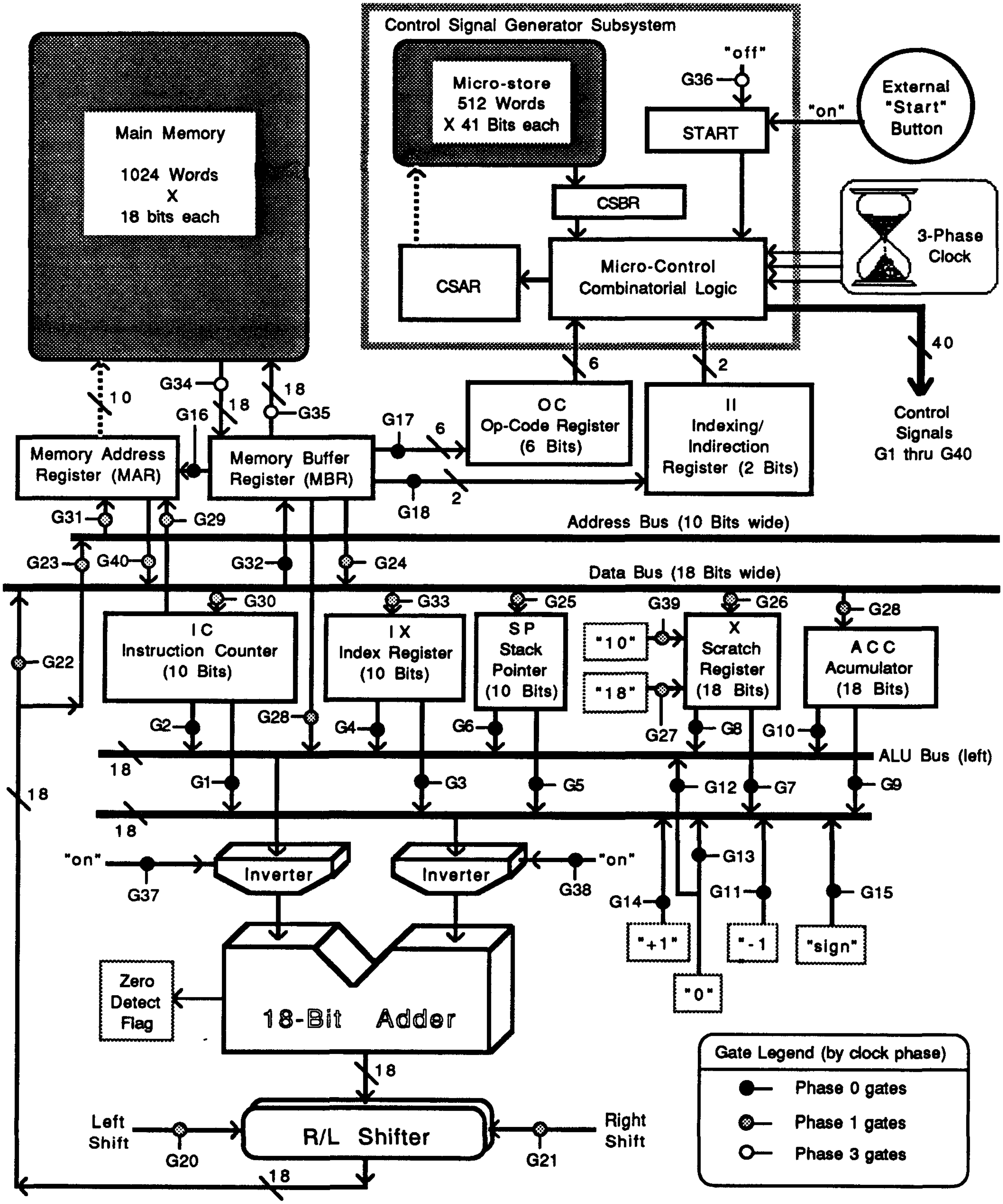
```
{ This program generates the first 25 Fibonacci numbers and places them in an array
in memory locations 50 thru 74 }

max         equ 25          { number of Fibonacci numbers we want }
array       equ 50          { array begins at 50 }
acc         equ 0           { defines the accumulator }
ix          equ 2           { defines the index register }
            call init       { initialize }
fibo        lda -2()        { get the Nth-2 Fibonacci number }
            add  -1()       { add to it the Nth-1 Fibonacci number }
            sta  0()        { store the result into the array }
            incr ix         { increment the index }
            ldai array      {}
            addai max       {}{ see if we have enough Fibonacci nums }
            subar ix        {}
            janz  fibo      { if not, go generate some more }
            hlt             { stop the machine }
            org 100         { place the routine starting at loc 100 }
init        ldai array      { initialize the array index }
            ldixr acc
            ldai 1
            sta  0()        { set the 1st Fibonacci number manually }
            incr ix
            sta  0()        { set the 2nd Fibonacci number manually }
            incr ix         { set the array pointer to the 3rd element }
            ret             { return to the caller }
            end             { end of assembly }
```

11.3. Main Memory Dump

Note the computed Fibonacci numbers beginning in memory location 50:

```
Location:    0 = 0000000000      Contents: 100000000001100100 = 131172
Location:    1 = 0000000001      Contents: 000011011111111110 = 14334
Location:    2 = 0000000010      Contents: 000001011111111111 = 6143
Location:    3 = 0000000011      Contents: 000100010000000000 = 17408
Location:    4 = 0000000100      Contents: 000101000000000010 = 20482
Location:    5 = 0000000101      Contents: 100101000000110010 = 151602
Location:    6 = 0000000110      Contents: 000111000000011001 = 28697
Location:    7 = 0000000111      Contents: 001110000000000010 = 57346
Location:    8 = 0000001000      Contents: 011101000000000001 = 118785
Location:    9 = 0000001001      Contents: 111111000000000000 = 258048
Location:   10 = 0000001010      Contents: 000000000000000000 = 0
   (intermediate locations have the same value)
Location:   49 = 0000110001      Contents: 000000000000000000 = 0
Location:   50 = 0000110010      Contents: 000000000000000001 = 1
Location:   51 = 0000110011      Contents: 000000000000000001 = 1
Location:   52 = 0000110100      Contents: 000000000000000010 = 2
Location:   53 = 0000110101      Contents: 000000000000000011 = 3
Location:   54 = 0000110110      Contents: 000000000000000101 = 5
Location:   55 = 0000110111      Contents: 000000000000001000 = 8
Location:   56 = 0000111000      Contents: 000000000000001101 = 13
Location:   57 = 0000111001      Contents: 000000000000010101 = 21
Location:   58 = 0000111010      Contents: 000000000000100010 = 34
Location:   59 = 0000111011      Contents: 000000000000110111 = 55
Location:   60 = 0000111100      Contents: 000000000001011001 = 89
Location:   61 = 0000111101      Contents: 000000000010010000 = 144
Location:   62 = 0000111110      Contents: 000000000011101001 = 233
Location:   63 = 0000111111      Contents: 000000000101111001 = 377
Location:   64 = 0001000000      Contents: 000000001001100010 = 610
Location:   65 = 0001000001      Contents: 000000001111011011 = 987
Location:   66 = 0001000010      Contents: 000000011000111101 = 1597
Location:   67 = 0001000011      Contents: 000000101000011000 = 2584
Location:   68 = 0001000100      Contents: 000001000001010101 = 4181
Location:   69 = 0001000101      Contents: 000001101001101101 = 6765
Location:   70 = 0001000110      Contents: 000000000000000000 = 0
   (intermediate locations have the same value)
Location:   99 = 0001100011      Contents: 000000000000000000 = 0
Location:  100 = 0001100100      Contents: 100101000000110010 = 151602
Location:  101 = 0001100101      Contents: 010010000000000000 = 73728
Location:  102 = 0001100110      Contents: 100101000000000001 = 151553
Location:  103 = 0001100111      Contents: 000100010000000000 = 17408
Location:  104 = 0001101000      Contents: 000101000000000010 = 20482
Location:  105 = 0001101001      Contents: 000100010000000000 = 17408
Location:  106 = 0001101010      Contents: 000101000000000010 = 20482
Location:  107 = 0001101011      Contents: 100001000000000000 = 135168
Location:  108 = 0001101100      Contents: 000000000000000000 = 0
   (intermediate locations have the same value)
Location: 1022 = 1111111110      Contents: 000000000000000000 = 0
Location: 1023 = 1111111111      Contents: 000000000000000001 = 1
```

12. Appendix II: The Hardware Diagram


Control Signal Generator Subsystem
Micro-store
512 Words
X 41 Bits each
"off"
G36
"on"
START
External
"Start"
Button
CSBR
CSAR
Micro-Control
Combinatorial Logic
3-Phase
Clock
40
Control
Signals
G1 thru G40
Main Memory
1024 Words
X
18 bits each
10
G34
18
18
G35
G16
Memory Address
Register (MAR)
Memory Buffer
Register (MBR)
G17
6
6
OC
Op-Code Register
(6 Bits)
2
II
Indexing/
Indirection
Register (2 Bits)
G18
2
G31
G29
Address Bus (10 Bits wide)
G23
G40
G32
G24
Data Bus (18 Bits wide)
G30
G33
G25
G39
G26
G28
G22
IC
Instruction Counter
(10 Bits)
IX
Index Register
(10 Bits)
SP
Stack
Pointer
(10 Bits)
"10"
"18"
X
Scratch
Register
(18 Bits)
ACC
Acumulator
(18 Bits)
G2
G28
G4
G6
G27
G8
G10
ALU Bus (left)
18
18
G1
G3
G5
G12
G7
G9
18
"on"
G37
Inverter
Inverter
"on"
G38
G13
G14
G11
G15
"+1"
"-1"
"sign"
"0"
Zero
Detect
Flag
18-Bit Adder
18
Left
Shift
G20
R/L Shifter
Right
Shift
G21
18
Gate Legend (by clock phase)
Phase 0 gates
Phase 1 gates
Phase 3 gates

13. Appendix III: The Microcode Instruction Format

(I) The **GATE** micro-instruction:

1	gates to be opened during current clock cycle
0	1 2 3 ... ... 39 40

(II) The **TEST** micro-instruction:

0	IC	IX	SP	X	ACC	MBR	MAR	CC	II	bit num	cmp	address	zd	unused
0	1	2	3	4	5	6	7	8	9	10...14	15	16...25	26	27...40

14. Appendix IV: The Default Assembly Instruction Set

op num	mnemonic	binary code	effect of operation
0	nop	000000	no operation
1	add	000001	add memory to register acc
2	sub	000010	subtract memory from register acc
3	lda	000011	load memory into register acc
4	sta	000100	store register acc into memory
5	incr	000101	increment register
6	decr	000110	decrement register
7	addai	000111	add to register acc immediate
8	subai	001000	subtract from register acc immediate
9	addixi	001001	add to ix immediate
10	subixi	001010	subtract from ix immediate
11	addspi	001011	add to sp immediate
12	subspi	001100	subtract from sp immediate
13	addar	001101	add register to acc
14	subar	001110	subtract register from acc
15	addixr	001111	add register to ix
16	subixr	010000	subtract register from ix
17	ldar	010001	load acc with register
18	ldixr	010010	load ix with register
19	ldicr	010011	load ic with register
20	inva	010100	invert acc
21	invix	010101	invert ix
22	anda	010110	and acc with memory
23	ora	010111	or acc with memory
24	xora	011000	xor acc with memory
25	rsfta	011001	right shift acc
26	lsfta	011010	left shift acc
27	jmp	011011	jump
28	jaz	011100	jump if acc is zero
29	janz	011101	jump if acc is not zero
30	jixz	011110	jump if ix is zero
31	jixnz	011111	jump if ix is not zero
32	call	100000	call a subroutine
33	ret	100001	return to caller
34	pusha	100010	push register acc onto stack
35	popa	100011	pop acc from stack

36	zeroa	100100	zero out the acc
37	ldai	100101	load acc immediate
63	hlt	111111	halt the machine

15. Appendix V: The Assembly Instruction Format

operation code	indirection	indexing	address of operand
17 ... 12	11	10	9 0

Bit 11, when on, causes indirection to occur. Bit 10, when on, causes indexing to occur via the IX register. Indexing takes precedence over indirection.

16. Appendix VI: The Main Display

```
---------Computer-Simulation-by--Gabriel-Robins--version-3-of-7/26/88---------
|                                                                             |
|  ACC=000100010100101111=17711          DATA-BUS=000000000000000000=0        |
|                                                                             |
|  MBR=000100010000000000=17408       ADDRESS-BUS=0000000000=0                |
|                                                                             |
|  MAR=0000000000=0                  ALU-LEFT-BUS=000000000000000011=3        |
|                                                                             |
|   IC=0000000011=3                 ALU-RIGHT-BUS=000000000000000001=1        |
|                                                                             |
|  open gates: 2 14 16 17 18                                                  |
|                                                                             |
|  micro-ops: alu-left=ic; alu-right=1; mar=mbr; oc=mbr; ii=mbr;              |
|                                                                             |
|-----------------------------------------------| CLOCK-PHASE=0               |
| OC=000100=4       II=01=1      Micro Program  |                             |
|                                Control Logic  | START=off   pausing         |
| CSAR=0000000011=3                             |                             |
|                                               | SP=0000000000=0             |
| CSBR=1010000000000010111000100000001000000000 |                             |
|                                               | IX=0001000111=71            |
| X=000000001111111111=1023      type=GATE█     |                             |
|-----------------------------------------------|                             |
-Pause-Continue-Stop-Quiet-Trace-Redraw-Values-Microcode-Object-Examine-Help--
```

17. Table of Contents

An Interactive Gate-Level Simulator[1] of a Classical Von Neumann Architecture, as an Educational Aid for Introducing Novices to the Fundamentals of Computer Organization

Gabriel Robins

Computer Science Department
University of California, Los Angeles

"Begin at the beginning," said the King very gravely,"and go on till you come to the end; then stop."

1. Abstract

I have developed an interactive tool for the simulation of a classical Von Neumann computer architecture. The simulation takes place at the register, bus, and gate level. The system consists of 9 registers, 4 buses, 40 gates, an adder, a memory, a micro-programmed control subsystem, a 3-phase clock, a "scratch" register, logical inverters, a bi-directional shift register, several constant registers, and zero-detect logic. A friendly user interface has been constructed, featuring an assembler, a microcode interpreter, and a terminal-independent full-screen display facility. My simulator prototype could effectively be used as an educational tool for the introduction of novices to the fundamentals of computer organization. Alternatively, the construction of such a simulator may in itself constitute a good term project for an upper division hardware course.

Keywords: Computer organization, simulation, learning tools, computer hardware, educational aids, user training systems.

Alice thought to herself, "I don't see how he can ever finish if he doesn't begin."

2. Introduction

We have developed an interactive tool for the simulation of a classical von Neumann computer architecture. The simulation takes place at the register, bus, and gate level. The components of our system include 9 registers, 4 buses, 40 gates, 1 adder, a memory, a micro-programmed control subsystem, a 3-phase clock, an extra "scratch" register, logical inverters, a bi-directional shift register, several constant registers, and zero-detect logic. In addition, we have constructed a friendly user interface, featuring an assembler, a microcode interpreter, and a terminal-independent full-screen display facility.

---

[1] "But it isn't old!" Tweedledum cried, in a greater fury than ever. "It's new, I tell you-"

There exists a distinct lack of software tools to aid and enhance the teaching of computer science at the undergraduate level. We believe that our interactive simulator prototype constitutes an extremely useful educational tool for the introduction of novices to the fundamentals of computer organization. The architecture we consider is based on the one discussed in [Tanenbaum].

3. Overview

"The Question is," said Alice, "whether you can make words mean so many different things."

This simulator requires 3 specification: the micro-code, the assembly instruction set, and the user program. When the simulator starts running, it loads the micro-program into the micro-store; next, it reads and assembles the user program into machine language, according to the instruction set specified (or else the default assembly instruction set). The resulting machine program is loaded into the main memory of the simulator. The simulator then begins to execute the micro-program; the micro-program, in turn, fetches, decodes, and executes instructions of the machine-language program.

By programming the simulator in micro-code, the user may thus create new and novel "instruction sets" for the "machine." For example, suppose the user wanted to add an assembly instruction "sqrt" which takes the integer square-root of the ACC register and leaves the result in the ACC register. The user will then need to add a new opcode called "sqrt" (and a corresponding machine-instruction code) to the assembly instruction set of the machine (by updating that file), and next modify the micro-program to perform the square root operation on the ACC register whenever the new instruction is encountered.

4. The Hardware

"The time has come," the Walrus said, "to talk of many things."

The computer system we chose to simulate is a simplified von Neumann-type single-processor micro-program controlled machine. The schematic organization of this system is given in appendix II. A detailed description of the components and topology of the system follows. Unless otherwise specified, when two registers/buses with different numbers of bits are connected, say m and n where m > n, the connection consists of bits 0 through n-1 of the first register/bus being connected to bits 0 through n-1 of the second register/bus. The rest of the m-n connections are connected to logical low (0).

4.1. registers

"Oh!" said Alice. She was too much puzzled to make any other remark.

IC - a 10-bit register whose output is connected to both adder buses, as well as to the MAR register (gates 1, 2, 29, respectively) and whose input is connected to the data bus (gate 30). This register is used as the instruction counter for the user's program.

IX - a 10-bit register whose output is connected to both adder buses (gates 3, 4) and whose input is connected to data bus (gate 33). This register is used as the index register by user programs for array-type addressing.

SP - a 10-bit register whose output is connected to both adder buses (gates 5, 6) and whose input is connected to the data-bus (gate 25). This register is used as a stack pointer for call/return instructions as well as for arbitrary push/pop operations.

X - an 18-bit register whose output is connected to both adder buses (gates 7, 8) and whose input is connected to the data bus, as well as to two constant registers "10" and "18" (gates 26, 39, 27, respectively). This register is only used as a scratch register in various micro-instructions, and is invisible to the assembly -language program.

ACC - an 18-bit register whose output is connected to both adder buses (gates 9, 10) and whose input is connected to the data bus (gate 28). This register is used in all arithmetic and logical operations, and serves as an "accumulator" in the user's program.

MAR - a 10-bit register whose output is connected to the data bus and to the memory (gate 40), and whose input is connected to the IC register as well as to the address bus (gates 29, 31, respectively). This register is also known as the memory address register, and is used primarily to store the address of where main memory is going to be written into or read from. It may also be used as a scratch register by the micro-program.

MBR - an 18-bit register whose output is connected to the main memory, data-bus, and left adder bus (gates 35, 24, and 19,respectively), and whose input is connected to the memory and to the data bus (gates 34, and 32, respectively). This register is also known as the memory buffer register and is used to store the data involved in all memory read/write operations (i.e., it contains the data to be read/written from/into a memory location referenced by register MAR).

OC - a 6-bit register whose output is directly connected to the micro-programmed control subsystem, and whose input is connected to the MBR register (gate 17). This register is used to store the op-code of the currently executed macro-instruction.

II - a 2-bit register whose output is directly connected to the micro-programmed control subsystem, and whose input is connected to the MBR register's bits 10 through 11 (gate 18). This register is used to store the indexing/indirection flags of the currently executing assembly instruction.

4.2. Buses

"Would you tell me please," said Alice, "what that means?"

Data bus - an 18-bit bi-directional bus that is connected to the various registers and to the adder output lines. Most movement of data between registers takes place via the data bus.

Address bus - a 10-bit bi-directional bus that is connected to the MAR register and to the adder output bus. This bus is used to supply the MAR register with the address of memory locations in read/write operations.

Left adder bus - an 18-bit bus that connects the various registers and several constant registers with the left input to the adder module. This bus is used to supply the adder with its left argument.

Right adder bus - an 18-bit bus that connects the various registers and several constant registers with the right input to the adder module. This bus is used to supply the adder with its right argument.

4.3. Gates

"I don't understand you," said Alice. "Its dreadfully confusing!"

There are 40 distinct gates, each, when open, initiates a micro-operation. Any number of gates may be open at the same time, but some combinations of gates are mutually exclusive (ex: left-shift and right-shift). A summary of the various gates and the micro operations they initiate follows:

1. alu-right = ic
2. alu-left = ic
3. alu-right = ix
4. alu-left = ix
5. alu-right = sp
6. alu-left = sp
7. alu-right = x
8. alu-left = x
9. alu-right = acc
10. alu-left = acc
11. alu-right = -1
12. alu-left = 0
13. alu-right = 0
14. alu-right = 1
15. alu-right = sign
16. mar = mbr
17. oc = mbr
18. ii = mbr
19. alu-left = mbr
20. left-shift
21. right-shift
22. data-bus = alu-output
23. address-bus = alu-output
24. data-bus = mbr
25. sp = data-bus
26. x = data-bus

27. x = 18
28. acc = data-bus
29. mar = ic
30. ic = data-bus
31. mar = address-bus
32. mbr = data-bus
33. ix = data-bus
34. mbr = mem(mar)
35. mem(mar) = mbr
36. start = off
37. invert-left-alu
38. invert-right-alu
39. x = 10
40. data-bus = mar

4.4. Memory

"It's a poor sort of memory that only works backwards," the Queen remarked.

The main memory consists of 1024 words of 18 bits each. The memory locations have addresses in the range 0 through 1023, inclusive. Each word has its bits numbered 0 through 17, inclusive, where bit 0 is considered to be the least significant when numeric values are represented.

4.5. Inverters

There is a single logical inverter between each of the adder left and right buses, and the adder. These may invert none, one, or both arguments to the adder, depending on whether neither, one, or both are enabled.

4.6. Adder

"Can you do Addition?" the White Queen asked. "What's one and one and one and one and one and one and one and one and one and one?"
"I don't know," said Alice. "I lost count."
"She can't do Addition," the Red Queen interrupted.

There is an 18-bit adder whose inputs are the outputs of the inverters. At each clock cycle, the adder (which consists of solid-state combinatorial logic) sums its inputs and outputs the answer to its output.

4.7. Shifter

There is a single bi-directional shift register between the adder output and the data and address buses. It may shift the adder output by one bit to either left or right, depending on whether it is enabled.

4.8. Zero-detect logic

After each addition operation of the adder, the zero-detect logic resets or presets a bit that can be later tested for branching purposes. The zero-detect logic is set to '1' if the last addition resulted in a zero answer, and to '0' if the last addition resulted in a non-zero answer.

4.9. The Control Subsystem

Alice remained looking thoughtfully at the mushroom for a minute, trying to make out which were the two sides of it; and, as it were perfectly round, she found this a very difficult question.

The micro-programmed control subsystem of the machine is implemented by a control store micro-memory, a CSAR (control store address register) and CSBR (control store data register) registers, and hard-wired micro control logic. This entire subsystem is invisible to the assembly-language user.

4.9.1. The Micro-memory

"-But there's one great advantage in it, that one's memory works both ways."

The micro-memory consists of 512 words of storage, each of which contains 41 bits. The micro-memory words are numbered 0 through 511, inclusive, while the bits in each micro word are numbered 0 through 40, inclusive.

4.9.2. Micro-registers

"And now which is which?" she said to herself

CSAR - this is a 9-bit register that is used to address the micro-memory. It is similar in function to the MAR register for the main memory. CSAR is an acronym for "Control Store Address Register".

CSBR - this is a 41-bit register that contains the current micro instruction being executed. This register is directly in control of the hard-wired control logic and supervises the opening and closing of control gates (i.e., the generation of control signals) by virtue of the values contained in its bits. CSBR is an acronym for "Control Store Buffer Register".

4.9.3. Control Logic

The control logic for the micro programmed control subsystem is hard-wired (in this simulation it is written in C). It supervises the loading of instructions from the micro-memory, incrementing the CSAR register, and generating the control signals from the value of the CSBR and the clock pulses.

4.9.4. Start Toggle

"I know something interesting is sure to happen," she said to herself

The start toggle is a single bit register that allows the system to commence execution (when high) or causes the entire operation of the system to be suspended (when low). This is used to halt execution of the simulation, so that the user may inspect the contents of various registers/buses.

4.9.5. Clock

The operation of the control subsystem is governed by a three-phase clock. The phases of the clock are numbered P0, P1, and P2. The set of 40 system gates is partitioned into 3 distinct non-empty disjoint subsets, each of which contains gates that can be open ONLY during a unique clock phase. These sets are:

phase 1 gates = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 37, 38 }

phase 1 gates = { 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 39, 40 }

phase 3 gates = { 34, 35, 36 }

This partition exists in order to eliminate certain nasty ambiguities that arise when several inputs are allowed to to enter into the same register simultaneously, thereby rendering its contents undefined.

4.9.6. Micro-Instruction Format

"You'll get used to it in time," said the Caterpillar;

Each micro instruction has one of the following formats:

(I) The **GATE** micro-instruction:

1	gates to be opened during current clock cycle
0	1 2 3 ... ... 39 40

(II) The **TEST** micro-instruction:

0	IC	IX	SP	X	ACC	MBR	MAR	CC	II	bit num	cmp	address	zd	unused
0	1	2	3	4	5	6	7	8	9	10...14	15	16...25	26	27...40

In the first format, the only operations that can occur are gates being opened according to which of bits 1 through 40 of the instruction are set high, during the appropriate clock phases.

In the second format, a certain bit (specified by bits 10 through 14) of a register (specified by bits 1 through 9) is examined and compared with bit 15 of that instruction. If the comparison was successful (i.e., they were equal), then micro-control is transferred to the micro-location specified by bits 16 through 25 of the instruction. Both the "bit num" and the "address" fields are encoded in binary; the rest of the fields are linearly encoded, and only one of the bits of all of these fields must be set high (the rest being set low) in order for the instruction to logically make sense.

Having the system be micro-programmed makes it very powerful with respect to non-micro-programmed systems. This is because new user-level assembly instruction sets can be easily implemented, and only by changing the micro-program, not having to touch the hardware at all. In fact, users may write their own micro-programs, thereby taking advantage of higher machine efficiency to suit their particular applications.

5. The Assembly Language

The Red Queen shook her head. "you may call it 'nonsense' if you like," she said, "but I've heard nonsense, compared with which that would be as sensible as a dictionary!"

This section describes the assembler for the machine. The purpose of the assembler is to "compile" the user's assembly language into machine language and to place the resulting object code into the main memory so that it may be later executed. The reason for having an assembler, is to make the task of programming less tedious for the user; otherwise, the user would have had to program directly in the hardware's binary machine language.

A reasonable instruction set has already been written (and is the default instruction set) in order to accommodate users who do not wish to go through the tedium of writing their own micro-programs. Sensible mnemonics were also assigned to the various operations. It should be noted that the assembler is written in a general manner. The opcode mnemonics are read from an external file, and thus subject to modification by the user. The rest of the functions of the assembler remain unchanged from language to language. In fact, the only difference between two assembly languages here is between their two respective opcode mnemonic sets.

5.1. Mnemonics

"I can't believe that!" said Alice.
"Can't you?" the Queen said in a pitying tone. "Try again: draw a long breath, and shut your eyes."

Here we give the mnemonics and their respective opcodes for the default assembly instruction set.

op num	mnemonic	binary code	effect of operation
0	nop	000000	no operation
1	add	000001	add memory to register acc
2	sub	000010	subtract memory from register acc
3	lda	000011	load memory into register acc
4	sta	000100	store register acc into memory
5	incr	000101	increment register
6	decr	000110	decrement register
7	addai	000111	add to register acc immediate
8	subai	001000	subtract from register acc immediate
9	addixi	001001	add to ix immediate
10	subixi	001010	subtract from ix immediate
11	addspi	001011	add to sp immediate
12	subspi	001100	subtract from sp immediate
13	addar	001101	add register to acc
14	subar	001110	subtract register from acc
15	addixr	001111	add register to ix
16	subixr	010000	subtract register from ix
17	ldar	010001	load acc with register
18	ldixr	010010	load ix with register
19	ldicr	010011	load ic with register
20	inva	010100	invert acc
21	invix	010101	invert ix
22	anda	010110	and acc with memory
23	ora	010111	or acc with memory
24	xora	011000	xor acc with memory
25	rsfta	011001	right shift acc
26	lsfta	011010	left shift acc
27	jmp	011011	jump
28	jaz	011100	jump if acc is zero
29	janz	011101	jump if acc is not zero
30	jixz	011110	jump if ix is zero
31	jixnz	011111	jump if ix is not zero
32	call	100000	call a subroutine
33	ret	100001	return to caller
34	pusha	100010	push register acc onto stack
35	popa	100011	pop acc from stack
36	zeroa	100100	zero out the acc
37	ldai	100101	load acc immediate
63	hlt	111111	halt the machine

In addition, there are three pseudo-operators that have effect only during the assembly process. These are the 'con', 'equ', and the 'org' operators, and will be discussed later.

5.2. Stack

"Why?" said the Caterpillar. Here was another puzzling question;

As can be easily seen, this language has a built-in stack facility for calling functions and for pushing values onto a stack. This makes the language posses substantial versatility. In the micro-program, the stack is rooted at the top of memory (location 1023) and grows toward smaller memory locations.

5.3. Instruction Format

"Come, there's half my plan done now! How puzzling all these changes are!"

The instruction format for this language calls for each instruction to be one word in length, having the following format:

operation code	indirection	indexing	address of operand
17 ... 12	11	10	9 0

Bit 11, when on, causes indirection to occur. Bit 10, when on, causes indexing to occur via the IX register. Indexing takes precedence over indirection.

5.4. Syntax

"Curiouser and curiouser!" cried Alice

This assembler recognizes the assembly language that is described by the following rules:

- Each line of code is composed of one to three fields: label, opcode, and address.
- The label field contains a label that may be referenced to anywhere else in the program. This label must consist of any non-white-space (i.e., non-blank-looking-when-printed) characters and can be of any length.
- The opcode field must contain a string that corresponds to one of the legal opcodes.
- The address field must contain a string that consists of the characters '0' through '9', optionally preceded by the character '-'. The address field may be immediately preceded by the character '*' which signified indirection, and succeeded by the two characters '()' which signify indexing. The last two options are not mandatory but should not be separated with a blank from the rest of the address field when included. Instead of a number, the address field may be a label, conforming to the rules of label-forming. The

address will be filled with the value of the label before the end of assembly. Each label must be unique in the program when appearing on the left on an opcode.

- The label field may start on or before column 3. The opcode field should start following the label field and anywhere on or before column 15. The address field should start after the opcode field and on or before column 40. There should be at least one white-space between each pair of fields.

- In addition to the default opcodes described earlier, there are three more pseudo-opcodes. These are used during assembly only, and are not used at all during execution. The first one is the '<u>equ</u>' opcode. It associates the label on its line with the number/address appearing on the same line, for future references in the program. This is useful for defining "global" constants once at the beginning of the program, and referring to them by name all throughout the program. The second one is the '<u>con</u>' pseudo-operator. It places the value of the address of the same line into the memory location specified by the assembly memory counter. The address here may be a label. This operator can be used to store data/constants into memory locations during the assembly process. The third is the '<u>org</u>' pseudo-operator. It simply modifies the assembler memory counter to become the address field specified with this operator. This is used to assemble code into several separate memory regions.

- The last line of any program must contain only the string "end" in the normal opcode field of that line.

- Comments may be included between curly brackets (i.e., '{' and '}'). All comments will be ignored. Blank lines may be inserted anywhere in the source program. All such lines will also be ignored. Comments may span several lines.

- Immediate instructions have their data in the address field, so the data can only be ten bits long, not eighteen.

- Instructions that operate on registers (ex: 'addar') take as an operand (i.e., as the address field) a number between 0 and 3, inclusive, that represents the register to be operated on as follows:

  0 - ACC
  1 - SP
  2 - IX
  3 - IC

  It is usually convenient to define these registers with 'equ's at the beginning of the program, for future references. For example:

```
ix      equ     2
.
.
.
        addar   ix
.
.
```

The last statement adds to the accumulator the contents of IX.

- Character case (upper vs. lower) matters, so the label 'FOO' is distinct from 'foo', and both are distinct from 'Foo'.

6. The Microcode Interpreter

It sounded an excellent plan, no doubt, and very neatly and simply arranged: the only difficulty was, that she had not the smallest idea how to set about it.

This section describes the microcode interpreter. The function of the microcode interpreter is to convert the microcode from the symbolic form it is written in, to the form that can be placed into the micro-memory. Alternatively, the microcode would have been coded in binary by the user, which makes for a very tedious and error-prone task.

6.1. Syntax

"I didn't say there was nothing better," the King replied. "I said there was nothing like it."

The micro-program in symbolic form should comply with the following rules:

- Each line shall be composed of as many occurrences of the following 40 strings as desired:

1. alu-right=ic
2. alu-left=ic
3. alu-right=ix
4. alu-left=ix
5. alu-right=sp
6. alu-left=sp
7. alu-right=x
8. alu-left=x
9. alu-right=acc
10 alu-left=acc
11. alu-right=-1
12. alu-left=0
13. alu-right=0
14. alu-right=1
15. alu-right=sign
16. mar=mbr
17. oc=mbr
18. ii=mbr
19. alu-left=mbr
20. left-shift
21. right-shift
22. data-bus=alu-output
23. address-bus=alu-output
24. data-bus=mbr
25. sp=data-bus

26. x=data-bus
27. x=18
28. acc=data-bus
29. mar=ic
30. ic=data-bus
31. mar=address-bus
32. mbr=data-bus
33. ix=data-bus
34. mbr=mem(mar)
35. mem(mar)=mbr
36. start=off
37. invert-left-alu
38. invert-right-alu
39. x=10
40. data-bus=mar

Each set of micro-operations that are specified on ONE input line, will be executed during ONE clock cycle (but maybe in different clock phases).

- The character ';' is used as a separator and should follow each one of the strings.
- Any line may be labeled by placing a label string at its beginning and by separating the label from the first micro-operation by the character ':'. Each label so used must be unique in the micro-program.
- Comments may be inserted between curly brackets. All comments will be ignored by the interpreter. Blank line will also be ignored and therefore can be inserted anywhere. Comments may span several lines.
- The last line of the micro program must contain the string "end", and nothing else.
- Case matters, so the label 'FOO' is distinct from 'foo', and both are distinct from 'Foo'.
- Long lines may be continued by placing the character '$' at the end of the line to be continued, and this may be done to continue a single micro instruction on as many lines as desired.
- In addition to the micro operations specified above, two more micro-instructions may be specified: the '<u>if</u>' and the '<u>goto</u>'. The 'if' has the following syntax:

if(**<u>reg</u>.<u>bit</u>**)=**<u>cmp</u>** then goto **<u>label</u>**;

where '<u>reg</u>' is one of the strings { ic, ix, sp, x, acc, mbr, mar, oc, ii, zero-detect }, '<u>bit</u>' is a decimal number that represents the bit to be tested, '<u>cmp</u>' is either 0 or 1 (the value to be tested against), and '<u>label</u>' is a valid label in the micro-program to be branched to if the test is successful (i.e. reg(bit)=cmp). The 'goto' micro-instruction is much simpler:

goto **<u>label</u>**;

This micro-instruction unconditionally transfers micro-control to the micro-location specified by '<u>label</u>'. If the label has any non-numeric character in it, it is considered to be a relative target address, to be determined when the same label is found at the beginning of another micro instruction. If the label consists entirely of numeric characters, it is taken to be an absolute address, specified in decimal. For example, '123-foo' is a label while '123' is a absolute micro-memory address. In the former case, the interpreter will search for a '123-foo' label on the other microcode source lines, and, if none are found, an 'undefined micro-label' will be generated. The label in the 'if' and 'goto' statements may optionally be replaced by an absolute address: this address will be taken literally to be the branching address, but in almost all cases, a label is *much* preferable to an absolute branching address, and so the provision for this capability here is only token.

- White-spaces may be inserted anywhere in the micro-program, as all white-spaces are completely ignored.

Any failures to comply with the above rules will cause syntax errors to be generated during the microcode-interpretation phase of this program. All error messages are explicit and are meant to be self-explanatory. After the microcode has been interpreted, it will be placed into the micro-memory, beginning at micro-memory location 0.

7. The User Interface

"Thank you very much," she whispered in reply, "but I can do quite well without."

7.1. Screen format

The simulator updates the terminal display in a screen-oriented fashion. Direct cursor control is exercised through a library package which is intelligent enough to look up the terminal type in the appropriate UNIX system file. The most current values of the various registers and buses are displayed on the screen at all times, unless the user specified to the simulator to run in the 'quiet' mode. This display makes possible for the user to trace only the specific system components of his/her choice, while possibly ignoring the rest, with minimal cognitive overhead. In addition, this method of display is generally considered particularly aesthetically pleasing (it is analogous to using a window-editor, whereas the alternative is a line-editor).

While the system is running, the display appears as follows:

```
---------Computer-Simulation-by--Gabriel-Robins--version-3-of-7/26/88---------
|                                                                             |
|  ACC=000100010100101111=17711        DATA-BUS=000000000000000000=0          |
|                                                                             |
|  MBR=000100010000000000=17408     ADDRESS-BUS=0000000000=0                  |
|                                                                             |
|  MAR=0000000000=0                ALU-LEFT-BUS=000000000000000011=3          |
|                                                                             |
|   IC=0000000011=3               ALU-RIGHT-BUS=000000000000000001=1          |
|                                                                             |
|  open gates: 2 14 16 17 18                                                  |
|                                                                             |
|  micro-ops: alu-left=ic; alu-right=1; mar=mbr; oc=mbr; ii=mbr;              |
|                                                                             |
|------------------------------------------------|  CLOCK-PHASE=0             |
| OC=000100=4      II=01=1      Micro Program    |                            |
|                               Control Logic    |  START=off   pausing       |
| CSAR=0000000011=3                              |                            |
|                                                |  SP=0000000000=0           |
| CSBR=101000000000001011100010000000100000000000|                            |
|                                                |  IX=0001000111=71          |
| X=000000001111111111=1023    type=GATE█        |                            |
|------------------------------------------------|                            |
-Pause-Continue-Stop-Quiet-Trace-Redraw-Values-Microcode-Object-Examine-Help--
```

7.2. The Interaction With the User

> *But Humpty Dumpty only shut his eyes, and said "Wait till you've tried."*

All commands are one letter long, which in all cases is the first letter of the word describing the command. A short menu is present at the bottom of the display at all times, summarizing the commands. A help facility makes it possible to review the functions of the commands at any given time. Some commands generate a sub-menu, which contains subcommands appropriate for the original command only. A major feature of the user interface is that pressing 'return' is never necessary, and most of the time even quite useless. Instead, the simulator 'senses' when a key was pressed, and accordingly takes the appropriate action. When no keys are pressed, it will continue merrily about its simulation, not bothering to prompt the user. Some commands, however, directly cause the simulator to pause and read keyboard input. This scheme tends to minimize both the number of key strokes and the elapsed time involved when controlling the behavior of the simulator.

7.3. The Commands

> *"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean -- neither more nor less."*

The various commands that are available at the top-level are:

- Pause - pause between clock cycles, and wait for a command. The pause command is two-level: a second pause will cause the machine to pause between clock phases. Subsequent pause commands will not have any effect.

- Continue - negate the last pause command.
- Stop - halt the machine, and create a final memory dump.
- Quiet - do all things silently, (do not update the display). This is useful for going quickly through thousands of simulation steps, without having to watch the changes in the system on the screen (which tends to greatly slow down the simulation).
- Trace - negate the 'quiet' command. All state changes of the simulator will be reflected on the screen.
- Redraw - clear the screen and redraw the display. This is useful when some renegade process writes bogus text to your terminal and 'messes up' the display.
- Values - allows the user to change the contents of registers and buses through a window-editor. This is very useful for experimenting with the simulator, and playing various 'what-if' scenarios.
- Microcode - list the interpreted microcode.
- Object - list the object code of the assembled program.
- Examine - list the contents of the entire main memory.
- Help - print this summary.

All commands that list things, do so by piping the output through a familiar system filter, for the convenience of the user. All standard system subcommands that can be used with this filter, can also be used when doing a listing. The filter used here is the UNIX utility more(1).

7.4. Error Handling

"It is wrong from beginning to end," said the Caterpillar, decidedly;

The microcode interpreter, as well as the assembler, may produce various diagnostic messages during normal operation. This usually occurs when the user fails to comply with the syntax rules built into the simulator. All such error messages are meant to be self-explanatory. The line number on which the error occurred is included in the error-message, when appropriate. When the microcode contains errors, assembly will not be attempted. When the source program contains errors, execution will not be attempted.

7.5. Special Files

"Ah, what is it now?" the Unicorn cried eagerly. "You'll never guess! I couldn't."

Several file names have special meaning to the simulator:

- "object.dump" - this file will contain the dump of the assembled user program, immediately after the assembly. It is not updated, so it will not be current when running self-modifying programs.

- "microcode.dump" - this file will contain the dump of the microcode. Since the microcode can not modify itself, this file is always current. For large micro-programs, however, the usefulness of the binary microcode dump is questionable.

- "final.memory" - this file will contain the final dump of the memory, after the entire system halted. It is meant to be used for debugging as well as for output presentation purposes (i.e., since there are no output devices associated with the simulator, programs can "print" to the memory, which can be examined by the user).

- "mnemonics" - this is the default file assumed by the simulator to initially contain the mnemonics (to the assembler) that define the assembly language being emulated in the microcode. This default may be overridden by specifying an appropriate argument when invoking the simulator. The format of the mnemonics file consists of two string per line, each enclosed in double quotes. what is before, after, or between the two strings is ignored. The first string is the opcode mnemonic, whose length is arbitrary, while the second string is the corresponding bit configuration, which should consist of the characters '0' and '1' only. The length of the second string must be 6 characters (a condition inherent in the topology of the system being simulated). The very last line of the file should contain the string "end" and nothing else. Here is an example of a valid "mnemonics" file:

  ```
  "nop", "000000" { this is the nop operation}
  {foo} "addr", { addition } "000111" { guess what this is }
  end
  ```

  This file defines an assembly language with two opcodes: 'nop' and 'addr'. Notice that column alignment and comment insertion are completely arbitrary.

- "microcode" - this is the default file assumed by the simulator to initially contain the microcode. This default may be overridden by specifying an appropriate argument when invoking the simulator.

- "program" - this is the default file assumed by the simulator to initially contain the source program. This default may be overridden by specifying an appropriate argument when invoking the simulator.

- "microload.log" - this file will be created only when errors were encountered during the microcode interpretation, and in this case it will contain the summary of the errors generated by the microcode interpreter.

- "assembly.log" - This file will be created only when errors were encountered during the assembly phase, and is this case it will contain the summary of the assembly errors generated by the assembler.

In addition the the above files, the simulator will leave behind files which contain the binary image of the corresponding microcode file. These files will have a '.o' extension attached to the end of their name. For example, if you invoked the simulator with the microcode file 'blah', a new file will be created under the name 'blah.o'. Any subsequent invocations of the simulator with the file 'blah' as the microcode file, will cause the microcode to be actually loaded from 'blah.o' instead, unless, of course, 'blah' has been modified since the creation of 'blah.o'. The underlying reasoning behind this scheme is concern with efficiency.

8. Invoking the Simulator

"Please then," said Alice, "how am I to get in?"

Invoking the simulator involves simply typing the name of the simulator to the UNIX operating system, followed by up to three positional arguments, all of which are optional. When omitted, each one of the arguments defaults to a value specified in a previous section. The arguments are:

1. The source program file (to be read by the assembler)
2. The microcode file (to be read by the microcode interpreter)
3. The mnemonics file (to be read by the assembler)

To override only one of the default values, specify null arguments for the rest of the previous arguments.

Examples follow:

emula - calls the simulator with no arguments (so the defaults will prevail).

emula mysource - calls the simulator with overriding the first argument only, accepting the defaults for arguments 2 and 3.

emula "" micro.prog - calls the emulator while overriding only the second argument.

emula "" "" mne - overrides argument 3, leaving the first two to default.

emula a b c - overriding all 3 arguments.

This scheme permits testing various programs with various microcode sets, without wasting time on file copying and manipulation. Note that the simulator "knows" what it is called. That is, the simulator will not run, unless it is invoked using the name "emula".

9. The Implementation

"If there is no meaning in it," said the King, "that saves a world of trouble, you know, as we needn't try to find any."

The hard-wired part of the control subsystem is written directly in the C language (after all, the simulation has to *end* somewhere). Execution of the microcode is done here and here only. Execution of the microcode commences at micro location 0 and proceeds logically unless "goto" instructions alter the logic flow. The Microcode is assumed to have been assembled and placed into the micro-memory. Execution of the microcode halts only after the microcode instruction 'start=off' has been executed. Each microcode instruction is fetched from the micro-memory, placed into the CSBR register, and combined with the clock pulses to generate control signals that will open various system gates.

As the microcode executes, it will fetch and interpret individual assembly/machine instructions from the user's program in main memory. Appropriate gates will open and close, and the desired effect will be achieved by having the corresponding micro operations take place. The types and effects of the various micro operations are described in earlier sections. The annotated C-code constituting the simulator is listed in appendix I.

10. Summary

"Pray don't trouble yourself to say it any longer than that," said Alice.

I have developed an interactive tool for the simulation of a classical Von Neumann computer architecture. The simulation takes place at the register, bus, and gate level, and features a friendly user interface, an assembler, a microcode interpreter, and a terminal-independent full-screen display facility.

There exists a distinct lack of software tools to aid the teaching of computer science at the undergraduate level. I believe that my interactive simulator prototype, or other similar tools, will prove to be useful educational tools for the introduction of novices to the fundamentals of computer organization. Indeed, the construction of such a simulator will in itself constitute a good term project for an upper division hardware course.

"Tut, tut, child" said the Duchess, "Everything's got a moral, if only you can find it."

11. Acknowledgements

Alice said afterwards that she had never seen such a fuss made about anything in all her life.

I originally conceived this project while attending a CS151B course, taught at UCLA by Dr. N. A. Alexandridis during the Winter of 1983. The simulation system that I developed was used in subsequent quarters to introduce computer science undergraduates to the fundamentals of computer hardware and organization; recently I decided that this system could still be very useful in computer science instruction to undergraduates; I therefore extracted this system from my tape archives and polished it. The various quotations sprinkled through this paper were taken from the classic works of [Carroll].

12. Bibliography

After a pause, Alice began. "Well! they were BOTH very unpleasant characters-"

Carroll, L., The Annotated Alice: Alice's Adventures in Wonderland & Through the Looking Glass, (with an introduction and notes by Martin Gardner), Signet Press, New York, 1960.

Tanenbaum, S., Structured Computer Organization, Englewood Cliffs, New Jersey, Prentice Hall, 1976.

13. Appendix I: The Annotated Source Code

"I'm afraid I can't put it more clearly," Alice replied very politely, "for I can't understand it myself, to begin with;"

13.1. Global Definitions Code

Alice sighed and gave it up. "Its exactly like a riddle with no answer!" she thought.

The following section defines all the constants used bu the program. The names are descriptive, and are meant to be self-explanatory.

```
#include    <stdio.h>
#include    <sgtty.h>

#define     DEBUG                           0
#define     BOOLEAN                         char
#define     STRING                          char
#define     MEMORY_LENGTH                   1024
#define     WORD_LENGTH                     18
#define     OPCODE_LENGTH                   6
#define     ADDRESS_LENGTH                  10
#define     TRUE                            1
#define     FALSE                           0
#define     NEWLINE                         '\n'
#define     END_OF_STRING                   '\0'
#define     ASSEMBLER_LABEL_COLUMN          3
#define     ASSEMBLER_OPCODE_COLUMN         15
#define     ASSEMBLER_ADDRESS_COLUMN        40
#define     ASSM_MAX_NUM_OF_LABELS          1000
#define     MAX_LENGTH_OF_LABEL_FIELD       9
#define     MAX_LENGTH_OF_OPCODE_FIELD      7
#define     MAX_LENGTH_OF_ADDRESS_FIELD     9
#define     BEGINNING_ASSEMBLY_ADDRESS      0
#define     SUCCESSFUL                      TRUE
#define     HIGH                            TRUE
#define     LOW                             FALSE
#define     OPCODE_BIT_POSITION             0
#define     INDIRECTION_BIT_POSITION        6
#define     INDEXING_BIT_POSITION           7
#define     ADDRESS_BIT_POSITION            8
#define     POINTER                         int
#define     HIGH_BIT                        "1"
#define     LOW_BIT                         "0"
#define     MEMORY_HIGH                     '1'
```

```
#define    MEMORY_LOW                       '0'
#define    PHASES_PER_CLOCK_CYCLE           3
#define    PHASES_PER_MICRO_CYCLE           3
#define    LENGTH                           int
#define    MICRO_OP_CODE_BIT_POSITION       0
#define    REGISTER                         int
#define    GATE                             TRUE
#define    TEST                             FALSE
#define    NUMBER_OF_GATES                  40
#define    LENGTH_OF_MICRO_INSTRUCTIONS     NUMBER_OF_GATES+1
#define    BUS                              int
#define    LATCH                            int
#define    FLAG                             BOOLEAN
#define    OPEN                             HIGH
#define    CLOSED                           LOW
#define    SIGN_WORD                        0400000
#define    WORD                             int
#define    INDEXING_BIT                     02000
#define    INDIRECTION_BIT                  04000
#define    NUMBER_OF_REGISTERS              10
#define    MICRO_MEMORY_LENGTH              512
#define    MAX_NUM_OF_MICRO_LABELS          MICRO_MEMORY_LENGTH
#define    WINDOW                           int
#define    ten_bits                         01777
#define    eighteen_bits                    0777777
#define    two_bits                         03
#define    six_bits                         077
#define    DOUBLE_QUOTE                     '"'
#define    NO_CLEAR                         12345
#define    OBJECT_DUMP_FILE                 "object.dump"
#define    FINAL_MEMORY_DUMP_FILE           "final.memory"
#define    MICROCODE_DUMP_FILE              "microcode.dump"
#define    ASSEMBLER_MNEMONICS_FILE         "mnemonics"
#define    ASSEMBLER_ERROR_LOG              "assembly.log"
#define    LOADER_ERROR_LOG                 "microload.log"
#define    DEFAULT_MICROCODE_FILE           "microcode"
#define    DEFAULT_PROGRAM_FILE             "program"
```

13.2. The Main Program Code

"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to," said the Cat.

"I don't care much where," said Alice.

"Then it does not matter which way you go," said the Cat.

This is the main body of the program. When invoking the program with no arguments, the microcode is read from the default file 'microcode' in the current directory, and the source assembly code is read from the default file 'program' in the current directory. If one argument is specified, it is taken to be the assembly source code file. If two arguments are specified, the second is taken to be the microcode file. If more than two arguments are given, the rest are ignored.

Three major events take place during the execution of this program: (I) The microcode file is read, the microcode is interpreted, and the resulting micro-instructions are placed into the micro memory. (II) The assembly source code file is read, the assembly program is interpreted, and the resulting object code is placed into the main memory. (III) Execution of the microcode commences. If errors were detected at any stage, all subsequent stages will not be attempted. Error messages are very explicit, and are meant to be self-explanatory.

Once execution begins, the display will be updated with the current values of the various registers and buses. Several commands may be issued from the keyboard at any point during the execution. These commands all consist of one letter (no carriage return is necessary) and a summary of those is printed at the bottom of the display at all times. In addition, one of these commands is a help command, that elaborates on the functions of the other commands. The program is meant to be used very interactively, although if no commands are issued, execution will proceed and reach its logical conclusion.

```
#include "defs.h"
#include <signal.h>

goodbye(action)
int action;
{
if(action==NO_CLEAR)
   system(" reset ; csh -f -c \"tset >& /dev/null\" ");
else
   system(" clear ; reset ; csh -f -c \"tset >& /dev/null\" ; clear ;
clear");
exit(0);
}

main(argc,argv)
int argc;          /*  the argument count */
char *argv[];      /*  the argument values */
{
```

The following declaration defines the mnemonics used for the various micro operations by the micro-program interpreter.

```
static STRING *micro_ops[] = {
   /* 1 */ "alu-right=ic",
   /* 2 */ "alu-left=ic",
   /* 3 */ "alu-right=ix",
   /* 4 */ "alu-left=ix",
   /* 5 */ "alu-right=sp",
   /* 6 */ "alu-left=sp",
   /* 7 */ "alu-right=x",
   /* 8 */ "alu-left=x",
   /* 9 */ "alu-right=acc",
   /*10 */ "alu-left=acc",
   /*11 */ "alu-right=-1",
   /*12 */ "alu-left=0",
   /*13 */ "alu-right=0",
   /*14 */ "alu-right=1",
   /*15 */ "alu-right=sign",
   /*16 */ "mar=mbr",
   /*17 */ "oc=mbr",
   /*18 */ "ii=mbr",
   /*19 */ "alu-left=mbr",
   /*20 */ "left-shift",
   /*21 */ "right-shift",
   /*22 */ "data-bus=alu-output",
   /*23 */ "address-bus=alu-output",
   /*24 */ "data-bus=mbr",
   /*25 */ "sp=data-bus",
   /*26 */ "x=data-bus",
   /*27 */ "x=18",
   /*28 */ "acc=data-bus",
   /*29 */ "mar=ic",
   /*30 */ "ic=data-bus",
   /*31 */ "mar=address-bus",
   /*32 */ "mbr=data-bus",
   /*33 */ "ix=data-bus",
   /*34 */ "mbr=mem(mar)",
   /*35 */ "mem(mar)=mbr",
   /*36 */ "start=off",
   /*37 */ "invert-left-alu",
   /*38 */ "invert-right-alu",
   /*39 */ "x=10",
   /*40 */ "data-bus=mar"
};
```

"Of course they answer to their names?" the Gnat remarked carelessly.
"I never knew them to do it."
"What's the use of their having names," the Gnat said, "if they won't answer to them?"
"No use to THEM," said Alice; but it's useful to the people that name them, I suppose. If not, why do things have names at all?"

The next declaration defines the space for the various operations that constitute the assembly language for this machine. These mnemonics are used by the assembler to decode the user's source program.

```
static STRING *opcodes[2*64];
```

The following declaration defines the main memory for the machine.

```
WORD memory[MEMORY_LENGTH];
```

The following declaration defines the micro memory of the micro-programmed control subsystem for the machine.

```
BOOLEAN micro_memory[MICRO_MEMORY_LENGTH][LENGTH_OF_MICRO_INSTRUCTIONS];

int i,tempo;
```

The next declaration defines the variables that will contain the names of the program file and of the microcode file.

```
    STRING program_file[80],microcode_file[80];
    int number_of_opcodes;

    if(strcmp((argv[0])+strlen(argv[0])-5,"emula")!=0)
         {
          printf("I am called 'emula', and will only answer to this
name.\n");
         exit(0);
         }

    signal(SIGINT,goodbye);
    signal(SIGQUIT,goodbye);

    system("reset ; csh -f -c \"tset >& /dev/null\" ; clear ");

    if(argc>2 && strcmp(argv[2],"")!=0)
            strcpy(microcode_file,argv[2]); /* grab the 2nd argument */
                                            /* use the default name */
    else strcpy(microcode_file,DEFAULT_MICROCODE_FILE);
    if(argc>1 && strcmp(argv[1],"")!=0 )
            strcpy(program_file,argv[1]);   /* grab the 1st argument */
                                            /* use the default name. */
    else strcpy(program_file,DEFAULT_PROGRAM_FILE);
```

The next statement will invoke the microcode interpreter. A success flag will be returned.

```
    tempo=load_micro_program_into_micro_memory(micro_memory,micro_ops,
                  microcode_file);
```

If the microcode interpretation was successful, proceed.

```
    if(tempo==SUCCESSFUL)

    {
       printf("Microcode load successful.\n");
       unlink(LOADER_ERROR_LOG);

       if(argc>3 && strcmp(argv[3],"")!=0)
```

```
            number_of_opcodes=grab_mnemonics(opcodes,argv[3]);
        else

    number_of_opcodes=grab_mnemonics(opcodes,ASSEMBLER_MNEMONICS_FILE);

            printf("Assembling user program from file '%s' ...
    \n",program_file);
```

The next statement will invoke the assembler.

```
        tempo = Assembler(memory,opcodes,number_of_opcodes,program_file);
```

If the assembly was successful, proceed.

```
        if(tempo == SUCCESSFUL)
        {
          printf("Assembly successful. \n");

          unlink(ASSEMBLER_ERROR_LOG);

          printf("Execution will commence at location 0.\n\n");
          printf("(press return to continue)");
          getc(stdin);
```

Start the execution of the microcode.

```
          Execute(micro_memory,memory,micro_ops);
```

Dump the contents of the main memory to a file for future reference.

```
          printf("Dumping the entire memory to file '%s'\n\n",
            FINAL_MEMORY_DUMP_FILE);
          dump_memory(memory,MEMORY_LENGTH,FINAL_MEMORY_DUMP_FILE);

        }
        else
          printf("Assembly unsuccessful, no execution attempted.\n");

      }
      else printf("Errors in microcode - cannot proceed.\n");

    unlink(OBJECT_DUMP_FILE);

    }
```

13.3. Assembler Code

"Why," said the Dodo, "the best way to explain it is to do it."

This section contains the assembler for the machine. The purpose of the assembler is to "compile" the user's assembly language into machine language and to place the resulting object module into the main memory so that it may be executed. The reason for having an assembler, is to make the task of program less tedious for the user.

```
#include "defs.h"
#include <ctype.h>

Assembler(memory,opcodes,number_of_opcodes,file)
WORD memory[];
STRING *opcodes[],file[];
int number_of_opcodes;
{
   BOOLEAN continue_assembling = TRUE;
   char buff[80];
   char label[MAX_LENGTH_OF_LABEL_FIELD+1];
   char opcode[MAX_LENGTH_OF_OPCODE_FIELD+1];
   char address[MAX_LENGTH_OF_ADDRESS_FIELD+1];
   POINTER      label_names[ASSM_MAX_NUM_OF_LABELS];
   int      label_targets[ASSM_MAX_NUM_OF_LABELS];
   int number_of_labels=0;
   int assembly_memory_counter=BEGINNING_ASSEMBLY_ADDRESS;
   STRING tmp[OPCODE_LENGTH+1];
   int i,j;
   BOOLEAN   success_flag = TRUE;
   int number_of_references=0;
   int reference_locations[ASSM_MAX_NUM_OF_LABELS*2];
   POINTER reference_labels[ASSM_MAX_NUM_OF_LABELS*2];
   BOOLEAN input_line_ok_flag;
   WORD current_word;
   FILE *fd,*assembler_err;
   extern int global_file_line_number;
   FLAG all_numeric;

   global_file_line_number=0;
   printf("Assembly begins:\n\n");
   fd=fopen(file,"r");
   assembler_err=fopen(ASSEMBLER_ERROR_LOG,"w");
```

Open the program source file.

```
   if(fd==NULL)
   {
      printf("The source file '%s' can not be found.",file);
      printf("  Check name and try again\n");
         fprintf(assembler_err,"The  source  file  '%s'  can  not  be
found.",file);
      fprintf(assembler_err,"  Check name and try again\n");
      return(FALSE);
```

```
}
for(i=0;i<MEMORY_LENGTH;i++)memory[i]=0;
while(continue_assembling)
{
   current_word=0;
   Getline(buff,fd);
```

Get the next source line.

```
   input_line_ok_flag=TRUE;
```

Parse the line.

```
   Parse(buff,label,opcode,address);
```

Watch out for 'end'.

```
   if(strcmp(opcode,"end")==0)
      continue_assembling=FALSE;
   else
```

Is this an 'equ' ?

```
      if(strcmp(opcode,"equ")==0)
      {
         label_names[number_of_labels]=
              strcpy(malloc(strlen(label)+1),label);
         label_targets[number_of_labels]=dec_string_to_num(address);
         number_of_labels++;
      }
      else
         if(strcmp(opcode,"con")==0)
         {
```

'con' was found.

```
            if(label[0]!=END_OF_STRING)
            {
               label_names[number_of_labels]=
                    strcpy(malloc(strlen(label)+1),label);

label_targets[number_of_labels]=assembly_memory_counter;
               number_of_labels++;
            }

memory[assembly_memory_counter]=dec_string_to_num(address);
            assembly_memory_counter++;
            if(assembly_memory_counter >= MEMORY_LENGTH)
            {
            printf("*** error: program exceeds memory bounds.\n");
            printf("Cannot continue with assembly: goodbye.\n");
            fprintf(assembler_err,
                  "*** error: program exceeds memory bounds.\n");
            fprintf(assembler_err,
                  "Cannot continue with assembly: goodbye.\n");
            goodbye(NO_CLEAR);
            }
```

```
            }
            else
            if(strcmp(opcode,"org")==0)
                  assembly_memory_counter=dec_string_to_num(address);
            else
            {

               i=0;
```

See which opcode it is.

```
                                    while(i<2*number_of_opcodes  &&
strcmp(opcodes[i],opcode)!=0)
                        i++;
               if(i==2*number_of_opcodes)
               {
                  printf(
                 "*** error: undefined operator mnemonic '%s' on line
%d.\n",
                        opcode,global_file_line_number);
                  fprintf(assembler_err,
                 "*** error: undefined operator mnemonic '%s' on line
%d.\n",
                        opcode,global_file_line_number);

                  success_flag=FALSE;
                  input_line_ok_flag=FALSE;
               }
               else
               {
                  strcpy(tmp,opcodes[i+1]);
```

Place the code into the current memory location.

```
                  current_word=current_word |
                        (binary_string_to_num(tmp) << 12);

               }
               if(label[0]!=END_OF_STRING && input_line_ok_flag)
               {
                  i=0;
```

Check for duplicate label.

```
                  while(strcmp(label_names[i],label)!=0
                        && i<= number_of_labels)i++;
                  if(strcmp(label_names[i],label)!=0)
                  {
```

Save the label away for future resolution.

```
                     label_names[number_of_labels]=
                         strcpy(malloc(strlen(label)+1),label);

label_targets[number_of_labels]=assembly_memory_counter;
                     number_of_labels++;
                  }
```

```
    else
        {
        printf(
  "*** error: duplicate label '%s' on line %d.\n",
  label,global_file_line_number);
        fprintf(assembler_err,
  "*** error: duplicate label '%s' on line %d.\n",
  label,global_file_line_number);

        success_flag=FALSE;
        input_line_ok_flag=FALSE;
    }
}
if(address[0]=='*')
{
```

Indirection is done in this instruction.

```
    current_word=current_word | INDIRECTION_BIT;
    strcpy(address,address+1);
}
if  (address[strlen(address)-1]==')'
  && address[strlen(address)-2]=='(' )
{
```

Indexing is done in this instruction.

```
    current_word=current_word | INDEXING_BIT;
    address[strlen(address)-2]=END_OF_STRING;
}
```

Is the address numeric, or is it a label ?

```
            all_numeric=TRUE;
            for(i=0;i<strlen(address);i++)
                    all_numeric=all_numeric &&
                        (isdigit(address[i]) || address[i]=='-');
    if(input_line_ok_flag)
            if(all_numeric)
```

The address is numeric.

```
current_word=current_word | (dec_string_to_num(address)
          & ten_bits);

else

{
```

The address is a label.

```
    reference_locations[number_of_references]
          =assembly_memory_counter;
    reference_labels[number_of_references]=
        strcpy(malloc(strlen(address)+1),address);
    number_of_references++;
}
```

Place the assembled instruction into the main memory.

```
memory[assembly_memory_counter]=current_word;
```

Increment the assembler memory counter.

```
if(input_line_ok_flag )assembly_memory_counter++;
if(assembly_memory_counter >= MEMORY_LENGTH)
{
printf("*** error: program exceeds memory bounds.\n");
printf("Cannot continue with assembly: goodbye.\n");
fprintf(assembler_err,
        "*** error: program exceeds memory bounds.\n");
fprintf(assembler_err,
        "Cannot continue with assembly: goodbye.\n");
goodbye(NO_CLEAR);
}

        }
    }

    assembly_memory_counter--;
```

Alice sighed and gave it up. "Its exactly like a riddle with no answer!" she thought.

Resolve all the references to labels throughout the program.

```
for(i=0;i<number_of_references;i++)
{
   j=0;
   while(strcmp(reference_labels[i],label_names[j])!=0
        && j< number_of_labels)j++;
   if(j<number_of_labels)
      memory[reference_locations[i]]=memory[reference_locations[i]] |
               label_targets[j];
   else
   {
               printf("*** Error - undefined label '%s'
\n",reference_labels[i]);
      fprintf(assembler_err,
                        "*** Error - undefined label '%s'
\n",reference_labels[i]);
      success_flag=FALSE;
   }
}
```

Close the source file.

```
fclose(fd);
```

Close the assembler error log file.

```
fclose(assembler_err);
```

Take a core dump of the object module.

```
    memory_dump(memory,assembly_memory_counter);

    if(success_flag==FALSE)
          printf("The error messages were placed into the file '%s'.\n",
                ASSEMBLER_ERROR_LOG);
```

Return the success result of the assembly.

```
    return(success_flag);
}
```

13.4. Microcode Interpreter Code

"Well, I never heard it before," said the Mock Turtle; "but is sounds uncommon nonsense."

This section contains the code for the microcode interpreter. The function of the microcode interpreter is to convert the microcode from the symbolic form it is written in to the form that can be placed into the micro-memory.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <ctype.h>
#include "defs.h"

load_micro_program_into_micro_memory(micro_memory,micro_ops,file)
BOOLEAN micro_memory[][LENGTH_OF_MICRO_INSTRUCTIONS];
STRING *micro_ops[],file[];
{
   STRING tmp[300],tmp2[300];
   int i,j,m,n,micro_memory_counter=0;
   POINTER micro_label_names[MAX_NUM_OF_MICRO_LABELS];
   int micro_label_targets[MAX_NUM_OF_MICRO_LABELS];
   int micro_reference_locations[MAX_NUM_OF_MICRO_LABELS];
   POINTER micro_reference_labels[MAX_NUM_OF_MICRO_LABELS];
   int number_of_micro_references=0;
   int number_of_micro_labels=0;
   BOOLEAN success_flag=TRUE;
   STRING current_word[LENGTH_OF_MICRO_INSTRUCTIONS+1];
   FLAG continue_interpreting=TRUE,no_dup,all_numeric;
```

The names of the various registers that may be specified in the micro-'if' statement.

```
   static STRING *register_names[] = {
      "ic",
      "ix",
      "sp",
      "x",
      "acc",
      "mbr",
      "mar",
      "oc",
      "ii",
      "zero-detect"
   };

   FILE *micro_code,*fd,*load_err;
   extern int global_file_line_number;

   struct stat stbuf;
   struct stat stbuf2;
   STRING save_file[15];

strncpy(save_file,file,12);
```

```
strcat(save_file,".o");
stat(file,&stbuf2);


if( (stat(save_file,&stbuf) == 0) && (stbuf.st_mtime > stbuf2.st_mtime)
)
{

      printf("The microcode has not changed since the last
interpretation.\n");
   printf("Therefore, the old interpreted microcode will be used.\n");
      printf("Reading the interpreted microcode from file
'%s'.\n\n",save_file);
   fflush(stdout);

   fd=fopen(save_file,"r");

   for(i=0;i<MICRO_MEMORY_LENGTH;i++)
   {
      for(j=0;j<LENGTH_OF_MICRO_INSTRUCTIONS;j++)
         micro_memory[i][j]=getc(fd);
      getc(fd);
   }
   fclose(fd);
   success_flag=TRUE;
   return(success_flag);

}
else
{

   global_file_line_number=0;
   load_err=fopen(LOADER_ERROR_LOG,"w");

   printf("loading microcode from file '%s' ...",file);

   micro_code=fopen(file,"r");
   if(micro_code==NULL)
   {
      printf("The microcode file '%s' can not be found. ",file);
      printf(" Check name and try again.\n");
       fprintf(load_err,"The microcode file '%s' can not be found.
",file);
      fprintf(load_err," Check name and try again.\n");
      return(FALSE);
   }
```

Start interpreting the microcode.

```
   while(continue_interpreting)
   {
```

Get a microcode line.

```
      Getline(tmp,micro_code);
```

Print a dot to show a sign of life (as the micro-code interpretation may take a while).

```
        fprintf(stdout,".");
        fflush(stdout);
```

Initialize the current micro-word being assembled.

```
        for(i=0;i<=NUMBER_OF_GATES;i++)current_word[i]=MEMORY_LOW;
        current_word[NUMBER_OF_GATES+1]=END_OF_STRING;
        i=j=0;
```

Squeeze white-spaces out of the input-line.

```
        for(i=0;i<strlen(tmp);i++)
           if(white_space(tmp[i])==FALSE)tmp[j++]=tmp[i];
        tmp[j]=END_OF_STRING;
```

watch out for the 'end' of the micro-program.

```
        if( strcmp(tmp,"end")==0 )continue_interpreting=FALSE;
        else
        {

           i=0;
           while(tmp[i]!=':' && i<strlen(tmp))i++;
```

Does this line contain a label ?

```
           if(tmp[i]==':')
           {
```

Yes, it does.

```
           tmp[i]=END_OF_STRING;
           no_dup=TRUE;
           j=0;
```

Make sure it is not a duplicate label so far.

```
           while(j<number_of_micro_labels && no_dup)
                                          no_dup   =   no_dup   &&   (
  strcmp(tmp,micro_label_names[j++])!=0 );

           if(no_dup)
              {
```

It was unique, so save it away in the label table.

```
              micro_label_names[number_of_micro_labels]=
                 strcpy(malloc(strlen(tmp)+1),tmp);

  micro_label_targets[number_of_micro_labels]=micro_memory_counter;
              number_of_micro_labels++;
              strcpy(tmp,tmp+i+1);
              }
           else
              {
```

Otherwise, generate an error.

```
                    success_flag=FALSE;
                     printf(">>> error - duplicate micro-label '%s' on line
    %d.\n",
                            tmp,global_file_line_number);
                    fprintf(load_err,
                            ">>> error - duplicate micro-label '%s' on line
    %d.\n",
                            tmp,global_file_line_number);
                    strcpy(tmp,tmp+i+1);
                    }
                  }
```

Is this a goto statement ?

```
                  if(tmp[0]=='g' && tmp[1]=='o' && tmp[2]=='t')
                  {
```

/* Yes, it is. */

```
                  if(tmp[strlen(tmp)-1]==';')tmp[strlen(tmp)-1]=END_OF_STRING;
```

Is the address numeric (absolute), or is it a label?

```
                  all_numeric=TRUE;
                  for(i=4;i<strlen(tmp);i++)
                       all_numeric=all_numeric && isdigit(tmp[i]);
```

The address is numeric.

```
                  if(all_numeric)
                  {
                  num_to_binary_string(dec_string_to_num(tmp+4),10,tmp);
                  m=0;
                  for(n=16;n<=25;n++)current_word[n]=tmp[m++];
                  }
                  else
```

The address is a label.

```
                  {
                  j=number_of_micro_labels-1;
```

Was the target label already encountered ?

```
                  while(strcmp(tmp+4,micro_label_names[j]) != 0  && j>0 ) j--;
                  if(j>=0)
                    {
```

Yes, it was.

```
                    num_to_binary_string(micro_label_targets[j],10,tmp);
                    m=0;
                    for(n=16;n<=25;n++)current_word[n]=tmp[m++];
                    }
```

```
            else
              {
```

Otherwise, save the target as an unresolved reference.

```
              micro_reference_locations[number_of_micro_references]=
                  micro_memory_counter;
              micro_reference_labels[number_of_micro_references]=
                  strcpy(malloc(strlen(tmp)-3),tmp+4);
              number_of_micro_references++;
              }
              }
            }
            else
```

Is this an 'if' statement ?

```
              if(!(tmp[0]=='i' && tmp[1]=='f'))
              {
```

No, it is not.

```
                while(strcmp(tmp,"")!=0)
                {
                  i=0;
```

Make sure the line ends with ';'.

```
                  while(tmp[i]!=';' && i<strlen(tmp))i++;
                  if(i<strlen(tmp))tmp[i]=END_OF_STRING;
                  else
                  {
                  printf(
                      ">>> error:  missing ';' at end-of-line on line
%d.\n",
                     global_file_line_number);
                  fprintf(load_err,
                      ">>> error:  missing ';' at end-of-line on line
%d.\n",
                     global_file_line_number);
                  success_flag=FALSE;
                  }

                  j=0;
```

Determine which micro-operation is it.

```
                                    while(j<NUMBER_OF_GATES  &&
strcmp(tmp,micro_ops[j])!=0)
{
j++;
}
                  if(j==NUMBER_OF_GATES)
                  {
```

If not found, generate an error.

```
          printf(
    ">>> error: undefined micro operation '%s' on line %d.\n"
    ,tmp,global_file_line_number);
          fprintf(load_err,
    ">>> error: undefined micro operation '%s' on line %d.\n"
              ,tmp,global_file_line_number);
          success_flag=FALSE;
      }
      else
      {
          current_word[j+1]=MEMORY_HIGH;
      }
```

But she could not help thinking to herself "What dreadful nonsense we are talking!"

Get to the next micro-op.

```
      if(strcmp(tmp,"")!=0)strcpy(tmp,tmp+i+1);
  }
  current_word[0]=MEMORY_HIGH;
}
else
{
```

We have an 'if' statement.

```
  current_word[0]=MEMORY_LOW;
  strcpy(tmp,tmp+6);
  i=0;
```

Look for the ',' separator.

```
  while(i<strlen(tmp) && tmp[i]!=',')i++;
  if(i==strlen(tmp))
  {
```

No ',' separator.

```
          printf(
">>> error: 'if' statement syntax (no ',' found) on line %d.\n",
           global_file_line_number);
          fprintf(load_err,
">>> error: 'if' statement syntax (no ',' found) on line %d.\n",
           global_file_line_number);
          success_flag=FALSE;
          }
          else
          {
              tmp[j=i]=END_OF_STRING;
              i=0;
```

Determine which register is to be tested.

```
                    while(i<NUMBER_OF_REGISTERS &&
                        strcmp(tmp,register_names[i])!=0)i++;
                    if(i>=NUMBER_OF_REGISTERS)
                    {
                       printf(
                   ">>> error: undefined register mnemonic '%s' on line
%d.\n",
                 tmp,global_file_line_number);
                       fprintf(load_err,
                   ">>> error: undefined register mnemonic '%s' on line
%d.\n",
                 tmp,global_file_line_number);
                    success_flag=FALSE;
                    }
                    else
                    {
```

Set up the word to test the right register, or zero-detect.

```
                       if(i==9)current_word[26]=MEMORY_HIGH;
                       else current_word[i+1]=MEMORY_HIGH;
                    }
                    strcpy(tmp,tmp+j+1);
                    i=0;
```

Look for the ')' separator.

```
                    while(i<strlen(tmp) && tmp[i]!=')')i++;
                    if(i==strlen(tmp))
                    {
                       printf(
          ">>> error: 'if' statement syntax (no ')') found on line %d.\n",
          global_file_line_number);
                       fprintf(load_err,
          ">>> error: 'if' statement syntax (no ')') found on line %d.\n",
          global_file_line_number);
                       success_flag=FALSE;
                    }
                    else
                    {
                       tmp[i]=END_OF_STRING;

  num_to_binary_string(dec_string_to_num(tmp),5,tmp2);

  for(j=0;j<strlen(tmp2);j++)current_word[10+j]=tmp2[j];
                       current_word[15]=tmp[i+2];
```

Extract out the label.

```
                       strcpy(tmp,tmp+i+11);
                       j=0;
                       while(tmp[j]!=';' && j<strlen(tmp))j++;
                       if(j==strlen(tmp))
                       {
                       printf(
                          ">>> error: missing ';' at end of line %d.\n",
                          global_file_line_number);
                       fprintf(load_err,
```

```
                "\>>> error: missing ';' at end of line %d.\n",
                global_file_line_number);
             success_flag=FALSE;
             }
             else tmp[j]=END_OF_STRING;
             j=0;
```

Is the address numeric (absolute), or is it a label?

```
             all_numeric=TRUE;
             for(i=0;i<strlen(tmp);i++)
                all_numeric=all_numeric && isdigit(tmp[i]);
```

The address is numeric (absolute).

```
             if(all_numeric)
             {
  num_to_binary_string(dec_string_to_num(tmp),10,tmp);
             m=0;
             for(n=16;n<=25;n++)current_word[n]=tmp[m++];
             }
             else
```

The address is a label.

```
             {
```

Was the label encountered before ?

```
             while(strcmp(tmp,micro_label_names[j]) != 0
                    && j<number_of_micro_labels)j++;
             if(j<number_of_micro_labels)
                {
```

Yes, it was.

```
  num_to_binary_string(micro_label_targets[j],10,tmp);
                m=0;
                for(n=16;n<=25;n++)current_word[n]=tmp[m++];
                }
             else
                {
```

No, it was not, so save it away as an unresolved reference.

```
  micro_reference_locations[number_of_micro_references]=
                         micro_memory_counter;

  micro_reference_labels[number_of_micro_references]=
                         strcpy(malloc(strlen(tmp)+1),tmp);
                number_of_micro_references++;
                }
          }
          }
       }
    }
```

Place the assembled word into the micro-memory.

```
              for(j=0;j<=LENGTH_OF_MICRO_INSTRUCTIONS;j++)
                 if(current_word[j]=='1')
                    micro_memory[micro_memory_counter][j]=MEMORY_HIGH;
                 else micro_memory[micro_memory_counter][j]=MEMORY_LOW;

              micro_memory_counter++;
              if(micro_memory_counter >= MICRO_MEMORY_LENGTH)
              {
                 printf(">>> error: micro-program exceeds micro-memory
bounds.\n");
              printf("Cannot continue with microcode loading: goodbye.\n");
              fprintf(load_err,
                      ">>> error: micro-program exceeds micro-memory
bounds.\n");
              fprintf(load_err,
                  "Cannot continue with microcode loading: goodbye.\n");
              goodbye();
              }
           }
       }
```

Close the microcode file.

```
       fclose(micro_code);
```

Resolve all the unresolved label references. */

```
       for(i=0;i<number_of_micro_references;i++)
       {
```

Print a dot to show a sign of life (as the micro-code interpretation may take a while).

```
          fprintf(stdout,".");
          fflush(stdout);

          j=0;
          while(strcmp(micro_reference_labels[i],micro_label_names[j]) != 0
              && j<number_of_micro_labels)j++;
          if(j<number_of_micro_labels)
          {
             num_to_binary_string(micro_label_targets[j],10,tmp);
             m=0;
             j=micro_reference_locations[i];
             for(n=16;n<=25;n++)micro_memory[j][n]=tmp[m++];

          }
          else
          {
```

Print error messages for labels which were not found to exist.

```
             printf(">>> error: undefined micro-label '%s'. \n",
                    micro_reference_labels[i]);
             fprintf(load_err,">>> error: undefined micro-label '%s'. \n",
```

```
                        micro_reference_labels[i]);
                  success_flag=FALSE;
               }
         }
```

"It's too ridiculous!" cried Alice, losing all her patience this time.

Dump the micro-memory onto a disk-file, for future reference.

```
      if(success_flag == TRUE)
         {
         save_interpreted_microcode(micro_memory,MICRO_MEMORY_LENGTH,file);
         micro_memory_dump(micro_memory,micro_memory_counter);
         }
```

Print interpretation diagnostics.

```
      printf("\n");
      printf("There are %d microcode instructions (maximum capacity is
   %d).\n",
            micro_memory_counter,MICRO_MEMORY_LENGTH);
   }

      fclose(load_err);

      if(success_flag==FALSE)
            printf("The error messages were placed into the file '%s'.\n",
                  LOADER_ERROR_LOG);

      return(success_flag);

   }


   micro_memory_dump(micro_memory,prog_length)
   BOOLEAN micro_memory[][LENGTH_OF_MICRO_INSTRUCTIONS];
   LENGTH prog_length;
   {
      int i,j;
      FILE *fd;
      FLAG same;
      BOOLEAN old_line[LENGTH_OF_MICRO_INSTRUCTIONS];


      printf("dumping-microcode-to-disk");

      fd=fopen(MICROCODE_DUMP_FILE,"w");
      fprintf(fd,"\n-----------------micro-memory-dump-----------------
   \n");
      for(i=0;i<prog_length;i++)
      {
         if(i%10==0)
         {
```

Print a dot to show a sign of life.

```
        fprintf(stdout,".");
        fflush(stdout);

        }

        for(j=0;j<1;j++)
        if(micro_memory[i][j]=='1')fprintf(fd,"1"); else fprintf(fd,"0");

        fprintf(fd," ");

        for(j=1;j<11;j++)
        if(micro_memory[i][j]=='1')fprintf(fd,"1"); else fprintf(fd,"0");

        fprintf(fd," ");

        for(j=11;j<21;j++)
        if(micro_memory[i][j]=='1')fprintf(fd,"1"); else fprintf(fd,"0");

        fprintf(fd," ");

        for(j=21;j<31;j++)
        if(micro_memory[i][j]=='1')fprintf(fd,"1"); else fprintf(fd,"0");

        fprintf(fd," ");

        for(j=31;j<41;j++)
        if(micro_memory[i][j]=='1')fprintf(fd,"1"); else fprintf(fd,"0");

        fprintf(fd,"  %d\n",i);

   }
   fprintf(fd,"---------------------------------------------------
\n\n");
   fclose(fd);
}


save_interpreted_microcode(micro_memory,length,microcode_file)
BOOLEAN micro_memory[][LENGTH_OF_MICRO_INSTRUCTIONS];
LENGTH length;
STRING microcode_file[];
{
   int i,j;
   FILE *fd;
   STRING save_file[15],b[20];

   strncpy(save_file,microcode_file,12);
   strcat(save_file,".o");

   printf("\n saving the microcode in file '%s'  ",save_file);
   fflush(stdout);

   fd=fopen(save_file,"w");

   for(i=0;i<length;i++)
   {
      if(i%10==0)
```

```
        {
        fprintf(stdout,".");
        fflush(stdout);
        }
        for(j=0;j<LENGTH_OF_MICRO_INSTRUCTIONS;j++)
                if(micro_memory[i][j]=='1')fprintf(fd,"1");  else
fprintf(fd,"0");
        fprintf(fd,"\n");
    }
    fclose(fd);

}
```

13.5. Control Subsystem Code

"I should like to have it explained," said the Mock Turtle.

This section is in fact the hard-wired micro-programmed control subsystem. Execution of the microcode is done here and here only. Execution of the microcode commences at micro location 0 and proceeds logically until goto instructions alter the logic flow. The Microcode is assumed to have been assembled and placed into the micro memory. Execution of the microcode halts only after the microcode instruction 'start=off' has been executed.

As the microcode executes, individual instructions from the user's program will be fetched from the main memory and interpreted. Appropriate gates will open and close, and the desired effect will be achieved by having the corresponding micro operations take place. The types and effects of the various micro operations are described in other sections.

```
#include "defs.h"

Execute(micro_memory,memory,micro_ops)
BOOLEAN micro_memory[][LENGTH_OF_MICRO_INSTRUCTIONS];
WORD memory[];
STRING *micro_ops[];
{
   WINDOW display,micro_display;
   int suspend=FALSE,show=TRUE;
   FLAG start=TRUE,ZERO_DETECT,go_to,dont_modify;
   REGISTER IC,IX,SP,X,ACC,MAR,MBR,OC,II,CSAR;
   STRING CSBR[LENGTH_OF_MICRO_INSTRUCTIONS+1];
   BOOLEAN gates[NUMBER_OF_GATES+1];
```

The partitioning of the gates into clock-phase-groups.

```
   static int phase_0_gates[] = {
      1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,37,38
   };
   static int phase_1_gates[] = {
      20,21,22,23,24,25,26,27,28,29,30,31,32,33,39,40
   };
   static int phase_2_gates[] = {
      34,35,36
   };

   BUS ADDER_RIGHT_BUS,ADDER_LEFT_BUS,ADDRESS_BUS,DATA_BUS;
   LATCH ADDER_OUTPUT_LATCH;

   FLAG zero_detect;
   int i,j,bitnum,test_result,CLOCK,foo;
   FILE *tty;

   STRING tmp[30],pp_if[30];
   FLAG DESCRIBE=TRUE,already_forked;
   int pause=0;
```

Initialize the window-world package.

```
initscr();
```

Define the display windows.

```
display = newwin(24,80,0,0);
micro_display = subwin(display,9,50,14,0);
```

Initialize the display.

```
redraw(display,micro_display);
```

Reset the various registers, gates, and buses.

```
CSAR=0;
ACC=MAR=MBR=OC=II=IC=IX=SP=X=0;
ADDER_RIGHT_BUS=ADDER_LEFT_BUS=ADDRESS_BUS=DATA_BUS=0;
ADDER_OUTPUT_LATCH=0;
start=TRUE;
MAR=BEGINNING_ASSEMBLY_ADDRESS;
for(i=1;i<=NUMBER_OF_GATES;i++)gates[i]=CLOSED;
```

Commence execution of the microcode.

```
while(start==TRUE)
{
```

Fetch a micro-instruction from the micro-memory.

```
    micro_fetch(micro_memory,CSAR,CSBR);

    CSAR++;
```

Is it a GATE or TEST instruction ?

```
    if(decode_micro_instruction(CSBR) == GATE)
```

It was a gate instruction, so start the clock ticking...

```
        for(CLOCK=0;CLOCK<PHASES_PER_CLOCK_CYCLE;CLOCK++)
        {
```

Adder operates during the clock cycle P1.

```
            if(CLOCK==1)
            {
                ADDER_OUTPUT_LATCH = (ADDER_LEFT_BUS + ADDER_RIGHT_BUS) &
                              eighteen_bits;
                if(ADDER_OUTPUT_LATCH < 0)ADDER_OUTPUT_LATCH=
                              ADDER_OUTPUT_LATCH & SIGN_WORD;
```

Reset/Preset the zero-detect logic.

```
                if(ADDER_OUTPUT_LATCH==0)ZERO_DETECT=TRUE;
                else ZERO_DETECT=FALSE;
```

```
}
```

Open the appropriate gates.

```
if(CLOCK==0)for(i=0;i<sizeof(phase_0_gates)/4;i++)
   if(CSBR[phase_0_gates[i]]==MEMORY_HIGH)
            gates[phase_0_gates[i]]=OPEN;

if(CLOCK==1)for(i=0;i<sizeof(phase_1_gates)/4;i++)
   if(CSBR[phase_1_gates[i]]==MEMORY_HIGH)
            gates[phase_1_gates[i]]=OPEN;

if(CLOCK==2)for(i=0;i<sizeof(phase_2_gates)/4;i++)
   if(CSBR[phase_2_gates[i]]==MEMORY_HIGH)
            gates[phase_2_gates[i]]=OPEN;
```

Do the appropriate micro-operations according to which gates are open.

```
if(CLOCK==0)
{
if(gates[1]==OPEN )ADDER_RIGHT_BUS=IC;
if(gates[2]==OPEN )ADDER_LEFT_BUS=IC;

if(gates[3]==OPEN )ADDER_RIGHT_BUS=IX;
if(gates[4]==OPEN )ADDER_LEFT_BUS=IX;

if(gates[5]==OPEN )ADDER_RIGHT_BUS=SP;
if(gates[6]==OPEN )ADDER_LEFT_BUS=SP;

if(gates[7]==OPEN )ADDER_RIGHT_BUS=X;
if(gates[8]==OPEN )ADDER_LEFT_BUS=X;

if(gates[9]==OPEN )ADDER_RIGHT_BUS=ACC;
if(gates[10]==OPEN )ADDER_LEFT_BUS=ACC;
if(gates[11]==OPEN )ADDER_RIGHT_BUS =
            (~0) & eighteen_bits;
if(gates[12]==OPEN )ADDER_LEFT_BUS = 0;
if(gates[13]==OPEN )ADDER_RIGHT_BUS = 0;
if(gates[14]==OPEN )ADDER_RIGHT_BUS = 1;
if(gates[15]==OPEN )ADDER_RIGHT_BUS = SIGN_WORD;
if(gates[16]==OPEN )MAR=MBR & ten_bits;
if(gates[17]==OPEN )OC=(MBR >> 12) & six_bits;
if(gates[18]==OPEN )II=(MBR >> 10) & two_bits;
if(gates[19]==OPEN )ADDER_LEFT_BUS=MBR;
if(gates[37]==OPEN )ADDER_LEFT_BUS=
            (~ADDER_LEFT_BUS) & eighteen_bits;
if(gates[38]==OPEN )ADDER_RIGHT_BUS=
            (~ADDER_RIGHT_BUS) & eighteen_bits;
}
else
if(CLOCK==1)
{
if(gates[20]==OPEN )ADDER_OUTPUT_LATCH=
            (ADDER_OUTPUT_LATCH << 1 ) & eighteen_bits;
if(gates[21]==OPEN )ADDER_OUTPUT_LATCH=
            (ADDER_OUTPUT_LATCH >> 1) & eighteen_bits;
if(gates[22]==OPEN )DATA_BUS=
            (ADDER_OUTPUT_LATCH) & eighteen_bits;
```

```
                if(gates[23]==OPEN )ADDRESS_BUS=
                          (ADDER_OUTPUT_LATCH) & ten_bits;
                if(gates[24]==OPEN )DATA_BUS=(MBR) & eighteen_bits;
                if(gates[40]==OPEN )DATA_BUS=(MAR)& ten_bits;
                if(gates[25]==OPEN )SP=(DATA_BUS) & ten_bits;
                if(gates[26]==OPEN )X=DATA_BUS;
                if(gates[27]==OPEN )X=18;
                if(gates[28]==OPEN )ACC=DATA_BUS;
                if(gates[29]==OPEN )MAR=IC;
                if(gates[30]==OPEN )IC=(DATA_BUS) & ten_bits;
                if(gates[31]==OPEN )MAR=ADDRESS_BUS;
                if(gates[32]==OPEN )MBR=DATA_BUS;
                if(gates[33]==OPEN )IX=(DATA_BUS) & eighteen_bits;
                if(gates[39]==OPEN )X=10;
                }
                else
                if(CLOCK==2)
                {
                if(gates[34]==OPEN )MBR=(memory[MAR]) & eighteen_bits;
                if(gates[35]==OPEN )memory[MAR]=MBR;
                if(gates[36]==OPEN )start=FALSE;
                }
```

Update the display if we are supposed to do so.

```
                if(show)
                  display_values(micro_memory,memory,CLOCK,CSAR,CSBR,IC,IX,

      SP,X,ACC,MAR,MBR,OC,II,DATA_BUS,ADDRESS_BUS,

      ADDER_RIGHT_BUS,ADDER_LEFT_BUS,ADDER_OUTPUT_LATCH,

      gates,!suspend,pause,GATE,micro_ops,0,display,
                                  micro_display);
```

If we are supposed to pause, time-out and accept a terminal input.

```
                if(pause==2)

      fork_out(&start,&pause,&show,micro_memory,memory,&CSAR,CSBR,

      &IC,&IX,&SP,&X,&ACC,&MAR,&MBR,&OC,&II,&DATA_BUS,
                                 &ADDRESS_BUS,&ADDER_RIGHT_BUS,&ADDER_LEFT_BUS,
                                 display,micro_display);
                for(i=1;i<=NUMBER_OF_GATES;i++)gates[i]=CLOSED;

            }
          else
          {
```

The current microcode command was a TEST command. The variable pp_if contains the print-form of the disassembeled micro instruction.

```
          if(show)
            {
            CLOCK=0;
            strcpy(pp_if,"if bit(");
            dont_modify=FALSE;
```

```
j=0;
go_to=FALSE;
for(i=10;i<=14;i++)tmp[j++]=CSBR[i];
tmp[j]=END_OF_STRING;
bitnum=binary_string_to_num(tmp);
```

See which register is to be tested.

```
if(CSBR[1]==MEMORY_HIGH)
{
   test_result=bit(IC,bitnum);
   strcat(pp_if,"ic,");
}
else
    if(CSBR[2]==MEMORY_HIGH)

{
   test_result=bit(IX,bitnum);
   strcat(pp_if,"ix,");
}
else
    if(CSBR[3]==MEMORY_HIGH)
{
   test_result=bit(SP,bitnum);
   strcat(pp_if,"sp,");
}
else
    if(CSBR[4]==MEMORY_HIGH)
{
   test_result=bit(X,bitnum);
   strcat(pp_if,"x,");
}
else
    if(CSBR[5]==MEMORY_HIGH)
{
   test_result=bit(ACC,bitnum);
   strcat(pp_if,"acc,");
}
else
    if(CSBR[6]==MEMORY_HIGH)
{
   test_result=bit(MBR,bitnum);
   strcat(pp_if,"mbr,");
}
else
    if(CSBR[7]==MEMORY_HIGH)
{
   test_result=bit(MAR,bitnum);
   strcat(pp_if,"mar,");
}
else
    if(CSBR[8]==MEMORY_HIGH)
{
   test_result=bit(OC,bitnum);
   strcat(pp_if,"oc,");
}
else
    if(CSBR[9]==MEMORY_HIGH)
```

```
        {
           test_result=bit(II,bitnum);
           strcat(pp_if,"ii,");
        }
        else
            if(CSBR[26]==MEMORY_HIGH)
        {
           test_result=bit(ZERO_DETECT,0);
           strcpy(pp_if,"if zero-detect then go to ");
           dont_modify=TRUE;
        }
        else
            {
```

This was an unconditional goto.

```
           go_to=TRUE;
           strcpy(pp_if,"go to ");
           dont_modify=TRUE;
        }
        if(dont_modify==FALSE)
        {
           strcat(pp_if,num_to_dec_string(bitnum,2,tmp));
           strcat(pp_if,")=");
           tmp[0]=CSBR[15];
           tmp[1]=END_OF_STRING;
           strcat(pp_if,tmp);
           strcat(pp_if," then go to ");
        }
        j=0;
        for(i=16;i<=25;i++)tmp[j++]=CSBR[i];
        tmp[j]=END_OF_STRING;
        foo=binary_string_to_num(tmp);
        strcat(pp_if,num_to_dec_string(foo,4,tmp));
        strcat(pp_if,";");
        }
     else
        {
        go_to=FALSE;
        j=0;
        for(i=10;i<=14;i++)tmp[j++]=CSBR[i];
        tmp[j]=END_OF_STRING;
        bitnum=binary_string_to_num(tmp);
```

See which register is to be tested.

```
        if(CSBR[1]==MEMORY_HIGH) test_result=bit(IC,bitnum);
        else
            if(CSBR[2]==MEMORY_HIGH) test_result=bit(IX,bitnum);
        else
            if(CSBR[3]==MEMORY_HIGH) test_result=bit(SP,bitnum);
        else
            if(CSBR[4]==MEMORY_HIGH) test_result=bit(X,bitnum);
        else
            if(CSBR[5]==MEMORY_HIGH) test_result=bit(ACC,bitnum);
        else
            if(CSBR[6]==MEMORY_HIGH) test_result=bit(MBR,bitnum);
        else
```

```
            if(CSBR[7]==MEMORY_HIGH) test_result=bit(MAR,bitnum);
        else
            if(CSBR[8]==MEMORY_HIGH) test_result=bit(OC,bitnum);
        else
            if(CSBR[9]==MEMORY_HIGH) test_result=bit(II,bitnum);
        else
            if(CSBR[26]==MEMORY_HIGH) test_result=bit(ZERO_DETECT,0);
        else go_to=TRUE;

        j=0;
        for(i=16;i<=25;i++)tmp[j++]=CSBR[i];
        tmp[j]=END_OF_STRING;
        foo=binary_string_to_num(tmp);
        }
```

Perform the actual test on the register.

```
        if( ((test_result == TRUE) && (CSBR[15] == MEMORY_HIGH)) ||
            ((test_result == FALSE) && (CSBR[15] == MEMORY_LOW)) ||
go_to )
            CSAR=foo;
```

"You don't know what you're talking about!" cried Humpty Dumpty.

Update the display if we are supposed to do so.

```
already_forked=FALSE;

for(CLOCK=0;CLOCK<PHASES_PER_CLOCK_CYCLE;CLOCK++)
{
        if(show)

                display_values(micro_memory,memory,CLOCK,CSAR,CSBR,IC,IX,

SP,X,ACC,MAR,MBR,OC,II,DATA_BUS,ADDRESS_BUS,

ADDER_RIGHT_BUS,ADDER_LEFT_BUS,ADDER_OUTPUT_LATCH,

gates,!suspend,pause,TEST,micro_ops,pp_if,display,
                                micro_display);
```

Pause and accept a terminal command if we are supposed to do so.

```
        if(pause==2)
                {

fork_out(&start,&pause,&show,micro_memory,memory,&CSAR,CSBR,

&IC,&IX,&SP,&X,&ACC,&MAR,&MBR,&OC,&II,&DATA_BUS,
                        &ADDRESS_BUS,&ADDER_RIGHT_BUS,&ADDER_LEFT_BUS,
                        display,micro_display);
                already_forked=TRUE;
```

```
                }
    }

    }
```

Pause and accept a terminal command if we are supposed to do so.

```
    if(!already_forked)
         if(pause==1)

    fork_out(&start,&pause,&show,micro_memory,memory,&CSAR,CSBR,

    &IC,&IX,&SP,&X,&ACC,&MAR,&MBR,&OC,&II,&DATA_BUS,
                        &ADDRESS_BUS,&ADDER_RIGHT_BUS,&ADDER_LEFT_BUS,
                        display,micro_display);
```

See if there are any input character in the buffer of stdin. This is the KEY for super-interactivity: the program will go on about its business UNTIL it noticed that the user touched ANY key (not necessarily the (return) key). It will then decide what to do based upon which character was entered.

```
         ioctl(0,FIONREAD,&suspend);

         if(suspend>0)

    fork_out(&start,&pause,&show,micro_memory,memory,&CSAR,CSBR,

    &IC,&IX,&SP,&X,&ACC,&MAR,&MBR,&OC,&II,&DATA_BUS,
                        &ADDRESS_BUS,&ADDER_RIGHT_BUS,&ADDER_LEFT_BUS,
                        display,micro_display);
```

Zero out the buses, as the are not latches and are not supposed to retain information over time.

```
         ADDER_RIGHT_BUS=ADDER_LEFT_BUS=ADDRESS_BUS=DATA_BUS=0;

      }
```

Update the display one last time.

```
              display_values(micro_memory,memory,CLOCK,CSAR,CSBR,IC,IX,

    SP,X,ACC,MAR,MBR,OC,II,DATA_BUS,ADDRESS_BUS,

    ADDER_RIGHT_BUS,ADDER_LEFT_BUS,ADDER_OUTPUT_LATCH,
                        gates,0,999,GATE,micro_ops,0,display,
                        micro_display);
```

Wait for one last command before finishing.

```
    fork_out(&start,&pause,&show,micro_memory,memory,&CSAR,CSBR,

    &IC,&IX,&SP,&X,&ACC,&MAR,&MBR,&OC,&II,&DATA_BUS,
                        &ADDRESS_BUS,&ADDER_RIGHT_BUS,&ADDER_LEFT_BUS,
                        display,micro_display);
```

Close up the window-world package.

```
    endwin();
```

Reset the terminal back into -raw and echo modes. Clear the display.

```
    system("reset ; csh -f -c \"tset >& /dev/null\" ; clear ");
}
```

13.6. Display Functions Code

"What is the use of repeating all that stuff," the Mock Turtle interrupted, "if you don't explain it as you go on? its by far the most confusing thing I ever heard!"

This section contains various functions that are associated mostly with updating and printing the display, as well as some functions that are used by the micro-programmed control subsystem.

```
#include "defs.h"
```

This function fetches a micro instruction from the micro-memory. The location from which the instruction is fetched is specified by CSAR, while the resulting instruction is placed into the CSBR buffer.

```
micro_fetch(micro_memory,CSAR,CSBR)
BOOLEAN micro_memory[][LENGTH_OF_MICRO_INSTRUCTIONS];
REGISTER CSAR;
STRING CSBR[];
{
   int i;

   for(i=0;i<LENGTH_OF_MICRO_INSTRUCTIONS;i++)
      CSBR[i]=micro_memory[CSAR][i];
   CSBR[i]=END_OF_STRING;

}
```

This function decodes the fetched micro instruction and returns its type. This is very simple to do, as there are only two types of instructions.

```
decode_micro_instruction(CSBR)
STRING CSBR[];
{

   if(CSBR[MICRO_OP_CODE_BIT_POSITION]==MEMORY_HIGH)return(GATE);
   else return(TEST);

}
```

This function prints the value of a register/bus onto the display in both binary and decimal form. A label for the field is also printed.

```
pretty_print(w,y,x,str,val,binlen,decp,simulate_sign)
WINDOW w;
int x,y,binlen;
STRING str[];
REGISTER val;
FLAG decp,simulate_sign;
{
   int i;
```

```
    STRING tmp[WORD_LENGTH+1];

    wmove(w,y,x);
    num_to_binary_string(val,binlen,tmp);
    if(simulate_sign)if(val < 0)tmp[0]=MEMORY_HIGH;

    wprintw(w,"%s=%s=%-8d",str,tmp,val);

}
```

This functions goes through the various different system components and prints their state (i.e., value). Printing is done by calling a proper print function with the value and label of the various fields.

```
display_values(micro_memory,memory,CLOCK,CSAR,CSBR,IC,IX,SP,X,ACC,MAR,MB
R,
OC,II,DATA_BUS,ADDRESS_BUS,ADDER_RIGHT_BUS,ADDER_LEFT_BUS,ADDER_OUTPUT_L
ATCH,
gates,start,pause,micro_type,micro_ops,mo,display,micro_display)

BOOLEAN micro_memory[][LENGTH_OF_MICRO_INSTRUCTIONS];
WORD memory[];
REGISTER IC,IX,SP,X,ACC,MAR,MBR,OC,II,CSAR;
STRING CSBR[LENGTH_OF_MICRO_INSTRUCTIONS+1],mo[];
BOOLEAN gates[NUMBER_OF_GATES];
BUS ADDER_RIGHT_BUS,ADDER_LEFT_BUS,ADDRESS_BUS,DATA_BUS;
LATCH ADDER_OUTPUT_LATCH;
int CLOCK,start,pause,micro_type;
STRING *micro_ops[];
WINDOW display,micro_display;
{
   int i;

   pretty_print(display,2,3,"ACC",ACC,18,TRUE,TRUE);
   pretty_print(display,18,53,"SP",SP,10,TRUE,FALSE);
   pretty_print(display,20,53,"IX",IX,10,TRUE,FALSE);
   pretty_print(display,8,4,"IC",IC,10,TRUE,FALSE);

   pretty_print(display,2,40,"DATA-BUS",DATA_BUS,18,TRUE,TRUE);
   pretty_print(display,4,37,"ADDRESS-BUS",ADDRESS_BUS,10,TRUE,TRUE);
   pretty_print(display,6,36,"ALU-LEFT-
BUS",ADDER_LEFT_BUS,18,TRUE,TRUE);
   pretty_print(display,8,35,"ALU-RIGHT-
BUS",ADDER_RIGHT_BUS,18,TRUE,TRUE);

   wmove(display,14,53);
   wprintw(display,"CLOCK-PHASE=%d",CLOCK);

   pretty_print(display,6,3,"MAR",MAR,10,TRUE,FALSE);
   pretty_print(display,4,3,"MBR",MBR,18,TRUE,TRUE);

   wmove(display,10,3);
   for(i=1;i<=70;i++)wprintw(display," ");
   wmove(display,10,3);
   wprintw(display,"open gates: ");
   for(i=1;i<=NUMBER_OF_GATES;i++)if(gates[i]==OPEN)wprintw(display,"%d
",i);
   wmove(display,12,3);
```

```
        for(i=1;i<=150;i++)wprintw(display," ");
        wmove(display,12,79);
        wprintw(display,"||");
        wmove(display,12,3);
        wprintw(display,"micro-ops: ");
        for(i=1;i<=NUMBER_OF_GATES;i++)
             if(gates[i]==OPEN)wprintw(display,"%s; ",micro_ops[i-1]);
        if(mo)wprintw(display,"%s",mo);

        wmove(display,16,53);
        wprintw(display,"START=");
        if(pause)wprintw(display,"off");
        else wprintw(display,"on ");
        if(pause==999)wprintw(display,"   system halted");
        else
           if(pause)wprintw(display,"   pausing");
        else      wprintw(display,"            ");


        pretty_print(micro_display,1,2,"OC",OC,6,TRUE,FALSE);
        pretty_print(micro_display,1,19,"II",II,2,TRUE,FALSE);
        pretty_print(micro_display,3,2,"CSAR",CSAR,10,TRUE,FALSE);
        pretty_print(micro_display,7,2,"X",X,18,TRUE,TRUE);

        wmove(micro_display,5,2);
        wprintw(micro_display,"CSBR=%s",CSBR);

        wmove(micro_display,7,33);
        wprintw(micro_display,"type=");
        if(micro_type==GATE)wprintw(micro_display,"GATE");
        else wprintw(micro_display,"TEST");

        wrefresh(display);
        wrefresh(micro_display);

        fflush(1);
    }
```

"Is that all?" Alice timidly asked.

The following function allows the user to interactively change the contents of a register/bus via a mini-window-editor that is implemented here. The usage of this mini-window-editor will become obvious once the user issues the 'Values' command during the execution phase of the program.

```
    change_reg(w,reg,y,x,len)
    WINDOW w;
    REGISTER *reg;
    int y,x,len;
    {
        int k,bit;
        STRING tmp[20];

        for(bit=len-1;bit>=0;bit--)
        {
           wmove(w,y,x+len-bit-1);
```

```
        wrefresh(w);
        k=getc(stdin);
        if(k=='1')
        {
           *reg = *reg | (01 << bit);
           wmove(w,y,x);
           wprintw(w,"%s=%-9d",num_to_binary_string(*reg,len,tmp),*reg);
           wrefresh(w);
        }
        if(k=='t')
        {
           *reg = *reg ^ (01 << bit);
           wmove(w,y,x);
           wprintw(w,"%s=%-9d",num_to_binary_string(*reg,len,tmp),*reg);
           wrefresh(w);
        }
        if(k=='0')
        {
           *reg = *reg & ~(01 << bit);
           wmove(w,y,x);
           wprintw(w,"%s=%-9d",num_to_binary_string(*reg,len,tmp),*reg);
           wrefresh(w);
        }

        if(k=='b')if(bit+1<len)bit=bit+2;
        else bit++;
        if(k=='n')return(TRUE);
        if(k=='q')return(FALSE);


    }

}
```

This function steps through the various registers/buses and enables the user to invoke a mini-window-editor on each.

```
change_values(micro_memory,memory,CSAR,CSBR,IC,IX,SP,X,ACC,MAR,MBR,OC,II
,
DATA_BUS,ADDRESS_BUS,ADDER_RIGHT_BUS,ADDER_LEFT_BUS,display,micro_displa
y)
BOOLEAN micro_memory[][LENGTH_OF_MICRO_INSTRUCTIONS];
WORD memory[];
REGISTER *IC,*IX,*SP,*X,*ACC,*MAR,*MBR,*OC,*II,*CSAR;
STRING CSBR[LENGTH_OF_MICRO_INSTRUCTIONS+1];
BUS *ADDER_RIGHT_BUS,*ADDER_LEFT_BUS,*ADDRESS_BUS,*DATA_BUS;
WINDOW display,micro_display;
{
   wmove(display,23,2);
   wprintw(display,"0-1-Toggle-Backward-Forward-Next-Quit---------------
-");
   wprintw(display,"----------------------");

   if(change_reg(display,ACC,2,7,18)==FALSE)goto quit;
   if(change_reg(display,MBR,4,7,18)==FALSE)goto quit;
   if(change_reg(display,MAR,6,7,10)==FALSE)goto quit;
   if(change_reg(display,IC,8,7,10)==FALSE)goto quit;
```

```
    if(change_reg(display,DATA_BUS,2,49,18)==FALSE)goto quit;
    if(change_reg(display,ADDRESS_BUS,4,49,10)==FALSE)goto quit;
    if(change_reg(display,ADDER_LEFT_BUS,6,49,18)==FALSE)goto quit;
    if(change_reg(display,ADDER_RIGHT_BUS,8,49,18)==FALSE)goto quit;

    if(change_reg(display,SP,18,56,10)==FALSE)goto quit;
    if(change_reg(display,IX,20,56,10)==FALSE)goto quit;
    if(change_reg(micro_display,X,7,4,18)==FALSE)goto quit;

quit:
    wmove(display,23,2);
    wprintw(display,"Pause-Continue-Stop-Quiet-Trace-Redraw-Values");
    wprintw(display,"-Microcode-Object-Examine-Help-");
    wrefresh(display);

}
```

This function dumps the contents of a specified number of main memory locations, beginning with location 0, to a specified file.

```
dump_memory(memory,len,file)
WORD memory[];
int len;
STRING file[];
{
    int i,old_loc;
    FILE *fd;
    STRING tmp1[80],tmp2[80];
    FLAG quiet=FALSE,first=TRUE;

    fd=fopen(file,"w");
    for(i=0;i<len;i++)
    if( (memory[i] == old_loc)
                    && ((i+1) < len)
                    && (memory[i] == memory[i+1]) )
    {
    if( ! quiet && ! first)
         fprintf(fd,"    (intermediate locations have the same value)\n");
    quiet=TRUE;
    }
    else
    {
       fprintf(fd,"Location: %4d = %s        Contents: %s = %d\n",
       i,num_to_binary_string(i,ADDRESS_LENGTH,tmp1),
       num_to_binary_string(memory[i],WORD_LENGTH,tmp2),memory[i]);
    old_loc=memory[i];
    quiet=FALSE;
    first=FALSE;
    }
    fclose(fd);

}
```

"I didn't know it," the Knight said, a shade of vexation passing over his face.

This function prints to the display a set of directions to the usage of the program while it is running. This is the on-line documentation.

```
print_help()
{
   printf("This is a computer simulation at the gate, register, and bus
\n");
   printf("level.  The contents of the various registers and buses are
\n");
   printf("displayed after the end of each clock phase, so the values
on\n");
   printf("the screen are always the most current. This program was\n");
   printf("designed to be very interactive.  All commands consist of
a\n");
   printf("single letter which in all cases is the first letter of
the\n");
   printf("command.\nA summary of commands follows:\n\n");
   printf("Pause     - pause between clock cycles, and wait for a
command\n");
   printf("           pause is two-level: a second pause will cause
the\n");
   printf("           machine to pause between clock phases.\n");
   printf("Continue - negate the last pause command.\n");
   printf("Stop      - halt the machine, and take a final memory
dump.\n");
   printf("Quite     - do all things silently, (don't update the
display).\n");
   printf("Trace    - negate the last quite command.\n");
   printf("Redraw   - clear the screen and redraw the display.\n");
   printf("Values   - change the contents of registers/buses.\n");
   printf("Microcode- list the microcode.\n");
   printf("Object    - list the object code of the assembled
program.\n");
   printf("Examine  - list the contents of the entire memory.\n");
   printf("Help     - print this summary.\n\n");
}
```

This function reads in a single-letter-command from the stdin, and invokes the appropriate subroutines/actions according to the command.

```
fork_out(start,pause,show,micro_memory,memory,CSAR,CSBR,IC,IX,SP,X,ACC,M
AR,
        MBR,OC,II,DATA_BUS,ADDRESS_BUS,ADDER_RIGHT_BUS,ADDER_LEFT_BUS,
        display,micro_display)

BOOLEAN micro_memory[][LENGTH_OF_MICRO_INSTRUCTIONS];
WORD memory[];
REGISTER *IC,*IX,*SP,*X,*ACC,*MAR,*MBR,*OC,*II,*CSAR;
STRING CSBR[LENGTH_OF_MICRO_INSTRUCTIONS+1];
BUS *ADDER_RIGHT_BUS,*ADDER_LEFT_BUS,*ADDRESS_BUS,*DATA_BUS;
int *pause,*start,*show;
WINDOW display,micro_display;
{
   int q;

   q=getc(stdin);
   if( q == 's' )*start=FALSE;
   if( q == 'p' )
```

```
         if(*pause < 2)*pause = *pause + 1;
      if( q == 'c' )
         if(*pause > 0)*pause = *pause - 1;
      if( q == 'r' )
         redraw(display,micro_display);
      if( q == 'm' )
      {
         system("clear");
         micro_memory_dump(micro_memory,MICRO_MEMORY_LENGTH);
         system(strcat("more ",MICROCODE_DUMP_FILE));
         printf("(press return to continue)");
         getc(stdin);
         redraw(display,micro_display);
      }
      if( q == 'e' )
      {
         system("clear");
         dump_memory(memory,MEMORY_LENGTH,"memory");
         system("more memory");
         printf("(press return to continue)");
         getc(stdin);
         redraw(display,micro_display);

      }
      if( q == 'h' )
      {
         system("clear");
         print_help();
         printf("(press return to continue)");
         getc(stdin);
         redraw(display,micro_display);
      }
      if( q == 'o' )
      {
         system("clear");
         system(strcat("more ",OBJECT_DUMP_FILE));
         printf("(press return to continue)");
         getc(stdin);
         redraw(display,micro_display);
      }
      if( q == 'q' )*show = FALSE;
      if( q == 't' )*show = TRUE;
      if( q == 'v' )change_values(micro_memory,memory,CSAR,CSBR,IC,IX,SP,X,
                           ACC,MAR,MBR,OC,II,DATA_BUS,ADDRESS_BUS,

   ADDER_RIGHT_BUS,ADDER_LEFT_BUS,display,micro_display);

   }
```

This function sets the terminal mode to cbreak noecho (a key to super-program-control) and redraws the display.

```
   redraw(display,micro_display)
   WINDOW display,micro_display;
   {
      system("stty cbreak -echo");
      wclear(display);
      wclear(micro_display);
```

```
    system("clear");
    box(display,'|','-');
    box(micro_display,'|','-');
    wmove(display,0,10);
    wprintw(display,"Computer-Simulation-by--Gabriel-Robins");
    wprintw(display,"--version-2-of-4/1/83");
    wmove(micro_display,1,33);
    wprintw(micro_display,"Micro Program");
    wmove(micro_display,2,33);
    wprintw(micro_display,"Control Logic");
    wmove(display,23,2);
    wprintw(display,"Pause-Continue-Stop-Quiet-Trace-Redraw-Values-");
    wprintw(display,"Microcode-Object-Examine-Help-");

    wrefresh(display);
    wrefresh(micro_display);

}
```

13.7. Utility Functions Code

"There is nothing to what I could say if I chose," the Duchess replied, in a pleased tone.

This section defines various utility functions used by the rest of the program.

```
#include "defs.h"
```

This function raises one integer to the power of another, as C does not have a built-in function to do that.

```
power(x,n)
int x,n;
{
   int i,p;

   if(n==0)return(1);
   p=1;
   for (i=1;i<=n;++i)p=p*x;
   return(p);
}
```

This function determines if a character is a white-space. A white-space is any character that will not be noticed easily when printed at the display. Examples of white-spaces are blanks, tabs, and carriage returns. The function returns a TRUE if the argument was a white-space and FALSE otherwise. */

```
white_space(c)
int c;
{
   if(c<'!' || c>'~')return(TRUE);
   else return(FALSE);
};
```

This function gets one input line from the named stream and places it into the first argument, which is assumed to be a buffer. This function will skip any line in the stream which contain only white-spaces. Comment lines are also skipped, so comments are invisible to the caller. Comments are assumed to be enclosed within curly braces, and may span several lines.

```
int global_file_line_number;

Getline(buff,stream)
char buff[];
FILE *stream;
{
   extern int global_file_line_number;

   FLAG comment;
   int i=0,c=0;
   BOOLEAN tmp;
```

```
next:
    c=getc(stream);
    if(c==NEWLINE)global_file_line_number++;

    if(c==EOF)
    {
    printf("\n\n======> Fatal Error: End-Of-File was reached.\n");
    printf("Did you forget or misplace the 'end' statement?\n\n2");
    exit(0);
    }
    if(c=='$')
    {
    while(getc(stream)!=NEWLINE);
    global_file_line_number++;
    goto next;
    }
    if(c=='}')
    {
       comment=FALSE;
       goto next;
    }
    if(c=='{')
    {
       comment=TRUE;
       goto next;
    }
    if(comment==TRUE)goto next;
    else if(c!=NEWLINE){
       buff[i++]=c;
       goto next;
    }

    buff[i]='\0';
    tmp=TRUE;
    for(i=0;i<strlen(buff);i++)tmp=tmp&&white_space(buff[i]);
    if(tmp)Getline(buff,stream);
}
```

This function returns the Nth bit out of a byte or a word. It is used heavily by the control subsystem of the simulated machine.

```
bit(reg,bit_position)
REGISTER reg;
int bit_position;
{

    return( ( reg >> bit_position) & 01);

}
```

This function converts a binary-encoded value to a string consisting of the characters '1' and '0' of the specified length. The result is placed into the named buffer.

```
num_to_binary_string(num,length_of_resulting_string,buf)
int num,length_of_resulting_string;
STRING buf[];
{
    int i;
```

```
    for(i=0;i<length_of_resulting_string;i++)
    {
       if( num%2 == 0 ) buf[length_of_resulting_string-1-i]='0';
       else buf[length_of_resulting_string-1-i]='1';
       num=(int)(num/2);
    }
    buf[length_of_resulting_string]=END_OF_STRING;
    return((int)buf);
}
```

This function converts a binary-encoded value into a string consisting of the characters '0' thru '9' of the specified length. The result is placed into the named buffer.

```
num_to_dec_string(num,length_of_resulting_string,buf)
int num,length_of_resulting_string;
STRING buf[];
{
    int i;

    for(i=0;i<length_of_resulting_string;i++)
    {
       buf[length_of_resulting_string-1-i]=(num%10)+48;
       num=(int)(num/10);
    }
    buf[length_of_resulting_string]=END_OF_STRING;
    return((int)buf);
}
```

This function parses an input line gotten by the assembler and determines which are the label, address, and opcode fields. It places the three results into the three named buffers.

```
Parse(buff,label,opcode,address)
char buff[],label[],opcode[],address[];
{
    int i=0,ind=0;

    while(white_space(buff[i]) && i<strlen(buff))i++;
    if(i<ASSEMBLER_LABEL_COLUMN)
       while(!white_space(buff[i]) && i<strlen(buff) &&
                  ind<MAX_LENGTH_OF_LABEL_FIELD)
          label[ind++]=buff[i++];
    label[ind]=END_OF_STRING;
    ind=0;
    while(white_space(buff[i]) && i<strlen(buff))i++;
    if(i<ASSEMBLER_OPCODE_COLUMN)
       while(!white_space(buff[i]) && i<strlen(buff)
                 && ind<MAX_LENGTH_OF_OPCODE_FIELD)
          opcode[ind++]=buff[i++];
    opcode[ind]=END_OF_STRING;
    ind=0;
    while(white_space(buff[i]) && i<strlen(buff))i++;
    if(i<ASSEMBLER_ADDRESS_COLUMN)
       while(!white_space(buff[i]) && i<strlen(buff)
                 && ind<MAX_LENGTH_OF_ADDRESS_FIELD)
          address[ind++]=buff[i++];
    address[ind]=END_OF_STRING;
```

```
}
```

"If I'd meant that, I'd have said it," said Humpty Dumpty.

This function converts a string representing a number in decimal into its binary-encoded form (int).

```
dec_string_to_num(str)
STRING str[];
{
   STRING tmp[80];
   int ans=0,i;
   BOOLEAN neg;

   if(str[0]=='-')
   {
      neg=TRUE;
      strcpy(tmp,str+1);
   }
   else
      {
      neg=FALSE;
      strcpy(tmp,str);
   }
   for(i=0;i<strlen(tmp);i++)ans=ans+power(10,i)*(tmp[strlen(tmp)-1-i]-
48);
   if(neg)ans=(~ans)+1;
   return(ans & eighteen_bits);
}
```

This function converts a string that represent a number in binary into its binary-coded form (int).

```
binary_string_to_num(str)
STRING str[];
{
   int ans=0,i;

   for(i=0;i<strlen(str);i++)ans=ans+power(2,i)*(str[strlen(str)-1-i]-
48);
   return(ans);
}
```

This function dumps the contents of the main memory up to the specified location into a well-known file.

```
memory_dump(memory,highest)
WORD memory[];
int highest;
{
   int i,j;
   FILE *fd;

   fd=fopen(OBJECT_DUMP_FILE,"w");
```

```
   fprintf(fd,"------------------memory-dump--------------------------
\n");

   for(i=0;i<=highest;i++)
   {
      fprintf(fd,"loc %d:",i);
      fprintf(fd,"opcode='");
      for(j=17;j>11;j--)
         if(bit(memory[i],j))fprintf(fd,"1");
         else fprintf(fd,"0");

      fprintf(fd,"' indirection-bit='");
      if(bit(memory[i],11))fprintf(fd,"1");
      else fprintf(fd,"0");
      fprintf(fd,"' index-bit='");
      if(bit(memory[i],10))fprintf(fd,"1");
      else fprintf(fd,"0");
      fprintf(fd,"' address='");
      for(j=9;j>=0;j--)
         if(bit(memory[i],j))fprintf(fd,"1");
         else fprintf(fd,"0");
      fprintf(fd,"' \n");
   }
   fprintf(fd,"-------------------------------------------------------\n\n");
   fclose(fd);
}

grab_mnemonics(opcodes,file)
STRING *opcodes[],file[];
{
  int t,tt,i,j,jj;
  FILE *fd;
  STRING buf[300];
  FLAG cont;
  STRING *malloc();

   printf("Trying to read the assembler mnemonics from the file
'%s'.\n",file);
  fd=fopen(file,"r");
  if(fd==NULL)
  {
  printf("The mnemonics file can not be found.  Goodbye.\n");
  exit(0);
  }
  cont=TRUE;
  i=0;

while(cont)
{
   Getline(buf,fd);

   if(buf[0]!='e')
   {
   tt=0;
   for(t=0;t<strlen(buf);t++)
      if(white_space(buf[t])==FALSE)buf[tt++]=buf[t];
   buf[tt]=END_OF_STRING;
```

```
    j=0;
    while(buf[j] != DOUBLE_QUOTE)j++;
    j++;
    jj=j;
    while(buf[jj] != DOUBLE_QUOTE)jj++;
    buf[jj]=END_OF_STRING;
    opcodes[i]=malloc(strlen(buf+j)+1);
    strcpy(opcodes[i],buf+j);

    i++;
    while(buf[jj] != DOUBLE_QUOTE)jj++;
    jj++;
    j=jj;
    while(buf[j] != DOUBLE_QUOTE)j++;
    buf[j]=END_OF_STRING;
    opcodes[i]=malloc(strlen(buf+jj)+1);
    strcpy(opcodes[i],buf+jj);
    i++;
    }
    else cont=FALSE;
}
printf("%d assembler mnemonics were successfully read.\n",i/2);

return(i/2);

}
```

13.8. Makefile Shell Script Code

"..No, it'll never do to ask: perhaps I shall see it written up somewhere."

```
emula: fns.o assembler.o display.o execute.o load.o main.o defs.h
        cc  fns.o assembler.o display.o execute.o load.o main.o \
                 -lcurses -ltermcap -lg ; mv a.out emula

fns.o: fns.c defs.h
        cc -c  fns.c

assembler.o: assembler.c defs.h
        cc -c  assembler.c

display.o: display.c defs.h
        cc -c  display.c

execute.o: execute.c defs.h
        cc -c  execute.c

load.o: load.c defs.h
        cc -c  load.c

main.o: main.c defs.h
        cc -c  main.c
```

14. Usage Examples

"Why did you call him Tortoise, if he wasn't one?" Alice asked. "We called him Tortoise because he taught us," said the Mock Turtle angrily.

14.1. Sample Micro-program

"Well! I've often seen a cat without a grin," thought Alice; "but a grin without a cat! Its the most curious thing I ever saw in my whole life!"

This section contains the default microcode for the simulated machine:

```
{ initialize the instruction counter and stack pointer to 0 }

alu-left=0 ; alu-right=0 ; data-bus=alu-output ; ic=data-bus; sp=data-
bus;

        { fetch a macro-instruction from the main memory }

fetch:  mar=ic; mbr=mem(mar);

        { transfer the opcode and the indexing and indirection flags and
          increment the instruction counter }

oc=mbr; ii=mbr; mar=mbr; alu-left=ic; alu-right=1; data-bus=alu-output;
$
        ic=data-bus;

{ the following section is a giant 'switch' construct, that decodes the
64
possible opcodes and branches to the appropriate place for the execution
of the corresponding machine instruction }

0-to-63: if bit(oc,5)=1 then goto 32-to-63;
0-to-31: if bit(oc,4)=1 then goto 16-to-31;
0-to-15: if bit(oc,3)=1 then goto 8-to-15;
0-to-7: if bit(oc,2)=1 then goto 4-to-7;
0-to-3: if bit(oc,1)=1 then goto 2-to-3;
0-to-1: if bit(oc,0)=1 then goto 1-to-1;

                { nop - no operation }
0-to-0: goto fetch;

{--------------------------------------------------------------}
                { add - add memory to register }
{--------------------------------------------------------------}

        { see if this instruction requires indexing }

1-to-1: if bit(ii,0)=0 then goto 1-to-1-no-indexing;
```

```
        { preform the indexing }

        data-bus=mar; x=data-bus;
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-
bus;

        { see if this instruction requires indirection }

1-to-1-no-indexing: if bit(ii,1)=0 then goto 1-to-1-no-indirection;

        { perform the indirection }

        mbr=mem(mar);
        mar=mbr;

        { fetch the data from memory }

1-to-1-no-indirection: mbr=mem(mar);

 alu-left=mbr; alu-right=acc; data-bus=alu-output; acc=data-bus;
         goto fetch;

2-to-3: if bit(oc,0)=1 then goto 3-to-3;


{---------------------------------------------------------------------}
                { sub - subtract memory from register }
{---------------------------------------------------------------------}

        { see if this instruction requires indexing }

2-to-2:  if bit(ii,0)=0 then goto 2-to-2-no-indexing;

        { preform the indexing }

        data-bus=mar; x=data-bus;
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-
bus;

        { see if this instruction requires indirection }

2-to-2-no-indexing: if bit(ii,1)=0 then goto 2-to-2-no-indirection;

        { perform the indirection }

        mbr=mem(mar);
        mar=mbr;

        { fetch the data from memory }

2-to-2-no-indirection: mbr=mem(mar);

alu-left=mbr; alu-right=0; invert-left-alu; data-bus=alu-output; x=data-
bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=x; alu-right=acc; data-bus=alu-output; acc=data-bus;
        goto fetch;
```

```
{--------------------------------------------------------------------}
                { lda - load memory into register a }
{--------------------------------------------------------------------}

        { see if this instruction requires indexing }

3-to-3:  if bit(ii,0)=0 then goto 3-to-3-no-indexing;

        { preform the indexing }

        data-bus=mar; x=data-bus;
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-
bus;

        { see if this instruction requires indirection }

3-to-3-no-indexing: if bit(ii,1)=0 then goto 3-to-3-no-indirection;

        { perform the indirection }

        mbr=mem(mar);
        mar=mbr;

        { fetch the data from memory }

3-to-3-no-indirection: mbr=mem(mar);

        data-bus=mbr; acc=data-bus;
        goto fetch;

4-to-7: if bit(oc,1)=1 then goto 6-to-7;
4-to-5: if bit(oc,0)=1 then goto 5-to-5;


{--------------------------------------------------------------------}
                { sta - store register a into memory }
{--------------------------------------------------------------------}

        { see if this instruction requires indexing }

4-to-4:  if bit(ii,0)=0 then goto 4-to-4-no-indexing;

        { preform the indexing }

        data-bus=mar; x=data-bus;
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-
bus;

        { see if this instruction requires indirection }

4-to-4-no-indexing: if bit(ii,1)=0 then goto 4-to-4-no-indirection;

        { perform the indirection }

        mbr=mem(mar);
        mar=mbr;
```

```
        { place the data into memory }

4-to-4-no-indirection: alu-left=acc; alu-right=0; data-bus=alu-output;
$
         mbr=data-bus; mem(mar)=mbr;
        goto fetch;


{-----------------------------------------------------------------------}
                { incr - increment register }
{-----------------------------------------------------------------------}

5-to-5: if bit(mar,0)=0 then goto incr-acc-or-ix;
        if bit(mar,1)=0 then goto incr-sp;
        alu-left=ic; alu-right=1; data-bus=alu-output; ic=data-bus;
        goto fetch;
incr-sp: alu-left=sp; alu-right=1; data-bus=alu-output; sp=data-bus;
        goto fetch;
incr-acc-or-ix: if bit(mar,1)=0 then goto incr-acc;
        alu-left=ix; alu-right=1; data-bus=alu-output; ix=data-bus;
        goto fetch;
incr-acc: alu-left=acc; alu-right=1; data-bus=alu-output; acc=data-bus;
        goto fetch;

6-to-7: if bit(oc,0)=1 then goto 7-to-7;


{-----------------------------------------------------------------------}
                { decr - decrement register }
{-----------------------------------------------------------------------}

6-to-6: if bit(mar,0)=0 then goto decr-acc-or-ix;
        if bit(mar,1)=0 then goto decr-sp;
        alu-left=ic; alu-right=-1; data-bus=alu-output; ic=data-bus;
        goto fetch;
decr-sp: alu-left=sp; alu-right=-1; data-bus=alu-output; sp=data-bus;
        goto fetch;
decr-acc-or-ix: if bit(mar,1)=0 then goto decr-acc;
        alu-left=ix; alu-right=-1; data-bus=alu-output; ix=data-bus;
        goto fetch;
decr-acc: alu-left=acc; alu-right=-1; data-bus=alu-output; acc=data-bus;
        goto fetch;


{-----------------------------------------------------------------------}
                { addai - add to register a immediate }
{-----------------------------------------------------------------------}

7-to-7: data-bus=mar; x=data-bus;
        alu-left=x; alu-right=acc; data-bus=alu-output; acc=data-bus;
        goto fetch;

8-to-15: if bit(oc,2)=1 then goto 12-to-15;
8-to-11: if bit(oc,1)=1 then goto 10-to-11;
8-to-9: if bit(oc,0)=1 then goto 9-to-9;
```

```
{---------------------------------------------------------------------}
                { subai - subtract from register a immediate }
{---------------------------------------------------------------------}

8-to-8: data-bus=mar; x=data-bus;
alu-left=x; alu-right=0; invert-left-alu; data-bus=alu-output; x=data-
bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=x; alu-right=acc; data-bus=alu-output; acc=data-bus;
        goto fetch;


{---------------------------------------------------------------------}
                { addixi - add to register ix immediate }
{---------------------------------------------------------------------}

9-to-9: data-bus=mar; x=data-bus;
        alu-left=x; alu-right=ix; data-bus=alu-output; ix=data-bus;
        goto fetch;

10-to-11: if bit(oc,0)=1 then goto 11-to-11;


{---------------------------------------------------------------------}
                { subixi - subtract from register ix immediate }
{---------------------------------------------------------------------}

10-to-10:  data-bus=mar; x=data-bus;
alu-left=x; alu-right=0; invert-left-alu; data-bus=alu-output; x=data-
bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=x; alu-right=ix; data-bus=alu-output; ix=data-bus;
        goto fetch;


{---------------------------------------------------------------------}
                { addspi - add to register sp immediate }
{---------------------------------------------------------------------}

11-to-11: data-bus=mar; x=data-bus;
        alu-left=x; alu-right=sp; data-bus=alu-output; sp=data-bus;
        goto fetch;

12-to-15: if bit(oc,1)=1 then goto 14-to-15;
12-to-13: if bit(oc,0)=1 then goto 13-to-13;
```

"But she said a great deal more than that!" the White Queen moaned, wringing her hands, "Oh, ever so much more than that!"

```
{---------------------------------------------------------------------}
                { subspi - subtract from register sp immediate }
{---------------------------------------------------------------------}

12-to-12:   data-bus=mar; x=data-bus;
alu-left=x; alu-right=0; invert-left-alu; data-bus=alu-output; x=data-
```

```
bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=x; alu-right=sp; data-bus=alu-output; sp=data-bus;
        goto fetch;


{----------------------------------------------------------------------}
                { addar - add register to acc }
{----------------------------------------------------------------------}

13-to-13: if bit(mar,0)=0 then goto addar-acc-or-ix;
        if bit(mar,1)=0 then goto addar-sp;
        alu-left=acc; alu-right=ic; data-bus=alu-output; acc=data-bus;
        goto fetch;
addar-sp: alu-left=acc; alu-right=sp; data-bus=alu-output; acc=data-bus;
        goto fetch;
addar-acc-or-ix: if bit(mar,1)=0 then goto addar-acc;
        alu-left=acc; alu-right=ix; data-bus=alu-output; acc=data-bus;
        goto fetch;
addar-acc: alu-left=acc; alu-right=acc; data-bus=alu-output; acc=data-
bus;
        goto fetch;

14-to-15: if bit(oc,0)=1 then goto 15-to-15;


{----------------------------------------------------------------------}
                { subar - subtract register from acc }
{----------------------------------------------------------------------}

14-to-14: if bit(mar,0)=0 then goto subar-acc-or-ix;
        if bit(mar,1)=0 then goto a-sp;
alu-left=ic; alu-right=0; invert-left-alu; data-bus=alu-output; x=data-
bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=acc; alu-right=x; data-bus=alu-output; acc=data-bus;
        goto fetch;
a-sp:alu-left=sp;alu-right=0;invert-left-alu;data-bus=alu-output;x=data-
bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=acc; alu-right=x; data-bus=alu-output; acc=data-bus;
        goto fetch;
subar-acc-or-ix: if bit(mar,1)=0 then goto subar-acc;
alu-left=ix; alu-right=0; invert-left-alu; data-bus=alu-output; x=data-
bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=acc; alu-right=x; data-bus=alu-output; acc=data-bus;
        goto fetch;
subar-acc: alu-left=0; alu-right=0; data-bus=alu-output; acc=data-bus;
        goto fetch;


{----------------------------------------------------------------------}
                { addixr - add register to ix }
{----------------------------------------------------------------------}

15-to-15: if bit(mar,0)=0 then goto addixr-acc-or-ix;
        if bit(mar,1)=0 then goto addixr-sp;
```

```
        alu-left=ix; alu-right=ic; data-bus=alu-output; ix=data-bus;
        goto fetch;
addixr-sp: alu-left=ix; alu-right=sp; data-bus=alu-output; ix=data-bus;
        goto fetch;
addixr-acc-or-ix: if bit(mar,1)=0 then goto addixr-acc;
        alu-left=ix; alu-right=ix; data-bus=alu-output; ix=data-bus;
        goto fetch;
addixr-acc: alu-left=ix; alu-right=acc; data-bus=alu-output; ix=data-
bus;
        goto fetch;

16-to-31: if bit(oc,3)=1 then goto 24-to-31;
16-to-23: if bit(oc,2)=1 then goto 20-to-23;
16-to-19: if bit(oc,1)=1 then goto 18-to-19;
16-to-17: if bit(oc,0)=1 then goto 17-to-17;


{-----------------------------------------------------------------------}
                { subixr - subtract register from ix }
{-----------------------------------------------------------------------}

16-to-16: if bit(mar,0)=0 then goto subixr-acc-or-ix;
        if bit(mar,1)=0 then goto subixr-sp;
alu-left=ic; alu-right=0; invert-left-alu; data-bus=alu-output; x=data-
bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=ix; alu-right=x; data-bus=alu-output; ix=data-bus;
        goto fetch;
subixr-sp:alu-left=sp; alu-right=0; invert-left-alu; data-bus=alu-
output;  $
        x=data-bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=ix; alu-right=x; data-bus=alu-output; ix=data-bus;
        goto fetch;
subixr-acc-or-ix: if bit(mar,1)=0 then goto subixr-acc;
        alu-right=0; data-bus=alu-output; ix=data-bus;
        goto fetch;
subixr-acc: alu-left=acc; alu-right=0; invert-left-alu;    $
                data-bus=alu-output; x=data-bus;
        alu-left=x; alu-right=1; data-bus=alu-output; x=data-bus;
        alu-left=ix; alu-right=x; data-bus=alu-output; ix=data-bus;
        goto fetch;


{-----------------------------------------------------------------------}
                { ldar - load a with a register }
{-----------------------------------------------------------------------}

17-to-17: if bit(mar,0)=0 then goto ldar-acc-or-ix;
        if bit(mar,1)=0 then goto ldar-sp;
        alu-left=0; alu-right=ic; data-bus=alu-output; acc=data-bus;
        goto fetch;
ldar-sp: alu-left=0; alu-right=sp; data-bus=alu-output; acc=data-bus;
        goto fetch;
ldar-acc-or-ix: if bit(mar,1)=0 then goto fetch;
        alu-left=0; alu-right=ix; data-bus=alu-output; acc=data-bus;
        goto fetch;
```

```
18-to-19: if bit(oc,0)=1 then goto 19-to-19;

{-----------------------------------------------------------------------}
                 { ldixr - load ix with register }
{-----------------------------------------------------------------------}

18-to-18:  if bit(mar,0)=0 then goto ldixr-acc-or-ix;
        if bit(mar,1)=0 then goto ldar-sp;
        alu-left=0; alu-right=ic; data-bus=alu-output; ix=data-bus;
        goto fetch;
ldixr-sp: alu-left=0; alu-right=sp; data-bus=alu-output; ix=data-bus;
        goto fetch;
ldixr-acc-or-ix: if bit(mar,1)=0 then goto ldixr-acc;
        goto fetch;
ldixr-acc: alu-left=0; alu-right=acc; data-bus=alu-output; ix=data-bus;
        goto fetch;


{-----------------------------------------------------------------------}
                 { ldicr - load ic with register }
{-----------------------------------------------------------------------}

19-to-19:  if bit(mar,0)=0 then goto ldicr-acc-or-ix;
        if bit(mar,1)=0 then goto ldar-sp;
        goto fetch;
ldicr-sp: alu-left=0; alu-right=sp; data-bus=alu-output; ic=data-bus;
        goto fetch;
ldicr-acc-or-ix: if bit(mar,1)=0 then goto ldicr-acc;
        alu-left=0; alu-right=ix; data-bus=alu-output; ic=data-bus;
        goto fetch;
ldicr-acc: alu-left=0; alu-right=acc; data-bus=alu-output; ic=data-bus;
        goto fetch;

20-to-23: if bit(oc,1)=1 then goto 22-to-23;
20-to-21: if bit(oc,0)=1 then goto 21-to-21;


{-----------------------------------------------------------------------}
                 { inva - invert acc }
{-----------------------------------------------------------------------}

20-to-20: alu-left=acc; alu-right=0; invert-left-alu; data-bus=alu-
output; $
                  acc=data-bus;
        goto fetch;


{-----------------------------------------------------------------------}
                 { invix - invert ix }
{-----------------------------------------------------------------------}

21-to-21: alu-left=ix; alu-right=0; invert-left-alu; data-bus=alu-
output; $
                ix=data-bus;
        goto fetch;

22-to-23: if bit(oc,0)=1 then goto 23-to-23;
```

```
{----------------------------------------------------------------}
                { anda - and acc with memory }
{----------------------------------------------------------------}

        { see if this instruction requires indexing }

22-to-22:  if bit(ii,0)=0 then goto 22-to-22-no-indexing;

        { preform the indexing }

        data-bus=mar; x=data-bus;
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-
bus;

        { see if this instruction requires indirection }

22-to-22-no-indexing: if bit(ii,1)=0 then goto 22-to-22-no-indirection;

        { perform the indirection }

        mbr=mem(mar);
        mar=mbr;

        { fetch the data from memory }

22-to-22-no-indirection: mbr=mem(mar);

        x=18;
anda-continue: if bit(mbr,0)=0 then goto anda-next;
        if bit(acc,0)=0 then goto anda-next;
alu-left=acc; alu-right=0; right-shift; data-bus=alu-output; acc=data-
bus;
        alu-left=acc; alu-right=sign; data-bus=alu-output; acc=data-bus;
        goto anda-skip;
anda-next: alu-left=acc; alu-right=0; right-shift; data-bus=alu-output;
$
                acc=data-bus;
anda-skip: alu-left=mbr; alu-right=0; right-shift; data-bus=alu-output;
$
                mbr=data-bus;
        alu-left=x; alu-right=-1; data-bus=alu-output; x=data-bus;
        if bit(zero-detect,0)=0 then go to anda-continue;
        goto fetch;
```

The Red Queen said to Alice "Always speak the truth - think before you speak - and write it down afterwards."

```
{----------------------------------------------------------------}
                { ora - or acc with memory }
{----------------------------------------------------------------}

        { see if this instruction requires indexing }
```

```
23-to-23:  if bit(ii,0)=0 then goto 23-to-23-no-indexing;

        { preform the indexing }

        data-bus=mar; x=data-bus;
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-
bus;

        { see if this instruction requires indirection }

23-to-23-no-indexing: if bit(ii,1)=0 then goto 23-to-23-no-indirection;

        { perform the indirection }

        mbr=mem(mar);
        mar=mbr;

        { fetch the data from memory }

23-to-23-no-indirection: mbr=mem(mar);

        x=18;
ora-continue: if bit(mbr,0)=1 then goto ora-ok;
        if bit(acc,0)=1 then goto ora-ok;
        go to ora-next;
ora-ok: alu-left=acc; alu-right=0; right-shift; data-bus=alu-output;
$
                acc=data-bus;
        alu-left=acc; alu-right=sign; data-bus=alu-output; acc=data-bus;
        goto ora-skip;
ora-next: alu-left=acc; alu-right=0; right-shift; data-bus=alu-output;
$
                acc=data-bus;
ora-skip: alu-left=mbr; alu-right=0; right-shift; data-bus=alu-output;
$
                mbr=data-bus;
        alu-left=x; alu-right=-1; data-bus=alu-output; x=data-bus;
        if bit(zero-detect,0)=0 then go to ora-continue;
        goto fetch;


24-to-31: if bit(oc,2)=1 then goto 28-to-31;
24-to-27: if bit(oc,1)=1 then goto 26-to-27;
24-to-25: if bit(oc,0)=1 then goto 25-to-25;



{----------------------------------------------------------------------}
                { xora - xor acc with memory }
{----------------------------------------------------------------------}

        { see if this instruction requires indexing }

24-to-24:  if bit(ii,0)=0 then goto 24-to-24-no-indexing;

        { preform the indexing }

        data-bus=mar; x=data-bus;
```

```
        alu-right=ix; alu-left=x; address-bus=alu-output; mar=address-
bus;

        { see if this instruction requires indirection }

24-to-24-no-indexing: if bit(ii,1)=0 then goto 24-to-24-no-indirection;

        { perform the indirection }

        mbr=mem(mar);
        mar=mbr;

        { fetch the data from memory }

24-to-24-no-indirection: mbr=mem(mar);

        x=18;
xora-continue: if bit(mbr,0)=1 then goto xora-one;
        if bit(acc,0)=1 then goto xora-ok;
        goto xora-next;
xora-one: if bit(acc,0)=0 then goto xora-ok;
        goto xora-next;
xora-ok: alu-left=acc; alu-right=0; right-shift; data-bus=alu-output;   $
                acc=data-bus;
        alu-left=acc; alu-right=sign; data-bus=alu-output; acc=data-bus;
        goto xora-skip;
xora-next: alu-left=acc; alu-right=0; right-shift; data-bus=alu-output;
$
                acc=data-bus;
xora-skip: alu-left=mbr; alu-right=0; right-shift; data-bus=alu-output;
$
                mbr=data-bus;
        alu-left=x; alu-right=-1; data-bus=alu-output; x=data-bus;
        if bit(zero-detect,0)=0 then go to xora-continue;
        goto fetch;
```

"But what am I to do?" said Alice.
"Anything you like," said the Footman, and began whistling.

```
{-----------------------------------------------------------------}
                { rsfta - right shift acc }
{-----------------------------------------------------------------}

25-to-25: alu-left=acc; alu-right=0; right-shift; data-bus=alu-output;
$
                acc=data-bus;
        goto fetch;

26-to-27: if bit(oc,0)=1 then goto 27-to-27;


{-----------------------------------------------------------------}
                { lsfta - left shift acc }
{-----------------------------------------------------------------}
```

```
26-to-26: alu-left=acc; alu-right=0; left-shift; data-bus=alu-output; $
                acc=data-bus;
        goto fetch;


{-----------------------------------------------------------------------}
                { jmp - jump }
{-----------------------------------------------------------------------}

27-to-27: data-bus=mar; ic=data-bus;
        goto fetch;

28-to-31: if bit(oc,1)=1 then goto 30-to-31;
28-to-29: if bit(oc,0)=1 then goto 29-to-29;


{-----------------------------------------------------------------------}
                { jaz - jump if acc is zero }
{-----------------------------------------------------------------------}

28-to-28: alu-left=acc; alu-right=1; data-bus=alu-output; acc=data-bus;
        if bit(zero-detect,0)=1 then goto fetch;
        data-bus=mar; ic=data-bus;
        goto fetch;


{-----------------------------------------------------------------------}
                { janz - jump if acc is not zero }
{-----------------------------------------------------------------------}

29-to-29: alu-left=acc; alu-right=0; data-bus=alu-output; acc=data-bus;
        if bit(zero-detect,0)=1 then goto fetch;
        data-bus=mar; ic=data-bus;
        goto fetch;

30-to-31: if bit(oc,0)=1 then goto 31-to-31;


{-----------------------------------------------------------------------}
                { jixz - jump if ix is zero }
{-----------------------------------------------------------------------}

30-to-30: alu-left=ix; alu-right=0; data-bus=alu-output; ix=data-bus;
        if bit(zero-detect,0)=0 then goto fetch;
        data-bus=mar; ic=data-bus;
        goto fetch;


{-----------------------------------------------------------------------}
                { jixnz - jump is ix is not zero }
{-----------------------------------------------------------------------}

31-to-31: alu-left=ix; alu-right=0; data-bus=alu-output; ix=data-bus;
        if bit(zero-detect,0)=1 then goto fetch;
        data-bus=mar; ic=data-bus;
        goto fetch;
```

```
32-to-63: if bit(oc,4)=1 then goto 48-to-63;
32-to-47: if bit(oc,3)=1 then goto 40-to-47;
32-to-39: if bit(oc,2)=1 then goto 36-to-39;
32-to-35: if bit(oc,1)=1 then goto 34-to-35;
32-to-33: if bit(oc,0)=1 then goto 33-to-33;


{------------------------------------------------------------------}
                { call - call a subroutine }
{------------------------------------------------------------------}

32-to-32: data-bus=mar; x=data-bus;
alu-left=sp; alu-right=-1; data-bus=alu-output; address-bus=alu-output;
$
                sp=data-bus; mar=address-bus;
        alu-left=ic; alu-right=0; data-bus=alu-output; mbr=data-bus;    $
                mem(mar)=mbr;
        alu-left=x; alu-right=0; data-bus=alu-output; ic=data-bus;
        goto fetch;


{------------------------------------------------------------------}
                { ret - return to caller }
{------------------------------------------------------------------}

33-to-33: alu-left=sp; alu-right=0; address-bus=alu-output;   $
                mar=address-bus; mbr=mem(mar);
        data-bus=mbr; ic=data-bus;
        alu-left=sp; alu-right=1; data-bus=alu-output; sp=data-bus;
        goto fetch;


34-to-35: if bit(oc,0)=1 then goto 35-to-35;


{------------------------------------------------------------------}
                { pusha - push acc onto stack }
{------------------------------------------------------------------}

34-to-34: alu-left=acc; alu-right=0; data-bus=alu-output; mbr=data-bus;
        alu-left=sp; alu-right=-1; data-bus=alu-output;     $
                address-bus=alu-output; mar=address-bus; sp=data-bus;
$
                        mem(mar)=mbr;
        goto fetch;


{------------------------------------------------------------------}
                { popa - pop acc from stack }
{------------------------------------------------------------------}

35-to-35: alu-left=sp; alu-right=0; address-bus=alu-output;    $
                mar=address-bus; mbr=mem(mar);
        data-bus=mbr; acc=data-bus;
        alu-left=sp; alu-right=1; data-bus=alu-output; sp=data-bus;
        goto fetch;

36-to-39: if bit(oc,1)=1 then goto 38-to-39;
```

```
36-to-37: if bit(oc,0)=1 then goto 37-to-37;

{---------------------------------------------------------------------}
                { zeroa - zero out the acc }
{---------------------------------------------------------------------}

36-to-36: alu-left=0; alu-right=0; data-bus=alu-output; acc=data-bus;
        goto fetch;


{---------------------------------------------------------------------}
                { ldai - load acc immediate }
{---------------------------------------------------------------------}

37-to-37: data-bus=mar; acc=data-bus;
        goto fetch;


{---------------------------------------------------------------------}
{ opcodes 388 thru 62 are not currently used.
{---------------------------------------------------------------------}

38-to-39: if bit(oc,0)=1 then goto 39-to-39;
38-to-38: { opcode 38 } goto fetch;

39-to-39: { opcode 39 } goto fetch;

40-to-47: if bit(oc,2)=1 then goto 44-to-47;
40-to-43: if bit(oc,1)=1 then goto 42-to-43;
40-to-41: if bit(oc,0)=1 then goto 41-to-41;
40-to-40: { opcode 40 } goto fetch;

41-to-41: { opcode 41 } goto fetch;

42-to-43: if bit(oc,0)=1 then goto 43-to-43;
42-to-42: { opcode 42 } goto fetch;

43-to-43: { opcode 43 } goto fetch;

44-to-47: if bit(oc,1)=1 then goto 46-to-47;
44-to-45: if bit(oc,0)=1 then goto 45-to-45;
44-to-44: { opcode 44 } goto fetch;

45-to-45: { opcode 45 } goto fetch;

46-to-47: if bit(oc,0)=1 then goto 47-to-47;
46-to-46: { opcode 46 } goto fetch;

47-to-47: { opcode 47 } goto fetch;

48-to-63: if bit(oc,3)=1 then goto 56-to-63;
48-to-55: if bit(oc,2)=1 then goto 52-to-55;
48-to-51: if bit(oc,1)=1 then goto 50-to-51;
48-to-49: if bit(oc,0)=1 then goto 49-to-49;
48-to-48: { opcode 48 } goto fetch;

49-to-49: { opcode 49 } goto fetch;
```

```
50-to-51: if bit(oc,0)=1 then goto 51-to-51;
50-to-50: { opcode 50 } goto fetch;

51-to-51: { opcode 51 } goto fetch;

52-to-55: if bit(oc,1)=1 then goto 54-to-55;
52-to-53: if bit(oc,0)=1 then goto 53-to-53;
52-to-52: { opcode 52 } goto fetch;

53-to-53: { opcode 53 } goto fetch;

54-to-55: if bit(oc,0)=1 then goto 55-to-55;
54-to-54: { opcode 54 } goto fetch;

55-to-55: { opcode 55 } goto fetch;

56-to-63: if bit(oc,2)=1 then goto 60-to-63;
56-to-59: if bit(oc,1)=1 then goto 58-to-59;
56-to-57: if bit(oc,0)=1 then goto 57-to-57;
56-to-56: { opcode 56 } goto fetch;

57-to-57: { opcode 57 } goto fetch;

58-to-59: if bit(oc,0)=1 then goto 59-to-59;
58-to-58: { opcode 58 } goto fetch;

59-to-59: { opcode 59 } goto fetch;

60-to-63: if bit(oc,1)=1 then goto 62-to-63;
60-to-61: if bit(oc,0)=1 then goto 61-to-61;
60-to-60: { opcode 60 } goto fetch;

61-to-61: { opcode 61 } goto fetch;
62-to-63: if bit(oc,0)=1 then goto 63-to-63;
62-to-62: { opcode 62 } goto fetch;
```

"It's a fabulous monster!" the Unicorn cried out, before Alice could reply.

```
{---------------------------------------------------------------------}
                 { hlt - halt the machine }
{---------------------------------------------------------------------}

63-to-63: start=off;
        goto fetch;

   end
```

14.2. Sample Mnemonics

"Must a name mean something?" Alice asked doubtfully.

```
/* 0 */     "nop",      "000000",   /* no operation */
/* 1 */     "add",      "000001",   /* add memory to register acc */
/* 2 */     "sub",      "000010",   /* subtract memory from register acc */
/* 3 */     "lda",      "000011",   /* load memory into register acc */
/* 4 */     "sta",      "000100",   /* store register acc into memory */
/* 5 */     "incr",     "000101",   /* increment register */
/* 6 */     "decr",     "000110",   /* decrement register */
/* 7 */     "addai",    "000111",   /* add to register acc immediate */
/* 8 */     "subai",    "001000",   /* subtract from register acc immediate */
/* 9 */     "addixi",   "001001",   /* add to ix immediate */
/*10 */     "subixi",   "001010",   /* subtract from ix immediate */
/*11 */     "addspi",   "001011",   /* add to sp immediate */
/*12 */     "subspi",   "001100",   /* subtract from sp immediate */
/*13 */     "addar",    "001101",   /* add register to acc */
/*14 */     "subar",    "001110",   /* subtract register from acc */
/*15 */     "addixr",   "001111",   /* add register to ix */
/*16 */     "subixr",   "010000",   /* subtract register from ix */
/*17 */     "ldar",     "010001",   /* load acc with register */
/*18 */     "ldixr",    "010010",   /* load ix with register */
/*19 */     "ldicr",    "010011",   /* load ic with register */
/*20 */     "inva",     "010100",   /* invert acc */
/*21 */     "invix",    "010101",   /* invert ix */
/*22 */     "anda",     "010110",   /* and acc with memory */
/*23 */     "ora",      "010111",   /* or acc with memory */
/*24 */     "xora",     "011000",   /* xor acc with memory */
/*25 */     "rsfta",    "011001",   /* right shift acc */
/*26 */     "lsfta",    "011010",   /* left shift acc */
/*27 */     "jmp",      "011011",   /* jump */
/*28 */     "jaz",      "011100",   /* jump if acc is zero */
/*29 */     "janz",     "011101",   /* jump if acc is not zero */
/*30 */     "jixz",     "011110",   /* jump if ix is zero */
/*31 */     "jixnz",    "011111",   /* jump if ix is not zero */
/*32 */     "call",     "100000",   /* call a subroutine */
/*33 */     "ret",      "100001",   /* return to caller */
/*34 */     "pusha",    "100010",   /* push register acc onto stack */
/*35 */     "popa",     "100011",   /* pop acc from stack */
/*36 */     "zeroa",    "100100",   /* zero out the acc */
/*37 */     "ldai",     "100101",   /* load acc immediate */
/*63 */     "hlt",      "111111",   /* halt the machine */
end
```

14.3. Sample Assembly Program

"It's too late to correct it," said the Red Queen: "When you've once said a thing, that fixes it, and you must take the consequences."

```
{ This program generates the first 25 Fibonacci numbers and places them
in an array in memory locations 50 thru 74 }

max        equ 25        { number of Fibonacci numbers we want }
array      equ 50        { array begins at 50 }
acc        equ 0         { defines the accumulator }
ix         equ 2         { defines the index register }


           call init     { initialize }

fibo       lda -2()      { get the Nth-2 Fibonacci number }
           add  -1()     { add to it the Nth-1 Fibonacci number }
           sta  0()      { store the result into the array }
           incr ix       { increment the index }
           ldai array    {}
           addai max      {}{ see if we have enough Fibonacci numbers
}
           subar ix      {}
           janz  fibo    { if not, go generate some more }
           hlt           { stop the machine }

           org 100       { place the routine starting at loc 100 }

init       ldai array    { initialize the array index }
           ldixr acc
           ldai 1
           sta  0()      { set the 1st Fibonacci number manually }
           incr ix
           sta  0()      { set the 2nd Fibonacci number manually }
           incr ix       { set the array pointer to the 3rd element }
           ret           { return to the caller }

           end           { end of assembly }
```

14.4. Interpreted Microcode Dump

"Its long," said the Knight, "but it's very, very beautiful."

```
-----------------micro-memory-dump-----------------
1 0000000000 0110000000 0100100001 0000000000  0
1 0000000000 0000000000 0000000010 0001000000  1
1 0100000000 0001011100 0100000001 0000000000  2
0 0000000100 0101101001 0000000000 0000000000  3
0 0000000100 0100100100 0111100000 0000000000  4
0 0000000100 0011100010 0110000000 0000000000  5
0 0000000100 0010100001 0100000000 0000000000  6
0 0000000100 0001100000 1001100000 0000000000  7
0 0000000100 0000100000 0101000000 0000000000  8
0 0000000000 0000000000 0000100000 0000000000  9
0 0000000010 0000000000 0110100000 0000000000  10
1 0000000000 0000000000 0000010000 0000000001  11
1 0010000100 0000000000 0010000000 1000000000  12
0 0000000010 0001000000 1000000000 0000000000  13
1 0000000000 0000000000 0000000000 0001000000  14
1 0000000000 0000010000 0000000000 0000000000  15
1 0000000000 0000000000 0000000000 0001000000  16
1 0000000010 0000000010 0100000100 0000000000  17
0 0000000000 0000000000 0000100000 0000000000  18
0 0000000100 0000100000 1111100000 0000000000  19
0 0000000010 0000000000 1011100000 0000000000  20
1 0000000000 0000000000 0000010000 0000000001  21
1 0010000100 0000000000 0010000000 1000000000  22
0 0000000010 0001000000 1101000000 0000000000  23
1 0000000000 0000000000 0000000000 0001000000  24
1 0000000000 0000010000 0000000000 0000000000  25
1 0000000000 0000000000 0000000000 0001000000  26
1 0000000000 0010000010 0100010000 0000001000  27
1 0000000100 0001000000 0100010000 0000000000  28
1 0000000110 0000000000 0100000100 0000000000  29
0 0000000000 0000000000 0000100000 0000000000  30
0 0000000010 0000000001 0001000000 0000000000  31
1 0000000000 0000000000 0000010000 0000000001  32
1 0010000100 0000000000 0010000000 1000000000  33
0 0000000010 0001000001 0010100000 0000000000  34
1 0000000000 0000000000 0000000000 0001000000  35
1 0000000000 0000010000 0000000000 0000000000  36
1 0000000000 0000000000 0000000000 0001000000  37
1 0000000000 0000000000 0001000100 0000000000  38
0 0000000000 0000000000 0000100000 0000000000  39
0 0000000100 0001100001 1110100000 0000000000  40
0 0000000100 0000100001 1001000000 0000000000  41
0 0000000010 0000000001 0110100000 0000000000  42
1 0000000000 0000000000 0000010000 0000000001  43
1 0010000100 0000000000 0010000000 1000000000  44
0 0000000010 0001000001 1000000000 0000000000  45
1 0000000000 0000000000 0000000000 0001000000  46
1 0000000000 0000010000 0000000000 0000000000  47
```

1	0000000001	0010000000	0100000000	0100100000	48
0	0000000000	0000000000	0000100000	0000000000	49
0	0000001000	0000000001	1100000000	0000000000	50
0	0000001000	0001000001	1011000000	0000000000	51
1	0100000000	0001000000	0100000001	0000000000	52
0	0000000000	0000000000	0000100000	0000000000	53
1	0000010000	0001000000	0100100000	0000000000	54
0	0000000000	0000000000	0000100000	0000000000	55
0	0000001000	0001000001	1101100000	0000000000	56
1	0001000000	0001000000	0100000000	0010000000	57
0	0000000000	0000000000	0000100000	0000000000	58
1	0000000001	0001000000	0100000100	0000000000	59
0	0000000000	0000000000	0000100000	0000000000	60
0	0000000100	0000100010	0100100000	0000000000	61
0	0000001000	0000000010	0010000000	0000000000	62
0	0000001000	0001000010	0001000000	0000000000	63
1	0100000000	1000000000	0100000001	0000000000	64
0	0000000000	0000000000	0000100000	0000000000	65
1	0000010000	1000000000	0100100000	0000000000	66
0	0000000000	0000000000	0000100000	0000000000	67
0	0000001000	0001000010	0011100000	0000000000	68
1	0001000000	1000000000	0100000000	0010000000	69
0	0000000000	0000000000	0000100000	0000000000	70
1	0000000001	1000000000	0100000100	0000000000	71
0	0000000000	0000000000	0000100000	0000000000	72
1	0000000000	0000000000	0000010000	0000000001	73
1	0000000110	0000000000	0100000100	0000000000	74
0	0000000000	0000000000	0000100000	0000000000	75
0	0000000100	0010100011	0000000000	0000000000	76
0	0000000100	0001100010	1011100000	0000000000	77
0	0000000100	0000100010	1010000000	0000000000	78
1	0000000000	0000000000	0000010000	0000000001	79
1	0000000100	0010000000	0100010000	0000001000	80
1	0000000100	0001000000	0100010000	0000000000	81
1	0000000110	0000000000	0100000100	0000000000	82
0	0000000000	0000000000	0000100000	0000000000	83
1	0000000000	0000000000	0000010000	0000000001	84
1	0010000100	0000000000	0100000000	0010000000	85
0	0000000000	0000000000	0000100000	0000000000	86
0	0000000100	0000100010	1110100000	0000000000	87
1	0000000000	0000000000	0000010000	0000000001	88
1	0000000100	0010000000	0100010000	0000001000	89
1	0000000100	0001000000	0100010000	0000000000	90
1	0010000100	0000000000	0100000000	0010000000	91
0	0000000000	0000000000	0000100000	0000000000	92
1	0000000000	0000000000	0000010000	0000000001	93
1	0000100100	0000000000	0100100000	0000000000	94
0	0000000000	0000000000	0000100000	0000000000	95
0	0000000100	0001100011	1001000000	0000000000	96
0	0000000100	0000100011	0011100000	0000000000	97
1	0000000000	0000000000	0000010000	0000000001	98
1	0000000100	0010000000	0100010000	0000001000	99
1	0000000100	0001000000	0100010000	0000000000	100
1	0000100100	0000000000	0100100000	0000000000	101
0	0000000000	0000000000	0000100000	0000000000	102
0	0000001000	0000000011	0110100000	0000000000	103
0	0000001000	0001000011	0101100000	0000000000	104
1	1000000001	0000000000	0100000100	0000000000	105

0	0000000000	0000000000	0000100000	0000000000	106
1	0000100001	0000000000	0100000100	0000000000	107
0	0000000000	0000000000	0000100000	0000000000	108
0	0000001000	0001000011	1000000000	0000000000	109
1	0010000001	0000000000	0100000100	0000000000	110
0	0000000000	0000000000	0000100000	0000000000	111
1	0000000011	0000000000	0100000100	0000000000	112
0	0000000000	0000000000	0000100000	0000000000	113
0	0000000100	0000100100	0010000000	0000000000	114
0	0000001000	0000000011	1110100000	0000000000	115
0	0000001000	0001000011	1100100000	0000000000	116
1	0100000000	0010000000	0100010000	0000001000	117
1	0000000100	0001000000	0100010000	0000000000	118
1	0000001001	0000000000	0100000100	0000000000	119
0	0000000000	0000000000	0000100000	0000000000	120
1	0000010000	0010000000	0100010000	0000001000	121
1	0000000100	0001000000	0100010000	0000000000	122
1	0000001001	0000000000	0100000100	0000000000	123
0	0000000000	0000000000	0000100000	0000000000	124
0	0000001000	0001000100	0001000000	0000000000	125
1	0001000000	0010000000	0100010000	0000001000	126
1	0000000100	0001000000	0100010000	0000000000	127
1	0000001001	0000000000	0100000100	0000000000	128
0	0000000000	0000000000	0000100000	0000000000	129
1	0000000000	0110000000	0100000100	0000000000	130
0	0000000000	0000000000	0000100000	0000000000	131
0	0000001000	0000000100	0101000000	0000000000	132
0	0000001000	0001000100	0100000000	0000000000	133
1	1001000000	0000000000	0100000000	0010000000	134
0	0000000000	0000000000	0000100000	0000000000	135
1	0001100000	0000000000	0100000000	0010000000	136
0	0000000000	0000000000	0000100000	0000000000	137
0	0000001000	0001000100	0110100000	0000000000	138
1	0011000000	0000000000	0100000000	0010000000	139
0	0000000000	0000000000	0000100000	0000000000	140
1	0001000010	0000000000	0100000000	0010000000	141
0	0000000000	0000000000	0000100000	0000000000	142
0	0000000100	0011100111	0111000000	0000000000	143
0	0000000100	0010100110	0001000000	0000000000	144
0	0000000100	0001100101	0110100000	0000000000	145
0	0000000100	0000100101	0010000000	0000000000	146
0	0000001000	0000000100	1110100000	0000000000	147
0	0000001000	0001000100	1100100000	0000000000	148
1	0100000000	0010000000	0100010000	0000001000	149
1	0000000100	0001000000	0100010000	0000000000	150
1	0001001000	0000000000	0100000000	0010000000	151
0	0000000000	0000000000	0000100000	0000000000	152
1	0000010000	0010000000	0100010000	0000001000	153
1	0000000100	0001000000	0100010000	0000000000	154
1	0001001000	0000000000	0100000000	0010000000	155
0	0000000000	0000000000	0000100000	0000000000	156
0	0000001000	0001000101	0000000000	0000000000	157
1	0000000000	0010000000	0100000000	0010000000	158
0	0000000000	0000000000	0000100000	0000000000	159
1	0000000001	0010000000	0100010000	0000001000	160
1	0000000100	0001000000	0100010000	0000000000	161
1	0001001000	0000000000	0100000000	0010000000	162
0	0000000000	0000000000	0000100000	0000000000	163

0	0000001000	0000000101	0101000000	0000000000	164
0	0000001000	0001000101	0100000000	0000000000	165
1	1000000000	0100000000	0100000100	0000000000	166
0	0000000000	0000000000	0000100000	0000000000	167
1	0000100000	0100000000	0100000100	0000000000	168
0	0000000000	0000000000	0000100000	0000000000	169

"And thick and fast they came at last, and more, and more, and more"

0	0000001000	0001000000	0000100000	0000000000	170
1	0010000000	0100000000	0100000100	0000000000	171
0	0000000000	0000000000	0000100000	0000000000	172
0	0000000100	0000100101	1100000000	0000000000	173
0	0000001000	0000000101	1010000000	0000000000	174
0	0000001000	0001000101	0100000000	0000000000	175
1	1000000000	0100000000	0100000000	0010000000	176
0	0000000000	0000000000	0000100000	0000000000	177
1	0000100000	0100000000	0100000000	0010000000	178
0	0000000000	0000000000	0000100000	0000000000	179
0	0000001000	0001000101	1011000000	0000000000	180
0	0000000000	0000000000	0000100000	0000000000	181
1	0000000010	0100000000	0100000000	0010000000	182
0	0000000000	0000000000	0000100000	0000000000	183
0	0000001000	0000000101	1110100000	0000000000	184
0	0000001000	0001000101	0100000000	0000000000	185
0	0000000000	0000000000	0000100000	0000000000	186
1	0000100000	0100000000	0100000001	0000000000	187
0	0000000000	0000000000	0000100000	0000000000	188
0	0000001000	0001000110	0000000000	0000000000	189
1	0010000000	0100000000	0100000001	0000000000	190
0	0000000000	0000000000	0000100000	0000000000	191
1	0000000010	0100000000	0100000001	0000000000	192
0	0000000000	0000000000	0000100000	0000000000	193
0	0000000100	0001100110	0100000000	0000000000	194
0	0000000100	0000100110	0011000000	0000000000	195
1	0000000001	0010000000	0100000100	0000001000	196
0	0000000000	0000000000	0000100000	0000000000	197
1	0001000000	0010000000	0100000000	0010001000	198
0	0000000000	0000000000	0000100000	0000000000	199
0	0000000100	0000100110	1101100000	0000000000	200
0	0000000010	0000000110	0110000000	0000000000	201
1	0000000000	0000000000	0000010000	0000000001	202
1	0010000100	0000000000	0010000000	1000000000	203
0	0000000010	0001000110	0111100000	0000000000	204
1	0000000000	0000000000	0000000000	0001000000	205
1	0000000000	0000010000	0000000000	0000000000	206
1	0000000000	0000000000	0000000000	0001000000	207
1	0000000000	0000000000	0000001000	0000000000	208
0	0000010000	0000000110	1011000000	0000000000	209
0	0000100000	0000000110	1011000000	0000000000	210
1	0000000001	0010000000	1100000100	0000000000	211
1	0000000001	0000100000	0100000100	0000000000	212
0	0000000000	0000000000	0000100000	0000000000	213
1	0000000001	0010000000	1100000100	0000000000	214
1	0000000000	0010000010	1100000000	0100000000	215

1	0000000100	1000000000	0100010000	0000000000	216
0	0000000000	0000000110	1000110000	0000000000	217
0	0000000000	0000000000	0000100000	0000000000	218
0	0000000010	0000000110	1111000000	0000000000	219
1	0000000000	0000000000	0000010000	0000000001	220
1	0010000100	0000000000	0010000000	1000000000	221
0	0000000010	0001000111	0000100000	0000000000	222
1	0000000000	0000000000	0000000000	0001000000	223
1	0000000000	0000010000	0000000000	0000000000	224
1	0000000000	0000000000	0000000000	0001000000	225
1	0000000000	0000000000	0000001000	0000000000	226
0	0000010000	0000100111	0011000000	0000000000	227
0	0000100000	0000100111	0011000000	0000000000	228
0	0000000000	0000000000	0000100000	0000000000	229
1	0000000001	0010000000	1100000100	0000000000	230
1	0000000001	0000100000	0100000100	0000000000	231
0	0000000000	0000000000	0000100000	0000000000	232
1	0000000001	0010000000	1100000100	0000000000	233
1	0000000000	0010000010	1100000000	0100000000	234
1	0000000100	1000000000	0100010000	0000000000	235
0	0000000000	0000000111	0001110000	0000000000	236
0	0000000000	0000000000	0000100000	0000000000	237
0	0000000100	0010101000	0110100000	0000000000	238
0	0000000100	0001101000	0100000000	0000000000	239
0	0000000100	0000101000	0011000000	0000000000	240
0	0000000010	0000000111	1010000000	0000000000	241
1	0000000000	0000000000	0000010000	0000000001	242
1	0010000100	0000000000	0010000000	1000000000	243
0	0000000010	0001000111	1011100000	0000000000	244
1	0000000000	0000000000	0000000000	0001000000	245
1	0000000000	0000010000	0000000000	0000000000	246
1	0000000000	0000000000	0000000000	0001000000	247
1	0000000000	0000000000	0000001000	0000000000	248
0	0000010000	0000100111	1110000000	0000000000	249
0	0000100000	0000100111	1111000000	0000000000	250
0	0000000000	0000000000	0000100000	0000000000	251
0	0000100000	0000000111	1111000000	0000000000	252
0	0000000000	0000000000	0000100000	0000000000	253
1	0000000001	0010000000	1100000100	0000000000	254
1	0000000001	0000100000	0100000100	0000000000	255
0	0000000000	0000000000	0000100000	0000000000	256
1	0000000001	0010000000	1100000100	0000000000	257
1	0000000000	0010000010	1100000000	0100000000	258
1	0000000100	1000000000	0100010000	0000000000	259
0	0000000000	0000000111	1100110000	0000000000	260
0	0000000000	0000000000	0000100000	0000000000	261
1	0000000001	0010000000	1100000100	0000000000	262
0	0000000000	0000000000	0000100000	0000000000	263
0	0000000100	0000101000	0101100000	0000000000	264
1	0000000001	0010000001	0100000100	0000000000	265
0	0000000000	0000000000	0000100000	0000000000	266
1	0000000000	0000000000	0000000001	0000000001	267
0	0000000000	0000000000	0000100000	0000000000	268
0	0000000100	0001101000	1011100000	0000000000	269
0	0000000100	0000101000	1001100000	0000000000	270
1	0000000001	0001000000	0100000100	0000000000	271
0	0000000000	0000100000	0000110000	0000000000	272
1	0000000000	0000000000	0000000001	0000000001	273

0	0000000000	0000000000	0000100000	0000000000	274
1	0000000001	0010000000	0100000100	0000000000	275
0	0000000000	0000100000	0000110000	0000000000	276
1	0000000000	0000000000	0000000001	0000000001	277
0	0000000000	0000000000	0000100000	0000000000	278
0	0000000100	0000101000	1110000000	0000000000	279
1	0001000000	0010000000	0100000000	0010000000	280
0	0000000000	0000000000	0000110000	0000000000	281
1	0000000000	0000000000	0000000001	0000000001	282
0	0000000000	0000000000	0000100000	0000000000	283
1	0001000000	0010000000	0100000000	0010000000	284
0	0000000000	0000100000	0000110000	0000000000	285
1	0000000000	0000000000	0000000001	0000000001	286
0	0000000000	0000000000	0000100000	0000000000	287
0	0000000100	0100101010	0111000000	0000000000	288
0	0000000100	0011101001	1111100000	0000000000	289
0	0000000100	0010101001	1011000000	0000000000	290
0	0000000100	0001101001	0111000000	0000000000	291
0	0000000100	0000101001	0101000000	0000000000	292
1	0000000000	0000000000	0000010000	0000000001	293
1	0000010000	1000000000	0110100000	1000000000	294
1	0100000000	0010000000	0100000000	0100100000	295
1	0000000100	0010000000	0100000001	0000000000	296
0	0000000000	0000000000	0000100000	0000000000	297
1	0000010000	0010000000	0010000000	1001000000	298
1	0000000000	0000000000	0001000001	0000000000	299
1	0000010000	0001000000	0100100000	0000000000	300
0	0000000000	0000000000	0000100000	0000000000	301
0	0000000100	0000101001	1001000000	0000000000	302
1	0000000001	0010000000	0100000000	0100000000	303
1	0000010000	1000000000	0110100000	1000100000	304
0	0000000000	0000000000	0000100000	0000000000	305
1	0000010000	0010000000	0010000000	1001000000	306
1	0000000000	0000000000	0001000100	0000000000	307
1	0000010000	0001000000	0100100000	0000000000	308
0	0000000000	0000000000	0000100000	0000000000	309
0	0000000100	0001101001	1110000000	0000000000	310
0	0000000100	0000101001	1101000000	0000000000	311
1	0000000000	0110000000	0100000100	0000000000	312
0	0000000000	0000000000	0000100000	0000000000	313
1	0000000000	0000000000	0000000100	0000000001	314
0	0000000000	0000000000	0000100000	0000000000	315
0	0000000100	0000101001	1111000000	0000000000	316
0	0000000000	0000000000	0000100000	0000000000	317
0	0000000000	0000000000	0000100000	0000000000	318
0	0000000100	0010101010	0011100000	0000000000	319
0	0000000100	0001101010	0010000000	0000000000	320
0	0000000100	0000101010	0001100000	0000000000	321
0	0000000000	0000000000	0000100000	0000000000	322
0	0000000000	0000000000	0000100000	0000000000	323
0	0000000100	0000101010	0011000000	0000000000	324
0	0000000000	0000000000	0000100000	0000000000	325
0	0000000000	0000000000	0000100000	0000000000	326
0	0000000100	0001101010	0101100000	0000000000	327
0	0000000100	0000101010	0101000000	0000000000	328
0	0000000000	0000000000	0000100000	0000000000	329
0	0000000000	0000000000	0000100000	0000000000	330
0	0000000100	0000101010	0110100000	0000000000	331

```
0 0000000000 0000000000 0000100000 0000000000   332
0 0000000000 0000000000 0000100000 0000000000   333
0 0000000100 0011101010 1111000000 0000000000   334
0 0000000100 0010101010 1011100000 0000000000   335
0 0000000100 0001101010 1010000000 0000000000   336
0 0000000100 0000101010 1001100000 0000000000   337
0 0000000000 0000000000 0000100000 0000000000   338
0 0000000000 0000000000 0000100000 0000000000   339
0 0000000100 0000101010 1011000000 0000000000   340
0 0000000000 0000000000 0000100000 0000000000   341
0 0000000000 0000000000 0000100000 0000000000   342
0 0000000100 0001101010 1101100000 0000000000   343
0 0000000100 0000101010 1101000000 0000000000   344
0 0000000000 0000000000 0000100000 0000000000   345
0 0000000000 0000000000 0000100000 0000000000   346
0 0000000100 0000101010 1110100000 0000000000   347
0 0000000000 0000000000 0000100000 0000000000   348
0 0000000000 0000000000 0000100000 0000000000   349
0 0000000100 0010101011 0011000000 0000000000   350
0 0000000100 0001101011 0001100000 0000000000   351
0 0000000100 0000101011 0001000000 0000000000   352
0 0000000000 0000000000 0000100000 0000000000   353
0 0000000000 0000000000 0000100000 0000000000   354
0 0000000100 0000101011 0010100000 0000000000   355
0 0000000000 0000000000 0000100000 0000000000   356
0 0000000000 0000000000 0000100000 0000000000   357
0 0000000100 0001101011 0101000000 0000000000   358
0 0000000100 0000101011 0100100000 0000000000   359
0 0000000000 0000000000 0000100000 0000000000   360
0 0000000000 0000000000 0000100000 0000000000   361
0 0000000100 0000101011 0110000000 0000000000   362
0 0000000000 0000000000 0000100000 0000000000   363
1 0000000000 0000000000 0000000000 0000010000   364
0 0000000000 0000000000 0000100000 0000000000   365
-----------------------------------------------------
```

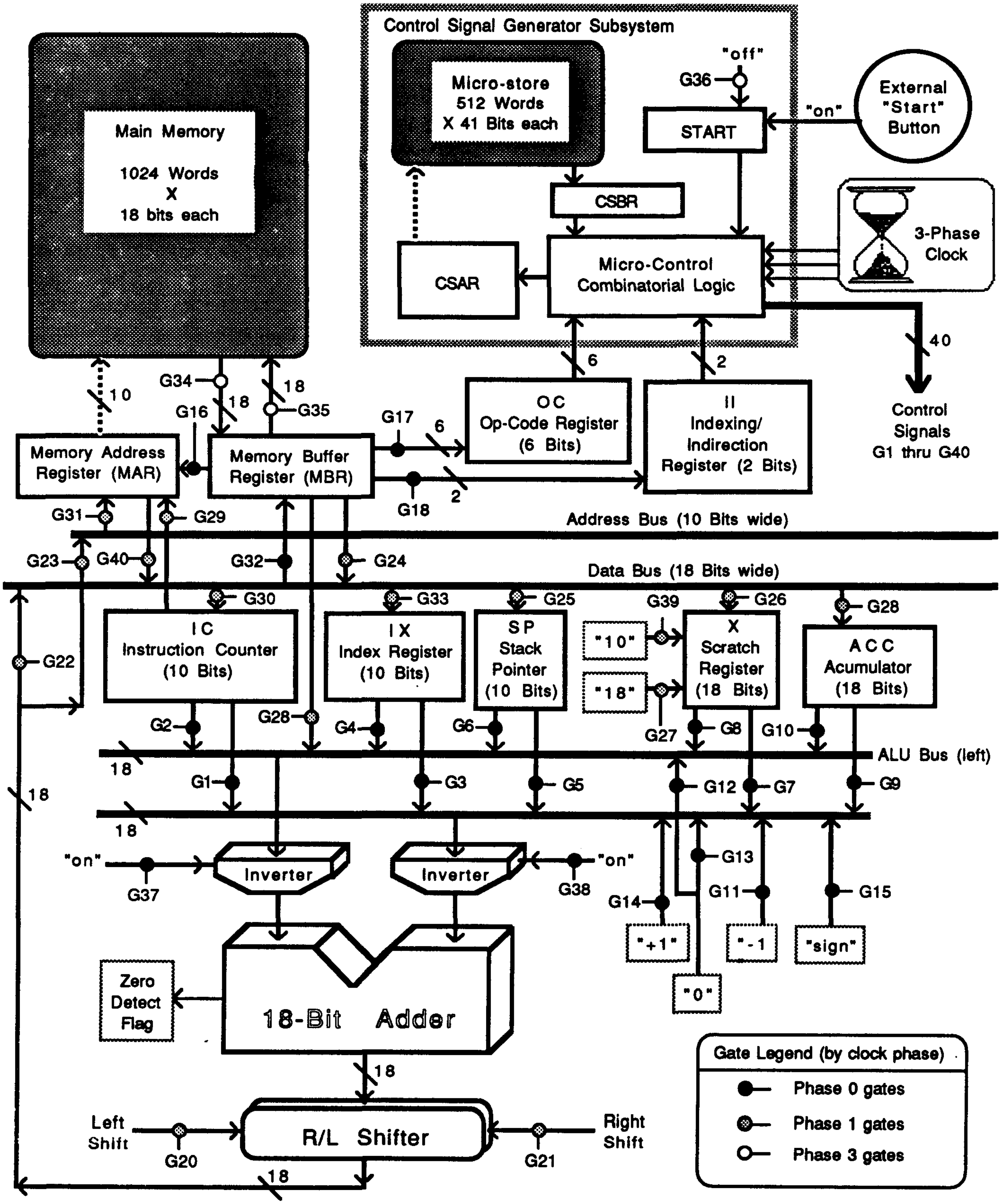
14.5. Main Memory Dump

Alice was just beginning to say "There's a mistake somewhere-"

```
Location:    0 = 0000000000      Contents: 100000000001100100 = 131172
Location:    1 = 0000000001      Contents: 000011011111111110 = 14334
Location:    2 = 0000000010      Contents: 000001011111111111 = 6143
Location:    3 = 0000000011      Contents: 000100010000000000 = 17408
Location:    4 = 0000000100      Contents: 000101000000000010 = 20482
Location:    5 = 0000000101      Contents: 100101000000110010 = 151602
Location:    6 = 0000000110      Contents: 000111000000011001 = 28697
Location:    7 = 0000000111      Contents: 001110000000000010 = 57346
Location:    8 = 0000001000      Contents: 011101000000000001 = 118785
Location:    9 = 0000001001      Contents: 111111000000000000 = 258048
Location:   10 = 0000001010      Contents: 000000000000000000 = 0
   (intermediate locations have the same value)
Location:   49 = 0000110001      Contents: 000000000000000000 = 0
Location:   50 = 0000110010      Contents: 000000000000000001 = 1
Location:   51 = 0000110011      Contents: 000000000000000001 = 1
Location:   52 = 0000110100      Contents: 000000000000000010 = 2
Location:   53 = 0000110101      Contents: 000000000000000011 = 3
Location:   54 = 0000110110      Contents: 000000000000000101 = 5
Location:   55 = 0000110111      Contents: 000000000000001000 = 8
Location:   56 = 0000111000      Contents: 000000000000001101 = 13
Location:   57 = 0000111001      Contents: 000000000000010101 = 21
Location:   58 = 0000111010      Contents: 000000000000100010 = 34
Location:   59 = 0000111011      Contents: 000000000000110111 = 55
Location:   60 = 0000111100      Contents: 000000000001011001 = 89
Location:   61 = 0000111101      Contents: 000000000010010000 = 144
Location:   62 = 0000111110      Contents: 000000000011101001 = 233
Location:   63 = 0000111111      Contents: 000000000101111001 = 377
Location:   64 = 0001000000      Contents: 000000001001100010 = 610
Location:   65 = 0001000001      Contents: 000000001111011011 = 987
Location:   66 = 0001000010      Contents: 000000011000111101 = 1597
Location:   67 = 0001000011      Contents: 000000101000011000 = 2584
Location:   68 = 0001000100      Contents: 000001000001010101 = 4181
Location:   69 = 0001000101      Contents: 000001101001101101 = 6765
Location:   70 = 0001000110      Contents: 000000000000000000 = 0
   (intermediate locations have the same value)
Location:   99 = 0001100011      Contents: 000000000000000000 = 0
Location:  100 = 0001100100      Contents: 100101000000110010 = 151602
Location:  101 = 0001100101      Contents: 010010000000000000 = 73728
Location:  102 = 0001100110      Contents: 100101000000000001 = 151553
Location:  103 = 0001100111      Contents: 000100010000000000 = 17408
Location:  104 = 0001101000      Contents: 000101000000000010 = 20482
Location:  105 = 0001101001      Contents: 000100010000000000 = 17408
Location:  106 = 0001101010      Contents: 000101000000000010 = 20482
Location:  107 = 0001101011      Contents: 100001000000000000 = 135168
Location:  108 = 0001101100      Contents: 000000000000000000 = 0
   (intermediate locations have the same value)
Location: 1022 = 1111111110      Contents: 000000000000000000 = 0
Location: 1023 = 1111111111      Contents: 000000000000000001 = 1
```

15. Appendix II: The Simulated Hardware Diagram

"It's very provoking." Humpty Dumpty said after a long silence.



Control Signal Generator Subsystem
Micro-store 512 Words X 41 Bits each
"off"
G36
"on"
START
External "Start" Button
CSBR
3-Phase Clock
CSAR
Micro-Control Combinatorial Logic
40
Control Signals G1 thru G40
Main Memory
1024 Words X 18 bits each
10
G34
18
G16
18
G35
6
2
G17
6
OC Op-Code Register (6 Bits)
II Indexing/ Indirection Register (2 Bits)
Memory Address Register (MAR)
Memory Buffer Register (MBR)
G18
2
G31
G29
Address Bus (10 Bits wide)
G23
G40
G32
G24
Data Bus (18 Bits wide)
G30
G33
G25
G39
G26
G28
IC Instruction Counter (10 Bits)
IX Index Register (10 Bits)
SP Stack Pointer (10 Bits)
"10"
"18"
X Scratch Register (18 Bits)
ACC Acumulator (18 Bits)
G22
G2
G28
G4
G6
G27
G8
G10
ALU Bus (left)
18
G1
G3
G5
G12
G7
G9
18
18
"on"
G37
Inverter
Inverter
"on"
G38
G13
G14
G11
G15
"+1"
"-1"
"sign"
"0"
Zero Detect Flag
18-Bit Adder
18
Gate Legend (by clock phase)
Phase 0 gates
Phase 1 gates
Phase 3 gates
Left Shift
G20
R/L Shifter
Right Shift
G21
18

16. Table of Contents

"Yes, I think you better leave off," said the Gryphon, and Alice was only too glad to do so.

"The name of the song is called 'Haddock's Eyes.'"
"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is ***called****. The name really is 'The Aged Aged Man.'"*
"Then I ought to have said 'That's what the ***song*** *is called'?" Alice corrected herself.*
"No you oughtn't: that's quite another thing! The ***song*** *is called 'Ways and Means': but that's only what its* ***called****, you know!"*
"Well, what is the song then?" said Alice, who was by this time completely bewildered.
"I was coming to that," the Knight said. "The song really ***is*** *'A-sitting On A Gate': and the tune's my own invention."*