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 'equ' opcode, which is used to associate a label with a number/address, the 'con' pseudo-operator, which is used to store data/constants into memory locations during the assembly process, and the 'org' 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 }

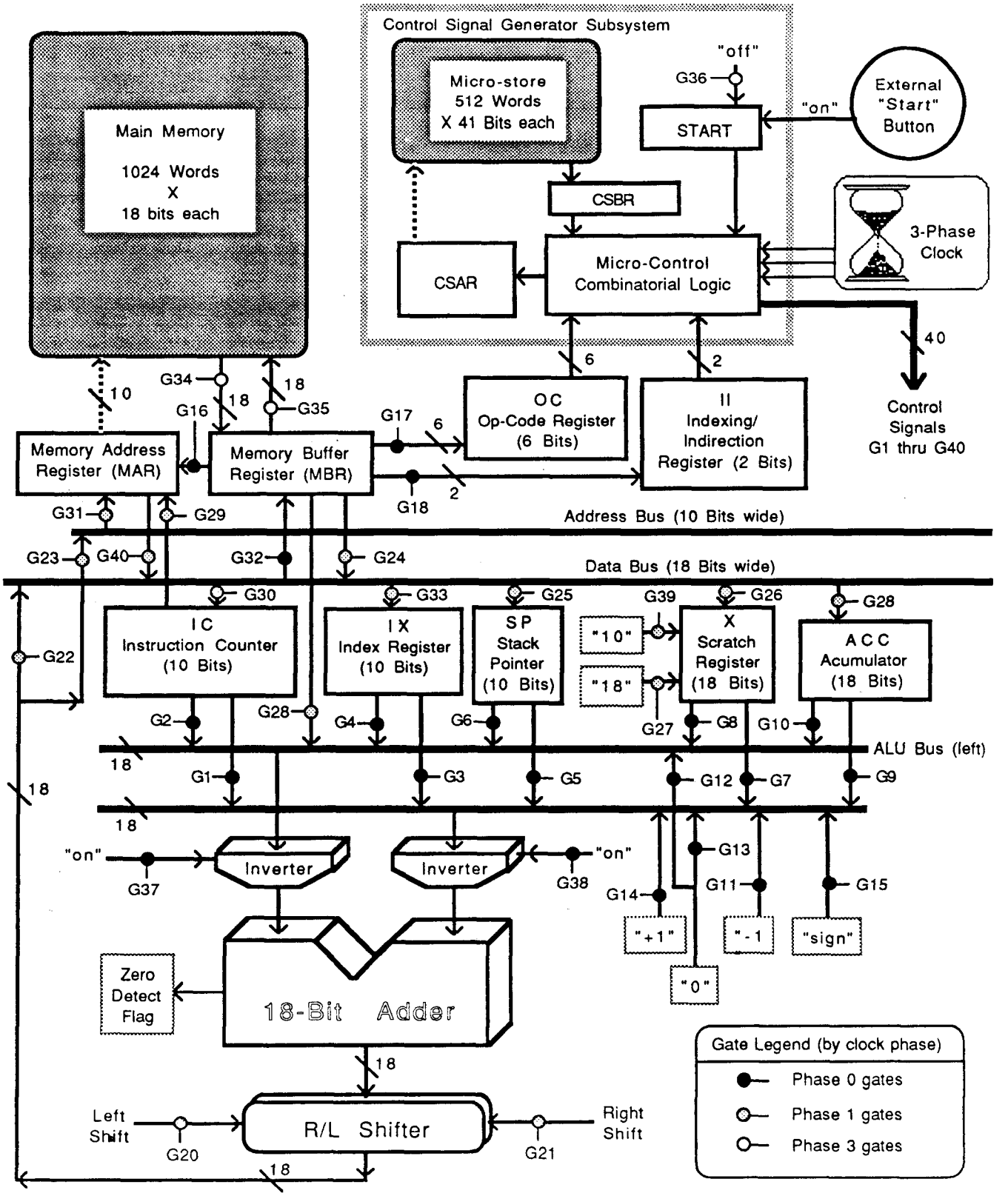
```
max         equ 25          { number of Fibonacci numbers we want }
array       equ 50          { array begins at 50 }
acc         equ 0           { defines the accumulator }
ix          equ 2           { defines the index register }
            call init       { initialize }
fibo        lda -2()        { 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:

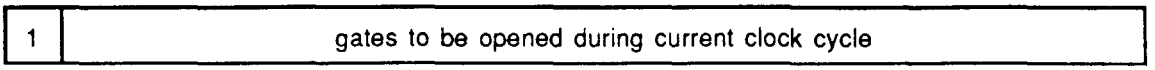
```
Location:    0 = 0000000000       Contents: 1000000000001100100 = 131172
Location:    1 = 0000000001       Contents: 0000110111111111110 = 14334
Location:    2 = 0000000010       Contents: 0000010111111111111 = 6143
Location:    3 = 0000000011       Contents: 0001000100000000000 = 17408
Location:    4 = 0000000100       Contents: 0001010000000000010 = 20482
Location:    5 = 0000000101       Contents: 1001010000000110010 = 151602
Location:    6 = 0000000110       Contents: 0001110000000011001 = 28697
Location:    7 = 0000000111       Contents: 0011100000000000010 = 57346
Location:    8 = 0000001000       Contents: 0111010000000000001 = 118785
Location:    9 = 0000001001       Contents: 1111110000000000000 = 258048
Location:   10 = 0000001010       Contents: 0000000000000000000 = 0
   (intermediate locations have the same value)
Location:   49 = 0000110001       Contents: 0000000000000000000 = 0
Location:   50 = 0000110010       Contents: 0000000000000000001 = 1
Location:   51 = 0000110011       Contents: 0000000000000000001 = 1
Location:   52 = 0000110100       Contents: 0000000000000000010 = 2
Location:   53 = 0000110101       Contents: 0000000000000000011 = 3
Location:   54 = 0000110110       Contents: 0000000000000000101 = 5
Location:   55 = 0000110111       Contents: 0000000000000001000 = 8
Location:   56 = 0000111000       Contents: 0000000000000001101 = 13
Location:   57 = 0000111001       Contents: 0000000000000010101 = 21
Location:   58 = 0000111010       Contents: 0000000000000100010 = 34
Location:   59 = 0000111011       Contents: 0000000000000110111 = 55
Location:   60 = 0000111100       Contents: 0000000000001011001 = 89
Location:   61 = 0000111101       Contents: 0000000000010010000 = 144
Location:   62 = 0000111110       Contents: 0000000000011101001 = 233
Location:   63 = 0000111111       Contents: 0000000000101111001 = 377
Location:   64 = 0001000000       Contents: 0000000001001100010 = 610
Location:   65 = 0001000001       Contents: 0000000001111011011 = 987
Location:   66 = 0001000010       Contents: 0000000011000111101 = 1597
Location:   67 = 0001000011       Contents: 0000000101000011000 = 2584
Location:   68 = 0001000100       Contents: 0000001000001010101 = 4181
Location:   69 = 0001000101       Contents: 0000001101001101101 = 6765
Location:   70 = 0001000110       Contents: 0000000000000000000 = 0
   (intermediate locations have the same value)
Location:   99 = 0001100011       Contents: 0000000000000000000 = 0
Location:  100 = 0001100100       Contents: 1001010000000110010 = 151602
Location:  101 = 0001100101       Contents: 0100100000000000000 = 73728
Location:  102 = 0001100110       Contents: 1001010000000000001 = 151553
Location:  103 = 0001100111       Contents: 0001000100000000000 = 17408
Location:  104 = 0001101000       Contents: 0001010000000000010 = 20482
Location:  105 = 0001101001       Contents: 0001000100000000000 = 17408
Location:  106 = 0001101010       Contents: 0001010000000000010 = 20482
Location:  107 = 0001101011       Contents: 1000010000000000000 = 135168
Location:  108 = 0001101100       Contents: 0000000000000000000 = 0
   (intermediate locations have the same value)
Location: 1022 = 1111111110       Contents: 0000000000000000000 = 0
Location: 1023 = 1111111111       Contents: 0000000000000000001 = 1
```

12. Appendix II: The Hardware Diagram

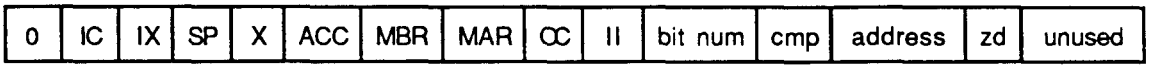

Control Signal Generator Subsystem
Micro-store
512 Words
X 41 Bits each
"off"
G36
START
"on"
External
"Start"
Button
CSBR
3-Phase
Clock
CSAR
Micro-Control
Combinatorial Logic
Main Memory
1024 Words
X
18 bits each
40
Control
Signals
G1 thru G40
6
2
10
G34
18
G16
18
G35
G17
6
OC
Op-Code Register
(6 Bits)
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
18
ALU Bus (left)
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
G21
Right
Shift
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:



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

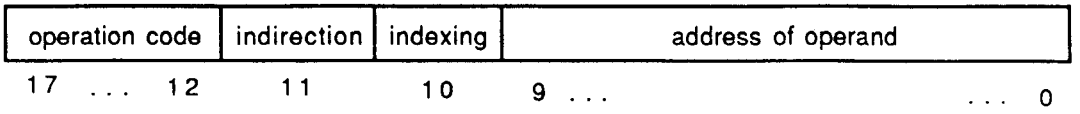


14. Appendix IV: The Default Assembly Instruction Set

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

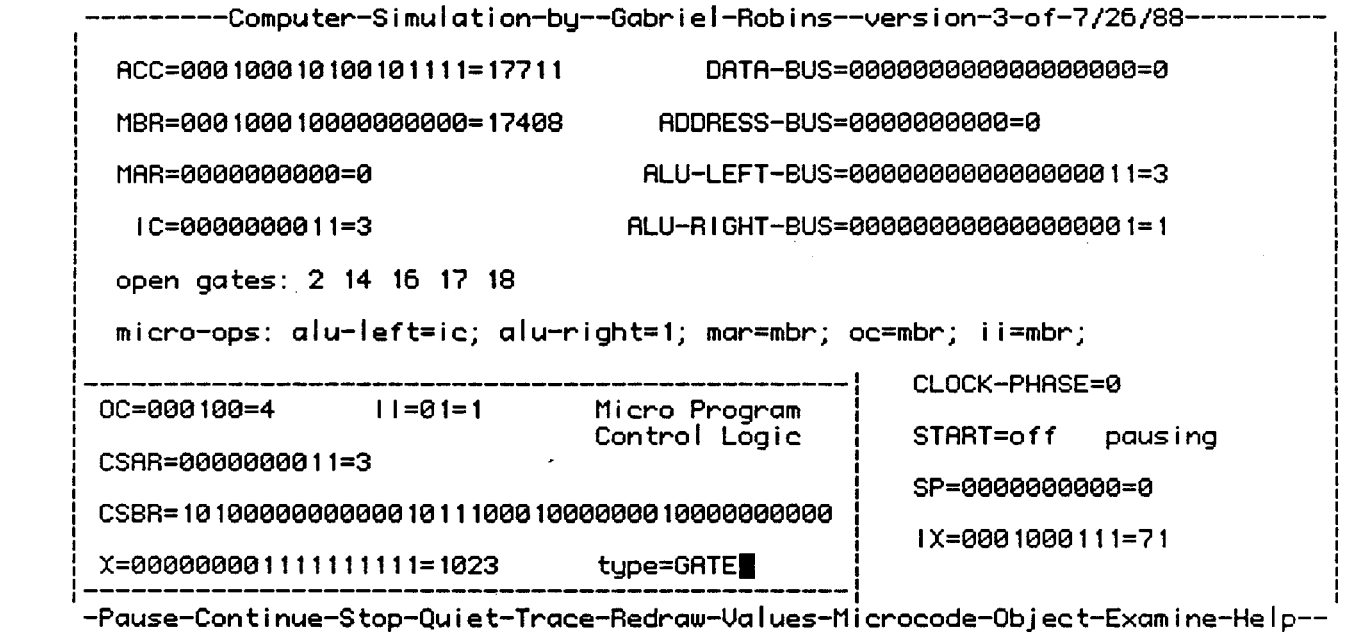
36	zeroa	100100	zero out the acc
37	ldai	100101	load acc immediate
63	hlt	111111	halt the machine

15. Appendix V: The Assembly Instruction Format



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



17. Table of Contents

1.....Abstract 1
2.....Introduction 1
3.....Overview 1
4.....The Hardware 2
4.1.....Registers 2
4.2.....Buses 2
4.3.....Gates 2
4.4.....Memory 2
4.5.....Inverters 2
4.6.....Adder 2
4.7.....Shifter 2
4.8.....Zero-detect logic 3
4.9.....The Control Subsystem 3
4.9.1.....The Micro-memory 3
4.9.2.....Micro-registers 3
4.9.3.....Control Logic 3
4.9.4.....Start Toggle 3
4.9.5.....Clock 3
4.9.6.....Micro-Instruction Format 3
5.....The Assembly Language 4
5.1.....Stack 4
5.2.....Instruction Format 4
5.3.....Assembly Syntax 4
6.....The Microcode Interpreter 4
6.1.....Microcode Syntax 4
7.....The User Interface 5
7.1.....Screen format 5
7.2.....The interaction With the User 5
7.3.....Error Handling 5
8.....The Implementation 5
9.....Summary 6
10.....Bibliography 6
11.....Appendix I: Usage Examples 6
11.1.....Sample Micro-program 6
11.2.....Sample Assembly Program 7
11.3.....Main Memory Dump 8
12.....Appendix II: The Hardware Diagram 8
13.....Appendix III: The Microcode Instruction Format 10
14.....Appendix IV: The Default Assembly Instruction Set 10
15.....Appendix V: The Assembly Instruction Format 11
16.....Appendix VI: The Main Display 11
17.....Table of Contents 12