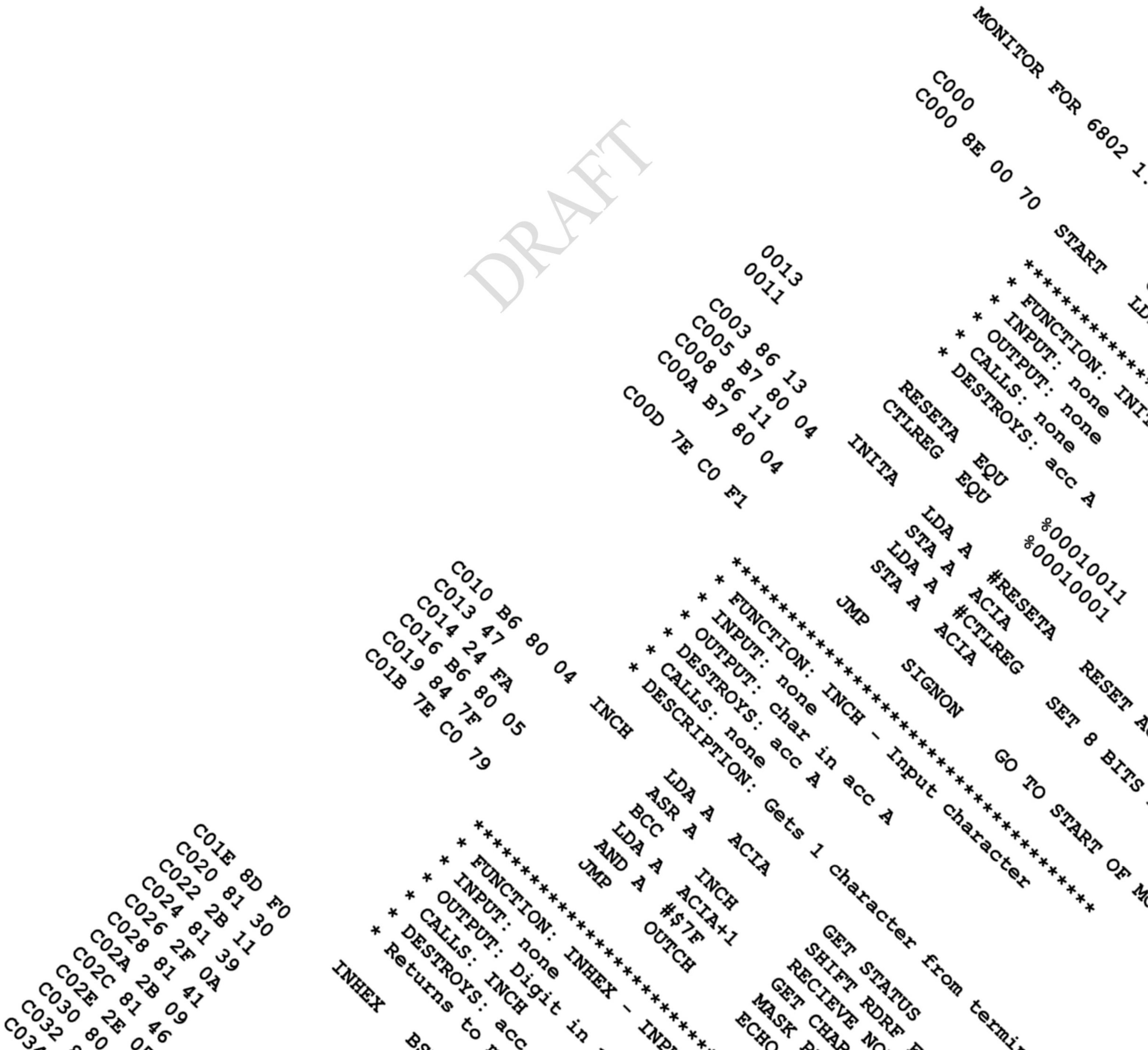
Carboni 1 Minicomputer

Technical Reference

Jules Carboni, 2016-2024

First Edition, 17 November 20XX



```
MONITOR FOR 6802 1.
C000
C000 8E 00 70     START
                  ***********
                  * FUNCTION: INI
                  * INPUT: none
                  * OUTPUT: none
                  * CALLS: none
                  * DESTROYS: acc A
     0013         RESETA  EQU   %00010011
     0011         CTLREG  EQU   %00010001
C003 86 13        INITA   LDA A #RESETA   RESET A
C005 B7 80 04             STA A ACIA
C008 86 11                LDA A #CTLREG   SET 8 BITS
C00A B7 80 04             STA A ACIA
C00D 7E C0 F1             JMP   SIGNON    GO TO START OF M
                  *****************************
                  * FUNCTION: INCH - Input character
                  * INPUT: none
                  * OUTPUT: char in acc A
                  * DESTROYS: acc A
                  * CALLS: none
                  * DESCRIPTION: Gets 1 character from termi
C010 B6 80 04     INCH    LDA A ACIA      GET STATUS
C013 47                   ASR A           SHIFT RDRF
C014 24 FA                BCC   INCH      RECIEVE NO
C016 B6 80 05             LDA A ACIA+1    GET CHAR
C019 84 7F                AND A #$7F      MASK
C01B 7E C0 79             JMP   OUTCH     ECHO
                  ******************
                  * FUNCTION: INHEX - INP
                  * INPUT: none
                  * OUTPUT: Digit in
                  * CALLS: INCH
                  * DESTROYS: acc
                  * Returns to
C01E 8D F0        INHEX   BS
C020 81 30
C022 2B 11
C024 81 39
C026 2F 0A
C028 81 41
C02A 2B 09
C02C 81 46
C02E 2E 0
C030 80
C032
C034
```

Contents

Part 1 — Introduction

1.1 Background

The *Abacus 1* was created as a research project with the purpose of exploring how computers function at the lowest level.

Abacus 2 uses an evolution of the *Abacus 1* architecture (based on Albert Paul Malvino's *SAP-1/2/3* architecture). The computer is Turing complete; it can compute anything computable within the constraints of its memory size and processing speed (explained further in parts 2.5 & 2.2 respectively). The *Abacus 2* instruction set architecture also provides integration with AMD's AM9511 floating point unit.

The physical construction of *Carbon 1* is highly reminiscent of home computers of the 1970s, i.e. *MITS Altair 8800*. The case opens to allow access to the internals. The "slot card" design allows components—called "modules"—to be easily removed for repair or replacement. The front panel uses LEDs to indicate the logical states of every component in the system, e.g. the registers, to assist the operator in debugging.

1.2 For Beginners

This operating manual will describe the architecture, functions and processes of the *Abacus 2*. An understanding of Boolean algebra, electronic logic, number systems (specifically the binary system) is assumed knowledge. It is highly recommended to understand these concepts fully before continuing. To assist beginners, the introductory section of MITS' 'Altair 8800 Operators Manual' has been included in appendix three "Supplementary Information" of this manual. The excerpt details these concepts to a beginner audience in sufficient detail for comprehension of this manual.

1.3 System Specifications

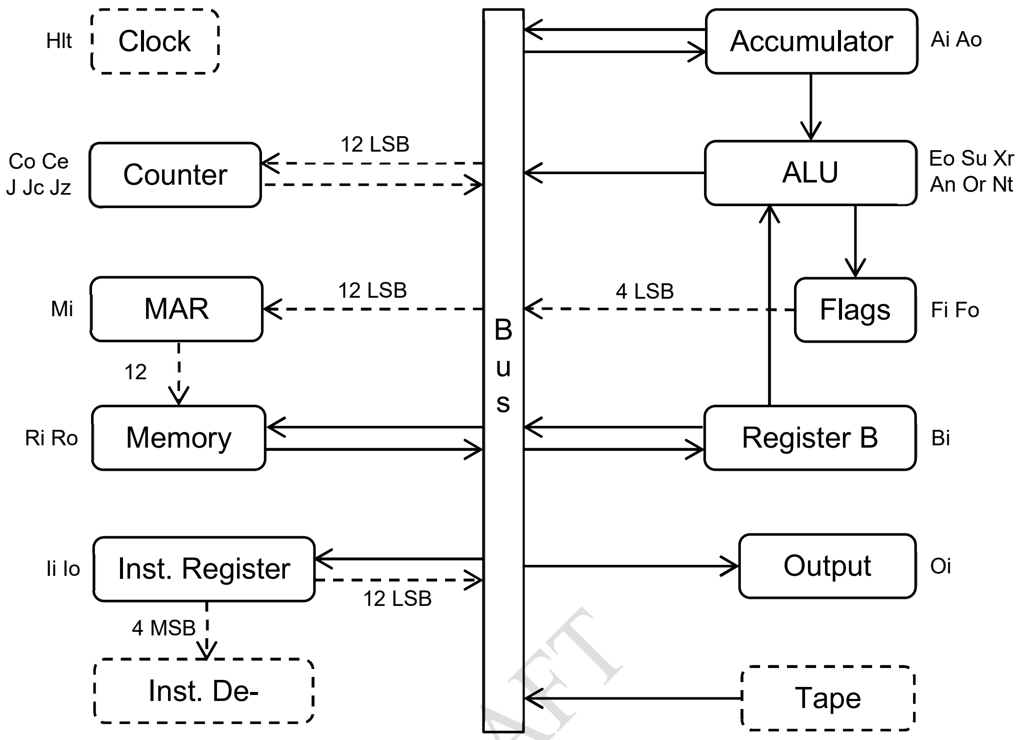
This section of the manual outlines the key technical specifications of the *Abacus 2* processor. For more detail see part two "Design Details".

Power	ATX standard
Architecture	16-bit, CISC, Von Neumann
Instruction Set	*Abacus ISA*, 44 instructions
Clock	FM, digital control, ~1Hz–16MHz
ALU	16-bit, 74181 based
Memory	4KB–16MB, 8-bit bytes, SRAM
Bus	16-bits, shared, internal/external

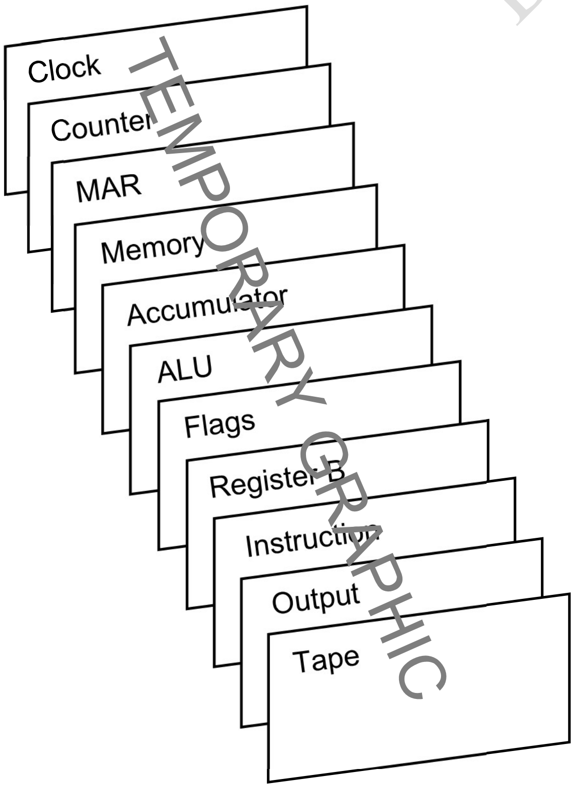
Part 2 — Design Details

2.1 Organisation

The organisation of the *Abacus 1* is based around “modules”. Each major component of the system comprises its own module. Data is shared between modules on the system’s bus.



Below can be observed the order of the *Abacus 1*’s modules as they are assembled.



2.2 Clock

The clock synchronises the actions of the whole system, keeping every component coordinated. Each component of the system activates and operates in time with the pulsing of the clock—the more frequent the clock pulses the faster the system operates.

The *Abacus 1* internal clock has an adjustable clock speed in the range of 0.721Hz to 4810Hz (~5KHz). The duty cycle ranges from 50% high at 0.7Hz to 66.67% high at 4.8KHz. The clock is not regulated by a crystal, using only a *555 timer* integrated circuit.

Each fetch-decode-execute cycle requires two to five clock pulses to complete, resulting in an execution rate in the range of 962 to 2405 instructions per second (IPS), with an approximate weighted average of 1150 IPS.

The Hlt control feeds back to the clock and halts it, so that neither a-stable nor monostable modes will output a clock pulse to the system, effectively freezing the computer until it is reset. This control is tied to the Hlt instruction which can be used in programs.

2.3 Arithmetic Logic Unit

At the heart of every computer is the Arithmetic Logic Unit, a device for performing operations on numbers. The *Abacus 1*'s ALU can perform six functions:

- Arithmetical Addition
- Arithmetical Subtraction
- Logical Exclusive Or
- Logical And
- Logical Or
- Logical Not

Note; all arithmetic operations are performed by addition (subtraction is just inverse addition); this means it is easily possible to write programs to perform multiplication and division, despite these functions not being implemented in hardware. Shift and rotate functions can also be implemented this way.

After the ALU has finished its operation, it will update the Flags Register. This enables the ALU to be used to implement comparative instructions that are not available as explicit instructions in the instruction set, such as “greater than” and “equal to”.

2.4 Registers

Every register in the *Abacus 1* minicomputer serves a special purpose, there are no "general purpose" registers that can be used freely for variable storage—variables must be stored in memory. The unique features and uses of each register will be described in the following parts.

The A Register, known as the accumulator, is imperative to the functionality of the *Abacus 1*. It holds the current value being operated on, particularly during arithmetic and logic operations, and is where the results of these operations are eventually loaded/located. The accumulator is accessible via instructions and can be read and written to directly.

The B Register is electrically identical to the accumulator and holds intermediate or secondary values during operations that require the arithmetic and logic unit. However, unlike the accumulator, the B Register is not accessible to the end user and cannot be read or written to directly.

The Instruction Register takes in the four most significant bits on the bus (the instruction) and relays the value to the instruction decoder/control logic component. This register cannot be accessed by the user at all.

The Memory Address Register will be discussed in 2.5 "Memory" of this manual.

The Program Counter will be discussed in 2.5 "Memory" of this manual.

The Flags Register operates differently to the other registers. It stores the status of four conditions which may or may not be affected by the result of an operation. It is only readable through the Ldf instruction and cannot be read or written to directly. The four status bits are:

- Carry bit—This bit is set to 1 if a carry has occurred during an arithmetical or logical operation. The bit is reset to 0 is no carry occurs.
- Zero bit—This bit is set to 1 if the result of arithmetic or logic instructions is zero and reset to 0 is the result is not zero.
- Sign bit—This bit is set to show the sign of a result. If set to 1, the result is negative; if set to 0 the result is positive. The sign bit reflects the condition of the most significant bit in the result (bit number 15).
- Overflow bit—This bit indicates that an operation incorrectly changed the sign bit of the result. This happens when the result of an arithmetic operation exceeds the maximum value that can be stored in the 16-bit ALU's "sum" register.

The carry bit and zero bit stored in the Flags Register affect the results of instructions that rely on comparisons between numbers, such as conditional jump instructions. The sign bit and overflow bit have no effect on the program/system at all and are used only as indicators to the operator.

The bit widths of the registers in the *Abacus 1* differ:

Register	Bit Width
A	16 bits
B	16 bits
Instruction	4 most significant bits
Memory Address	12 least significant bits
Flags	4 (least significant) bits

Note; most/least significant bits (MSB/LSB) refers to the significant figures of a binary integer. For example, the most significant bit in 1000 is 1.

2.5 Memory

The *Abacus 1* minicomputer contains four kilowords (4,096 words) of random access memory. Each word is 16-bits in length. When instructions are stored in memory, the four most significant bits constitute the instruction and the remaining twelve bits form the address (if necessary). Values can also be stored in memory using the full 16-bits of data per address.

The memory can be read and written to directly with the use of the front panel memory address and memory data switches.

The Memory Address Register takes in the twelve least significant bits on the bus (the address) and relays the value to the (random access) memory module. This register cannot be accessed by the user at all.

The Program Counter is a special 12-bit register which stores the address of the next program step (memory address) to be executed. The Program Counter is automatically advanced to the next sequential program address upon completion of the "fetch phase" of the instruction cycle. The PC is directly writable via jump and conditional jump instructions but cannot be read from.

2.6 External Storage

The *Abacus 1* has a 16-bit system-wide bus. This bus extends beyond the internal components and can be accessed via 16 external pins. The *Abacus 1* is fitted with an optional "tape reader" module that can automate the process of loading programs. A punched tape reader, or any other functionally compatible device, can be plugged into the 16 external bus pins for use with the Tape module.

2.7 Instruction Decoding & Control Logic

The Instruction Register (explained in part 2.4 "Registers") is connected to the bus. When the fetch phase of the instruction cycle commences, the IR reads the contents supplied by the memory and splits it into two sections; the instruction (4 MSB) and the address (12 LSB). It forwards the address (12 LSB) to where it needs to go and increments the Program Counter. All the while, the Instruction Decoder has taken in the instruction (4 MBS) and fed the value into the three EEPROMs that comprise its control logic. The logic activates certain control signals at specific times in the instruction cycle, allowing many unique and complex instructions to be performed depending on the input value.

2.8 Input & Output

The *Abacus 1* can be interfaced with the on-board switches or using a "tape compatible" input device. Its output consists of a five-digit seven-segment display (with one additional digit for signifying negative integers). The display can be hexadecimal or signed or unsigned decimal. Additionally, myriad status indicator LEDs report the contents of every internal register in the system.

Part 3 — Operation Details

3.3 Settings & Modes of Operation

The clock has two operational modes; a-stable “pulse mode” and monostable “step mode”. Pulse mode is considered the normal operating mode, where the clock rhythmically pulses at its set speed. Step mode overrides the internal clock with a non-latching switch labelled Step on the front panel. When Step is pressed, the clock will go high until the switch is released. Note; a logical process is used when switching between modes to ensure the wire never floats.

The LED labelled Clock on the front panel indicates the computer’s current clock speed. The LED labelled Halt indicates whether the computer is halted… If so, the computer has finished executing the program in memory. When Halt is high, the clock will be stopped.

Part 4 — Abacus 1 Instruction Set

4.1 Control Signals

The *Abacus 1* has 15 basic machine language instructions grouped into the following four categories:

- Command Instructions
- Data Transfer Instructions
- Accumulator Instructions
- Branching Instructions

Each instruction is performed over a maximum of five clock pulses through a series of signals sent to many different components of the system. Listed below are all the modules of the *Abacus 1* and their control signals.

Module	Control Signals
Clock	Hlt
Program Counter	Ce, Co, J, Jc, Jz
Instruction Register	Ii, Io
Memory Address Register	Mi
(Random Access) Memory	Ri, Ro
Accumulator	Ai, Ao
Register B	Bi
Arithmetic Logic Unit	Eo, Su, Xr, An, Or, Nt
Flags Register	Fi, Fo

The special control signal Tr (time reset) is activated after an instruction has finished executing if it does not use all five "time" periods. It asynchronously resets the control logic counter back to "time one", beginning the fetch phase of the next instruction cycle.

Listed below are all the instructions of the *Abacus 1* and their active controls for every "time". The first two "times" comprise the "fetch" component of the instruction (fetch-decode-execute) cycle.

Instruction	Active Controls				
	T1	T2	T3	T4	T5
Nop	Mi, Co	Ro, Ii, Ce	Tr	–	–
Lda	Mi, Co	Ro, Ii, Ce	Io, Mi	Ro, Ai	Tr
Add	Mi, Co	Ro, Ii, Ce	Io, Mi	Ro, Bi	Eo, Ai, Fi
Sub	Mi, Co	Ro, Ii, Ce	Io, Mi	Ro, Bi	Eo, Ai, Su, Fi
Sta	Mi, Co	Ro, Ii, Ce	Io, Mi	Ao, Ri	Tr
Ldi	Mi, Co	Ro, Ii, Ce	Io, Ai	Tr	–
Jmp	Mi, Co	Ro, Ii, Ce	Io, J	Tr	–
Jc	Mi, Co	Ro, Ii, Ce	Io, Jc	Tr	–
Jz	Mi, Co	Ro, Ii, Ce	Io, Jz	Tr	–
Xor	Mi, Co	Ro, Ii, Ce	Io, Mi	Ro, Bi	Eo, Ai, Xr, Fi
And	Mi, Co	Ro, Ii, Ce	Io, Mi	Ro, Bi	Eo, Ai, An, Fi
Or	Mi, Co	Ro, Ii, Ce	Io, Mi	Ro, Bi	Eo, Ai, Or, Fi
Not	Mi, Co	Ro, Ii, Ce	Eo, Ai, Nt, Fi	Tr	–
Ldf	Mi, Co	Ro, Ii, Ce	Fo, Ai	Tr	–
Out	Mi, Co	Ro, Ii, Ce	Ao, Oi	Tr	–
Hlt	Mi, Co	Ro, Ii, Ce	Hlt	Tr	–

The physical paths and connections of the control signals can be observed in part 2.1 "Organisation" of this manual.

4.2 Control Instructions

Nop – no operation. **Syntax: 0000.** This instruction carries out the "fetch phase" of the instruction (fetch-decode-execute) cycle but nothing more. It simply increments the program counter, progressing to the next memory location.

Out – output accumulator. **Syntax: 1110.** This instruction takes the current value loaded in the accumulator and pushed it to the display module. The value will be displayed on the five-digit seven-segment display.

Hlt – halt clock. **Syntax: 1111.** This instruction signals the clock to stop pulsing, freezing the computer at "time five"—the Hlt instruction's "execute phase".

4.3 Data Transfer Instructions

Lda – load accumulator. **Syntax: 0001 <Address>.** This instruction takes the value stored at the specified address in memory and loads it into the accumulator. It does not affect the contents in memory.

Ldi – load immediate. **Syntax: 0101 <Integer>.** This instruction takes the value specified (which would normally be an address) and loads it into the accumulator. It cannot load values less than zero or greater than 4095 (0x000-0xfff). This is because it only has 12-bits to work with. It does not access the memory at any time.

Sta – store accumulator. **Syntax: 0100 <Address>.** This instruction takes the value loaded into the accumulator and stores it at the specified address in memory. It will overwrite any memory contents at the specified address.

Ldf – load flags. **Syntax: 1101.** This instruction takes the value stored at the special flags register and loads it into the accumulator. The contents can then be masked to make use of the system's many flags. It does not affect the contents in memory.

4.4 Accumulator Instructions

Accumulator instructions will update the flags register. Note; they can cause overflow errors.

Add – arithmetical addition. **Syntax: 0010 <Address>.** This instruction takes the value stored at the specified address in memory and adds it to the current value loaded in the accumulator.

Sub – arithmetical subtraction. **Syntax: 0011 <Address>.** This instruction takes the value stored at the specified address in memory and subtracts it from the current value loaded in the accumulator. It works by adding the twos-compliment of a number.

Xor – logical exclusive or. **Syntax: 1001 <Address>.** This instruction takes the value stored at the specified address in memory and “exclusively ors” it with the current value loaded in the accumulator.

And – logical and. **Syntax: 1010 <Address>.** This instruction takes the value stored at the specified address in memory and “ands” it with the current value loaded in the accumulator.

Or – logical or. **Syntax: 1011 <Address>.** This instruction takes the value stored at the specified address in memory and “ors” it with the current value loaded in the accumulator.

Not – logical not. **Syntax: 1100.** This instruction “nots” the current value loaded in the accumulator. It does not require any additional parameters.

4.5 Branching Instructions

Jmp – unconditional jump. **Syntax: 0110 <Address>.** This instruction will overwrite the program counter’s value with the value stored at the specified address in memory.

Jc – jump on carry. **Syntax: 0111 <Address>.** This instruction will overwrite the program counter’s value with the value stored at the specified address in memory only when the carry bit is set.

Jz – jump on zero. **Syntax: 1000 <Address>.** This instruction will overwrite the program counter’s value with the value stored at the specified address in memory only when the zero bit is set.

An instruction reference card is included in appendix three “Supplementary Information” of this manual.

Part 5 — Programming Guide

5.1 Fundamentals of Programming

As has become apparent through the exploration of the *Abacus 1* and its control logic, to be meaningfully applied, the computer must be programmed. Sets of digital instructions, called programs, routines, code or software, implement variations in hardware, altering the functionality of the computer. In programming a computer, the operator is effectively wiring an invisible circuit; and so long that the circuit does not change, the process which it carries out will not change. For example, imagine the programmed “circuit” is one that adds numbers; given this “circuit” or process remains the same, the outputs will always be the same for given inputs. Computers are a piece of variable hardware, that can be programmed to imitate a constant piece of hardware reliably.

The software instructions for the *Abacus 1* must be inputted into the machine in the form of sequential 16-bit words known as machine language. Machine code will be discussed in part 5.3.

The basics of computer programming are quite simple. In fact, often the most difficult part of programming is defining the problem you with to solve with the computer. Below are listed the three main steps in generating a program.

1. Defining the problem.
2. Establishing an approach.
3. Writing the program.

Once the problem has been defined, an approach to its solution can be developed. This step is simplified by making a diagram which shows the orderly, step-by-step solution of the problem. Such a diagram is called a flow diagram. After a flow diagram has been made, the various steps can be translated into the computer's language. This is the easiest of the three steps since all you need is a general understanding of the instructions and a list showing each instruction and its machine language equivalent.

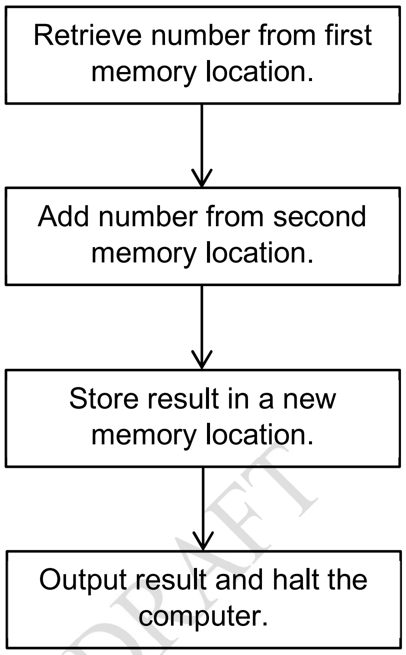
The *Abacus 1* has a small instruction set with extensive capability. For example, a program can cause data to be transferred between the computer's memory and the registers. The program can even cause the computer to make logical decisions. For example, if a specified condition is met, the computer can jump from one place in the program to any other place and continue program execution at the new place.

The *Abacus 1* instruction set was described in detail in part four of this manual.

5.2 A Simple Program

Assume you wish to use the *Abacus 1* to add two numbers located at two different memory locations and store the result elsewhere in the memory. Of course, this is a very simple problem, but it can be used to illustrate several basic programming techniques. Here are the steps used in generating a program to solve this problem:

1. Define the problem—Add two numbers located in memory and store the result elsewhere in memory, then display the number and halt execution.
2. Establish an approach—A flow diagram can now be generated:


Retrieve number from first memory location.
Add number from second memory location.
Store result in a new memory location.
Output result and halt the computer.

3. Write the program—This step will be covered in the next part, part 5.3, after an explanation of assembly language.

Note; many flow diagrams can be created to compare the structure of different approaches to a problem.

5.3 Assembly Language

The software for any computer must be entered the machine in the form of binary words called machine language/code. Machine language programs are generally written with the help of mnemonics which correspond to the bit patterns for various instructions. For example, 0010 is an add instruction for the *Abacus 1* and the corresponding mnemonic is Add. Obviously, the mnemonic Add is much more convenient to remember than its corresponding machine language bit pattern.

Ultimately, however, the machine language bit pattern for each instruction must be entered into the computer one step at a time.

We are now ready to translate the simple program from part 5.2 into assembly language. Translating the flow diagram into a language or format suitable for use by the computer may seem complicated at first. However, a general knowledge of the computer's organization and operation makes the job simple. In this case, the four-part flow diagram translates into five separate instructions:

Line	Instruction	Address
0	Lda (load value into accumulator)	5
1	Add (add value to accumulator)	6
2	Sta (store accumulator to memory)	7
3	Out (output result to display)	null
4	Hlt (halt/stop the computer)	null

Make note of the line numbering, the program begins at line zero. This is because each line of code represents an address in physical memory, which is indexed from zero.

As described in part four, many instructions have arguments that are required for the instruction to execute. For example, the Lda instruction on line zero looks up the value at memory address three and moves it into the accumulator. If the address space were to be empty, the instruction would attempt to load memory address zero into the accumulator, which happens to be itself—a value of 0001—not what we intended.

For this reason, we need to store our values/integers in the memory too. The following lines of this program could look like this.

Line	Instruction	Address
5	137 (the first number)	
6	6200 (the second number)	
7	null (the result will go here)	

The lines of a program that contain values should always be written in the final lines of the program, sequentially after the program is halted. Values stored in memory can take up the whole 16-bits of data at any address, so were these placed in the middle of a program, they could be interpreted as an instruction and cause errors in execution. Shared instruction and data memory is one limitation of the Von Neumann architecture.

5.4 Machine Code

To put it simply, “machine code” is ones and zeroes. The mnemonics of assembly language are directly interchangeable with the raw binary instructions of machine code. This makes “assembling programs” simple.

Assembled, the program from part 5.3 would be written like this:

Line	Instruction	Address
… 0000	0001	0000 0000 0101
… 0001	0010	0000 0000 0110
… 0010	0100	0000 0000 0111
… 0011	1110	0000 0000 0000
… 0100	1111	0000 0000 0000
… 0101	0000	0000 1000 1001
… 0110	0001	1000 0011 1000
… 0111	0000	0000 0000 0000

Make note of how binary representation of the decimal value 6200 flows into the instruction space.

The binary view of the program is not elegant nor space efficient. To neaten things up we can use hexadecimal notation to display this program; where in the contents the first digit represents the 4-bit instruction and the following three represent the 12-bit address.

Memory Address	Memory Contents
000	1 005
001	2 006
002	4 007
003	E 000
004	F 000
005	0 089
006	1 838
007	0 000

Congratulations, you’ve just written your first program for the *Abacus 1* minicomputer! This code can be loaded into the memory of the computer and executed. Additionally, the code can be stored on paper or punch tape for storage outside of the volatile system memory for later recall/usage.

5.5 Debugging

Occasionally, the code loaded into the computer's memory will not execute as expected. This is most likely the result of an error, called a "bug", in the program. Rarely, this can be the result of a hardware failure; appendix one details the proper method of identifying and repairing faulty hardware. A bug often arises at three times in the process of programming a computer:

- When establishing an approach.
- When writing the program.
- When loading the program into memory.

There are a few things that will make debugging your code easier. These include:

- The program's flow diagram.
- The program itself, written in assembly code.
- The instruction set reference, including all control signals (see part four).

The process of debugging can be both time consuming and infuriating; patience is essential. Rest assured, finally executing your program without error will be a very satisfying and rewarding experience. To successfully debug a program, follow these steps:

1. Switch the clock mode from Pulse to Step; this ensures that the computer will not run before we're ready.
2. Load the program from scratch; sometimes a "buggy" program can alter its own code, further worsening the problem.
3. Reset the computer by pressing the Reset switch. Initialisation of the computer is done.
4. Look up the current instruction in your program and make note of the controls that it activates at each "time". The clock should be at "time zero", the first half of the "fetch phase". It is safe to step the clock once, moving to "time one", the second half of the "fetch phase". If there are any erroneous controls at this point, there is an error in the computer's microcode; see appendix 1.4 "Bad Microcode".
5. Step the clock by one, progressing to "time two". Check that the instruction in the Instruction Register is the same as the one in your program. If not, there is an error in translation from assembly language to machine code.
6. Compare the active controls to that of the instruction to check that there are no errors in the Microcode. If there are any erroneous controls at this point, there is an error in the computer's microcode; see appendix 1.4 "Bad Microcode".
7. Repeat steps five to six until at "time four".
8. Compare the result of the instruction to the expected results described in your flow diagram and program (check register indicators). If there is a discrepancy, there may be an error in the usage of certain instructions, the implementation of code from the flow diagram, or simply a flow diagram that does not describe the process desired for execution.
9. Repeat steps four to eight until a bug is found. If the program is not to blame for the erroneous operation of the computer, there may be an error in the wiring of the computer's circuitry; see appendix 1.3 "Bad Circuitry".

5.6 Sample Programs

Note; "–" denotes a variable value that is supplied by the programmer at the time of assembly.

Fibonacci Sequence:

#	Assembly			Code	Comments
	Top:				
0		Ldi	0	7000	Load 0 into accumulator (Acc).
1		Sta	X	400D	Store Acc value in X variable.
2		Out		E000	Output Acc.
3		Ldi	1	5001	Load 1 into Acc.
	Next:				
4		Sta	Y	400E	Store Acc in Y variable.
5		Out		E000	Output Acc.
6		Add	X	200D	Add Y to Acc.
7		Jc	Top	7000	If carry, start over (jump to "Top").
8		Sta	Z	400F	Store Acc in Z variable.
9		Lda	Y	100E	Load Y into Acc.
10		Sta	X	400D	Store Acc into X.
11		Lda	Z	100F	Load Z into Acc.
12		Jmp	Next	6004	Jump to "Next".
13	X;	0		0000	
14	Y;	0		0000	
15	Z;	0		0000	

Multiplication:

#	Assembly			Code	Comments
0		Lda	Y	100E	
1		Sta	Count	4010	Store multiplier (Y) as Count.
	Top:				
2		Lda	Count	1010	
3		Sub	One	300D	For every addition, decrement Count.
4		Jc	Next	7008	If Count > 0, go to addition.
5		Lda	Result	1011	
6		Out		E000	Output result.
7		Hlt		F000	
	Next:				
8		Sta	Count	4010	Store decremented Count value.
9		Lda	Result	1011	Load previous sum.
10		Add	X	200F	Add multiplicand.
11		Sta	Result	4011	Store sum as Result.
12		Jmp	Top	6002	Do it all again.
13	One;	1		0001	
14	X;	–		–	Multiplicand variable.
15	Y;	–		–	Multiplier variable.
16	Count;	0		0000	
17	Result;	0		0000	Product value.

5.6 Sample Programs (continued)

Loop Counter:

#	Assembly			Code	Comments
	Top:				
0		Add	Inc	2003	Increment by “inc” value.
1		Out		E000	Display number.
2		Jmp	Top	6000	Do it again.
3	Inc;	1		0001	Value to increment by.

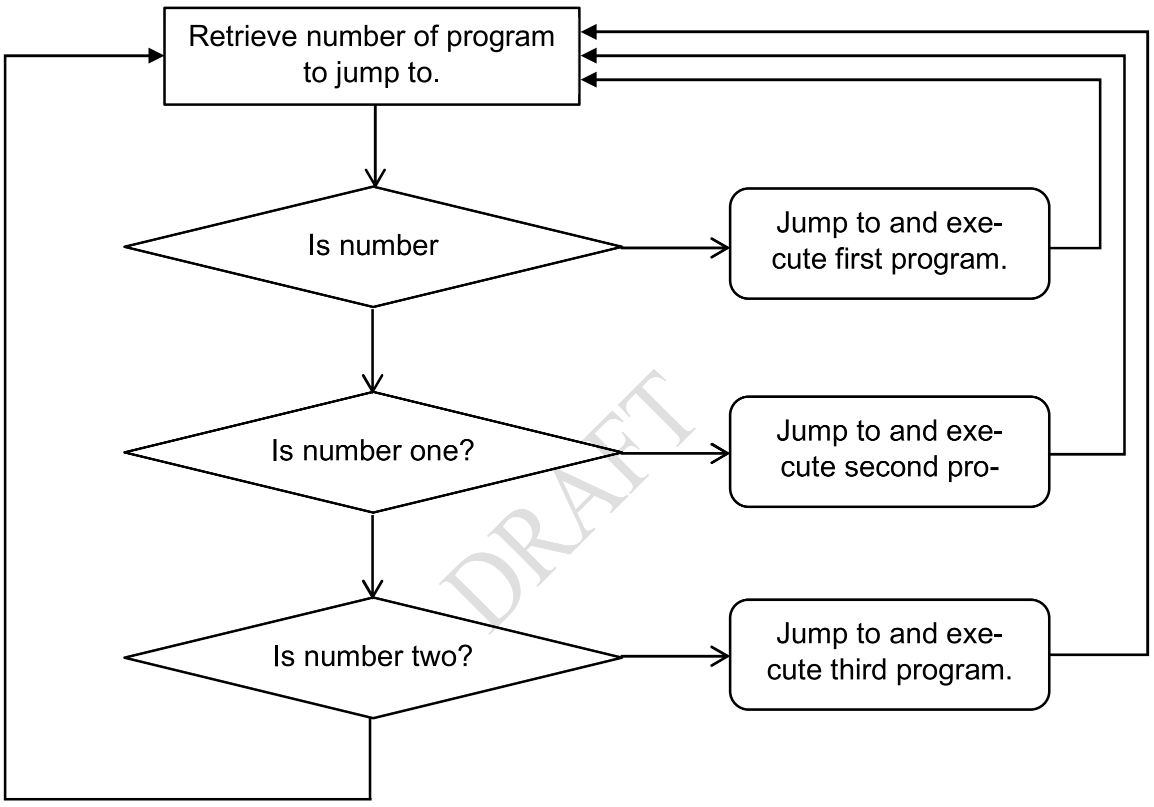
Floor Division with Modulo:

#	Assembly			Code	Comments
	Top:				
0		Ldi	0	7000	
1		Or	Num	A012	
2		Sta	Res	4016	Res = Num.
3		Jz	Result	800E	If Num == 0, skip division.
4		Ldi	1	5001	
5		Add	Cnt	2014	Increment Cnt.
6		Sta	Cnt	4014	
7		Lda	Res	1016	
8		Sub	Den	3013	
9		Jc	Modulo	700C	If Num < Den, store Mod and output Cnt.
10		Sta	Res	4016	Store current value as Res.
11		Jmp	Top	6000	Do it again.
	Modulo:				
12		Sta	Mod	4015	Mod = Res.
13		Nop		0000	Optionally output modulo.
	Result:				
14		Lda	Cnt	1014	
15		Sta	Res	4016	Res = Cnt.
16		Out		E000	Output result.
	End:				
17		Hlt		F000	End of program.
18	Num;	–		–	Numerator/Dividend variable.
19	Den;	–		–	Denominator/Divisor variable.
20	Cnt;	0		0000	
21	Mod;	0		0000	Modulus value.
22	Res;	0		0000	Quotient value.

5.7 Concept of Operating Systems

Traditionally, a program is loaded into a computer's memory and is executed until completed, at which point the computer halts. Only one program can be loaded into a computer at once and the computer must be reprogrammed before each use. Operating systems can be used in computers to elevate this limitation, allowing the user to select from hundreds of programs which have all been loaded into memory at once.

The simplest form of an operating system is a small program located at the beginning of the system's memory which, referencing a user-defined value, can jump to the location of and execute many other programs. This can be represented in a flow diagram:


Retrieve number of program to jump to.
Is number
Jump to and execute first program.
Is number one?
Jump to and execute second pro-
Is number two?
Jump to and execute third program.

Appendix 1 — Maintenance Information

Appendix 2 — Technical Information

Appendix 3 — Supplementary Information