CFT Minicomputer Programming Guide

Alexios Chouchoulas
alexios@bedroomlan.org

Abstract

This document discusses the architecture of the CFT computer from a programmer's perspective. The CFT is a solid-state, 16-bit, microcoded architecture reminiscent, among others, of the DEC PDP-8. The computer incorporates a 16-bit word width with separate memory and input/output addressing spaces and a minimal, orthogonal instruction set that is still particularly versatile. The design includes separate internal (processor) and external (peripheral) buses and is extensible both via processor extensions and peripherals.
A brief explanation of the architecture is provided, along with a discussion of its programming model, instruction set, and limitations. Short examples of CFT Assembly code are provided, along with a complete opcode table with semantics and timing information.

History of Changes

2011-10-10 First draft compiled from notes, hardware description and schematics.
2011-10-18 Fixed hexadecimal opcode of macro `ING` in the instruction table. Corrected minor semantic and typesetting errors.

1 Introduction

This document discusses the architecture of the CFT computer from a programmer's perspective. The CFT is a solid-state, 16-bit architecture reminiscent, among others, of the DEC PDP-8.[1]

The computer incorporates a 16-bit word, 65,536 words of addressable main memory and 65,536 words of input/output (I/O) space.

Instructions are 16-bits. There are seven registers and an simple, orthogonal instruction set. The computer is able to initialise itself without operator intervention using read-only ROM and a hardwired, turn-key bootstrap address. Communication to the outside world is attained using an interrupt facility, as part of an expansion bus that connects to main memory and peripherals. An autoindexing feature allows for very tightly coded looping structures.

The design is microcoded for ease of upgrading.

2 Architecture

The CFT is a Von Neumann architecture stored program computer. Data is primarily manipulated in memory, using the systems' single general-purpose register as an intermediate. The design is such that up to 1,024 general purpose registers may be accessed at any time. The computer is completely solid-state, implemented using modern 74xxx integrated circuits and some LSI memory.

The computer is microcoded. This simplifies upgrading and patching the architecture to provide new features. Microcode consists of a number of microprograms, each handling one CFT major state or instruction. Each microprogram consists of up to 16 24-bit microinstructions which operate directly the various physical units in the computer.

The entire architecture is built around a 16-bit word. All data and instructions is stored in 16-bit quantities,

[1] Newer versions of this document and additional documentation and downloads may be found at the following URL: www.bedroomlan.org/hardware/cft.

and all registers except two are 16 bits wide. This allows programs and data to share the same 16-bit memory, and also makes programming significantly easier. An rational, orthogonal instruction set with 15 main instructions and a number of sub-instructions allows for ease of programming, yet significant versatility.

The computer has separate address spaces for memory and input/output devices, allowing the two to be kept separate. Through microcode alterations additional wait states may be introduced for I/O, while leaving memory accesses fast. This also allows processor extensions to be built in I/O space, such as multiplication and division units, novel instructions, et cetera.

2.1 Word Size

The word size is 16 bits. There are no facilities for accessing quantities smaller than one word, and no single-instruction facilities for accessing quantities longer than one word.

In this context, one kiloword (KWord) is 2^{10} or 1,024 kilowords.

2.2 Data Types

CFT defines a single data type: a 16-bit unsigned integer, i.e. integers in the range 0–65,535. There are no instructions explicitly intended to handle signed numbers, but two's complement operations may still be performed and the hardware has been designed to make this easier.

2.3 Addressing

The CFT architecture can address up to 64 KWords of memory, plus up to 64 KWords of I/O space[2].

2.3.1 Memory Space

Main memory is split up into 64 *pages*, each 1 KWord in size. Instructions usually reference memory addresses relative to the page they are executing in.

The first page, Page Zero, is given special treatment by the instruction set. If bit 10 (starting at zero) of an instruction is clear, addresses in Page Zero may be accessed. This is the fastest and most convenient way of accessing memory beyond the current page. As such, Page Zero is always used for system variables, constants and other data that must be globally accessible.

The 128 words in Zero Page addresses `0080-00FF` (inclusive) are so-called Autoindex registers. Using the Autoindex addressing mode, each time one of these addresses is referenced, it is automatically incremented by one. This allows loops to be coded very tightly, with index register increments happening at the microcode level.

`0000`	`SUBRET` — return address for last `JSR`.
`0001`	`TRAPRET` — return address for last `TRAP`
`0002`	`ISRRET` — return address for last ISR.
`0080`	`IR0` — first autoindex register.
`00FF`	`IR7F` — last (128th) autoindex register.
⋮	⋮
`03FF`	Last word of Page Zero.
`1000`	First word of Page 1.
⋮	⋮
`8000`	First word of Page 32.
⋮	⋮
`FC00`	First word of Page 63.
`FFF0`	Boot/reset address.
`FFF8`	Interrupt service routine.
`FFFF`	Highest memory address.

Figure 1: Memory map of the CFT CPU's memory space.

2.3.2 I/O Space

Although there are 64 KWords of I/O space, this is limited by practical limitations: Only the first 1,024 I/O addresses may be accessed from anywhere in memory space. Due to page-relative addressing, accessing the rest of the I/O space is mostly convenient via indirect addressing. For speed and ease of use, most I/O devices will thus occupy the first KWord of address space. Page-relative addressing is discussed in detail in section 2.6 (p. 3).

2.4 Registers

There are seven registers in the CFT architecture.

2.4.1 Accumulator (**A**)

The Accumulator (`A`) is a 16 bit register. It is the only 16-bit register directly and fully accessible via the instruction set and the one almost all instructions operate on.

2.4.2 Link Register (**L**)

The Link Register (`L`) is a single-bit register. It is used as a carry or overflow bit during arithmetic and as the 17th bit during roll instructions. It may be used as a generic flag by user programs.

[2] Although, for practical reasons, 1 KWord of I/O space is more readily available.

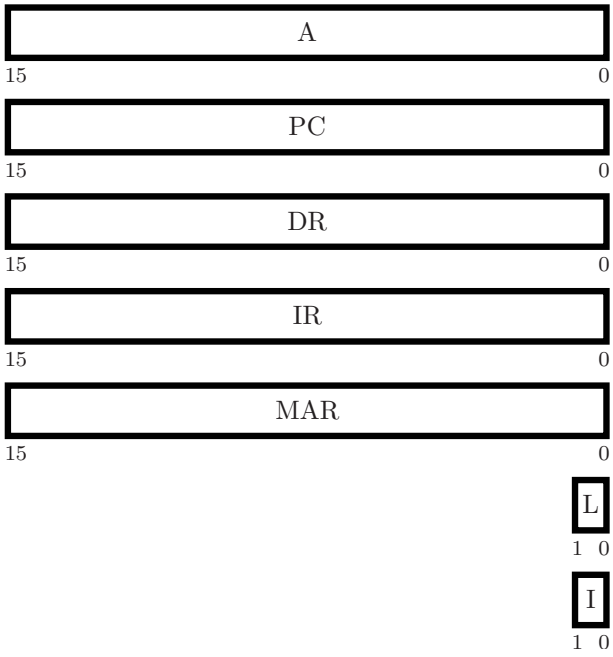


Figure 2: Full Register file. Only the A, L and I registers are made directly accessible to the user.

2.4.3 Program Counter (PC)

The Program Counter is a 16-bit register. It contains the address of the next instruction to be executed. It may not be read, but can be indirectly affected by way of skips, jumps, subroutine calls, traps, and interrupts.

2.4.4 Interrupt Register (I)

This 1-bit register controls the computer's behaviour on detecting an interrupt request. The register may be manipulated by the user to allow or mask interrupts.

2.4.5 Instruction Register (IR)

The Instruction Register (IR) holds the instruction currently being executed. It is 16 bits wide and is used internally only. There is no way to access it.

2.4.6 Memory Address Register (MAR)

The Memory Address Register (MAR) is a 16-bit register that buffers most addresses that are put out on the CFT address bus. The exception is indirect addressing, which uses the Data Register. It cannot be accessed by the user.

2.4.7 Data Register (DR)

The Data Register (DR) is a 16-bit register used internally by the CFT to buffer addresses during indirect addressing. Like the MAR, the DRcannot be accessed by the user.

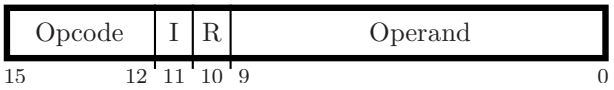


Figure 3: Instruction Format of the CFT architecture.

2.5 Instruction Format

The CFT computer uses a single instruction format. All instructions are exactly 16 bits wide and comprise the same fields, as shown in figure 3 (p. 3). Each instruction contains four fields. From most to least significant, they are:

The Instruction opcode (most significant 4 bits). This field identifies the instruction to be performed.

The Indirection mode (1 bit). Depending on the instruction, this bit selects between the literal and direct, or the direct and indirect addressing mode.

The Register Mode (1 bit). This bit controls whether addresses and literals are relative to the current page, or relative to page zero (the register page).

The Operand or address offset (least significant 10 bits). This allows only ten bits of literals or addresses to be specified in an instruction. The most significant six bits are filled in from one of two sources as follows:

- If the Register Mode bit is set (1), the six most significant bits of the PC register are used. This is page-relative addressing.
- If the Register Mode bit is clear (0), the six most significant bits of the operand are zero. This is Register addressing, also known as Page-Zero addressing, since Page Zero is used for system registers.

This introduces a number of limitations, discussed in section 2.6 (p. 3).

2.6 Page-Relative Limitations

Unless the Register Mode (R) bit is enabled (field R in the instruction equal to 0), an instruction at address addr may only access memory (and I/O) address between addr `AND FC00` and `(ADDR AND FC00) OR 3FF`. This introduces a number of limitations and defines the programming style of CFT more than any other single aspect of the design.

There are four obvious solutions to the limitation:

Subprograms need to store temporary data on the local page, or to use special 'scratch' registers in Page Zero. Both addressing modes take the same time to execute.

In the case of constants, the programmer can either limit themselves to constants in the range `0000–03FF`, store commonly-used large constants (such as -1, `FFFF` and -2, `FFFE`) in a Page Zero constant table, or a combination of these techniques[3].

Special care needs to be taken for subprograms longer than 1,024 words. If code crosses a page boundary, references to any local data will in fact refer to the same offset within the new page, and will of course be invalid.

The standard assembler issues a cross-page warning when using symbolic names (labels) for such local data, which is a very good practice.

Jumps, subroutine calls and traps suffer from the same issue. Indirect Register addressing is commonly used as a solution to this problem: the address (vector) of the subprogram in question is stored in a Page Zero location, and the jump is made using indirection. This has the added benefit of allowing the vectors to be changed so that system services may be overridden.

Perhaps the most significant limitation, however, is on the size of the I/O address space. When performing page-relative I/O operations (`IN`, `OUT` and `IOT`), the most significant six bits of the I/O address will *still* be taken from the six most significant bits of the PC. Because of this, different I/O devices will be accessed depending on the location of the program in memory. There are two ways to avoid this: the primary one is to only use 10 bits of I/O space. Either the R field in all I/O instructions must be 0, or the hardware must decode only the 10 least significant address lines[4], or a combination of both. The secondary means of avoiding this issue is indirect addressing. It is assumed that a mixture of these techniques will be used in practice. Please refer to your system documentation for exact details.

2.7 Addressing Modes

CFT instructions have two bits that modify how instructions treat the operand field: the R (register) field, and the I (indirection) field.

The R field controls how the six most significant bits of the operand are derived, as described in section 2.5 (p. 3).

The I field controls how the resultant operand is used. This depends on the instruction.

Combined, there are up to four addressing modes per instruction. There is also a fifth addressing mode, autoindex mode, which is selected for specific memory addresses only. Each instruction supports five addressing modes.

Naturally, no instructions support all addressing modes. Depending on the nature of the instruction, either literal and direct modes, or direct and indirect modes are used. There also some instructions in which the addressing mode does not apply. This is indicated as 'special' mode.

The addressing modes are as follows:

- Immediate.
- Immediate page-relative.
- Direct register.
- Direct page-relative.
- Indirect register.
- Indirect page-relative.
- Indirect autoindex.
- Special.

2.7.1 Immediate Mode

When using applicable instructions with R=0 and I=0, the value field in the instruction is used as the least significant 10 bits of a literal value. The most significant six bits are zero. This allows literal values in the range `0000 03FF` to be specified.

For example, executing the Load Immediate (LI R) instruction `LI R 0350` will set A=`0350`.

2.7.2 Immediate Page-Relative Mode

When using applicable instructions with with R=1 and I=0, the value field in the instruction is used as the least significant 10 bits of a literal value. The most significant six bits of the PC are used as the most significant six bits of that value. This allows addresses in the current page to be specified.

For example, executing the Load Address (LIA) instruction `LIA 0042` at address `2130` will set A=`2042`.

[3] A note for PDP-8 programmers: the CFT has a `LI` (Load Immediate) instruction that can load the Accumulator with a 10-bit literal. This is considerably faster the the only PDP-8 alternative, constructing basic constants using microcoded instructions.

[4] Thus providing six don't care bits, which is 64 copies of every I/O device, one for every page.

2.7.3 Direct Register Mode

When using applicable instructions with R=0 and I=0, the value field in the instruction is used as the least significant 10 bits of a memory or I/O address. The most significant six bits are zero. This allows Page Zero registers and I/O devices to be addressed.

For example, executing the Load instruction `LOAD R 0010` instruction will load the contents of address `0010` into A.

2.7.4 Direct Page-Relative Mode

When using applicable instructions with R=1 and I=0, the value field in the instruction is used to construct an address which is used directly to reference a value in memory or I/O space. This is the most commonly used addressing mode.

For example, executing the Store instruction `STORE 0010` at address `8514` will write the contents of A onto memory address `8410`.

2.7.5 Indirect Register Mode

When using applicable instructions with R=0 and I=1, the value field in the instruction is used to construct an address. The 16-bit value *at that address in memory* is used to reference memory or I/O space.

For example, executing the LOAD instruction `LOAD I 009A` will read into the DRregister a 16-bit value from memory location `009A`. A will then be loaded with the value of the address in memory stored in DR.

2.7.6 Indirect Mode

When using applicable instructions with R=1 and I=1, the value field in the instruction is used to construct an address. The 16-bit value at that address in memory is used to reference memory or I/O space.

For example, executing the LOAD instruction `LOAD I 009A` at address `1234` will read into the Data Register a 16-bit value from memory location `109A`. That number is treated as the address in memory to be read into A.

2.7.7 Autoindex Mode

Autoindex mode operates like indirect mode above, but the memory location indicated in the instruction field is incremented after being used. Autoindex mode cannot be selected directly. It is automatically enabled for addresses in the range `0080–00FF`. The value of the R field is immaterial, only the address is examined.

For example, executing the LOAD instruction `LOAD I R 009A` will read into the DRregister a 16-bit value from memory location `109A`. That number is used as the address in memory to be read into A, then incremented by one and written back to location `009A`.

If the instruction were `LOAD I 009A` and it executed at address `0003`, the result would have been the same.

2.8 Operating States

The CFT CPU has four major operating states: initialisation, interrupt, operation and waiting.

2.8.1 Initialisation

When the computer initialises via a system reset or initial boot, it sets PC to the boot address `FFF0` and clears the I register to disallow interrupts. All other registers are undefined and may contain arbitrary values — boot code should initialise any register necessary for its operation. Normal operation is then resumed.

2.8.2 Interrupt

When the computer encounters an interrupt, and if the I register is 1 (allowing interrupts), the event is buffered in a minor state register. When the currently executing instruction is completed, the I register is cleared to disable further interrupts and the current value of the PC is written to memory address `0002`. Finally, the PC is loaded with the value `FFF8`, which is the address of the Interrupt Service Routine (ISR). Normal operation is then resumed.

2.8.3 Operation

During normal fetch-execute operation, the computer's microcode fetches instructions from main memory address indicated by PC, increments the PC, then executes the fetched instructions. This is the mode the computer spends most of its time in.

2.8.4 Wait

On occasion, external slow hardware or the operator's console may halt the processor's clocks. In this state, the sequencer does not work through the microcode and the computer does not operate.

This mode may be entered at any time. The operator's console allows it to be entered either inside a microprogram, or immediately before the fetch part.

The Wait state is transient: as soon as the appropriate signals are de-asserted by the hardware, the previous state resumes.

3 Instruction Set Reference

The first edition of the CFT microcode allowed for fourteen different instructions. The second edition added the IOT instruction, which makes for fifteen mutually orthogonal instructions:

- TRAP: save the PC and jump to the specified location. Used for operating system services.
- IOT: new in version 2.0 of the microcode. Writes to a specified output device, then reads a result back from it. Used to implement computer extensions via I/O-addressable extension units.
- LOAD: load from memory.
- STORE: write to memory.
- IN: read from an input device.
- OUT: write to an output device.
- JMP: unconditional jump to the specified location.
- JSR: save the PC and jump unconditionally to the specified location.
- ADD: load from memory and add to the accumulator.
- AND: load from memory and perform a bitwise AND with the value in the accumulator.
- OR: load from memory and perform a bitwise OR with the value in the accumulator.
- XOR: load from memory and perform a bitwise exclusive OR with the value in the accumulator.
- OP1: minor operations, group one.
- OP2: minor operations, group two.
- LIA: load accumulator with literal value or address.

To simplify understanding of the instruction set, opcode mnemonics are used rather than actual machine code. In many cases, the notation used is that of the Standard CFT Assembly language which merits some description. This is not a full definition of CFT Assembly language, merely enough of it to facilitate discussing the instruction set in a more human-readable form.

```
LOAD 0000     ; Load direct
LOAD I 0342   ; Load indirect
0342 I LOAD   ; Valid, but against conventions.
LOAD I R 007F ; Load zero page and indirect
LOAD I R 0080 ; Load, autoindexing
IN R PANEL 0  ; Read panel switches
CLL RBL       ; Shift left one bit
HALT          ; Halt the system
```

Figure 4: Examples of CFT instructions in Assembly language .

CFT Assemblers parse lexical tokens denoting either symbolic instruction names or hexadecimal numbers. Symbols are converted to numbers using symbol tables. The resultant numbers, which must be 16 bits in width, are ORred together to form an instruction[5].

This provides great simplicity and generality at the expense of making the code slightly less readable from a modern Assembly perspective[6]).

Anything after the first semicolon (;) on a line is considered a comment and ignored.

A brief example of CFT Assembly language can be found in figure 4 (p. 6).

3.1 Memory

The following instructions operate on memory space.

3.1.1 LOAD — Load Accumulator

Load a word from memory into A. Direct, indirect and autoindex modes are available.

3.1.2 STORE — Store Accumulator

Write A to the specified memory location. Direct, indirect and autoindex modes are available.

3.2 I/O

The following instructions operate on I/O space.

[5] One exception is the R symbol, which *clears* bit 10 of the instruction to turn on Page Zero/register addressing modes. In practice, this is not actually an exception. After assembling an instruction, however, the Assembler performs an XOR with the hexadecimal value 0400 (complementing bit 10), which allows the bitwise Or semantics to work for everything, including the R flag, and the resultant code to run as expected.

[6] PDP-8 Assembly programmers will feel right at home, though.

3.2.1 IN — Read from I/O Device

Read a word from I/O space into A. Direct, Indirect and autoindex modes are available. IN instructions may have additional side-effects that depend on the peripheral being addressed.

3.2.2 OUT — Write to I/O Device

Write the contents of A to the specified I/O space address. Direct, Indirect and autoindex modes are available. OUT instructions almost always have side-effects that depend on the peripheral being addressed.

3.2.3 IOT — I/O Transaction

Implements a processor extension. The IOT instruction writes A to I/O space, the reads A back from the same address. The I/O-mapped processor extension can introduce wait states at the end of the I/O space read cycle, in order to get more time to perform its task. Direct, Indirect and autoindex modes are available.

3.3 Arithmetic and Logic

The following instructions perform common arithmetic and logic operations.

3.3.1 ADD — Add to Accumulator

Read a word from memory and add it to A. If the addition results in a carry-out, L is toggled. Direct, Indirect and autoindex modes are available.

3.3.2 AND — Bitwise And with Accumulator

Read a word from memory and perform a bitwise AND operation with A. Direct, Indirect and autoindex modes are available.

3.3.3 OR — Bitwise Or with Accumulator

Read a word from memory and perform a bitwise OR operation with A. Direct, Indirect and autoindex modes are available.

3.3.4 XOR — Bitwise Exclusive OR with Accumulator

Read a word from memory and perform a bitwise XOR operation with A. Direct, Indirect and autoindex modes are available.

3.4 Flow Control

The following instructions implement flow control via modification of the PC register.

3.4.1 TRAP — Jump to System Service

Writes the value of the PCto memory location 0001, then sets the PCto the address specified in the instruction. Direct, Indirect and autoindex modes are available.

3.4.2 JMP — Jump to Address

Sets the PCto the address specified in the instruction. Direct, Indirect and autoindex modes are available.

3.4.3 JSR — Jump to Subroutine

Writes the value of the PCto memory location 0000, then sets the PCto the address specified in the instruction. Direct, Indirect and autoindex modes are available.

3.5 Specials

These are operations that do not fit in the categories above.

3.5.1 OP1 — Operations 1

The OP1 instruction provides a number of minor operations, any number of which may be performed in a preset order. This is very similar to the PDP-8 'microcoded' instructions.

The operand of the OP1 instruction is seen as a bitfield, where set bits trigger a particular minor operation. The table below outlines the possible operations that can be ORred together.

Bitfield	OP1 Instruction
1---------	CLA — Clear A
-1--------	CLL — Clear L
--1-------	NOT — Complement A
---1------	INC — Increment (L,A) by one
----1-----	CPL — Complement L
-------010	RBL — Roll Bit Left (L,A)
-------001	RBR — Roll Bit Right (L,A)
-------110	RNL — Roll Nybble Left (L,A)
-------101	RNR — Roll Nybble Right (L,A)

These operations is performed in the order in which they appear above, from top to bottom. If no operations are specified, i.e. the instruction is C000, the instruction

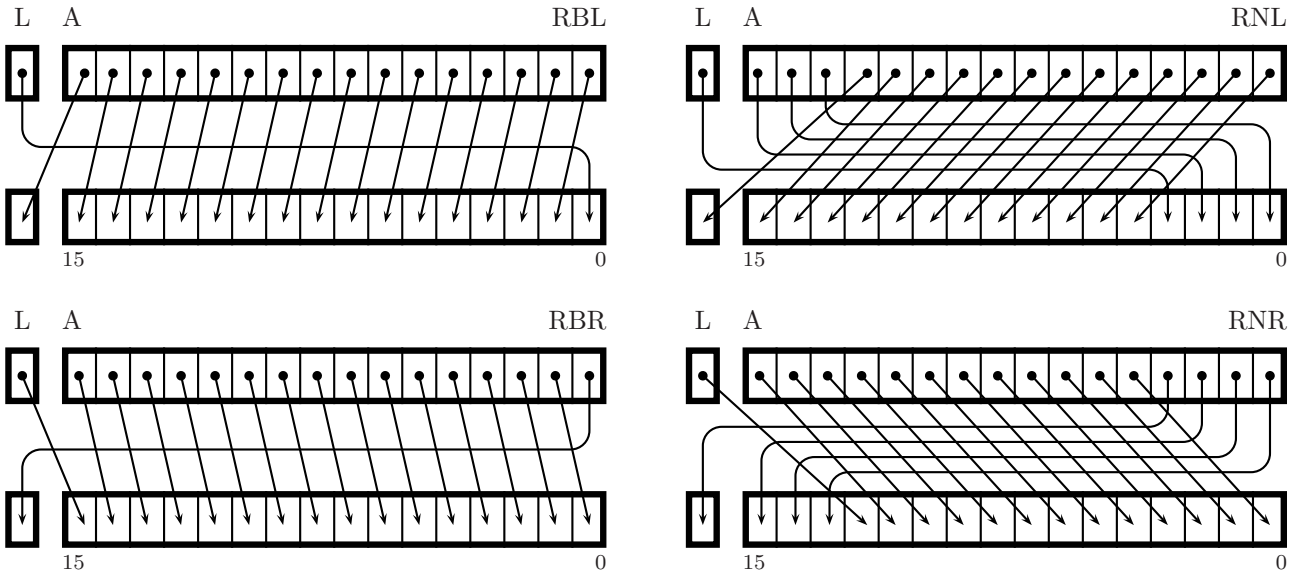


Figure 5: Roll Instructions.

is effectively a NOP. The four roll instructions are mutually exclusive. The notation (L,A) indicates that both L and A are used together as a 17-bit quantity, with L becoming the most significant bit.

To clear both A and L, the instruction would be `OP1 CLA CLL`.

If the `INC` instruction is issued when A=`FFFF`, L is toggled. As expected, A becomes `0000`.

To calculate the two's complement of A `OP1 NOT INC` (hexadecimal instruction `C300`) will first invert A, then increment it by one. Note that `OP1 INC NOT` has exactly the same result, as the order in which minor operations are performed is fixed.

A binary left shift by one bit can be defined as `OP1 CLL RBL` (`C102`).

The standard Assembler defines many of these combinations of instructions as convenient macros. When using these macros, specifying `OP1` is optional.

The exact operation of the four roll instructions is illustrated in figure 5.

3.5.2 `OP2` — Operations 2

This instruction is very similar to the `OP1` instruction. The main feature of the `OP2` instruction is skipping. Skip instruction skip over the following instruction if the corresponding condition is true. Conditions involve the A and L registers. The following table lists the `OP2` bitfield values that may be ORred together.

Bitfield	`OP2` Instruction
`------01--`	`SNA` — Skip if A negative (G1)
`------0-1-`	`SZA` — Skip if A zero (G1)
`------0--1`	`SSL` — Skip if L set (G1)
`------1000`	`SKIP` — Always skip (G2)
`------11--`	`SNN` — Skip if A non-negative (G2)
`------1-1-`	`SNZ` — Skip if A non-zero (G2)
`------1--1`	`SCL` — Skip if L clear (G2)
`---1------`	`CLA2` — Clear A
`----1-----`	`CLI` — Clear I flag
`-----1----`	`STI` — Set I flag

There are two groups of branching instructions: G1 (bit 3 of the instruction operand is `0`) and G2 (bit 3 of the instruction operand is `1`). When G1 instructions `SNA`, `SZA` and `SSL` are specified together, the skip is performed when any of the specified conditions hold (logical disjunction or Or). For example, the instruction `SZA SNA` is 'skip if A less than or equal to zero', or 'skip if A non-positive' (which is the standard CFT Assembly macro `SNP`).

When G2 instructions are specified together, the skip is performed when all of the specified conditions hold (logical conjunction or And). For example, `SNN SNZ` is 'skip if A is non-zero and non-negative', or 'skip if A is positive' (the standard Assembler macro for this is `SPA`). The full set of combinations to check the value of A is as follows:

Instruction	Macro	Semantics
SNA		Skip if A < 0
SNA SZA	SNP	Skip if A ≤ 0
SZA		Skip if A $= 0$
SNZ		Skip if A $\neq 0$
SNN		Skip if A ≥ 0
SNN SNZ	SPA	Skip if A > 0

3.5.3 LIA — Load Immediate Address

Loads A with the literal value specified in the instruction. The page-relative form of this instruction is used to load A with a page-relative address.

When R=0, LIA becomes the Load Immediate (LI) instruction, which can load a value in the range 0000–03FF into A.

This instruction always operates in the Immediate addressing mode. The value of the instruction I bit is *ignored*.

3.6 Standard Macros

3.6.1 RET — Return from Subroutine

This is defined as JMP I R 0000. It jumps to the return address saved by the JSR instruction, which stores it at memory address 0000.

3.6.2 RTT — Return from Trap

This is defined as JMP I R 0001. It jumps to the return address saved by the TRAP instruction, which stores it at memory address 0001.

3.6.3 NEG — Negate A

Obtains the two's complement of A by performing OP1 NOT INC.

3.6.4 ING — Increment and Negate A

Obtains the two's complement of A and increases it, thereby calculating A $\leftarrow -(A + 1)$. Due to the use of twos complement work, this is equivalent to a simple OP1 NOT.

3.6.5 LI — Load Immediate

This is equivalent to LIA R. It loads A with the 10-bit literal value specified as operand. The most significant six bits are zero.

3.6.6 SPA — Skip if Positive A

This is equivalent to OP2 SNN SNZ. It skips the next instruction if A is neither negative, nor zero (thus positive).

3.6.7 SNP — Skip if Non-Positive A

This is equivalent to OP2 SNA SZA. It skips the next instruction if A is less than or equal to zero (non-positive).

3.6.8 SBL, SBR, SNL, SNR — Bitwise shifts

The four bitwise shift macros are formed by combining OP1 CLL with the corresponding OP1 roll operations. They provide bitwise (unsigned arithmetic) shift operations which are fundamental in implementing multiplication and division, among others.

3.7 Common Tasks

This section shows how some common, simple Assembly language tasks can be performed using the CFT instruction set.

3.7.1 Addition With Carry

The L flag may be used as carry in using this short program:

```
adc:    SCL         ; Skip if L=0
        INC
        ADD addr
```

The program increases the Accumulator by one if the L register is set, then performs addition as normal.

3.7.2 Subtraction

There is no explicit subtraction instruction, but the benefit of two's complement is that one is unnecessary. Subtraction can be reduced to addition as follows:

```
sub:    NEG         ; OP1 NOT INC
        ADD addr
```

The program does a one's complement (binary negation) of the accumulator, then increments it by one, which is a two's complement (decimal negation). This is the NEG macro. The addition is then performed with the negative value of the Accumulator, to obtain the value addr - A.

3.7.3 Bitwise Shifts

To convert bitwise rolls to bitwise shifts, the L register needs to be cleared before the roll takes place. The following code performs a bitwise (or unsigned arithmetic) shift one bit to the right.

```
shr:    CLL RBR
```

This operation is mathematically tantamount to $\lfloor A/2 \rfloor$. Obviously, other roll instructions may be substituted for SBR.

3.7.4 Arithmetic Shifts

These are slightly more involved.

```
asr:    CLL       ; Clear L (L=0)
        SNN       ; Skip if A >= 0
        CPL       ;   A < 0: toggle L (L=1)
        RBR       ;   Roll 1 bit right.
```

At the end of this short program, the most significant (i.e. sign) bit of A will be the same as before. Due to the use of two's complement, this is still equivalent to $\lfloor A/2 \rfloor$, but the behaviour is now specialised to signed numbers.

3.7.5 Bitwise Or of a Small Array

The autoindex registers can be used to simplify short loops. Here, the A is loaded with the address of the first word of the array. The first 5 elements of it will be ORred together, and the result left in A.

```
or6:    STORE R 80  ; Autoindex register
        LOAD I R 80 ; Load 1st value
        OR I R 80   ; OR with 2nd value
        OR I R 80   ; OR with 3rd value
        OR I R 80   ; OR with 4rd value
        OR I R 80   ; Or with 5th value
```

3.7.6 Simple Loops

This is a simple loop. It iterates as many times as the value of A on entry to the subroutine. In this example, the loop body sends the hexadecimal value 2A to an I/O device designated TTY0 0. The routine is expected to be called with JSR stars.

```
stars:  NEG           ; A = -A
        STORE R 10    ; Loop variable
loop:   SNZ           ; A = 0?
        RET           ;   Yes. Return.
        LIA 21        ; A = 0021 (ASCII '!')
        OUT TTY0 0    ; Send it out.
        LOAD R 10     ; Load the loop counter.
        INC           ; Step it.
        STORE R 10    ; And store it back.
        JMP loop      ; Loop again.
```

If the operating system has set up a Page Zero store of commonly used constants, and it includes minus1 (which contains -1 (FFFF), this loop may be made shorter:

```
stars:  STORE R 10    ; Loop variable
loop:   SNZ           ; A = 0?
        RET           ;   Yes. Return.
        ADD minus1    ; Decrement loop counter
        STORE R 10    ; Store it back.
        LIA 21        ; A = 0021 (ASCII '!')
        OUT TTY0 0    ; Send it out
        LOAD R 10     ; Load the loop counter.
        JMP loop      ; Loop again.
```

3.7.7 Sum of a Block of Words

A somewhat more complex example leverages the Autoindex feature to calculate the sum (modulo 65,536) of a block of words. A pointer to the block of words should be stored at location 0010, and the size (in words) should be in A. On exit, memory address 0012 will contain the sum of the words.

```
sum_n:  NEG
        STORE R 11    ; Loop variable
        LOAD R 10     ; Array base
        STORE I R 80  ; Autoindex
        LI 0          ;
        STORE I R 12  ; Sum = 0
        LOAD R 11     ; Number of words left
loop:   SNZ           ; Is it zero?
        RET           ;   Yes. Return.
        LOAD R 12     ; Load running total
        ADD I R 80    ; Sum a word
        STORE R 12    ; Store it back
        LOAD R 11     ; Loop variable
        INC           ; Increment
        STORE R 11    ; Store it back
        JMP loop      ; Loop again.
```

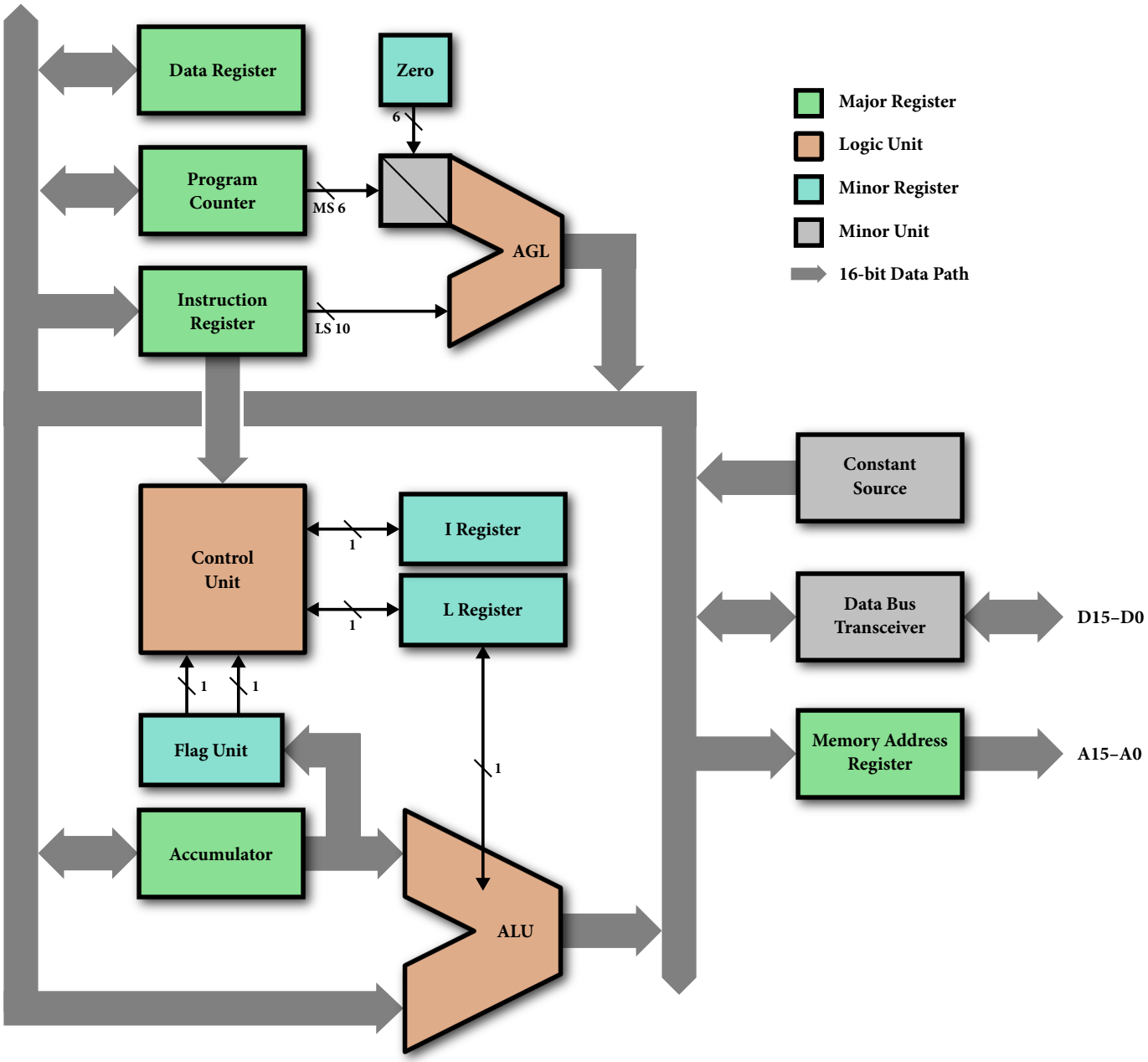
4 Further reading

For further information on the CFT computer, please consult the following URL:

www.bedroomlan.org/hardware/cft

Bit pattern (base hex)	Mnemonic	Cycles[1]	Notes
0000 1 R aaaaaaaaaa (0000)	TRAP a	7/10/14	Trap. **mem** [0001] ← PC; PC ← a (section 3.4.1 (p. 7))
0001 I R aaaaaaaaaa (1000)	IOT a	8/11/15	I/O Transaction. **io** [a] ← A; A ← **io** [a] (section 3.2.3 (p. 7))
0010 I R aaaaaaaaaa (2000)	LOAD a	7/10/13	Load. A ← **mem** [a] (section 3.1.1 (p. 6))
0011 I R aaaaaaaaaa (3000)	STORE a	7/10/13	Store. **mem** [a] ← A (section 3.1.2 (p. 6))
0100 I R aaaaaaaaaa (4000)	IN a	6/9/12	Input. A ← **io** [a] (section 3.2.1 (p. 7))
0101 I R aaaaaaaaaa (5000)	OUT a	6/9/13	Output. **io** [a] ← A (section 3.2.2 (p. 7))
0110 I R aaaaaaaaaa (6000)	JMP a	4/7/11	Jump. PC ← a (section 3.4.2 (p. 7))
0110 1 0 0000000000 (6400)	RET	11	Return from subroutine. **Macro:** RET I R 0[5]
0110 1 0 0000000001 (6401)	RTT	11	Return from trap. **Macro:** RET I R 1[5]
0111 I R aaaaaaaaaa (7000)	JSR a	8/11/14	Jump to subroutine. **mem** [0000] ← PC; PC ← a (section 3.4.3 (p. 7))
1000 1 R aaaaaaaaaa (8000)	ADD a	7/10/14	Add. A ← A + **mem** [a] (section 3.3.1 (p. 7))
1001 I R aaaaaaaaaa (9000)	AND a	7/10/14	Bitwise And. A ← A AND **mem** [a] (section 3.3.2 (p. 7))
1010 I R aaaaaaaaaa (A000)	OR a	7/10/14	Bitwise Or. A ← A OR **mem** [a] (section 3.3.3 (p. 7))
1011 I R aaaaaaaaaa (B000)	XOR a	7/10/14	Bitwise exclusive Or. A ← A XOR **mem** [a] (section 3.3.4 (p. 7))
1100 - - 0000000000 (C000)	OP1 NOP	11[2]	No operation. OP1 discussion: section 3.5.1 (p. 7)
1100 - - 1--------- (C200)	OP1 CLA	11[2]	Clear A. A ← 0
1100 - - -1-------- (C100)	OP1 CLL	11[2]	Clear L. L ← 0
1100 - - --1------- (C080)	OP1 NOT	11[2]	Complement A. A ← NOT A
1100 - - ---1------ (C040)	OP1 INC	11[2]	Increment A. (L, A) ← (L, A) + 1
1100 - - ----1----- (C020)	OP1 CPL	11[2]	Complement L. L ← NOT L
1100 - - -------010 (C002)	OP1 RBL	11[2]	Roll (L,A) 1 bit left.
1100 - - -------001 (C001)	OP1 RBR	11[2]	Roll (L,A) 1 bit right.
1100 - - -------110 (C006)	OP1 RNL	11[2]	Roll (L,A) 4 bits left.
1100 - - -------101 (C005)	OP1 RNR	11[2]	Roll (L,A) 4 bits right.
1100 0 0 0011000000 (C0C0)	NEG	11	A ← −A. Two's complement of A. **Macro:** OP1 NOT INC[5]
1100 0 0 0010000000 (C080)	ING	11	A ← −(A + 1). One's complement of A. **Macro:** OP1 NOT[5]
1100 0 0 0100000010 (C102)	SBL	11[2]	Bitwise shift (L,A) 1 bit left. **Macro:** OP1 CLL RBL[5]
1100 0 0 0100000001 (C101)	SBR	11[2]	Bitwise shift (L,A) 1 bit right. **Macro:** OP1 CLL RBR[5]
1100 0 0 0100000110 (C106)	SNL	11[2]	Bitwise shift (L,A) 4 bits left. **Macro:** OP1 CLL RNL[5]
1100 0 0 0100000101 (C105)	SNR	11[2]	Bitwise shift (L,A) 4 bits right. **Macro:** OP1 CLL RNR[5]
1101 - - 0000000000 (D000)	OP2 NOP2	12[3]	No operation. OP2 discussion: section 3.5.2 (p. 8)
1101 - - ------01-- (D004)	OP2 SNA	12[3]	G1 skip: A < 0 ⟹ PC ← PC + 1 [4]
1101 - - ------0-1- (D002)	OP2 SZA	12[3]	G1 skip: A = 0 ⟹ PC ← PC + 1 [4]
1101 - - ------0--1 (D001)	OP2 SSL	12[3]	G1 skip: L = 1 ⟹ PC ← PC + 1 [4]
1101 - - ------1000 (D008)	OP2 SKIP	12[3]	G2 skip: PC ← PC + 1 [4]
1101 - - ------11-- (D00C)	OP2 SNN	12[3]	G2 skip: A ≥ 0 ⟹ PC ← PC + 1 [4]
1101 - - ------1-1- (D00A)	OP2 SNZ	12[3]	G2 skip: A ≠ 0 ⟹ PC ← PC + 1 [4]
1101 - - ------1--1 (D009)	OP2 SCL	12[3]	G2 skip: L = 0 ⟹ PC ← PC + 1 [4]
1101 - - ---1------ (D040)	OP2 CLA2	12[3]	A ← 0
1101 - - ----1----- (D020)	OP2 CLI	12[3]	Clear (disallow) interrupt flag. I ← 0
1101 - - -----1---- (D010)	OP2 STI	12[3]	Set (allow) interrupt flag. I ← 1
1101 0 0 0000000110 (D006)	SNP	12[3]	G1 Skip if A non-positive. **Macro:** OP2 SNA SZA[5]
1101 0 0 0000001110 (D00E)	SPA	12[3]	G2 Skip if A positive. **Macro:** OP2 SNN SNZ[5]
1111 - R aaaaaaaaaa (F000)	LIA a	4	Load immediate address: A ← a (section 3.5.3 (p. 9))
1111 - 0 aaaaaaaaaa (F000)	LI a	4	Load immediate. **Macro:** LI R[5]

Table 1: CFT Instruction Set. I is the indirection bit. R is the register mode bit. Dashes in the instruction bit pattern indicate 'don't care' values. Notes: [1] Processor Cycles shown for Direct/Indirect/Autoindex modes, where available. [2] Any number of these sub-instructions may be combined and will always need exactly 11 cycles in total. [3] Any number of these sub-instructions may be combined for a total of exactly 12 cycles. [4] G1 and G2 skips are groupwise mutually exclusive. When G1 skips are combined, their conditions are disjuncted (ORred). When G2 skips are combined, their conditions are conjuncted (ANDed). [5] This is a convenience macro, not an independent instruction.


Data Register
Zero
6
Program Counter
MS 6
AGL
Instruction Register
LS 10
Major Register
Logic Unit
Minor Register
Minor Unit
16-bit Data Path
Constant Source
Control Unit
1
I Register
1
L Register
Data Bus Transceiver
D15–D0
1
1
Flag Unit
1
Memory Address Register
A15–A0
Accumulator
ALU