CFT Minicomputer Assembler Guide

Alexios Chouchoulas
alexios@bedroomlan.org

Abstract

This document discusses the design and operation of the CFT Standard Assembler, introduced as the first and main implementation tool for writing CFT software. 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. The instruction set is extensible in the style of the PDP-8 using instruction-level aliasing (instruction macros) and assembler macros.
The Standard Assembler is capable of producing optimised code for the CFT that takes advantage of the architecture's idiosyncrasies, and avoids the pitfalls of its many peculiarities.

History of Changes

2012-05-16 First draft outline.

1 Introduction

This document discusses the design and operation of the CFT Standard Assembler, introduced as the first and main implementation tool for writing CFT software. 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. The instruction set is extensible in the style of the PDP-8 using instruction-level aliasing (instruction macros) and assembler macros.

The Standard Assembler is capable of producing optimised code for the CFT that takes advantage of the architecture's idiosyncrasies, and avoids the pitfalls of its many peculiarities.

The Standard Assembler is a cross-compilation tool, written in Python to run on Unix and Unix-like computers. A possibly simplified version of the Standard Assembler is intended to be included as part of the CFT Forth Operating System, and possibly as a more full-featured stand-alone, native version.

2 Architecture

The Standard Assembler is a text-book multiple pass assembler with a simple macro facility. It includes a 'permanent' symbol table with macro definitions for the standard CFT instruction set as outlined in the Programming Guide, in section **??** (p. **??**).

2.1 The Cross Assembler

The Cross Assembler is implemented in Python 2.6, but will probably run on Python 2.4 or newer.

3 Syntax

Assembler syntax is simpler than the average assembler, to match the simpler structure of the CFT.

Lines consisting of white space are ignored. Every other line must contain one or more of the following:

- Comments.
- Instructions.
- Origin specifications.
- Label declarations.
- Assembler directives.
- Macros.

3.1 Comments

Comments are introduced with the slash (`/`) or semicolon (`;`) characters. Conventional comments follow a C++-style (`//`) or Assembly (`;;`) conventions, but this is not mandatory.

Any text after the first comment character is ignored, as expected.

3.2 Instructions

An instruction is one 16-bit value.

Each 16-bit instruction must be composed of one or more 'fields', each separated by one or more spaces. All the values specified on a line get ORred together to form an aggregate which is then assembled at the current address.

Each field must be one of the following things:

- Symbols.
- Literal values.
- Expressions involving symbols and/or literal values.

3.2.1 Symbols

A symbol is a sequence of one or more symbol characters. The first character must be an alphabetic character or an underscore (`_`). Subsequent characters may be alphanumeric or underscores.

Any symbol in the symbol table may be used as an instruction field, no matter what its semantics are. Examples include opcodes such as `LOAD`, simple field values such as `R`, labels, or other symbols.

Labels are treated specially if included in an instruction: the label's value (its address), is masked by ANDing with value $\texttt{3FF}_{16}$. In this format, a label will fill the instruction's 10-bit operand field.

3.2.2 Literal Values

CFT Assembly supports 16-bit literal values in three formats:

Decimal A plain numerical value consisting of one to five digits between `0` and `9`.

Hexadecimal The character `&` or sequence `0x` followed by 1–4 digits, each between `0`–`9` or `A`–`F`. Lower case letters may also be used.

Binary The character `#` followed by 1–16 binary digits (`0` or `1`). To aid in defining readable bitfields, single quotes `'` and commas `,` are ignored. Additionally, dashes (`-`), periods (`.`) and underscores (`_`) are treated as `0`. This allows bitfields to be defined as `#----'0010`, clearly indicating 'don't-care' bits.

3.2.3 Expressions

Expressions perform simple arithmetic at assembly time. They come in three basic formats:

Current address: `@`. A single at-sign (`@`) is replaced by the current address.

Relative address: `@+value` or `@-value`. The value (which may be any parseable field value) is added or subtracted from the current address, and the result replaces the expression.

Full expression: `@a op b`. The operator **op** is one of `+`, `-`, `*` (multiplication), `/` (division), `|` (bitwise OR), `&` (bitwise AND), or ` ` (bitwise XOR). There should *no space between the operator and the two operands*. For example, `@someLabel+&80` is valid, while `@someLabel + &80` is invalid. The two operands **a** and **b** are any parseable fields.

3.3 Setting the Origin

The origin is the next address to be assembled at. To set the origin, include a literal value followed by a colon (`:`). For example, this sets the origin to the CFT Reset vector at address `FFF0`:

```
&FFF0:       ;; The CFT Reset Vector address
```

3.4 Labels

Labels name a new symbol after the current address, so that the address can then be referred to by name. Labels are essential in Assembly languages. They simplify flow

control and many other advanced tasks. A label is declared by typing its name followed by a colon character (`:`). Multiple labels may be set for the same address, one to a line. The origin may also be set. This is a common idiom:

```
&FFF0:       ;; The CFT Reset Vector address
reset_vec:
reboot:
```

Although declaring multiple labels for the same address is wasteful, it can often lead to better readability.

3.5 Directives

A sequence of letters starting with a period (`.`) is a directive. Directives assemble special values and modify the behaviour of the assembler. They are similar in spirit to C preprocessor directives.

3.5.1 Defining Arbitrary Symbols with `.equ`

The `.equ` directive defines a new symbol with an arbitrary value. The syntax is:

```
.equ symbol value
```

Any valid symbol may be named, and any parseable value may be used: instruction, label, or literal. The most common uses of `.equ` are in defining instructions or constants. The following definitions are actually part of the permanent symbol table of the CFT Standard Assembler and cover the basic instruction set:

```
.equ TRAP  0x0000
.equ IOT   0x1000
.equ LOAD  0x2000
.equ STORE 0x3000
.equ IN    0x4000
.equ OUT   0x5000
.equ JMP   0x6000
.equ JSR   0x7000
.equ ADD   0x8000
.equ AND   0x9000
.equ OR    0xA000
.equ XOR   0xB000
.equ OP1   0xC000
.equ OP2   0xD000
.equ ISZ   0xE000
.equ LIA   0xF000

.equ R     0x0400
.equ I     0x0800
```

After definition, symbols may be used wherever parseable values may be used, including in subsequent `.equ` definitions:

```
.equ LIA   &F000
.equ LI    LIA  R   ; &F000 | &0400
```

This is exactly how the assembler defines the standard instruction macros, of which `LI` is one.

3.5.2 Declaring Registers with `.reg`

The `.reg` directive works like the `.equ` directive, with one exception: it declares a location in memory (usually Page 0) to be used as a register. For example:

```
.reg  R0   R &10
.reg  R1   R &11
.reg  R2   R &12
.reg  R3   R &13

.reg  I0   R &81 ; Autoincrement registers
.reg  I1   R &82
.reg  I2   R &83
.reg  I3   R &84
```

The difference between `.equ` and `.reg` is mainly one of semantics: `.equ` can define such diverse entities as I/O locations, addresses, labels, constants, flags, instructions, bit masks, et cetera — this sometimes makes it difficult to divine the purpose of a symbol, even if good coding practices are used. In contrast, `.reg` is meant only for register declaration, and this is reflected in the various output files (map, symbol tables and graphical maps).

3.5.3 Including Assembly Files with `.include`

The `.include` directive instructs the Assembler to read, process and assemble an external file. It used like this:

```
.include "some_file.asm"
```

This may be used to break large programs into manageable, semantically distinct blocks, or to allow libraries of definitions or code to be defined and shared among programs.

3.5.4 Assembling Arbitrary Data with `.word`

The `.word` directive assembles an arbitrary word at the current address. Some examples of its syntax:

```
.word some_label  ; label address
.word @           ; assemble current address
.word @+10        ; current address + 10
.word STORE       ; assemble an instruction
.word 1 #10 &4 8  ; assemble decimal value 15
```

Any parseable value may be assembled using the `.word` directive, including instructions.

Unlike plain instruction assembly, `.word` does not treat labels specially: when assembling the value of a label in an instruction, only the lower 10 bits are used. With `.word`, all 16 bits are used. This makes `.word` invaluable in generating vector tables and aiding in indirection.

3.5.5 Filling Regions with a Value with `.fill`

The `.fill` directive assembles a value and uses it to fill a block in the object code starting with the current address. It syntax is:

```
.fill COUNT VALUE
```

The `COUNT` must be a literal value. The `VALUE` may be any parseable value, including symbols and instructions. For example, to assemble 50 copies of the value `0000`:

```
.fill 50 &0000
```

To fill a region with an instruction:

```
.fill &10 HALT
```

Please note that the value is calculated *once*, so relative addressing and expressions will not have the expected effect. The following directive will assemble 10 copies of the instruction `JMP 0`, *not* a chain of jumps with increasing addresses, despite the use of `@`.

```
&1000:  .fill 10 JMP @
```

3.5.6 Declaring Register Arrays with `.regfill`

Sometimes blocks of registers need to be declared — for example, to account for a memory-mapped device with arrays of mapped values (such as a framebuffer). The `.regfill` directive is used for that. It works exactly like `.fill`, but also marks the filled area as a register array.

3.5.7 Unpacked Strings with `.str` or `.data`

An unpacked string is a block of memory which represents character text. Each character occupies an entire word and has a 16-bit range. Unpacked strings can contain Unicode characters in UCS-2 or UTF-16 encodings. They can, of course, contain ASCII characters as well. Since the CFT is a word-addressed machine, these are the easiest strings for the computer to handle.

The `.str` directive assembles a string of values starting at the current address:

```
.str "Hello World!" 10 0
```

Character data are enclosed in double quotes, but `.str` can handle multiple fields including any parseable value from literals to instructions (of course, the utility of assembling instructions with `.str` is very limited). This may be used to embed non-printable values (and double quotes) in strings. In the example above, there is an ASCII 10 (line feed, return or new line) after the exclamation point, followed by an ASCII 0 (NUL), which is conventionally used to terminate variable-length strings.

3.5.8 Negative-Terminated Strings with `.strn`

Negative-terminated strings are unpacked strings in which end of string is indicated by setting bit 15 on the last character. This allows characters with codepoints between 0–32,767 to be used. If the upper 32,767 codepoints are not necessary, this representation saves one word per string. The `.strn` directive works in exactly the same way as the `.str` directive:

```
.strn "Hello World!" 10
```

However, the last character of the string will be ORred with the value `8000`, to produce this sequence of values in memory:

```
0048 0065 006c 006c 006f 0020 0057 006f
0072 006c 0064 0021 800a
```

Note that the terminating null conventionally added to the end of `.str` directives is unnecessary here.

3.5.9 Packed Strings with `.strp`

Packed strings are the most compact means of encoding strings of 8-bit characters (or Unicode characters encoded in UTF-8) on the CFT architecture. Two characters are assembled per word, with the first character in the lower eight bits. This byte order mirrors the way byte-addressable machines work when dealing with strings, which makes CFT binary files easier to examine on other computers. The syntax of the `.strp` is identical to that of `.str`:

```
.strp "Hello World!" 10 0
```

This will generate the following sequence of words:

```
6548 6c6c 206f 6f57 6c72 2164 000a
```

Obviously, strings with odd length will have 8 unused bits in the last position. This string is even-length (the terminating null counts). This, however, still makes the string much more compact than the equivalent unpacked string. Unfortunately, it also makes it more difficult to handle in machine code. CFT Forth makes heavy use of packed strings to keep the size of the dictionary down.

3.5.10 Skipping to the Next Page with `.page`

Since CFT is unable to reference 'far' addresses (across page boundaries) without indirection, the vast majority of loops must not cross page boundaries. Take this trivial loop, for example:

```
        LI 50
        NEG
        STORE R &10
loop:   ISZ R &10
        JMP loop
```

Since addresses in instructions are 10 bits long, only the lower 10 bits of the address of label `loop` are assembled in the `JMP` instruction. The upper six bits are populated using the upper six bits of the current address (i.e. the page). Now, assume `loop` is at address `1FFF`. This places `JMP` at address `2000`, crossing a page boundary. The address in `JMP` is still `3FF` (the lower 10 bits of `1FFF`, the address of `loop`). However, the upper six bits are no longer `1C00` (shifted) — they are now `2000`. Thus, the machine will jump to address `23FF` and the loop will fail. This is a problem common will all paged instruction sets.

The CFT Standard Assembler refuses to fix this automatically, instead producing a warning:

```
source.asm:5:warning: Label 'loop' on
  page 1c00, but page 2000 accessed instead.
```

To solve this issue, code must be rearranged so that the loop does not cross the page boundary. While developing, the easiest way to do this is to force crossing the page boundary by changing the origin. This is the purpose of the `.page` directive: it changes the origin to the first address of the next page.

Please note that this should not be done inside subroutines, since a block of unassembled values will be introduced. It should, instead, be placed just before the entry point of a subroutine preceding the problem addresses.

The syntax is trivial:

```
.page
```

A common issue is for `.page` directives to be left in the code while developing, leading to large amounts of unused space. The assembler will let the programmer know about potential cases of unnecessary `.page` directives by issuing a warning:

```
file.asm:924:warning: .page generated
  865 words of slack space.
```

In such case, it is standard practice to comment out the directive (since further coding is likely to push the problem code across another page boundary, so the `.page` could be needed again).

3.6 Instruction Macros

The CFT instruction set contains sixteen main instructions, of which two allow combinations of up to 24 minor operations. Because of both the small size of the instruction set and its versatility, many new instructions a programmer would expect to have may be formed as combinations (or special cases) of other instructions that are still technically one instruction.

For example, the CFT instruction set lacks shifts, but has rolls which roll the bits in the `AC`register through the single-bit `L`register. A shift may be implemented by clearing `L`before rolling, so that the new instruction `SBL` is implemented by `OP1 CLL RBL`.

These instruction macros are defined inside the Standard Assembler using the `.equ` directive and can be treated as *single* instructions. For example:

```
.equ   SBL   OP1 CLL RBL   ; Shift bit left
.equ   SBR   OP1 CLL RBR   ; Shift bit right
```

3.7 Assembler Macros

In addition to defining instruction macros which expand the CFT *instruction set*, the Assembler also provides Assembler Macros which expand the *assembly language*. Unlike instruction macros, assembler macros can incorporate multiple instructions and are parametric. They are invaluable in simplifying assembly listings and maintaining consistency.

3.7.1 Defining New Macros

New macros are defined with the `.macro` and `.end` directives. This is an example of an 'add with carry' macro to provide addition with carry-in (which cannot be implemented as a single CFT instruction):

```
.macro ADC (addr)
    OP1 IFL INC    ; Inc AC if L is set
    ADD %addr
.end
```

The name of the macro is `ADC`, and this is how it will be called after its definition. It accepts a single argument named `addr`. Please note that macros without parameters *still* require the open and closing parentheses, in the same way as C function definitions do.

Multiple arguments may be specified, separated by commas.

The code between `.macro` and `.end` is not assembled immediately — it is stored for inclusion later.

Argument values may be used by prefixing their names with `%`, hence the use of `%addr`.

3.7.2 Calling Macros

To call the `ADC()` macro, use this syntax:

```
ADC(R &10)
```

The presence of the parentheses in an instruction line trigger a search for the macro nominated before the opening parentheses. White space may be used between the name and open parenthesis, but is not recommended to keep the code compact and allow longer macro names.

Multiple argument values must be separated by commas.

The same number of values must be specified as there are arguments in the macro definition. Macro calls may not have variable arguments lists. In the case of `ADC`, exactly one argument may be specified. The value of the argument depends on its use in the macro definition, but the vast majority are parseable values.

3.7.3 Unique Labels

If a macro contains loops or declares labels and the macro is expanded multiple times, some or all of those labels may need to be made unique. This may be done by including the magic argument `%_` in any label names:

```
.macro MINILOOP(n, loopreg, op)
    LI %n
    NEG
    STORE %loopreg
miniloop%_:
    %op
    ISZ %loopreg
    JMP miniloop%_
.end
```

Note the idiom of appending `%_` to the labels. The `%_` argument will expand to a number to each macro expansion, making the label unique across the assembly process. This is generally not recommended. If loops are necessary, it may be better to implement them as subroutines.

4 Features

The CFT Assembler includes various features to aid in debugging the object code, simulating its execution in the CFT Emulator, as well as locating and fixing issues with the Assembler itself.

5 Output

At the end of the assembly process, the Assembler outputs a number of files, most of which are both machine and human-readable.

5.1 Object Files

The object code generated by the assembler is a pure binary file with no headers or magic numbers. They are files containing 16-bit words, one per assembled value, starting at the start address specified to the assembler, and ending at the end address, also as specified to the assembler.

5.2 Verilog List Files

To help using the code with the Verilog-based hardware verification suite, the Assembler also outputs a pair of `.list` files with suffixes `-00.list` (low order bits) and `-01.list` (high order bits). These are encoded in Verilog-compatible human-readable binary and represent low and high ROM images for the assembled code.

5.3 The Map File

The map file is generated with a `.map` suffix, and contains a map of the labels declared in the source code. It contains three columns:

1. Label Name.
2. Address in zero-padded hexadecimal.
3. Location of definition in the source code in `filename:line` format.

5.4 The Symbol Table File

Similar to the map file, this file (with a `.sym` suffix) contains a dump of the entire symbol table, not just labels. It has a two-line header for human readability, and contains lines arranged in four columns:

1. Symbol Name.
2. Value in zero-padded hexadecimal.
3. Symbol type:
   - `symbol`: a symbol declared by `.equ`.
   - `label`: a label.
   - `register`: a register declared by `.reg` or `.regfill`.
4. The location of the symbol definition in the source code in `filename:line` format.

5.5 The Preprocessed Assembly File

Files with a `.pasm` suffix are 'preprocessed assembly' files. They contain a digested form of the program, which is valid input to the CFT assembler, but much less human-readable. Each line contains an explicit address followed by the exact hexadecimal value at that address. The original assembly line leading to that value follows as a comment. Strings and other multi-word directives are expanded to show what is assembled. Include directives and macros are expanded. No labels or other symbols are used (but remain in the comments). Lines without assembled values are excluded. For example, label declarations on their own lines will be left out.

These files may be used as a kind of assembly-annotated hex dump of the object file, and are also used by the CFT emulator to print out the assembly source it is executing.

5.6 Graphical Memory Maps

These are bitmap images useful in visualising the memory map of the program, and the layout of the object file. They are output in the `.xpm` format, which *is* human readable, although these files are too large for a human to visualise without software. Like all `.xpm` files, this file is also valid C code. It may be viewed with most image viewers or converted to a binary bitmap file to save space.

The memory map displays the 65,536 word address space of the CFT as an array of 256×256 locations, with rows representing higher order bits of the address, so that address `0000` is at the upper left, and address `FFFF` is at the lower right.

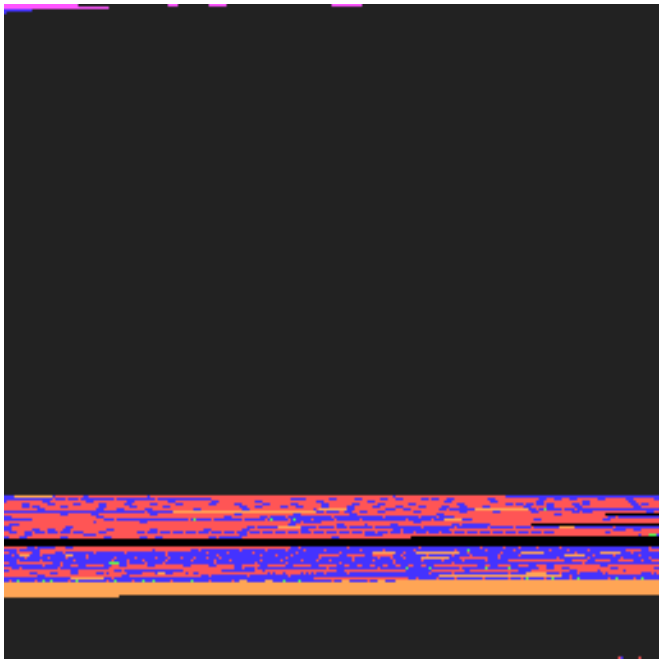

Figure 1: A graphical memory map of a CFT assembly program.

Grey represents unused locations. Black represents slack space left by `.page` directives. Magenta areas are those declared with `.reg`, and those are expected to be at or near the top, on page 0. Red represents assembled instructions. Orange represents packed strings. Green is for unpacked strings. Blue represents `.word` values, and cyan marks `.fill` areas.

6 Further reading

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

www.bedroomlan.org/hardware/cft