The WARM Assembler and Interpreter Guide

Duane A. Bailey

November 15, 2011 (software version 2.135)

1 Introduction

In order to convert the WARM assembler language instructions into the equivalent machine language, we use the WARM assembler, `waa`. To run an program on the hypothetical processor, we make use of the WARM interpreter, `wai`. This document describes use of these two programs in the workflow of writing programs for the WARM.

2 The Warm Assembler

As is true with other assemblers that we have used, the WARM assembler takes in assembly language files—files that contain assembly language instructions and directives, as well as comments that describe the logic behind the program. It is convenient to use the suffix `.s` at the assembly language programs. For example, the emacs editor will recognize this suffix and volunteer the use of asm-mode to edit your programs.

2.1 Instructions

Comments in the WARM assembler begin with a semicolon (`;`) and extend to the end of the current source line. In the emacs editor, you will find that typing a semicolon after an assembly language instruction will insert spaces until you get to a common comment column. If you type a single semicolon on a line by itself, it indents the comment and doubles the semicolon. Typically, this is comment for a region of code. If you type two semis on an empty line, three are inserted and the comment is forced to the left column. This is a comment appropriate for the top of your program.

The assembler keeps track of an instruction offset from the beginning of your program. This begins at one, and increases for each word of memory you define.

If you write an identifier—strings of letters and numbers beginning with a letter—in the left column, followed by a colon (`:`), the identifier is equated with the current instruction offset. These labels are typically the target of branches. If the label begins with an underscore—it is a *local* label—the label goes out of scope (becomes unavailable) after the next non-local label.

Operations are specified using the following general format:

```
label:  opcode  operands
```

Here, the assembler equates the label with the current instruction offset and assembles one instruction into the current word. Because the WARM has word-length instructions, each instruction occupies exactly one word, incrementing the current instruction offset by 1.

The opcode is one of a select group of identifiers that indicate the operation to be performed. You can read more about these operation codes in the WARM architecture manual. Each operand is the specification of a data value or location that will be required to complete the operation. For example, to add one to `r0` and place the result in `r1`, we might use

```
add     r1,r0,#1
```

Notice this instruction is not labeled. It is likely that this instruction is not the target of a branch. When values are required (for example in the immediate operand, above) they may be specified as a decimal value (the default), a hexadecimal value (beginning with an `0x`), an octal value (beginning with `0`), or an ASCII character value (preceded with an apostrophe, `'`).

2.2 Operand Specification

The WARM Architecture Manual describes the specifics of the operand modes that are legal for each type of instruction. We review, here, the format of these operands.

Register Direct: `r3`. This operand is simply the name or alias of the register involved. For example, `sub r1,r2,#2` the destination register, `r1`, and the left hand source, `r2` are always specified in register direct format.

Immediate: `#4`. Immediate operands are symbols or values prefixed with a hash mark. For example,

```
div     r1, r2, #0x17
```

divides `r2` by the value 23 (or `0x17`, hexadecimal).

Register Shifted Immediate: `r0, lsr #4`. A register, followed by a shifter operation and an immediate value indicating the number of bits shifted. Though there is a comma in the middle of the operand specification, the register, the shift operation and the immediate value form a single right hand source operand. If the shift operation and count are not specified, the effect is similar to a register direct.

Register Shifted Register: `r0, lsr r1`. A register, followed by a shifter operation and a second register indicating the number of bits shifted. Again, while there is a comma in the middle of the operand specification, all the parts form a single right hand source operand.

Register Product: `r2, r3`. Two registers specify a product to be used in a fused multiply add instruction.

Register Indirect: `[r1, #3]`. When in square brackets, the operand specifies a memory effective address used in a load/store type instruction. The register is called the base register, and the immediate value is the offset.

Register Indexed: `[r1, r2, lsl #3]`. This mode specifies two registers (a base and index register) and a shift operation with a bit count. It is used with load/store type instructions.

PC-relative: `gcd`. For branch instructions, which take a single operand, the use of label specifies a computation of a destination offset from the current instruction.

2.3 Assembler Directives

Assembler directives control the assembly process. They do not necessarily generate code.

The `.origin` Directive

The `.origin` directive resets the instruction offset to a particular value. If, for example, it is important to have the program start at a particular address, the origin directive allows you to specify the location of the following instructions. For example, the following code starts by jumping to a main procedure, located at address 100.

```
        b       main        ; all program start at offset 0
        .origin 100
    main:                   ; the true start of the program, at address 100
```

The `.align` Directive

The `.align n` directive resets the instruction offset to a multiple of n. For example, the following code starts by jumping to a main procedure, located at address 100.

```
        b       main        ; all program start at offset 0
        .origin 97
        .align 4
    main:                   ; the true start of the program, at address 100
```

The `.bss` Directive

The `.bss n` directive reserves memory for uninitialized data. For example, the following code reserves a buffer suitable for storing an 80 character string, with its zero terminator.

```
buffer: .bss   81
```

The `.equ` Directive

The `.equ` directive binds a value to a symbol. It is useful for giving names to constants. The command

```
    .equ    nl,10
```

defines the identifier, `cr`, to be 10. This can then be used in the command

```
    mov    r0,#cr
    swi    #SysPutChar
```

to print a newline.

2.3.1 The `.requ` Directive

Registers can be given alias names with the `.requ` command. It takes an alias and a register name as arguments. For example, one can give the name `fp` to `r11` with:

```
    .requ   fp,r11
```

From this point onward, `fp` can be used as an alternative name for `r11`.

2.3.2 The `.data` Directive

Raw data values may be stored at the current instruction pointer through the use of the `.data` directive. It has the form

```
.data   values
```

Each value that appears after the .data keyword is stored in the next location. The values may be specified as lists of constants:

```
monthLengths:
    .data   31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
```

The `.string` Directive

Zero-terminated ASCII strings can be savedq in successive memory locations starting at the current instruction offset. It has the form

```
.string "Hello world."
```

n Assembles the 12 ASCII values, one per memory location, followed by a zero terminator.

2.4 Running the Assembler

The WARM assembler is invoked with the `waa` command. Its simplest form is

```
waa program.s
```

This command assembles the instructions in `program.s` and writes a machine code object file in `program.o`. This file contains all the information needed to interpret the program.

To get a listing of the program, as it was assembled, use:

```
waa -l program.s
```

This leaves the listing in the file `program.lst`.

You can get more help on the waa command with

```
waa -h
```

3 The Warm Interpreter

The warm interpreter loads instructions from an object file into memory and then executes them, beginning with the instruction at location 0. The stack pointer is initialized to a suitable high-memory location. To simply run the program, you can use

```
wai program.o
```

The program is run, reading input from `program.in` and writing output to `program.out`. (The various input and output instructions are documented in the Section 4 where software interrupts are described.) You can run the program in an interactive mode, you can throw the `-g` switch:

```
wai -g program.o
```

This starts the program in an interactive debugger. The debugger supports a number of operations that allow you to step through the program one or more steps, to examine registers, or to set breakpoints. You can get help from within the debugger.

4 Software Interrupts

The WARM provides access to high level input and output functions through the software interrupt facility, the swi instruction. Currently the WARM interpreter supports the following operations:

Interrupt SysHalt (0), halt machine. When the user signals software interrupt 0, the program is halted.

Interrupt SysGetChar (1), read character. Software interrupt 1 causes the interpreter to read a single character from the input file. By default, the input file has same name as the program, but with the suffix .in. The resulting character is left in register `r0`.

Interrupt SysGetNum (2), read number. Software interrupt 2 causes the interpreter to read a single decimal value from the input file. By default, the input file has same name as the program, but with the suffix .in. The resulting value is left in register `r0`.

Interrupt SysPutChar (3), write character. Software interrupt 3 causes the interpreter to write a single character—found in register `r0`—to the output file. By default, the output file has same name as the program, but with the suffix .out.

Interrupt SysPutNum (4), write number. Software interrupt 4 causes the interpreter to write a single decimal value—found in register `r0`—to the output file. By default, the output file has same name as the program, but with the suffix .out.

Interrupt SysEntropy (5), generate random number. Software interrupt 5 generates a random value in register `r0`.

Interrupt SysOverlay (6), load overlay. Software interrupt `SysOverlay` loads an overlay file (specified by the `-x` switch on the interpreter) at the location specified in `r0`. Symbol definitions and equivalances are discarded.

Interrupt SysPLA (7), PLA access. The `SysPLA` interrupt passes `r0` to the loadable programmed array logic device and leaves the resulting value in `r0`. For more information on this device, read the *WARM/WIND PLA Guide*.