MrASM assembler

Embedded Systems Research Group
Summer Research Project 2004/5

Ravikesh Chandra
rcha108@ec.auckland.ac.nz

Introduction

One of the most important aspects of a customised processor design is powerful and easy to use accompanying software development tools. Good tools result in faster code development with fewer errors and possibly increased performance through compile time optimisations. MrASM, the Macro ReMiCORE Assembler, was specifically designed to address these development issues for the ReMiCORE processor and its customised derivatives.

Features

MrASM can be used as a basic assembler that purely translates assembly language constructs to ReMiCORE machine code; however it includes a number of enhanced features often found in higher-level programming languages to help software developers:

- Support for user-defined macros that act in a similar way to subroutines or functions
- Ability to include other source files at any position in a source file
- Support for compile-time conditional constructs
- Run-time configuration of the instruction set used
- Output to Altera MIF and Intel HEX file formats

Report structure

The first chapter of this report will be devoted to the usage of the MrASM assembler from an end-user point of view. The second chapter will detail the design and inner workings of the assembler.

TABLE OF CONTENTS

USAGE

DESIGN

USAGE

1 Compilation

MrASM has been successfully compiled and tested under Linux, and MS Windows when using the Cygwin POSIX emulation environment. It is written using ANSI C so it should compile with any ANSI C compliant compiler. However it makes use of the `flex` library with a minimum version of 2.5.31 required and the `bison` utility.

1.1 Stand-alone compilation

To generate a stand-alone compile from the MrASM distribution source compile with the following, assuming you are using the GNU `gcc` compiler:

```
yacc –d yaccer.y
lex lexer.l
gcc -c iniparser.c machine.c table.c macstack.c
gcc -c y.tab.c lex.yy.c mrasm.c
gcc *.o main.c –lfl –o mrasm
```

This will generate the assembler as an output binary named `mrasm`.

2 Running the assembler

The assembler can be run with the following general format:

```
mrasm <input_file> [options]
```

The *input_file* is the filename of the source file which must include extension. There are three options that can be specified either to change the ROM size in bytes which defaults to *1024* or change the output file name set which defaults to *out.** or to manually chose the input instruction set file which defaults to *mrasm.ini*.

For example,

```
mrasm filter.asm
```

This will assemble the file `filter.asm` assuming a 1024 byte ROM size and generate the output files: `out.hex`, `out.mif`, and `out.txt`.

Alternatively one or more options may be specified,

```
mrasm filter.asm –ofilter –s2048 –inewset
```

This will assemble the file `filter.asm` with a ROM size of *2048* bytes, using the instruction set defined within `newset.ini`, and generate the output files: `filter.hex`, `filter.mif`, and `filter.txt`.

3 Usage notes

3.1 Case sensitivity

MrASM is case-sensitive. Thus care must be taken to be consistent with variations in case. Therefore `Loop` is not equivalent to `loop`. All reserved words or keywords are uppercase; therefore legibility may be increased by making user-defined labels or symbols lowercase.

3.2 Comments

Comments are lines or text that is not processed by the assembler. These can be used to document a source file or for debugging purpose. A comment begins with a semicolon (`;`) and runs till the end of the line. They can be placed anywhere within an assembly file.

3.3 Number representation

Numbers can be represented in two ways. Either in base-10 using numbers normally, or in base-16 by prefixing the number with a dollar sign (`$`). For example twelve can be represented as either `12` in base-10 or `$c` in base-16.

3.4 User-defined keywords

User-defined keywords are used in the definition of custom symbols, labels, and macros. They must be at least two characters in length and begin with an alphabetic character (`a..z`, `A..Z`) and may be followed by any combination of alphanumeric characters or an underscore (`a..z`, `A..Z`, `0..9`, and `_`). Labels also have the additional constraint of being flush with the start of a line when being declared.

4 Assembler directives

4.1 Equate (EQU)

EQU is used to specify a user-defined symbol. The symbol label must follow the user-defined keyword rules and should never be flush with the left margin of a line. For example to define the SIR symbol to the value $FFFF_{16}$:

```
        SIR        EQU        $ffff
```

4.2 Origin (ORG)

The origin directive is used to set the location address for the following code. For example to write a block of code at 1000_{16} in the program ROM use:

```
        ORG       $1000
```

4.3 End of source (END)

The END directive is used to indicate the end of assembly code to be process in the input source file. Upon reaching this directive assembler processing of the source file will be terminated.

4.4 Conditional block (IF, THEN, ELSE, ENDIF)

Conditional assembly directives are evaluated at compile time. They can be used to implement conditionally executed code blocks in an assembly program. Conditional directives may be nested and can be used anywhere in a source file but the user must be aware of context. Usage follows the general structure of:

```
        IF test
        THEN
            true code
        ELSE
            false code
        ENDIF
```

The *test* condition can make use of a number of conditional operators: >, <, ==, `!=`, >=, <=. Also be aware that the `THEN` directive must be on a trailing line, and `ENDIF` must be specified. The `ELSE` statement and code block is optional.

4.5 Include (INCLUDE)

The include directive allows the user to directly include another assembly source file into the current file at the point it was called. Include calls can be nested. In conjunction with macro definitions, the include directive can be used to develop user code libraries. Note that included assembly files should not call the `END` directive as this will terminate the assembly process. For example:

```
        INCLUDE library.asm
```

4.6 Macro definition (MACRO, ENDMAC)

Macros are user-defined code blocks that may accept a number of input arguments and will be automatically included and expanded into an assembly program when called. Thus they are akin to subroutines or functions in higher-level programming languages but do not support a *return* mechanism to pass back values. Macros can be defined using arbitrary labels as parameters and then when called these can be substituted with formal arguments. For example the definition of a macro to add and subtract the same two parameters and store the result in R1 and R2:

```
addSub   MACRO (num1, num2)
         ADD R1 num1 num2
         SUB R2 num1 num2
         ENDMAC
```

The macro can then be called to operate on the contents of memory address 100_{16} and 101_{16} with:

```
         addSub ($100, $101)
```

5 Instruction set

MrASM was designed to allow for a flexible ReMiCORE based instruction set. Thus at run time the entire instruction set is loaded into memory from an initialisation file. There are ten different instruction formats that an instruction may take, depending on the specified operands. These are shown in Table 1.

Type	*Operand 1*	*Operand 2*	*Operand 3*
0	REG	REG	REG
1	REG	REG	NUM
2	REG	REG	IMM NUM
3	REG	NUM	
4	REG	REG	
5	REG	IMM NUM	
6	REG		
7	IMM NUM		
8	NUM		
9			

Table 1 Assembly instruction formats

5.1 Default instruction set

The default instruction set included with MrASM is for the ReMiCORE processor:

Mnemonic	Format Type	Operands	Operation

ABORT	3	S*i* address	AAAR ← address
			AASR ← S*i*

ABS	4	Rz Rx	Rz ← |Rx|
ADD	0	Rz Ry Rx	Rz ← Ry + Rx
ADDX	2	Rz Rx #value	Rz ← Rx + value
ADDY	2	Rz Ry #value	Rz ← Ry + value
AND	0	Rz Ry Rx	Rz ← Rx AND Ry

ANDV	2	Rz Rx #value	Rz ← Rx AND value

CAWAIT	1	S*i* S*j* address	SWR1 ← S*i*
			SWR2 ← S*j*
			SWA2 ← address

CLF	8	CZVN	If C, Cflag ← 0 If Z, Zflag ← 0 If V, Vflag ← 0 If N, Nflag ← 0
CMP	4	Rz Rx	Rz ← NOT Rx
EMIT	7	#signals	SOP ← signals
JMP	8	address	PC ← address

JSR	8	address	M[SR] ← address
			SR ← SR – 1
			PC ← address

LDR	3	Rz address	Rz ← M[address]
	4	Rz Ry	Rz ← M[Ry]
	5	Rz #value	Rz ← value
LDSP	7	#value	SR ← value
	8	address	SR ← M[address]
LRT	4	Rz Rx	Rz ← Rx <<< 1
LSR	4	Rz Rx	Rz ← Rx << 1
NOOP	9		No operation
OR	0	Rz Ry Rx	Rz ← Rx OR Ry
ORV	2	Rz Rx #value	Rz ← Rx OR value
PRESENT	3	S*i* address	If S*i* not present, PC ← address

PSH	6	Rx	M[SR] ← Rx
			SR ← SR – 1

PUL	6	Rz	SR ← SR + 1
			Rz ← M[SR]

RET	9		SR ← SR + 1
			PC ← M[SR]

RRT	4	Rz Rx	Rz ← Rx >>> 1
RSH	4	Rz Rx	Rz ← Rx >> 1
SAWAIT	6	S*i*	SWR1 ← S*i*
SC	8	address	If C, PC ← address
SN	8	address	If N, PC ← address
STR	3	Rx address	M[address] ← Rx
	4	Rx Ry	M[Ry] ← Rx
STRY*	5	Ry #value	M[Ry] ← value
SUB	0	Rz Ry Rx	Rz ← Rx – Ry
SUBV	2	Rz Rx #value	Rz ← Rx – value

SUSTAIN	7	#signals	SOP ← signals
SV	8	address	If V, PC ← address
SZ	8	address	If Z, PC ← address
TAWAIT0	5	#cycles PS*i*	CLK_SEL0 ← PS*i*
	7	#cycles	TIMER_REF0 ← cycles
TAWAIT1	5	#cycles PS*i*	CLK_SEL1 ← PS*i*
	7	#cycles	TIMER_REF1 ← cycles
TRF	4	Rz Rx	Rz ← Rx
TSTART0	5	#cycles PS*i*	CLK_SEL0 ← PS*i*
	7	#cycles	TIMER_REF0 ← cycles
TSTRAT1	5	#cycles PS*i*	CLK_SEL1 ← PS*i*
	7	#cycles	TIMER_REF1 ← cycles
XOR	0	Rz Ry Rx	Rz ← Rx XOR Ry
XORV	3	Rz Rx #value	Rz ← Rx XOR value

Table 2 Default MrASM instruction set

5.2 Configuring the instruction set

The instruction set is stored in an accompanying initialisation file, the default value is `mrasm.ini` but can be changed using the `-i` command line option. The instruction set can be edited by modifying this initialisation file.

An example instruction definition for the XOR mnemonic is shown below:

```
1| [XOR]
2| type=o
3| base=0c000000
4| format0=Izyx
5| format2=Izxv
```

The first line contains the mnemonic declaration. The *type* field in line two specifies the type that this definition represents. This should always be `o` to represent an operation. The *base* field in the third line indicates the operational code used for the instruction in 32-bit hexadecimal. The following lines specify the instruction formats used by the instruction. Up to 10 lines can be specified, one for each format type. Each line begins with a field made up from *format* concatenated with the number of the format type. Each format field begins with an uppercase letter to indicate the addressing mode, Table 3 shows the available modes. There can be zero to three trailing lower case fields to specify the mapping between assembly operands and the processor instruction word fields. The possible values are show in Table 4.

In the case of the XOR definition above, line four specifies `format0=Izyx`. Thus we are defining format type 0, immediate addressing mode is to be used, the first operand will be mapped to the Rz field in the instruction word, the second operand will be mapped to the Ry field in the instruction word, and the third operand will be mapped to the Rx field in the instruction word. In the fifth line format type 2 is defined: immediate addressing mode, first operand mapped to the Rz field, second operand mapped to the Rx field, and the third operand is mapped to the 16-bit operand field.

For example, lets assume we have re-configured the processor to include multiplication hardware and thus we know have a native multiplication command with a base opcode of $1a000000_{16}$. The multiplication instruction uses the immediate addressing mode and has two forms: inherent mode where multiplication is performed on two registers and immediate mode where the multiplication is performed on a register with a constant. In both cases the result is stored back into a register. Thus our assembly instructions could look like:

```
        MUL R3 R1 R2
        MUL R1 #CONSTANT
```

We can see that instruction format types 0 and 5 are used. The instruction can now be added to the MrASM initialisation file as follows:

```
[MUL]
type=o
base=1a000000
format0=Izyx
format5=Izv
```

5.3 Addressing mode and operand field designations

Designation	Addressing mode
D	Direct
I	Immediate/Inherent
R	Register Indirect
S	Stack

Table 3 Instruction set addressing mode designations used

Designation	Instruction word field
X	Register X
Y	Register Y
Z	Register Z
V	16-bit operand
S	Signal 1
B	Signal 2

Table 4 Instruction set word field designations used

6 Example—square root approximation

Let's look at combining all the assembler concepts by writing a program to compute an approximation of the square root. Let's define the algorithm:

$$\sqrt{A^2 + B^2} = \max((0.875x + 0.5y), x)$$

Where x and y are defined as:

$$x = \max(|A|, |B|) \quad y = \min(|A|, |B|)$$

We shall assume that the input variables A and B are received across the SIR input port with a 5-cycle delay between them. After computing the result we should output it across the SOR output port for a minimum of 5-cycles. The full program code is listed below.

6.1 SRA code listing

```
; -------------------------------
; MIN/MAX function
; -------------------------------
; INPUT:    a, b, type
; EFFECT:   if type == 1 R7=max(a,b)
;           if type == 0 R7=min(a,b)
; -------------------------------
minmax  MACRO (a, b, type)
        SUB R7 a b
        SN bLarger
        IF type == 1
        THEN
            LDR R7 a
        ELSE
            LDR R7 b
        ENDIF
        JMP end
bLarger IF type == 1
        THEN
            LDR R7 b
        ELSE
            LDR R7 a
        ENDIF
end     ENDMAC

; -------------------------------
; delay function
; -------------------------------
; INPUT:    start, finish
; EFFECT:   finish-start+1 NOOPs
;           will produced
; -------------------------------
delay   MACRO (start, finish)
        NOOP
        IF start < finish
        THEN
            delay (start+1, finish)
        ENDIF
        ENDMAC

; ------------------------------
; SQUARE ROOT APPROXIMATION MAIN PROGRAM
; ------------------------------

        ORG     $0

        ; symbolic constants
SIR     EQU     $ffff
SOR     EQU     $ffff
xVar    EQU     $8000
yVar    EQU     $8001

        ; load input variables R1=a, R2=b
        LDR     R1 SOR
        delay   (0, 4)
        LDR     R2 SOR

        ; get |a| and |b|
        ABS     R1 R1
        ABS     R2 R2

        ; compute x
        minmax  (R1, R2, 1)
        STR     R7 xVar

        ; compute y
        minmax  (R1, R2, 0)
        STR     R7 yVar

        ; R1 = 0.875*x
        LDR     R1 xVar
        RSH     R2 R1
        RSH     R2 R2
        SUB     R1 R1 R2

        ; R2 = 0.5*y
        LDR     R2 yVar
        RSH     R2 R2

        ; compute final result
        ADD     R1 R1 R2
        minmax  (R1, R3, 1)
        STR     R7 SOR
        delay   (0, 4)

        END
```

MRASM DESIGN

1 Operation overview

MrASM is a two-pass assembler which means that the assembly source-code is scanned twice during the assembly process. Each parse is identical and sequential in nature and aims to break down the assembler input into a series of tokens to match the recognized instruction formats of the assembly grammar, as well as analysing the assembler directives and features used by the programmer, for example macro, label, and symbolic constant definitions. In the first parse forward-referenced labels and other symbolic constants can not be evaluated, and thus the second pass is required to rectify this. During the second scan a program listing is created in-memory storing the processor machine code for each valid instruction of the assembly source-code. In the stand-alone assembler, after the parsing process is complete the program listing is scanned to generate the desired output files, in this case an Intel HEX format programming file and an Altera memory initialization file (MIF).

2 Assembler design structure

The design structure of MrASM is shown in Figure 1. For a stand-alone version the main module is included to provide the generated machine code as output files for ROM programming.

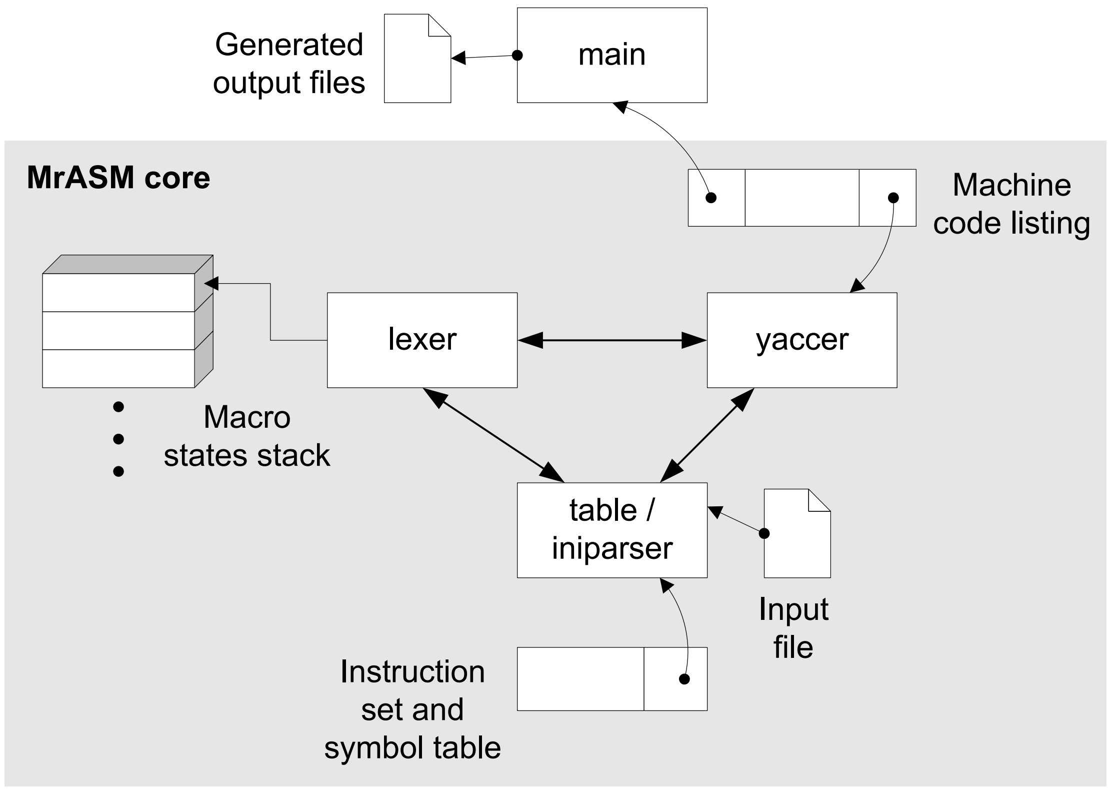

Figure 1 MrASM component diagram

The MrASM design consists of three main entities: the lexical analysis module, the grammar parsing module, and the MrASM core which controls the assembly process.

However there are a number of auxiliary modules and data structures used. The table module is a wrapper for the instruction set and symbolic references linked list and is used to perform operations on this data. The macro state module is a wrapper for the macro state stack which is used to support recursive macro calls by saving the assembler *state* onto a stack. And the machine code module is used as a wrapper for the machine code listing linked list which is essentially a table containing all the compiled machine code instructions. This machine code data structure is accessed by the MrASM main function to generate the output files.

2.1 Lexical analysis

MrASM makes use of the `flex` lexical analyser. The lexical analyser's job is to break down the input assembly code into a series of defined tokens which may include an accompanying data value. Input is matched by the use of *regular expressions* which specify the type of input to be looked for. At this initial stage it also examines the tokens and is responsible for recognizing certain assembler directives and take action accordingly.

The analyzer has multiple states so it may produce different tokens for the same recognized input when in different states. For example during a conditional assembly block the valid-condition code will be processed normally but for the invalid-condition code the analyzer is put into an ignore state and thus does not process any code.

The analyzer returns one of 11 possible defined tokens that are of three types. These types include *num* or a number type that is implemented using the integer data type; *str* or a string type that is a pointer to a character array in memory; and the *ptr* type to designate a pointer to an entry in the main symbol table.

For further detail on the lexical analyser operation please refer to the `lexer.l` source file and `flex` manual.

2.2 Grammar parsing

MrASM makes use of the `bison` grammar parser generator. Bison is used to analyse the tokens provided by the lexical analyser and with them attempt to form *grammars*. The grammar specifies the way in which tokens can form *sentences* or, specifically in the case of MrASM, *assembly instructions*.

MrASM uses a simple grammar that is primarily based upon the different types of instruction formats that are specified, as listed in Table 1. Once the tokens have formed a recognised assembly instruction the operands are used to compute the precise machine code instruction and this is stored in the machine code listing data structure.

For further detail on the grammar generator operation please refer to the `yaccer.y` source file and the `bison` manual.

3 Customising the assembler

MrASM has been designed as a complete and long-lasting assembler for the ReMiCORE processor and customised variations of it. However it has also been designed to be easily incorporated into other applications and thus there is a clear separation between the *assembly processing core* and the *compiled output generation* modules of the program, this can be seen in Figure 1 where the main module consists of the output generation and the MrASM core processes the input assembly code.

Thus by using this architecture it is easy to incorporate the MrASM assembler into different applications, for example a processor simulator. In this case the `main` module can be modified or re-written to directly incorporate the machine code level results of the assembler.

If the actual assembler functionality must be changed the assembler core can be modified. Modifications could alter the lexical analyser and or grammar definitions, `lexer.l` and `yaccer.y`. Modifications should endeavour to make use of the data structures and their wrappers that have been already provided.

4 Future work

In the future MrASM could be improved in a number of areas. One of these is more testing to remove undiscovered bugs. Support for more abstract data types could be included, for example simple variables and corresponding assignment and modifier operations. For better performance and reliability a move to a parser-tree design structure could be looked at as opposed to the current two-pass solution.

5 Acknowledgements

MrASM makes use of the `iniparser` initialisation file parsing library written by Nicolas Devillard. I would also like to acknowledge Tejas Mistry and Dong Hui for their work on the FLIX and ReMiCORE assemblers respectively as concepts from these assemblers were used.