flat assembler g

Contents

What is flat assembler g?

It is an assembly engine designed as a successor of the one used in flat assembler 1, one of the recognized assemblers for x86 processors. This is a bare engine that by itself has no ability to recognize and encode instructions of any processor, however it has an ability to become an assembler for any CPU architecture. It has a macroinstruction language that is substantially improved compared to the one provided by flat assembler 1 and it allows to easily implement instruction encoders in form of customizable macroinstructions. This approach has a great flexibility at the cost of performance.

If it happened that very fast assembly comparable to flat assembler 1 was required and the performance drawback of macroinstructions was not acceptable, it would be possible to address the issue by building a custom assembler based on this engine, and the complete source code is available to anyone who would like to try it. But the focus of this package is on the use of flat assembler g in its pure form.

The source code of this tool can be compiled with flat assembler 1, but it is also possible to use flat assembler g itself to compile it. The source contains clauses that include different header files depending on the assembler used. When flat assembler g compiles itself, it uses the macroinstructions that come with the supplied example programs, since they implement x86 instructions and formats with a syntax compatible with flat assembler 1.

The macroinstructions that process the syntax of x86 instructions are complex and take a long time to assemble, but incidentally the time it takes for flat assembler g to compile itself on an ordinary modern machine is comparable to the time that an early version of flat assembler 1 needed to assemble itself a decade and a half earlier on a computer that was then similarly mediocre. This can be seen as an interesting demonstration of how the software may be getting slower at the same rate as the hardware becomes faster.

The example programs for x86 architecture that come in this package are the selected samples that originally came with flat assembler 1, with an addition of the sets of the macroinstructions that implement instruction encoders and output formatters required to assemble them just like the original flat assembler did. While they are not complete, they are intended to encourage the creation of further sets of macroinstructions that would provide more instructions and output formats.

To demonstrate how the instruction sets of different architectures may be implemented, there are some example programs for the microcontrollers, 8051 and AVR. They have been kept simple and therefore they do not provide a complete framework for programming such CPUs, though they may provide a solid base for the creation of such environments.

There is also an example of assembling the JVM bytecode, which is a conversion of the sample originally created for flat assembler 1. For this reason, it is somewhat crude and does not fully utilize the capabilities offered by the new engine. However, it is good at visualizing the structure of a class file.

How does flat assembler g Work?

The essential function of flat assembler g is to generate output defined by the instructions in the source code. Given the one line of text as shown below, the assembler would generate a single byte with the stated value:

```
        db 90h
```

The macroinstructions can be defined to generate some specific sequences of data depending on the provided parameters. They may correspond to the instructions of chosen machine language, as in the following example, but they could as well be defined to generate other kinds of data, for various purposes.

```
        macro int number
                if number = 3
                        db 0CCh
                else
                        db 0CDh, number
                end if
        end macro

        int 20h         ; generates two bytes
```

The assembly as seen this way may be considered a kind of interpreted language, and the assembler certainly has many characteristics of the interpreter. However, it also shares certain aspects with a compiler. It is possible for an instruction to use the value which is defined later in the source and may depend on the instructions that come before that definition, as demonstrated by the following sample.

```
        macro jmpi target
                if target-($+2) < 80h & target-($+2) >= -80h
                        db 0EBh
                        db target-($+1)
                else
                        db 0E9h
                        dw target-($+2)
                end if
        end macro

                jmpi start
                db 'some data'
        start:
```

The "jmpi" defined above produces the code of jump instruction as in 8086 architecture. Such code contains the relative offset of the target of a jump, stored in either single byte or 16-bit word. The relative offset is computed as a difference between the address of the target and the address of the

next instruction. The special symbol "$" provides the address of current instruction and it is used to calculate the relative offset and determine whether it may fit in a single byte.

Therefore, the code generated by "jmpi start" in the above sample depends on the value of an address labeled as "start", and this in turn depends on the length of the output of all the instructions that precede it, including the said jump. This creates a loop of dependencies and the assembler needs to find a solution that fulfills all the constraints created by the source text. This would not be possible if assembler was just an imperative interpreter. Its language is thus in some aspects declarative.

Finding a solution for such circular dependencies may resemble solving an equation, and it is even possible to construct an example where flat assembler g is indeed capable of solving one:

```
x = (x-1)*(x+2)/2-2*(x+1)
db x
```

The circular reference has been reduced here to a single definition that references itself to construct the value. The flat assembler g is able to find a solution in this case, though in many others it may fail. The method used by this assembler is to perform multiple passes over the source text and then try to predict all the values with the knowledge gathered this way. This approach is in most cases good enough for the assembly of machine codes, but rarely suffices to solve the complex equations and the above sample is one of the exceptions.

What are the Means of Parsing the Arguments of an Instruction?

Not all instructions have a simple syntax like then ones in the previous examples. To aid in the processing of arguments that may contain special constructions, flat assembler g provides a few capable tools, demonstrated below on the examples that implement selected few instructions of the Z80 processor. The rules governing the use of presented features are found in the manual.

When an instruction has a very small set of allowed arguments, each one of them can be treated separately with the "match" construction:

```
macro EX? first,second
        match (=SP?), first
                match =HL?, second
                        db 0E3h
                else match =IX?, second
                        db 0DDh,0E3h
                else match =IY?, second
                        db 0FDh,0E3h
                else
                        err "incorrect second argument"
                end match
        else match =AF?, first
                match =AF'?, second
                        db 08h
                else
                        err "incorrect second argument"
                end match
        else match =DE?, first
                match =HL?, second
                        db 0EBh
                else
                        err "incorrect second argument"
                end match
```

```
                else
                        err "incorrect first argument"
                end match
        end macro

        EX (SP),HL
        EX (SP),IX
        EX AF,AF'
        EX DE,HL
```

The "?" character appears in many places to mark the names as case-insensitive and all these occurrences could be removed to further simplify the example.

When the set of possible values of an argument is larger but has some regularities, the textual substitutions can be defined to replace some of the symbols with carefully chosen constructions that can then be recognized and parsed:

```
        A? equ [:111b:]
        B? equ [:000b:]
        C? equ [:001b:]
        D? equ [:010b:]
        E? equ [:011b:]
        H? equ [:100b:]
        L? equ [:101b:]

        macro INC? argument
                match [:r:], argument
                        db 100b + r shl 3
                else match (=HL?), argument
                        db 34h
                else match (=IX?+d), argument
                        db 0DDh,34h,d
                else match (=IY?+d), argument
                        db 0FDh,34h,d
                else
                        err "incorrect argument"
                end match
        end macro

        INC A
        INC B
        INC (HL)
        INC (IX+2)
```

This approach has a trait that may not always be desirable: it allows to use an expression like "[:0:]" directly in an argument. But it is possible to prevent exploiting the syntax in such way by using a prefix in the "match" construction:

```
        REG.A? equ [:111b:]
        REG.B? equ [:000b:]
        REG.C? equ [:001b:]
        REG.D? equ [:010b:]
        REG.E? equ [:011b:]
        REG.H? equ [:100b:]
        REG.L? equ [:101b:]

        macro INC? argument
                match [:r:], REG.argument
                        db 100b + r shl 3
                else match (=HL?), argument
```

```
                db 34h
        else match (=IX?+d), argument
                db 0DDh,34h,d
        else match (=IY?+d), argument
                db 0FDh,34h,d
        else
                err "incorrect argument"
        end match
end macro
```

In case of an argument structured like "(IX+d)" it could sometimes be desired to allow other algebraically equivalent forms of the expression, like "(d+IX)" or "(c+IX+d)". Instead of parsing every possible variant individually, it is possible to let the assembler evaluate the expression while treating the selected symbol in a distinct way. When a symbol is declared as an "element", it has no value and when it is used in an expression, it is treated algebraically like a variable term in a polynomial.

```
element HL?
element IX?
element IY?

macro INC? argument
        match [:r:], argument
                db 100b + r shl 3
        else match (a), argument
                if a eq HL
                        db 34h
                else if a relativeto IX
                        db 0DDh,34h,a-IX
                else if a relativeto IY
                        db 0FDh,34h,a-IY
                else
                        err "incorrect argument"
                end if
        else
                err "incorrect argument"
        end match
end macro

INC (3*8+IX+1)

virtual at IX
        x db ?
        y db ?
end virtual

INC (y)
```

There is a small problem with the above macroinstruction. A parameter may contain any text and when such value is placed into an expression, it may induce erratic behavior. For example, if "INC (1|0)" was processed, it would turn the "a eq HL" expression into "1|0 eq HL" and this logical expression is correct and true even though the argument was malformed. To prevent this from happening, a local variable may be used as a proxy holding the value of an argument:

```
macro INC? argument
        match [:r:], argument
                db 100b + r shl 3
        else match (a), argument
                local value
                value = a
```

```
                        if value eq HL
                                db 34h
                        else if value relativeto IX
                                db 0DDh,34h,a-IX
                        else if value relativeto IY
                                db 0FDh,34h,a-IY
                        else
                                err "incorrect argument"
                        end if
                else
                        err "incorrect argument"
                end match
        end macro
```

There is an additional advantage of such proxy variable, thanks to the fact that its value is computed before the macroinstruction begins to generate any output. When an expression contains a symbol like "$", it may give different values depending where it is calculated and the use of proxy variable ensures that the value taken is the one obtained by evaluating the argument before generating the code of an instruction.

When the set of symbols allowed in expressions is larger, it is better to have a single construction to process an entire family of them. An "element" declaration may associate an additional value with a symbol and this information can then be retrieved with the "metadata" operator applied to a linear polynomial that contains given symbol as a variable. The following example is another variant of the previous macroinstruction that demonstrates the use of this feature:

```
        element register
        element A? : register + 111b
        element B? : register + 000b
        element C? : register + 001b
        element D? : register + 010b
        element E? : register + 011b
        element H? : register + 100b
        element L? : register + 101b

        element HL?
        element IX?
        element IY?

        macro INC? argument
                local value
                match (a), argument
                        value = a
                        if value eq HL
                                db 34h
                        else if value relativeto IX
                                db 0DDh,34h,a-IX
                        else if value relativeto IY
                                db 0FDh,34h,a-IY
                        else
                                err "incorrect argument"
                        end if
                else match any more, argument
                        err "incorrect argument"
                else
                        value = argument
                        if value eq value element 1 & value metadata 1 relativeto
register
```

```
                        db 100b + (value metadata 1 - register) shl 3
                else
                        err "incorrect argument"
                end if
        end match
end macro
```

The "any more" pattern is there to catch any argument that contains a complex expressions consisting of more than one token. This prevents the use of syntax like "INC A+0" or "INC A+B-A". But in case of some of the instructions sets, the inclusion of such constraint may depend on a personal preference.

The "value eq value element 1" condition ensures that the value does not contain any terms other than the name of a register. Even when an argument is forced to contain no more than a single token, it is still possible that is has a complex value, for instance if there were definitions like "X = A + B" or "Y = 2 * A". Both "INC X" and "INC Y" would then cause the operator "element 1" to return the value "A", which differs from the value checked in either case.

If an instruction takes a variable number of arguments, a simple way to recognize its various forms is to declare an argument with "&" modifier to pass the complete contents of the arguments to "match":

```
element CC

NZ? := CC + 000b
Z?  := CC + 001b
NC? := CC + 010b
C?  := CC + 011b
PO  := CC + 100b
PE  := CC + 101b
P   := CC + 110b
M   := CC + 111b

macro CALL? arguments&
        local cc,nn
        match condition =, target, arguments
                cc = condition - CC
                nn = target
                db 0C4h + cc shl 3
        else
                nn = arguments
                db 0CDh
        end match
        dw nn
end macro

CALL 0
CALL NC,2135h
```

This approach also allows to handle other, more difficult cases, like when the arguments may contain commas or are delimited in different ways.

How are the Labels Processed?

A standard way of defining a label is by following its name with ":" (this also acts like a line break and any other command, including another label, may follow in the same line). Such label simply defines a symbol with the value equal to the current address, which initially is zero and increases when any bytes are added into the output.

In some variants of assembly language, it may be desirable to allow label to precede an instruction without an additional ":" in between. It is then necessary to create a labeled macroinstruction that after defining a label passes processing to the original macroinstruction with the same name:

```
struc INC? argument
        .:
        INC argument
end struc

start   INC A
        INC B
```

This has to be done for every instruction that needs to allow this kind of syntax. A simple loop like the following one would suffice:

```
iterate instruction, EX,INC,CALL
        struc instruction? argument
                .: instruction argument
        end struc
end iterate
```

Every built-in instruction that defines data already has the labeled variant.

By defining a labeled instruction that has "?" in place of name it is possible to intercept every line that starts with an identifier that is not a known instruction and is therefore assumed to be a label. The following one would allow a label without ":" to begin any line in the source text (it also handles the special cases so that labels followed with ":" or with "=" and a value would still work):

```
struc ? tail&
        match :, tail
                .:
        else match : instruction, tail
                .: instruction
        else match == value, tail
                . = value
        else
                .: tail
        end match
end struc
```

Obviously, it is no longer needed to define any specific labeled macroinstructions when a global effect of this kind is applied. A variant should be chosen depending on the type of syntax that needs to be allowed.

Intercepting even the labels defined with ":" may become useful when the value of current address requires some additional processing before being assigned to a label - for example when a processor uses addresses with a unit larger than a byte. The intercepting macroinstruction might then look like this:

```
struc ? tail&
        match :, tail
                label . at $ shr 1
        else match : instruction, tail
                label . at $ shr 1
                instruction
        else
                . tail
```

```
                end match
        end struc
```

The value of current address that is used to define labels may be altered with "org". If the labels need to be differentiated from absolute values, a symbol defined with "element" may be used to form an address:

```
        element CODEBASE
        org CODEBASE + 0

        macro CALL? argument
                local value
                value = argument
                if value relativeto CODEBASE
                        db 0CDh
                        dw value - CODEBASE
                else
                        err "incorrect argument"
                end if
        end macro
```

To define labels in an address space that is not going to be reflected in the output, a "virtual" block should be declared. The following sample prepares macroinstructions "DATA" and "CODE" to switch between generating program instructions and data labels. Only the instruction codes would go to the output:

```
        element DATA
        DATA_OFFSET = 2000h
        element CODE
        CODE_OFFSET = 1000h

        macro DATA?
                _END
                virtual at DATA + DATA_OFFSET
        end macro
```

```
        macro CODE?
                _END
                org CODE + CODE_OFFSET
        end macro

        macro _END?
                if $ relativeto DATA
                        DATA_OFFSET = $ - DATA
                        end virtual
                else if $ relativeto CODE
                        CODE_OFFSET = $ - CODE
                end if
        end macro

        postpone
                _END
        end postpone

        CODE
```

The "postpone" block is used here to ensure that the "virtual" block always gets closed correctly, even if source text ends with data definitions.

Within the environment prepared by the above sample any instruction would be able to distinguish data labels from the ones defined within program. For example, a branching instruction could be made to accept an argument being either a label within a program or an absolute value, but to disallow any label of data:

```
	macro CALL? argument
		local value
		value = argument
		if value relativeto CODE
			db 0CDh
			dw value - CODE
		else if value relativeto 0
			db 0CDh
			dw value
		else
			err "incorrect argument"
		end if
	end macro

	DATA

	variable db ?

	CODE

	routine:
```

In this context either "CALL routine" or "CALL 1000h" would be allowed, while "CALL variable" would not be.

When the labels have values that are not absolute numbers, it is possible to generate relocations for instructions that use them. A special "virtual" block may be used to store the offsets of values inside the program that need to be relocated when its base changes:

```
	virtual at 0
		Relocations::
		rw RELOCATION_COUNT
	end virtual

	RELOCATION_INDEX = 0

	postpone
		RELOCATION_COUNT := RELOCATION_INDEX
	end postpone

	macro WORD? value
		if value relativeto CODE
			store $ - CODE : 2 at Relocations : RELOCATION_INDEX shl 1
			RELOCATION_INDEX = RELOCATION_INDEX + 1
			dw value - CODE
		else
			dw value
		end if
	end macro

	macro CALL? argument
		local value
		value = argument
		if value relativeto CODE | value relativeto 0
```

```
                        db 0CDh
                        word value
                else
                        err "incorrect argument"
                end if
        end macro
```

The table of relocations that is created this way can then be accessed with "load". The following two lines could be used to put the table in its entirety somewhere in the output:

```
        load RELOCATIONS : RELOCATION_COUNT shl 1 from Relocations : 0
        dw RELOCATIONS
```

The "load" reads the whole table into a single string, then "dw" writes it into output (padded to multiple of a word, but in this case the string never requires such padding).

What Options are there to Parse other kinds of Syntax?

In some cases, a command that assembler needs to parse may begin with something different than a name of instruction or a label. It may be that a name is preceded by a special character, like "." or "!", or that it is an entirely different kind of construction. It is then necessary to use "macro ?" to intercept whole lines of source text and process any special syntax of such kind.

For example, if it was required to allow a command written as ".CODE", it would not be possible to implement it directly as a macroinstruction, because initial dot causes the symbol to be interpreted as a local one and globally defined instruction could never be executed this way. The intercepting macroinstruction provides a solution:

```
        macro ? line&
                match .=CODE?, line
                        CODE
                else match .=DATA?, line
                        DATA
                else
                        line
                end match
        end macro
```

The lines that contain either ".CODE" or ".DATA" text are processed here in such a way, that they invoke the global macroinstruction with corresponding name, while all other intercepted lines are executed without changes. This method allows to filter out any special syntax and let the assembler process the regular instructions as usual.

Sometimes unconventional syntax is expected only in a specific area of source text, like inside a block with defined boundaries. The parsing macroinstruction should then be applied only in this place, and removed with "purge" when the block ends:

```
        macro concise
                macro ? line&
                        match =end =concise, line
                                purge ?
                        else match dest+==src, line
                                ADD dest,src
                        else match dest-==src, line
                                SUB dest,src
                        else match dest==src, line
```

```
                        LD dest,src
                else match dest++, line
                        INC dest
                else match dest--, line
                        DEC dest
                else match any, line
                        err "syntax error"
                end match
        end macro
end macro

concise
        C=0
        B++
        A+=2
end concise
```

A macroinstruction defined this way does not intercept lines that contain directives controlling the flow of the assembly, like "if" or "repeat", and they can still be used freely inside such a block. This would change if the declaration was in the form "macro ?! line&". Such a variant would intercept every line with no exception.

Table of Contents