The Moncky assembler

Kris Demuynck

17th August 2022

Abstract

This document describes the Moncky assembler that can translate assembly language to machine code. It also contains some tricks to program the Moncky-3 processor. The home of the Moncky project is

https://hackaday.io/project/181269-the-moncky-project

and

https://gitlab.com/big-bat/moncky

for all source code.

Copyright

Kris Demuynck

The Moncky assembler

Contents

Chapter 1

Introduction

This document describes the Moncky assembler. It is a program written in Java that translates Moncky assembly language to machine code. It supports all Moncky processors.

The assembler will take a text file as input and will output the machine code in different formats.

The assembler knows 3 modes:

- CLI mode: the assembler is run on a given file and exits after assembling
- GUI mode: the assembler shows a simple editor in which a user can type code. It is assembled immediatly, which is great for people who want to learn the language
- External mode: the assembler is called from another program that generated the assembly. A compiler can use this to automatically convert all generated assembly language to machine code.

The home page for the Moncky project is:

https://hackaday.io/project/181269-the-moncky-project

All code and documentation can be found in gitlab:

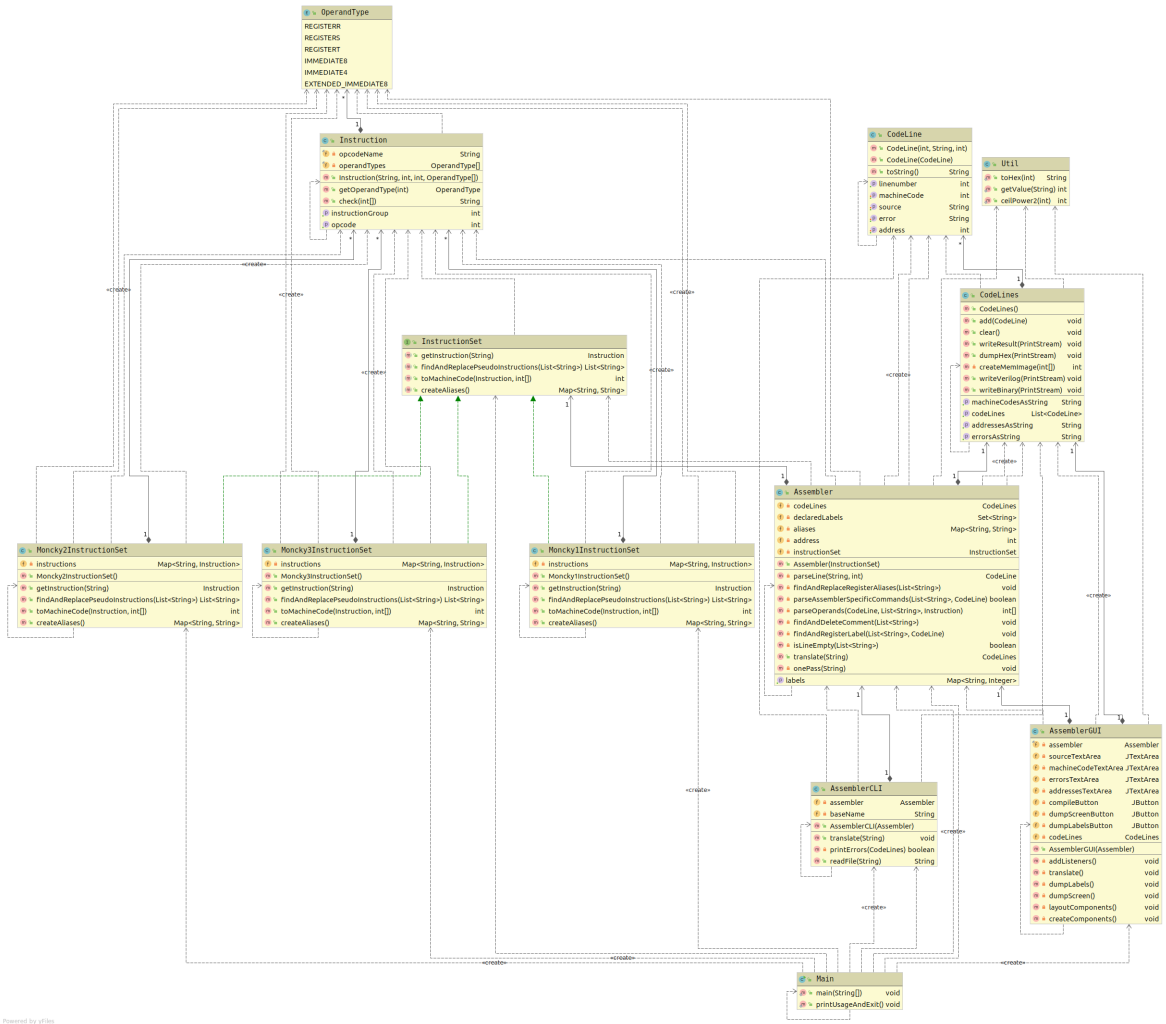
https://gitlab.com/big-bat/moncky

Chapter 2

Architecture

This chapter describes the overall architecture and functionality of the assembler.

The code was made so that extension to other processors can be done very easily. The class diagram looks like this:



The program starts in the Main class. There must be at least one command-line parameter which determines the instruction set used. It must be one of 'Moncky1', 'Moncky2', 'Moncky3'. If only one parameter is given the AssemblerGUI is used to show a simple graphical user interface in

which the user can type code and see the resulting machine code directly. When two or three parameters are given, the AssemberCLI is used. This has no graphical interface. It has the following extra command-line parameters: an optional path to the firmware files and the source file (must end in '.asm'). If a path to firmware is given, the files are added to the source.

Parsing is done in a very simple way. Every line of the source is regarded as one instruction (or comment). There are no multi-line instructions possible nor is it possible to put more than one instruction on one line. Each line is split into its tokens (called 'parts'). It uses spaces, tabs, braces, square braces, and the plus sign as white space. Characters can be specified between single quotes and string can be given with double quotes. Strings are automatically null-terminated and encoded with 8 bits per character (2 characters per word).

All instructions consist of an opcode followed by operands.

The parser is a two-pass parser. In the first stage, all code is read and labels are set to the correct values. In the second stage code is generated with the right values for all labels.

The output of the assembler consists of 5 files:

- a '.txt' file that contains the source code together with the generated code in a readable form
- a '.hex' file that can be used to load a ROM in Digital or Logisim
- a '.v' file that contains Verilog code that describes a ROM with the assembled code. This file can be used in the Verilog implementation of the Moncky-3 processor. It was used to develop the ROM code.
- a '.img' file containing a binary version of the code. This can be used to store in a file on an SD card. The Moncky-3 computer can read this file into memory and execute it. When the assembler runs in external mode the name of the file will be 'MONCKYOS'. If one puts this file on an SD card, it can be put in the Moncky computer which will execute the code within this file.

Chapter 3

Source files

The source files contain one instruction per line. Special instructions are implemented to guide the assembler and to make programming more easy.

3.1 Comments

Comments can be put in the code using ';'. The comment extends to the end of the line. This is comparable to '//' in the C programming language or '#' in scripting languages.

3.2 Assembler specific instructions

The assembler adds additional instructions that all start with a period '.':

.org with an address as parameter will set the current address to the given value

.def with a label and a value defines a label with the given value. The label must start with ':'. This can for instance be used to define the location of a variable or the value of a constant

.data followed by literals can be used to fill the memory with numbers. The literals can be decimal, hexadecimal, octal, or binary numbers. They can also be defined as a character if put between single quotes, or as a single string between double quotes (the string will be encoded with 8 bits/char and will be null-terminated).

.alias defines an alias. It is similar to the .def instruction, but aliases can also be 'unaliased'. An alias must start with a '$'. Aliases can be used to replace register names with variable names when a register is used to contain its value.

.unalias removes an alias

.rmAliases removes all aliases

3.3 Labels

In the beginning of each line a label can be specified. This will associate the label with the current address. Labels always start with a ':' character.

They can be referenced in an operand using ':' or '::'. The following code demonstrates this:

```
            andi r0, 255
            li r0, :addr
            lih r0, ::addr
            jpz [r0]
            ...
:addr       li r0, 42
            sub r0, r1
```

This code will jump to ':addr' when the value of the LSB of r0 is equal to zero.

The double colon '::' is used to get the MSB of the label.

As shown a label does not need to be defined before it is used.

3.4 Virtual opcodes

The assembler can add other opcodes that make code more readable. They map on other existing opcodes. For the Moncky-3 processor the following virtual opcodes are defined:

nop without any other operands will translate to 'nop r0, r0'. This causes the processor to do nothing but use one clock cycle (although it will set the flags according to the value of r0). It can be used for timing.

jpi with one immediate operand (addr) will translate to 'li r15, addr'. It will cause the processor to jump to a specific address. The address must fit in an 8-bit unsigned number. This is handy when writing small programs

jpr with one operand (a register r) will translate to 'add r15, r'. This will jump to an address relative to the current address. This can be used to write code that needs to be relocated

jpfi with one immediate operand (offset) will translate to 'addi r15, offset'. This will jump forward to an immediate offset relative to the current address.

jpbi with one immediate operand (offset) will translate to 'subi r15, offset'. This will jump backward to an immediate offset relative to the current address.

set with two operands (registers r and s) will translate to 'nop r, s'. This copies the value of register s to register r.

st with two operands (registers r and s) will translate to 'sti r, (s+0)'. It is equivalent to the 'st' instruction on the Moncky-1 processor.

ld with two operands (registers r and s) will translate to 'ldi r, (s+0)'. It is equivalent to the 'ld' instruction on the Moncky-1 processor.

ret with no operands translates to 'pop pc'. It returns from a subroutine by taking the return address from stack.

cmp with two operands (registers r and s) will translate to 'subf r, s'. It will cause the processor to subtract the two registers, only setting flags (not storing the result)

inc with one operand (register r) will translate to 'addi r, 1'. It increments a register

dec with one operand (register r) will translate to 'subi r, 1'. It decrements a register

3.5 Aliases

As described before, aliases can be defined using the .alias instruction. The Moncky-3 instruction set defines the following aliases by default:

- pc: this is an alias for r15 which is the program counter
- sp: this is an alias for r14 which is used as a stack pointer
- bp: this is an alias for r12 which can be used as a base pointer

3.6 Operands

Operands can be a register, a label, or a number.

A register must always begin with the character 'r' followed by a number between 0 and 15. It can also be an alias defined by the instruction set (sp, bp, pc).

Labels always start with a ':' so they can easily be recognised. Normally a label is replaced by the LSB of the value of the label. If the label starts with '::' the MSB is used. This can be used to quickly load an address into a register. For instance:

```
li r0, :addr
lih r0, ::addr
jp [r0]
```

This will load the value of :addr into r0 and jump to that location.

Numbers can be specified in decimal, hexadecimal (starting with 0x), octal (starting with 0), binary (starting with 0b), or as a character (between single quotes). They are checked to have the correct number of bits.

Chapter 4

Coding tricks

This chapter contains some coding tricks and hints that can be used when programming the Moncky-3 processor.

4.1 Loading 16 bit numbers

The 'li' instruction loads an 8 bit immediate number into a register. But in many cases one needs to load 16 bits into a register.

For this reason the 'lih' instruction was added to the instruction set. It loads an 8-bit immediate into the MSB of a register, leaving the LSB intact.

As an example loading the number 0x1234 into register r5 can be done as follows:

```
li r5, 0x34
lih r5, 0x12
```

Remark that switching the instructions will not work since the 'li' instruction clears the MSB of the register.

In order to load the value of a label into a register, one can use the ':' and '::' notation:

```
li r5, :address
lih r5, ::address
```

The double colon notation denotes the MSB of the label.

4.2 32 bit arithmetic

In order to do 32 bit arithmetic the carry bit can be used. Two examples are given here. It is assumed that registers r0 and r1 contain a 32 bit number. The LSB are in r0 and the MSB are in r1. Another number is present in registers r2 (LSB) and r3 (MSB).

Adding these numbers can be done as follows:

```
add r0, r2
addc r1, r3
```

The result is now in r0 and r1.

Subtracting these numbers can be done in a similar way:

```
sub r0, r2
subc r1, r3
```

4.3 Calling convention

There are many ways to call a subroutine with parameters. In the Moncky-3 firmware the following convention is always used (register r12 is called 'bp' which stands for 'base pointer'):

- parameters are pushed on the stack in the order of appearance (i.e. the parameters are in reversed order in memory). If a parameter is longer than 16 bits the MSB are pushed first.
- the subroutine can then be called (pushing the return address on stack)
- if there are parameters or local variables the subroutine will do 'push bp' to set the base pointer to the current stack frame
- the subroutine will then make room for local variables on the stack (with 'addi sp, -...')
- the subroutine will execute 'set bp, sp' to set the new value of the base pointer
- the subroutine can access the local variables and parameters with: 'ldi r0, (bp+...)'. The old base pointer and the return address are in between the local vars and the parameters. So those offsets must be taken into account.
- the subroutine is not expected to save the value of any register other than sp and bp.
- the subroutine puts the return value in r0. If more than 16 bits need to be returned register r1 is used to hold the most significant bits of the return value
- the subroutine ends by freeing up the space for local variables ('addi sp, ...'). Then it does 'pop bp' and finally 'ret' to return to the caller
- after this the parameters are removed from stack with 'addi sp, ...'

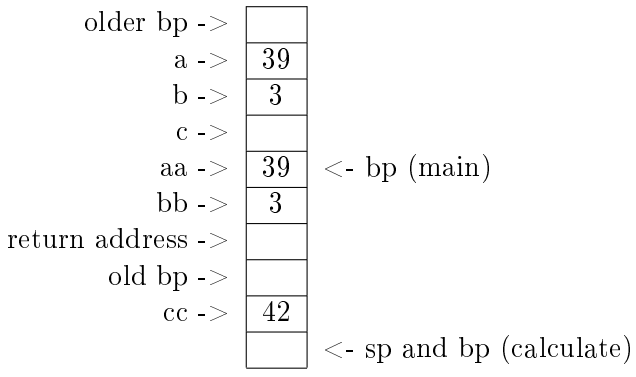
Here is an example in the Moncky programming language (explained in a separate document):

```
fun calculate(aa: uint16 bb: uint16): uint16
cc : uint16
{
    let cc = (aa + bb)
    return cc
}
main
a : uint16
b : uint16
c : uint16
{
    let a = 39
    let b = 3
    let c = call calculate(a b)
}
```

This can be translated (without any optimisation) as follows:

```
:calculate
push bp          ; save base pointer
addi sp, -1      ; make room for local var cc
set bp, sp       ; new base pointer
ldi r0, (bp+5)   ; read value of aa
ldi r1, (bp+4)   ; read value of bb
add r0, r1       ; add aa and bb
sti r0, (bp+1)   ; store cc
ldi r0, (bp+1)   ; set return value to cc
addi sp, 1       ; clean up local var cc
pop bp           ; restore base pointer
ret
:main
push bp          ; save base pointer
addi sp, -3      ; make room for a, b, and c
set bp, sp       ; new base pointer
li r0, 39
sti r0, (bp+3)   ; store 39 in a
li r0, 3
sti r0, (bp+2)   ; store 3 in b
ldi r0, (bp+3)   ; load value of a
push r0          ; first parameter aa
ldi r0, (bp+2) ; load value of b
push r0          ; second parameter bb
li r0, :calculate
lih r0, ::calculate
call [r0]
addi sp, 2       ; clean up parameters
sti r0, (bp+1)   ; store return value in c
addi sp, 3       ; clean up local vars a, b, and c
pop bp           ; restore base pointer
ret
```

The stack frame will look as follows (while executing calculate()):



4.4 Using masks

Masks can be used to set, unset or toggle individual bits without altering the other bits.

To set bits in a register, create a mask that has a 1 for each bit that needs to be set. Then 'or' the mask with the register. For instance, setting bits 0 and 5 in register r0 can be done as follows:

```
li r1, 0b00100001 ; mask
or r0, r1
```

To unset bits in a register, create a mask that has a 0 for each bit that needs to be unset. Then 'and' the mask with the register. For example, resetting bits 0 and 5 in register r0 can be done as follows:

```
li r1, 0b00100001 ; mask
not r1, r1        ; take inverse
and r0, r1
```

Bits can also be toggled using the 'xor' operation. For instance, toggling bits 0 and 5 in register r0 can be done as follows:

```
li r1, 0b00100001 ; mask
xor r0, r1
```

4.5 Unsigned overflow

The carry flag can be used to detect overflow for unsigned numbers. If two unsigned numbers are added, the carry flag will be 1 if an overflow occurred.

But this is not true for subtractions. In this case the inverse carry flag will indicate the overflow. This is because a subtraction is executed as an addition using 2-complement.

In fact $a - b$ is implemented as $a + \bar{b} + 1$.

The carry flag is not the same as a 'borrow' flag. The processor will set the carry flag to the value of the 'extra bit' that can appear. Some processors will invert this bit for a subtraction but the Moncky processor doesn't do that.

The programmer must be aware if a subtraction or an addition was done and use the carry bit accordingly.

4.6 Comparing numbers

Conditional jumps use the flags to determine if a jump needs to be done. The flags are set when an arithmetic operation is performed. They are left unchanged when no arithmetic operation is executed.

The standard way to compare numbers is to subtract one from the other so that the flags are set. This section gives a quick overview on how to perform most comparisons.

Imagine that the r0 contains the value of variable 'a' and r1 contains the value of variable 'b'.

When a and b are 2-complement signed integers the following comparisons can be done:

a==b subf r0, r1; jpz ...

a!=b subf r0, r1; jpnz ...

a<b subf r0, r1; jps ...

a>b subf r1, r0; jps ...

a<=b subf r1, r0; jpns ...

a>=b subf r0, r1; jpns ...

If a and b are unsigned integers, the comparisons are like this (we use the inverse of the carry flag in stead of the sign flag):

a==b subf r0, r1; jpz ...

a!=b subf r0, r1; jpnz ...

a<b subf r0, r1; jpnc ...

a>b subf r1, r0; jpnc ...

a<=b subf r1, r0; jpc ...

a>=b subf r0, r1; jpc ...

In the C programming language the result of a comparison can be stored in a Boolean variable. This is an integer that can be zero (false) or nonzero (true). The Moncky-3 processor has instructions to save the flags in a register. The value 'true' will be stored as 0xFFFF. In the following example the result of the comparison is put in register r2.

For signed integers the following code can be used:

a==b subf r0, r1; sz r2

a!=b subf r0, r1; snz r2

a<b subf r0, r1; ss r2

a>b subf r1, r0; ss r2

a<=b subf r1, r0; sns r2

a>=b subf r0, r1; sns r2

For unsigned numbers the following code can be used:

a==b subf r0, r1; sz r2

a!=b subf r0, r1; snz r2

a<b subf r0, r1; snc r2

a>b subf r1, r0; snc r2

a<=b subf r1, r0; sc r2

a>=b subf r0, r1; sc r2

4.7 Avoiding conditional jumps

Saving the status of a flag in a register has another advantage. It allows to avoid conditional jumps. This can be used for the following kind of expression:

```
c = (a==b)?d:e
```

The idea is to set c to the value of d if the condition is met. Otherwise c should be set to e. Let's assume the values of a, b, c, d, and e are kept in registers r0, r1, r2, r3, and r4 respectively. Then the following code will execute the previous statement:

```
subf r0, r1 ; compare a and b
li r5, :else
lih r5, ::else
jpnz [r5]
set r2, r3
li r5, :endif
lih r5, ::endif
jp [r5]
:else
set r2, r4
:endif
```

Now look at the following code. It will have the same result, but is does not contain any conditional jumps:

```
subf r0, r1 ; compare a and b
sz r5       ; save the zero flag (r5 is now 0 or 0xFFFF)
snz r6      ; save the inverse zero flag
and r5, r3  ; r5 is now 0 or d
and r6, r4  ; r6 is now 0 or e
or r5, r6   ; r5 now contains the right value
set r2, r5  ; save the result in the right register
```

Remark that the saved flag is used as a mask. When 'and'-ing this mask to a value, the result is 0 or the value. This code doesn't need any labels or conditional jumps and it will always execute in the same amount of clock cycles (which is less than the worst case of the previous code).

The latter code is also very interesting when the processor would have been pipelined since no branch prediction needs to be done.

4.8 Absolute value

Finding the absolute value of a 2-complement number can also be done without any conditional jumps. For example, the following code calculates the absolute value of r0:

```
set r1, r0   ; save r0 to r1
ashri r1, 15 ; extend the sign bit (r1 is now 0 or 0xFFFF)
xor r0, r1   ; invert all bits in r0 if r1==0xFFFF
sub r0, r1   ; add one if r1==0xFFFF
```