CajeASM Manual
by Tarek701

CajeASM is a MIPS R4300i Assembler for N64 ROM Hacking. It's made by Tarek701 (aka Cajetan) and it's goal is to replace the old LemASM and provide a newer better MIPS Assembler for creating ASM Hacks for several N64 ROMs.
The tool can be downloaded from SMWCentral.net

Preface

CajeASM is a N64 Assembler and specially made for the N64 ROM Hacking Community. The goal of CajeASM is, to replace older assemblers especially the known LemASM, which is now 14 years old. *CajeASM* provides a lot of nice features, which may come in handy for ASM Hackers. This is a manual, written by me, Tarek701, the creator of CajeASM, on how to use CajeASM and what features are provided. First off, let me introduce myself and who I am.

I am Tarek701, a staff member on SMWCentral and currently 17 years old. I don't have much to tell about me, except for that I have a high interest into N64 ROM Hacking in general and love to program in C#, C and sometimes C++. And I hate Python, lol. I first started with hacking N64 ROMs around 2007/2008 and learned a lot about assemblers and assembly in general. And here I am. Full of knowledge and ready to explain you my hard work in simple words.

So, that's basically everything I wanted to say. Now, you may start right in and read through the small reference which may guide you through CajeASM and act very well as reference. On the last page, there's a list of the instructions which are used here and what they stand for. Have fun!

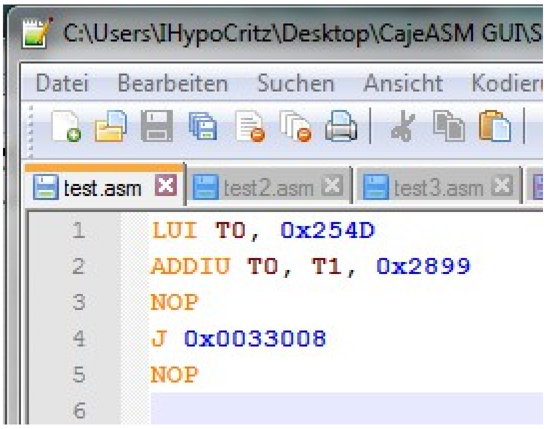
– Tarek701

How to assemble code with CajeASM?

CajeASM offers a very nice and user-friendly front-end, allowing you to quickly select your ROM and ASM code and assemble it to the ROM. The front-end is so easy, that you're able to intuitively realize what to click. There's also a help button, if there still should be any problem.

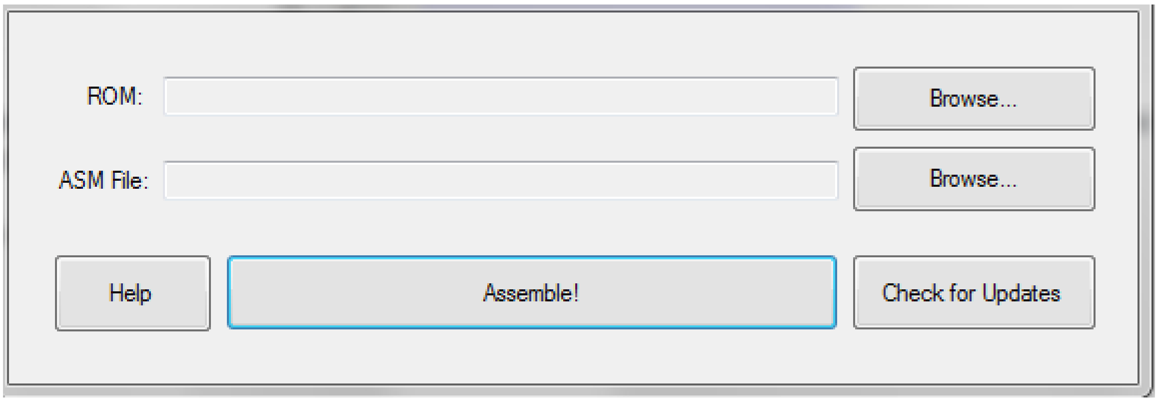
So, let's assemble something. It won't do anything, as this is just an example. But for the sake of testing, we do this now.

Open up Notepad++ (recommended; as I offer a special syntax highlighting Add-On which highlights your MIPS ASM code) or any Notepad program of your choice. Now, let's write some test code:

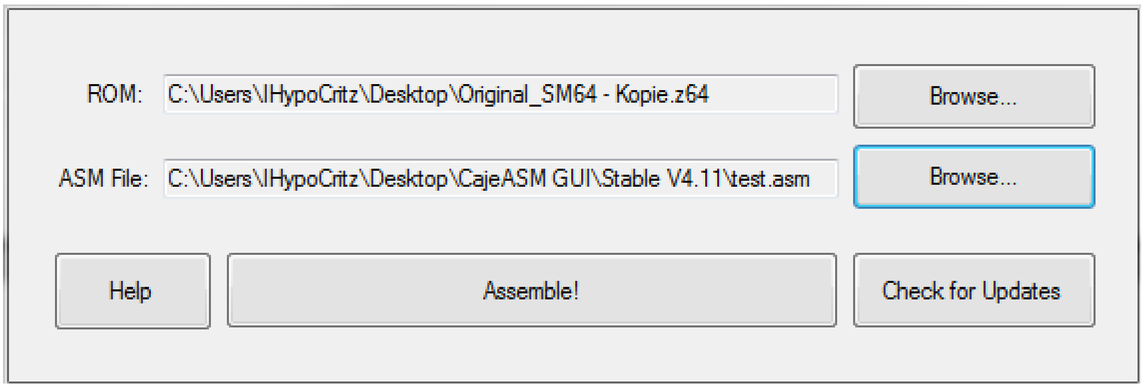


Once you're done, save your text as .asm, .s, .txt or anything else. (You will later have to select “All Files” in order to find the file with the special extension then.)

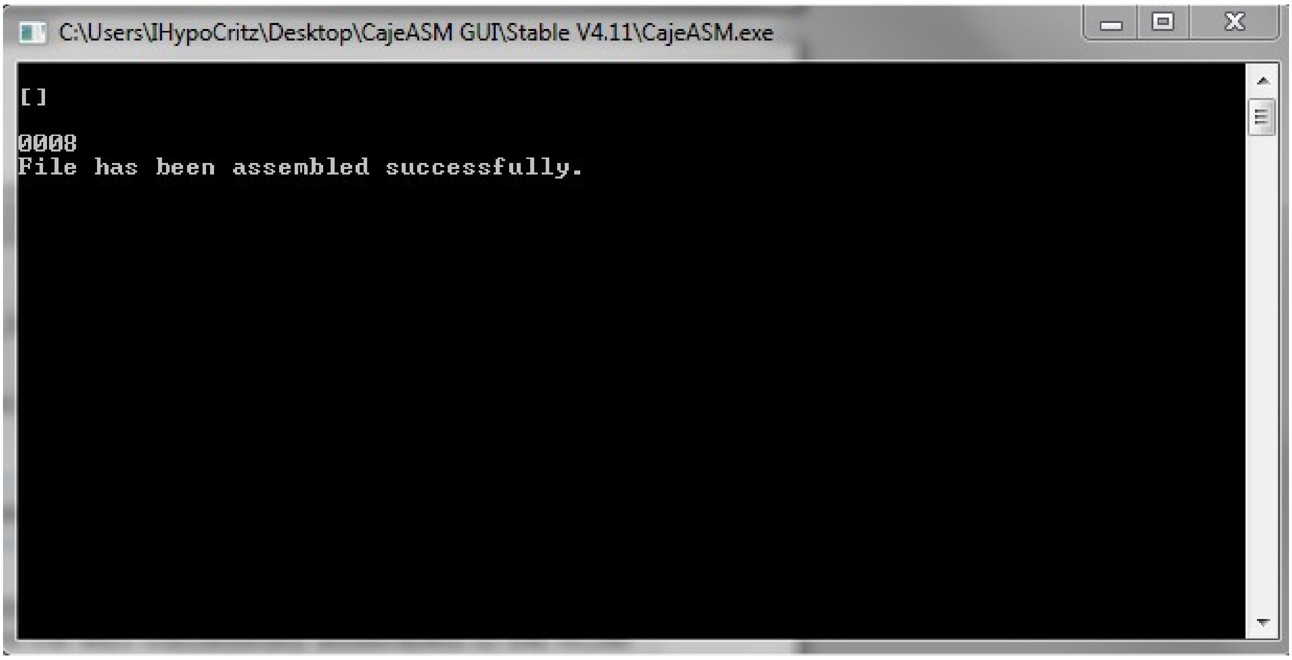
Now open up *CajeASM*. You should see the following:



This is the *CajeASM* front-end. As you can see, there's a “Browse...” button for ROMs and for your ASM File. Now, we select our ROM file and our ASM file (the one we just wrote). And then we have:

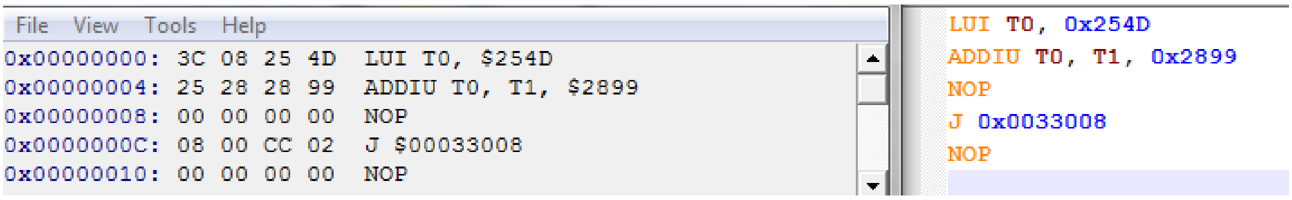


That's it. Now, you just press that long "Assemble" Button and a 2nd window will come up, telling you (if no problems occurred) that the file was successfully assembled to the ROM.



Now you can press any key (or press "X" on the upper right) to close the window. You might notice that two other windows will quickly open and close. But don't worry about this. This are the chksum64.exe and rn64crc.exe which recalculate your checksum and CRC of your ROM. This is required as some areas in the ROM are "checksum-protected" and changing this area without recalculating the checksum would result in an infinite loop respectively the ROM crashes. So, always check if both files are available and in the same folder like *CajeASM*.

After you know now how to assemble your code to a ROM, let's see if it actually worked. For this we open "LemASM". Unfortunately, *CajeASM* does not offer a "disassembling" feature (yet) (the opposite of assembling. Assembling = Assemble code to binary, Disassembling = Convert binary to readable code (aka ASM Code, like we wrote above in our text file)) and so we take LemASM to check if everything went right.



And it worked successfully. We assembled our first code with CajeASM to a ROM. Very well. LemASM is very useful for correcting small errors and/or disassemble code. For assembling code however, I strongly recommend using CajeASM as LemASM only allows line-per-line ASM coding and this can take extremely long and making one mistake could lead of rewriting the whole line again. So, please use *CajeASM* for "assembling" code and LemASM for "disassembling" code.

Basics: Labels, Directives, Pseudo-Instructions, Comments

Now, we come to one of the very neat features of CajeASM. These are pretty much the basic ones and are the "core" of CajeASM. They're important and you should make use of them when writing own code. Our first big subject are:

Labels

CajeASM offers the feature of setting "Labels". So, what exactly is a label? A label is a name immediately followed by a semi-colon ' **:** '. The "label" then represents the current value of the active location counter, and is, for example, a suitable instruction operand. Basically, it's calculating the current address of the instruction. It's a counter which increases by four. To show off an example:

```
ADDIU T0, T1, 0x44D0
ADD T0, T1, T2
BEQ T0, T1, Check
NOP
LUI T0, 0x444D
LH T0, 0x8880(T0)
NOP

Check:
LWC1 F0, 0x25D0(T0)
NOP
```

The expression "Check:" tells CajeASM, that we set a label here, while calling, respectively jumping/branching to label happens by writing the label name in the address operand we want to jump to. This happens in conditional jumps like the above one (BEQ) or in absolute, unconditional jumps like J or JAL. Simply said, CajeASM calculates the target address we want to jump to. As each instruction is four bytes long, you could also calculate the address yourself. Let's take the above example and count up 'till LWC1 (as this is the address the label gets).

```
ADDIU T0, T1, 0x44D0          // 0x00
ADD T0, T1, T2                // 0x04
BEQ T0, T1, Check             // 0x08
NOP                           // 0x0C
LUI T0, 0x444D                // 0x10
LH T0, 0x8880(T0)             // 0x14
NOP                           // 0x18

Check:
LWC1 F0, 0x25D0(T0)           // 0x1C, Check = 0x1C.
NOP
```

So, the instruction **BEQ T0, T1, Check** would be (once the code is assembled to the ROM):

BEQ T0, T1, 0x0000001C

So, it's pretty simple. However, not everyone has enough time to calculate the address, so CajeASM does it automatically by using Labels. So, they're really useful. We later learn that they can be useful for even more stuff.

Pseudo-Instructions

CajeASM offers so-called: “Pseudo-Instructions”. Pseudo-Instructions are, as the name implies, not real MIPS Instructions and are later converted to real ASM instructions. Usually we want to prevent a too high amount of pseudo-instructions as much as possible, so the amount will be kept small. I list the pseudo-instructions here and explain what they do.

BGE (Branch on Greater Than or Equal To), BGT (Branch on Greater Than), BGTU (Branch on Greater Than Unsigned), BLE (Branch on Lesser than or Equal to), BLT (Branch on Less Than)

These pseudo-instructions are, as you can see, for branching. As there are unfortunately no “greater than or equal to” (etc.) instructions, I've created pseudo-instructions which solve this problem. They're basically converted to SLT and a BEQ (or BNE).

To show off an example. (The syntax is always the same!)

BGE rd, rs, rt, address/label

rd = The register, where the temporary value is stored. It's mostly AT.
Rs = the first operand register which is compared to the target register.
Rt = the target register.
Address/label = The address or label to jump to if the condition is true.

An example:

BGE AT, **T0**, **T1**, **Check**
ADDIU T0, T1, 0x114A
NOP

Check:
JR RA
NOP

The above instruction (**BGE**) checks if T0 >= T1. If this is the case, we will jump to the check label.

If you would open up LemASM and check how the instruction looks converted, then we'll see:

SLT AT, T0, T1
BEQ AT, **R0**, **0x00000010**
ADDIU T0, T1, 0x114A
NOP

Check:
JR RA
NOP

As you the code is simply converted to a **SLT**. To explain what it does: **SLT** checks if T0 < **T1**. If this is the case, then **AT** is set to 1. Then **BEQ** checks if **AT** == **R0** (R0 is always zero, so we can simply write 0). To be more specific: If **AT** is 0, then we know that **T0** is either greater than or equal to **T1**, as it's **not less than T1**. Now with a simple **BEQ** check and checking if **AT** is 0, makes this a reality and so we have a "If … >= ..." statement. BLE works the same way, just that the registers switch places and it would be **SLT AT**, **T1**, **T0** instead of **SLT AT**, **T0**, **T1**. The same works with **BGT** and **BLT**, just that there's a **BNE**(Branch on Not Equal) instead of a **BEQ**.

BAL, B

BAL is "Branch and Link" and **B** (Branch) is a "direct" relative jump. Yes, they most likely won't have a big use. But according to some trustful sources, these instructions can save some processing time and may be useful for subroutines which are not too far away.

SUBI, SUBIU

SUBI and **SUBIU** are special two-pseudo-instructions. Basically, they're just **ADDI**/**ADDIU** instructions which contain the negative value of your positive value. For example, if you write:

SUBI T0, **T1**, **0x25**

The hex value would be multiplied by -1. This turns the number into a negative number. The translated instruction looks like this:

ADDI T0, **T1**, **0xFFDB**

Whereas 0xFFDB stands for "-25". In short, we add -25 to T1. This is logically equivalent to subtracting 25 from T1.

T1 + (-25) ↔ T1 – 25

BEQI, BNEI and BGTI, BLTI, BLEI, BGEI

They're most likely the same like BEQ, BNE, BGT, BLT, BLE and BGE. The difference is just, that instead of a target register, we are allowed to use an immediate value or a define(explaining later, but still showing the example).

The "I" behind the instruction names just stand for "Immediate".

Ex.:
BEQI T0, 0x2254, Label

This checks if T0 == 0x2254. If this is the case, we jump to Label.

[Def1]: 0x6666
BEQI T0, @Def1, Label

This checks if T0 == Def1. If this is the case, we jump to label.

And the same works with the other pseudo-instructions:

BGTI AT, T0, 0x254A, Label

This checks if T0 > 0x254A. If this is the case, we jump to Label.

[Def1]: 0x69AD
BGTI AT, T0, @Def1, Label

This checks if T0 > Def1. If this is the case, we jump to Label.

LI (Load Immediate)

LI let's you load an immediate value of a length over 32-bit. To make it short: If the immediate value is in 16-bit range(2 bytes) like 0x244D, 0x66AD, 0x254, 0x25, 0x2 etc. then the instruction is converted to an ADDI. Else if the immediate value is in 32-bit range(4 bytes) like 0x244D9, 0x444D8A, 0x244D920, 0xAAAACCCC, then the instruction is converted to a LUI (Load upper immediate) and ORI (OR Immediate)

Ex.:
LI T0, 0x2224

is translated to:
ADDI T0, T0, 0x2224

LI T0, 0x26AD5

is translated to:
LUI T0, 0x0026
ORI T0, T0, 0x0AD5

That's it.

Directives

Now, CajeASM allows (beside the pseudo-instructions) direct commands which give CajeASM orders. This are the so-called “Directives” which mostly start with a dot followed by the name of the order. I'm explaining each of them here.

.org

This directive tells CajeASM where to put the code following beneath under .org. Once you write another .org, the code beneath is then put to the new offset. You should look out when using labels. If you define a label before the .org, then the label location is not properly set. So always set the .org first and then put the label.

Ex.:

```
.org 0x4D40
ADDI T0, T0, 0x2224
ADD T2, T0, T0
NOP
SUB T0, T0, T6
NOP
```

This would put the code beneath .org to offset 0x4D40.

.ascii/.asciiz

This is the .ascii directive. As you may have suggested already, this directive let's you write an ASCII String to the target file. The difference between .ascii and .asciiz is that both write a string to the ROM while .asciiz however terminates the string with a NUL character (00 in Hex). In MIPS it's always recommended to use .asciiz as some games read 'till the next NUL character. The directive is called by “.ascii/.asciiz” followed by the text string between two quote marks.

```
.asciiz “This is a test”
```

This would write the ASCII-String to the target file and terminate it with a NUL character.

.include/.incl/.inc

This let's you include other asm files to your main asm file. This also includes all labels and defines of the included file. This may come handy in many situations, like creating a kind of library of specific game variables or game functions.

Ex.:

```
.include “test.asm”
```

This would include the asm file “test.asm” to your current asm file. It gets assembled before your code starts. Nested including is also possible, like that test.asm includes other files, etc.

Comments

CajeASM offers, besides all the technical stuff, a method to document your code properly. Those are “comments”. Comments are ignored by CajeASM and won't be assembled, as you may have noticed. This may come in handy when testing out a code and temporarily leave out a code without ignoring it or to document a code to make it easier for other people to understand it.

CajeASM offers two kind of comments. The first method being so-called “line comments”. Those comments are simply just for ONE line. So after a specific symbol the rest of the line is ignored. Then the 2nd method is a “block comment”. This block comment has a start symbol and an ending symbol. Everything between those two symbols is ignored by CajeASM and may be useful for a bigger documentation or a short summary what the whole code does.

To do a line comment, you either use:

; or //

And an example on use:

```
ADDIU T0, T1, 0x244D     // This adds T0 = T1 + 0x244D.
JR RA                    ; This jumps to the return address.
```

So, basically // and ; do the same. You may choose, which one of both you want to do more. The above is now a line comment. Basically everything after “//” and “;” is ignored TILL the next line! The MIPS syntax highlighter colors comments always green.

To do a block comment, you use:

/* … */

Everything between /* and */ is ignored.

Example:

```
/* The instruction below calculates
T0 = T1 + 0x244D. Then the next
instruction is a jump to the return address. */
ADDIU T0, T1, 0x244D
JR RA
```

That's it already. It's always recommended that you document your code properly or else many people are gonna have a hard time figuring out what the code does.

Advanced: Defines and Hexwrite

Now, we come to a more complex topic. Defines and Hexwrite. This may be eventually be useful for more advanced users. This is also one of the special features CajeASM offers!

Defines

You may have seen such a "Define" (or variable) above already. Basically, they're nothing else than "name" macros which can be later called in Immediate Type instructions. Defines contain values. You define them and later use them in your instructions. Such a variable can be defined by writing:

[VarNameHere]: <Value>

To show off an example:

```
[Coins]: 0x8034B218
```

This now means, that define/variable "Coins" now contains the value, 0x8034B218. CajeASM creates a log file showing you that the variable was found. Now, if we want to actually make use of it, we have to call the variable by using the "@" prefix and writing the variable name behind it. Depending on the instruction, it will load now the upper half and lower half of the 32-bit value.

Ex.:

```
[Coins]: 0x8034B218

LUI T0, @Coins
LH T1, @Coins(T0)
```

The above instruction would be later translated to:

```
0x00000000: 3C 08 80 34  LUI T0, $8034
0x00000004: 85 09 B2 18  LH T1, $B218 (T0)
```

Exactly. LUI loads only the "upper half" (respectively 0x8034), while LH and many other lower half instructions load the lower half (respectively 0xB218). This is very useful when creating a list of specific variables, like above for example Mario's Coin Counter. Thanks to this, we can now simply write "@Coins" and it will load (after LH) the current coin amount of Mario into T1. If you now create a big list of variables and later use an include on it, this can be really useful.

However, this isn't everything yet. If we want to load a small value, you can simply shorten it to a 16-bit range. Then there's no lower half anymore and it loads the 16-bit half only.

Next is (which I explain later more in detail) that you can also write decimal or binary values into defines. A decimal value is always prefixed with "#" and a binary value is always prefixed with "%".

Ex.:

```
[Var1]: #255

LUI T5, @Var1
```

This will then be later translated into Hex, resulting in:

```
0x00000000: 3C 0D 00 FF  LUI T5, $00FF
```

As 255 (decimal) is in hex: 0x00FF. So, we see $00FF in LemASM.

If we want a binary value, we write the prefix “%” and then the value behind it:

```
[Var1]: %101010

LUI T5, @Var1
```

And translated we get:

```
0x00000000: 3C 0D 00 2A  LUI T5, $002A
```

As binary value “101010” means 0x2A in Hexadecimal.

Hexwrite

Another feature is so-called: “Hexwrite”. Hexwrite allows you write raw hex bytes into the target file. It may be useful in many cases, like for example overwriting behavior scripts in SM64 (which is not ASM code!). To tell CajeASM to write raw hex to the ROM, you have to write:

hex { hexbytes here }

Everything between the brackets must be a hex.

Ex.:

```
hex { 24 58 D9 24 }    0x00000000: 24 58 D9 24  ADDIU T8, V0, $D924
```

As I said earlier, this may be useful for editing specific values or non-ASM code.

Decimal, Binary and Hexadecimal Values

CajeASM offers also a value types. I-Type Instructions can take either decimal, binary or hexadecimal values. Each value type has a "prefix" indicating which system we take.

Immediate values starting with a "#" prefix are *decimal values*. (Ex.: #25)
Immediate values starting with a "%" prefix are *binary values*. (Ex.: %101011)
Immediate values starting with a "0x" or "$" prefix are *hex values*. (Ex.: 0x24D, $24D)

To show off a few examples here:

```
ADDIU T0, T1, #242      ; Would be 0xF2 (Hex)
ORI T2, R0, %101011     ; Would be 0x2B (Hex)
LUI T4, $258D
LUI T1, 0x22BC
```

All in all, it's really simple and useful when for example check for a specific decimal coin amount. Always calculating the hex value can be annoying, so you can take the decimal prefix instead and write the decimal number behind it. That makes it really easier for many people.

Warnings and Notes relating to CajeASM

These are a few warnings you should consider when you code in CajeASM. Basically, the most important are:

1. *First .org, then Label!*

The above means that when you write to a specific location, that you always look that the .org stands before the label! Or else the location is still set to the default one or older one (if there was another .org before).

To illustrate it:

```
Label:
.org 0x8440
LUI T0, 0x244D
BEQ T0, T1, Label
NOP
```

WRONG!

And:

```
.org 0x8440
Label:
LUI T0, 0x244D
BEQ T0, T1, Label
NOP
```

RIGHT!

2. Labels and Variables/Defines with same names are overridden! The label offset is set to the NEWEST label name.

This means, if you define a label or variable which has been defined before, then it's overridden to the current one. Ex.:

```
.org 0x8440
Label:
LUI T0, 0x244D
BEQ T0, T1, Label
NOP

Label:
ADDIU T0, T1, 0x244D
NOP
```

BEQ wouldn't (as you first thought) jump to the first “Label”. Instead it jumps to the newest definition below. The same with variables/defines:

```
[Var1]: 0x234D
[Var1]: #120

.org 0x8440
Label:
LUI T0, @Var1
BEQ T0, T1, Label
NOP

Label:
ADDIU T0, T1, 0x244D
NOP
```

LUI wouldn't (as you first thought) load the hex value 0x234D. Instead it loads the newest definition with the decimal number “120”. So, please remember this. Always.

List of MIPS Instructions used in this manual

LUI = Load Upper Immediate → Loads a value to the upper half (first four bytes from left to a register.)

ADD = Add → Adds register contents and stores the result in a destination register.

ADDI = Add Immediate → Adds register content with a immediate value and stores the result in a destination register.

ADDIU = Add Immediate Unsigned → Does the same like ADDI, just that no overflow exception is thrown if the value is over 0x7FFF.

BEQ = Branch on Equal → Checks if the content of two registers is equal. If this is the case, the code will branch to an offset.

BNE = Branch on NOT Equal → Checks if the content of two registers is not equal. If this is the case, the code will branch to an offset.

SLT = Set on Less Than → Checks if the source register is less than a target register. If this is the case, the destination register is set to 1. Otherwise, it is set to 0.

LWC = Load Word to Coprocessor → This loads a word (32-bit) from the memory to a coprocessor general register.

LH = Load Halfword → Loads a value from an address to a register by calculating the base register + the given immediate value and loads the value into a destination register.

NOP = No Operation → Does nothing. Mostly used in delay slots for Jumps or Branches.

JR = Jump Register → Jumps to the content of the register.

ORI = OR Immediate → Does an OR operation with a register and an immediate value and shifts the immediate value to the first four digits from right.

List of MIPS Registers used in this manual

R0 **:** First register (R0 = Register 0) and is permanently zero. It can't be changed.

T0 – T7 : Temporary Registers, often used for temporary operations which are later overridden.

AT **:** Reserved by assembler and is commonly used for SLT operations.

SP **:** Stack Pointer. Points to the top of the stack. Increments from high to low memory.

RA **:** Return Address register. Contains the current return address (the address we came from and we later jump back to).

F0 - F31: Floating-Point registers. Commonly used for floating-point operations.