

# CajeASM v7.0+ Manual

## Table Of Contents:

### Basics:

1. Introduction into *CajeASM*
2. Command-Line Options
3. CajeASM GUI

### Coding:

#### 1. Pseudo-Instructions

- 1.1. BLT (Branch on less than), BGT (Branch on greater than)
- 1.2. BGE, BLE (Branch on greater/less than or equal to)
- 1.3. BLTI, BGTI (Branch on less/greater than immediate)
- 1.4. BGEI, BLEI (Branch on less/greater than or equal to immediate)
- 1.5. BEQI/BNEI (Branch on equal/not equal to immediate)
- 1.6. SUBI/SUBIU Instruction
- 1.7. LI (Load Immediate) Instruction
- 1.8. MOV (move) Instruction
- 1.9. B (Branch)
- 1.10. BAL (Branch and Link)
- 1.11. CL (Clear)

#### 2. Directives

- 2.1. .org address
- 2.2. .byte/.halfword/.word/.float
- 2.3. .align alignment, (optional) fill
- 2.4. .skip size, (optional) fill
- 2.5. .incbin "filename.bin"
- 2.6. .incasm/.inc/.include "filename.asm"
- 2.7. hex { hex values }
- 2.8. .ascii/.asciiz

#### 3. Labels & Defines

#### 4. Additional Information


- 4.1. About Upper/Lower Values
- 4.2. Decimal, Hexadecimal, Binary values
- 4.3. Comments

# **Basics**

**Welcome, soon-to be ASM Hacker. I'm going to lead you through this manual. First we start with the very basics and explain what CajeASM is, what it does and how to use the console-commands and the GUI. If you don't want to know anything about the command-line version of CajeASM, just skip to the CajeASM GUI part. But be sure to read "Introduction into CajeASM" if you're new.**

## **1. Introduction into CajeASM**

So, what is "**CajeASM**" you might ask. Well, CajeASM is a **N64 Assembler**, more specific a **MIPS R4300i Assembler**. An assembler does nothing else than "converting" human-readable code into machine binary, aka the binary which the MIPS CPU is able to read. We call this "human-readable" code, the so-called "Instruction Set" and it's basically all the coding you saw before. A great example would be:



```
LUI T0, 0x8033  
ORI T0, T0, 0xB264
```

The code above is our "human-readable" code. It later gets translated to something like this:



```
10101011001101001111110101010101010
```

And that's basically what CajeASM does. So, instead of figuring out what each binary order is supposed to do (just imagine if you have around 3000 lines of binary.. ewww) you simply learn the human-readable way and then write your code, which CajeASM then

translates into MIPS Binary. So, the “human-readable” code is also called: “Assembly” or shortened to simply “ASM”. If you actually want to learn MIPS ASM, then look out on origami64.net as I'm going to post there my ASM Tutorial soon.

## **2. Command-Line Options**

Let's move on to the console command line options. If you're not interested into the command line way, then skip this and look at chapter “CajeASM GUI”. Basically, CajeASM is a console-command application and let's you pass a few arguments and allows you to tell the assembler directly what “command” it should do.

Here, the console-command list:

```
CajeASM - N64 Assembler by Tarek701<Cajetan> <For help, PM me on origami64.net>
Syntax: CajeASM [options]
Options:
-a /rom:<filename> /asm:<filename> Assembles ASM file to your ROM.
-u Checks for updates.
-l Create listing file to show symbol locations.
-n /asm:<filename> -o:<filename> Assemble only and output bin.
-h To see this help message again.
```

The first option is “-a”. Basically, it stands for “assemble” and does what it says. It assembles an ASM file of your choice to a N64 ROM of your choice. So, you would start CajeASM like this:

CajeASM -a /rom:myROM /asm:test.asm

This would now start CajeASM and assembles the code in test.asm to your ROM.

The next option is “-u”. It lets you check if there are any updates available. It can take a while until CajeASM reacts (because it connects to the internet etc.) and sooner or later you receive a console message whether there's an update or not.

CajeASM -u

The third option is “-l”, it creates a listing file to show symbol locations. It's useful for identifying where global data or procedures are placed by the assembler in case you need to look them up or

pass them into a debugger.

```
CajeASM -l /rom:myROM /asm:test.asm
```

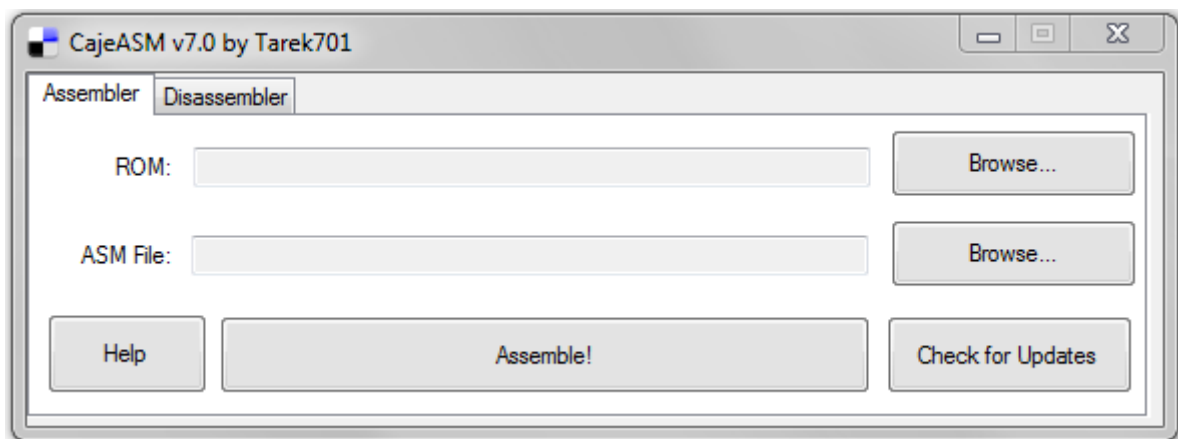
The fourth option is “-n” is nearly the same as -a, just that your code isn't assembled to an already existing file, but instead put into a new file.

```
CajeASM -n /asm:test.asm -o:output.bin
```

The fifth and last option is “-h” and is nothing else than the help message itself again.

### **3. CajeASM GUI**

The CajeASM GUI is probably most likely what you want to see because you probably don't give a fuck about command-lines and it's way too hard for you. The GUI is the front-end of the console-application and is faster and easier for normal users and/or for those who want to assemble their code quickly.



As you can see above, the GUI is really simple to understand. It's user-friendly and intuitive. Anyone should really understand what each button is supposed to do. Select ROM, select ASM file, press “Assemble!” and your code is assembled to the ROM and done. So, there's nothing more to say to this.

# **Coding**

**Now we finally come to the more interesting part, I guess. The “coding” part. Here, I'm going to (sum it up) explain the functionality of CajeASM, including pseudo-instructions, directives, the way how CajeASM handles decimal, hex and binary values, labels and defines, etc. If you're not aware of what some instructions here do, look at the end of the text file. There's a list of all instructions used in this manual.**

## **1. Pseudo-Instructions**

The first chapter of the coding section. What are “pseudo-instructions” you might ask. As the name implies already, pseudo-instructions are nothing else than instructions, that do **not** truly exist on the *MIPS* Instruction Set. These instructions are often made to give the ASM programmer shortcuts and faster ways of assembling their code without writing the same 2 or 3 lines again. A great example would be BGE (Branch on greater than or equal to) which is not a real MIPS instruction, but is implemented as pseudo-instruction into CajeASM and CajeASM automatically translates it to the real two instructions (SLT and BNE). So, the ASM programmer doesn't have to care about this again and again. To sum it up: Pseudo-instructions are not real MIPS instructions and are translated by the assembler. They're mostly shortcuts for ASM programmers, so they don't have to consider any of this stuff themselves.

### **1.1 BLT (Branch on less than), BGT (Branch on greater than)**

This are our first two pseudo-instructions. **BLT** and **BGT**. **BLT** checks if the source register is less than the target register and **BGT** does the opposite and checks whether the source register is greater than the target register. As you can most likely guess, these are no real MIPS instructions. They're actually translated to two real MIPS instructions. Let me give you an example.

```
ORI T0, R0, 0x3645
ORI T1, R0, 0x3314
BGT T0, T1, 0x00003680
NOP
```

The above instruction now checks if **T0** > **T1**. This is the case, as **T0** is **0x00003645** and **T1** is **0x00003314**. So, we would now jump to the specified offset. But... what does the instruction look like when it's translated to a real MIPS instruction? Here's the answer:

```
SLT AT, T1, T0
BNE AT, R0, 0x00003680
```

So, what happens there? First, **SLT** checks if **T1** < **T0**. If this is the case, then **AT** = 1. You see now, that both registers were just swapped around. And obviously **T1** < **T0** is equivalent to **T0** > **T1**. Now **BNE** (Branch on NOT equal, real MIPS instruction) checks if **AT** != 0. (**R0** is the first register, but it's always 0. The value can't be changed) As this is the case, we now branch to the specified offset. CajeASM also allows you to let you use a different destination register for **SLT**. If you didn't specify it, like I showed above with **BGT**, then on default **AT** is used. If you want a different one you write:

```
ORI T0, R0, 0x3645
ORI T1, R0, 0x3314
BGT T2, T0, T1, 0x00003680
NOP
```

Which is then translated to:

**SLT T2, T1, T0**  
**BNE T2, R0, 0x00003680**

The same, I told you, applies to **BLT** also. The only difference is that **BLT** is translated to a different real MIPS instruction. Can you guess it? Correct. It simply puts **T0** as source and **T1** as target, so we get **T0 < T1** in **SLT**. Very simple.

**SLT AT, T0, T1**  
**BNE AT, R0, 0x00003680**

See, it's really simple. And also the same like above applies and you can also specify a different destination register for **SLT**.

### Syntax:

**BGT/BLT**        **RS** , **RT** , **OFFSET** (or **label**)  
**BGT/BLT** **RD** , **RS** , **RT** , **OFFSET** (or **label**)

### **1.2 BGE/BLE (Branch on greater/less than or equal to)**

This part is not going to be any more special than the last one, as this is exactly the same like above. The syntax is still the same. But still, I'm going to explain how the instruction do look like if they're translated to real instructions.

Let's start with BGE. BGE checks whether the source register is greater than or equal to the target register. BLE does the opposite and checks whether the source register is less than or equal to the target register. Let's do a quick example:

```
ORI T0, R0, 0x1058
ORI T1, R0, 0x1058
BGE T0, T1, 0x000861CC
NOP
```

This checks now if **T0**  $\geq$  **T1**. This is the case and so we branch to the specified offset. The instruction is translated to:

```
SLT AT, T0, T1
BEQ AT, R0, 0x000861CC
```

This time nothing is swapped here, instead we now use **BEQ** instead of **SLT**. To clarify it: First **SLT** checks if **T0**  $<$  **T1**. This is NOT the case, as they're equal. So, **AT** = 0. Now **BEQ** checks if **AT** == 0. This is the case and that's why we jump. Even if **T0** was **0x3665**, **AT** still would be 0, as it checks if it is less than the target register. And therefore it also would jump.

And now how would we do this with **BLE**? Well, you kinda guessed it (maybe) we just swap **T0** and **T1** again in **SLT**.

```
SLT AT, T1, T0
BEQ AT, R0, 0x000861CC
```

It checks if **T1**  $<$  **T0**. This is not the case, as **T1** and **T0** are equal. So we get **AT** = 0. And then **BEQ** checks if **AT** == 0. Again, this is the case and we branch. Now if **T0** had the value **0x0056**, **SLT** checks if **T1**  $<$  **T0**. That's not the case. **T1** is bigger. So we get again **AT** = 0 and then **BEQ** will branch again.

The difference is, as you see now, not that really big. Also, the same with the “destination register” applies like explained in 1.1.



## Syntax:

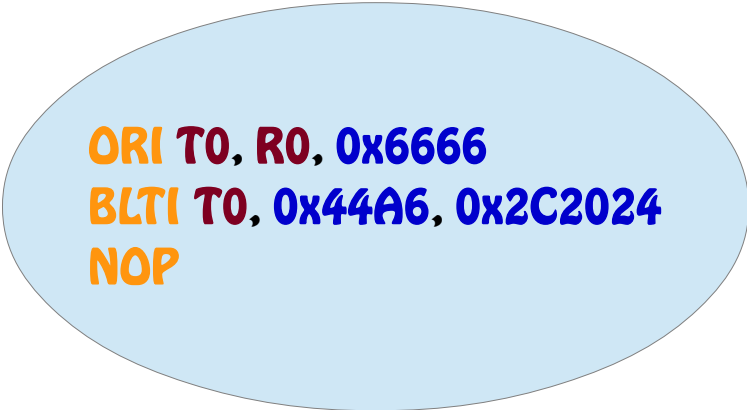
BGE/BLE        RS , RT , OFFSET (or label)

BGE/BLE RD , RS , RT , OFFSET (or label)

### 1.3 BLTI/BGTI (Branch on less than/greater than immediate)

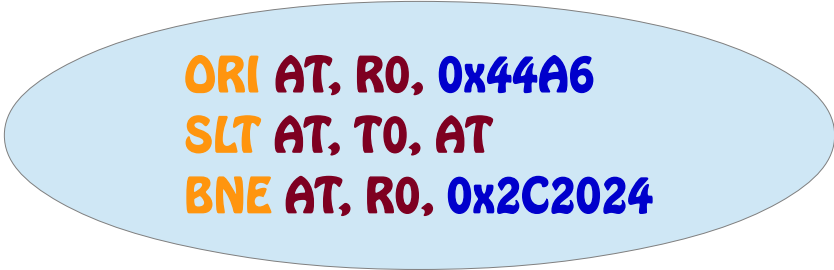
The purpose of these two instructions are exactly the same like BLT and BGT, the only difference being that you're able to specify an immediate value. Basically BLTI checks whether the source register is less than the immediate value and BGTI checks whether the source register is greater than the immediate value. The translation is not different, just that there's now one more instruction.

Ex.:



```
ORI T0, R0, 0x6666  
BLTI T0, 0x44A6, 0x2C2024  
NOP
```

This checks now if **T0** < 0x44A6. This is not the case, so we won't branch. The translation of BLTI to real MIPS code:



```
ORI AT, R0, 0x44A6  
SLT AT, T0, AT  
BNE AT, R0, 0x2C2024
```

You see, the difference is pretty small once again. First we load 0x44A6 into the lower half of **AT** (**AT** = **0x000044A6**) and then we check if **T0** < **AT**. That's not the case. So, **AT** = 0. And BNE checks

now if **AT** != 0. **AT** is 0, so we won't branch. You see, it's pretty simple once again. The same with BGTI.

Ex.:

```
ORI AT, R0, 0x44A6
SLT AT, AT, T0
BNE AT, R0, 0x2C2024
```

Again, the difference is like with **BGT**. We just swap **AT** and **T0**.

### Syntax:

```
BLTI/BGTI      RS , IMM (or define), OFFSET (or label)
BLTI/BGTI RD , RS , IMM (or define), OFFSET (or label)
```

### 1.4 BGEI/BLEI (Branch on greater/less than or equal to immediate)

This one here also isn't special. As you know already what **BGE** and **BLE** do, this one doesn't need any more information. Just showing you the translation.

For BGEI:

```
ORI T0, R0, 0x6666
BGEI T0, 0x44A6, 0x2C2024
NOP
```

Translated:

```
ORI AT, R0, 0x44A6
SLT AT, T0, AT
BEQ AT, R0, 0x2C2024
```

For BLEI:

```
ORI T0, R0, 0x6666  
BLEI T0, 0x44A6, 0x2C2024  
NOP
```

Translated:

```
ORI AT, R0, 0x44A6  
SLT AT, AT, T0  
BEQ AT, R0, 0x2C2024
```

Syntax:

```
BGEI/BLEI    RS , IMM (or define), OFFSET (or label)  
BGEI/BLEI RD , RS , IMM (or define), OFFSET (or label)
```

### 1.5 BEQI/BNEI (Branch on Equal/not Equal to)

This is the last branch pseudo-instruction. It's basically clear what this does and it's not really needed to mention what it does. The difference is here that the source register is compared to an immediate value. And don't fucking come up and ask why I added these branch-immediate instructions. I did it, because I wanted it. If you hate it, don't use it. Just saying.

**BEQI:**

```
ORI T0, R0, 0x6666  
BEQI T0, 0x44A6, 0x2C2024  
NOP
```

**BEQI** is Translated to:

**ORI AT, R0, 0x44A6**  
**BEQ T0, AT, 0x2C2024**

For BNEI:

**ORI T0, R0, 0x6666**  
**BNEI T0, 0x44A6, 0x2C2024**  
**NOP**

is translated to:

**ORI AT, R0, 0x44A6**  
**BNE T0, AT, 0x2C2024**

### Syntax:

**BEQI/BNEI**        **RS** , **IMM** (or **define**), **OFFSET** (or **label**)  
**BEQI/BNEI** **RD** , **RS** , **IMM** (or **define**), **OFFSET** (or **label**)

### **1.6 SUBI/SUBIU (Subtract Immediate (Unsigned))**

Now, after we're done with pseudo branch instructions, we will now continue with the other instructions. This one here is SUBI/SUBIU. If you know MIPS, then you also know that there's no subtract immediate instruction, instead there's only a register-type instruction (**SUB**) available, but that's it. So, to “subtract” values you just made use of the negative rule. All immediate values over **0x7FFF** are considered negative. So, you just converted the number you want to subtract from the target register, into a negative number.

**SUBI/SUBIU** prevents this and simply lets you put in the normal

number already (without converting to negative) and CajeASM assembles it correctly to **ADDI/ADDIU**.

Ex.:



**SUBI T0, T1, 0x4444**

The above subtracts  $T1 - 0x4444$  and stores the result into T0. CajeASM translates it to the real MIPS instruction:



**ADDI T0, T1, 0xBBBC**

**0xBBBC** = **-0x4444**. Basically said. I think there's nothing more to add to this as this is kinda clear. It's simply a shortcut for ASM programmers so they won't have to take out their fucking calculator.


### Syntax:

**SUBI/SUBIU**      **RD** , **RT** , **IMM** (or **define**)

### **1.7 LI (Load Immediate)**

This pseudo-instruction is a bit more interesting. Simply said, it allows you to load immediate values greater or less than 16-bit. In this case **LI** translates to two different ways, depending on which range the immediate value is in.

Ex.:



**LI T0, 0x3648**

This instruction will load the immediate value **0x3648** into **T0**.  
CajeASM translates the above instruction to:



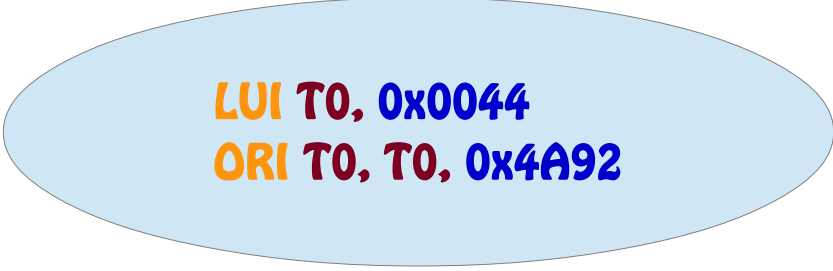
**ORI T0, R0, 0x3648**

No surprise, I guess. But **LI** also allows you to specify a value which is above the 16-bit range. In this case **LI** translates differently. Ex.:



**LI T0, 0x444A92**

This would load the immediate value **0x444A92** into **T0**. CajeASM translates the above to:



**LUI T0, 0x0044**  
**ORI T0, T0, 0x4A92**

Once again, if you know these two simple instructions then you

know that this is quite obvious. **T0** is **0x00440000** first and then **ORI** shifts **0x4A92** to the lower half of **T0**, resulting in **T0 = 0x00444A92**.

And that's all I can say about it.

### Syntax:

**LI**      **RT** , **IMM** (or **define**)

### 1.8 MOV (move) instruction

This instruction is also simple to explain. Basically it let's you move the target register content to the target register content. It's also a pseudo-instruction, but the translation is pretty simple. You should be able to get it quickly.



**MOV T0, T1**

Would move **T1** to **T0**.

**MOV** is translated to:



**ADD T0, R0, T1**

As you see, pretty simple. **T0 = R0 + T1**.

## **1.9 B (Branch) Instruction**

B, stands for “branch”, and is a branch without a condition. People who think this is useless, please turn on your brain. Simply said, branches can't jump far in memory, so if you want position-independent code, then B can be actually useful.

Ex.:



**B 0x00006000**

is translated to:



**BEQ R0, R0, 0x00006000**

That's it basically. As I said, this can be actually useful for people who want position-independent code.



### **1.10 BAL (Branch and Link) Instruction**

The next pseudo-instruction is similar to the real MIPS instruction JAL (Jump and Link) being the only difference that branch does not have a far range like JAL and is once again useful for position-independent code.

Ex.:



**BAL 0x0046000C**

Would branch and link to 0x0046000C. Translation:



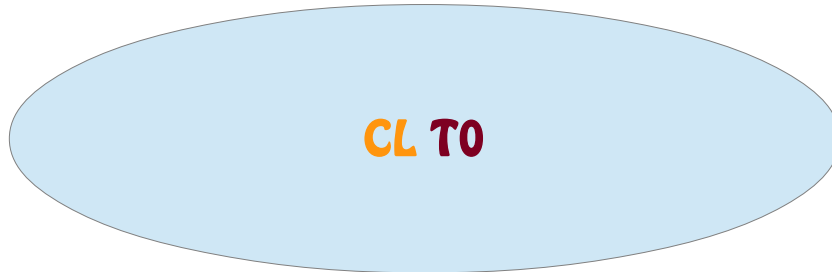
**BGEZAL R0, 0x0046000C**

Again: None of these instructions have to be used. They're only useful for people who want their code to be position-independent.

### **1.11 CL (Clear) Instruction**

This is one of the instructions I personally would call useless, however, who knows, maybe it is useful somewhere. Basically it just clears a whole instruction and sets it to **0x00000000**.

Ex.:



Translated to:



I guess that one is clear too. Why I implemented it? I have no idea. I implemented it earlier (around CajeASM v2.1) and it stayed there since then and didn't delete it so far. Maybe it's useful somewhere, who knows.

## 2. Directives

After we're done with the pseudo-instructions and you now know of the pseudo-instructions, we now may move on to something more assembler-specific: "Directives". Directives allow the ASM programmer to give direct commands to the assembler itself. It "directs" and "leads" the assembler and are not related to MIPS instructions itself. In short, they're not translated or written to your ROM and don't influence it. It only influences the assembler. I'm going to explain each directive here. Trust me, they're really useful.

### 2.1 .org address

This is our first directive and most likely going to be the most used directive. This directive tells the assembler to put the following code to the ROM offset you specify in there. Addresses are either prefixed with '0x' or '\$'. Both is allowed. That goes for all other immediate instructions. (Later more to this)

Ex.:



```
.org 0x861C0  
ADDIU T0, T1, 0x33A6
```

This puts the following code below **.org** to ROM offset **0x861C0**. I think I don't have much to add here, it's really simple to understand. But one thing: Yes, you can put more than one **.org** directives, respectively you can put as much **.org** directives you want:



```
.org 0x861C0  
ADDIU T0, T1, 0x33A6  
  
.org 0x86300  
ORI T0, S1, 0x4466  
  
.org 0x86400  
ANDI T0, A0, 0x0022
```

## **2.2 .byte/.halfword/.word/.float**

This directive is a bit more special. It let's you insert numeric values to the ROM. This can be bytes, halfwords (16-bit), words (32-bit) or floating-point values. The same like above applies, you use the prefixes '0x', '\$' for hexadecimal, '#' for decimal and '%' for binary. (Again: We later explain this in ... ) However, floating-point values should be represented in decimal.

Ex.:

```
.float #25.668  
.byte 0x68, 0x66, 0x4A, 0x3C, 0x22  
.halfword 0x6C2A, 0x366A  
.word 0x80361482, 0x803A64D1
```

As you can see, you can also put more bytes, floats, halfwords or words by separating each of them by commas. Remember to use the .align directive to re-align in case you write MIPS ASM code after one or more of these directives or else your MIPS instructions are not written to a 4 byte boundary.

## **2.3 .align alignment, (optional) fill**

This directive is also important in case your code isn't properly aligned anymore, for example, after putting in amount of bytes which are not divisible by 4. This would cause following instructions not be properly put to a 4 byte boundary, resulting in a different instruction that you originally wanted. I'm going a bit into more detail in this, because this is important to know.

Ex.:

```
.org 0x7E4E00  
.byte 0x24, 0x5A, 0x26, 0x66, 0x47, 0x36  
LUI T0, 0x8034  
ADD T0, T1, T2  
ADDIU T0, T1, 0x366A
```

If we now assemble the above and look up the code in a disassembler of your choice (CajeASM gets one soon)

```
0x007E4E00: 24 5A 26 66 ADDIU K0, V0, $2666
0x007E4E04: 47 36 3C 08 ? (COP 1)
0x007E4E08: 80 34 01 2A LB S4, $012A (AT)
0x007E4E0C: 40 20 25 28 ? (COP 0)
0x007E4E10: 36 6A FF FF ORI T2, S3, $FFFF
```

It properly inserted the bytes, but look what happened with our MIPS instructions. Each instruction is 4 bytes long. We inserted 6 bytes. That are 2 bytes out of the normal range. The solution is our directive: `.align`. With the help of this directive we can force the assembler to keep the following code in a byte boundary of your choice. So, we have to align our code to a 2 byte boundary. Here:



```
.org 0x7E4E00
.byte 0x24, 0x5A, 0x26, 0x66, 0x47, 0x36
.align #2
LUI T0, 0x8034
ADD T0, T1, T2
ADDIU T0, T1, 0x366A
```

And if we now look at our output:

```
0x007E4E00: 24 5A 24 5A ADDIU K0, V0, $245A
0x007E4E04: 26 66 47 36 ADDIU A2, S3, $4736
0x007E4E08: 3C 08 80 34 LUI T0, $8034
0x007E4E0C: 01 2A 40 20 ADD T0, T1, T2
0x007E4E10: 25 28 36 6A ADDIU T0, T1, $366A
```

Cool, isn't it? Now our code is properly aligned again by simply forcing the current position of our assembler to be a multiply of two.

Optionally you can also fill the skipped bytes, by specifying the 2<sup>nd</sup> operand in `.align`:

**.org 0x7E4E00**  
**.byte 0x24, 0x5A, 0x26, 0x66, 0x47, 0x36**  
**.align #2, 0xFF**  
**LUI T0, 0x8034**  
**ADD T0, T1, T2**  
**ADDIU T0, T1, 0x366A**

And we get:

0x007E4E00:	FF 5A 24 5A	SD K0, \$245A (K0)
0x007E4E04:	26 66 47 36	ADDIU A2, S3, \$4736
0x007E4E08:	3C 08 80 34	LUI T0, \$8034
0x007E4E0C:	01 2A 40 20	ADD T0, T1, T2
0x007E4E10:	25 28 36 6A	ADDIU T0, T1, \$366A

(Look the first byte at 0x007E4E00. It's 0xFF)

## **2.4 .skip size, (optional) fill**

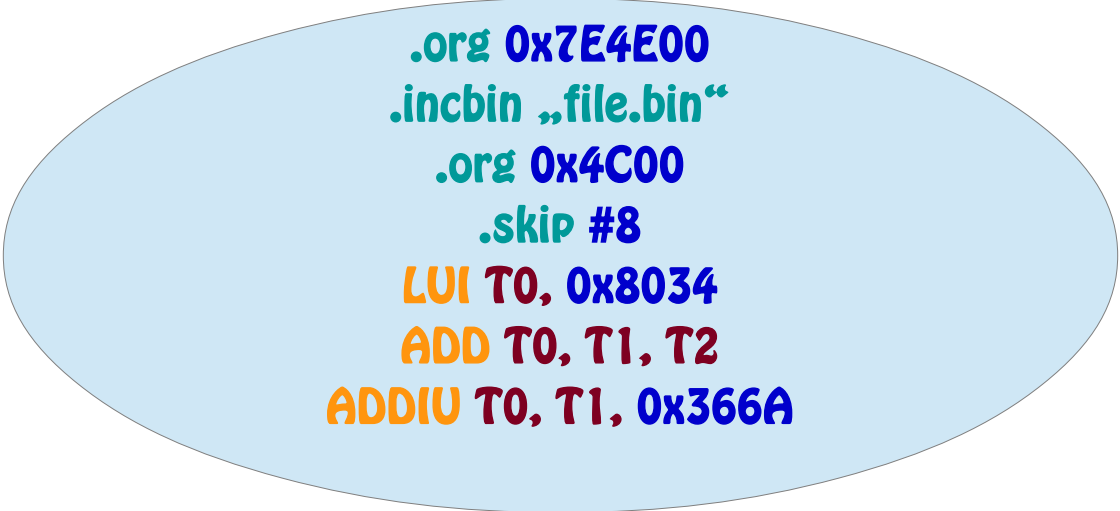
This directive simply let's you skip bytes and optionally you can fill the skipped bytes with 'fill'.

**.org 0x7E4E00**  
**.skip #8**  
**LUI T0, 0x8034**  
**ADD T0, T1, T2**  
**ADDIU T0, T1, 0x366A**

We skip 8 bytes and then write our instructions. I think not any further explanation is needed. You can optionally fill the skipped bytes with 'fill' argument.

## **2.5 .incbin "filename.bin"**

This directive tells the assembler to include a binary file into your code. You can use the .org directive and then use the .incbin directive and your binary file is put there.



```
.org 0x7E4E00  
.incbin „file.bin“  
.org 0x4C00  
.skip #8  
LUI T0, 0x8034  
ADD T0, T1, T2  
ADDIU T0, T1, 0x366A
```


This would put the binary file into offset 0x7E4E00.

## **2.6 .incAsm/.inc/.include "filename.asm"**

This directive is another include directive, but let's you include ASM files instead. It's basically the same like above, you simply write the path to your ASM file. Just saying, that you can either write .incAsm, .inc, .include. All do the same.

Ex.:

TestASM.asm:



```
.org 0x7E4E00  
LUI T0, 0x8034  
ADD T0, T1, T2  
ADDIU T0, T1, 0x366A
```

Main.asm:

```
.incasm „TestASM.asm“  
    .org 0x861C0  
    ORI T0, R0, 0x3645  
    ORI T1, R0, 0x3314  
    BGT T0, T1, 0x00003680  
    NOP
```

This would first assemble the included file to offset **0x7E4E00** and then after this is assembled it will return to the normal routine and assemble the rest at **0x861C0**. Yes, you can also include more than one files and you can include files in an included file.

### **2.7 hex { hex values }**

The next directive is similar to the numeric value directives. In this case however we can insert an infinite number of hex values without a specified length like word, halfword or byte. (In SM64 it's useful for behavior scripts, mostly also because you just can copy it out of VL-Tone's docs and put it right into this directive)

```
    .org 0x21CCDC  
    hex { 0C 00 00 00 80 2C B1 C0 }
```

This would put this array of bytes into offset **0x21CCDC**.



## **2.8 .ascii/.asciiiz**

The last directive we're gonna look at are these two. As you can kinda guess, it's the directive, which let's you insert ASCII string text into the ROM. The major difference between .ascii and .asciiiz is, that .asciiiz is a zero-terminated string. (After the string is written, a NULL (**0x00**) is added after the string) It's recommended to use .asciiiz, as most N64 games read ASCII text 'till the next null byte.



```
.org 0x861C8  
.asciiiz „Hello World“
```

Would put ASCII string “Hello World” to offset **0x861C8**.

## **3. Labels & Defines**

Our next subject is: “Labels & Defines”. This one is probably more interesting to you, as this is again more code-specific. It allows you some benefits and eventually makes your asm programming life easier. So, continue and read carefully.

### **3.1 Labels**

What is a “Label” you ask. Well, a label is nothing else than a name of the current location the label was placed at. If my current position is at 0x60004 and I place a label called “MyLabel”, then this label would've the offset 0x60004. Now, we could branch with branch instructions to that label. The benefit? Well, you no longer have to calculate the ROM offset you want to branch to. Instead, the assembler settles all this stuff and you just need to declare a label and then you can branch to it.

I'm showing a small example to give you clear picture of this. First, let's do it the “hard way”:

<b>ADDIU</b> SP, SP, 0xFFE8	; 0x00
<b>SW</b> RA, 0x14(SP)	; 0x04
<b>ORI</b> T0, R0, 0x000A	; 0x08
<b>ORI</b> T1, R0, 0x000C	; 0x0C
<b>BEQ</b> T0, T1, 0x00000020	; 0x10
<b>NOP</b>	; 0x14
<b>BNE</b> T0, T1, 0x00000028	; 0x18
<b>NOP</b>	; 0x1C
<b>LUI</b> T0, 0x8034	; 0x20
<b>SH</b> T1, 0xB218(T0)	; 0x24
<b>LW</b> RA, 0x14(SP)	; 0x28
<b>JR</b> RA	; 0x2C
<b>ADDIU</b> SP, SP, 0x18	; 0x30

Please take a look at BEQ and BNE. Each instruction is 4 bytes long, so we just have to count each 4 bytes to get the offset we want to branch to. It's simple. But then on the other side it's really annoying. Imagine if you just forgot an instruction and then you have to always add 4 again. Or what if you didn't count right? All these problems are solved if we use labels. With labels we don't have to count or even think about the offset.

The Easy Way (and most recommended way):

<b>ADDIU</b> SP, SP, 0xFFE8	; 0x00
<b>SW</b> RA, 0x14(SP)	; 0x04
<b>ORI</b> T0, R0, 0x000A	; 0x08
<b>ORI</b> T1, R0, 0x000C	; 0x0C
<b>BEQ</b> T0, T1, IfTrue	; 0x10
<b>NOP</b>	; 0x14
<b>BNE</b> T0, T1, Exit	; 0x18
<b>NOP</b>	; 0x1C

<b>IfTrue:</b>	
<b>LUI</b> T0, 0x8034	; 0x20
<b>SH</b> T1, 0xB218(T0)	; 0x24

<b>Exit:</b>	
<b>LW</b> RA, 0x14(SP)	; 0x28
<b>JR</b> RA	; 0x2C
<b>ADDIU</b> SP, SP, 0x18	; 0x30

You see now how simple it is? The assembler automatically calculates the position of where the label is placed at and converts it to the real offset. That will make things definitely easier! Plus, labels can also give a better overview what the below code is supposed to do. Like label "Exit" we know that this part of code is for exiting the current subroutine.

Labels work in all branch instructions and also work in jump instructions, if it's really needed.

### **3.2 Defines**

The next thing we come to are “Defines” or “Variables”. They are recommended for overview, structure and allowing (in case you release your ASM code) let others modify it. Like for example, people would just have to change a define which is declaring the costs for some item instead of looking through the code and finding that part.

Defines are made by writing

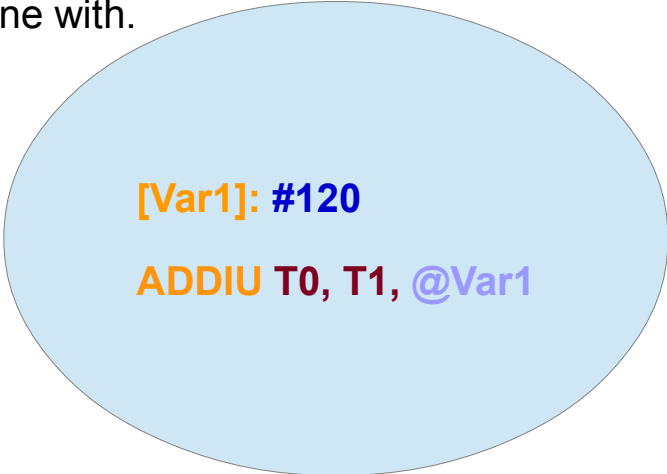
**[NameHere]:** (\$, 0x, # or %) value

(\$, 0x = Hexadecimal; # = Decimal, % = Binary)

If you use a define in an immediate instruction, you would call it by writing:

**@NameHere**

The assembler then automatically replaces the define with the value you declared the define with.



**[Var1]: #120**  
**ADDIU T0, T1, @Var1**

The assembler recognizes that “Var1” is decimal 120 and this is what the assembler writes:



**ADDIU T0, T1, 0x78**

(0x78 = 120 (decimal))

This also works with hexadecimal and binary.

## **4. Additional Information**

So, we finally came to the end of my CajeASM manual. This last chapter will give you some important additional information relating to labels, defines and hexadecimal, decimal and binary values. You should consider reading this one too, as it can be also helpful. But on this part, I'm going to say goodbye already and have fun coding!

### **4.1 About Upper/Lower Half**

This part is pretty important for defines. CajeASM is able to recognize 32-bit defines and splits them up into upper half and lower half. So, if you write a LUI and ORI instruction with the define, it will take the upper half into LUI and the lower half into ORI. The same for all loading/storing instructions and other lower half instructions. To give a small example:



**[Var1]: 0x8034B218**

**LUI T0, @Var1**

**LH T0, @Var1(T0)**

Is translated to:



**[Var1]: 0x8034B218**

**LUI T0, 0x8034**

**LH T0, 0xB218(T0)**

You see, this can be useful for anything you can imagine.

## **4.2 Decimal, Hexadecimal, Binary values**

As you saw throughout the whole manual, we've made use of different value types. The usual, most used type in ROM Hacking generally is obviously hexadecimal. In CajeASM hexadecimal values are always prefixed with "0x" or "\$". Personally I use "0x" prefix for values and "\$" for addresses. But as I said, both are the same and indicate that the following value is a hexadecimal value. Then there are decimal values, which are prefixed with "#". CajeASM automatically converts them to their hexadecimal representatives. Then, at last, there are binary values, which are prefixed with "%". I guess there's nothing more to explain it, but remember to always check that you're using the correct format or else CajeASM crashes.

### **Syntax:**

Binary: %value

Decimal: #value

Hexadecimal: 0xValue or \$Value

## **4.3 Comments**

CajeASM allows you to put comments in your code, which are ignored by the assembler. Commenting your code is always a good idea and might give others, who use your code, an overview and eventual documentation of what exactly the code does.

There are two types of comments: Line comments and block comments. Line comments are only ignored until the next line, while block comments are ignored in a block 'till the symbol follows which ends the comment.

For line comments you use the symbol ; or //

**[Var1]: 0x8034B218**

**LUI T0, 0x8034 // Everything is ignored, 'till the next line.**

**LH T0, 0xB218(T0)**

And then there are block comments, where everything between `/* ... */` is ignored.

```
[Var1]: 0x8034B218
```

```
LUI T0, 0x8034 /* Everything is ignored,  
'till the block end.  
LH T0, 0xB218(T0)*/
```

That's it basically. Always be sure to document your code.