MIPS R4300i Assembly Tutorial
Written by:
Tarek701

(The Final Revision of the Tutorial)

Preface

I congratulate you, that you're really going to do this. To sum it up, this is an ASM Tutorial written by me, Tarek701. The goal of this tutorial is to teach N64 Hackers how to code in MIPS Assembly, successfully assemble it to the game and make big use of it. Before, I had to individually explain everyone how “this” or “that” works, which, after a certain amount of time, started to annoy me and so my decision to write a MIPS ASM Tutorial became truth. And looking from my own side, I really think that explaining each stuff “generally” in a document works way better for me than explaining it individually for everyone. Of course, this won't mean that I'm not going to explain it as easy as possible. Still, you should be able to have some requirements before we can start. I'm explaining them later.

I asked myself, whether I keep this tutorial game-specific (in my case, SM64 Hacking) or not. After a few struggles, I've finally decided to develop the tutorial on SM64. (Super Mario 64) But this doesn't mean that this tutorial couldn't serve any other purpose than SM64 Hacking, such like Zelda64 Hacking or even more. The main topic is still going to be MIPS Assembly, while I just use SM64 to show some practical examples. As you know, theory is important, but practice is the most important thing. What's the point of reading through all this tutorial without doing anything practical? So, expect some theory-parts and then a practical test in-game.

For those, who don't know who I am, my name is Tarek701 and I'm active on SMWCentral.net (from where you most likely downloaded my tutorial) and YouTube. I'm a staff member of the SM64 Hacking Forums, which can be found here. I'm the developer of the current modern MIPS Assembler “CajeASM” which offers a lot of nice features and is going to be used in my tutorial. It can be found here. I guess you should use it too, as it's currently the only legit MIPS Assembler out there which offers that much of features.

Now, after you know me now, I'm going to explain a few “document-specific” things. This is for structuring purposes.

Importance & Optionality:
If you see a chapter marked as: “!!!!“ → This means the chapter is important and a “must-have” knowledge to continue.

If you see a chapter marked as: “****” → This means that the chapter is optional and may advance you in MIPS Assembly knowledge, but isn't required to continue in the tutorial.

If there's no symbol, then it's an usual chapter. This doesn't mean however, that the chapter isn't important. It just isn't one of those “must-have” basics, as the chapter either makes practical use of the stuff you've learned before or is a part of the tutorial itself, which you probably want to do, lol.

Levels & Exercises
If you see a chapter marked as: “EX” → This means that this is an exercise. Mostly those chapters are also marked with the four exclamation marks. Basically, this chapter is testing your skills you've learned so far in the tutorial.

Exercises are marked with Levels, showing the severity of the following test. However, this may depend on each person. So, this is just an objective rating of severity of the test. So, don't demotivate yourself just because the Level is Hard.

LV1 → The test is easy and most likely only asks for some theoretical questions unrelated to MIPS Assembly. If LV1.5 then you get ASM related questions, which are a bit harder.

LV2 → The test is middle and is a practical test related to MIPS Assembly. LV2 Exercises most likely want you to code something in MIPS Assembly.

LV3 → This test is harder, most likely mixed questions about MIPS Assembly and wants you to code something more big.

Many special thanks to the ASM MasterRace who helped me to learn MIPS Assembly, especially
Kazeshin (Kaze Emanuar), Skelux, Xavior, LevelUp(from HF, cool guy) and shyguy

Table Of Contents

Basics:

MIPS Assembly:

Chapter 1: Introduction into MIPS ASM !!!!

Sitting here and thinking right now, what is ASM? ASM stands for **AS**se**M**bly. MIPS is another acronym standing for: “**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages”. So, what makes MIPS so special? As the name implies, MIPS does not make use of locking pipeline stages.

Usually, processors are processing commands in steps through a so-called Pipeline, so it's able to have several instructions in different steps of handling coincident in the CPU. Now, if a subsequent instruction is instructed to a preceding instruction, the instruction will eventually have to be interrupted until it's available. This is done by “Locks”. So, in short: A pipeline lock is nothing else than a simple interrupt to a subsequent instruction that is instructed to a preceding instruction and this stays until availability is confirmed.

MIPS is now special, because it denies those locks and instead wants a corresponding action from either the ASM Programmer (me and you) or from the compiler like sorting or adding a NOP instruction. (NOP = No Operation. Basically, it does nothing) Thanks to this the architecture can be kept simple and that's what I personally like on MIPS. Without this simplicity done through the declination of pipeline locks, MIPS would've been way harder.

I know, that you may have not understood everything but I tried my best to explain it as a easy as possible. If you didn't understand this, don't be sad. Just be aware of that, MIPS is really easy to learn if we talk about the actual Assembly Code.

With MIPS ASM you can code all kind of stuff for N64 Games (such as custom features for SM64). Assembly is a 2nd generation programming language and is therefore “low-level” when compared to higher programming languages like C++, Java or C#. ASM is readable machine code, which is later translated into real machine code.

To show off an example, look here:
A “specific” order of bits adds two values together, like:

10000 1001 00001 00001

(10000 would mean: do Add and 1001 means some “container” we want to store the result, while 00001 and 00001 are our operands we want to add)

So, basically the above wouldn't do anything else than Container = 1 + 1 and store the result in “Container”. So, Container would contain the result 00010 (10 = 2 in decimal)

Now imagine having thousands of those binary values. You can't imagine it, it would get complicated sooner or later. So, people decided to write “Assemblers” which let you write human readable code which is then translated later (once it's assembled) to something like our above binary order.

Ex.:
We do the above example. Some assembler could allow us to write:
ADD Container, 1, 4

Which would be translated to:
10000 1001 00001 00100

(100 in binary is 4 in decimal, by the way). Cool stuff, right?

Chapter 2: Hexadecimal, the Base 16 Numeral System !!!!

To program in MIPS ASM, you will have to learn the basics of the so-called “Hexadecimal” system. It's often referred just to as “Hex” and is another counting system, similar to counting systems like decimal, which is used everyday when we buy stuff, when I count my potatoes (lel), etc. The only difference between Hexadecimal and Decimal is, that the hexadecimal system consists of additional 6 digits per place value, while the decimal system only has 10 digits (0-9). This can be illustrated in a table and may help you to understand the difference.

Decimal	Hexadecimal
0	0x0
1	0x1
2	0x2
3	0x3
4	0x4
5	0x5
6	0x6
7	0x7
8	0x8
9	0x9
10	0xA
11	0xB
12	0xC
13	0xD
14	0xE
15	0xF
16	0x10

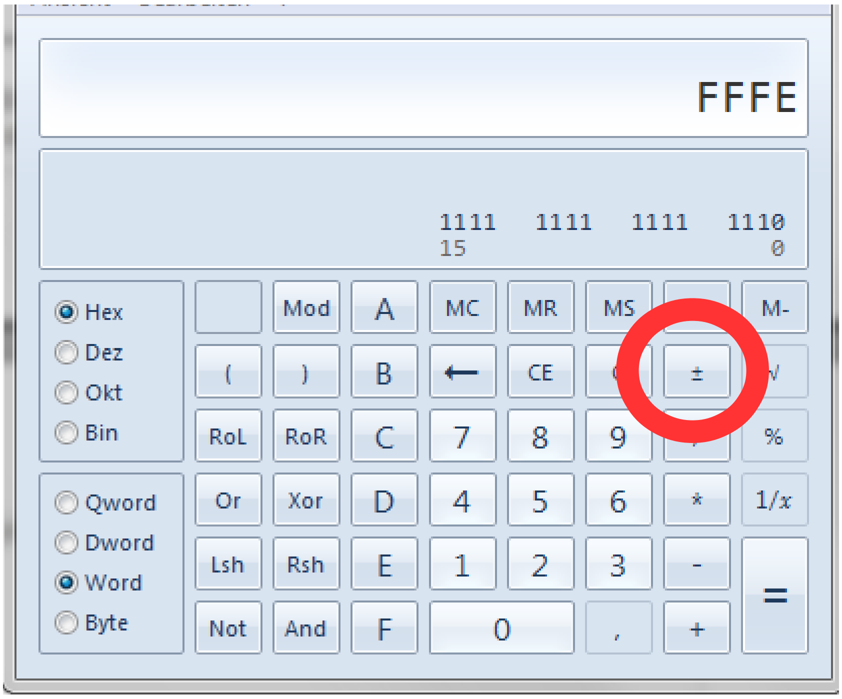
And the numbers will continue, 11, 12, 13, 14, 15, …., 1A, 1B, 1C, 20, …, and so on.

To not confuse hex numbers with decimal numbers, hex numbers are often prefixed with 0x, $ or suffixed with h. (ex.: 0x25, $25 or 25h) CajeASM (my MIPS Assembler) allows you to use “0x” and “$” for Hex numbers, while you use “#” for decimal and “%” for binary values.

In MIPS assembly there's a so-called “Negative Rule” meaning that values which are greater than 0x7FFF are considered as **negative** numbers, starting from -32768, and counting down as the hex number increases.

Decimal	Hexadecimal
32767	0x7FFF
-32768	0x8000
-32767	0x8001
[...]	[...]
-2	0xFFFE
-1	0xFFFF

Now, you probably want to know how to quickly calculate a positive number out of a negative number or the opposite. Well, that's quite easy. Basically, you just open up your Windows Calculator (or any calculator of your choice):



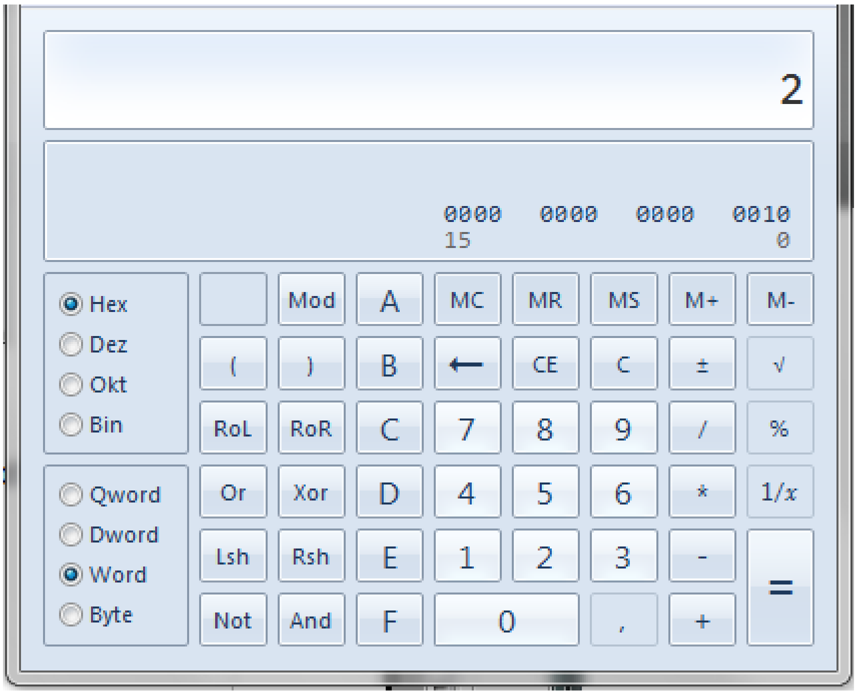
```
FFFE

1111   1111   1111   1110
15                      0
```

● Hex
○ Dez
○ Okt
○ Bin

○ Qword
○ Dword
● Word
○ Byte

(0xFFFE would be -2 in decimal)

Now, you just press the +- Button and your value is automatically converted to a positive value:



```
2

0000   0000   0000   0010
15                      0
```

● Hex
○ Dez
○ Okt
○ Bin

○ Qword
○ Dword
● Word
○ Byte

Remember, that you always have to select “Word” View, as the negative rule is different on Dword and Qword.

Chapter 3: Binary, the Base 2 Numeral System

Now, after we've learned about hexadecimal, we will now learn one another numeral system, the “Binary” system. As you may know already, binary only allows two digits which are 1 and 0. (They're often called true and false in programming)

If you open up a HexEditor you might have noticed that Hex numbers are always displayed as 2 digit numbers like “22 6D 8A CC 29”. Basically, this was made to keep the hex values “compatible” to the binary system. A would represent a “nibble” in binary aka 4-bits. (A = 1000 in binary) Two hexadecimal digits represent therefore a “byte” aka 8-bits, which is more comfortable for us as every PC in general starts with the “byte” count and doesn't count the nibbles.

Learning binary can be a bit time consuming and not easy, but actually counting in binary is a really easy thing. Basically, you just need to imagine that after 1, the 1 will move one to left.

Ex.:
0000
000**1**
00**1**0
00**1**1
0**1**00
0**1**01
0**1**10
0**1**11
1000

So, if we have 0000, 1 goes to the first zero: 0001. Now, first 1 is filled and the 1 moves now one to left: 0010. Now, it won't move as we first have to fill that zero on the right, resulting in: 0011. Now, our 1 moves one to left and zero's all other digits behind the 1: 0100. And this goes so on... It's actually really easy.

Where is binary useful? Well, it can be quite useful if you give a hex number multiple purposes. That way, you can save some bytes in ROM or RAM. As an example, let's indicate that bit 9 sets your level to “day” or “night”, bit 8 that a specific object appears or not. I'm using a lot the word “or”. That's because bits can have only two values, so the effect can be treated as “either **this** or **that**”. The level is either day or night, there's no inbetween or anything else. Such values are also called “flags”.

Here's a little table to give you general overview of counting in hex and binary:

Decimal	**Hexadecimal**	**Binary**
00	$00	%0000 0000
01	$01	%0000 0001
02	$02	%0000 0010
03	$03	%0000 0011
04	$04	%0000 0100
05	$05	%0000 0101
06	$06	%0000 0110
07	$07	%0000 0111
08	$08	%0000 1000
09	$09	%0000 1001
10	$0A	%0000 1010
11	$0B	%0000 1011
12	$0C	%0000 1100
13	$0D	%0000 1101
14	$0E	%0000 1110
15	$0F	%0000 1111
16	$10	%0001 0000

As I said earlier, “%” prefix indicates a binary value and is also used in CajeASM.

Chapter 4: ROM, RAM, Save Chips and Addresses !!!!

You may have heard those terms already, like “ROM” and “RAM”, but probably never knew the difference between them. But it's really important to know the difference, as you're later going to modify both ROM and RAM in-game.

ROM means “Read-Only Memory” and can't be modified by using ASM. Basically, it's the game itself containing all the graphics, music, ASM data, etc. which emulators read, load and run. In short: It's the .z64/.v64/.n64 file.

RAM is “Random Access Memory” and a bit different than **ROM**. **RAM** is mostly also just referred to as “Memory”, while (in my eyes) RAM should be the preferred term. The **RAM** consists of a bunch of dynamic variables (like (in our case) Mario’s current coin amount, Mario’s current lives, etc.) used by the ROM. That's actually what we modify through ASM coding.

Now, there are so-called “Addresses” or also referred to as “Offsets”. These are simply a order of numbers which “point” to a specific hex number An example:

```
0x0000000: 25 69 DD 55
0x0000004: AA 22 D5 66
```

The hex value 0x25 would be 0x0000000, 0x69 would be 0x0000001, 0xDD would be 0x0000002, 0x55 would be 0x0000003, and 0xAA (the next row) would be 0x0000004.

Now, a “RAM Address” is simply a specific place in the RAM, which can't be accessed through the ROM. (Except when we calculate the ROM Offset out of the RAM Address) For example, if 0x8034B262 contains the current coin amount, like 0x000A (10 coins in decimal), then 0x8034B262 contains the hex value 0x000A (halfword). It's really simple.

RAM is not “saved”. Once the N64 resets, the **RAM** also resets (On the real hardware it's put to random values. Emulators often do however specific values, like 0xFF or 0x55) and that's basically it. Really simple.

The N64 also consists of so-called “Save Chips” which save your game play to the game cartridge. The N64 knows five types of data saving:

EEPROM (512 Bytes), **4x EEPROM** (2KBytes), **SRAM** (32KBytes), **FlashRAM** (128KBytes) and the **Controller Pak** (256KBytes). All save chips, except for the controller Pak, are built into individual game cartridges. The Controller Pak instead is plugged into the N64 Controller.

Those are also “RAMs” (if you want to call them like that) and are located in several different addresses. In our case, the **EEPROM** ranges in RAM from **0x80207700** up to **0x80207900.** However, there's more space after it and therefore allowing to allocate even more **EEPROM**. I'm later explaining how to work with **EEPROM**. Generally said, the EEPROM and the other save chips are behaving like the RAM; You can store anything and load anything from it except that the values do not get cleared when the N64 resets. On Emulators, the EEPROM is stored in a separate file, known as .eep files (in PJ64). (They're mostly located in: **AppData/Local/VirtualStore/Program Files (x86)/Project 64 1.6/Save**)

LV1 Test:

~~**Q:** What does MIPS stand for?~~

Q: What does ASM stand for and what does it mean?

Q: What's an “Assembler”?

Q: What programming generation is ASM? Is it high or low-level when compared to for example, Java?

Q: What is the hexadecimal system? What are the differences between decimal and hex?

Q: What does 0xFF mean in decimal?

Q: What is the “negative rule”? And why is it important?

Q: What happens when the hex value is over 0x7FFF? What decimal value would be this (in 16-bit range please)

Q: What is the base 2 digit numeral system?

Q: Where (for example) could you use the base 2 system?

Q: What is ROM and RAM, and where's the difference between them?

Q: What's an “address”?

Q: What are “save chips” (especially the EEPROM)?

A little theoretical test for you, before we continue. Try to be honest with yourself and don't cheat. If you're not able to solve it, you should re-read everything again and re-try. This is the pure basic stuff you should have knowledge about. Later it's really important to remember it.

Chapter 5: The General Purpose Registers (GPR) !!!!

Now, we're finally moving on. If you went through all this until here, I congratulate you. We can now finally start with MIPS Assembly. Before we actually "code" however, we still have yet to learn a "Basic" for the MIPS Assembly itself: "Registers".

In the very beginning of my tutorial I showed you an example how ASM code could look like. There I used a so-called: "Container" to store my result. Basically, that's what a "register" in general does. It's a container, which contains values. You can load/store values from it.

The MIPS CPU consists of exactly 32 so-called: "General Purpose Registers" (GPR) registers and are 32-bit wide (**XXXX** **YYYY**). Then there are another "special" registers, regarding floating-point operations and are simply called single/double-precision registers.

As the name "General Purpose Registers" implies, I'm going to explain the "general" purpose of the registers, which doesn't mean that the GPRs couldn't serve another purpose.

Don't be confused by the register numbers (R0, R1, R2, …, R31). They're simply the "numeral" expression of the register and it's recommended to always use the register names as this is way more overviewable.

The **zero, R0** register is a hardwired and is permanently zero. This value can't be changed, no matter what you do. It's often used for ORI operations or anything which require the value zero.

The **AT**, **R1** register is reserved by the assembler. It will later get our attention if we start going deeper into branches. But in general, this shouldn't be touched.

The **V0-V1, R2-R3** registers are so-called "return value" registers. As the name says already, these registers are supposed to return values from either a subroutine(= function) or a specific address. Usually the results are 32-bit wide and uses therefore V0 only. If the result is higher than 32-bit (64-bit) then V1 will be used too.

The **A0-A3, R4-R7** registers are the argument registers. Obviously, they're used for a subroutine/function. They're simply said, our parameters for the subroutine we're calling. If we have more than four arguments, then we need to pass them to the stack. As I said before, this is the general purpose. Theoretically you even could use V0 or T0-T7 as arguments.

The **T0-T7, R8-R15** registers are "temporary" registers, used for values which are only used in this subroutine. If you do an subroutine call (JAL) the callee wouldn't have to save these registers.

The **S0-S7, R16-R23** registers are the "saved" registers, which will save the callee. Subroutines can only use this, if this register is pushed on the stack.

The **T8-T9, R24-R25** registers are (again) the temporary registers and are used in addition to T0 - T7 above.

The **K0-K1, R26-R27** registers are reserved by the kernel of the system and reserved for use by the interrupt handler. Do not modify or play around with it. It will crash your emulator.

The following registers are not "directly" containers of values (aka what we understand as registers), but more like "Pointers" to values. Still, they can be used to "refer" to the actual container (aka load and store values), meaning that we have more like indirect registers here:

The **GP, R28** register is the Global Pointer. It points to the middle of the block memory in the static data segment. This would be our RDRAM in N64.

The **SP, R29** register is the Stack Pointer. It points to the top of the stack. We're gonna learn about it more and show you the big use of it. Also, remember that the stack grows from high memory to low memory. This means, if you SP with a positive number you go backwards, if you SP with a negative number you go forward.

The **FP, R30** register is the Frame Pointer. The FP register is a bit similar to SP register, however the FP register does point to what the SP register DID point before. This can be useful if you're not sure anymore where a specific parameter was stored. I explain this later. This register here can also be considered another save register (S8).

The next register is a normal register again, but also special in it's own kind, as it receives it's value through subroutine calls.

The **RA, R31** register is the Return Address register. Once we made a JAL to a subroutine, the RA register will save the address we jumped from and once the subroutine is done we will jump back to the usual routine. Sometimes you will have to push the RA register on the stack, because if you do a **JAL** in another subroutine then the old return address is lost.

Now, there are also some quite special registers such like the 32 single-precision FP registers:
F0-F31 are used for floating-point operations. We're getting later confronted with them.

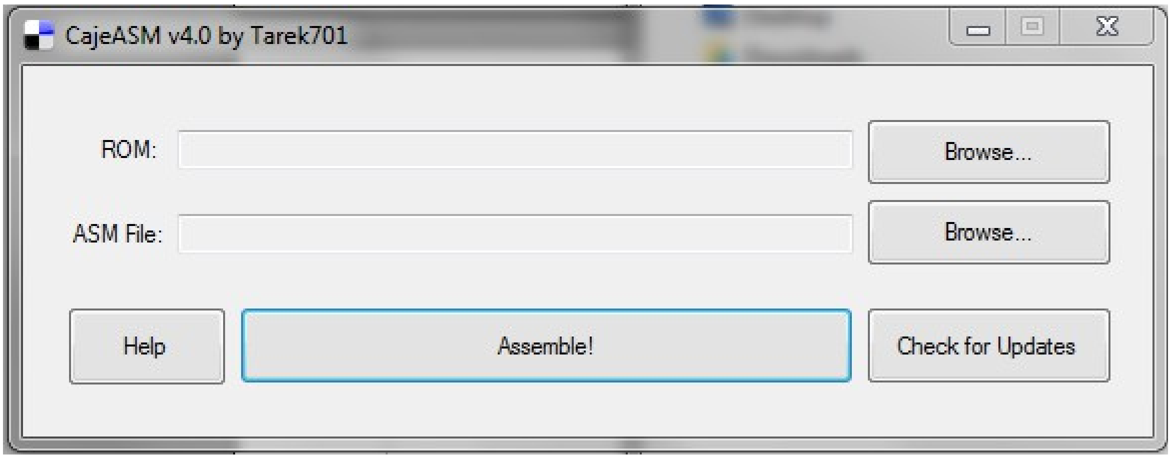
There are also another two special registers, called **Lo** and **Hi**, which store the results of multiplication and division operations. They can't be addressed directly, but the contents can be accessed by special instructions **MFHI** ("move from **Hi**") and **MFLO** ("move from **Lo**"). We also learn more about them later.

Summary:

The general purpose of a register is to be filled with values, to be used in arithmetic operations (+, -, * and /), to be load/store from/to an address. For example, you load a value from an address to a register, progress the register content and later store the calculated value back to the address.

Chapter 6: First View into MIPS Assembly

So, after we've learned about the registers, we can finally start and write our first MIPS ASM code! For this, you will need an actual MIPS Assembler. Download CajeASM, the link is here. After you're done with downloading, extract CajeASM in a folder of your choice. If you open up CajeASM vX.XX.exe (the GUI) you should see this:



It's so easy, that even a five year old could use it! It's built up, so you intuitively know what to do. Just select your target ROM and your ASM file and press assemble. That's it!

And now, let's stop waiting and finally start writing a simple MIPS ASM code. Just learn and watch:

```
.org 0x861C0
ADDIU SP, SP, 0xFFE8
SW RA, 0x14(SP)

LUI A0, 0x240C
ORI A0, A0, 0x0001
JAL 0x802CA190
NOP

LW RA, 0x14(SP)
JR RA
ADDIU SP, SP, 0x18
```

Paste this short code into the notepad of your choice and save it as “whatever.asm”. Now try to assemble your code to a clean SM64 ROM. Start your ROM... What will you see... umm hear? Yes, “Here We Go” sounds playing non-stop. So, that's what the code does. It simply plays the “Here We Go” sound which is looped (but not through the asm code, but by the behavior itself! Just saying). Also, “.org” means that our code should be put in a specific address.

Now, let me explain Step-For-Step:

First, we do

```
ADDIU SP, SP, 0xFFE8
```

Which (if you know already) has something to do with the Stack. Basically, we allocate memory by subtracting 24 from the stack. (Remember, 0xFFE8 means -24!) So, if we want to “allocate” memory ready to be used, we always “subtract” from the stack. When we we want to fill the memory again, we simply add back 24 to the stack. You see, it's really simple. Now, the instruction itself just says:

SP + (-24) and store result in SP. Basically, the SP will now point -24 backwards, which is “forward” in our case as the Stack grows from “higher” to “low” memory. So, remember this! ADDIU also means Add Immediate Unsigned. However, the values are not really unsigned (as you can see). Difference between ADDI and ADDIU is, that ADDIU will not error when our value goes over 0x7FFF.

SW RA, 0x14(SP)

This instruction stores the current return address (in RA) to the stack, on SP + 0x14 aka 20-bits. A return address is, as I explained before, the address of the routine we came from. The purpose of storing the return address to the stack simply is, that once we jump to another subroutine the old return address of the current routine is lost and thanks to pushing the return address to the stack, we can later get it back again and jump to it, back to our main routine. The instruction itself does:

Store Word, to SP + 0x14 (lower half).

I'm explaining this later more in detail.

LUI A0, 0x240C
ORI A0, A0, 0x0001
JAL 0x802CA190
NOP

This code snippet is actually our “first” subroutine code. **LUI** and **ORI** load the “argument” we pass to the subroutine, which is the argument that plays the “Here We Go” sound. Let me explain the instructions more clearly:

LUI = Load Upper Immediate.

LUI basically loads the “upper” half of an address or value, in this case it simply loads the value to the first four digits from left:

LUI A0, 0x240C

A0 = **0x240C0000**

ORI A0, A0, 0x0001

ORI = OR Immediate

ORI is a simple OR expression. Basically, in expressions you say “this or that” indicating that you have an option between two things. Whatever you take, both are legit.

1 or 0 = 1
0 or 1 = 1
0 or 0 = 0
1 or 1 = 1

So, the output is only 1, if one of the options is 1. In a hexadecimal case everything over 0x00 would count as 1.
So, our ORI will not change the value “0x0001” and it will be still the very same. It even works with 0x011D or 0x144A or whatever you want. As long as the last four digits are zero, this won't be a problem. And that's what ORI basically does.

ORI shifts the value to the LAST four digits from left, resulting in:

```
A0 = 0x240C0001
```

As you see now, 0x0001 is now standing behind the 0x240C. So, remember: **LUI** does first four from left, **ORI** does last four from left.

Now instructive, LUI is:

```
LUI rt, imm
```

(rt being the register where the “imm” (16-bit value, immediate value) is load **to**)

```
ORI rd, rt, imm
```

(rd being the destination where the result of the OR operation is stored and rt being the operand and imm being the operand which both are OR'd)

Next, there's the JAL instruction:

```
JAL 0x802CA190
```

Basically, this is the “call” for the subroutine, which will execute the PlaySound function. As you may have realized, the subroutine just takes one argument (A0). After we passed the argument to the subroutine, the routine takes it up and will execute it.

JAL means Jump and Link.

```
LW RA, 0x14(SP)
JR RA
ADDIU SP, SP, 0x18
```

The last three instructions are the “exit” out of our main routine. Once the subroutine is done, our old return address is lost. As we pushed the return address to the stack however, we can simply load it from the stack again by using LW (the opposite to SW = Store Word, LW = Load Word). Now we don't “yet” jump back to our return address. This is really important, so listen: Each jump/branch instruction consists of a “delay slot”. This delay slot is executed BEFORE the actual jump instruction. This means, that **JR RA** is executed **after ADDIU SP, SP, 0x18.**

For usually, if there's nothing important to put into the delay slot, it's NOPed:

```
JR RA
NOP
```

So, basically the addiu instruction simply adds back 24 to the stack and then jumps back to the return address, the address we came from.

As the behavior is “looped” the code is executed over and over again and therefore isn't my doing. Of course, you theoretically could built in your own loops, but this is gonna be explained later.

I'm not angry or anything if you didn't understand some parts. This was more like a “first view into MIPS Assembly” and we get more concrete with the following chapters, which will sooner or later bring you to realize what each code does from alone. The explanations here were also very rogue.

Chapter 7: Instruction Formats

Maybe you're still confused on how the instruction “format” is like. Currently you just saw instructions having only two operands, while some had three. I'm going to explain you now in much detail, what instruction formats do exist in MIPS. Trust me, they're really easy to learn.

Each instruction consists of a format, a “syntax”. There are exceptions (i.e LUI) but in general it applies to every instruction we use(d) here. So, let's start:

R Instructions

R Instructions are used, when all contents used by the instruction are located in the registers. Basically, R Instructions are the “non-value” instructions. They're completely register-specific. That's why it's called R Instructions, standing for “R = Register”.

The syntax is as following:

Opcode RD, RS, RT

RD = Destination Register
RS = Source Register
RT = Target Register.

The “ADD” instruction is a really good example for this:
ADD T0, T1, T2

The above instruction will calculate the content of **T1** and **T2** and store it into **T0**. So, **T1** is our RS, **T2** our RT and **T0** our RD.

In a more mathematical sense, it looks like this:
T0 = **T1** + **T2**

The “full” operation(in binary) can be also represented in a table, which I'm showing here:

Opcode	RS	RT	RD	Shamt	Func
6 Bits	5 Bits	5 Bits	5 Bits	5 Bits	6 Bits

Opcode

This is the instruction. In our table case, this is the machine code representation of the instruction mnemonic (like **ADD**)

RS, RT, RD
These being our registers.

RS = Source Register Operand
RT = Target Register Operand (or 2nd Source Register Operand)
RD = Destination Register Operand (where results are stored)

Shamt/Shift
This is used along with shift and rotate instructions. It's usually zero on classic arithmetic instructions like ADD, SUB, MUL, DIV, etc. Basically, Shamt is the amount by which the source operand RS is rotated or shifted. This field is 5 bits long (6 to 10).

Func
For instructions that share an opcode, the func parameter contains the necessary control codes to differentiate the instructions. Ex.: Opcode 0x00 accesses the ALU, and the func selects which ALU function to use. This field is 6 bits long (0 to 5). (ALU = Arithmetical/Logical Unit, basically anything logical/arithmetical related stuff such like shifting, mathematical instructions, etc.)

Usually, you won't see R Instructions that often. You most likely will work more with the following instruction format.

I Instructions

I instructions are used when an instruction has to operate an immediate (16-bit) value and the content of a register. Immediate Values are only 16-bit long.

The Syntax is as following:

Opcode **RT**, **RS**, **IMM**

IMM = Immediate Value (16-bit Value)

The ADDIU instruction can be taken as an example here:
ADDIU **T0**, **T1**, **0x269D**

The above instruction will calculate the sum of the content of **T1** and the immediate value and store the result in **T0**.

This would be equivalent in maths as:
T0 = **T1** + **0x269D**

Again, the full operation looks like this in a table:

Opcode	RS	RT	Immediate
6 Bits	5 Bits	5 Bits	16 Bits

Immediate
This is our 16-bit immediate value. (0 to 15) This value is used as the offset value in various instructions and depending on the instruction, may be expressed in two's complement.

These kind of instructions are appearing very often, way more often than R Instructions and the following J Instructions.

J Instructions

J Instructions are used when a jump needs to be performed. J Instructions allows the most space for an immediate value, as the addresses are large numbers and 32-bit long.

The full syntax:

Opcode LABEL

The opcode is (once again) our mnemonic for the particular jump instruction and LABEL is our target address we want to jump to. CajeASM allows to use actual addresses or let CajeASM calculate them automatically and instead use “Label Names”.

An example of such a jump is a very simple J opcode, which just does a jump to a target:

J 0x8033D666

This would jump to address: 0x8033D666.

Or (if you defined a label in your ASM code) jump to the label by calling it in your J instruction:

J !LabelName

Which would jump to the !LabelName address. (CajeASM calculates the address automatically).

In a table, this looks like:

Opcode	Target
6 Bits	26 Bits

B Instructions

B Instructions are similar to J Instructions, just that these are, in opposite to J Instructions, conditional jumps while J Instructions are absolute jumps. B stands for Branch, while J stands for Jump. Basically both do the same, but branch is referred to “conditional jumps” while J to usual, direct jumps.

The full syntax:

Opcode RS, RT, LABEL

In this case, RS and RT are the both registers which are compared and LABEL the address/label we jump to, if the condition is true.

An example opcode is BEQ:
BEQ T0, T1, 0x80038A4D

This checks if **T0** is equal to **T1**. If the condition is true, then we branch to **0x80038A4D**.
In a more mathematical sense, it looks like this:
If **T0** == **T1** → **0x80038A4D**

Illustrating this again in a table:

Opcode	**RS**	**RT**	**Branch Target**
6 Bits	5 Bits	5 Bits	16 Bits

If you're wondering about why branch target is 16-bits but the input above is 32-bit, then listen:

Basically, a normal J Instruction is a “direct” jump to an address while a branch is more like a step-by-step jumping. J Jumps are direct, while Branches jump in four byte steps from the address it currently sits on.

Ex.:

```
0x00000000: BEQ T0, T1, 0x0000000C
0x00000004: ADDIU T0, T1, 0x0001
0x00000008: ADD T0, T1, T2
0x0000000C: SUB T0, T1, T2
```

Basically, the BEQ would be translated as:

```
BEQ T0, T1, 0x0002
```

This means that if T0 == T1, we should jump 2(3, as 0xFFFF counts here as -1) forward. So, start: 0x00000000 = -1 (0), 0x00000004 = 0 (1), 0x00000008 = 1 (2), 0x0000000C = 2 (3).

So, branches are nothing else than jump forward in a specific number. As you may have realized already, this works only for a specific area as the range is only 16-bits long. That's why Branches shouldn't be very far away from the instruction.

Now after you've learned some of the Instruction Formats, we can move on and get more concrete to some specific MIPS instructions. In the next chapter, we're going to work with Load/Store Operations and what practical use we can make of them. Be sure that you're able to remember each instruction format. Later, when we come to floating-point operations, we're going to learn two other instruction formats. Until then, this should be enough to know about the instruction formats.

Chapter 8: Load/Store Operations

So, now we can finally get more concrete after we've learned the instruction formats. We've met load/store in Chapter 5 already, when we wrote our first little test code. This time, I get into more detail and try to explain it (but easier and more detailed) again. If we want to load a value from **an address to a register** we make use of so-called: "**Base Addressing**" instructions. **Base Addressing** doesn't mean anything else than that we have a "Base" which contains the upper half of an address and we add a lower half to it and then MIPS automatically loads the value from that base address + lower half into the destination register.

An example, which should be known to you already:

LUI T0, 0x8033   ; Load upper half in T0. T0 = 0x80330000
LW T1, 0x3D54(T0) ; This loads a word value (32-bit) from address 0x80333D54 into T1. T0 is UNCHANGED!

So, what exactly happened here? Well, as you know already **LUI** loads a value to the first four digits from left. We call this the "upper half" of a register. In this moment:

T0 = **0x80330000**

In the next instruction, **LW** simply calculates **0x3D54** + **T0**, resulting in **0x80333D54** and loads the value from that address into **T1**. **T0** isn't changed however! **T0** still would be **0x80330000**! So, basically the instruction loads a value from an address which is calculated by adding the lower half(**0x3D54**) to the content of the base register (**T0**) into the destination register (**T1**).

To show up the differences between a Word, Halfword and a byte, let me show an example:

0x80333D54: 4D 8A 99 5D 2C 66 1A 21

If we now do the above instruction, then **T1** would contain:

T1 = **0x4D8A995D**

Word   → 32-Bit: XX XX XX XX
Halfword → 16-Bit: XX XX
Byte   → 8-Bit: XX

0x36 is a **byte**.

0x3548 is **a halfword** or **two bytes**.

0x3548AA50 is **a word** or **four bytes**.

So, what if we want to load a halfword to T1? Well, then we simply take LH, which let's you load 16-bit values to a destination register:

```
LUI T0, 0x8033        ; Load upper half in T0. T0 = 0x80330000
LH T1, 0x3D54(T0) ; This loads a halfword from address 0x80333D54 into T1. T0 is UNCHANGED!
```

If we take our example again;
0x80333D54: 4D 8A 99 5D 2C 66 1A 21

Result from above instruction:
T1 = **0x00004D8A**

If we want to load a byte, we'll take LB:

```
LUI T0, 0x8033        ; Load upper half in T0. T0 = 0x80330000
LB T1, 0x3D54(T0) ; This loads a byte from address 0x80333D54 into T1. T0 is UNCHANGED!
```

Result from above instruction:
T1 = **0x0000004D**

Pretty simple, isn't it? Now, let's say we want to “store” another value to an address now. Loading is good at all, but what about storing? Well, you can pretty much imagine yourself what the instruction names could be. If we have:

LW = Load Word
LH = Load Halfword
LB = Load Byte

to load values,

Then we also have:
SW = Store Word
SH = Store Halfword
SB = Store Byte

to store values.

Let's say for example, we want to store a byte to our address now. We basically just need a value which we want to store. So, one more instruction and that's basically it:

```
LUI T0, 0x8033         ; Load upper half in T0. T0 = 0x80330000
ORI T1, T1, 0x00BC ; Our byte value is load in T1. T1 = 0x000000BC.
SB T1, 0x3D54(T0)   ; Store value of T1 in address 0x80333D54. T0 is still 0x80330000.
```

As you know already, ORI does an OR and as the lower half (the last four digits from left) are zero, the value **0xBC** will be the same and is shifted there, resulting in **T1** being **0x000000BC**.

Now, the instruction SB says that the byte value IN **T1** should be stored in **0x80333D54**. So, if our example looked like this before:

0x80333D54: 4D 8A 99 5D 2C 66 1A 21

It would now (after the SB instruction) look like this:
0x80333D54: **BC** 8A 99 5D 2C 66 1A 21

This works the same way with **SH** and **SW**. It's really simple. And now it's time to do something more practical! Let's say we want to store 10 coins to Mario's current coin counter. Our code should look like this:

```
.org 0x861C0
ADDIU SP, SP, 0xFFE8
SW RA, 0x14(SP)

LUI T1, 0x8034
ORI T2, R0, #10
SH T2, 0xB262(T1)

LW RA, 0x14(SP)
JR RA
ADDIU SP, SP, 0x18
```

If you look at the middle part, you see that we're doing an ORI. That's our value we want to store. In CajeASM you can write “#” indicating that the value is a decimal number. This makes it easier for you. (It's later translated to hexadecimal as 0x000A) If you now assemble the code and go in-game into some level you will see that your coin counter is now having 10 coins. As Mario's Behavior is looped, you're not able however to receive more coins as the game continuously stores 10 to the coin counter.

So, but now **ATTENTION** guys. The real address of Mario's Coin Counter is NOT **0x8034B262**! It's actually **0x8033B262**! But why did we wrote **0x8034** instead? That's easy, because the lower half value “**0xB262**” is OVER **0x7FFF**. In this case, our address is subtracted by 1. So, if we would write **0x8033B262**, the game would load the value from **0x8032B262**, which is incorrect. So, we obviously have to increase the value by one and write: **0x8034B262**. This subtracts one, resulting in that LH loads the halfword value from **0x8033B262**, which is now correct. **PLEASE REMEMBER THIS RULE**! It's important that you don't forget about it. CajeASM may soon get an integration for this.

Chapter 9: Arithmetic Operations

Now, we're moving to some arithmetic operations. Under this category, this would be Add, Subtract, Multiply and Division. This chapter is gonna be pretty short, as most of this is pretty self-explanatory. We're going to take the code from the last chapter again and play around with the coin counter a bit.

First off, we will add two values together. It's so extremely simple, that you should be able to understand this in no time. **Note:** If you still don't understand the whole code itself, then please read the recent chapters again, especially the parts you didn't understand before. Or else, you're going to fail in the next chapters.

Now, let's try out adding two values together. We load 10 coins and then add 2 to the coin amount, resulting in that our code stores 12 coins to Mario’s current coins. Basically, we just take **ADDI** or **ADDIU** instruction for this. (We take **ADDIU**, as it's commonly used anyway and we don't want to get annoyed with possible overflow exceptions) The operands are obviously the register containing the value we want to store and the 2nd operand being once again the value we want to store. Logically, we want to calculate:

T2 = **T2** + **#2** (decimal)

```
.org 0x861C0
ADDIU SP, SP, 0xFFE8
SW RA, 0x14(SP)

LUI T1, 0x8034
ORI T2, R0, #10
ADDIU T2, T2, #2
SH T2, 0xB262(T1)

LW RA, 0x14(SP)
JR RA
ADDIU SP, SP, 0x18
```

As you can see, it's the very same code except for this part:

```
ADDIU T2, T2, #2
```

This calculates now:

T2 = **T2** + **#2**

The result is stored in **T2**, meaning that **T2** is now **#12** (decimal) and is stored to Mario's current coin counter. See, really simple. That's all what **ADDI**/**ADDIU** does. Adding an immediate value with the content of a register.

Now, we come to subtraction. Actually, a true subtraction does not exist in MIPS and it's done by using again an **ADDIU** instruction but instead of a positive value, a negative value is used. CajeASM provides a **SUBI** and **SUBIU** instruction however, making this less complicated for you.

In reality, you would write **ADDI**/**ADDIU** to subtract some value from a register. If we want to subtract 23 (decimal) from T0 (for example) and save the result to T1 we would write:

```
ADDIU T1, T0, 0xFFE9
```

0xFFE9 being “-23”. As you see, we “add” -23 to **T0**, which is (logically) equivalent of subtracting 23 (decimal) from T0.

As I said, CajeASM provides an instruction for this already, saving your time of calculating the negative number of the positive number and simply translates it to the above instruction later when the code is assembled.

So, the above example can be written with SUBIU like this:

```
SUBIU T1, T0, 0x17
```

(**0x17** = 23 (decimal))

See. That's far more easier. You could also use decimal numbers now (CajeASM does not allow expressions like “#-23”. This is no problem anymore, as **SUBI** let's you write your number like #23 and translates it later to a negative number.) instead of hexadecimal numbers, when subtracting some value or whatever.
With our example, we can now write the following:

```
.org 0x861C0
ADDIU SP, SP, 0xFFE8
SW RA, 0x14(SP)

LUI T1, 0x8034
ORI T2, R0, #10
SUBIU T2, T2, #6
SH T2, 0xB262(T1)

LW RA, 0x14(SP)
JR RA
ADDIU SP, SP, 0x18
```

The above code now subtracts 6 from 10, resulting in 4. So, our coin counter shows now 4. See, it really isn't that hard as many think.

The next subject is “Multiplication” and “Division”. Unfortunately, there's no multiplication or division operation allowing an immediate value. Theoretically I could implement it as a pseudo-opcode in CajeASM, but this would just confuse people more. Basically, multiplication and division is available as register operation only. So, you would've to load the value to a register first and then multiply or divide. We will use two opcodes for this: **MULT** and **DIV**. I guess you know what **MULT** and **DIV** mean. But here's a slight difference in opposite to addition and subtraction! The result of the operations are (if you know already from the register chapter) stored in **HI** and **LO** register and not inside of our operands. **HI** and **LO** are special registers, as they can't be called directly but only from a few instructions.

An example would be:

```
ORI T0, R0, #2
ORI T1, R0, #3
MULT T0, T1
```

The above operation consists out of three instructions. First, we load the decimal value 2 to **T0** and decimal value 3 to **T1**. Then we multiply the contents of **T0** and **T1**. The result is now stored in **HI** and **LO**. Now, let's say the result is greater than 32-bit aka like a doubleword(64-bit). Then the result is saved in **HI** and **LO**. Usually however our value is only a word, halfword or byte and so it's mostly stored in **LO** register. So, we use another instruction to move the result **FROM** **LO** to our destination!

Ex.:

```
ORI T0, R0, #2
ORI T1, R0, #3
MULT T0, T1
MFLO T0
```

Now, this does the same like above, just that in the end (after the multiplication) the result is moved from **LO** register to **T0** register. So, **T0** would be (after MFLO) #6 (decimal) now. See, it's actually a really simple thing. There are more instructions like MFLO. I'm listing them all here:

MFLO = Move From **LO** Register
MFHI = Move From **HI** Register

Then there's the opposite, which allows you to move the register content **to** the special registers **LO** and **HI**.

MTLO = Move to LO Register
MTHI = Move to HI Register

Now, let's show off an example with our coin counter code again:

```
.org 0x861C0
ADDIU SP, SP, 0xFFE8
SW RA, 0x14(SP)

LUI T1, 0x8034
ORI T2, R0, #10
ORI T3, R0, #3
MULT T2, T3
MFLO T2
SH T2, 0xB262(T1)

LW RA, 0x14(SP)
JR RA
ADDIU SP, SP, 0x18
```

This would now calculate 10 * 3 = 30, meaning that Mario would have 30 coins in his counter now. Wow, this is really awesome, isn't it?

Now, we come to division. Well, basically it's the same like with multiplication. However, there's a difference now! While multiplication stored the upper 32-bit half to **HI** and the lower half to **LO** (whereas we mostly use **LO** as our results probably aren't going to be 64-bit long. Way too big) the division uses HI as the remainder and **LO** for the quotient aka the result of the division. REMEMBER this difference! I show you again:

Multiplication:
If the result is in a 32-bit range such like **0x2554** or **0x2444D8** or **0x2554D89A**, then the result is stored in **LO** register only and **HI** is zero! If the result is in a 64-bit range, then the lower 32-bit half is stored in **LO** while the upper 32-bit half is stored in **HI**. Example:

0x**25449874|2554255D**

Basically, **HI** would contain: **0x25449874** and **LO** would contain: **0x2554255D**.
Division:
Here it's completely different from multiplication. Here, **HI** is used as the remainder (if you divide 3 / 4 for example, the remainder would be 1) and **LO** is used as the quotient aka the result of the division.

So, let's do one short example with our coin counter again and then we're actually done here. I think you should know already how this works:

```
.org 0x861C0
ADDIU SP, SP, 0xFFE8
SW RA, 0x14(SP)

LUI T1, 0x8034
ORI T2, R0, #10
ORI T3, R0, #3
DIV T2, T3
MFLO T2
SH T2, 0xB262(T1)

LW RA, 0x14(SP)
JR RA
ADDIU SP, SP, 0x18
```

As you can see, it's the same again. We divide **T2** / **T3** and then the result is stored in **LO** and we load it from there into **T2**. 10 / 3 is 3.333 and so on. In short, it's 3. So, Mario's coin counter would be now 3.

Now, we learned a lot about the arithmetic operations which can be done on MIPS. In later chapters, we are going to learn more about shifting, what right and left shifts are and many other stuff. After this chapter, we're finally going to turn more "dynamic" in ASM coding. So, please prepare yourself once again, look if you understood everything and if not, try to reread everything again and eventually try to code something yourself. If you're ready, then continue.

Chapter 10: Branches and direct branching

Now, in this chapter, we're finally going to move on. Before we always used the same old example storing a value to an address. But it would be completely boring and nonsense to do this. We want our code to be more dynamic, like let our code check for a certain amount of coins and then do a specific action. Now... it gets interesting and this is our next subject: "Branches". First off, what exactly is a branch? If you were familiar with any programming language before, then you know that branches are nothing else than If … then … statements. You also find them in your language you speak currently.

If the weather is sunny, I will go to the beach.

Our condition is that the weather must be sunny in order that we go to the beach. Now, the very same happens in programming. There's a condition which has to be fullfilled and only then a specific action is done. Let's illustrate this in a more "code" context:

```
[Branch Instruction] rs, rt, !Label
[Opcode]                        ;\
[Opcode]                        ;  | CASE 1 (Condition is false)
[Opcode]                        ;/
[…]

!Label:
[Opcode]                        ;\
[Opcode]                        ;  | CASE 2 (Condition is true)
[Opcode]                        ;/
[…]
```

So, the above is the general tree structure of a condition. In the beginning, we have the branch instruction which checks for a condition. (more specifically, compares two registers for …) If the condition is true, then the code which I marked as “CASE 1” is skipped and our routine jumps to the code below the !Label. Else, if the condition is false, then CASE 1 is not skipped and executed. However, you still have to keep in memory that even if the condition isn't true that the code in CASE 2 is still executed. By the way, the name “!Label” can be also any name you can think of. Just look, that you don't use the same name, as this overrides the old label position to the newest label definition! Instead of writing !Label you could also call it: “!ConditionTrue:”

To sum it up, a branch instruction compares two registers and, depending on if the condition is true or false, decides whether to jump to the label or to continue the code below the label.

Of course, there are also other uses for branch instructions, such like the known “Loops”. The tree structure is similar to normal labels, but in this case **backwards**. Example:

```
!Label:
[Opcode]                   ;\
[Opcode]                   ; | CASE 1 (Condition true)
[Opcode]                   ;/
[…][Branch Opcode] !Label
[Opcode]                   ;\
[Opcode]                   ; | CASE 2 (Condition false)
[Opcode]                   ;/
[…]
```

So, what would this do? Well, first CASE 1 is executed and then it checks for the condition. Now, what happens? So, if the condition is true, then we jump **backwards** in our code. That means CASE 1 is executed again. If the condition is true again, then CASE 1 is executed once again. This is going to happen until the condition is false. Only then CASE 2 is executed. So, this tree structure is useful when you want to execute a code for x times. However, you should look out to not create infinite loops, as this is most likely going to crash the game. Now, you know the general structure of a branch. Now, I'm going to list a few branches, which you should keep in mind:

BEQ (Branch on EQual) → Branches, if both operands are equal to each other.

BNE (Branch on not EQual) → Branches, if both operands are NOT equal to each other.

BGT (Branch on Greater Than) → Branches, if the source operand is greater than the target operand.

BLT (Branch on Lesser Than) → Branches, if the source operand is lesser than the target operand.

BGE (Branch on Greater than or Equal To) → Branches, if the source operand is greater or equal to the target operand.

BLE (Branch on Lesser than or Equal to) → Branches if the source operand is lesser or equal to the target operand.

There are even way more branch instructions, however the most of them are completely useless to us, as we only need them very rarely. The above instructions are the common and most important ones you should remember.

Now, let's move on and write a bit more dynamic code. For example, we will write a code which checks if Mario has 10 coins. If the condition is true, Mario's life counter should be increased by one and Mario's coins should be reset to 0 again. If you are ready, then let's start!