### **Preface**

Continuing with the obj struct from the first tutorial, this tutorial will cover how float and branch operations work and go over some examples on how to use them. I will also cover how to use the stack to save variables.

### **Floats**

Floats are numbers that are represented using 4 bytes instead of a binary representation like an integer. Floats use exponentials to get a wide range of values, they are the only way to represent decimals on the N64. Floats in sm64 have their own registers, F0 to F31. They do not interact with integers, which are the numbers we dealt with last tutorial, or their registers. The only way to interact between floating point registers and integer registers is to use specific asm cmds. Floats also have their own cmds which I will go over next.

### **Floating point cmds**

When you want to load a value into a floating point register, you use *LWC1,* load word from coprocessor 1. Store word is similar, *SWC1*. You can only load/store words with floats because they always take 4 bytes no matter their value, there is no *SHC1* for example.

Add, subtract division and multiplication are *ADD.S SUB.S MUL.S DIV.S* respectively. You can also negate, take an absolute value and square root floats. These will be *NEG.S, ABS.S* and *SQRT.S*

Finally to interact between integers and floats you have to use the cmds *MFC1 MTC1, CVT.W.S* and *CVT.S.W*

These cmds move from and to coprocessor 1 and convert from float to int, and int to float respectively.

Ex. 1: loading and adding

* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LWC1 F0, 0XA4 (T0) //y pos
* ADD.S F0, F0, F0
* JR RA
* SWC1 F0, 0XA4 (T0)

This function doubles an objects y pos every frame. Here you can see that to load and store the object’s y position, which is a float, I use floating point variation of load and store cmds.

Ex 2: Converting int to float

* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LW T1, 0X188 (T0) //BPARAM
* MTC1 T1, F0
* CVT.S.W F0, F0
* JR RA
* SWC1 F0, 0XA4 (T0) //y pos

Here I load the Bparam of an object, and store it in the y position. Notice that I use *MTC1,* move to coprocessor 1. This moves T1 to F0. Next I convert it to a float. Note the order, S.W means single (another word for float) from word. Finally I store with *SWC1.* All together this takes the number in the object’s Bparam and stores that to the y position.

Ex 3. Rotation from movement

* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LWC1 F0, 0XA4 (T0) //y pos
* CVT.W.S F0, F0
* MFC1 T9, F0
* JR RA
* SW T9, 0XD4 (T0)

Here I take the y position and store it as a y rotation. I first load the objects y pos, then convert it to an integer. I use *CVT.W.S* which means word from single. Then I just *MFC1* to put f0 into t9. Even though the cmd is going the opposite order from *MTC1* the int and float registers still go in that order. Finally I store the y rotation. Note that y pos will increase much more slowly than rotation so in game the object will be out of sight before it makes any noticeable rotation.

### **Branches and jumps**

Branches are a method to goto another section of code on a condition. A jump is when you jump to an entirely different section of code with no condition. Branches use a relative address while jumps use an absolute address. While both have a similar purpose how you use them will be different.

There are three main jump cmds you should be aware of. *J* which is a simple jump. Using this cmd will make the code goto the specified address. Next is *JAL* which is jump and link. This works like *J* but it saves the return address to the register RA. This means you can return to the address you jumped from by jumping to RA. Finally is *JR* which jumps to an address, but the argument is a register instead of a written address. You may recognize this as what I've been using to end functions with. *JR RA* returns to where the function was called from and allows the rest of the games code to execute. You can also use this cmd to create a switch statement by changing the argument you jump to based on the respective case.

Branches are mostly used as a conditional jump, or a short distance jump. A conditional jump simply means that if a specified condition is true, the code goes to the address, else it does nothing. There are three main branches. Branch on equal/not equal, branch on greater than, and branch on less than.

Finally, all branches and jumps have a delay slot. What this means is that the line of code below the branch or jump will also execute before the code goes to the new address.

There is an option to not use the delay slot unless you branch. To use this, simply put an L after your branch cmd; this stands for *Likely*. In certain scenarios this can save you a line of code or make things easier to understand.

Ex. 1 Go upwards if rotation is zero

* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LW T1, 0XD4 (T0)
* BNEZ T1, STAY
* LWC1 F0, 0XA4 (T0) //y pos
* ORI T9, R0, 0X100
* MTC1 T9, F2
* CVT.S.W F2, F2
* ADD.S F4, F2, F0
* SWC1 F4, 0XA4 (T0)
* STAY:
* JR RA
* NOP

Here I use a branch to set the condition of rotation == zero. In this code, we can use labels because it is a feature of the compiler. You should almost always try to use labels instead of branching to an address. When the condition I set is true, the code will continue from the point of “STAY” instead of executing the lines below.

The branch I set has the condition: branch on T1 not equal to zero; *BNEZ*. So this means the code finishes early when T1 is not zero, and executes all code when it is.

Ex. 2 Branch on equal

* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LW T1, 0XD0 (T0)//X ROT
* LW T2, 0XD8 (T0)//Z ROT
* BEQ T1, T2, PLANAR
* ADDIU T1, T1, 0X50
* SW T1, 0XD0 (T0)
* PLANAR:
* JR RA
* NOP

This code checks if the x and z rotation of an object are equal, and if not it rotates x by 0x50 every frame. Eventually the x rotation will line up with the z and it will stop rotating assuming both have the same %0x50 (modulo 0x50) value.

Ex. 3 Jump and Link

* LUI A0, 0X8036
* LW A1, 0X1158 (A0)
* JAL 0X8029E2F8 //GET DIST 3D, RETURN F0
* LW A0, 0X1160 (A0)
* LUI T0, 0X8036
* LW, T0, 0X1160 (T0)
* JR RA
* SWC1 F0, 0XA4 (T0)

This code jumps to another piece of code and then stores f0 to the objects y pos. There are a few things you may notice about this. First I used *JAL* which if you recall, stores the return address to *RA.* This is how the code on the following lines can execute. If you recall what I mentioned last tutorial, all registers are temporary. That is why we have to load 0x80361160 after the *JAL.* *A0* is not remembered. You can now see why we always end our code with *JR RA.* It is because our code is called from the main game function and must return so the game can continue.

Before I explain how the code I jump to actually works, we must first talk about storing variables. If you understood the tutorial up to this point, you should have noticed that *RA* is overwritten in the middle of the code. That means when we end with *JR RA*, we actually return to line 4 of our code. This is an infinite loop and will cause the game to freeze. We have to save *RA* before we use *JAL* so that we can return properly.

### **Storing variables**

The only way to store a variable is with the store cmd. In general, you can store your variables anywhere in ram, but storing in random places will quickly get confusing and the chance of overwriting something else is high. To solve this, we separate the data we need to store into two categories.

Long term/global variables and short term/local variables. A good example of the first is 0x80361160. It is a variable we have been making use of this entire time and has a dedicated spot in ram. Many important global variables are stored this way. Short term variables are ones like our return address which we just need to store so that we can access it in case the register is overwritten. There is no need to allocate a spot in ram just for our *RA* because it will only be used once. This is what the stack is for.

The stack is just space in memory allocated for temporary use and storage. To use the stack, you simply get the location of where it is, allocate a section of memory from that location for your function, and then once you're done you tell the game that the memory is free to use again. This is done by using the stack pointer, or *SP.*

The stack pointer is the location of current memory usage. Everything below it is being used to store variables in the chain of functions that came before your function. Everything after it is no longer in use. You advanced the stack pointer by the number of bytes you need for your function and then that memory will not get erased until you remove the bytes you gave to *SP*. Doing so looks like this:

* ADDIU SP, SP, 0XFFE8
* JR RA
* ADDIU SP, SP, 0X18

Here I take 0x18 from the stack, then add 0x18 to the stack. Now with this we can finish the previous example properly.

Ex. 3 Jump and Link

* ADDIU SP, SP, 0XFFE8
* SW RA, 0X14 (SP)
* LUI A0, 0X8036
* LW A1, 0X1158 (A0)
* JAL 0X8029E2F8 //GET DIST, RETURN F0
* LW A0, 0X1160 (A0)
* LUI T0, 0X8036
* LW, T0, 0X1160 (T0)
* LW RA, 0X14 (SP)
* ADDIU SP, SP, 0X18
* JR RA
* SWC1 F0, 0XA4 (T0)

Here you can see that we allocate 0x18 bytes, store the return address, load it after our *JAL* and then return those 0x18 bytes. The return address always goes in 0x14 of *SP.* Bytes below 0x14 are used for passing additional variables to functions and bytes after are free for any use.

You may be wondering what I am actually jumping to. I call a function that takes two arguments, and returns another value. 99% of the time you use *JAL* it will be to call a function. The function I call takes two object structs, and returns the distance between them as a float. I use 0x80361160 the current obj and 0x80361158 which is mario. As previously mentioned, arguments for functions usually go in *A0* and *A1.*

If you want to learn some functions used by the game these are the best starting points:  
[function list](https://hack64.net/wiki/doku.php?id=super_mario_64:function_list)

[kaze's function list](https://sites.google.com/site/kazemario64/tons-of-functions)

They both are rather incomplete but for most things you can get by with these functions. I am working on adding more to the function list on hack64 but ideally if you need a specific function you will have to find it yourself with a debugger, which I may go over in a much later tutorial. It’s always a good idea to experiment and keep your own notes. If you find a new function or something unexplained on the wiki you should try to add it to the resources for other people to use. Nothing is too trivial to add but if you’re unsure you can always ask on the discord page linked on the website.

### **More examples**

Ex. 1 Branching on function return

* ADDIU SP, SP, 0XFFE8
* SW RA, 0X14 (SP)
* JAL 0X802A3CFC//is mario on this obj (must be solid)
* NOP
* BEQ V0, R0, OFF
* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LWC1 F0, 0XA4 (T0)
* ORI T9, R0, 0X50
* MTC1 T9, F2
* CVT.S.W F2, F2
* ADD.S F4, F2, F0
* SWC1 F4, 0XA4 (T0)
* OFF:
* LW RA, 0X14 (SP)
* JR RA
* ADDIU SP, SP, 0X18

This function checks if mario is on top of the current object, then goes upwards if he is. Even though *JAL* has a delay slot I do not use it because I have to store my *RA* before the *JAL*. I put a *NOP* afterwards so that my branch does not go off before the *JAL.*

The comment that the object must be solid just means that mario has to be able to stand on the obj. I will go over some properties of objects in the next tutorial.

Ex. 2 switch + case

* .ORGA 0X80401000
* LUI T0, 0X8036
* LW T0, 0X1160 (T0)
* LW T1, 0X188 (T0)
* LUI T2, 0X8040
* ADDIU T2, T2, 0X1028
* SLL T4, T1, 2 //times 0x4
* SLL T3, T1, 3 //times 0x8
* ADDU T1, T4, T3 //times 0xC
* ADDU T1, T1, T2
* JR T1
* LW T3, 0XD4 (T0)
* ADDIU T3, T3, 0X50
* B END
* SW T3, 0XD4 (T0)
* SUBIU T3, T3, 0X50
* B END
* SW T3, 0XD4 (T0)
* ADDIU T3, T3, 0X150
* B END
* SW T3, 0XD4 (T0)
* SUBIU T3, T3, 0X150
* SW T3, 0XD4 (T0)
* END:
* JR RA
* NOP

This code rotates the object by different amounts based on the Bparam. This is commonly known as a switch/case statement in modern computing languages. This method works only when each case has the same length. If they're different lengths a more elaborate method is required.

Compared to earlier examples, this code has a start address. This is required for all code, I just left it out for simplicity in earlier examples.

First I load the objects Bparam. Next I multiply it by 0xC. This is done by using bit shifts which are functionally the same as multiplication by 2. In general, the easiest way to multiply numbers is to do it in powers of two, then add the products. I add the result to 0x80401028 which is the address two lines below the *JR RA*. Another way of seeing this is T1 is the offset of our case. So case 0 is 0x0 bytes from the start so a Bparam of 0 \* 0xC = 0x0 -> case 0 is selected. Bparam of 1 = 0xC -> case 1 is selected etc.

One thing to note is that I branch to the end with each case. This is because I only want the 3 lines of each case to execute. This is the equivalent of a break in modern computing languages. Usually you would put a break for every case, but it is functionally equivalent to have none in the last case. This saves us a line of code.

Note that storing *RA* is not required because we do not use any *JAL* cmd, you should always look to minimize code and save space on the stack, even a single line.

Ex. 3 Raycast

* ADDIU SP, SP, 0XFFE0
* SW RA, 0X14 (SP)
* LUI A0, 0X8036
* LW A1, 0X1158 (A0)
* LW A0, 0X1160 (A0)
* LWC1 F0, 0XA4 (A1)
* LWC1 F2, 0XA4 (A0)
* SUB.S F4, F2, F0
* LWC1 F0, 0XAC (A1)
* LWC1 F2, 0XAC (A0)
* SUB.S F6, F2, F0
* SWC1 F6, 0X18 (SP) //z
* JAL 0X8029E27C //dist 2d, return F0
* SWC1 F4, 0X1C (SP) //x
* LUI T9, 0X8038
* ORI T8, T9, 0X6000
* ORI T9, T9, 0X7000
* LUI T0, 0X8036
* LW T1, 0X1158 (T0)
* LHU T2, 0XD6 (T1) //y rot
* SRL T2, T2, 0X4 //DIV 0X10
* SLL T2, T2, 0X2 //MULT 2
* ADDU T8, T8, T2
* ADDU T9, T9, T2
* LWC1 F2, 0X0 (T8) //cos
* MUL.S F2, F0, F2
* LWC1 F4, 0X1C (SP)
* SUB.S F4, F4, F2
* LWC1 F2, 0X0 (T9) //sin
* MUL.S F2, F0, F2
* LWC1 F6, 0X18 (SP)
* SUB.S F6, F6, F2
* MUL.S F6, F6, F6
* MUL.S F4, F4, F4
* ADD.S F8, F6, F4
* SQRT.S F8, F8 //ray dist
* CVT.W.S F0, F8
* MFC1 T9, F0
* SUBIU T9, T9, 0X100
* BGEZ T9 MISS
* NOP
* JAL 0X803839CC //be solid
* NOP
* MISS:
* LW RA, 0X14 (SP)
* JR RA
* ADDIU, SP, SP, 0X20

This function calculates if a ray will go within a certain distance of an object. Basically imagine the object as a circle, and this function will test if a line goes inside that circle.

First I store *RA* in the stack. I take 0x20 from the stack so I can store two other variables later. I get a line from mario to my object by subtracting the X+Z components of their positions. I then get the distance between the objects using an existing in game function. I don't want to lose my X+Z distances so I store those in the stack.

Next I get the X+Z components of my line from mario to the object. I do this by multiplying the length by sin+cos of marios facing angle. These values are just a lookup table so that is why I can take the angle, do some math and load them directly.

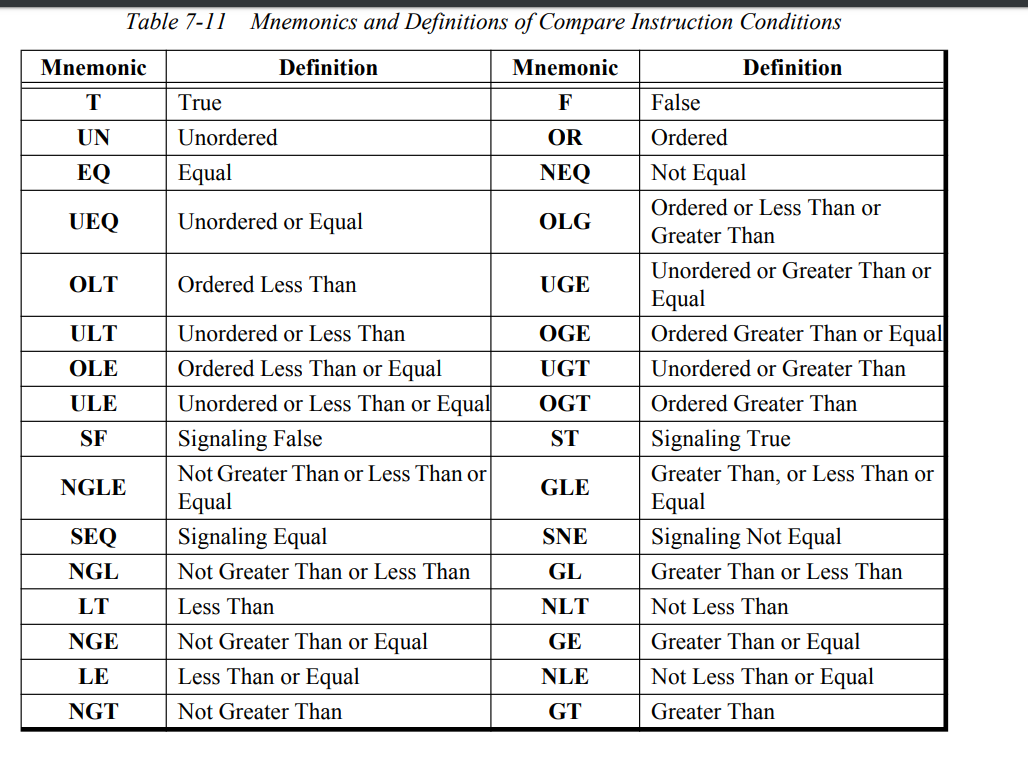
Finally I get the distance between mario’s facing vector and the object mario distance vector and check if it is under a certain value, 0x100 in this example. I branch if the distance is greater than 0x100 so it does nothing, and I use a function that processes collision when it is less than 0x100. The function I use there is just a dummy example. You can use any function you want there.

Note: the sin/cos may be backwards since I did not double check the function in game but the concept is the same.

### **Floating Branch Cmds**

It is possible to use branches with floating point cmds as well. First you must use a comparison operation. The format for that is:  
*C.cond.fmt* **Source register**, Target register

The format is either single, or double. It will match the size of the float. So far I only ever used single precision floating point numbers, which are a word in size. The condition can be any of the following from this image:



You can see that there are many options. You do not have to worry about *Ordered* and *Signaling* conditions listed here for general computations. They refer to NaN comparisons. The basic idea behind them is that you cannot compare something when it is NaN, so *Ordered* is used to determine that none of them are NaN; e.g. Ordered means that a comparison can be completed, and that neither register has NaN. *Signaling* will throw an exception if either register is unordered, e.g. either register has a NaN value.

The other values are much more straightforward. You compare the source register to the target register: e.x. C.LT.S *F0, F2* compares if *F0* is less than *F2*.

After you do the comparison, you will do a branch operation on whether than comparison is true. The associated cmds for that are *BC1T,* branch on coprocessor 1 true,and *BC1F,* branch on coprocessor 1 false*.* Between the comparison and branch cmd there needs to be a delay slot, on top of the delay slot after the branch that every other branch cmd has.

Ex. 1 Slingshot

* .dw 0x00040000
* .dw 0x11010041
* .dw 0x08000000
* .dd 0x0c00000080408014
* .dw 0x09000000
* addiu sp, sp, -0x18
* sw ra, 0x14 (SP)
* lw t0, 0x80361160
* lw t1, 0x14c (T0)
* //0 for mario not in
* //1 for mario inside
* //2 for mario is ready to launch
* //3 for launching mario
* //4 for reset when mario lands
* beq t1, r0, @@searchmario
* li t2, 1
* lh t4, 0x8033af92 //vert tilt
* bgtz t4, @@absVT
* @@absVT:
* subu t4, r0, t4
* beq t2, t1, @@marioinside
* li t3, 2
* beq t3, t1, @@primed
* li t2, 3
* beq t2, t1, @@launching
* li t3, 4
* beq t3, t1 @@reset
* nop
* @@searchmario:
* lui at, 0x4280
* lwc1 f0, 0x15c (T0)//dist to mario
* mtc1 at, f2
* c.lt.s f0, f2
* li t8, 1
* bc1f @@end
* li t9, 0x00001371 //inside cannon
* sw t8, 0x14c (T0)
* lui t7, 0x8034
* b @@end
* sw t9, 0xb17c (T0)
* @@marioinside:
* subu t3, t4, 0x50
* blez t3, @@end
* li t4, 2
* sw t4, 0x14c (T0)
* @@primed:
* subu t3, t4, 0x46
* bgez t3, @@end
* li t4, 3
* b @@end
* sw t4, 0x14c (T0)
* @@launching:
* lw t5, 0x154 (T0)
* li t6, 0x40
* li t7, 1
* beql t5, t6, @@end
* sw t7, 0x14c (T0)
* subu t3, t4, 0x10
* bgez t3, @@end
* li t4, 3
* sw t4, 0x14c (T0)
* mtc1 t5, f6
* cvt.s.w f6, f6
* lui at, 0x4220 //40
* mtc1 at, f0
* lui at, 0x41a0
* mtc1 at, f2
* div.s f0, f0, f6 //40/time to flick
* add.s f2, f0, f0 //20 + 40/time
* lui t1, 0x8034
* swc1 f2, 0xb1c4 (T1) //h spd
* li t2, 0x00880898 //shot out of cannon
* b @@end
* sw t2, 0xb17c (T1)
* @@reset:
* lui t7, 0x8034
* lw t7, 0xb17c (T0)
* andi t6, t7, 0x1c0
* li t5, 0x80 //in air
* beq t5, t6, @@end
* li t4, 0
* sw t4, 0x14c (T0)
* @@end:
* lw ra, 0x14 (SP)
* jr ra
* addiu sp, sp, 0x18

This code interacts with mario which I will actually go over in the next tutorial, but I will leave this code as an intro to that. The basic idea behind this code is that each action corresponds to a specific state of its interaction with mario. First I set the behavior to just run a function, but I also set the object flags to have 0x41. This means that it sets 0x15c (distance to mario) in the object struct automatically every frame. From here I will go over what I do in each state.  
In state zero, I am searching for mario to be near my object so I can put him inside of it. I do this by loading a value into *AT* and then comparing that to my distance to mario. If the distance to mario is less than 0x4280 0000 or 64 in decimal, then I will not branch, otherwise I will. When the comparison fails, I change the action of mario (I will go over this next tutorial) and I change the action of my object so that I can run different code.  
While the action is 1, I run “@@marioinside” where I check the absolute value of the vertical tilt of the control stick. When its greater than 0x50, I change the action again.  
In the next action I am checking for the beginning of a release of the control stick. I check this again by using the absolute value of vertical tilt.

I do this once more in another action to measure the speed at which the stick was released. This works because each action change resets the object timer (0x154), and I can use that timer to measure the “flick” of the control stick.

Once the control stick has been returned to near neutral, I set the speed of mario. I do this by doing some simple math:  
Mario->Hspd = 40/obj->time + 20;  
The arrow notation here is from C, it basically means the owner of that struct value, it is not too important as its just pseudo code to understand what I wrote.

After I calculate the speed, I store it in mario’s struct and change his action again (I will go over in the next tutorial how this works).

Finally I wait until mario lands to reset the slingshot. I do this because the first frame after launching, if I go back to state 0 mario will instantly be caught again.

### **Conclusion**

The previous code is a little bit ahead of what I had previously gone over in this tutorial but I had thought it was important to go over floating point branches. The section probably feels disconnected and that’s because I wrote it several months after the previous sections.

With this tutorial you will have most of what you need in terms of editing objects. Tutorial 3 will solidify this basis and give you almost all the tools you need for custom objects. Afterwards I will delve into some basic graphics and afterwards hopefully cover some practical examples.