COMP1521

Week 4 - MIPS Advanced Data and Functions

Overview

Admin

MIPS 2D Arrays

MIPS Structs

MIPS Functions

Admin

W4 lab due next monday.

W3 weekly test was released last thursday, is due this thursday.

Assignment content has all been covered in lectures, and this tut will go over the rest of the content that wasn’t covered in last week’s labs.

Assignment is due W5 friday

Help sessions are running and more will run as the deadline for the assignment approaches (see times and locations on course website)

Admin: Assignment 1

If you start early, you can go to earlier help sessions (if you need) and tutors will be able to spend more time helping you.

If you attend the help sessions in the couple days before the assignment is due, the amount of time tutors can spend on each student is limited.

The same applies to asking for help on the forum. Earlier posts will receive faster replies, and as we get closer to the deadline it will take longer for tutors to get through the posts.

The style mark is worth 20%. 12/20 of those marks are allocated to FILLING OUT THE FUNCTION COMMENTS.

Please fill out the function comments, we hate deducting 12 marks from a student who has beautiful MIPS code because they didn’t fill out the function comments.

MIPS 2D Arrays

How to implement 2D array

We already implemented a 1D array. A 2D array is, in one interpretation, an array of 1D arrays.

So, we can apply the principles of 1D arrays, where each element of that array is a 1D array.

This will effectively create a 2D array.

What would that look like in memory?

It looks like an array with really big elements.

If we look closer at those elements, those element in fact contain smaller elements.

We can index once to access a big element (an array).

If we index twice we can access an element within that array (an element of the 2D array).

```
char arr[10][10] // An array of 10 arrays
                 // Each element contains 10 chars

00 _ _ _ _ _ _ _ _ _ _ Element 0 (arr[0])
10 _ _ _ _ _ _ _ _ _ _ Element 1 (arr[1])
90 _ _ _ _ _ _ _ _ _ _ Element 2 (arr[2])
20 _ _ _ _ _ x _ _ _ _ Element 3 (arr[3])
30 _ _ _ _ _ _ _ _ _ _ Element 4 (arr[4])
40 _ _ _ _ _ _ _ _ _ _ Element 5 (arr[5])
50 _ _ _ _ _ _ _ _ _ _ Element 6 (arr[6])
60 _ _ _ _ _ _ _ _ _ _ Element 7 (arr[7])
70 _ _ _ _ _ _ _ _ _ _ Element 8 (arr[8])
80 _ _ _ _ _ _ _ _ _ _ Element 9 (arr[9])

x = Element 5 inside arr[3]
x = arr[3][5]
```

How can you calculate the address?

```
char arr[10][10] // An array of 10 arrays
                 // Each element contains 10 chars

00 _ _ _ _ _ _ _ _ _ _ Element 0 (arr[0])
10 _ _ _ _ _ _ _ _ _ _ Element 1 (arr[1])
90 _ _ _ _ _ _ _ _ _ _ Element 2 (arr[2])
20 _ _ _ _ _ x _ _ _ _ Element 3 (arr[3])
30 _ _ _ _ _ _ _ _ _ _ Element 4 (arr[4])
40 _ _ _ _ _ _ _ _ _ _ Element 5 (arr[5])
50 _ _ _ _ _ _ _ _ _ _ Element 6 (arr[6])
60 _ _ _ _ _ _ _ _ _ _ Element 7 (arr[7])
70 _ _ _ _ _ _ _ _ _ _ Element 8 (arr[8])
80 _ _ _ _ _ _ _ _ _ _ Element 9 (arr[9])

x = arr[3][5]
&x = arr + (3 * 10) + (5) // To get to X we should access the big element 3 (arr[3])
                          // (at arr + (3 * 10))
                          // And we should index that element by 5          (arr[3[5])
                          // (at arr + (3 * 10) + 5)
```

How can you calculate the address?

A general solution involves the size of the element AND the length of each Big element.

For an array char[10][10], to access an element at arr[i][j], we should go to the address

arr + 10 * 1 * i + 1 * j

=   arr+ 10 * i + j

Since we need to index to the i’th Big element, and within that we should index to the j’th element.

How can you calculate the address?

For an array like int[10][10], we need to remember each individual element (small element) is 4 bytes large.

So, to access arr[i][j], we need:

arr + 10 * 4 * i + 4 * j

Because each Big element is an array of 10 ints, so is 4 * 10 bytes large.

And to index to the element inside that, each element is still 4 bytes large, so we multiply j by 4 as well.

Distinguishing between number and size of big elements

The array type int[10][5] is an array of 10 elements, where each element is an array int[5].

The array type int[5][10] is an array of 5 elements, where each element is an array int[10].

Keep this in mind when translating, to account for the correct size of the “big elements”.

A general solution

So, in general:

For an 2D array of a type with size “size”,

Where the number of big elements in the array is x and the “size” of each big element is y

(like type[x][y])

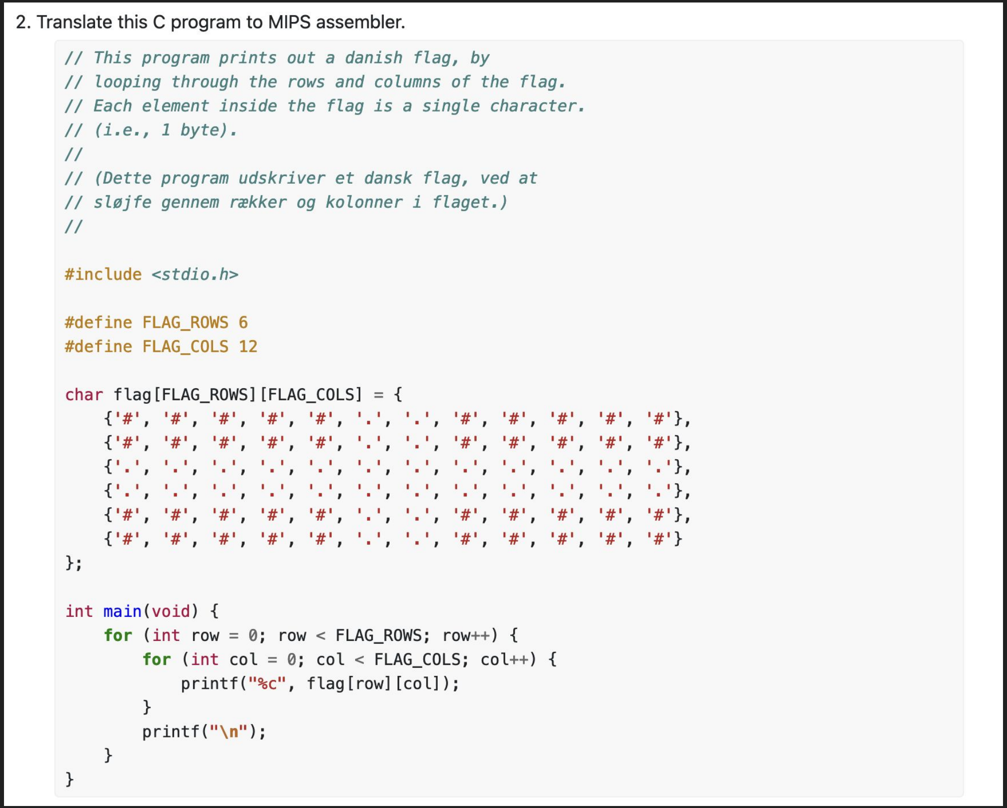
To access at arr[i][j], we should go to the address:

arr + size * y * i + size * j

So a concrete example is to access arr[2][3] in an array of type int[5][6],

We should go the address arr + 4 * 6 * 2 + 4 * 3

Tutorial Q2



2. Translate this C program to MIPS assembler.

```c
// This program prints out a danish flag, by
// looping through the rows and columns of the flag.
// Each element inside the flag is a single character.
// (i.e., 1 byte).
//
// (Dette program udskriver et dansk flag, ved at
// sløjfe gennem rækker og kolonner i flaget.)
//

#include <stdio.h>

#define FLAG_ROWS 6
#define FLAG_COLS 12

char flag[FLAG_ROWS][FLAG_COLS] = {
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'},
    {'.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'},
    {'#', '#', '#', '#', '#', '.', '.', '#', '#', '#', '#', '#'}
};

int main(void) {
    for (int row = 0; row < FLAG_ROWS; row++) {
        for (int col = 0; col < FLAG_COLS; col++) {
            printf("%c", flag[row][col]);
        }
        printf("\n");
    }
}
```

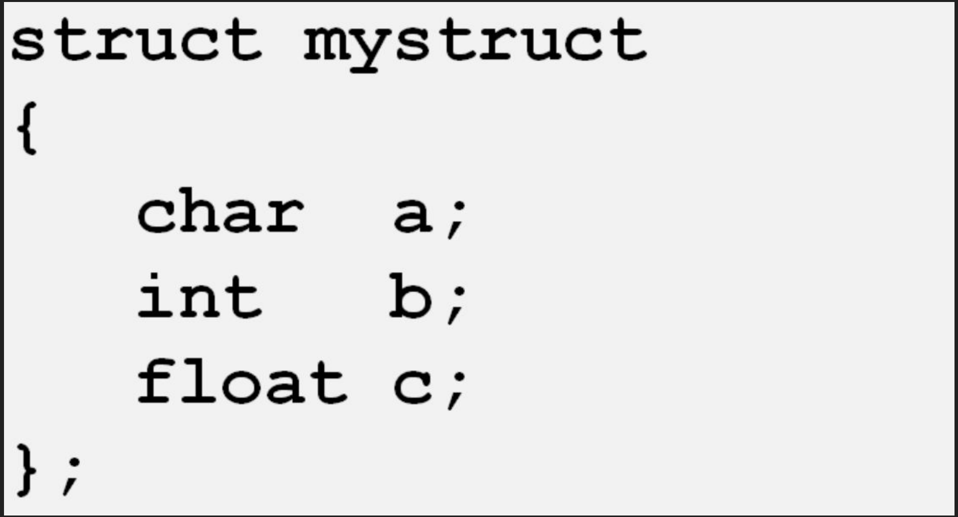
MIPS Structs

How to make a struct

A struct in C is a type which contains multiple variables.

In MIPS structs cannot fit inside registers, since registers only have 4 bytes.

Thus, they must be stored in memory.



```
struct mystruct
{
   char  a;
   int   b;
   float c;
};
```

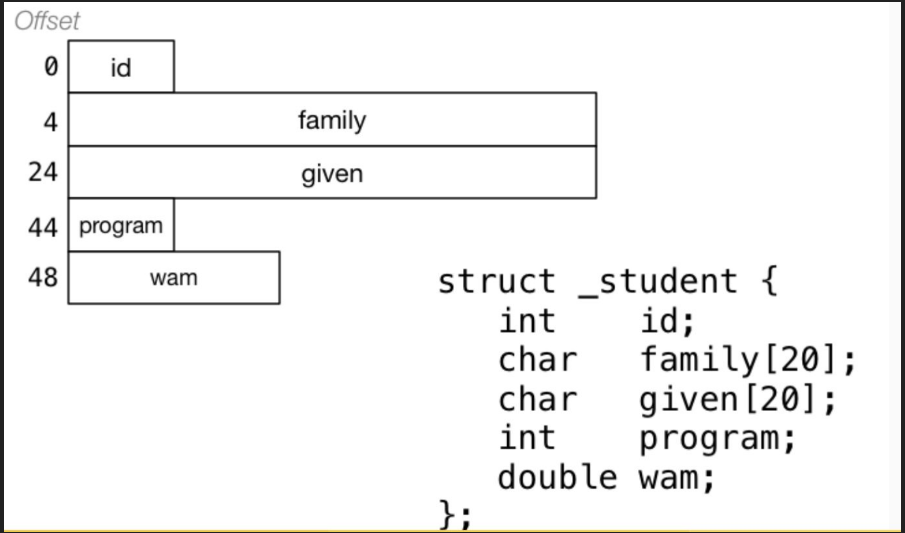
How to make a struct

So, how should we store them in memory?

Well, what if we just put all of them in a row, in memory?

Then, we can define a label for where the struct starts in memory, and add an appropriate offset to access different types within the struct.

Remember types like INT are 4 bytes large, so are aligned to addresses divisible by 4



```
struct _student {
    int    id;
    char   family[20];
    char   given[20];
    int    program;
    double wam;
};
```

Example struct in MIPS

We will implement a struct which just stores two INT values.

We can define it initially using directives:

```
my_struct:
        .word 0, 0
```

Then we can access the first int:

```
lw          $t0,        word
```

Example struct in MIPS

For the second INT, we should typically define the offset using a constant:

```
MY_STRUCT_SECOND_INT_OFFSET = 4   # (4 bytes from the start of the struct)
```

And we can easily access it using the bracket notation for lw:

```
la      $t0,    my_struct
lw      $t1,    MY_STRUCT_SECOND_INT_OFFSET($t0)
                # loading from (my_struct + 4), effectively
```

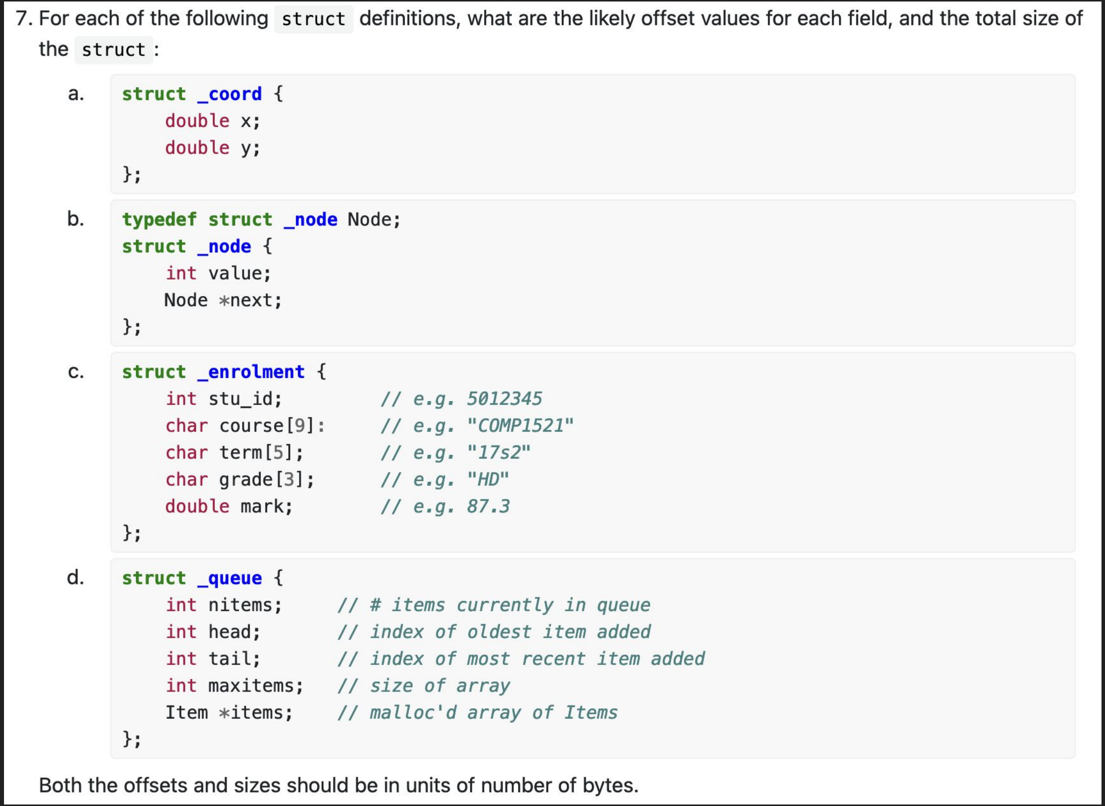
Example struct in MIPS

MY_STRUCT_SECOND_INT_OFFSET = 4   # (4 bytes from the start of the struct)

We could also add the address and offset into a single register:

```
la      $t0,    my_struct
add     $t0,    $t0,    MY_STRUCT_SECOND_INT_OFFSET
lw      $t1,    ($t0)
                # loading from (my_struct + 4), effectively
```

Tutorial Q7



7. For each of the following `struct` definitions, what are the likely offset values for each field, and the total size of the `struct`:

a.
```c
struct _coord {
    double x;
    double y;
};
```

b.
```c
typedef struct _node Node;
struct _node {
    int value;
    Node *next;
};
```

c.
```c
struct _enrolment {
    int stu_id;         // e.g. 5012345
    char course[9]:     // e.g. "COMP1521"
    char term[5];       // e.g. "17s2"
    char grade[3];      // e.g. "HD"
    double mark;        // e.g. 87.3
};
```

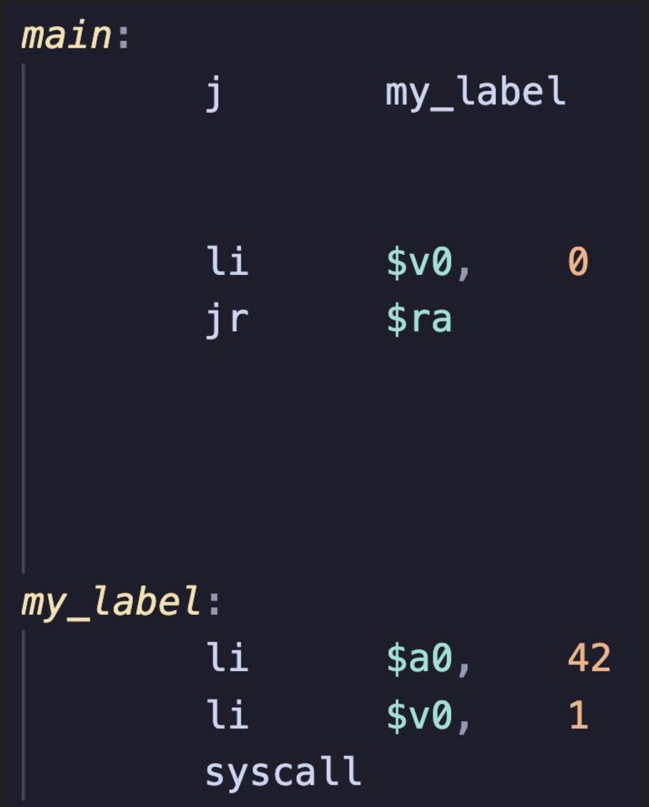
d.
```c
struct _queue {
    int nitems;     // # items currently in queue
    int head;       // index of oldest item added
    int tail;       // index of most recent item added
    int maxitems;   // size of array
    Item *items;    // malloc'd array of Items
};
```

Both the offsets and sizes should be in units of number of bytes.

MIPS Functions

Basic function

Here is the basic idea of a function

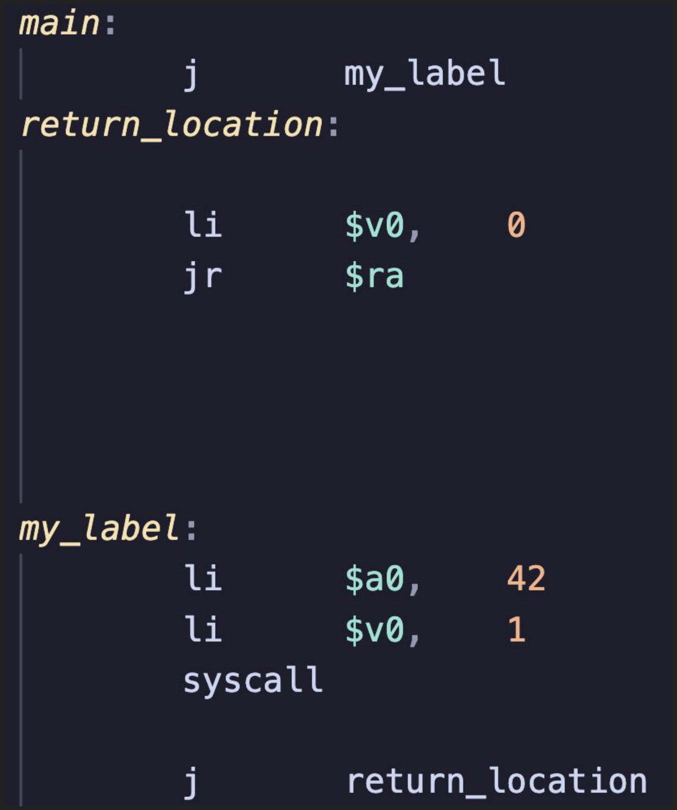


```
main:
        j       my_label



        li      $v0,    0
        jr      $ra







my_label:
        li      $a0,    42
        li      $v0,    1
        syscall
```

Basic function

But, how can we return to main?

We could use a label?



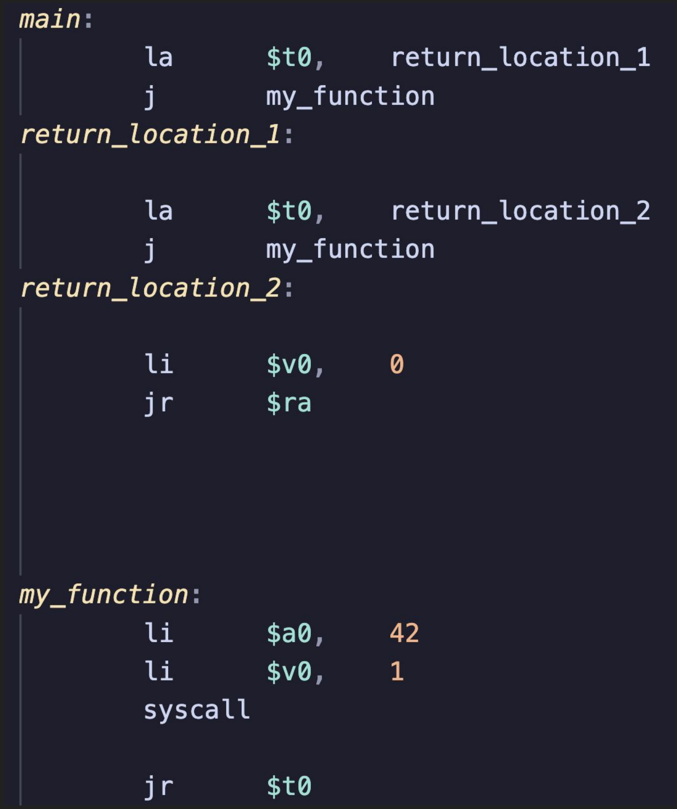
```
main:
        j       my_label
return_location:

        li      $v0,    0
        jr      $ra




my_label:
        li      $a0,    42
        li      $v0,    1
        syscall

        j       return_location
```

Basic function

But what if I want to call the function from two different places?

Maybe I should store the location to return in a register?

We can then use the “jr” instruction, which moves execution to the address in the provided register.

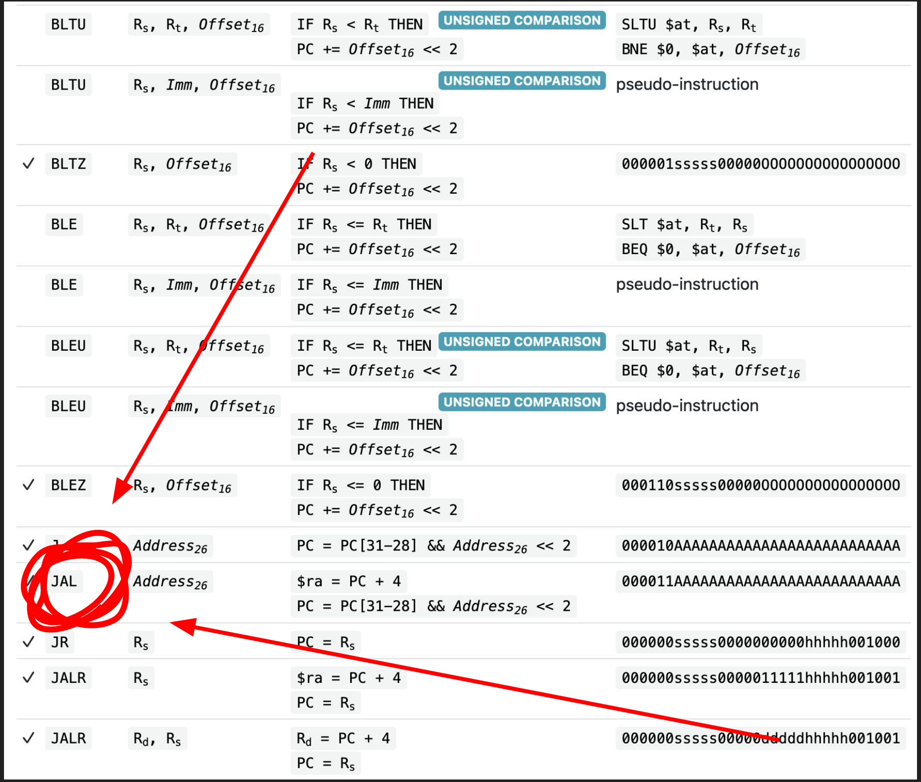


```
main:
        la      $t0,    return_location_1
        j       my_function
return_location_1:

        la      $t0,    return_location_2
        j       my_function
return_location_2:

        li      $v0,    0
        jr      $ra




my_function:
        li      $a0,    42
        li      $v0,    1
        syscall

        jr      $t0
```

Basic function

Ok, sure. But this is annoying, maybe there is an instruction that can automatically jump to the function AND save the “return address” to some register?

Maybe this is why $ra stands for “return address”?



	BLTU	R_s, R_t, *Offset*$_{16}$	IF R_s < R_t THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	SLTU $at, R_s, R_t BNE $0, $at, *Offset*$_{16}$
	BLTU	R_s, *Imm*, *Offset*$_{16}$	IF R_s < *Imm* THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	pseudo-instruction
✓	BLTZ	R_s, *Offset*$_{16}$	IF R_s < 0 THEN PC += *Offset*$_{16}$ << 2		000001sssss0000000000000000000000
	BLE	R_s, R_t, *Offset*$_{16}$	IF R_s <= R_t THEN PC += *Offset*$_{16}$ << 2		SLT $at, R_t, R_s BEQ $0, $at, *Offset*$_{16}$
	BLE	R_s, *Imm*, *Offset*$_{16}$	IF R_s <= *Imm* THEN PC += *Offset*$_{16}$ << 2		pseudo-instruction
	BLEU	R_s, R_t, *Offset*$_{16}$	IF R_s <= R_t THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	SLTU $at, R_t, R_s BEQ $0, $at, *Offset*$_{16}$
	BLEU	R_s, *Imm*, *Offset*$_{16}$	IF R_s <= *Imm* THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	pseudo-instruction
✓	BLEZ	R_s, *Offset*$_{16}$	IF R_s <= 0 THEN PC += *Offset*$_{16}$ << 2		000110sssss0000000000000000000000
✓	J	*Address*$_{26}$	PC = PC[31–28] && *Address*$_{26}$ << 2		000010AAAAAAAAAAAAAAAAAAAAAAAAAA
✓	JAL	*Address*$_{26}$	$ra = PC + 4 PC = PC[31–28] && *Address*$_{26}$ << 2		000011AAAAAAAAAAAAAAAAAAAAAAAAAA
✓	JR	R_s	PC = R_s		000000sssss0000000000hhhhh001000
✓	JALR	R_s	$ra = PC + 4 PC = R_s		000000sssss0000011111hhhhh001001
✓	JALR	R_d, R_s	R_d = PC + 4 PC = R_s		000000sssss00000dddddhhhhh001001

Sidenote: PC

Wait, the docs said “ $ra = PC + 4”. What is PC??

PC = program counter.

This is just a fancy way of saying “the address currently being executed”.

So PC + 4 is the instruction after the current executed instruction (since an instruction is 4 bytes, if we add 4 to PC we get the start of the next instruction).

So, jal is setting $ra to the instruction immediately after that jal instruction.

This makes sense, that’s where we were putting our return labels anyway.

Basic function

Okay, using jal now.

We have a proper function!

```
main:
        jal     my_function

        jal     my_function

        li      $v0,    0
        jr      $ra




my_function:
        li      $a0,    42
        li      $v0,    1
        syscall

        jr      $ra
```

Clobbering

Function

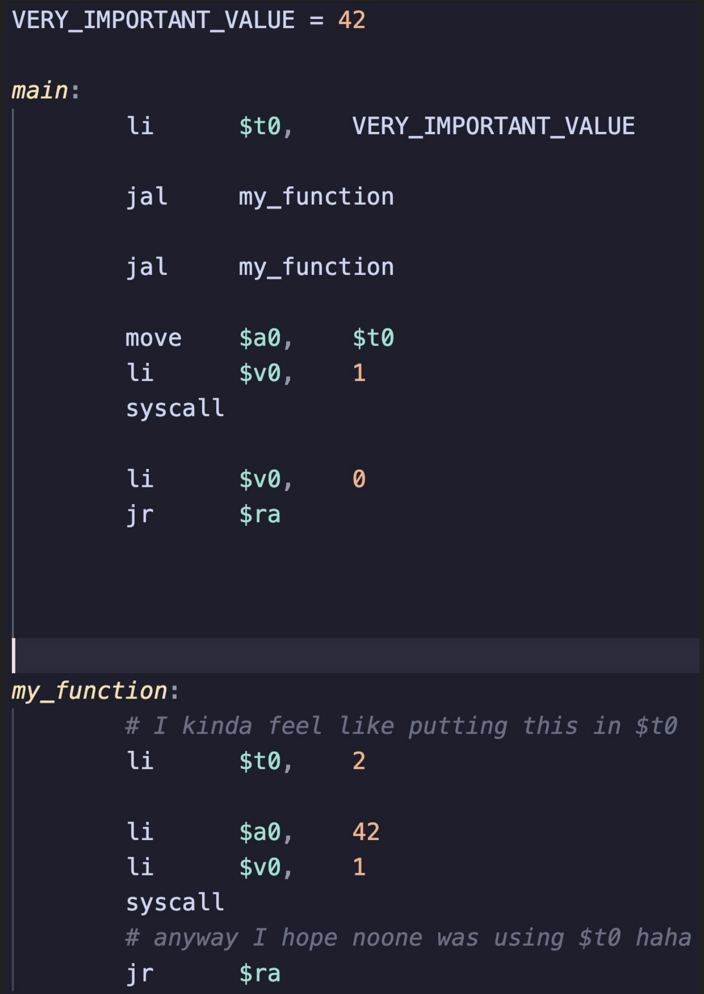
What if main was using $t0.

And my function also uses $t0.

What happens?

$t0 gets overwritten and my program breaks.

$t0 was “clobbered” by the function.



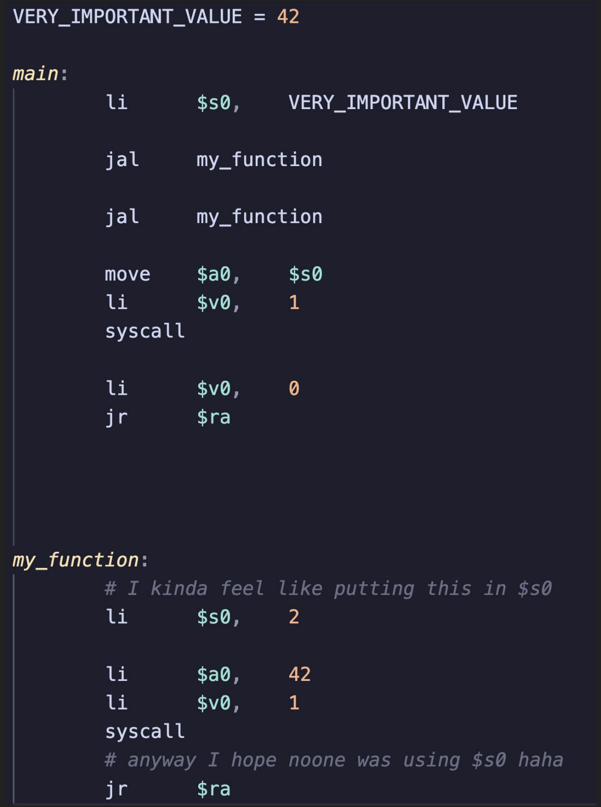
```
VERY_IMPORTANT_VALUE = 42

main:
        li      $t0,    VERY_IMPORTANT_VALUE

        jal     my_function

        jal     my_function

        move    $a0,    $t0
        li      $v0,    1
        syscall

        li      $v0,    0
        jr      $ra




my_function:
        # I kinda feel like putting this in $t0
        li      $t0,    2

        li      $a0,    42
        li      $v0,    1
        syscall
        # anyway I hope noone was using $t0 haha
        jr      $ra
```

Function

How can we fix this? Well, this technically is intended behaviour.

$t registers are “temporary” because their value is allowed to be changed by function calls.

So let’s use an $s register , a “save” register. Those registers are not allowed to be changed by function calls.



```
VERY_IMPORTANT_VALUE = 42

main:
        li      $s0,    VERY_IMPORTANT_VALUE

        jal     my_function

        jal     my_function

        move    $a0,    $s0
        li      $v0,    1
        syscall

        li      $v0,    0
        jr      $ra




my_function:
        # I kinda feel like putting this in $s0
        li      $s0,    2

        li      $a0,    42
        li      $v0,    1
        syscall
        # anyway I hope noone was using $s0 haha
        jr      $ra
```

Function

Wait, this isn’t working either? I thought $s registers would save the value?

The truth is, $s registers are just normal registers. The property of saving their value is actually created because by MIPS conventions, functions ARE NOT ALLOWED to change $s registers.

So that function was an illegal function.

```
VERY_IMPORTANT_VALUE = 42

main:
        li      $s0,    VERY_IMPORTANT_VALUE

        jal     my_function

        jal     my_function

        move    $a0,    $s0
        li      $v0,    1
        syscall

        li      $v0,    0
        jr      $ra




my_function:
        # I kinda feel like putting this in $s0
        li      $s0,    2

        li      $a0,    42
        li      $v0,    1
        syscall
        # anyway I hope noone was using $s0 haha
        jr      $ra
```

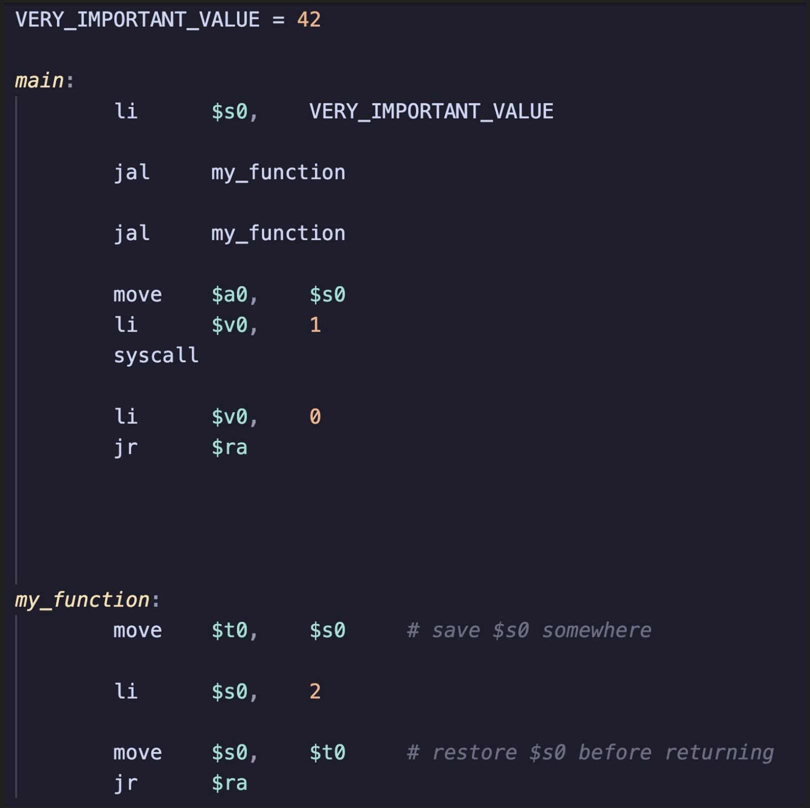
Function

But what if we want to use the $s registers in a function? But we aren’t allowed to change their value?

I have an idea, we can just save their value somewhere else, and restore it before we return.

But saving it in a register is kind of annoying, now I can’t use that register.

I think it would be easier to save it to memory.



```
VERY_IMPORTANT_VALUE = 42

main:
        li      $s0,    VERY_IMPORTANT_VALUE

        jal     my_function

        jal     my_function

        move    $a0,    $s0
        li      $v0,    1
        syscall

        li      $v0,    0
        jr      $ra




my_function:
        move    $t0,    $s0     # save $s0 somewhere

        li      $s0,    2

        move    $s0,    $t0     # restore $s0 before returning
        jr      $ra
```

Function

But where in memory? I don’t want to have to make a bunch of labels just for saving $s registers.

Enter the STACK!!!

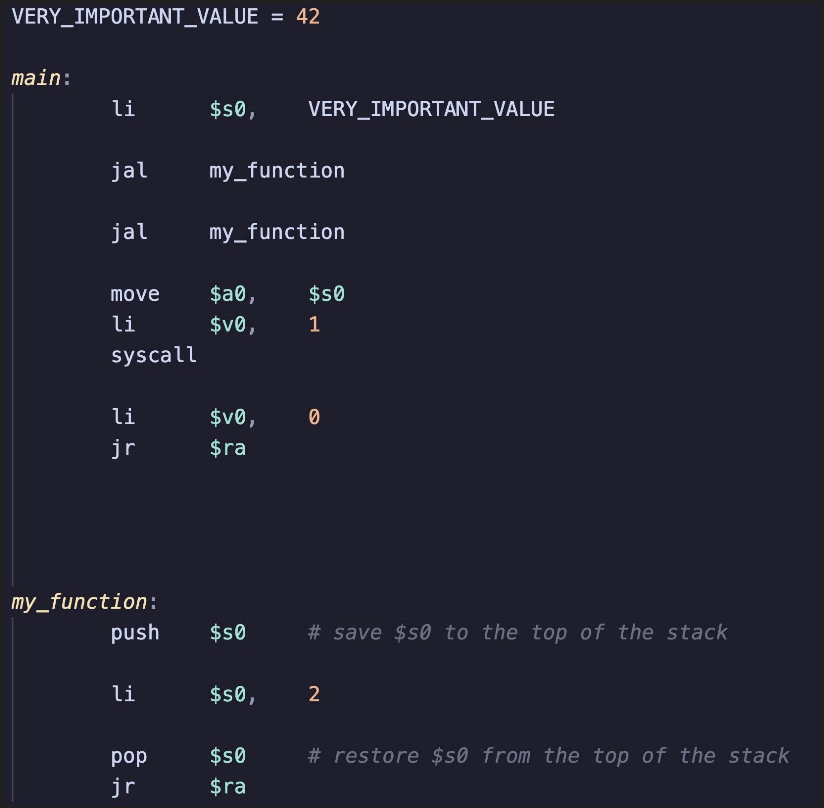
WE can save to the STACK.

And it’s so easy to do that, we can save to the top of the stack using:

push $s0

And we can restore the value from the top of the stack using:

pop $s0



```
VERY_IMPORTANT_VALUE = 42

main:
        li      $s0,    VERY_IMPORTANT_VALUE

        jal     my_function

        jal     my_function

        move    $a0,    $s0
        li      $v0,    1
        syscall

        li      $v0,    0
        jr      $ra




my_function:
        push    $s0     # save $s0 to the top of the stack

        li      $s0,    2

        pop     $s0     # restore $s0 from the top of the stack
        jr      $ra
```

Sidenote: What is the stack, exactly?

The stack is a region of memory just like .data

The stack “grows” every time we add something to the top (using push)

And it shrinks every time we take it back off (using pop)

The stack is actually implemented using the $sp register which tracks the address in memory where the top of the stack is.

Under the hood, push and pop use sw and lw instructions to save and load from the stack address (which is just an address in memory, really) in $sp, and then move $sp appropriately so it always points to the top of the stack.

For example, if I push to the stack then I should add 4 to $sp so it points to where the new “top” is.

Sidenote: the stack is upside down

The stack actually grows downwards and shrinks back up.

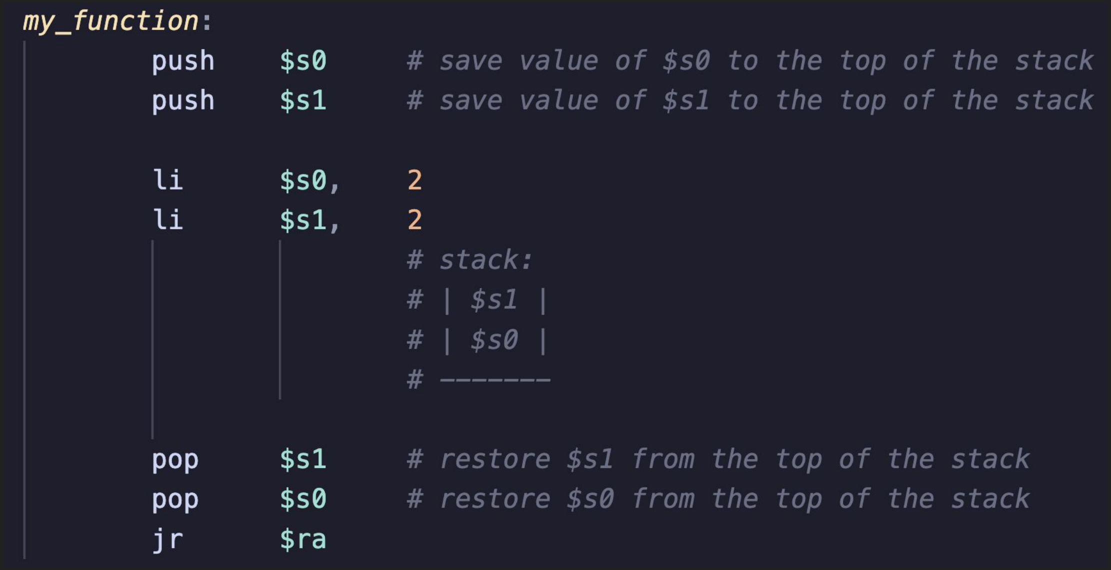
This means that when I said we would add 4 to $sp when we push, I lied.

We actually would subtract 4. But it works the same way, so if the stack being upside down confuses you, then you can just pretend it is the right way round and everything will work the same.

Function

Let’s try doing this with multiple $s registers

Notice how we pop in the opposite order, because whatever we push on the stack last is the first thing that gets taken off when we start doing pop instructions.



```
my_function:
        push    $s0     # save value of $s0 to the top of the stack
        push    $s1     # save value of $s1 to the top of the stack

        li      $s0,    2
        li      $s1,    2
                        # stack:
                        # | $s1 |
                        # | $s0 |
                        # -------

        pop     $s1     # restore $s1 from the top of the stack
        pop     $s0     # restore $s0 from the top of the stack
        jr      $ra
```

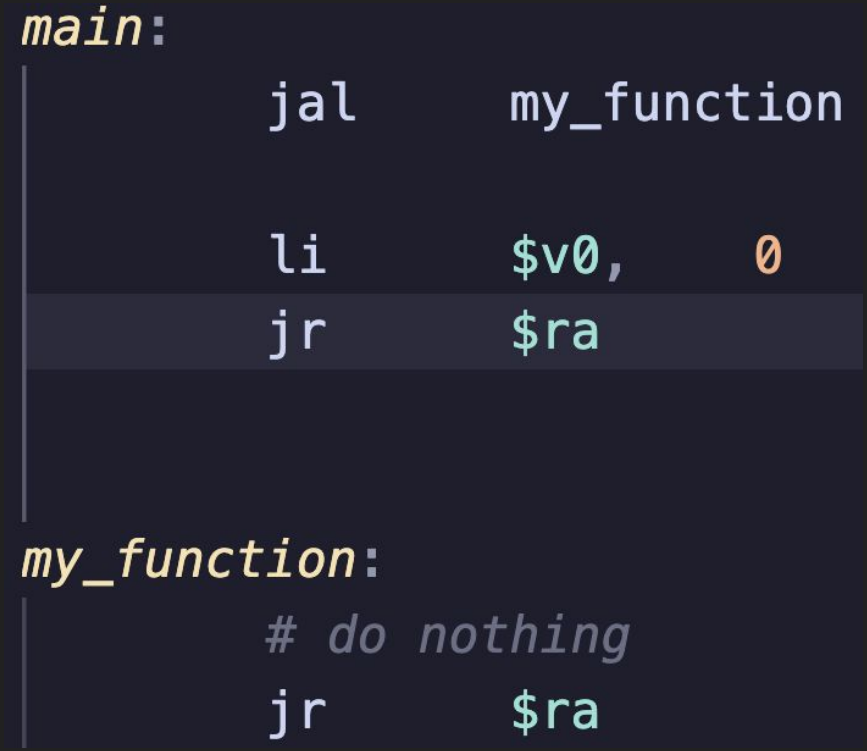
Function

There is one last thing. For some reason, my program is stuck in an infinite loop and never returns. Why???

Well, remember that $ra will contain the address that main should return to, set by the operating system when it called main with a “jal” instruction.

But, my jal instruction inside main sets $ra to something else. Main is now returning there instead.

So, in main, we need to save the original value of $ra somewhere, and probably restore it right before we return…



```
main:
        jal     my_function

        li      $v0,    0
        jr      $ra


my_function:
        # do nothing
        jr      $ra
```

Function

I know, let’s save it to the stack!

Keep in mind the reason for saving $ra to the stack is a bit different to the reason for saving $s registers.

I save $s registers so my function doesn’t mess up whoever called me.

I save $ra so my function is able to return to the correct address.

So, I save $s out of the kindness of my heart and to obey the MIPS function laws.

I save $ra because I would like to be able to return (so maybe for more selfish reasons)

Function

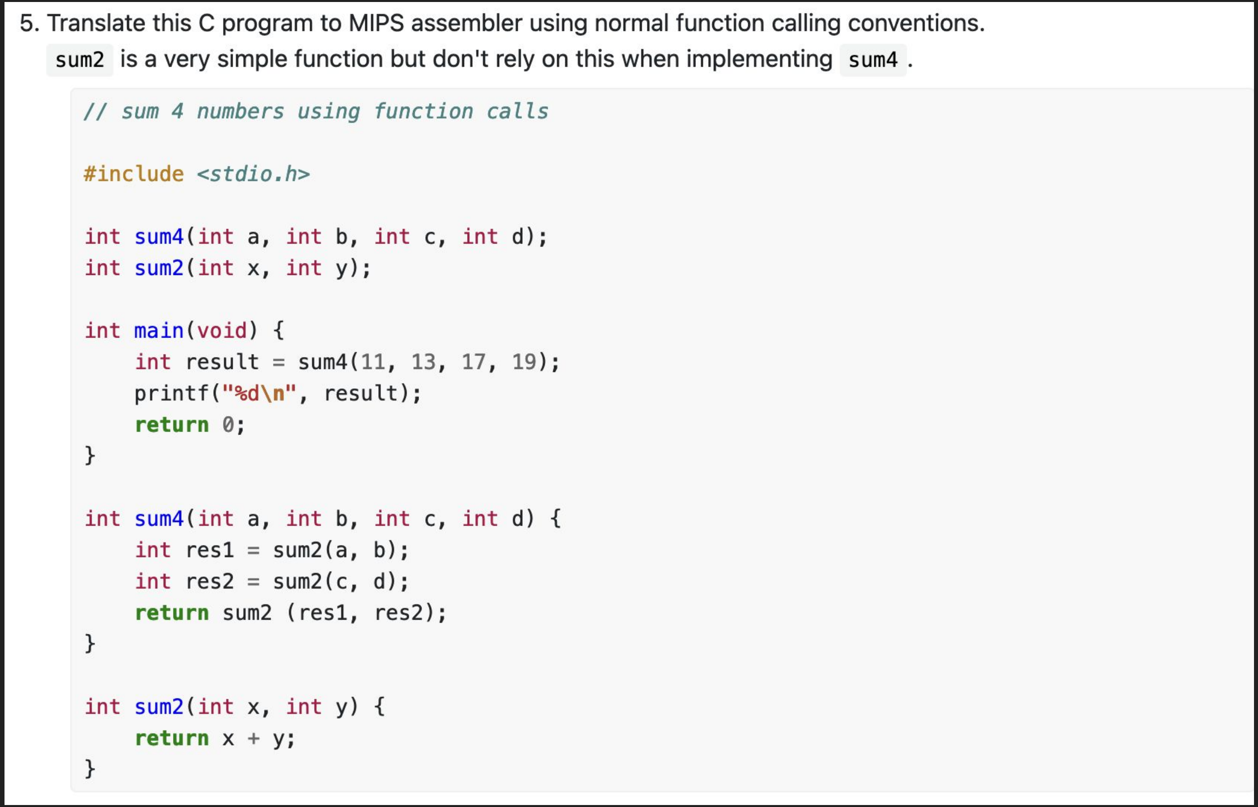
And let’s add a couple of labels to separate all this push/pop stuff from the actual function code. Then, we have a “proper” function!

See how $ra is being saved in main (because main was overwriting its original $ra value)

While $s registers are being saved in my_function to make sure that it obeys the MIPS function laws (one must not clobber $s registers)

```
main:
main__prologue:
        push    $ra     # save $ra so I can return correctly
main__body:
        jal     my_function
main__epilogue:
        pop     $ra     # restore $ra with correct address
        li      $v0,    0
        jr      $ra


my_function:
my_function__prologue:
        push    $s0
        push    $s1
        # don't need to push $ra, because there aren't any jal instructions
        # that could overwite it
my_function__body:
        li      $s0,    2
        li      $s1,    2
my_function__epilogue:
        pop     $s1
        pop     $s0
        # do nothing
        jr      $ra
```

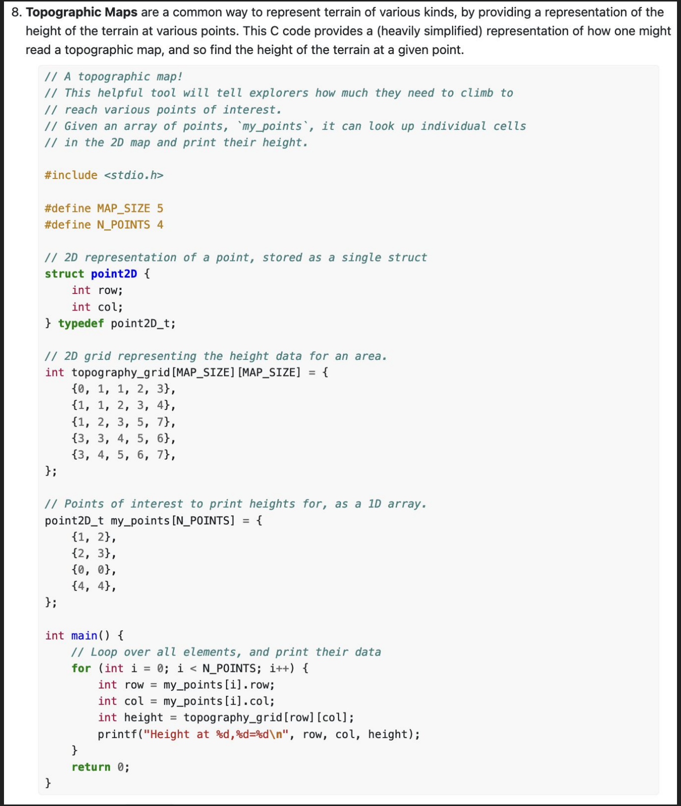
Tutorial Q5



5. Translate this C program to MIPS assembler using normal function calling conventions.
   `sum2` is a very simple function but don't rely on this when implementing `sum4`.

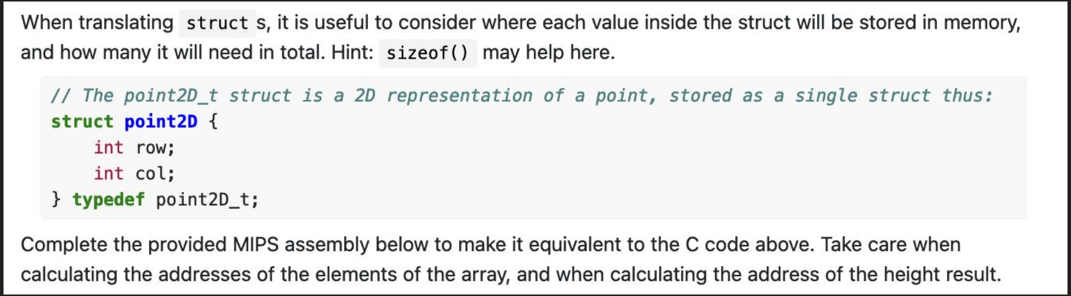
```c
// sum 4 numbers using function calls

#include <stdio.h>

int sum4(int a, int b, int c, int d);
int sum2(int x, int y);

int main(void) {
    int result = sum4(11, 13, 17, 19);
    printf("%d\n", result);
    return 0;
}

int sum4(int a, int b, int c, int d) {
    int res1 = sum2(a, b);
    int res2 = sum2(c, d);
    return sum2 (res1, res2);
}

int sum2(int x, int y) {
    return x + y;
}
```

Tutorial Q8



8. **Topographic Maps** are a common way to represent terrain of various kinds, by providing a representation of the height of the terrain at various points. This C code provides a (heavily simplified) representation of how one might read a topographic map, and so find the height of the terrain at a given point.

```c
// A topographic map!
// This helpful tool will tell explorers how much they need to climb to
// reach various points of interest.
// Given an array of points, `my_points`, it can look up individual cells
// in the 2D map and print their height.

#include <stdio.h>

#define MAP_SIZE 5
#define N_POINTS 4

// 2D representation of a point, stored as a single struct
struct point2D {
    int row;
    int col;
} typedef point2D_t;

// 2D grid representing the height data for an area.
int topography_grid[MAP_SIZE][MAP_SIZE] = {
    {0, 1, 1, 2, 3},
    {1, 1, 2, 3, 4},
    {1, 2, 3, 5, 7},
    {3, 3, 4, 5, 6},
    {3, 4, 5, 6, 7},
};

// Points of interest to print heights for, as a 1D array.
point2D_t my_points[N_POINTS] = {
    {1, 2},
    {2, 3},
    {0, 0},
    {4, 4},
};

int main() {
    // Loop over all elements, and print their data
    for (int i = 0; i < N_POINTS; i++) {
        int row = my_points[i].row;
        int col = my_points[i].col;
        int height = topography_grid[row][col];
        printf("Height at %d,%d=%d\n", row, col, height);
    }
    return 0;
}
```



When translating `struct`s, it is useful to consider where each value inside the struct will be stored in memory, and how many it will need in total. Hint: `sizeof()` may help here.

```c
// The point2D_t struct is a 2D representation of a point, stored as a single struct thus:
struct point2D {
    int row;
    int col;
} typedef point2D_t;
```

Complete the provided MIPS assembly below to make it equivalent to the C code above. Take care when calculating the addresses of the elements of the array, and when calculating the address of the height result.

(Time permitting)

And time will probably not permit.