COMP1521

TUTORIAL 4

MIPS 2D ARRAYS -> MIPS STRUCTS-> MIPS FUNCTIONS

NOTE:

- Remember assignment 1 is due next Friday!

- Week 3 weekly test is due this Thursday!
  - Week 4 weekly test typically is released after this due date

MIPS 2D ARRAYS

MIPS 2D Arrays

- From last week we already know how 1D arrays work, and we can use this basis to help understand 2D arrays in memory.
  - Think of a 2D array as an array of 1D arrays
  - Consider the initialisation of `char array[3][5]`
    - If we index once we index into any of the 3 arrays of length 5, e.g array[1] would be the address where the 2nd 1D array begins (array + 5 in this case)

Indexing to Individual Elements in 2D Arrays

char array[3][5]

num of ID arrays(rows) length of each array (cols)

- So how do we now get individual elements?
  - We find the array an element is in that is taking the row (first index) and multiplying it by the size of each array in our case 5 (number of cols)
  - To index into array[2][3]
    - We have size of each 1D array (5) * row (2) for finding our array
    - We have size of each element (1) * col (3) for finding our element
    - Then we add this to our address (array)
    - array + (5 * 2) + (1 * 3) to find our elements address

array + 0	0	1	2	3	4
array + 5	5	6	7	8	9
array + 10	10	11	12	13	14

Indexing to Individual Elements in 2D Arrays

char array[3][5]

num of 1D arrays(rows) length of each array (cols)

- So how do we now get individual elements?
  - We find the array an element is in that is taking the row (first index) and multiplying it by the size of each array in our case 5 (number of cols)
  - To index into array[2][3]
    - We have size of each 1D array (5) * row (2) for finding our array
    - We have size of each element (1) * col (3) for finding our element
    - Then we add this to our address (array)
    - array + (5 * 2) + (1 * 3) to find our elements address (X)

array + 0	0	1	2	3	4
array + 5	5	6	7	8	9
array + 10	10	11	12	X	14

Indexing to Individual Elements in 2D Arrays

char array[3][5]

array + 0	0	1	2	3	4
array + 5	5	6	7	8	9
array + 10	10	11	12	13	14

- These arrays are not exactly stored how we visualise them in memory, they are like one big array, and we use our offsets to index in correctly into them.
  - So, the array below is what it resembles in memory, however it is much simpler to visualise 2D arrays as above

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

array + 0
array[0][0]

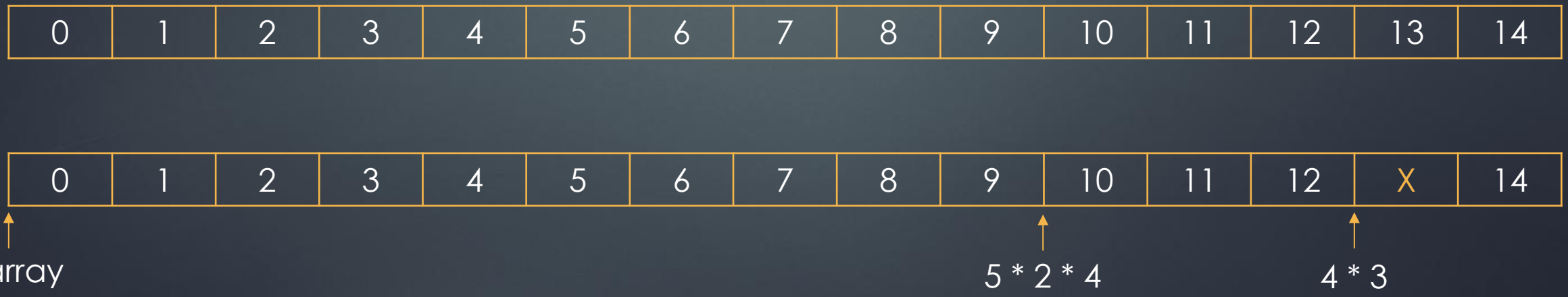
array + 5
array[1][0]

array + 10
array[2][0]

What about different element sizes?

int array[3][5]

- As we know an int is 4 bytes, we simply just multiply our offsets during our calculations by 4
  - Consider the same example of array[2][3]
    - We have size of each 1D array (5) * row (2) * size of element (4) for finding our array
    - We have size of each element (4) * col (3) for finding our element
    - Then we add this to our address (array)
    - array + (5 * 2 * 4) + (4 * 3) to find our elements address (array + 52)
  - Quick Ref: array + (row * num_of_cols * size_of_element) + (col * size_of_element)



Example: q2.s

- This program prints out a Danish flag; by looping through the columns and rows of the flag, each element is a single character (1 byte), translate the program into MIPS!

```c
#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");
    }
}
        i++;
    }
}
```

MIPS STRUCTS

MIPS Structs

- We know from C, structs are a type that can contain many different types of variables, and as registers only fit 4 bytes, we generally need to store our structs in memory
- We can create our structs by *sort of* having each field laid out next to each other in memory and we calculate our offsets into each field of the struct (like how we do for arrays)
  - Keep in mind some types (such as ints, 4-byte) require specific alignment so some bytes in memory may be padded to account for this

```
struct Student {
  char[20] name;
  char[9] zid;
  int program;
  double wam;
}
```

Could we change the struct to not waste/pad 7 bytes?

Indexing into Structs

- Consider we had a struct Employee

```
struct Employee {
  int id;
  int wage;
}
```

```
    .data
employee:
        .word   1234567     # id
        .word   100100      # wage
```

- Now to index into structs we will typically use #defines that will map out the offsets for each field in the struct, in this case we would have:
  - EMPLOYEE_ID_OFFSET = 0
  - EMPLOYEE_WAGE_OFFSET = 4
  - To find the employee wage we would do employee + EMPLOYEE_WAGE_OFFSET

Array of Structs

struct Employee[3]

- Now array of structs works quite similarly, however we just need the size of each struct to calculate our offset into each struct in the array

```
employees:
        .word   1234567, 100100        # employees[0]
        .word   7654321, 40012         # employees[1]
        .word   2135123, 0             # employees[2]
```

- As the employee struct is the same we would just add one new #define to our previous two:
  - SIZE_OF_EMPLOYEE = SIZE_OF_INT * 2
  - EMPLOYEE_ID_OFFSET = 0
  - EMPLOYEE_WAGE_OFFSET = 4
- How do we index into employees[2].wage to update their wage? (index_struct.s)

(SIZE_OF_EMPLOYEE * INDEX) + EMPLOYEE_WAGE_OFFSET

Example (q7)

For each of the following struct definitions, what are the likely offset values for each field, and the size of the struct

```
struct _coord {
    double x;
    double y;
};
```

Field	Offset
double x	
double y	
Size of _coord	

```
struct _node {
    int value;
    Node *next;
};
```

Field	Offset
int value	
Node *next	
Size of _node	

Example (q7)

For each of the following struct definitions, what are the likely offset values for each field, and the size of the struct

```
struct _coord {
    double x;
    double y;
};
```

Field	Offset
double x	0
double y	8
Size of _coord	**16 bytes**

```
struct _node {
    int value;
    Node *next;
};
```

Field	Offset
int value	0
Node *next	4
Size of _node	**8 bytes**

Example (q7)

For each of the following struct definitions, what are the likely offset values for each field, and the size of the struct

```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
};
```

Field	Offset
int stu_id	
char course[9]	
char term[5]	
char grade[3]	
double mark	
Size of _enrolment	

```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
};
```

Field	Offset
int nitems	
int head	
int term[5]	
int grade[3]	
Item *items	
Size of _queue	

Example (q7)

For each of the following struct definitions, what are the likely offset values for each field, and the size of the struct

```
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
};
```

Field	Offset
int stu_id	0
char course[9]	4
char term[5]	13
char grade[3]	18
double mark	24
Size of _enrolment	**32 bytes**

```
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
};
```

Field	Offset
int nitems	0
int head	4
int term[5]	8
int grade[3]	12
Item *items	16
Size of _queue	**20 bytes**

MIPS FUNCTIONS

MIPS Functions

- We know in C functions optionally take in arguments, execute their functionality, and optionally return a value? But how do we do this in MIPS with just labels and instructions

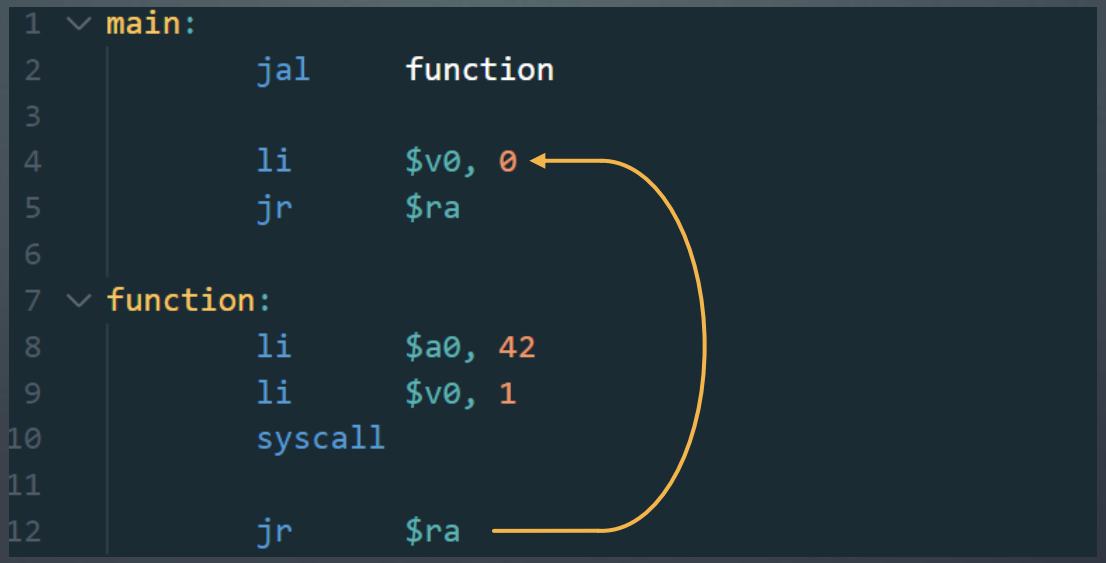
- Our intuition might be to just branch to a label that has the functionality we wish to run, but how do we get back? Do we need a return label just to call a function?

- Something like this isn't extensible , what if we want to call it from another place too

```
1   main:
2   return_location:
3           j       function
4
5   function:
6           li      $a0, 42
7           li      $v0, 1
8           syscall
9
10          j return_location
```

MIPS Functions

- Additionally, you might think to use a register to store the return labels address before we call the function, that's almost it.
  - We have an instruction that will do this functionality for us, and eliminate the need for extra labels, outside of the function itself
  - JAL – jump and link
    - This takes the current program counter, and stores PC + 4 in the $ra register, this means when this instruction is executed the address of the next instruction in memory is in $ra
    - And as we've seen before we can use jr $ra to move execution to the address provided in $ra
  - We're not quite done yet; there is a major issue with this we'll learn about soon!



Clobbering

- Consider our main function is holding a value in $t0 that it wants to use later, but another function we call make alters the value in this register without us realising
  - This is called clobbering, when a function overwrites the value in a temporary register, we say that register ($t0) was clobbered by the function
- How can we avoid this? Using $s registers? Yes, but not just that as these two programs, will have identical functionality despite using $s registers, the value is still clobbered because we aren't following the MIPS conventions

Using $t registers

```
main:
        li      $t0, 5          # hold this here
        jal     square_5        # square_5()

        move    $a0, $t0        # print(5)
        li      $v0, 1
        syscall

        li      $v0, 0          # return 0
        jr      $ra

square_5:
        li      $t0, 5          # int $t0 = 5 * 5
        mul     $t0, $t0, 5

        move    $a0, $t0        # print($t0)
        li      $v0, 1
        syscall

        jr      $ra
```

Using $s registers

```
main:
        li      $s0, 5          # hold this here
        jal     square_5        # square_5()

        move    $a0, $s0        # print(5)
        li      $v0, 1
        syscall

        li      $v0, 0          # return 0
        jr      $ra

square_5:
        li      $s0, 5          # int $s0 = 5 * 5
        mul     $s0, $s0, 5

        move    $a0, $s0        # print($s0)
        li      $v0, 1
        syscall

        jr      $ra
```

MIPS Conventions ($s registers)

- Turns out $s registers aren't magically saved for us, we must follow conventions that ensure the value in the $s registers is what we expect even after function calls, where they may be altered, this is done by restoring the value.
  - We do this by moving the value somewhere else
    - Another register? No. You'll just run into the same issue with subsequent function calls
    - Into data section? No. You'll just run into the same issue with subsequent function calls
    - The stack? Yes.

```
main:
        li      $s0, 5          # hold this here
        jal     square_5        # square_5()

        move    $a0, $s0        # print(5)
        li      $v0, 1
        syscall

        li      $v0, 0          # return 0
        jr      $ra

square_5:
        sw      $s0, ($sp)      # put $s0 on the stack
        sub     $sp, 4          # update sp for next time

        li      $s0, 5          # int $s0 = 5 * 5
        mul     $s0, $s0, 5

        move    $a0, $s0        # print($s0)
        li      $v0, 1
        syscall

        jr      $ra
```

- When any function, including main, uses a $s register it must save the value to the stack, and once finished place the value back in the $s register
- We have the $sp register which points to the next unused section on the stack, and update $sp for the next use

Pushing registers to the stack

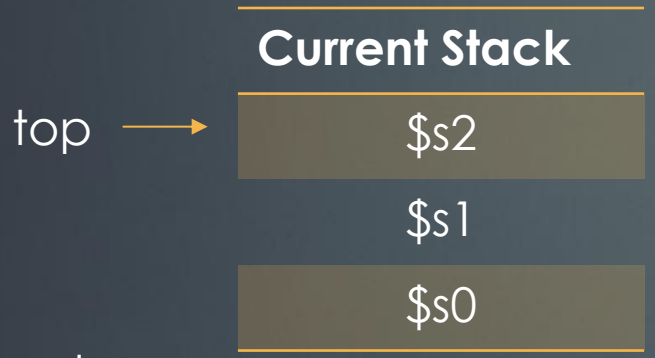
- Also managing the stack manually is a bit of a pain, especially when you start pushing more and more, fortunately mipsy has some pseudo-instructions to improve this
  - push $register – pushes the register to the stack + updates sp for us
  - pop $register – pops the value off the stack into the register pushes the register + updates sp for us

```
main:
        li      $s0, 5          # hold this here
        jal     square_5        # square_5()

        move    $a0, $s0        # print(5)
        li      $v0, 1
        syscall

        li      $v0, 0          # return 0
        jr      $ra

square_5:
        push    $s0             # puts $s0 on the top stack

        li      $s0, 5          # int $s0 = 5 * 5
        mul     $s0, $s0, 5

        move    $a0, $s0        # print($s0)
        li      $v0, 1
        syscall

        pop     $s0             # pops value from top
                                # of stack into $s0
        jr      $ra
```

Pushing multiple registers to the stack

- As the name suggests, the stack when you push to it, places each value you can imagine on top of each other (stacked).
- Consider the example on the right, if we were to pop the values in order, rather than reverse, our stack looks like this

```
main:
        jal     function        # function()

        li      $v0, 0          #return 0
        jr      $ra

function:
        push    $s0             # push $s0 on the top stack
        push    $s1             # push $s1 on the top stack
        push    $s2             # push $s2 on the top stack

        li      $s0, 42
        li      $s1, 42
        li      $s2, 42

        pop     $s2             # pop top of stack into $s2
        pop     $s1             # pop top of stack into $s1
        pop     $s0             # pop top of stack into $s0

        jr      $ra             # return to main
```



In order

register	value received
$s0	$s2
$s1	$s1
$s2	$s0

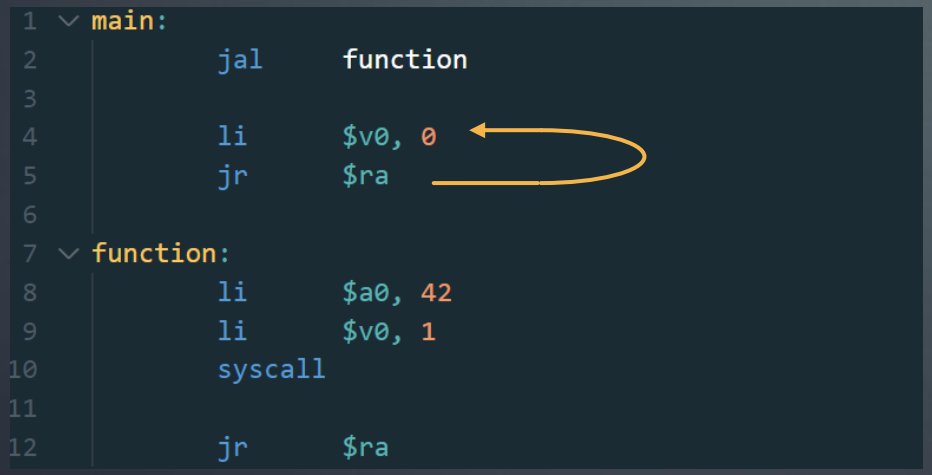
Reverse order

register	value received
$s2	$s2
$s1	$s1
$s0	$s0

Note: Registers must always be popped in reverse order to ensure the data is correctly restored

Going back to our first function

- You'll notice if we run this program, we have an infinite loop. Why?
  - Because when we use jal, it overwrites mains return address in $ra
    - This is set by the operating system when main is called
  - So, every time we call jr $ra in main, it goes back to the instruction after our jal function call
- Like $s registers we can save $ra to the stack, however this is to ensure functions have their correct return address rather than ensure validity of data (MIPS conventions)
- We only push and pop $ra if a function calls another function, if it does not call another function (leaf function) it does not have to.



```
main:
        jal     function

        li      $v0, 0
        jr      $ra

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

        jr      $ra
```

```
main:
        push    $ra

        jal     function

        pop     $ra
        li      $v0, 0
        jr      $ra

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

        jr      $ra
```

What our functions should look like

- We have 4 labels in every function we write:
  - function name (main)
  - prologue (main__prologue)
    - Contains all the operations involving the stack (push)
  - body (main__body)
    - Your functions main body
  - epilogue (main__epilogue)
    - Contains the operations involving popping from the stack
    - Also includes the return (jr $ra)

```
main:
main__prologue:
        push    $ra             # as not a leaf function push $ra
main__body:
        jal     function        # function()
main__epilogue:
        pop     $ra             # pop return address off stack
        li      $v0, 0          # return 0
        jr      $ra

function:
function__prologue:
        # no $ra as this is a leaf function (no jal instructions)
        push    $s0             # push $s0
function__body:
        li      $s0, 2
function__epilogue:
        pop     $s0             # restore $s0

        jr      $ra             # return
```

Function Arguments

- If we have a function that has arguments, we can pass up to 4 arguments to a function in MIPS using the $a registers
  - Consider function(int a)
  - We would put our int into $a0, before we jal
    - If there is more than one argument, you would put the second in $a1, and so on
  - We don't ask you to call any functions with more than 4 arguments

void function(int a) {}

```
main:
        li      $a0, 42
        jal     function
```

Return Values

- Maybe you've realised from when we return from main, that $v0 holds our return value
  - A function int function(), would return an integer, in MIPS this would be returned in $v0

```
int function() {
    return 42;
}
```

```
main__body:
        jal     function        # function()
        move    $t0, $v0        # move our returned value
```

```
function__epilogue:
        li      $v0, 42
        jr      $ra             # return
```

Note: If a value is too large for one register, $v1 also exists, however you are generally not asked to use this in the course

Assignment Style (quick reference)

- Function Comments
  - Frame: All registers pushed to stack
  - Uses: All registers used in the function
  - Clobbers: All temporary registers used and not restored ($t, $a, $v)
  - Locals: All registers used and their uses ($t0 – int i)
  - Structure: All labels used, and their structure indented like C
- Labels
  - Have descriptive names and consistent structure, follow structure taught for loops (always starts with function name)
  - e.g function_name__body_loop_init -> function_name__body_loop_cond
- Indenting
  - Labels never indented
  - Instructions always indented once (8 size tabs)
- Whitespace
  - Be consistent with vertical whitespace, if you leave a line between an instruction and label, always do that, or the inverse
  - Each C line *usually* is a block of MIPS, vertical whitespace in between each logical block
  - Don't randomly have vertical spaces of more than one between blocks of code
- In-line Comments
  - Put C code from assignment beside each block of MIPS
- zID, Name, Description
  - EASIEST MARK EVER DON'T FORGET THIS
- Line Length
  - Keep line length less than 120, unless unavoidable but keep it a one-time thing
- NOTE: Check q3.s sum4 function for an expected style

Example: q3.s

- Translate this C program to MIPS, that sums four numbers together using MIPS calling conventions

```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;
}
```

MIPS code is getting too large for slides, check the q3.s file on the github repo for the solution!

FIN