

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 and is due today 9pm

W4 weekly test will be released today

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 (about one week from now).

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

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.

An array of 1D arrays.

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).

```
arr[5][10] // An array of 5 arrays
```

```
arr + 00  _ _ _ _ _ _ _ _ _ _  
arr + 10  _ _ _ _ _ _ _ _ _ _  
arr + 20  _ _ _ _ _ _ _ _ _ _  
arr + 30  _ _ _ _ _ _ _ _ _ _  
arr + 40  _ _ _ _ _ _ _ _ _ _
```

How can you calculate the address?

For `arr[3][2]`

Size of big element * i to go
past i many 1D arrays = +30

Size of single element * j to go
past j single elements = +2

So go to `arr + 30 + 2`

```
arr[5][10] // An array of 5 arrays
```

```
arr + 00 _ _ _ _ _ _ _ _ _ _  
arr + 10 _ _ _ _ _ _ _ _ _ _  
arr + 20 _ _ _ _ _ _ _ _ _ _  
arr + 30 _ _ x _ _ _ _ _ _ _  
arr + 40 _ _ _ _ _ _ _ _ _ _
```

```
arr[5][10] // An array of 5 arrays
```

```
arr + 00 _ _ _ _ _ _ _ _ _ _ arr + 10 _ _ _ _ _ _ _ _ _ _ arr + 20 _ _ _ _ _ _ _ _ _ _ arr + 30 _ _ x _ _ _ _ _ _ _ arr + 40 _ _ _ _ _ _ _ _ _ _  
|                                                                 |  
----- +10 ----- +10 ----- +10 +1 +1|
```

What about 2D arrays of ints?

`arr[3][2]:`

Go past i many big elements (4 byte
int + 10 elements in 1D array = 40) =
 $+3 * 40 = +120$

Go past j many single elements (4
byte int) = $+2 * 4 = +8$

So go to `arr + 120 + 8`

`arr[5][10] // An array of 5 arrays`

<code>arr + 00</code>	— — — — — — — — — —
<code>arr + 40</code>	— — — — — — — — — —
<code>arr + 80</code>	— — — — — — — — — —
<code>arr + 120</code>	— — x — — — — — — —
<code>arr + 160</code>	— — — — — — — — — —

`arr[5][10] // An array of 5 arrays`

<code>arr + 00</code>	— — — — — — — —	<code>arr + 40</code>	— — — — — — — —	<code>arr + 80</code>	— — — — — — — —	<code>arr + 120</code>	— — — — — — — —	x	— — — — — — — —	<code>arr + 160</code>	— — — — — — — —
----- +40		----- + 80		----- + 120		+4 +4					

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:

$$\text{arr} + \text{size} * y * i + \text{size} * j$$

Tutorial Q2

2. Translate this C program to MIPS assembler.

```
// 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;
};
```

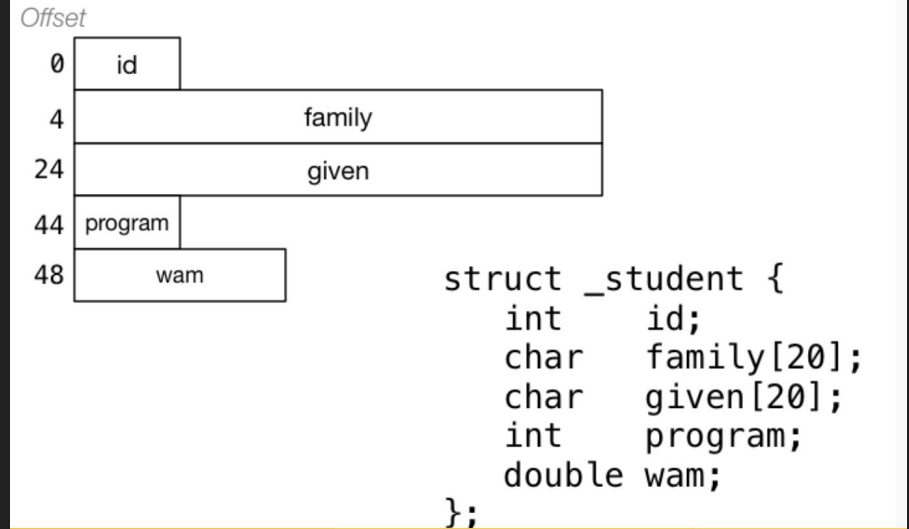
How to make a struct

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



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_FIRST_INT_OFFSET = 0 # (0 bytes from the start)
```

```
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

Array of structs?

Each element has a size equal to the size of the struct.

Index to a specific struct, then add the desired offset.

To access `arr[2].y`, first access `arr[2]`

`arr + 16`

Then add the offset for the second field

```
arr[4] // An array of 4 structs
```

```
arr + 0  ____  ____  arr + 8  ____  ____  arr + 16  ____  yyyy  arr + 24  ____  ____  
|                                               |  
----- + 8 ----- + 16 - +4 -|
```

```
arr[4] // An array of 4 structs
```

```
arr + 0  ____  ____
```

```
arr + 8  ____  ____
```

```
arr + 16 ____  ____
```

```
arr + 24 ____  ____
```

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.

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

b.

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

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.

```
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

What's a function exactly?

Takes some input (optionally)

Does something

Makes some output (optionally)

Basic function

Here is the basic idea of a function

We jump to a label that does something

```
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

How about we store it in the \$ra register?

After all, we are storing a “return address”

```
main:
    la    $ra,    return_location_1
    j     my_function
return_location_1:
    la    $ra,    return_location_2
    j     my_function
return_location_2:
    li    $v0,    0
    jr    $ra

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

    jr    $ra
```


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 \$ra?

Maybe even we can do this, without the need for a label to return to!

BLTU	$R_s, R_t, \text{Offset}_{16}$	IF $R_s < R_t$ THEN PC += $\text{Offset}_{16} \ll 2$	UNSIGNED COMPARISON SLTU \$at, R_s, R_t BNE \$0, \$at, Offset_{16}
BLTU	$R_s, \text{Imm}, \text{Offset}_{16}$	IF $R_s < \text{Imm}$ THEN PC += $\text{Offset}_{16} \ll 2$	UNSIGNED COMPARISON pseudo-instruction
✓ BLTZ	R_s, Offset_{16}	IF $R_s < 0$ THEN PC += $\text{Offset}_{16} \ll 2$	000001sssss00000000000000000000
BLE	$R_s, R_t, \text{Offset}_{16}$	IF $R_s \leq R_t$ THEN PC += $\text{Offset}_{16} \ll 2$	SLT \$at, R_t, R_s BEQ \$0, \$at, Offset_{16}
BLE	$R_s, \text{Imm}, \text{Offset}_{16}$	IF $R_s \leq \text{Imm}$ THEN PC += $\text{Offset}_{16} \ll 2$	pseudo-instruction
BLEU	$R_s, R_t, \text{Offset}_{16}$	IF $R_s \leq R_t$ THEN PC += $\text{Offset}_{16} \ll 2$	UNSIGNED COMPARISON SLTU \$at, R_t, R_s BEQ \$0, \$at, Offset_{16}
BLEU	$R_s, \text{Imm}, \text{Offset}_{16}$	IF $R_s \leq \text{Imm}$ THEN PC += $\text{Offset}_{16} \ll 2$	UNSIGNED COMPARISON pseudo-instruction
✓ BLEZ	R_s, Offset_{16}	IF $R_s \leq 0$ THEN PC += $\text{Offset}_{16} \ll 2$	000110sssss00000000000000000000
✓ J	Address_{26}	PC = PC[31-28] && $\text{Address}_{26} \ll 2$	000010AAAAAAAAAAAAAAAAAAAAAAAA
✓ JAL	Address_{26}	$\$ra = PC + 4$ PC = PC[31-28] && $\text{Address}_{26} \ll 2$	000011AAAAAAAAAAAAAAAAAAAAAAAA
✓ JR	R_s	PC = R_s	000000sssss00000000000000000000
✓ JALR	R_s	$\$ra = PC + 4$ PC = R_s	000000sssss00000111110000000000
✓ JALR	R_d, R_s	$R_d = PC + 4$ PC = R_s	000000sssss00000000000000000000

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 original 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

We could save it to the data segment?

But this won't really work - what if we call another function from my_function, and need to save \$s0 again?

```
VERY_IMPORTANT_VALUE = 42
```

```
main:
```

```
    li      $s0, VERY_IMPORTANT_VALUE
```

```
    jal my_function
```

```
    move    $a0, $s0
```

```
    li      $v0, 1
```

```
    syscall
```

```
    li      $v0, 0
```

```
    jr      $ra
```

```
my_function:
```

```
    sw      $s0, save_s0
```

```
    li      $s0, 2
```

```
    lw      $s0, save_s0
```

```
    jr      $ra
```

```
.data
```

```
save_s0:
```

```
.word 0
```

Function

Enter a new region of memory: the stack.

\$sp register points to the next unused part of the stack

Save the value there, and update \$sp for next person to use it.

```
VERY_IMPORTANT_VALUE = 42
```

```
main:
```

```
    li    $s0, VERY_IMPORTANT_VALUE
```

```
    jal   my_function
```

```
    move  $a0, $s0
```

```
    li    $v0, 1
```

```
    syscall
```

```
    li    $v0, 0
```

```
    jr    $ra
```

```
my_function:
```

```
    sw    $s0, ($sp)
```

```
    sub   $sp, $sp, 4 # move sp down for next time we save
```

```
    li    $s0, 2
```

```
    add   $s0, $s0, 4 # move sp back to where we saved it
```

```
    lw    $s0, ($sp)
```

```
    jr    $ra
```

Function

But moving \$sp round is a real pain..

Fortunately, mipsy introduces some awesome pseudo-instructions (which are common features in modern assembly languages) that we can use instead!

push \$s0 saves \$s0 value onto the top of the stack and moves \$sp for us

pop \$s0 stores top value of the stack into \$s0 and moves \$sp for us

```
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
```

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 overwrite it
my_function__body:
    li      $s0,   2
    li      $s1,   2
my_function__epilogue:
    pop     $s1
    pop     $s0
    # do nothing
    jr      $ra
```

Function

Pass arguments?

We can just leave them in the \$a registers

```
main:  
    li    $a0,    1           # pass arg 1 as the integer 1  
    la    $a1,    my_str      # pass arg 2 as the string my_str  
    jal   my_function
```


Function

Return value?

We can put it into \$v registers (\$v0)

```
main:
```

```
    jal    my_function
```

```
    move   $t0, $v0
```

```
    # get return value, put it in $t0
```

Function

Function comments?

Frame: (All registers saved to the stack)

Uses: (All registers modified by function)

Clobbers: (All registers modified by function AND not saved to the stack)

Locals: (All registers in uses with a description of their use)

Structure: (All labels in the function indented to show structure)

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`.

```
// 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.

```
// 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.

```
// 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.