# COMP1521 25T1

## Week 3 Lecture 1

# MIPS FUNctions

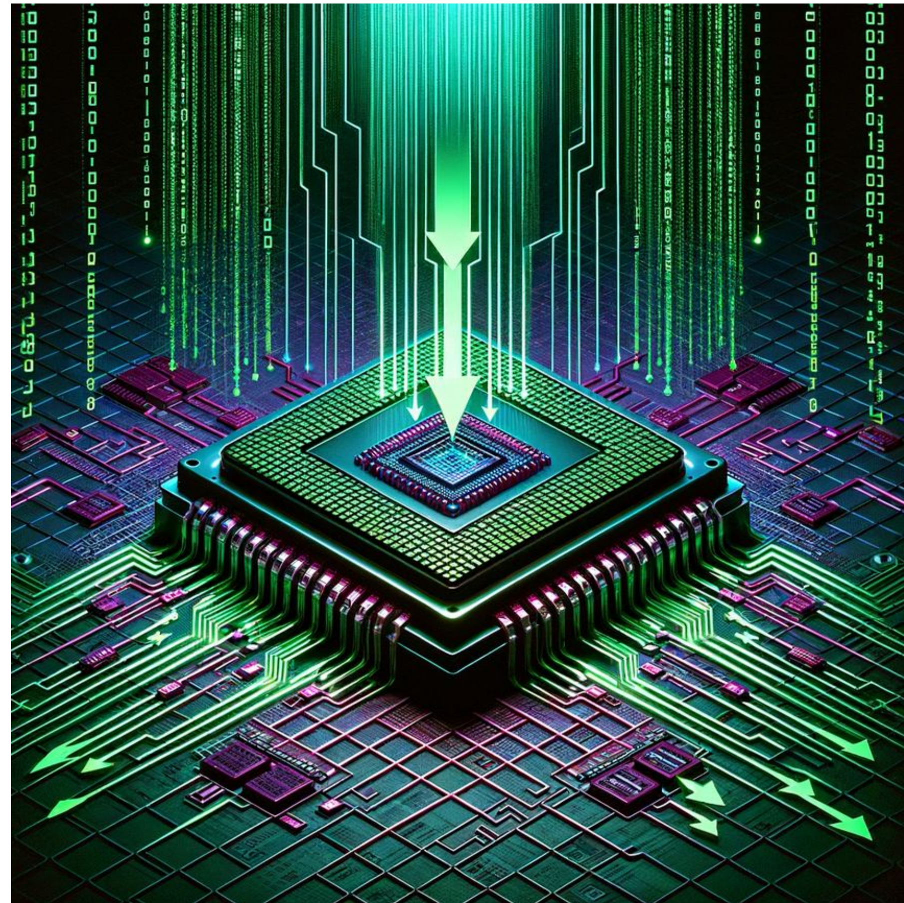
Adapted from Abiram Nadarajah, Hammond Pearce, Andrew Taylor and John Shepherd's slides

# Today's Lecture

- Recap last lecture
  - Arrays
  - structs
- Functions

# Wednesday's Lecture

We have 2 lectures a week.
Wednesday 2-4pm
live streamed on Youtube.



# Help Sessions and other admin

- Help Session Schedule is out
  - Sessions starting on monday!
- Extra practice code for each topic:
  https://cgi.cse.unsw.edu.au/~cs1521/25T1/topic/mips_data/code
  etc
- Assignment 1 out later this week

# Weekly Tests start this week

**Released:** Thursday 3pm

**Time limit:** 1 hour

**Due:** Thursday Week 4 at 3pm. (And then another test comes out)

Submitted via **give**

**You can get 50% max for questions submitted after the hour is up**

**Topic for week 3 test:** MIPS basics, control.

Can use mips documentation

# Arrays of 1 byte elements (array.c demo)

char a[5] = {‘a’, ‘z’, ‘b’, ‘f’, ‘G’};

| a[0] | a[1] | a[2] | a[3] | a[4] |
|---|---|---|---|---|
| 'a' | 'z' | 'b' | 'f' | 'G' |
| 0x100 | 0x101 | 0x102 | 0x103 | 0x104 |

- If we have the address of the start of the array:
  - How can I work out the address of the a[3]?
  - How can I work out the address of the a[i]?

# Arrays of 4 byte elements

```
int a[5] = {16, 4, 1, 9, 2};
```

| a[0] | a[1] | a[2] | a[3] | a[4] |
| --- | --- | --- | --- | --- |
| 16 | 4 | 1 | 9 | 2 |
| 0x100 | 0x104 | 0x108 | 0x10c | 0x110 |

- If we have the address of the start of the array:
    - How can I work out the address of the a[3]?
    - How can I work out the address of the a[i]?

# Address of Array Elements

char array: address of a[i] = address of a + i

integer array: address of a[i] = address of a + (i * 4)

In general:

address of element = address of array + index * sizeof(element)

# 2D Arrays in MIPS

| | 0 | 1 | 2 | 3 | < col |
|---|---|---|---|---|---|
| 0 | a | b | c | d | |
| 1 | e | f | g | h | |
| 2 | i | j | k | l | |

^ row

**Offset of start of relevant row:**

(row * N_COLS) * sizeof(element)

**Offset within row:**

col * sizeof(element)

**Total offset:**

(row * N_COLS + col) * sizeof(element)

| a | b | c | d | e | f | g | h | i | j | k | l |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

# Recap: MIPS array coding examples

read_array_words.c
modify_2d.c

Try these for an exercise:
array_words_pointer.c
array_bytes_pointer.c

# Recap Structs: recap_struct.c

```
struct person{
    char first_initial;
    char last_initial;
    int age;
};
```

What size will this struct be?

What offsets will each field have?

Let's write code in MIPS to create a global variable of this type

Read in data and print it out again.

# Recap Structs: recap_struct.c

```c
struct person{
    int age;
    char first_initial;
    char last_initial;
};
```

What if we do it like this?

# Structs

```
struct student {
    int zid;
    char first[20];
    char last[20];
    int program;
    char alias[10];
}
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| zID (4) | 5308310 | | | | | | | | | |
| first (20) | A | b | i | r | a | m | \0 | | | |
| last (20) | N | a | d | a | r | a | j | a | h | \0 |
| program (4) | 3778 | | | | | | | | | |
| alias (10) | a | b | i | r | a | m | n | \0 | | |

# Structs

```
struct student {
    int zid;            //Offset 0
    char first[20];     //Offset 4
    char last[20];      //Offset 44
    int program;        //Offset 48
    char alias[10];     //Offset 52
};
```

structs are really just sets of variables at known offsets

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| zID (4) | 5308310 | | | | | | | | | |
| first (20) | A | b | i | r | a | m | \0 | | | |
| last (20) | N | a | d | a | r | a | j | a | h | \0 |
| program (4) | 3778 | | | | | | | | | |
| alias (10) | a | b | i | r | a | m | n | \0 | | |

# Functions

# Here’s a function

```
int timesTwo(int x) {
    int two_x = x*2;
    return two_x;
}
```

- It takes an argument (x)
- It does some calculations
- It returns a value (two_x)

# Functions have "prototypes"

```
//timesTwo takes an int argument and returns an int result
int timesTwo(int x);
```

- Also known as "signatures"
- These define the number and types of parameters
- And define the type of the return value

When calling a function, we must supply an appropriate number of values each with the correct type

*(Some functions are special and can take "variable" numbers of arguments, e.g. printf - out of scope for COMP1521 but feel free to Google! varargs c)*

# A Typical Function Call

```
result = func(expr1, expr2, ...);
```

- Expressions are evaluated and associated with each parameter
- Control flow transfers to the body of `func`
- Local variables are created for `func`
- A return value is computed
- Control flow transfers to the caller which can make use of `result`

# Here's a very basic program with function

```
#include <stdio.h>

void f(void);

int main(void) {
    printf("calling function f\n");
    f();
    printf("back from function f\n");
    return 0;
}

void f(void) {
    printf("in function f\n");
}
```

Signature comes first

Function implementation

# Let’s write it in assembly

How?

Well, functions are a bit like the **labels** we have been “goto”-ing

Maybe we can use branch instructions “b”

# What if we want to call the function again???

```c
#include <stdio.h>

void f(void);

int main(void) {
    printf("calling function f\n");
    f();
    printf("back from function f\n");
    f();
    printf("back from function f again\n");
    return 0;
}
```

Signature comes first

Calling function again

# How do we actually call other functions?

- We use the `jal` instruction to call functions
- `jal` is a *special* version of the `j` (or pseudo-instruction `b`)
  - It also jumps to the given label
  - However, it also sets **`$ra` (return address)** to point to the next instruction before jumping
  - This gives us a mechanism to return to the caller function!
- However, this presents a problem…
  - Let's try run our program!

# Clobbering the $ra register

- We are overwriting the $ra register when we use **`jal`**
  - we can't return properly from the main function!
  - we end up in an infinite loop!
- Maybe we could temporarily save it in a register, like $t0 and then restore it when we need it again?

# Clobbering the $ra register

- We are overwriting the $ra register when we use **`jal`**
  - we can't return properly from the main function!
  - we end up in an infinite loop!
- Maybe we could temporarily save it in a register, like $t0 and then restore it when we need it again?
  - What is the worry with this?
    - Function could change $t0
    - Functions can call functions can call functions and we have recursive functions too. How many registers would we need? We have 32 registers max…

# How do we pass data to a function??

- We can use the $a registers to pass in arguments
  - We have $a0 - $a3 – four registers to pass in arguments
  - Can use the stack (more soon) if we theoretically had more than 4 arguments, or arguments that don't fit in a register.
    - However, you won't have to deal with this in COMP1521

# Implement this: Arguments

```
void f(int x);

int main(void) {
  printf("calling function f\n");
  f(22);
  printf("back from function f\n");
  return 0;
}

void f(int x) {
  printf("in function f\n");
  printf("%d", x);
  putchar('\n');
}
```

# How do functions return values?

- We can use the $v registers to retrieve a function’s result
  - Values occupying 32-bits or fewer should be returned using $v0
  - We don’t have to deal with $v1 in COMP1521

# Implement this: return value

```
int f(int x);

int main(void) {
  printf("calling function f\n");
  int result = f(22);
  printf("back from function f\n");
  printf("%d", result);
  putchar('\n');
  return 0;
}
```

```
int f(int x) {
  printf("in function f\n");
  printf("%d", x);
  putchar('\n');
  x = x + 1;
  return x;
}
```

# There must be a better way...

We have made a mess using t registers to

- save and restore the $ra
- save $a0 and $v0
- local variables and temporary values

This is with only a tiny program with 1 main and 1 other simple function!

There must be a more consistent way of doing this!

# Functions - a summary

- Functions are named pieces of code (**labels**)
  - Which you can call
  - Which you can (optionally) supply arguments
  - Perform computations using those arguments
  - And return a value to a caller

# Functions - a summary

- Functions are named pieces of code (**labels**)
  - Which you can call (**`jal`**)
  - Which you can (optionally) supply arguments
  - Perform computations using those arguments
  - And return a value to a caller

# Functions - a summary

- Functions are named pieces of code (**labels**)
  - Which you can call (**`jal`**)
  - Which you can (optionally) supply arguments (**`$a0 - $a3`**)
  - Perform computations using those arguments
  - And return a value to a caller

# Functions - a summary

- Functions are named pieces of code (**labels**)
  - Which you can call (**`jal`**)
  - Which you can (optionally) supply arguments (**`$a0 - $a3`**)
  - Perform computations using those arguments
  - And return a value (**`$v0`**) to a caller

## We’ve now laid some ground rules on communicating with functions.

But it gets better!

# The MIPS calling conventions

- lay out rules on how we should be using registers when interfacing between different functions
- forms the MIPS ABI (application binary interface), which lays out how different code should interact with each other
- We *theoretically* could break these rules
    - However, makes it hard to have code that works interoperably with code from other sources

# The MIPS calling conventions

- It is Important to follow these rules to make sure that functions work nicely with each other
- “You know the rules, and so do I” - Richard Paul Astley, never gonna give you up

# The MIPS calling conventions - $t registers

- $t registers are free real estate for a function
  - Functions can completely obliterate any existing values in a $t register
- However, this has implications for the function's caller
  - The *caller* function **must** assume that the *callee* function completely obliterated any values in $t registers

# Hey, but my function doesn't actually obliterate values in $t0 …

- Too bad - we MUST treat other functions like black boxes
  - We have to assume they will delete everything in our t registers.
- In fact, 'strict' autotesting for assignment 1 will intentionally destroy the existing values in your $t registers.
- The term for 'obliterating' an existing value inside a register without eventually restoring it is **clobbering**

# So we can’t preserve values between function calls in MIPS??

- could *theoretically* use global variables to preserve values
    - This could still get messy
    - However, what if we call a function recursively?
        - Global variables need to be pre-allocated,
        - We don’t know how many instances of a recursive function might exist at a given time
- Instead, we use **$s** registers to **save** values between function calls

# The MIPS calling convention - $s registers

- Functions **cannot** permanently change the value of a $s register
- This means that we can rely on our callee functions not clobbering any values we keep in $s registers
- Problem solved?? Store `$ra` in a $s register?

# Uh oh!

- But now our `main` function violates the “rules” by modifying $s0
  - The `main` function is not special, and must also abide by these rules
- All functions can potentially call other functions and also have this issue - they also “change” $ra!

# Solution: Save and restore on the stack

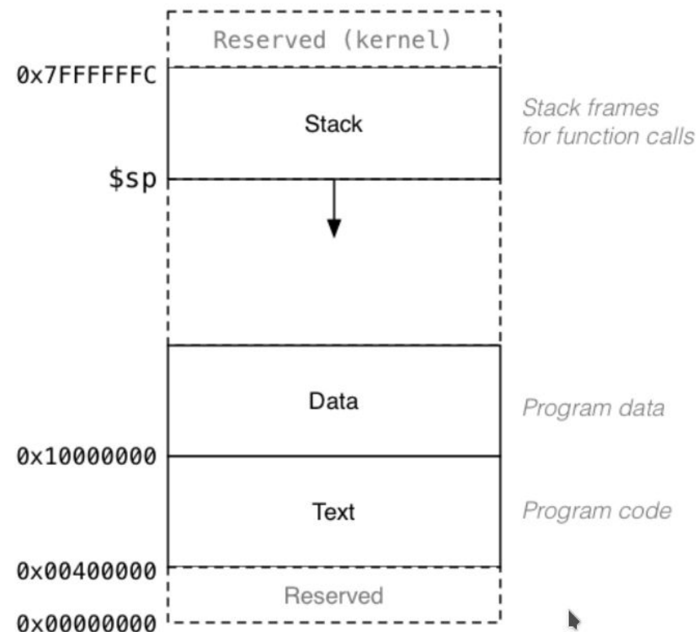
- **Solution**: functions can *temporarily* make changes to $s/$ra registers, as long as they save and restore them afterwards

- How do we do this?
    - Save the $s/$ra register's original value to RAM ( the stack) at start of the function
    - Restore the $s/$ra register's original value from RAM (the stack) once complete

# Saving to the Stack

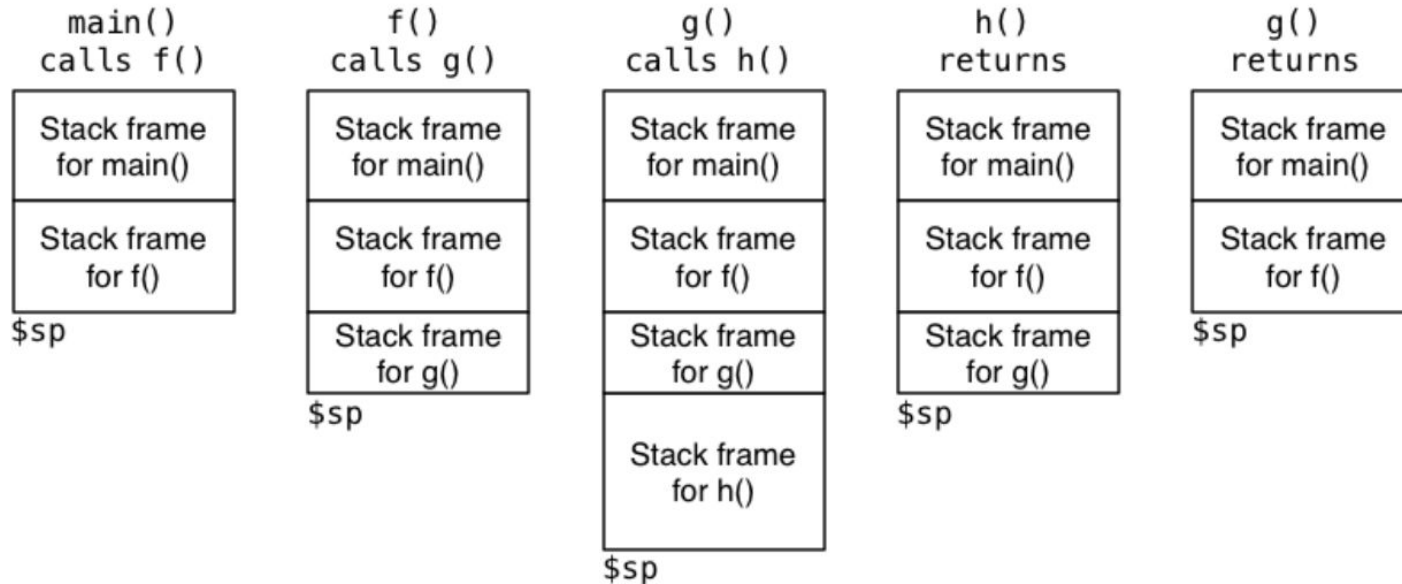
The stack

- is a region of memory which we can grow and expand
- uses the $sp (**stack pointer**) register to keep track of the top of the stack
- We can modify the stack pointer to allocate more room on the stack for us to store values



# The stack: growing and shrinking

This is how the stack changes as functions are called and return:



# The MIPS calling conventions - $sp

- Functions are free to use the stack as they need - as long as they restore $sp to its original value once done
  - That is, a function must restore the stack to its original size
- Failure to do so may lead to disastrous consequences

# Example - $sp  and the stack (the hard way)

If I subtract a total of 8 from $sp at the start of my function, and store $ra and $s0

```
addi $sp, $sp, -4
sw   $ra, ($sp)
addi $sp, $sp, -4
sw   $s0, ($sp)
```

I must reverse this by adding a total of 8 from $sp and restore $s0 and $ra at the end of the function

```
lw   $s0, ($sp)
addi $sp, $sp, 4
lw   $ra, ($sp)
addi $sp, $sp, 4
```

# Example - $sp and the stack (the easy way)

- For convenience, we provide you with two pseudo-instructions to interact with the stack: **push** and **pop**
- **push $R_t$**
  - 'allocates' 4 bytes on the stack ($sp = $sp - 4)
  - stores the value of $R_t$ to the stack
- **pop $R_t$**
  - restores the value on the top of the stack into $R_t$
  - 'deallocates' 4 bytes on the stack ($sp = $sp + 4)

# Example - $sp  and the stack (the easy way)

- These are **pseudo-instructions** provided by mipsy
  - They won't work on other MIPS emulators
- This means that you can get through this course without ever directly interacting with $sp

# Prologues and Epilogues

Prologues: the start of a function's story

- We use the **begin** instruction (more on this soon)
- We need to **push** $ra onto the stack
- We **push** the values of any $s registers we want to use

Epilogues: the end of a function's story

- We restore (**pop**) any $s registers we saved to the stack, in reverse order
- We **pop** $ra
- We use the **end** instruction (more on this soon)
- We then return to the caller with `jr $ra`

# Leaf Functions

- Are functions that don’t call any other functions
- Leaf functions don’t need to preserve $ra
  - They don’t use jal, so they never actually modify $ra
- Leaf functions *shouldn’t* need to even use $s registers
  - We only use $s registers when we want to preserve a value across a function call
  - But leaf functions don’t have any function calls within them (by definition), so they can use $t registers
- So they do not *need* a prologue and epilogue

# Out of scope for COMP1521

- Floating point registers exist to pass/return floats/doubles
    - These have similar conventions
- Stack used to pass more than 4 arguments
- Stack used to pass/return values too large for registers
    - eg. we can pass structs to functions in C, but structs can be much larger than 4 bytes

# The Frame Pointer

- $fp is another register that points to the stack
  - It points to the bottom of a given function's stack frame
  - In other words, it points to the same value as **`$sp`** before a function does any pushes/pops
- Used by debuggers to analyse the stack
  - The frame pointer, combined with saving older values of **`$fp`** to the stack essentially forms a linked list of stack frames

# The Frame Pointer

- Using a frame pointer is optional (both in COMP1521 and generally)
    - Compilers omit the use of a frame pointer when fast execution/smaller code is a priority
- Since the frame pointer tracks the original value of the stack pointer (at the start of the function), it gives us a mechanism to prevent chaos if a function pushes/pops too much
- We don’t expect you to fully understand the frame pointer in COMP1521
- Instead, we provide you with two pseudo-instructions in mipsy
    - **begin** and **end**

# The Frame Pointer: Easy Mode

- `begin`
  - saves the old $fp to the stack (keep track of the previous stack frame)
  - sets $fp to the current $sp
  - should be the first thing in the prologue
- `end`
  - restore $sp to point to the top of the previous stack frame
  - restore the $fp to point to the previous value of $fp (bottom of the previous stack frame)
  - should be right before `jr $ra`
- Not *necessary* but makes debugging in situations where you push and pop much much easier - **strongly advised**

# Function Skeleton

```
func:
        # [header comment]
func__prologue:
        begin
        push    $ra
        push    $s0
        push    $s1

func__body:
        # do stuff

        li      $a0, 42
        jal     foo         # foo(42)

        # foo return val in $v0

# at the end of the function
func__epilogue:
        pop     $s1
        pop     $s0
        pop     $ra
        end

        jr      $ra
```

# MIPS function rules: Summary

- **`$t`** registers are free real estate
  - So we must assume that other functions destroy them
- A function must restore the original values of **`$sp`**, **`$fp`**, **`$s0`**..**`$s7`**
  - So we can assume that any function we call leaves these registers unchanged
- Functions need to preserve **`$ra`** if they overwrite it
  - Otherwise, our function will lose track of where to return to
- **`$a0`**..**`$a3`** contain arguments -
  - these are also not preserved by callees (like $t)
- **`$v0`** contains the return value

# What did we learn today?

- MIPS
  - recap arrays, structs
  - Functions in MIPS
- Next lecture:
  - More examples of functions in MIPS
  - A MIPS application, putting everything together

# Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/44yfBBqDXD

# Reach Out

Content Related Questions:
Forum

Admin related Questions email:
cs1521@cse.unsw.edu.au

# Student Support | I Need Help With...

| I need help with... | | Service | Contact |
|---|---|---|---|
| **My Feelings and Mental Health**<br>Managing Low Mood, Unusual Feelings & Depression | | **Mental Health Connect** | student.unsw.edu.au/**counselling**<br>Telehealth |
| | | **In Australia Call Afterhours UNSW Mental Health Support Line** | 1300 787 026<br>5pm-9am |
| | | **Mind HUB** | student.unsw.edu.au/**mind-hub**<br>Online Self-Help Resources |
| | | **Outside Australia Afterhours 24-hour Medibank Hotline** | +61 (2) 8905 0307 |
| **Uni and Life Pressures**<br>Stress, Financial, Visas, Accommodation & More | | **Student Support Indigenous Student Support** | — student.unsw.edu.au/**advisors** |
| **Reporting Sexual Assault/Harassment** | | **Equity Diversity and Inclusion (EDI)** | — edi.unsw.edu.au/**sexual-misconduct** |
| **Educational Adjustments**<br>To Manage my Studies and Disability / Health Condition | | **Equitable Learning Service (ELS)** | — student.unsw.edu.au/**els** |
| **Academic and Study Skills** | | **Academic Language Skills** | — student.unsw.edu.au/**skills** |
| **Special Consideration**<br>Because Life Impacts our Studies and Exams | | Special Consideration | — student.unsw.edu.au/**special-consideration** |