COMP1521 25T2

Week 3 Lecture 1

MIPS FUNctions

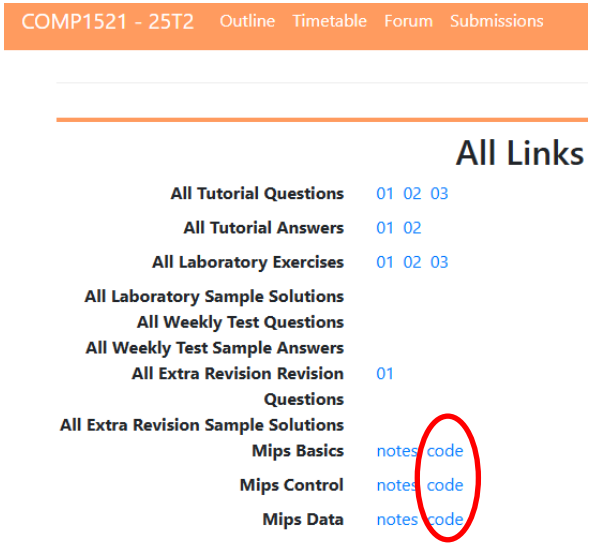
Adapted from Angela Finlayson, Abiram Nadarajah, Hammond Pearce, Andrew Taylor and John Shepherd's slides

Help Sessions and other admin

- Help Session schedule is out
  - COMP1521 25T2 — COMP1521 Help Sessions
  - Sessions started Today!
  - BYOD as they are not in labs

- Extra practice code for each topic

- Assignment 1 out later this week



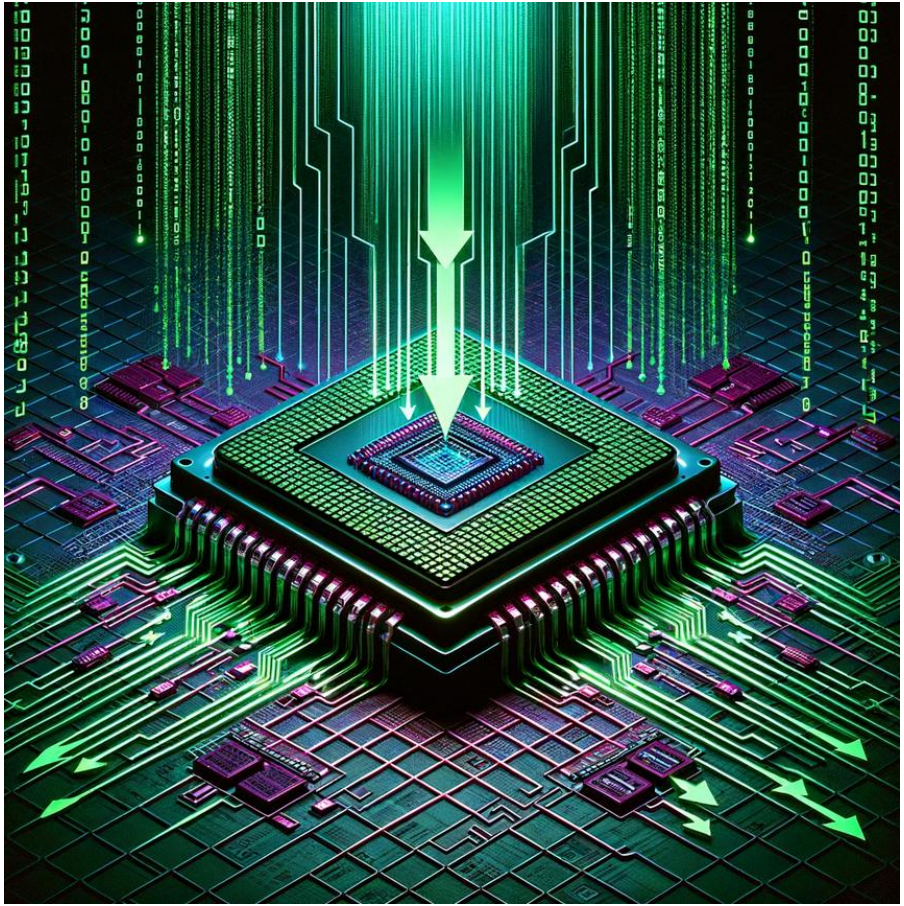
Weekly Tests start this week

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.
Self-enforced exam conditions!
You can use mips documentation

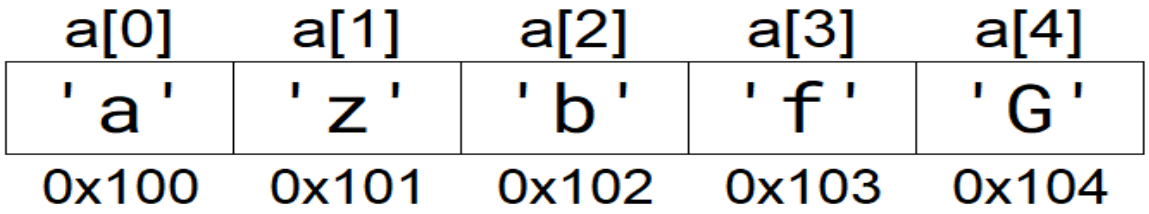
Today's Lecture

- Recap last lecture
  - Arrays
  - structs
- Functions



Recap: Arrays of 1 byte elements

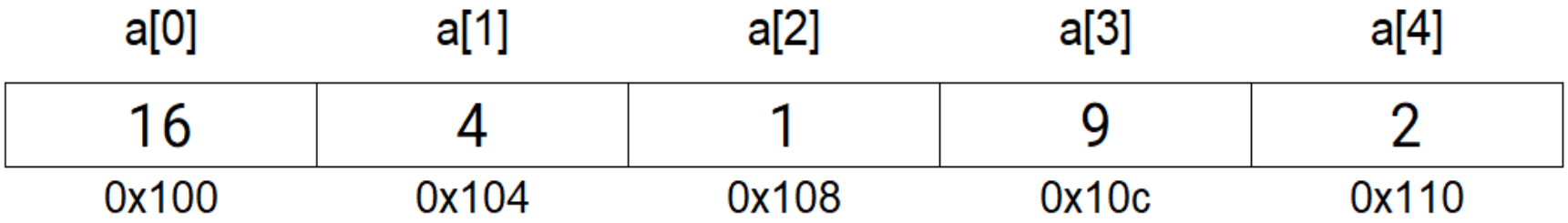
char a[5] = {'a', 'z', 'b', 'f', 'G'};



- 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]?

Recap: Arrays of 4 byte elements

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



- 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]?

Recap: 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)

Recap: 2D Arrays in MIPS

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

^ row

< col

Offset of start of relevant row:

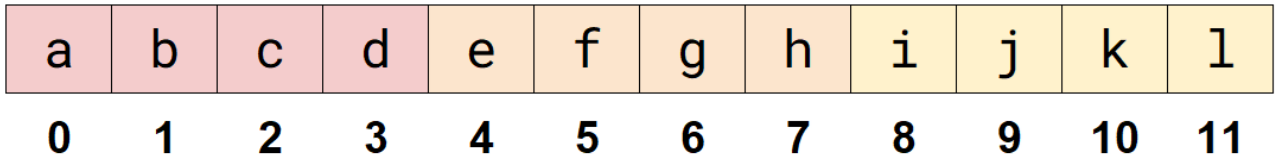
(row * N_COLS) * sizeof(element)

Offset within row:

col * sizeof(element)

Total offset:

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



Recap: MIPS array coding examples

pointer.c
print5.c
flag.c

Many more examples at:
cgi.cse.unsw.edu.au/~cs1521/25T2/topic/mips_data/code/

Try these for an exercise:
print2d.c
pointer5.c

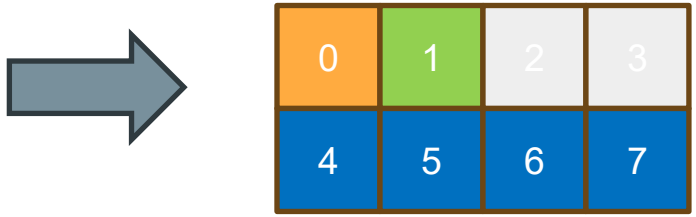
Recap Structs:

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

What size will this struct be?
What offsets will each field have?

Recap Structs:

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



What size will this struct be?

What offsets will each field have?

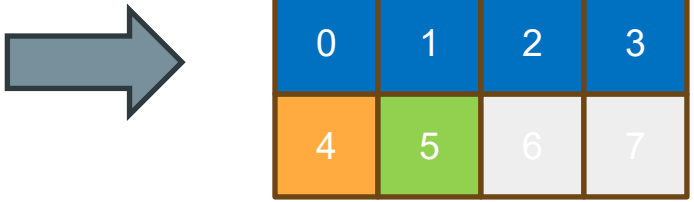
Recap Structs:

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

What if we do it like this?

Recap Structs:

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



What if we do it like this?

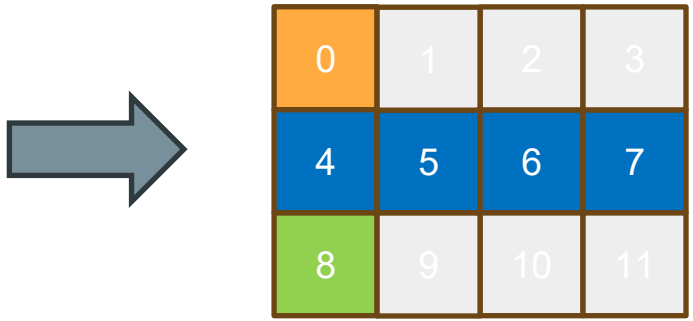
Recap Structs:

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

What if we do it like this?

Recap Structs:

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

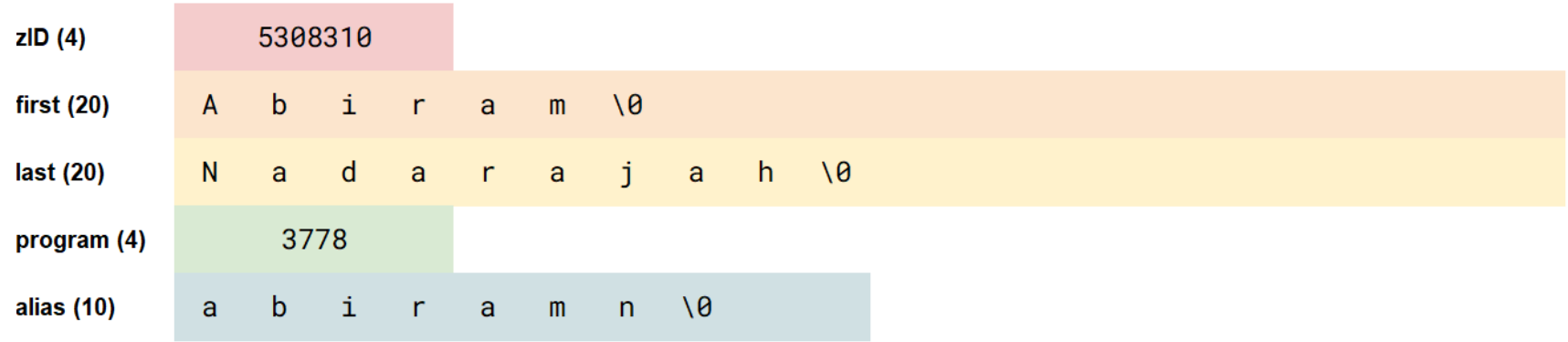


What if we do it like this?

Recap: Structs

```
struct student {
    int zid;          //Offset 0
    char first[20];   //Offset 4
    char last[20];    //Offset 24
    int program;      //Offset 44
    char alias[10];   //Offset 48
};
```

structs are really just sets of variables at known offsets



Recap: Structs

C code:
cgi.cse.unsw.edu.au/~cs1521/25T2/topic/mips_data/code/struct.c

We will jump straight to ASM today:
cgi.cse.unsw.edu.au/~cs1521/25T2/topic/mips_data/code/struct.s

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”

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

- 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 a function

```c
#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");
}
```

Declaration comes first

Definition comes later

Let’s write it in assembly

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;
}
```

Declaration 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?
  - Yes.... But...
  - Function could change $t0
    - Functions can call functions that can call functions.
    - We have recursive functions too.
    - How many registers would we need? We have 32 registers max...

Solution: Save and restore on the stack

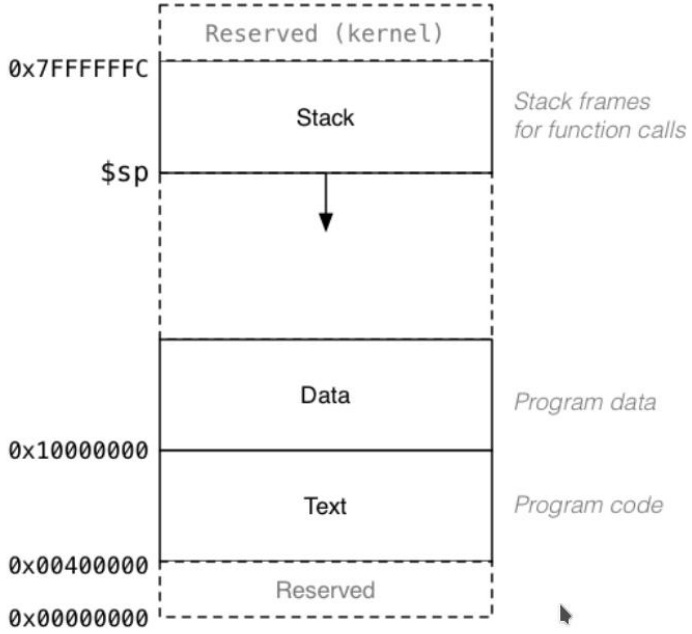
- **Solution**: functions can *temporarily* make changes to registers, as long as they save them first and restore them afterwards.

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

Saving to the Stack

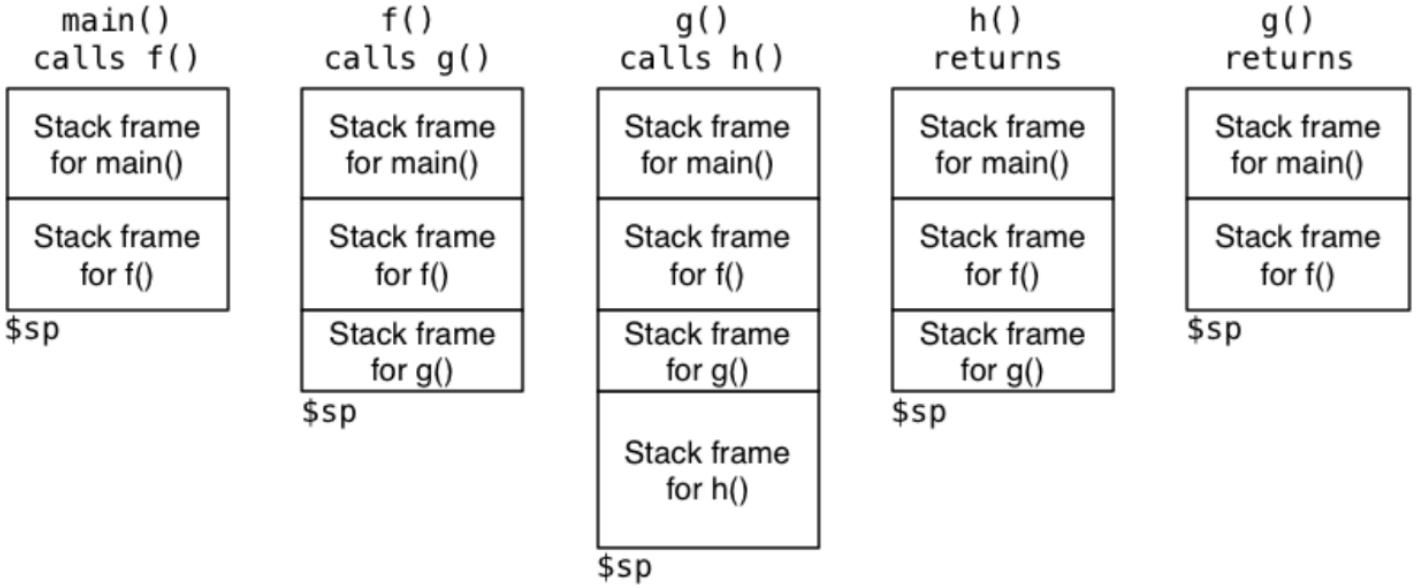
The stack

- 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, -8
sw   $ra, 0($sp)
sw   $s0, 4($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   $s0, 4($sp)
lw   $ra, 0($sp)
addi $sp, $sp, 8
```

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

- For convenience, mipsy provides us with two pseudo-instructions for stack interaction: **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`

Why only $s?

- This is by convention
- Burdensome for callee to save/restore all registers that it clobbers
- Convention:
  - Choose a limited number of registers and agree across all functions that the value in those registers must be preserved
  - Registers 16..23 ($s0..$s7) must be preserved by the callee
  - Caller ***must*** assume that other registers will be clobbered

Hey, but my function doesn't actually clobber 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.

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
  - Leaf functions don't have any function calls within them (by definition), so they can assume $t registers are never clobbered
- So leaf functions do not *need* a prologue and epilogue

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
```

Passing arguments and returning values

How do we pass data to/from a function??

- Registers keep their value across function calls
- We could use any registers that we like to pass values!
  - What if all functions expected arguments in different registers?
  - What if we edit a function to use different registers?
  - This could lead to confusion...
  - And fragile code...

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

The MIPS calling conventions - $t registers

- $t registers are free real estate for a function
  - Functions can clobber any existing values in a $t register
  - Callers must assume that called functions have clobbered $t

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 save values in $s registers before they are used and restore them before returning.

Passing data to a function

- We use the $a registers to pass in arguments
  - We have $a0 - $a3 – four registers to pass in arguments

Out of scope for COMP1521:

- Using the stack if we have more than 4 arguments, or arguments don't fit in a register (structs).
- Floating point registers to pass/return floats/doubles

Implement this: Arguments

```c
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 of 32-bits (or fewer) should be returned using $v0
  - Values of 64-bits should also use $v1
    (But we don't have to deal with $v1 in COMP1521)

Implement this: return value

```c
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;
}
```

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

Functions - a summary

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

MIPS ABI: 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 (e.g. using `jal`)
  - 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

Reach Out

Content Related Questions:
Forum

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

Student Support | I Need Help With…

My Feelings and Mental Health
Managing Low Mood, Unusual Feelings & Depression

- **Mental Health Connect** — student.unsw.edu.au/**counselling** Telehealth
- **Mind HUB** — student.unsw.edu.au/**mind-hub** Online Self-Help Resources

In Australia Call Afterhours UNSW Mental Health Support Line — 1300 787 026 5pm-9am

Outside Australia Afterhours 24-hour Medibank Hotline — +61 (2) 8905 0307

Uni and Life Pressures
Stress, Financial, Visas, Accommodation & More

- **Student Support** — student.unsw.edu.au/**advisors**
- **Indigenous Student Support**

Reporting Sexual Assault/Harassment

- **Equity Diversity and Inclusion (EDI)** — edi.unsw.edu.au/**sexual-misconduct**

Educational Adjustments
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
Because Life Impacts our Studies and Exams

- Special Consideration — student.unsw.edu.au/**special-consideration**