COMP1521 26T3

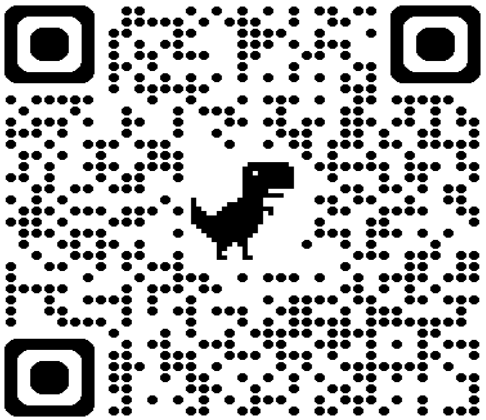
Week 3 Lecture 2

MIPS FUNctions and MIPS recap

Help Sessions and Assignment 1

- Help sessions schedule is out

Week	Mon	Tue	Wed	Thu	Fri	Sat
3		online 18:00–20:00: Rosemary Ai	f2f 14:00–16:00: H13 Lawrence East 1042 Jennifer Yu	f2f 11:00–13:00: CSE Help K17 G05 Halogen Truong	f2f 16:00–18:00: K17 SEM113 Kai Lu	online 10:00–12:00: Jayden Chen



https://cgi.cse.unsw.edu.au/~cs1521/26T3/help-sessions/

  - BYOD
  - Optional, drop in help!
  - They will get busy around assignment 1 deadline!
- Assignment 1 out this week

First Weekly Tests Out Tomorrow

Released: Thursday 3pm
Time limit: 1 hour
Due: Thursday Week 4 at 9pm. (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

Today's Lecture

- More about Functions
- MIPS Pizzeria Application
  - Putting it all together

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
  - Return a value to a caller
  - Return from the function

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
  - Return a value to a caller
  - Return from the function

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
  - Return a value to a caller
  - Return from the function

Functions - a summary

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

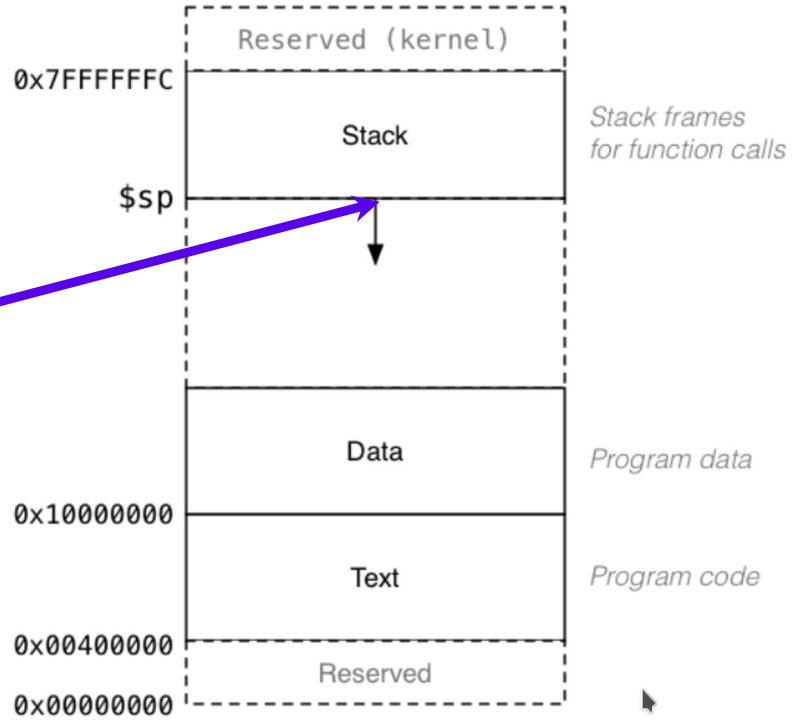
Functions - a summary

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

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

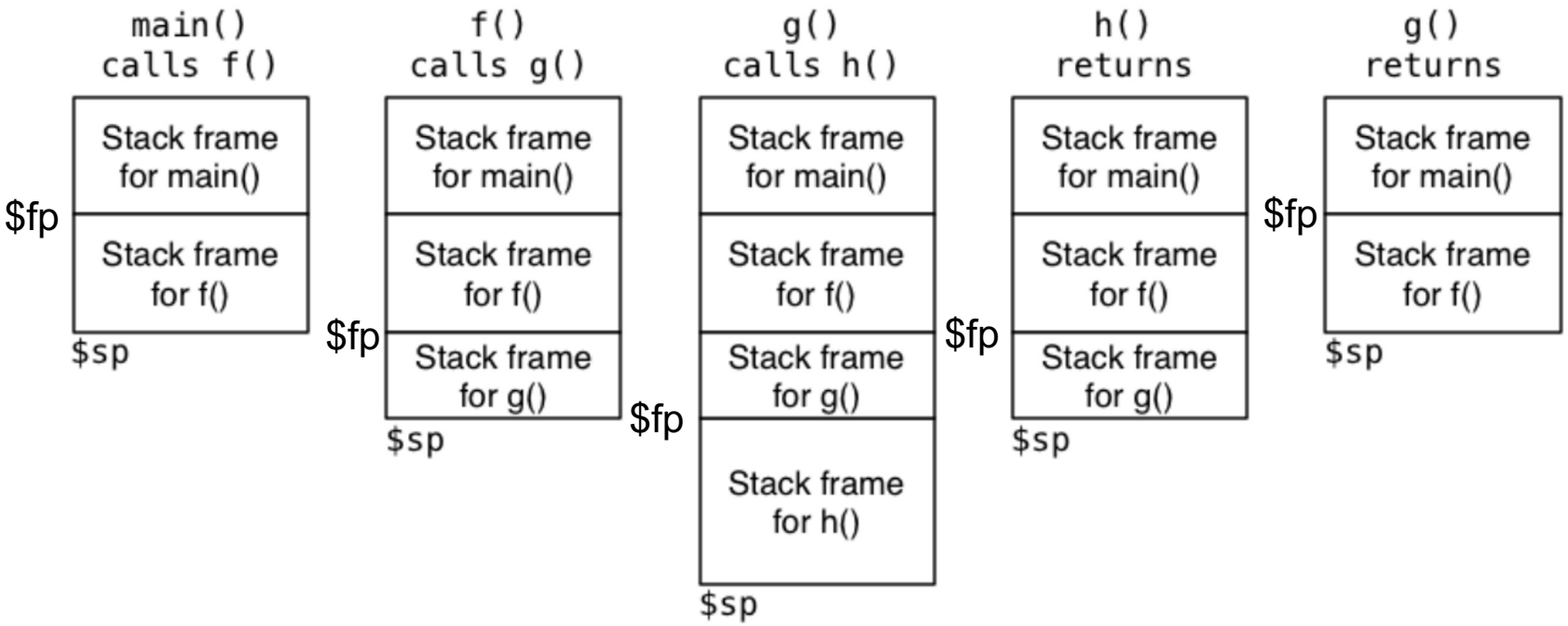


The Frame Pointer

- $fp is another register that points to the stack
  - It points to the bottom of a given function's stack frame
  - In other words, it points to the same value as **`$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 stack: growing and shrinking

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



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**

Prologues and Epilogues

Prologues: the start of a function's story

- We use the **begin** instruction
- 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
- We then return to the caller with `jr $ra`

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

Recap Exercise

sum_to.c

sum_to.s

MIPS Pizzeria Application 🍕

MIPS Pizzeria: Data Types

```
include <stdio.h>

struct pizza_t {
    char size[10];
    int price_cents;
};

struct pizza_t pizza_options[3] = {
    {"small", 300},
    {"medium", 550},
    {"large", 800}
};
```

MIPS Pizzeria: Main

```
int main(void) {
    printf("The available pizza options are:\n");
    printf("Classic:\n");
    for (int i = 0; i < 3; i++) {
        print_pizza_t(&pizza_options[i], 0);
    }
    printf("With extra toppings:\n");
    for (int i = 0; i < 3; i++) {
        print_pizza_t(&pizza_options[i], 100);
    }
    return 0;
}
```

MIPS Pizzeria: Functions

```
void print_pizza_t(struct pizza_t *pizza, int increase_cents);

void print_pizza_t(struct pizza_t *pizza, int increase_cents) {
    printf("Size: %s, ", pizza->size);
    printf("price: %d cents\n", pizza->price_cents +increase_cents);
}
```

Don’t forget before jumping into MIPS

- For each function
  - Simplify function in C
  - Compile and rerun the program to check it still works
- Don’t change everything at once without testing!

Writing Code in MIPS

- Plan register usage
- Style - consistent naming of labels
- Indentation
- Comments - equivalent line of C Code for MIPS code.

That’s all! No more MIPS!

Well ok… A little more MIPS!

Next Week

Integer Representation

Bitwise Operations

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	**In Australia Call Afterhours UNSW Mental Health Support Line** 1300 787 026 5pm-9am
		**Mind HUB**	student.unsw.edu.au/**mind-hub** Online Self-Help Resources	**Outside Australia Afterhours 24-hour Medibank Hotline** +61 (2) 8905 0307
Uni and Life Pressures 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 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**	