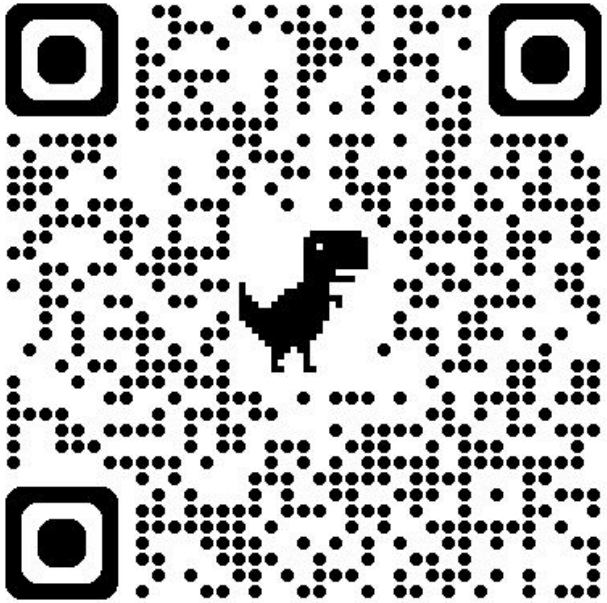
COMP1521 26T1

Week 3 Lecture 2

MIPS FUNctions and MIPS recap

Help Sessions

- Help Session Schedule is out
  - BYOD
  - Optional, drop in help!
  - They will get busy around assignment 1 deadline!
  - Get in now!



https://cgi.cse.unsw.edu.au/~cs1521/26T2/help-sessions/

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 Recap

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`**)

Stacks

LIFO(Last in first out)

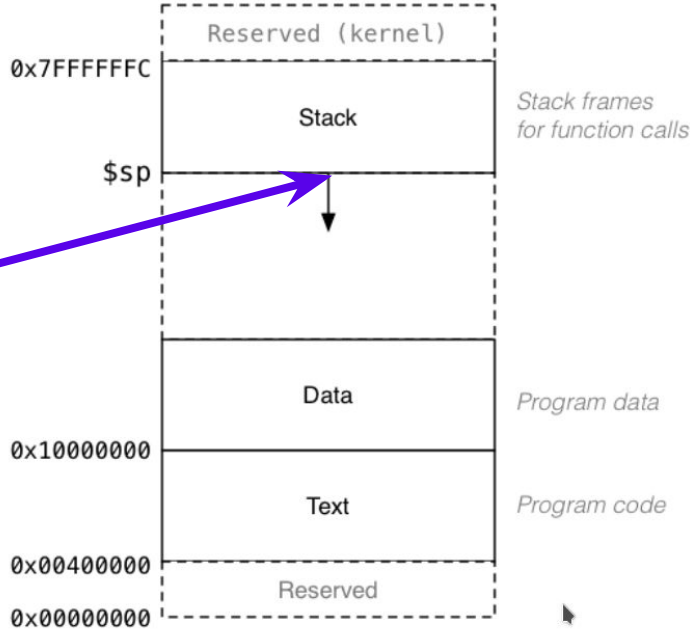
Like a stack of plates if we

- always add plates to the **top**
- always remove plates from the **top**

Saving to the Stack

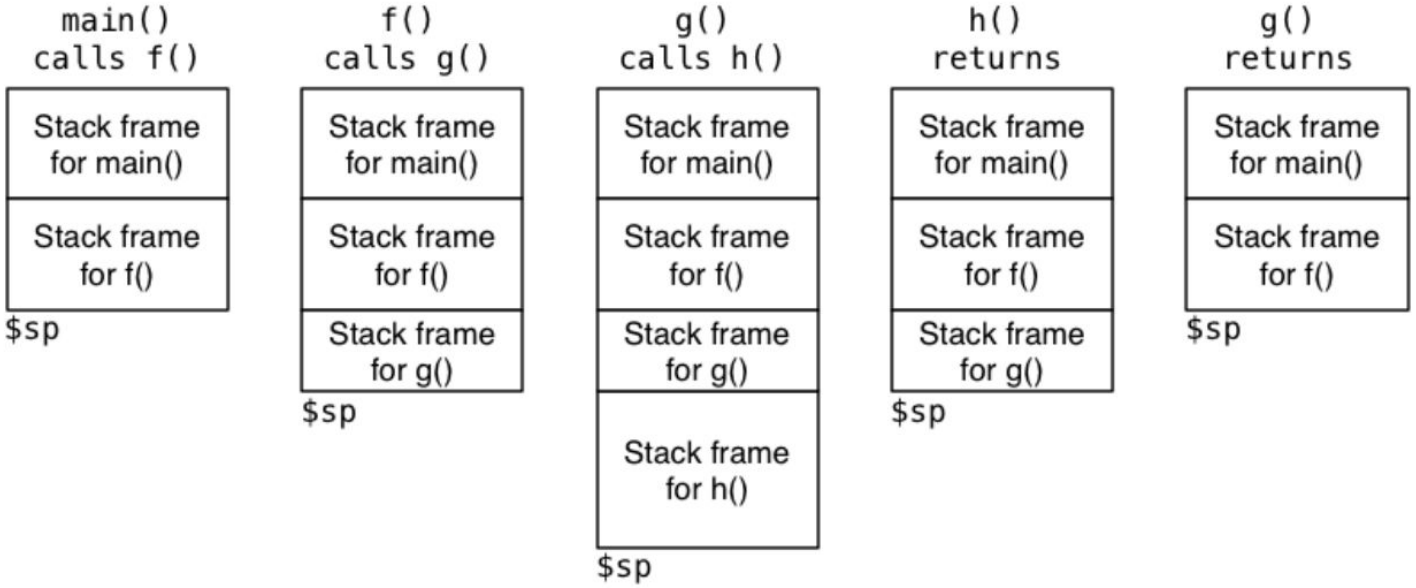
The stack

- 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:



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)

Hard way vs the easy way

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

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

```
prologue:
        push $ra

        push $s0


epilogue:
        pop $s0

        pop  $ra
```

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
- You may need it for challenge questions where you might store local variables or extra arguments on the stack.

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
- They do not usually *need* a prologue and epilogue in our course

Let’s fix our implementation of this

```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 * 2;
  return x;
}
```

Exercise

function_example_broken.s

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
```

Register Usage

Can use the other 30 registers however we want, technically, but:
There are conventions to prevent utter chaos and madness

Number	Names	Conventional Usage
0	zero	Constant 0
1	at	Reserved for assembler
2,3	v0,v1	Expression evaluation and results of a function
4..7	a0..a3	Arguments 1-4
8..16	t0..t7	Temporary (not preserved across function calls)
16..23	s0..s7	Saved temporary (preserved across function calls)
24,25	t8,t9	Temporary (not preserved across function calls)
26,27	k0,k1	Reserved for Kernel use
28	gp	Global Pointer
29	sp	Stack Pointer
30	fp	Frame Pointer
31	ra	Return Address (used by function call instructions)

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 Exercises

sum_to.c

Out of scope for COMP1521

- Floating point registers exist to pass/return floats/doubles
  - These have similar conventions
- The following may appear in challenges:
  - Stack is used to pass more than 4 arguments
  - Stack is 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
  - Stack is used to store local variables

MIPS Pizzeria Application 🍕

MIPS Pizzeria: Data Types

```c
// Written by Hammond Pearce
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

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

MIPS Pizzeria: Functions

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

void increase_price(struct pizza_t *pizza, int increase_cents) {
    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.

Revision

If time:

array_words_pointer.c

That’s all! No more MIPS!

Well ok… A little more MIPS!

Assignment 1

Out now!

Watch the walk through video.

Run the C code and play around with the game.

Try a stage 0 function tonight!!!!

Style:

- Follow the style in the code you are given
- Also look at the Style Marking Rubric
- Ask your tutors questions about style

Next Week

Integer Representation

Bitwise Operations

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/iqq1fDbMK5

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
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**