

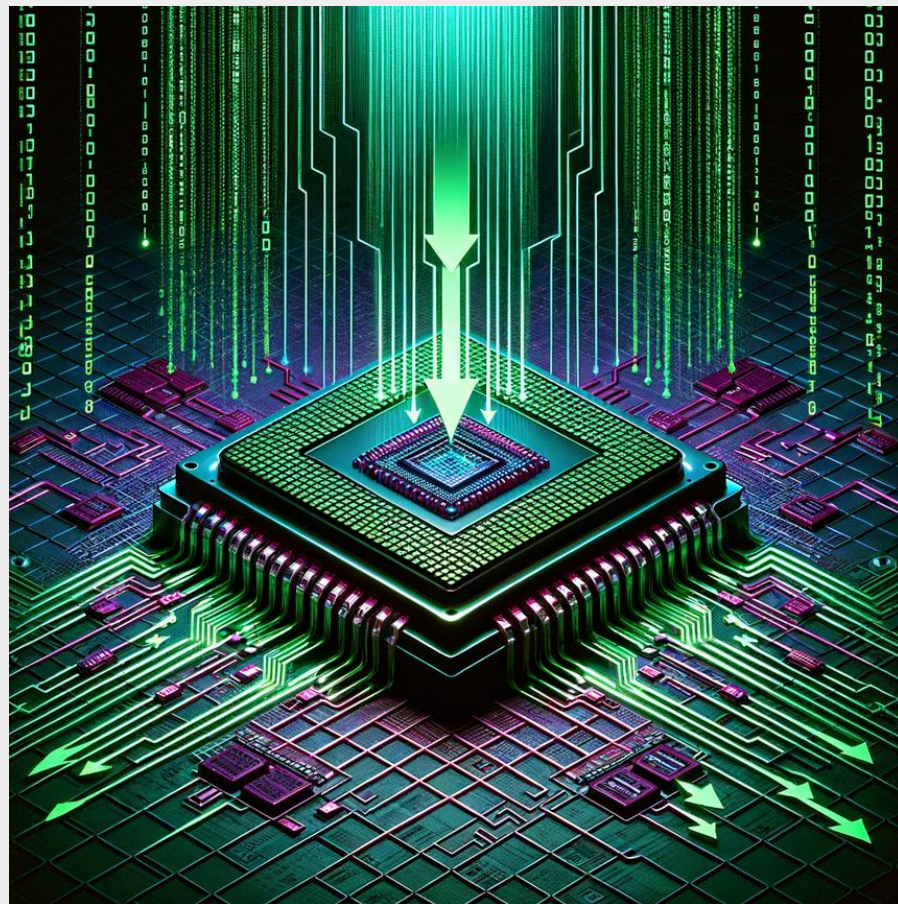


UNSW
SYDNEY

COMP1521 24T3 Lec05

MIPS: Functions

**Based on Hammond & Abiram's
Slides**



Recap challenge

What does this code do?

```
.text
main:
    la $t1, some_addr
    li $v0, 11

some_label:
    lb $a0, ($t1)
    beq $a0, $0, some_other_label
    syscall
    addi $t1, $t1, 1
    b some_label

some_other_label:
    li $v0, 0
    jr $ra
```

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`

What really is a function??

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

Here's a function

```
void timesTwo (int x) {  
    int two_x = x*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!)

Pure functions vs “impure”

- Functions take arguments and return values
- But in C we can define functions that don't take arguments or don't return values
- What would these be useful for??

Here's a very basic program with function

```
#include <stdio.h>
```


```
void f(void);
```



Signature comes first

```
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");  
}
```



Function implementation

Let's write it in assembly

How?

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

Let's start with that, using branch instructions “b”

Let's write it in assembly

How?

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

Let's start with that, using branch instructions “b”

... but what happens now if we want to call the function twice?

How do we actually call other functions?

- We use the **jal** instruction to call functions
- **jal** is a *spicy* version of the **j** (or pseudo-instruction **jb**)
 - 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!

Let's fix up the function `call_return.c`

How do we pass info 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
 - However, you won't have to deal with this in COMP1521

Implement this:

```
#include <stdio.h>

void f(int c);

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

void f(int c) {
    printf("in function f\n");
    printf("%d", c);
    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
 - Don't have to deal with \$v1 in COMP1521

Implement this:

```
#include <stdio.h>

int f(int c);

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

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


Functions - a summary

- Functions are named pieces of code
 - 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
 - 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 (**ja1**)
 - 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 (**ja1**)
 - 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.

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

The MIPS calling conventions

- *Theoretically* could break these rules
 - However, makes it hard to have code that works interoperably with code from other sources
- Important to follow these rules to make sure that functions work nicely with each other

The MIPS calling conventions

- *Theoretically* could break these rules
 - However, makes it hard to have code that works interoperably with code from other sources
- Important to follow these rules to make sure that functions work nicely with each other

“The pirates’ code of MIPS” - Zac Kologlu, former COMP1521 admin, COMP6991 lecturer

“You know the rules, and so do I” - Richard Paul Astley, won’t 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
- 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
 - 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!

- Our `main` function violates the pirates' code by modifying `$s0`
 - The `main` function is not special, and must also abide by these rules
- Recursive functions also have this issue - they “change” `$ra`!
- **Solution:** functions can *temporarily* make changes to `$s/$ra` registers, as long as they restore them afterwards

Uh oh!

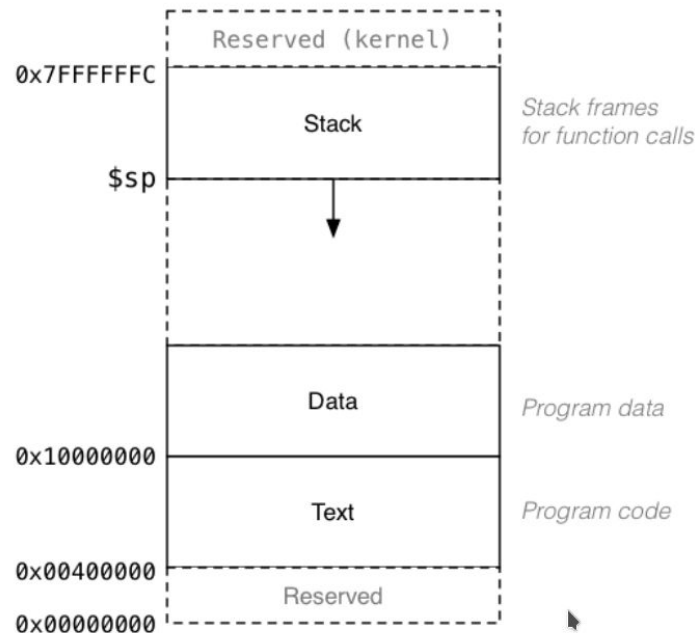
- How do we do this?
 - Save the `$s/$ra` register's original value to RAM at start of the function
 - Restore the `$s/$ra` register's original value from RAM once complete
- As far as the caller is concerned - `$s/$ra` register is still good

“Does it almost feel like nothing's changed at all?” - Dan Smith, lead vocalist, Bastille

Saving to the stack

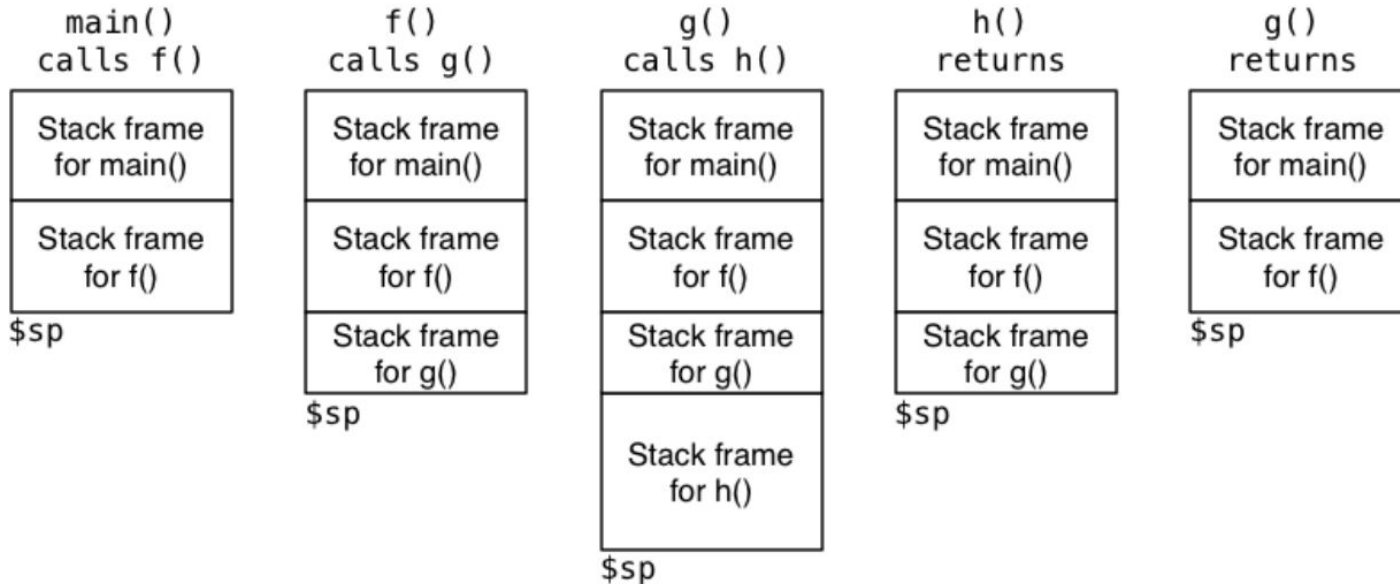
The stack

- 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

The MIPS calling conventions - \$sp

- For example,
 - If I subtract a total of 8 from \$sp at the start of my function,

```
addi $sp, $sp, -4
sw   $s0, ($sp)
addi $sp, $sp, -4
sw   $s1, ($sp)
```

- I must add 8 to \$sp before my function returns,

```
lw   $s1, ($sp)
addi $sp, $sp, 4
lw   $s0, ($sp)
addi $sp, $sp, 4
```

push and pop: the stack on easy mode

- 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 ($\$sp = \$sp + 4$)

push and pop: the stack on easy mode

- These are **pseudo-instructions** provided by mipsy - 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 are 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 are the end of a function's story
 - You may sometimes set the return value (`$v0`) here
 - We restore (**pop**) any `$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 $ra`
- You should **not** do anything else in the prologue/epilogue
- You should **not** need to push/pop outside prologue/epilogue in this course
 - Caller-preservation of `$t` registers is theoretically possible but out of scope and discouraged

Leaf functions

- Are functions that don't call any other functions (eg. every main function you saw before this lecture)
- 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
- Since there is no need to preserve values for a leaf function, they do not *need* a prologue and epilogue

The MIPS pirates' code: a summary

- **\$t** registers are free real estate
 - But they're free real estate for other functions too, so we must assume that other functions destroy them
- A function must restore the original values of **\$sp**, **\$fp**, **\$s0..\$s7**
 - But as a result, 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

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
- 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

The frame pointer - easy mode

- We don't expect you to fully understand the frame pointer in COMP1521
- Instead, we provide you with two pseudo-instructions in mipsy
 - **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**
- This makes situations where you push/pop too much easier to debug
- Not *necessary* but makes debugging 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

func__epilogue:
    pop    $s1
    pop    $s0
    pop    $ra
    end

    jr     $ra
```