Slide 1

- Functions and register conventions.
- Stacks
- Implementing Stacks on the MIPS

Slide 2

- As in high level languages , when programming in assembly language you should split up your program into smaller functions, that you can reuse.
- One of the key ideas with functions is that you can call them from any where and return back to where you called the function from.
- The MIPS processor has two instructions that enable you to call functions, `jr` and `jal`.

Slide 3

- Jump and link.

  `jal label`

  Copies the address of the next instruction into the register `$ra` (register 31) and then jumps to the address `label`.

- `jr $register` jumps to the address in `$register` most common use

  `jr $ra`

Slide 4

```
         .data
str:     .asciiz "Hello mum!.\n"
         .text
         .globl main #necessary for the assembler
main:    jal message
         jal message
         li $v0,10
         syscall     #exit the program gracefully
message: la $a0,str
         li $v0,4
         syscall     #Magic to printhings on the screen.
         jr $ra
```

Slide 5

- There are many way of passing values to functions, but there is a convention that most programs on the MIPS follow.
- `$a0-$a3` (registers 4 to 7) arguments 1-4 of a function.
- `$v0-$v1` (registers 2 and 3) results of a function.

Slide 6

```
        li $a0,10
        li $a1,21
        li $a3,31
        jal silly #Now the result of the function is is $v0.
        li $v0,10
        syscall
silly: add $t0,$a0,$a1
        sub $v0,$a3,$t0
        jr $ra
```

- On the previous slide in our function we needed an extra register to do part of a calculation.
- How do we know what registers to use?

As with function calls there is a convention.

Slide 7

- `$s0-$s7` the saved registers, these registers should be unchanged after a function call.
- `$t0-$t9` these are temporaries, are are not necessarily preserved across function calls.

So in the previous example it would of been a bad thing to use `$s0` in the function `silly`.

- What happens if we run out of registers? What happens if we have to use `$s0`?
- We would have to save it.
- But where?

Slide 8

Soon we will find a good place to store things.

Slide 9

```
       jal silly
       .
       .
       .
silly: jal silly2
       .
       .
       .
       jr $ra
```

So we have to save `$ra` as well.

Slide 10

- A stack is a data structure, at least two operations:
  - *push* put a value on the top of the stack
  - *pop* remove an item from the top of the stack.
- The important thing about a stack is that it is a LIFO (Last in First Out) data structure. This is useful for nested functions.
- You store your temporary data by pushing it onto the stack and restore things by popping things from it.

Slide 11

- The MIPS has no specialised `push` and `pop` instructions (Other processors do).
- Instead the stack is implemented using the register `$sp` (number 29), `lw` and `sw`.

Slide 12

- Unless you are writing an operating system the register `$sp` points to the top of the stack.
- On the MIPS stacks grow downwards.
- You have to manipulate the value of the register `$sp` and then use store and load.

Slide 13

To push the contents of register `$s0` onto the stack. Do the following:

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

To pop the stop of the stack into register `$s0` do the following:

```
lw $s0,0($sp)
add $sp,$sp,4
```

Basic rules:

- Every thing you push onto the stack, you must pop from the stack.
- Never touch anything on the stack that does not belong to you.

Slide 14

```
silly: addi $sp,$sp,-4
       sw $ra,0($sp)
       jal silly2
       lw $ra,0($sp)
       add $sp,$sp,4
       jr $ra
```

Slide 15

How can we make the following code more efficient?

```
silly: addi $sp,$sp,-4
       sw $s0,0($sp)
       addi $sp,$sp,-4
       sw $ra,0($sp)
       jal silly2
       lw $ra,0($sp)
       addi $sp,$sp,4
       lw $s0,0($sp)
       addi $sp,$sp,4
       jr $ra
```

We have obeyed all the rules, but we are wasting some instructions. We don't need to add or subtract four twice, we could just add or subtract 8 and then change the loads and stores.

Slide 16

```
silly: addi $sp,$sp,-8
       sw $s0,4($sp)
       sw $ra,0($sp)
       jal silly2
       lw $ra,0($sp)
       lw $s0,4($sp)
       addi $sp,$sp,8
       jr $ra
```

General rule (applies to all programs you'll every write):

- Write the inefficient version once that is correct optimise.

Slide 17

```
result: .space 4  #the place for the result

        .text
        .globl main

main:   addi $sp,$sp,-4 #save the return address.
        lw $a0, 5

        jal fact

        la $t0,result
        sw $v0, 0($t0)
        lw $ra,0($sp)
        addi $sp,$sp,4
        jr $ra
```

Slide 18

```
fact:   addi $sp,$sp,-4
        sw $ra,0($sp) #push $ra on the stack
        #fact of 0 is 1
        bne $a0,$zero,not_zero
        #Set $v0 to be 1
        addi $v0,$zero,1
        #Restore $ra from the stack
        lw $ra,0($sp) #Read $ra from the stack
        addi $sp,$sp,4 #restore the stack pointer.
        jr $ra
```

Slide 19

```
not_zero:
        addi $sp,$sp,-4
        sw $a0,0($sp) #push n on the stack ($a0=n)
        addi $a0,$a0,-1
        #So call fact with our new parameter
        jal fact # $v0=fact(n-1)
        lw $t0,0($sp) #restore n from the stack.
        addi $sp,$sp,4
        mul $v0,$v0,$t0         #$v0 = fact(n-1) ($v0) * n ($t0)
        #Restore the stack for $ra
        lw $ra,0($sp)
        addi $sp,$sp,4
        jr $ra
```