COMP1521

TUTORIAL 2

MIPS TOOLING -> MIPS BASICS -> MIPS CONTROL -> MIPS IF STATEMENTS -> MIPS LOOPS

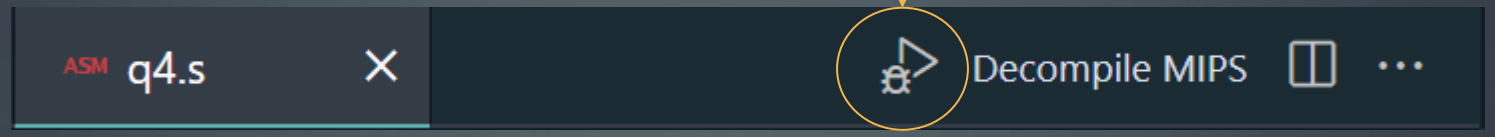
MIPS TOOLING

MIPS Tooling

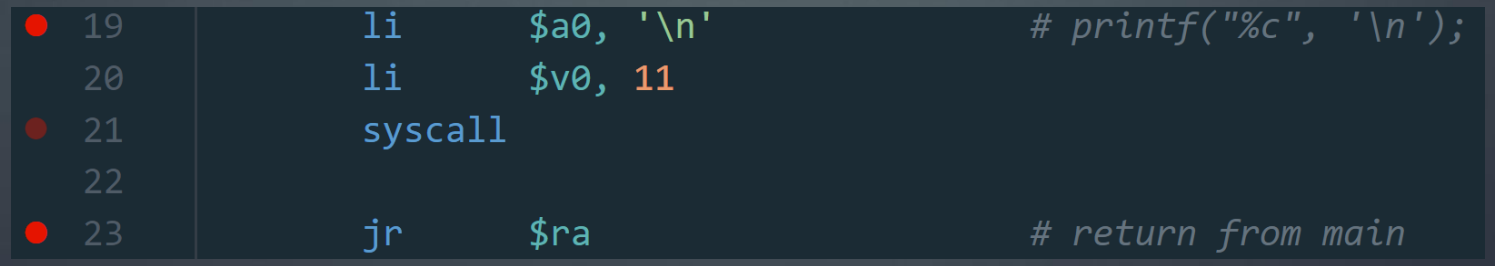
- As we learnt last week mipsy is a MIPS emulator we will use to run our assembly code
- The two main tools to write our mips code are:
  - VS Code with the Mipsy Editor Features extension
  - mipsy-web
- We use these because they provide syntax highlighting, debugging help, and ways to inspect our data segments
- To run your code on cse machines use `1521 mipsy file.s` (replacing file accordingly)

Mipsy Editor Features Toolbar

- We can open the debugger features by pressing the run and debug feature in vscode



- This will open the toolbar where we can run, step through, and inspect many aspects of our program
- Inside the debug panel, it also has information about the state of our registers and stack, and breakpoints.
- To add breakpoints, to make the program stop when we choose by clicking these red circles to the left of our code (they need to be hovered to be visible)



Mipsy Toolbar quick reference

Executes the program until next breakpoint, system call (like scanf), or program finish

Execute program until next function call, will stop at the first line of it

Undoes the previous step, moving the state back one step

Resets the program to its initial state (memory & registers)

Execute current line, if it is a function call will run the entire function in the background and stop at the next line in the current scope

Finishes executing the current function and stops the line after the function was called

Runs the program backwards until previous breakpoint or event

Terminates the program and debugging session

Opens a pane to view the contents of the programs data section

mipsy-web

https://cgi.cse.unsw.edu.au/~cs1521/mipsy/

- Very similar functionality as the Mipsy Editor Features, however it is quite a bit more intuitive to use, and a cleaner UI to work with
  - A lot of these features are a bit hidden away in the vscode extension ☹

- Some things to keep in mind using mipsy-web:
  - Must save before running the file
  - Breakpoints must be added in the decompiled section
  - Can be difficult to test assignment code as without autotests it is quite hard to black-box test your functions (not important just yet!!)

Debugging Tips

- Read error messages! It may just have the reason (not always ☹)
- Step through the code to confirm correct execution
- Find where the program is going wrong, we can work out why later!
  - Use breakpoints
  - Inspect data
  - Check register state
- Worst case – just read the code line by line to make sure you didn't mess up anywhere
  - MIPS debugging often comes down to this

MIPS BASICS

MIPS Basics

- The MIPS documentation on the course website is the best reference for MIPS, that is registers, memory, instructions, syntax, syscalls, and directives (you have this in the exam)
  - https://cgi.cse.unsw.edu.au/~cs1521/25T2/flask_tutors.cgi/resources/mips-guide.html

- What do those words mean in simple terms before we go in depth?
  - Register – storage on the cpu
  - Instructions – Similar to a function and may take in arguments, will then execute a certain functionality
    - Syntax is just the structure/format of these instructions!
  - Memory – What our MIPS programs memory looks like (very similar to process memory from last week)
    - Directives – tells the assembler how to set up the program's memory (.text, .data etc)
  - System calls (syscall) – How we interface with the operating system for input, output and other services (printing and reading to and from terminal)
  - Labels – represent the memory address of instructions, or global variables (e.g main:)

MIPS Memory/Directives

- As you may have noticed in a MIPS program a .text above our main: label, and a .data above strings used in the program
- These are directives for the assembler not the MIPS processor, it's essentially a note on how to build the programs memory, that is to identify where parts of the code will be stored in the process
- Instructions should be written below the .text directive
- Global variables and arrays should be written below a .data directive

MIPS Registers

- A CPU has many registers in our case the MIPS architecture we work with has 32! Think of registers like tiny little high-speed storage built directly into the CPU.

- We know RAM (main memory) can hold millions or billions of values so why would we use registers if it can only store 32?
  - RAM (main memory) can store all the data our program needs but is relatively far away (compared to registers) and slow for the CPU to access
  - Registers are like the CPUs personal workbench, very small, but access is nearly instant
- To conclude, registers == fast, RAM == slower

- Functionally, we can think of registers as storage our variables, except they are not initialised we just move the data to the location!

- Check the MIPS documentation for the full list of registers!

MIPS Registers (q3)

- Our MIPS CPU has 32 registers these are numbered 0-31, they can be referred to in two ways:
  - Using their number – e.g$2
  - Using their symbolic name – e.g $v0

- Overview of some important registers:
  - $0 - $zero – Read only register and always contains 0
  - $1- $at – 'assembler temporary' register, used for various purposes for the assembler (not us)
  - $2 - $v0 – This is used to hold the return value from a function, ($v1) is used if value is larger than 32 bits
  - $4 - $a0 – This is used to hold the first argument of a function, there are 3 other registers $a1-$a3, for additional arguments, otherwise the stack is used
  - $8 - $t0 – Used for holding temporary values when calculating expressions, there are 10 ($t0 -> $t9)
  - $16 - $s0 – Used for holding values that are required to persist through function calls (This is a convention, and you are responsible for restoring the values)
  - $26 - $k0 – Reserved for use by the operating system (not us)
  - $29 - $sp – 'stack pointer' register, the address of the top of the programs run-time stack. Each function call needs to reduce $sp by an amount large enough to hold non-register variables and space for saving/restoring register values
  - $31 - $ra – Holds the return address, when a linking instruction such as jal (we'll learn about this later) is used, the address of the following instruction is stored in $ra, used so function calls know where to return to
    - You'll notice all our programs have a li, $v0, 0 then a jr $ra at the bottom, this is basically return 0 we will learn more about this when we get to functions!

Moving data into registers

- When we move data into registers we typically do this in 3 ways:
  - li – (load immediate)
    - Used to store an immediate value such as a number or a character ('\n')

```
li      $v0, 4
```

  - la – (load address)
    - Use to store an address, this is referred to via a label

```
la      $a0, prompt_str         # printf("Enter a number: ");
```

  - move
    - Used to copy a value from one register to another

```
move    $t0, $v0
```

- Note: there are additional ways to move data into registers, but we will learn about them when we start working with values from the .data segment and the stack

MIPS Arithmetic Instructions

- MIPS has many instructions to perform arithmetic they generally follow this structure

  Instruction [destination register], [source 1], [source 2]

  - Instruction – the instruction you want to execute
  - [destination register] – register you want to store the value in after execution
  - [source 1] – the first value in the calculation (MUST BE A REGISTER)
  - [source 2] – can be a register or immediate value
- Some common instructions you'll use are:
  - add, sub, mul, div, rem
  - addi – is a pseudo instruction to make incrementing a value in a register simpler and more concise

```
add     $t0, $t0, $t1           # adds $t0 to $t1, store in $t0
addi    $t0, 5                  # equivalent to add $t0, $t0, 5

sub     $t0, $t0, $t1           # subtracts $t1 from $t0, store in $t0
sub     $t0, $t0, 5             # subtracts 5 from $t0, store in $t0

mul     $t0, $t0, $t1           # multiplies $t0 with $t1, store in $t0

div     $t0, $t0, $t1           # divides $t0 by $t1, store in $t0

rem     $t0, $t0, $t1           # stores remainder of $t0 divided by $t1
```

MIPS Syscalls

- We have a special instruction syscall, that allows us to request a service from the operating system. Since programs aren't allowed to access hardware directly, we must ask the OS to do it for us

- How do we setup a syscall?
  - Set $v0 to the syscall we wish to perform
  - If required provide arguments
    - e.g address of string for a print string syscall
  - execute syscall

- What does the operating system do once 'syscall' is executed?
  - Inspects $v0 to know what syscall to perform
  - Inspect other registers such as $a0 for arguments (if required)
  - Perform the requested operation
  - Return control back to our program, immediately after the 'syscall' instruction

```
la      $a0, prompt_str         # printf("Enter a number: ");
li      $v0, 4                  # print string syscall
syscall
```

MIPS Style

- Above your functions (main for now) comment registers you will be using and for what use
  - # x in $t0
- Use meaningful label names (more important once we get to functions)
- We do not indent labels even for loops
- Instructions are indented once (8 tab size)
- Beside instructions comment in-line C code (do this one) or description of assembly logic

```
# x,y in $t0, $t1
        .text
main:
        la      $a0, prompt_str         # printf("Enter a number: ");
        li      $v0, 4                  # print string syscall
        syscall
```

Example: q4.s

- Translate this C code that takes in a number from terminal and prints the squared value

```c
// Prints the square of a number

#include <stdio.h>

int main(void) {
    int x, y;

    printf(format: "Enter a number: ");
    scanf(format: "%d", &x);

    y = x * x;

    printf(format: "%d\n", y);

    return 0;
```

```
# Prints the square of a number
# x,y in $t0, $t1
        .text
main:
        la      $a0, prompt_str         # printf("Enter a number: ");
        li      $v0, 4                  # print string syscall
        syscall

        li      $v0, 5                  # scanf("%d", x);
        syscall
        move    $t0, $v0

        mul     $t1, $t0, $t0           # y = x * x

        move    $a0, $t1                # printf("%d", y);
        li      $v0, 1
        syscall

        li      $a0, '\n'               # printf("%c", '\n');
        li      $v0, 11
        syscall

        jr      $ra                     # return from main


        .data
prompt_str:
        .asciiz "Enter a number: "
```

MIPS CONTROL

Branch Instructions

- Normally, the CPU executes the instructions sequentially, one after the other, however with branch instructions we can change that flow
  - This allows us to implement control structures like loops and if/else statements!
  - They work by if the condition is met (or if unconditional), we supply a label change the programs execution to the instruction below the label supplied

- There are two fundamental types of branch instructions we use:
  - Unconditional: These change the program's flow every time they are executed
    - Often used to skip over else blocks are jump back to the start of a loop condition
    - For the scope of this course j and b are interchangeable

```
b       main_loop_cond          # goto loop_cond;
j       main_loop_cond          # goto loop_cond;
```

  - Conditional: These instructions require checking a condition and if true we change the program's flow
    - Our building blocks for conditional checks in if statements and for/while loops

```
bge     $t0, 42, main_loop_end  # if (x >= 42) goto loop_end;
```

Conditional Branch Instructions

```
main_loop_one_cond:
        # register or immediate can be used in second source
        bgt     $t0, 10, main_loop_one_end      # if (i > 10) goto loop_one_end
        bgt     $t0, $t1, main_loop_one_end     # if (i > $t1) goto loop_one_end
```

- BEQ: branch if equal
- BNE: branch if not equal
- BEQZ: branch if equal to zero (only requires one register)
- BGT: branch if first value is greater than second
- BGE: branch if first value is greater than or equal to second
- BLT: branch if first value is less than second
- BLE: branch if first value is less than or equal to second

MIPS IF STATEMENTS

MIPS If Statements

- When writing If statements we typically like the have a label structure:
  - Cond: Our condition for branching
  - Body: Code to be executed inside the if statement
  - End: The label we go to when we wouldn't enter the if statement

```
example_cond:
        beqz    $t0, example_body       # if ($t0 == 0) execute body
        b       example_end
example_body:
        li      $a0, 42                 #print 42
        li      $v0, 1
        syscall
example_end:
        li      $v0, 0                  # return 0;
        jr      $ra
```

De Morgan's Law

- Expressions such as if (A && B) can be quite tricky to translate directly into MIPS, as we can only check one condition at a time.

- The solution? Invert the condition!
  - Instead of figuring out how to branch to the next condition check and then to the body, we find the conditions that make it false and branch over the if body
- To do this we can use De Morgan's Law, it tells us how to negate logical expressions, if we have (A && B) then:
  1. !(A && B) is equivalent to (!A || !B)
  2. !(A || B) is equivalent to (!A && !B)

- In simple terms: to negate a condition, we flip each part of the condition (A -> !A) and change the (&& -> ||) or vice versa.

MIPS && Conditions

Without Inverting &&
($t0 == 0 && $t1 == 0)

1. Check cond1 if true branch to cond2, otherwise branch to end
2. Check cond2 if true branch to body, otherwise branch to end
3. Run body
4. Drop down into end normally

```
example_cond1:
        beqz    $t0, example_cond2      # if ($t0 == 0) goto cond2
        b       example_end
example_cond2:
        beqz    $t1, example_body       # if ($t1 == 0) goto body
        b       example_end
example_body:
        li      $a0, 42                 #print 42
        li      $v0, 1
        syscall
example_end:
        li      $v0, 0                  # return 0;
        jr      $ra
```

Inverting &&
($t0 != 0 || $t1 != 0)

1. If inverted cond1 is true branch to end
2. If inverted cond2 is true branch to end
3. Drop into body
4. Drop into end

```
example_cond:
        bne     $t0, 0, example_end     # if ($t0 != 0) goto end
        bne     $t1, 0, example_end     # if ($t1 != 0) goto end
example_body:
        li      $a0, 42                 #print 42
        li      $v0, 1
        syscall
example_end:
        li      $v0, 0                  # return 0;
        jr      $ra
```

I guess we accidentally also learnt how to write MIPS || conditions too!

MIPS If/Else Statements

- When continuing from our other If statements else statements can almost just be placed onto the end, we will just branch to the else body if our condition fails
- Note: We must make sure our if body branches to the end, and doesn't drop into the else body

Without Inverting

```
example_cond1:
        beqz    $t0, example_body       # if ($t0 == 0) goto body
        b       example_else            # go to else
example_body:
        li      $a0, 42                 #print 42
        li      $v0, 1
        syscall
        b       example_end             # branch to end
example_else:
        li      $a0, '\n'               # print new line
        li      $v0, 11
        syscall                         # drop down into end
example_end:
        li      $v0, 0                  # return 0;
        jr      $ra
```

With Inverting (allows one less branch instruction)

```
example_cond1:
        beqz    $t0, example_else       # if ($t0 != 0) goto else
example_body:
        li      $a0, 42                 #print 42
        li      $v0, 1
        syscall
        b       example_end             # branch to end
example_else:
        li      $a0, '\n'               # print new line
        li      $v0, 11
        syscall                         # drop down into end
example_end:
        li      $v0, 0                  # return 0;
        jr      $ra
```

Example: q6.s

Translate this C code that takes in a number from terminal and if it is less than 100, or greater than 1000 prints ‘small/big’, otherwise, prints ‘medium’

```c
#include <stdio.h>

int main(void) {
    int x;
    printf(format: "Enter a number: ");
    scanf(format: "%d", &x);

    if (x > 100 && x < 1000) {
        printf(format: "medium\n");
    } else {
        printf(format: "small/big\n");
    }
}
```

MIPS code is getting too large for slides, check the q6.s file on the github repo for the solution!

Example: SIMPLE C q6.s

Sometimes code doesn't lend itself to easy translation from C to MIPS so we can use labels and goto in C to write Simple C to make the translation easier

DON'T USE GOTO IN YOUR ACTUAL CODE ☹

Normal C

```c
#include <stdio.h>

int main(void) {
    int x;
    printf(format: "Enter a number: ");
    scanf(format: "%d", &x);

    if (x > 100 && x < 1000) {
        printf(format: "medium\n");
    } else {
        printf(format: "small/big\n");
    }
}
```

Simple C

```c
int main(void) {
    int x;
    printf(format: "Enter a number: ");
    scanf(format: "%d", &x);

    if (x <= 100) goto small_big_case;
    if (x >= 1000) goto small_big_case;

    printf(format: "medium\n");
    goto epilogue;

small_big_case:
    printf(format: "small/big\n");

epilogue:
    return 0;
}
```

MIPS code is getting too large for slides, check the q6.s file on the github repo for the solution!

MIPS LOOPS

MIPS Loops

- The simplest loop we could create in MIPS is an infinite one

```
loop_start:
        b       loop_start
```

- However, generally we want a bit more structure to our loops, the recommend structure to avoid bugs once your loops get more complicated has 5 labels:
  - Init: Contains the initialisations our variables (int i = 0)
  - Cond: Contains the exit condition of our loop (i >= 10)
  - Body: Contains the contents of our loop (code we want to run)
  - Step: Contains the increment of our loop condition and branch back to condition (i += 1) -> (b cond)
  - End: Where we branch to when our loop is finished

```
# $t0 stores i
example_init:
        li      $t0, 0                  # int i = 0
example_cond:
        bge     $t0, 10, example_end    # if (i >= 10) goto end
example_body:
        li      $a0, 42                 # print 42
        li      $v0, 1
        syscall
example_step:
        addi    $t0, 1                  # i += 1
example_end:
```

Example: q7.s

- Translate this C code into MIPS to print every third number from 24 to 42

```c
// Print every third number from 24 to 42.
#include <stdio.h>

int main(void) {
    // This 'for' loop is effectively equivalent to a while loop.
    // i.e. it is a while loop with a counter built in.
    for (int x = 24; x < 42; x += 3) {
        printf(format: "%d\n", x);
    }
}
```

MIPS code is getting too large for slides, check the q7.s file on the github repo for the solution!

Example: q8.s

- Translate this C code into MIPS to print a triangle of '*' characters

```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 10; i++) {
        for (int j = 0; j < i; j++) {
            printf(format: "*");
        }
        printf(format: "\n");
    }
    return 0;
}
```

MIPS code is getting too large for slides, check the q8.s file on the github repo for the solution!

FIN