COMP1521

Week 2 - MIPS

Overview

Admin

Tools (for running and debugging MIPS) (Q1, Q2)

Basic MIPS (registers, instructions, syscalls) (Q3, Q4)

Control MIPS (if conditions, loops) (Q6, Q7)

More tut questions:

Admin

Remember to complete the W1 challenge exercises (free marks)

W1 Lab is not due until next week monday.

W2 lab is also due next week monday.

Help sessions starting in week 3

Tools

Running/debugging MIPS: vscode

Add MIPS code to a file ending in .s

Run the code on the terminal with: “1521 mipsy program.s”

Replacing program.s with your file.

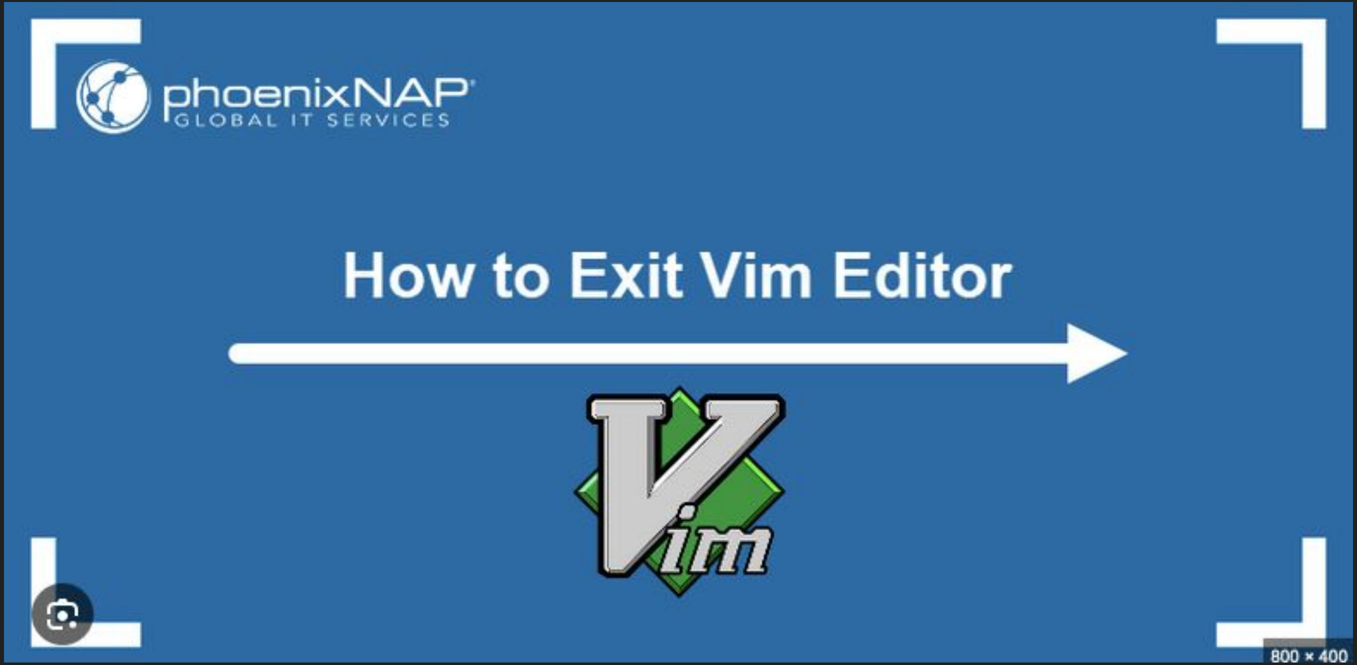
Debug (Install Xavier Cooney’s MIPSY editor features VSCODE extension!!!!!):

Add breakpoints

Step back/forth

Inspect data

Other editors



same as above but no vscode extension

Running/debugging MIPS: mipsy web

Copy paste your MIPS code into mispy web editor

Press “save”

Press “run”

Debug:

Add breakpoints

Step back/forth

Inspect data

General debugging strategies

Read the error message in full. It might just tell you exactly what the problem is.

If something is going wrong, try find exactly WHEN and WHERE it is going wrong (ie, when/where a value changes to the incorrect value), before you try and figure out WHY. Breakpoints and data inspection can help greatly with this.

Step through the code slowly and ensure it executes as you expect it to.

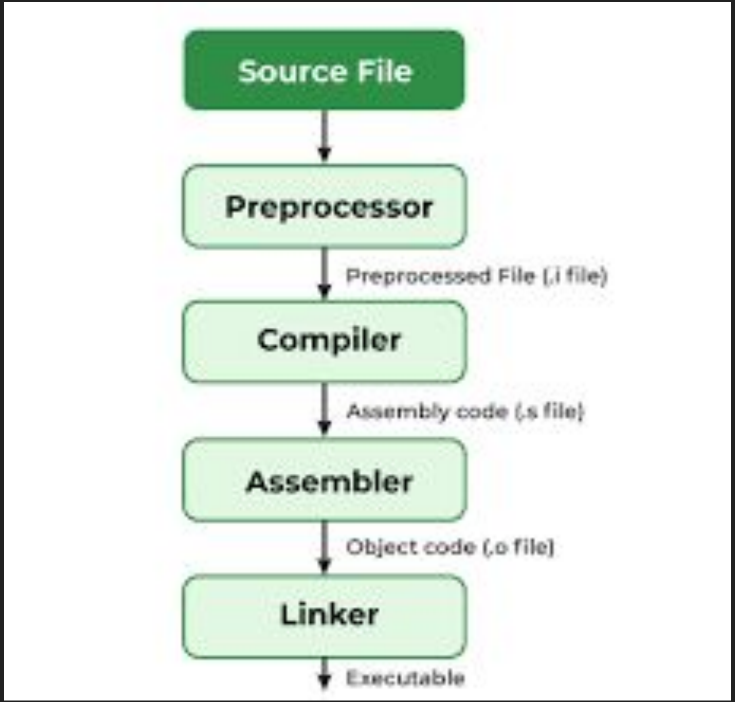
MIPS Basic

What is MIPS?

The second step of a compiler is: transform C code into assembly code for a certain processor.

MIPS is an assembly language which was used in old CPUs, and is now mostly used to teach people about assembly (and program old CPUs).

For the first half of this course we will learn to become a compiler, and convert C code into MIPS assembly code by hand.

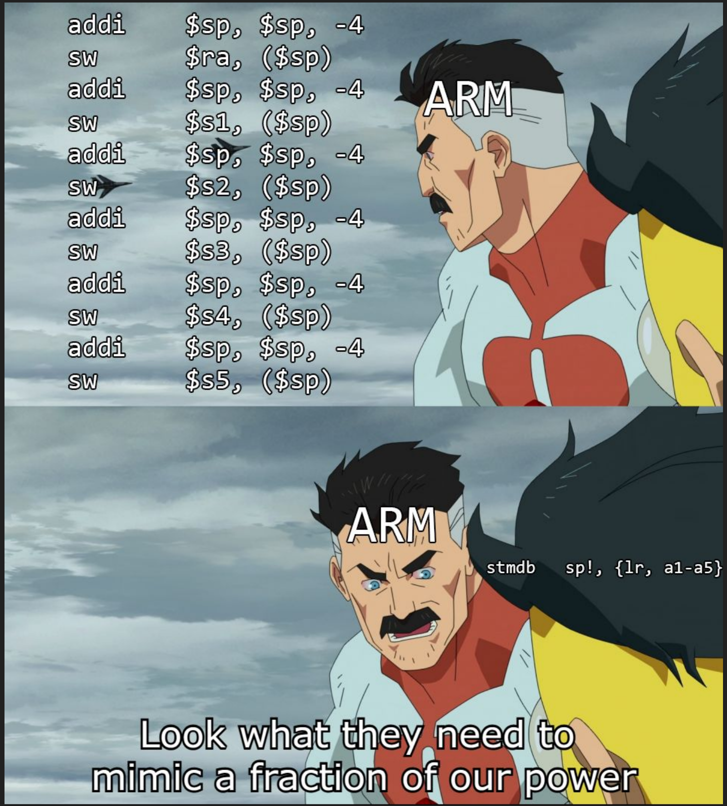


Why MIPS, why not something that is used more?

MIPS has great educational resources

Stuff like x86 and ARM can be much more complicated

By learning MIPS first you will have a much easier time learning x86 or ARM if that's something you are interested in.



```
addi    $sp, $sp, -4
sw      $ra, ($sp)
addi    $sp, $sp, -4
sw      $s1, ($sp)
addi    $sp, $sp, -4
sw      $s2, ($sp)
addi    $sp, $sp, -4
sw      $s3, ($sp)
addi    $sp, $sp, -4
sw      $s4, ($sp)
addi    $sp, $sp, -4
sw      $s5, ($sp)
```

```
stmdb   sp!, {lr, a1-a5}
```

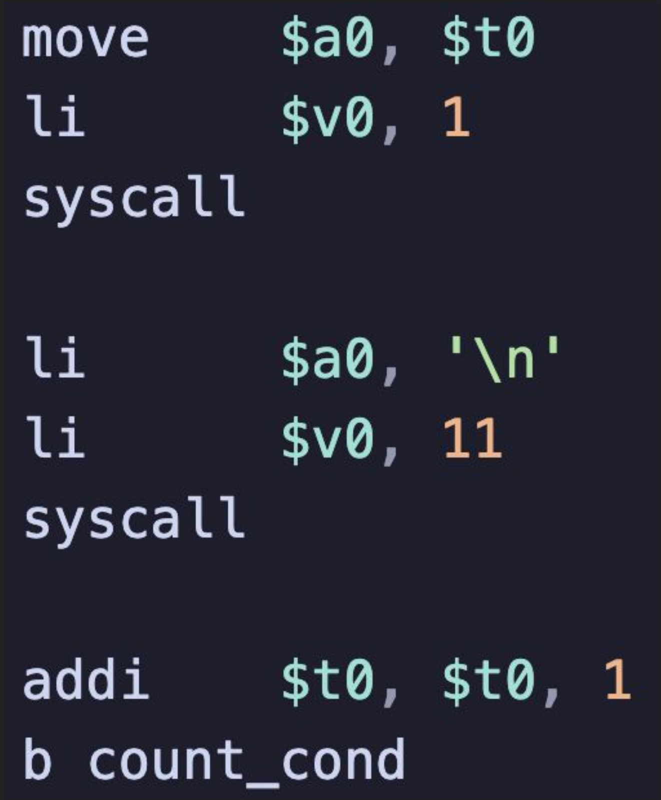
Basics of MIPS assembly

An assembly instruction looks like:

[instruction], [arg1], [arg2]

An instruction is like a function, it takes a number of arguments and does something with them

Arguments can be registers, immediate values (like 1 or ‘\n’), or they can be labels like “count_cond” (which represent memory addresses)



```
move    $a0, $t0
li      $v0, 1
syscall

li      $a0, '\n'
li      $v0, 11
syscall

addi    $t0, $t0, 1
b count_cond
```

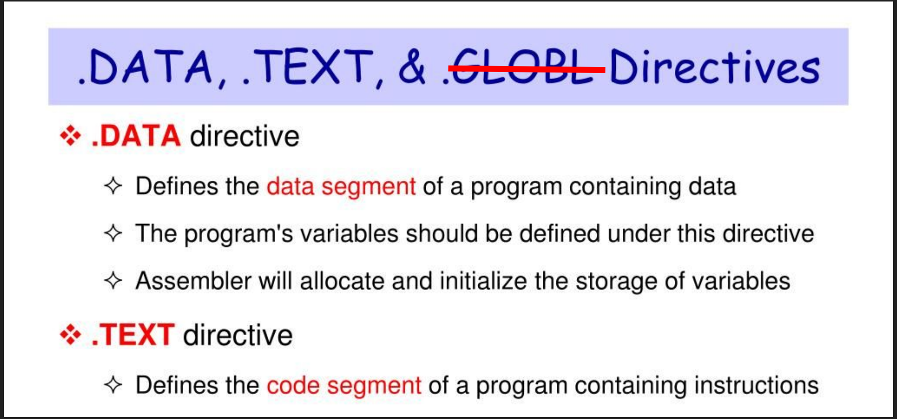
Basics of MIPS assembly

MIPS programs also contain directives like “.text” and “.data”.

These directives identify where the code below the directive should be stored in the process.

Mips instructions should be written below a “.text” directive.

Global variables/arrays should be written below a “.data” directive.



.DATA, .TEXT, & ~~.GLOBL~~ Directives

- **.DATA** directive
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- **.TEXT** directive
  - Defines the code segment of a program containing instructions

Basics of MIPS assembly

MIPS also contain labels, like “main:”.

Labels are used to represent the memory address of something, whether that is a particular instruction, or a global variable.

“main” refers to the memory address of the instruction immediately below the “main:” label.

“var1” refers to the memory address of a variable immediately below the “var1:” label in the data section.

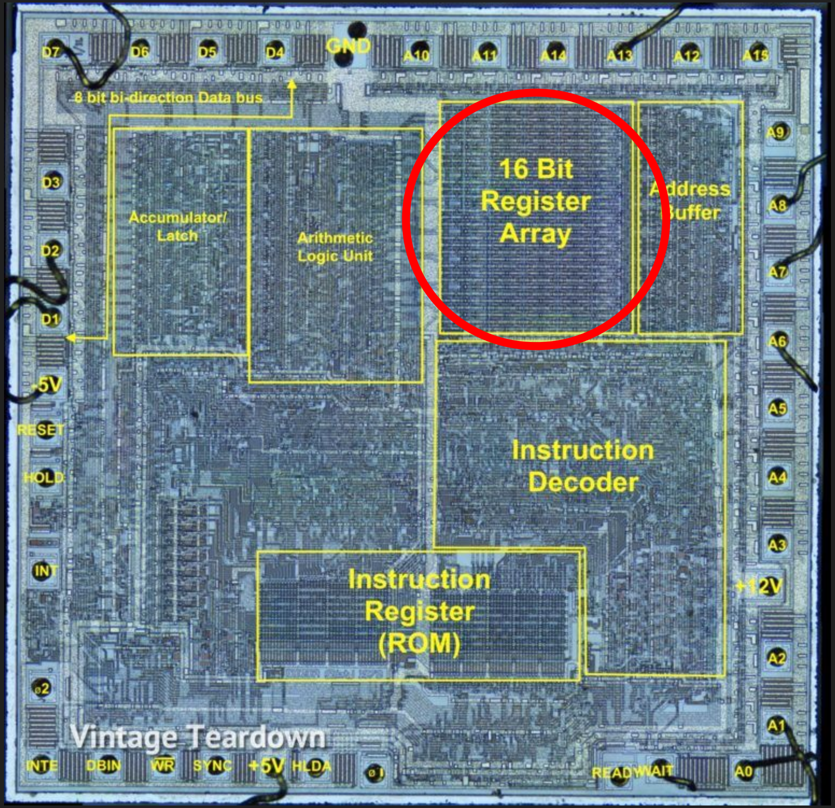
Functionally, labels are the same things as integer literals.

Registers

Inside a CPU there exist registers.

Registers store values (like variables)

Functionally, register = variable, except you don't need to initialise registers (they already exist in the CPU)

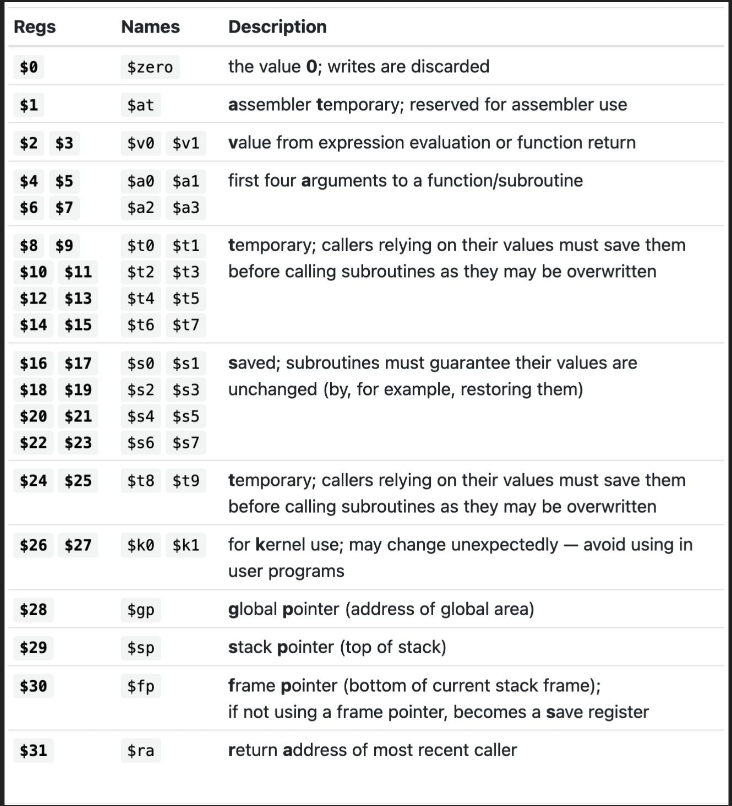


MIPS Registers

MIPS has 32 registers, numbered 0-31.

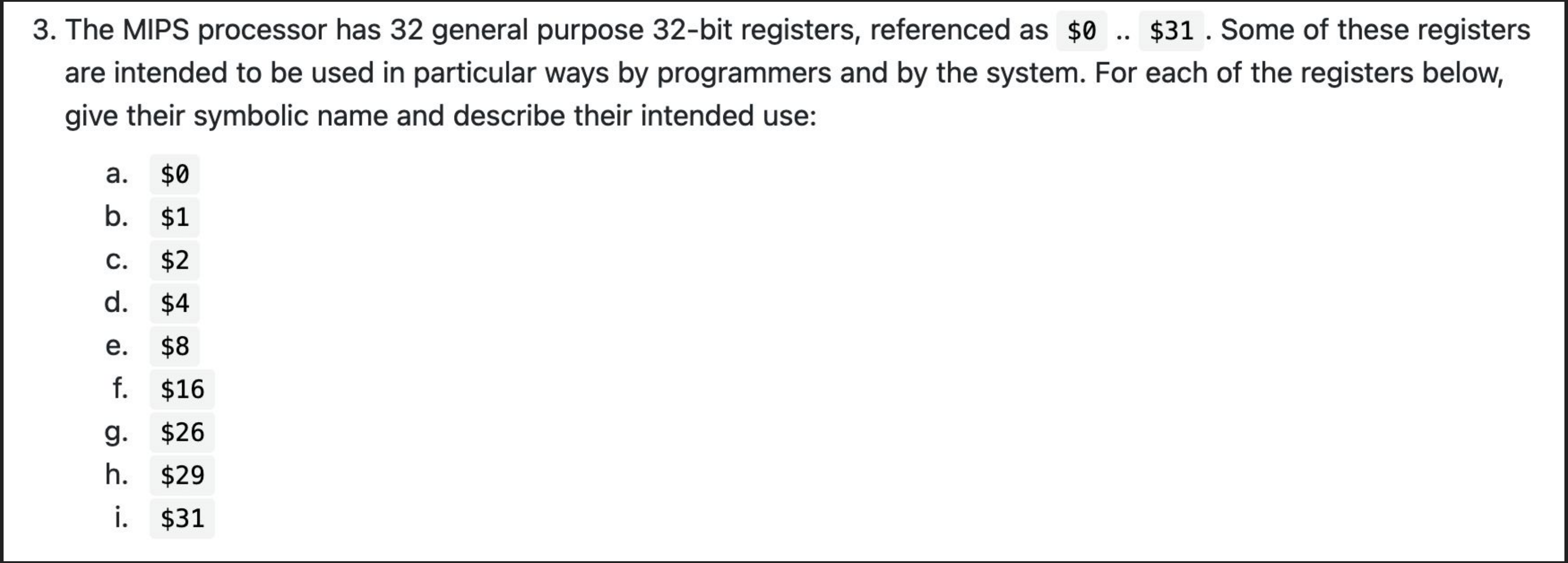
They can be referred to in a MIPS program using $[number] (for example $2).

They can also be referred to by their symbolic name (for example $v0, see second column on the right)



Regs	Names	Description
`$0`	`$zero`	the value **0**; writes are discarded
`$1`	`$at`	**a**ssembler **t**emporary; reserved for assembler use
`$2` **`$3`**	`$v0` `$v1`	**v**alue from expression evaluation or function return
`$4` **`$5`** **`$6`** **`$7`**	`$a0` `$a1` `$a2` `$a3`	first four **a**rguments to a function/subroutine
`$8` **`$9`** **`$10`** **`$11`** **`$12`** **`$13`** **`$14`** **`$15`**	`$t0` `$t1` `$t2` `$t3` `$t4` `$t5` `$t6` `$t7`	**t**emporary; callers relying on their values must save them before calling subroutines as they may be overwritten
`$16` **`$17`** **`$18`** **`$19`** **`$20`** **`$21`** **`$22`** **`$23`**	`$s0` `$s1` `$s2` `$s3` `$s4` `$s5` `$s6` `$s7`	**s**aved; subroutines must guarantee their values are unchanged (by, for example, restoring them)
`$24` **`$25`**	`$t8` `$t9`	**t**emporary; callers relying on their values must save them before calling subroutines as they may be overwritten
`$26` **`$27`**	`$k0` `$k1`	for **k**ernel use; may change unexpectedly — avoid using in user programs
`$28`	`$gp`	**g**lobal **p**ointer (address of global area)
`$29`	`$sp`	**s**tack **p**ointer (top of stack)
`$30`	`$fp`	**f**rame **p**ointer (bottom of current stack frame); if not using a frame pointer, becomes a **s**ave register
`$31`	`$ra`	**r**eturn **a**ddress of most recent caller

Tutorial Q3



3. The MIPS processor has 32 general purpose 32-bit registers, referenced as `$0` .. `$31`. Some of these registers are intended to be used in particular ways by programmers and by the system. For each of the registers below, give their symbolic name and describe their intended use:

   a. `$0`
   b. `$1`
   c. `$2`
   d. `$4`
   e. `$8`
   f. `$16`
   g. `$26`
   h. `$29`
   i. `$31`

MIPS Instructions

For moving values into and between registers, use instructions:

li (load immediate): Store an immediate value (like 1 or ‘\n’) into a register:

```
    li,      $t0,     0
```

la (load address): Store an address (referred to by a label) into a register:

```
    la,      $t0,     my_label
```

move: Copy a value from one register to another

```
    move,    $t0,     $t1         (copy $t1 value into $t0)
```

MIPS Instructions

Common arithmetic instructions:

add: add r1, r2, r3 means r1 = r2 + r3

sub: sub r1, r2, r3 means r1 = r2 - r3

mul: mul r1, r2, r3 means r1 = r2 * r3

div: div r1, r2, r3 means r1 = r2 / r3

rem: rem r1, r2, r3 means r1 = r2 % r3

MIPS Instructions

The following instructions perform the same role as “return 0” in a C main function.

We will learn how they work in a week or two. For now just include them at the end of main.

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

MIPS Syscalls

“syscall” is a special MIPS instruction.

Syscalls are used mainly to do things like input and output (ie, scanf and printf).

When we have a “syscall” instruction (just “syscall”, no arguments), the program hands control to the operating system.

First, the operating system inspects the value of register $v0, in order to decide which syscall is to be performed (for example, if $v0 contains 4, then we will perform the “print string” syscall).

Then, the operating system will perform that syscall. This may involve inspecting the values of other registers (like print string, which will read $a0 to find the address of the string it should print).

MIPS Syscalls

To use a syscall you should first set $v0 and other registers appropriately, before writing “syscall” to execute it.

Then to summarise:

“syscall” -> “read $v0” -> “read other registers” -> “perform syscall operation”

Is what the computer will do after that “syscall” instruction

MIPS Syscalls

Typical MIPS syscall code:

```
li          $v0,      4     # we are performing “print string” syscall, which is syscall 4
li          $a0,      my_string # load label/address of string into $a0 for OS to read
syscall                           # hand control to OS to perform syscall
```

The operation is not finished until after the “syscall” instruction occurs.

MIPS general style

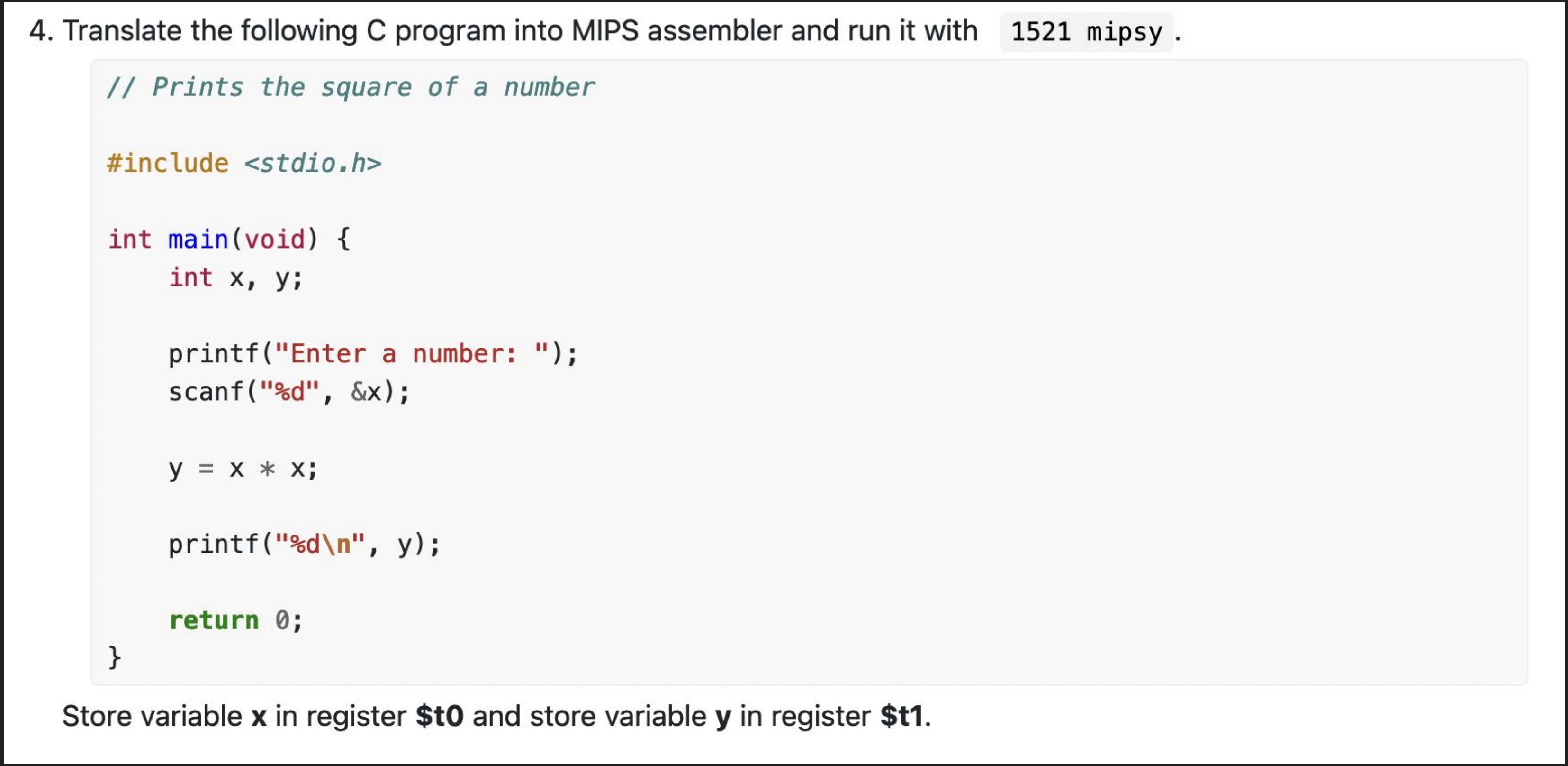
Comment registers at top of code (what are you using them for?)

Indent instructions, do not indent labels. Never indent multiple times to indicate structure (like nesting loops).

Comment in-line with equivalent C code (preferred) or description of assembly logic.

Use meaningful names for labels.

Tutorial Q4



4. Translate the following C program into MIPS assembler and run it with `1521 mipsy`.

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

#include <stdio.h>

int main(void) {
    int x, y;

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

    y = x * x;

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

    return 0;
}
```

Store variable **x** in register **$t0** and store variable **y** in register **$t1**.

MIPS Control

Branch instructions

In order to implement logic like loops and conditionals, we need to use branch instructions.

For example:

```
    b        my_label
```

Will move execution to the instruction immediately following the label “my_label”.

This can allow us to “skip” over sections of code, and to jump “backwards” to earlier lines of code.

Conditional branch instructions

There are a lot of “conditional” branch instructions.

These act the same as the unconditional branch, but they only execute the branch if the condition is true.

For example ,we can use beq (branch if equal) like so:

```
	beq	$t0,	$t1,	my_label
```

Which will jump to the instruction immediately following the label “my_label”, as long as the contents of $t0 is equal to the contents of $t1.

These instructions are crucial for implementing if/else/while/for.

Conditional branch instructions

Other common conditional branch instructions:

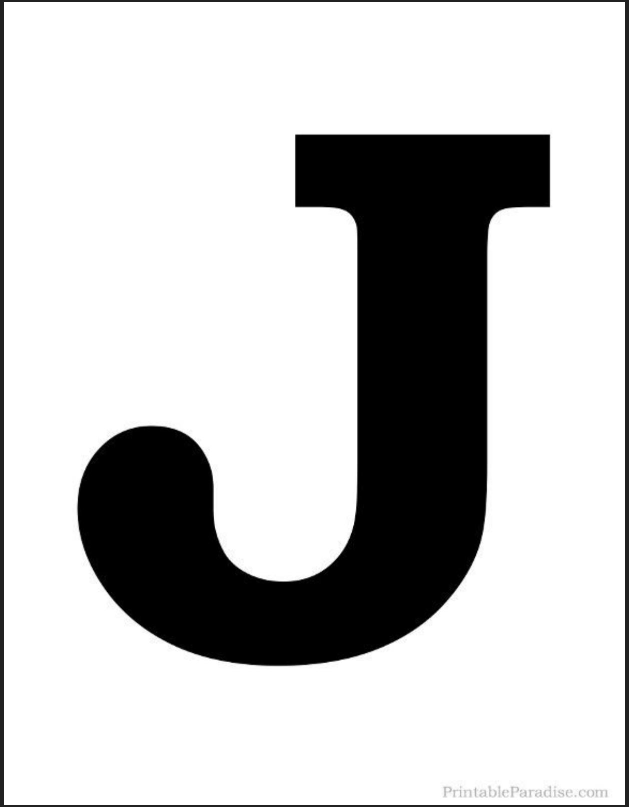
bne	:	branch if not equal
bgt	:	branch if greater than
bge	:	branch if greater than or equal to
blt	:	~~bacon lettuce tomato~~ branch if less than
ble	:	branch if less than or equal to
beqz	:	branch if equal to zero

Branch instructions

There is also the “j” instruction.

It’s equivalent to the “b” instruction. The only difference is that “j” can technically jump “further”. But in the programs we write, that difference will never matter.

Use whichever you prefer.



If statements

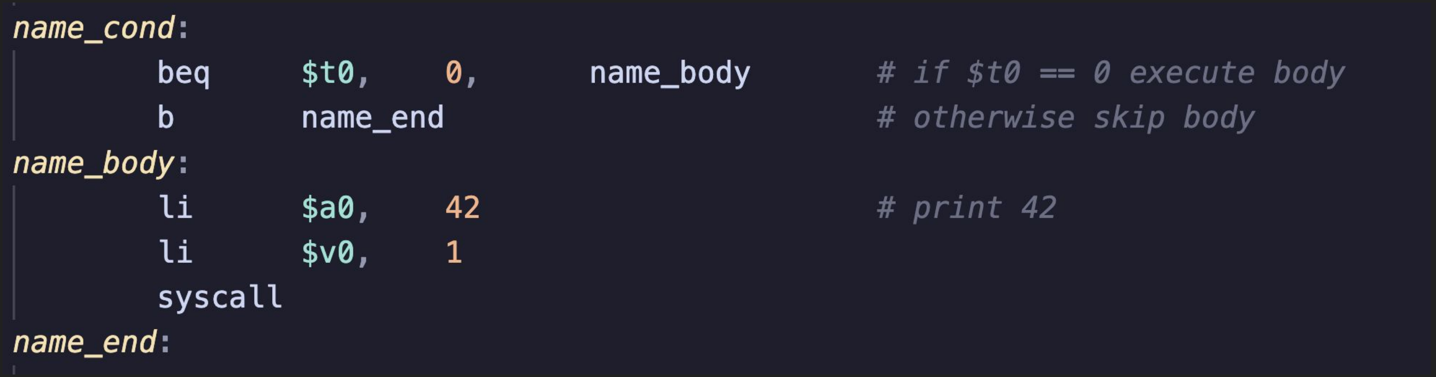
When writing structures like these, ALWAYS write the labels first!!!!!

Recommended label structure for if, where “name” is replaced with an appropriate name for the structure:

name_cond:

name_body:

name_end:

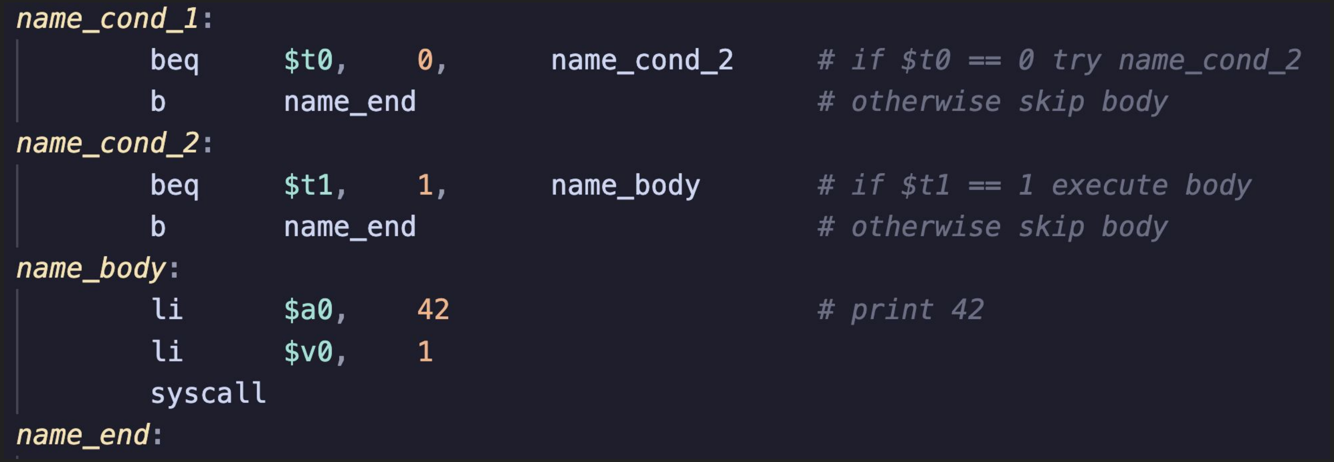


```
name_cond:
        beq     $t0,    0,      name_body       # if $t0 == 0 execute body
        b       name_end                        # otherwise skip body
name_body:
        li      $a0,    42                      # print 42
        li      $v0,    1
        syscall
name_end:
```

&& conditions

With AND conditions, we can introduce an extra label for each condition.

This way, we must pass both conditions to reach the body. Failing either test will cause us to skip.



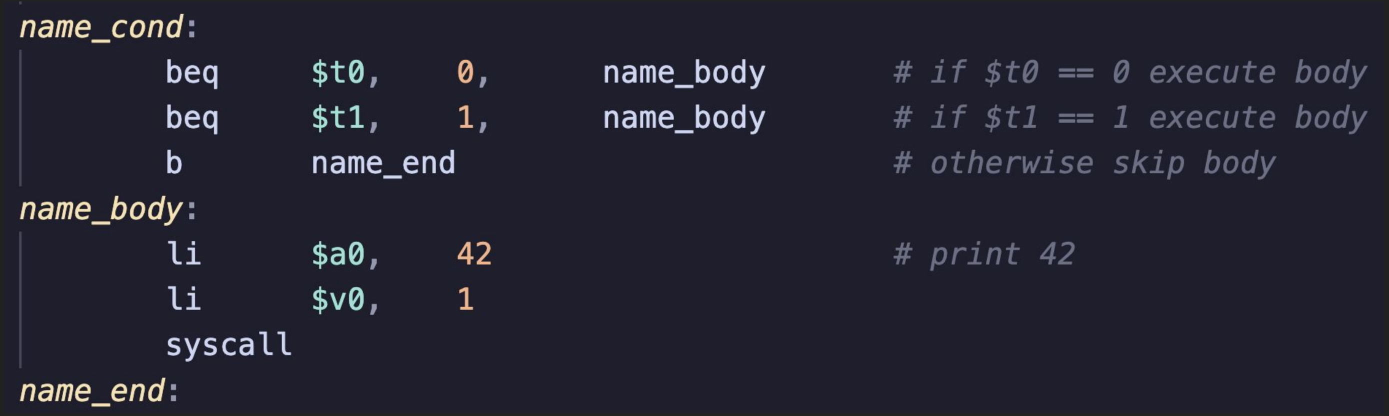
```
name_cond_1:
        beq     $t0,    0,      name_cond_2     # if $t0 == 0 try name_cond_2
        b       name_end                        # otherwise skip body
name_cond_2:
        beq     $t1,    1,      name_body       # if $t1 == 1 execute body
        b       name_end                        # otherwise skip body
name_body:
        li      $a0,    42                      # print 42
        li      $v0,    1
        syscall
name_end:
```

|| conditions

With OR conditions, we can introduce a branch statement for each condition.

If any of the conditions are met then we will branch to the body.

Only after failing all tests we will skip the body



```
name_cond:
        beq     $t0,    0,      name_body       # if $t0 == 0 execute body
        beq     $t1,    1,      name_body       # if $t1 == 1 execute body
        b       name_end                        # otherwise skip body
name_body:
        li      $a0,    42                      # print 42
        li      $v0,    1
        syscall
name_end:
```

Else statements

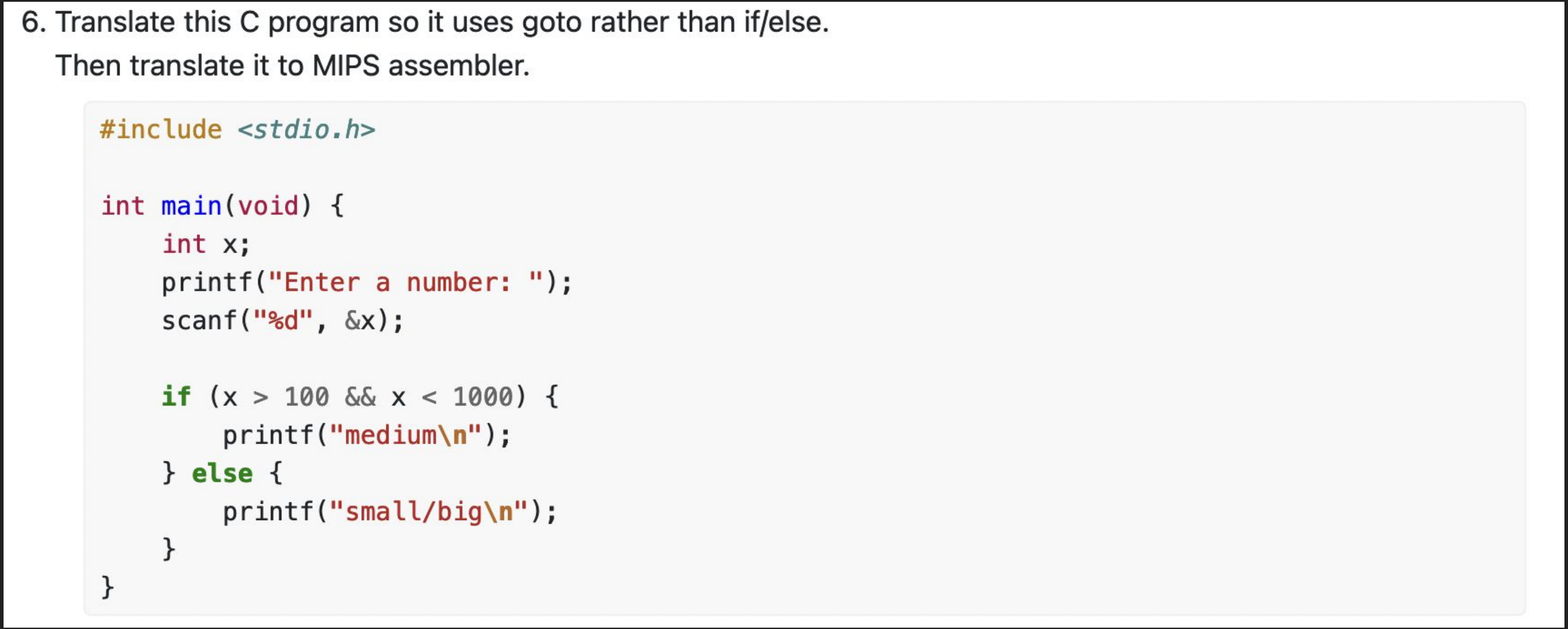
We can introduce some extra code following the body, and allow it to run only when all conditions fail.

Be careful to avoid the main body accidentally also running the else branch:

Ensure you include the branch from the body to the end of the statement.

```
name_cond:
        beq     $t0,    0,      name_body       # if $t0 == 0 execute body
        b       name_else                       # otherwise execute else branch
name_body:
        li      $a0,    42                      # print 42
        li      $v0,    1
        syscall
        b       name_end                        # don't run the else branch
name_else:
        li      $a0,    7                       # print 7
        li      $v0,    1
        syscall
name_end:
```

Tutorial Q6



6. Translate this C program so it uses goto rather than if/else.
   Then translate it to MIPS assembler.

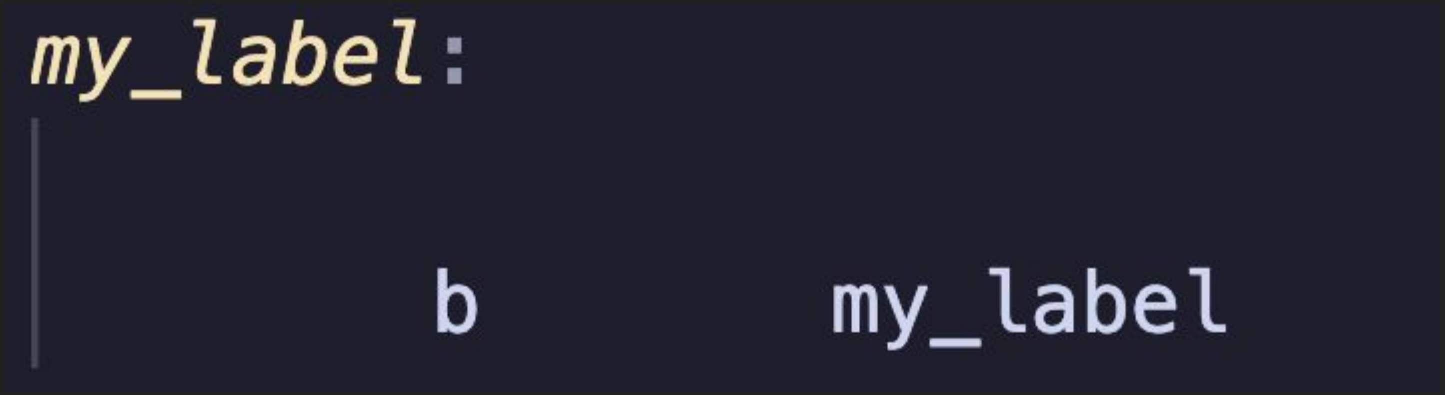
```c
#include <stdio.h>

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

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

Loops

The most basic loop we could make:

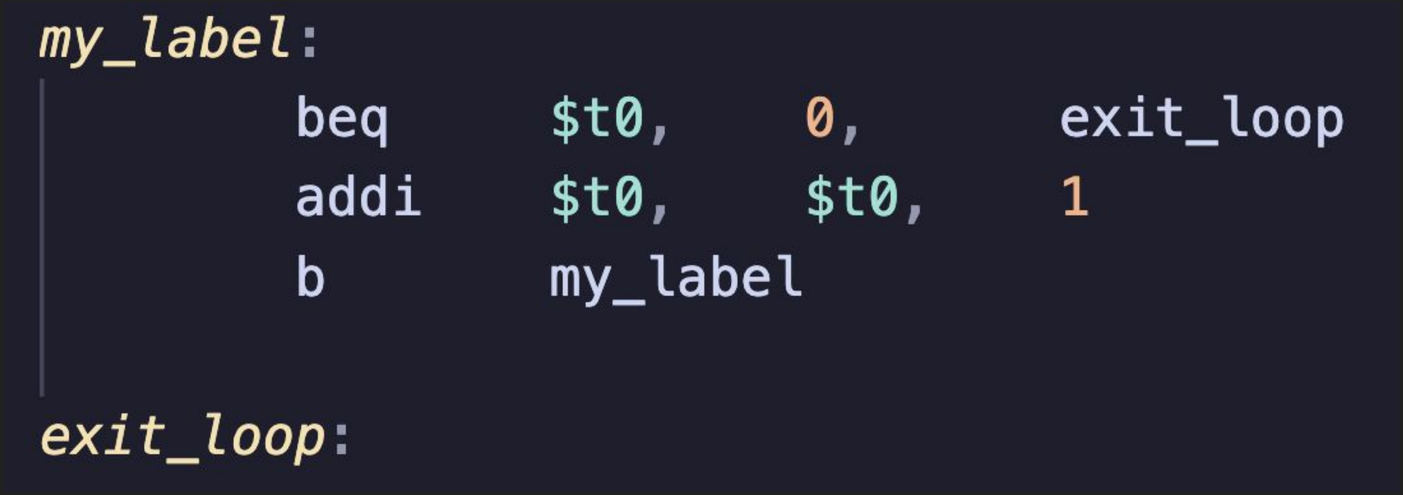


```
my_label:
        b       my_label
```

Loops

Want to exit the loop?

Conditional branch to another label



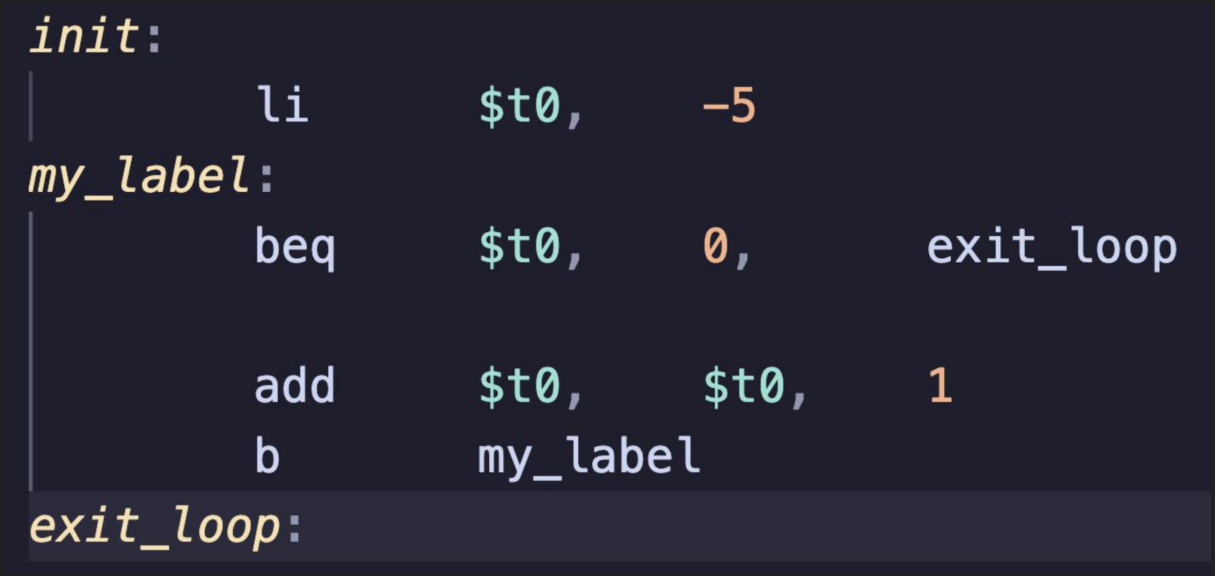
```
my_label:
        beq     $t0,    0,      exit_loop
        addi    $t0,    $t0,    1
        b       my_label

exit_loop:
```

Loops

Maybe we can structure this a bit better…

Let’s add a label to include initialising variables (ie, i = 0)

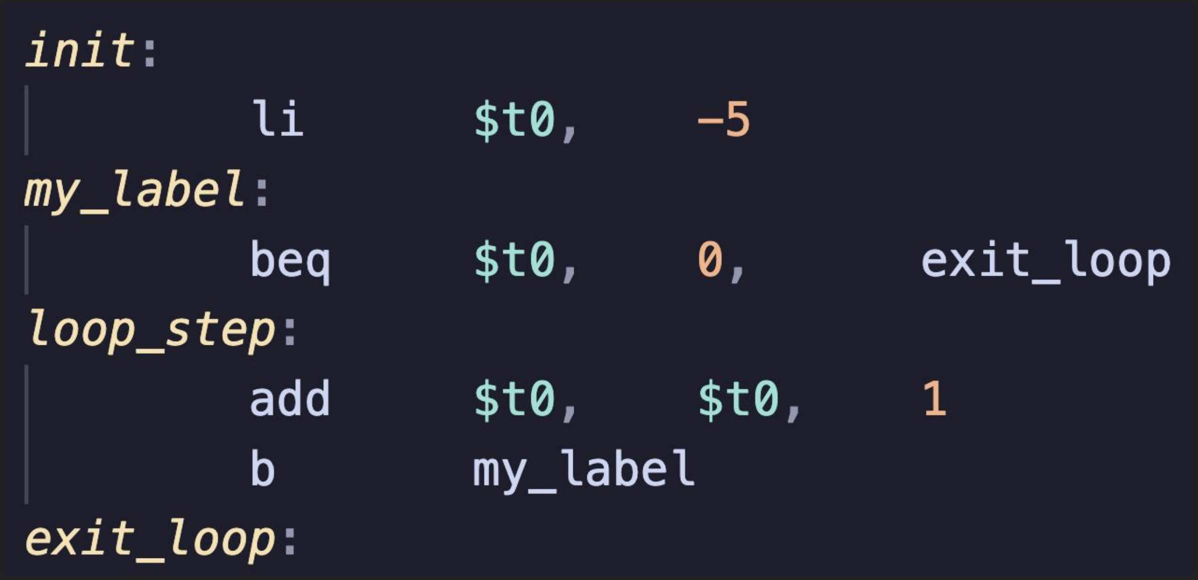


```
init:
        li      $t0,    -5
my_label:
        beq     $t0,    0,      exit_loop

        add     $t0,    $t0,    1
        b       my_label
exit_loop:
```

Loops

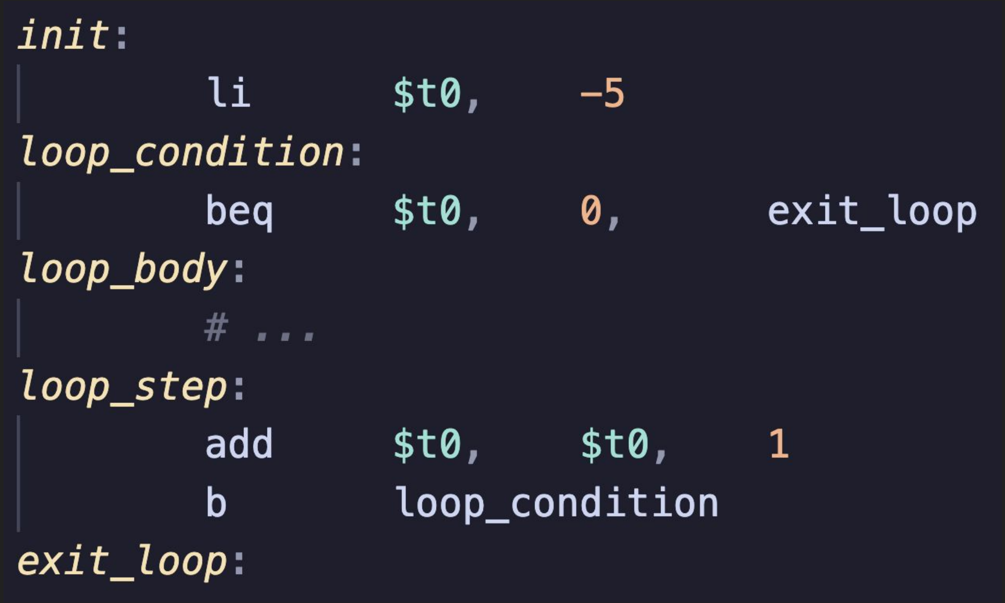
We can also add a label for the increment/branch back to start section



```
init:
        li      $t0,    -5
my_label:
        beq     $t0,    0,      exit_loop
loop_step:
        add     $t0,    $t0,    1
        b       my_label
exit_loop:
```

Loops

Finally let's add a label for the actual contents of the loop (which this loop lacks), as well as renaming the my_label label to something more descriptive.



```
init:
        li      $t0,    -5
loop_condition:
        beq     $t0,    0,      exit_loop
loop_body:
        # ...
loop_step:
        add     $t0,    $t0,    1
        b       loop_condition
exit_loop:
```

Loops

We can follow this general structure for implementing loops (with “name” appropriately replaced) (NOTE recommended structure, but I highly recommend you use this):

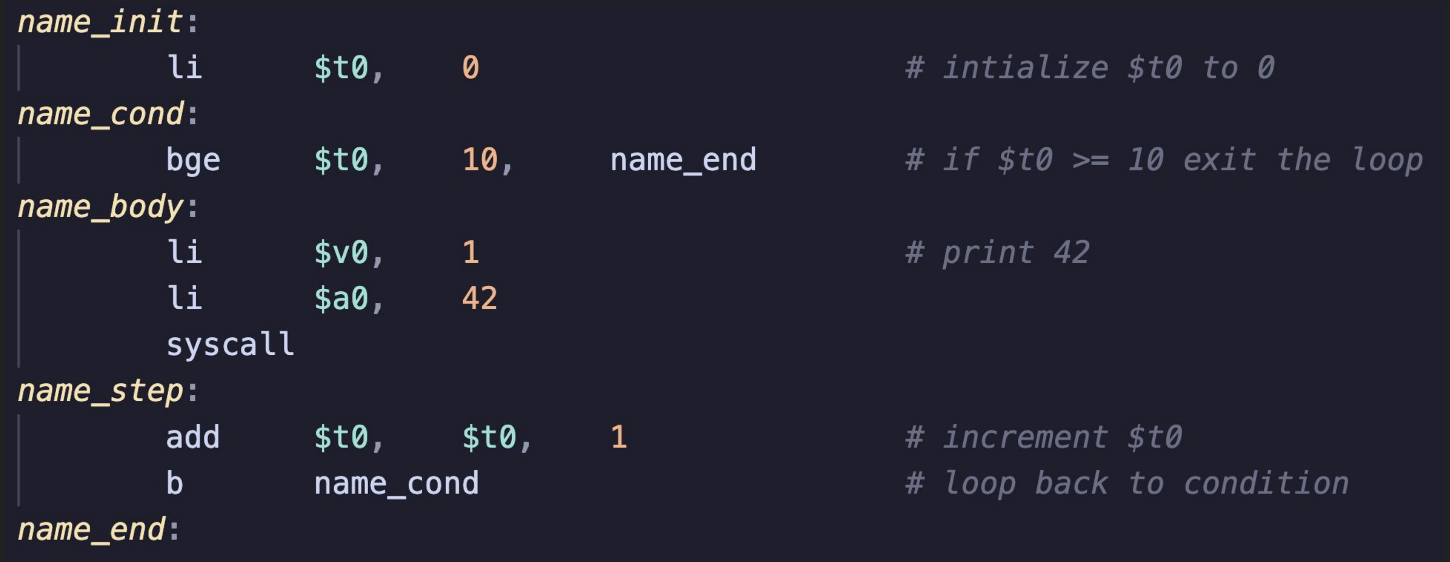
name_init:

name_cond:

name_body:

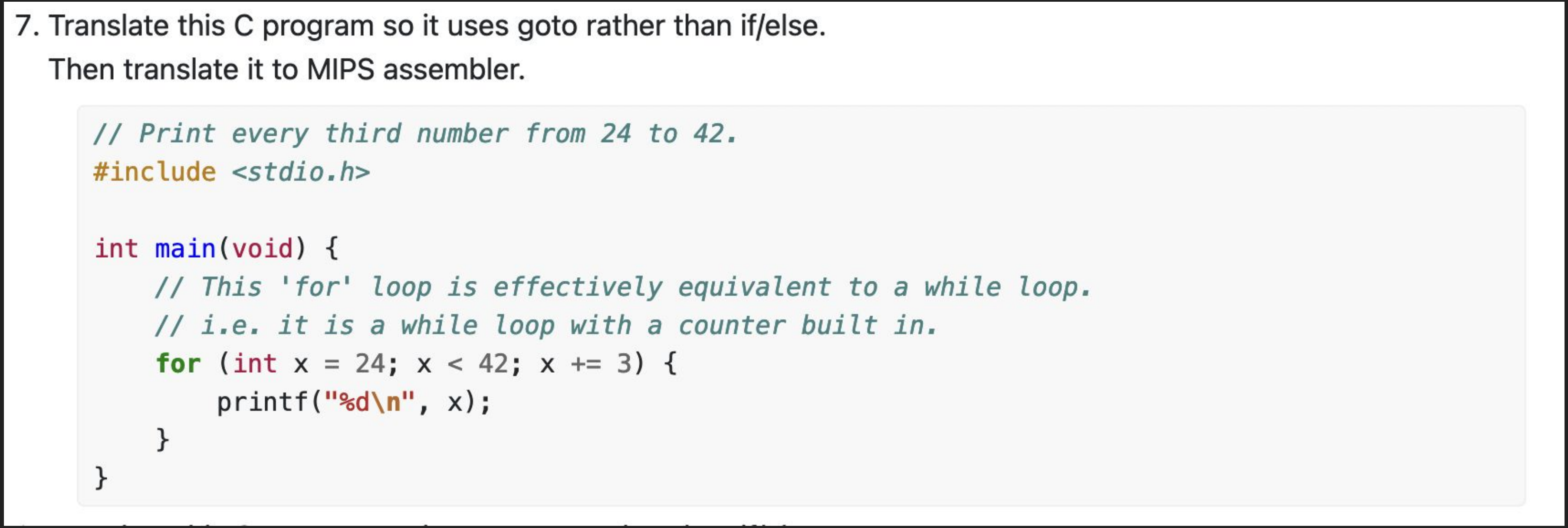
name_step:

name_end:



```
name_init:
        li      $t0,    0                               # intialize $t0 to 0
name_cond:
        bge     $t0,    10,     name_end                # if $t0 >= 10 exit the loop
name_body:
        li      $v0,    1                               # print 42
        li      $a0,    42
        syscall
name_step:
        add     $t0,    $t0,    1                       # increment $t0
        b       name_cond                               # loop back to condition
name_end:
```

Tutorial Q7



7. Translate this C program so it uses goto rather than if/else.

   Then translate it to MIPS assembler.

```
// 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("%d\n", x);
    }
}
```

Congrats

If you made it here we didn’t go over time