INTRODUCTION TO MIPS ASSEMBLY

COMP1521 TUTORIAL 2

01.

MIPS INTRO AND MIPSY

Why learn assembly?

- Optimisation

Quake III's inverse square root

- Reverse engineering

Comp6447

- Embedded Systems

FPGA development, operating systems, computer architecture

Why learn assembly?

- Optimisation

Quake III's inverse square root

- Reverse engineering

Comp6447

- Embedded Systems

FPGA development, operating systems, computer architecture

Why MIPS?

Why learn assembly?

- Optimisation

Quake III's inverse square root

- Reverse engineering

Comp6447

- Embedded Systems

FPGA development, operating systems, computer architecture

Why MIPS?

PS1, PS2, Nintendo 64 were written on MIPS!

MIPSY

MIPSY is a MIPS emulator which simulates the execution of a MIPS CPU and lets you run MIPS assembler on any computer (regardless of native architecture).

- The native architecture of your computer is the design and organisation of your CPU
- MIPS is one such design - others include x86 and ARM :)

MIPSY

MIPSY is a MIPS emulator which simulates the execution of a MIPS CPU and lets you run MIPS assembler on any computer (regardless of native architecture).

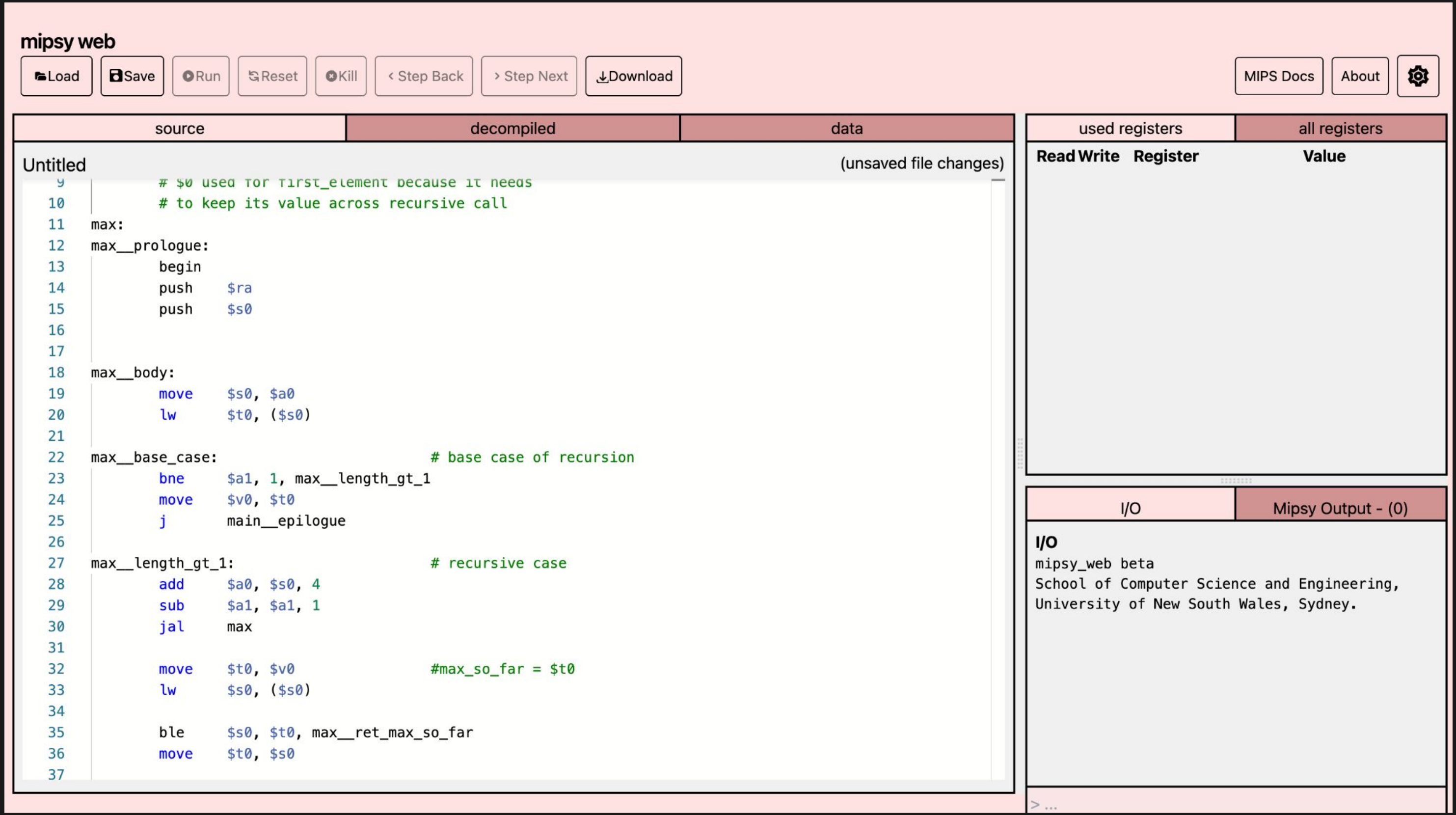
- The native architecture of your computer is the design and organisation of your CPU
- MIPS is one such design - others include x86 and ARM :)
-

MIPSY WEB

mipsy_web is a web-based version of mipsy that is still in very early stages.

- We can use it for debugging!

MIPSY WEB


mipsy web
Load
Save
Run
Reset
Kill
Step Back
Step Next
Download
MIPS Docs
About
source
decompiled
data
Untitled
(unsaved file changes)
$0 used for first_element because it needs
to keep its value across recursive call
max:
max__prologue:
begin
push $ra
push $s0
max__body:
move $s0, $a0
lw $t0, ($s0)
max__base_case: # base case of recursion
bne $a1, 1, max__length_gt_1
move $v0, $t0
j main__epilogue
max__length_gt_1: # recursive case
add $a0, $s0, 4
sub $a1, $a1, 1
jal max
move $t0, $v0 #max_so_far = $t0
lw $s0, ($s0)
ble $s0, $t0, max__ret_max_so_far
move $t0, $s0
used registers
all registers
Read Write
Register
Value
I/O
Mipsy Output - (0)
I/O
mipsy_web beta
School of Computer Science and Engineering,
University of New South Wales, Sydney.

MIPS TOOLS

- VSCODE extension: Mipsy Editor Features (Xavier Cooney)
- CSE COMMAND: 1521 mipsy <name of file>
- MIPSY web: https://cgi.cse.unsw.edu.au/~cs1521/mipsy/
- MIPS Docs: https://cgi.cse.unsw.edu.au/~cs1521/25T2/resources/mips-guide.html

What does MIPS look like?

Rather than using variables to hold values, assembly languages use registers, which are more like physical storage spaces in your CPU

- They don’t get cleaned up like variables - rather we have full control of what happens to them

What does MIPS look like?

Rather than using variables to hold values, assembly languages use registers, which are more like physical storage spaces in your CPU

- 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.

What does MIPS look like?

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`

What does MIPS look like?

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`

$0 - zero
$1 - $at
$2 - $v0
$4 - $a0
$8 - $t0
$16 - $s0
$26 - $k0
$29 - $sp
$31 - $ra

What does an instruction look like?

Let's look at the addi instruction:

```
addi    $t0, $t1, 1
```

What does an instruction look like?

Let's look at the addi instruction:

```
addi    $t0, $t1, 1
```

Operation

What does an instruction look like?

Let's look at the addi instruction:

```
addi    $t0, $t1, 1
```

Operation

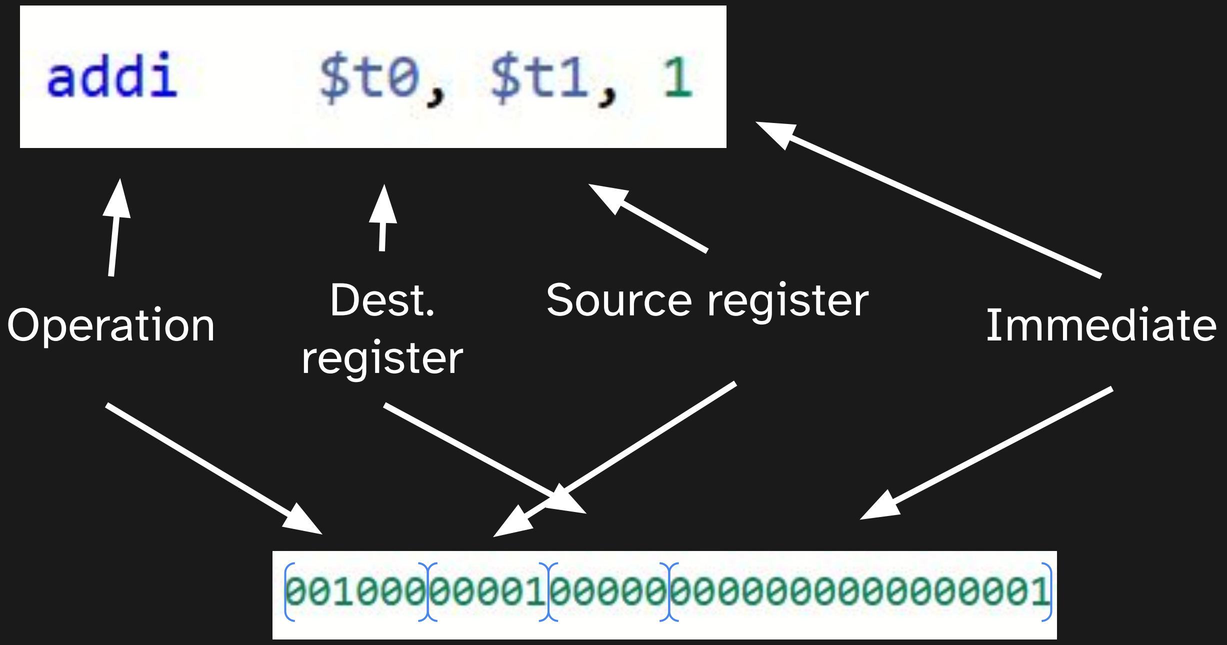
Dest. register

Source register

Immediate

What does an instruction look like?

Let's look at the addi instruction:



C TO MIPS

Translate the following to MIPS assembler.

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

```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;
}
```

C TO MIPS

Translate the following to MIPS assembler.

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

ANSWER:

```
# Prints the square of a number

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

GOTO's

- In COMP1511, we were specifically told NOT to use goto's
- GOTO's are pretty self explanatory - they let you keep executing at a random line in your code (within the same function)
- (please dont use them while actually writing C code tho)

Suuuuuper useful for visualising assembly flow - whyyy?

JUMP INSTRUCTIONS !!! 😱

- They let us jump to any instruction we specify
- Basically a teleportation spell

We don't have if statements in assembly so this is all we have to work with....

✓	J	$Address_{26}$	PC = PC[31-28] && $Address_{26}$ << 2	000010AAAAAAAAAAAAAAAAAAAAAAAAAA

B	$Offset_{16}$	PC += $Offset_{16}$ << 2	00010000000000000000000000000000 BEQ \$t0, \$t0, $Offset_{16}$

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

```c
// Squares a number, unless its square is too big for a 32-bit integer.
// If it is too big, prints an error message instead.

#include <stdio.h>

#define SQUARE_MAX 46340

int main(void) {
    int x, y;

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

    if (x > SQUARE_MAX) {
        printf("square too big for 32 bits\n");
    } else {
        y = x * x;
        printf("%d\n", y);
    }

    return 0;
}
```

```c
// Squares a number, unless its square is too big for a 32-bit integer.
// If it is too big, prints an error message instead.
// Simplified C version.

#include <stdio.h>

#define SQUARE_MAX 46340

int main(void) {
    int x, y;

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

    if (x <= SQUARE_MAX) goto x_le_square_max;

    // This is the "else" part of the if-statement.
    printf("square too big for 32 bits\n");

    goto epilogue;

x_le_square_max:
    // This is the "if-then" part of the if-statement.
    y = x * x;
    printf("%d\n", y);

epilogue:
    return 0;
}
```

```
# Constant
SQUARE_MAX = 46340

main:
        # Locals:
        # - $t0: int x, The number to square.
        # - $t1: int y, The result of the square.

        la      $a0, prompt             # printf("Enter a number: ");
        li      $v0, 4
        syscall

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

        ble     $t0, SQUARE_MAX, x_le_square_max        # if (x <= SQUARE_MAX) goto square;

        la      $a0, too_big            # printf("square too big for 32 bits\n");
        li      $v0, 4
        syscall

        j       epilogue                # goto epilogue;

x_le_square_max:
        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
epilogue:
        jr      $ra                     # return from main

        .data
prompt_str:
        .asciiz "Enter a number: "
too_big_str:
        .asciiz "square too big for 32 bits\n"
```

Thinking in assembly

Which code would be easier to translate into assembly?

```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");
    }
}
```

```c
#include <stdio.h>

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

    char *message = "small/big\n";
    if (x > 100 && x < 1000) {
        message = "medium";
    }

    printf("%s", message);
}
```

Translate the following program into MIPS assembler

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

```
# Prints every 3rd number from 24 to 42

main:                               # int main(void) {
        # Locals:
        #   - $t0: int i, loop counter

count3_loop_init:
        li      $t0, 24             # i = 24;
count3_loop_cond:
                                    # Loop Condition: while(i <= 42)
        bge     $t0, 42, count3_loop_end        # if (i >= 42) goto count3_loop_end;

                                    # Loop Body:
        move    $a0, $t0            # printf("%d" i);
        li      $v0, 1
        syscall

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


                                    # Loop Increment and back to Loop Condition.
        addi    $t0, $t0, 3         # i += 3;
        j       count3_loop_cond        # goto print_loop;
count3_loop_end:
                                    # Loop End:

epilogue:
        jr      $ra                 # return from main.
```

A loop in a loop?

Translate into MIPS...

```
// Prints a right - angled triangle of asterisks, 10 rows high.

#include <stdio.h>

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

MORE MIPS

Translate into MIPS...

```
main:
        # Registers:
        # i in register $t1
        # j in register $t2

        li      $t1, 1                  # i = 1
line_loop_start:
        bgt     $t1, 10, row_loop_end   # if (i > 10) goto row_loop_end;

        li      $t2, 0                  # j = 0

character_loop_start:
        bge     $t2, $t1, character_loop_end    # if (j >= i) goto character_loop_end;

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

        addi    $t2, $t2, 1             # j++

        j       character_loop_start    # goto character_loop_start;

character_loop_end:
                                        # End of character loop.
                                        # Print newline and go to next row.

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

        addi    $t1, $t1, 1             # i++

        j       line_loop_start         # goto line_loop_start;

row_loop_end:
                                        # End of the row loop.

epilogue:
        jr      $ra                     # return from main
```

factorial

Translate into MIPS...

```c
// Simple factorial calculator - without error checking

#include <stdio.h>

int main(void) {
    int n;
    printf("n  = ");
    scanf("%d", &n);

    int fac = 1;
    for (int i = 1; i <= n; i++) {
        fac *= i;
    }

    printf("n! = %d\n", fac);
    return 0;
}
```

factorial

Translate

into

MIPS...

```
main:
        # Registers:
        # - $t0: int n   - number to compute factorial up to
        # - $t1: int i   - number to multiply by in each loop iteration. Serves as loop
counter
        # - $t2: int fac - factorial of $t0

        li      $t0, 0                  # n = 0

        la      $a0, input_msg
        li      $v0, 4
        syscall                         # printf("n  = ")

        li      $v0, 5
        syscall                         # scanf("%d", into $v0)

        move    $t0, $v0

        li      $t2, 1                  # fac = 1

main_fac_init:
                                        # Loop initialisation
        li      $t1, 1                  # i = 1

main_fac_cond:
                                        # Loop condition
        bgt     $t1, $t0, main_fac_end  # while (i <= n)  -->  if (i > n) break
        mul     $t2, $t2, $t1           # fac = fac * i

main_fac_step:
                                        # Loop step and back to the condition
        addi    $t1, $t1, 1             # i++
        j       main_fac_cond

main_fac_end:
                                        # Prints the results
        la      $a0, output_msg
        li      $v0, 4
        syscall                         # printf("n! = ")

        move    $a0, $t2                # assume $t2 holds n!
        li      $v0, 1
        syscall                         # printf("%d", fac)

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

        # la    $a0, newline            # Alternative to print a newline using a string:
        # li    $v0, 4
        # syscall                       # printf("\n")

        jr      $ra                     # return from main

        .data
```