COMP1521 24T2 Lec02

MIPS:
Basics + Control

2024
Hammond Pearce
Adapted from Abiram’s slides
Lecture chat

- Place to ask questions/make comments in the lecture (mostly) anonymously, if you like
  - Can deanonymise if the need arises - please follow UNSW Code of Conduct
  - Don’t spam
  - Supports Discord Markdown!
- Mild shitposting is fine, in moderation
- Don’t make us blacklist you >:(
Lecture chat

https://cgi.cse.unsw.edu.au/~cs1521/accord/
Recap of lec01

- Exploring different types of storage/memory
- RAM contains everything a program needs in a given moment
- Instructions!
- Assembly language!
- Registers!
- System calls!
The system call workflow

- We specify which system call we want in $v0
  - eg. `print_int` is syscall 1:
    
    ```
    li $v0, 1
    ```

- We specify arguments (if any)
  
    ```
    li $a0, 42
    ```

- We transfer execution to the operating system
  - The OS will fulfil our request if it looks sane
    ```
    syscall
    ```

- Some syscalls may return a value - check syscall table
Recap exercise

- Open Mipsy
- Store the value “1” in $t0
- Store the value “4” in $t1
- Sum these and print the result to the I/O window
Recap exercise

```
.text
main:
    li  $t0, 1  #t0 = 1
    li  $t1, 4  #t1 = 4
    add  $a0, $t0, $t1  #a0 = t0 + t1 (5)
    li  $v0, 1  #
    syscall  # syscall 1 - print_int
    li  $v0, 0  #
    jr  $ra  # exit the program
```
DISCLAIMER:

Code written in lectures may not necessarily have the best style!

- Lecture code is meant to be quick and dirty, to demonstrate a concept
- Will quickly overview good style soon, but refer to your tutor, tut solutions, lab solutions
li vs la vs move

- **li** (load immediate) is for immediate, *fixed values* that you need to load into a register with an instruction.
- **la** (load address) is for loading *fixed addresses* into a register.
  - Remember, labels really just represent addresses!
- **move** is for copying values *between two registers*. 
Syntax - Assembly language contains:

- **Assembly instructions**, each on their own line
  - Generally a 1:1 mapping from CPU instructions to real instructions
  - However, assemblers also provide **pseudo-instructions** for convenience
  - Some of pseudo-instructions turn into 2-3 real CPU instructions
    - `li` is an example - ask why on the forum if curious!
- **Labels** ... appended with : 
- **Comments** ... starting with a #
- **Directives** ... symbol beginning with .
- **Constant definitions** - like #defines in C: \[\text{MAX\_NUMBERS} = 256\]
Style

- We generally don’t indent to show structure
  - i.e. no indenting within conditionals, if statements, etc.
- Instead:
  - don’t indent labels
  - indent instructions by one step
  - have equivalent C code as inline comments
- Huge recommendation: indent with 8-wide tabs
  - Ask on forum if anyone wants my vscode config
Simplified C

Translating C code directly to MIPS is not fun

Pro strat - simplify your C code and then translate it:

- Map down to ‘simplified’ C
  - Simplified C is generally written so that each line of C code maps to one MIPS instruction
  - Compile your simplified C and make sure it still works as expected
  - Translate each line of simplified C to MIPS
  - Profit!!
Example: square_and_add
MIPS Control
So far...

Our programs have implemented fixed, predictable behaviour.

- Execute linearly - we always go down to the next instruction

However, what if we want to implement logic in our code?

- if statements - conditional code execution
- for/while loops - repeat some instructions?

if/else and loops don't exist in MIPS - we have to use branching to implement these ourselves
### Branch/jump instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Binary Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b label</code></td>
<td>pc += I&lt;&lt;2</td>
<td>pseudo-instruction</td>
<td>Allows you to transfer the flow of execution to a different instruction <strong>conditionally</strong> except <code>b</code>, which is unconditional</td>
</tr>
<tr>
<td><code>beq rs, rt, label</code></td>
<td>if (<code>rs == rt</code>)</td>
<td>000100ssssstttt IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td></td>
</tr>
<tr>
<td><code>bne rs, rt, label</code></td>
<td>if (<code>rs != rt</code>)</td>
<td>000101ssssstttt IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td></td>
</tr>
<tr>
<td><code>ble rs, rt, label</code></td>
<td>if (<code>rs &lt;= rt</code>)</td>
<td>pseudo-instruction</td>
<td></td>
</tr>
<tr>
<td><code>bgt rs, rt, label</code></td>
<td>if (<code>rs &gt; rt</code>)</td>
<td>pseudo-instruction</td>
<td></td>
</tr>
<tr>
<td><code>blt rs, rt, label</code></td>
<td>if (<code>rs &lt; rt</code>)</td>
<td>pseudo-instruction</td>
<td></td>
</tr>
<tr>
<td><code>bge rs, rt, label</code></td>
<td>if (<code>rs &gt;= rt</code>)</td>
<td>pseudo-instruction</td>
<td></td>
</tr>
<tr>
<td><code>blez rs, label</code></td>
<td>if (<code>rs &lt;= 0</code>)</td>
<td>000110ssss000000IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td></td>
</tr>
<tr>
<td><code>bgtz rs, label</code></td>
<td>if (<code>rs &gt; 0</code>)</td>
<td>000111ssss000000IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td></td>
</tr>
<tr>
<td><code>bltz rs, label</code></td>
<td>if (<code>rs &lt; 0</code>)</td>
<td>000000ssss000000IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td></td>
</tr>
<tr>
<td><code>bgez rs, label</code></td>
<td>if (<code>rs &gt;= 0</code>)</td>
<td>000001ssss000000IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td></td>
</tr>
<tr>
<td><code>bnez rs, label</code></td>
<td>if (<code>rs != 0</code>)</td>
<td>000001ssss00001IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td></td>
</tr>
<tr>
<td><code>beqz rs, label</code></td>
<td>if (<code>rs == 0</code>)</td>
<td>000001ssss00000IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII</td>
<td></td>
</tr>
</tbody>
</table>

- **Also** `j, jal, jalr, jr` - unconditional jump instructions which we will talk about in MIPS Functions
- **Can replace** `rt` with a constant in mipsy
In other words

A lot of these branch instructions are of the form:

“if condition is true, jump to instruction”

How do we implement this for our simplified C code?
COMP1511 staff hate this one simple trick!

In C, `goto` allows jumping to any arbitrary label within a program.

This means we can effectively yeet around within a program however we wish.
Simplifying if, if/else:

Example: print_if_even
With great power comes great responsibility

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Go To Considered Harmful (1968)
Don’t (ab)use goto

don’t use it in your actual C programs.

- **goto** makes programs more difficult to read
- **goto** makes it hard for compilers to optimise code, resulting in slower programs
- In general, do not use **goto** without good reason!
  - Typically only kernel/embedded programmers use goto
More complex conditionals:

Split combined “or” conditions

```c
if (milk_age > 48 || milk_level < 10) {
    printf("Replace milk\n");
} else {
    printf("Milk okay!\n");
}
printf("Done!\n");
```
More complex conditionals:

Split combined “or” conditions

```c
if (milk_age > 48 ||
    milk_level < 10) {
    printf("Replace milk\n");
} else {
    printf("Milk okay!\n");
}
printf("Done!\n");
```

```c
if (milk_age > 48) goto milk_replace;
if (milk_level < 10) goto milk_replace;
printf("Milk okay!\n");
goto milk_replace__end;

milk_replace:
    printf("Replace milk\n");

milk_replace__end:
    printf("Done!\n");
```
More complex conditionals: &&

Invert the condition to use || (De Morgan’s Law)

```java
if (x >= 0 && x <= 100) {
    // in bounds
} else {
    // out of bounds
}
return 0;
```
More complex conditionals: &&

Invert the condition to use || (De Morgan’s Law)

```java
if (x >= 0 && x <= 100) {
    // in bounds
} else {
    // out of bounds
}
return 0;
```

```java
if (x < 0 || x > 100) {
    // out of bounds
} else {
    // in bounds
}
return 0;
```
More complex conditionals:

Split into separate conditionals:

```plaintext
if (x < 0 || x > 100) {
    // out of bounds
} else {
    // in bounds
}
return 0;
```
More complex conditionals:

Split into separate conditionals:

```c
if (x < 0 || x > 100) {
    // out of bounds
} else {
    // in bounds
}
return 0;
```

```c
if (x < 0) goto x_out_of_bounds;
if (x > 100) goto x_out_of_bounds;

// in bounds
goto epilogue;

x_out_of_bounds:
    // out of bounds

epilogue:
    return 0;
```
Your turn

```java
if (y < 10 || z > 50) {
    // condition met
} else {
    // condition not met
}
return 1;
```
if (y < 10 || z > 50) {
    // condition met
} else {
    // condition not met
}
return 1;

if (y < 10) goto condition_met;
if (z > 50) goto condition_met;
goto condition_not_met;
condition_met:
    // condition met
    goto epilogue;
condition_not_met:
    // condition not met
epilogue:
    return 1;
if (y < 10 || z > 50) {
    // condition met
} else {
    // condition not met
}
return 1;

if (y < 10) goto condition_met;
if (z > 50) goto condition_met;
// condition not met
goto epilogue;
condition_met:
    // condition met

epilogue:
return 1;
if (y < 10 || (z > 50 && w < 5)) {
    // condition met
} else {
    // condition not met
}
return 1;
if (y < 10 || (z > 50 && w < 5)) {
    // condition met
} else {
    // condition not met
}
return 1;

if (y < 10) goto condition_met;
if (z <= 50) goto condition_not_met;
if (w >= 5) goto condition_not_met;

condition_met:
    // condition met
    goto epilogue;
condition_not_met:
    // condition not met

epilogue:
    return 1;
Simplifying loop structures

- **for** loops should be broken down to **while** loops
- **while** loops should be broken down into **if/goto**

General structure:

- loop init
- loop condition (do we need to exit the loop?)
- loop body
- loop step
- loop end

Use labels to show structure!
Simplifying for loops:

sum_100_squares
Counting to 10

```c
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

```c
int i = 0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```
Counting to 10

```
int i = 0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

```
loop_i_to_10__init:
    int i = 0;
loop_i_to_10__cond:
    if (i >= 10) goto loop_i_to_10__end;
loop_i_to_10__body:
    printf("%d", i);
    putchar('\n');
loop_i_to_10__step:
    i++;
loop_i_to_10__end:
    // ...
```
Sidenote: C break/continue

`break` can be used in a loop to completely exit the loop.

The loop condition here makes this look like an infinite loop:

```c
while (1) {
    int c = getchar();
    if (c == EOF) break;
}
```

but `break` means it’s possible for the loop to be exited.

In simplified C/MIPS, a `break` is really just equivalent to going to the loop’s end label.
Sidenote: C break/continue

`continue` can be used to proceed to the next iteration of a for loop.

This would be a (terrible) way to print even numbers:

```c
for (int i = 0; i < 10; i++) {
    if (i % 2 != 0) continue;
    printf("%d\n", i);
}
```

In simplified C/MIPS, a `continue` is really just equivalent to going to the loop’s step label.