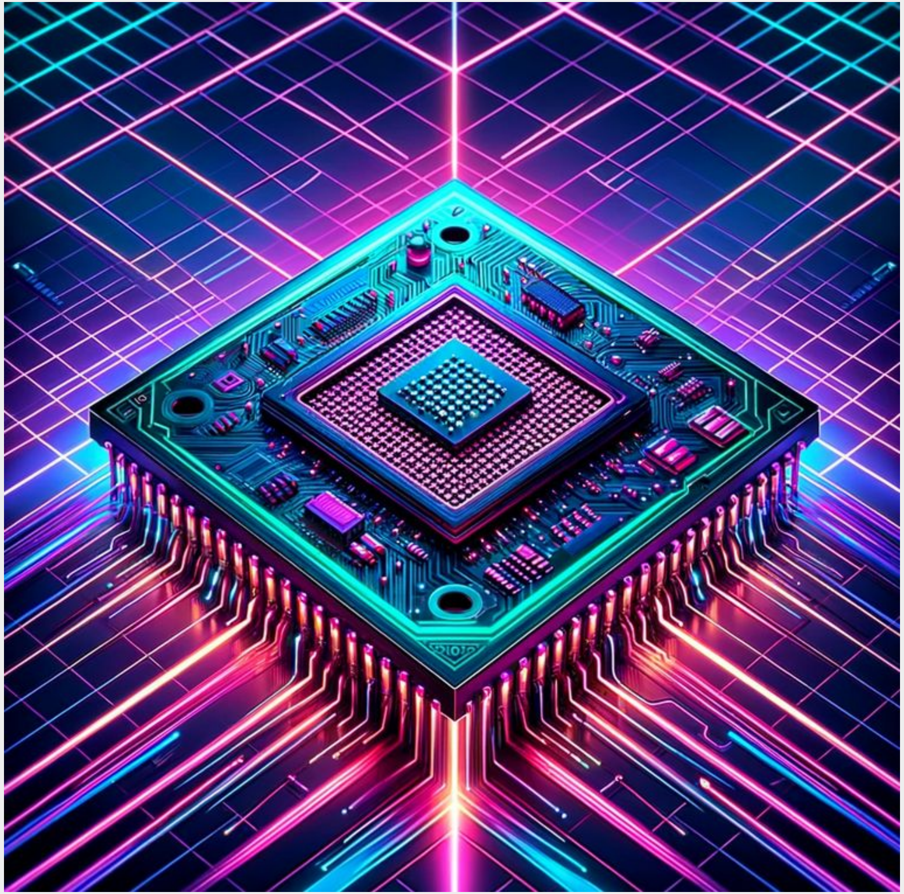
COMP1521 24T2 Lec02

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: `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.

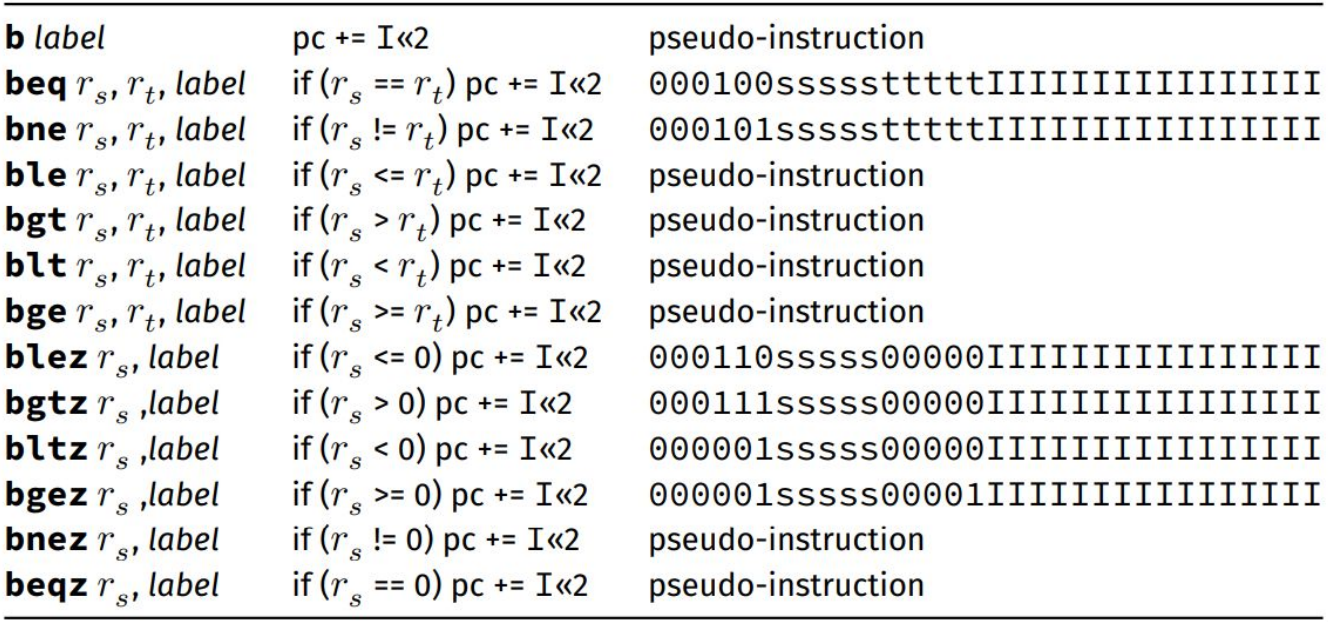
- Execute linearly - we always go down to the next instruction

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



b *label*	pc += I«2	pseudo-instruction
beq r_s, r_t, *label*	if (r_s == r_t) pc += I«2	`000100ssssstttttIIIIIIIIIIIIIIII`
bne r_s, r_t, *label*	if (r_s != r_t) pc += I«2	`000101ssssstttttIIIIIIIIIIIIIIII`
ble r_s, r_t, *label*	if (r_s <= r_t) pc += I«2	pseudo-instruction
bgt r_s, r_t, *label*	if (r_s > r_t) pc += I«2	pseudo-instruction
blt r_s, r_t, *label*	if (r_s < r_t) pc += I«2	pseudo-instruction
bge r_s, r_t, *label*	if (r_s >= r_t) pc += I«2	pseudo-instruction
blez r_s, *label*	if (r_s <= 0) pc += I«2	`000110sssss00000IIIIIIIIIIIIIIII`
bgtz r_s ,*label*	if (r_s > 0) pc += I«2	`000111sssss00000IIIIIIIIIIIIIIII`
bltz r_s ,*label*	if (r_s < 0) pc += I«2	`000001sssss00000IIIIIIIIIIIIIIII`
bgez r_s ,*label*	if (r_s >= 0) pc += I«2	`000001sssss00001IIIIIIIIIIIIIIII`
bnez r_s, *label*	if (r_s != 0) pc += I«2	pseudo-instruction
beqz r_s, *label*	if (r_s == 0) pc += I«2	pseudo-instruction

- Allows you to transfer the flow of execution to a different instruction ***conditionally***
  - except **b**, which is unconditional
- Also **j**, **jal, jalr, jr** - unconditional jump instructions which we will talk about in MIPS Functions
- Can replace r_t 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!

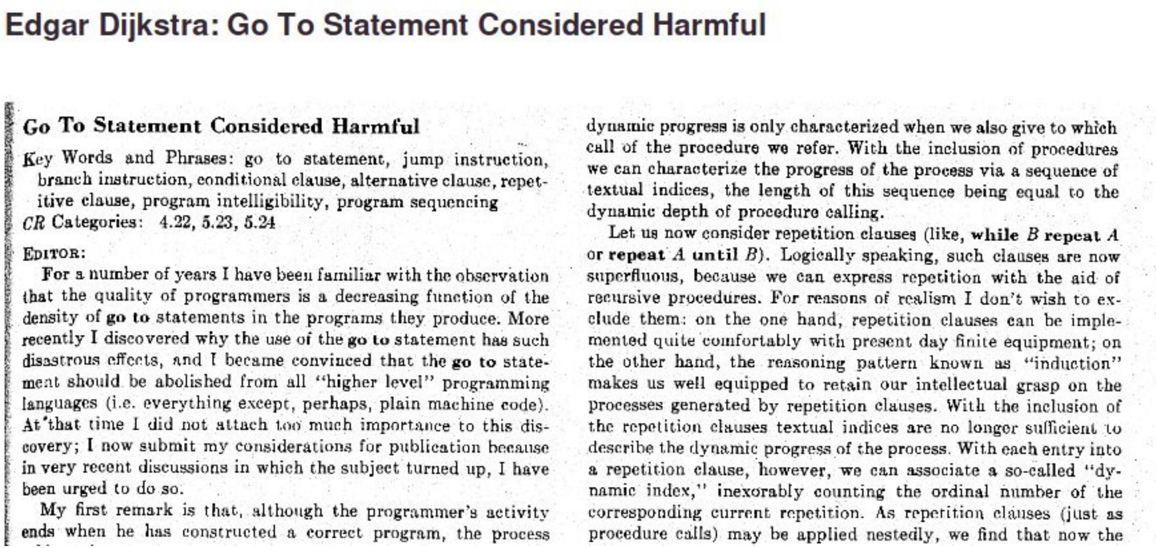
In C, `goto` allows jumping to any arbitrary label within a program.

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

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the

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

```
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

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

printf("Done!\n");
```

```
    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!");
```

More complex conditionals: &&

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

```
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)

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

return 0;
```

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

return 0;
```

More complex conditionals:

Split into separate conditionals:

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

return 0;
```

More complex conditionals:

Split into separate conditionals:

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

return 0;
```

```
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

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

Your turn

```
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;
```

Your turn

```
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;
```

Your turn

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

Your turn

```
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

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

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

Counting to 10

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

```c
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:

```
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:

```
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.