COMP1521 25T2

Week 1 Lecture 2

MIPS: Basics and Control

Adapted from slides by Angela Finlayson, Abiram Nadarajah, Hammond Pearce, Andrew Taylor and John Shepherd's slides

ATT: CompEng students

- You may find the Computer Engineering Resource Page useful: https://cgi.cse.unsw.edu.au/~compbh/

- It provides:
    - Advice on purchasing personal computers
    - How to structure your degree
    - Which electives to take

C Revision and recursion Lab Week 2

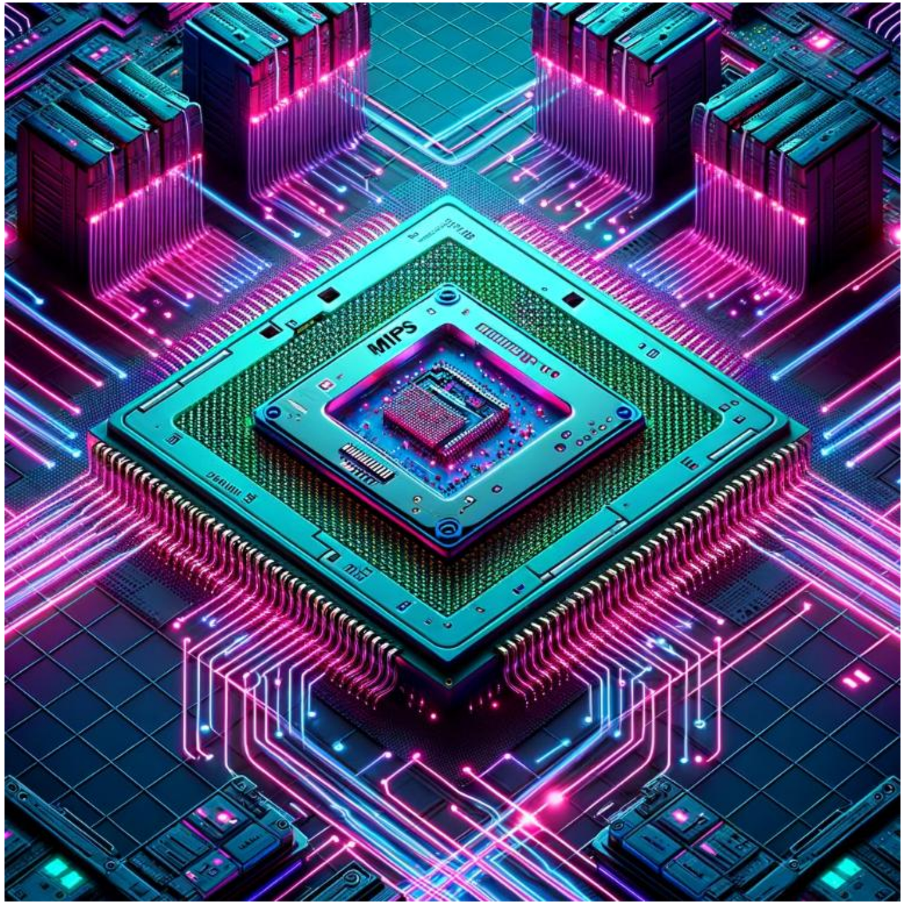
- Thursday 10am - 12pm week 2
- Online via BlackBoard Collaborate

Content:

  - C revision and recursion lab style based questions
  - Can also get help with regular lab week 1 if struggling

Today's Lecture

- Recap Lecture 1
- System Calls
- Style
- Simplified C
  - and goto
- MIPS Control
  - if statements
  - boolean expressions
  - while loops/for loops



DISCLAIMER:

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

Refer to:

- C Style guide
- Assembly style guide
- Your tutor, tut solutions, lab solutions and assignment resources.

Recap of Lecture 1: Intro to MIPS

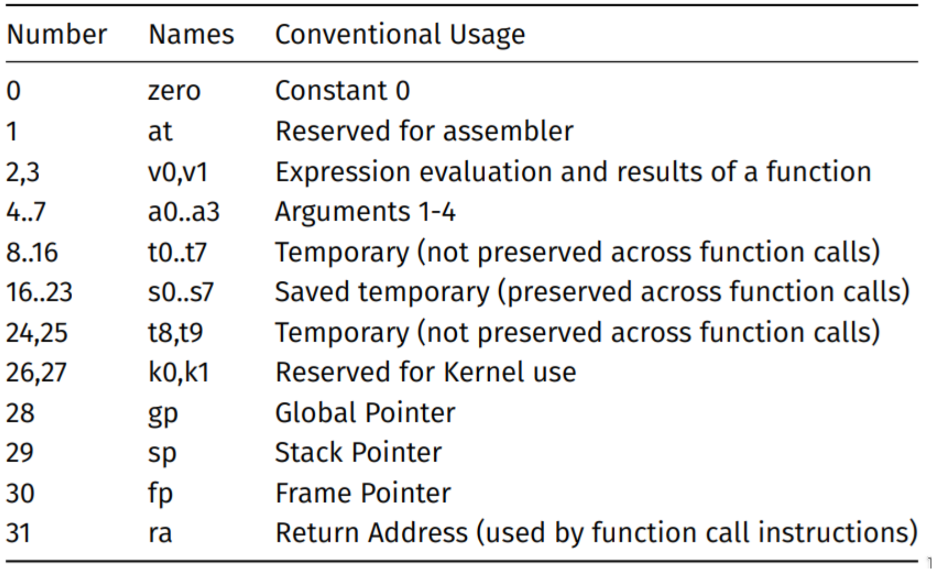
- Explored different types of storage/memory
- Loading programs into RAM for execution
- The different program segments (code, data, heap, stack)
- Hexadecimal number representation
- Machine instructions
    - Unique to the Instruction Set Architecture (ISA) of the CPU
    - E.g. (x86, MIPS, ARM, RISC-V)
- Assembly language (MIPS)
- MIPS emulation using mipsy_web (and others)

More about registers

Registers have symbolic names and also numeric names

$t0 is also known as $8

There are many registers we won't learn about or use until week 3.



Number	Names	Conventional Usage
0	zero	Constant 0
1	at	Reserved for assembler
2,3	v0,v1	Expression evaluation and results of a function
4..7	a0..a3	Arguments 1-4
8..16	t0..t7	Temporary (not preserved across function calls)
16..23	s0..s7	Saved temporary (preserved across function calls)
24,25	t8,t9	Temporary (not preserved across function calls)
26,27	k0,k1	Reserved for Kernel use
28	gp	Global Pointer
29	sp	Stack Pointer
30	fp	Frame Pointer
31	ra	Return Address (used by function call instructions)

Back to mipsy

MIPS Computations with Registers

Almost ***all*** of our computations happen between registers!

Want to multiply 2 and 3 and store the result

```
li $t0, 2              # load 2 into register $t0
li $t1, 3              # load 3 into register $t1
mul $t2, $t0, $t1      # Store $t0 x $t1 into $t2
```

Your turn!

Code this up in mipsy_web.

- Set $t0 to 10
- Set $t1 to 7
- Subtract $t1 from $t0 and store in $t2
- Add 5 to $t2

What expression is this equivalent to?

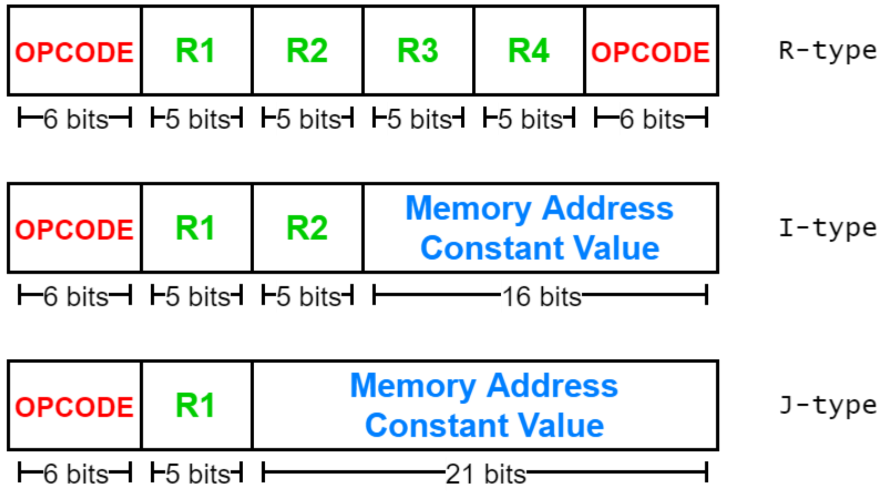
Do you end up with the correct answer in $t2?

Assembly Syntax overview

- **Assembly instructions**, each on their own line
- Generally a 1:1 mapping from assembly instructions to binary instructions
- However, assemblers also provide **pseudo-instructions** for convenience
- pseudo-instructions turn into 1-3 real CPU instructions
  - Example:
    - li $t0, 5 gets mapped to real equivalent CPU instruction
    - addi $t0, $zero, 5
  - You will see many more as you write more code in MIPS.

What do MIPS instructions look like?

- 32 bits long
- Specify:
    - An operation
        - (The thing to do)
    - 0 or more operands
        - (The thing to do it over)



- For example:

```
00100001000010010000000000001100
```

```
addi $t1, $t0, 12
```

But how can we do input / output?

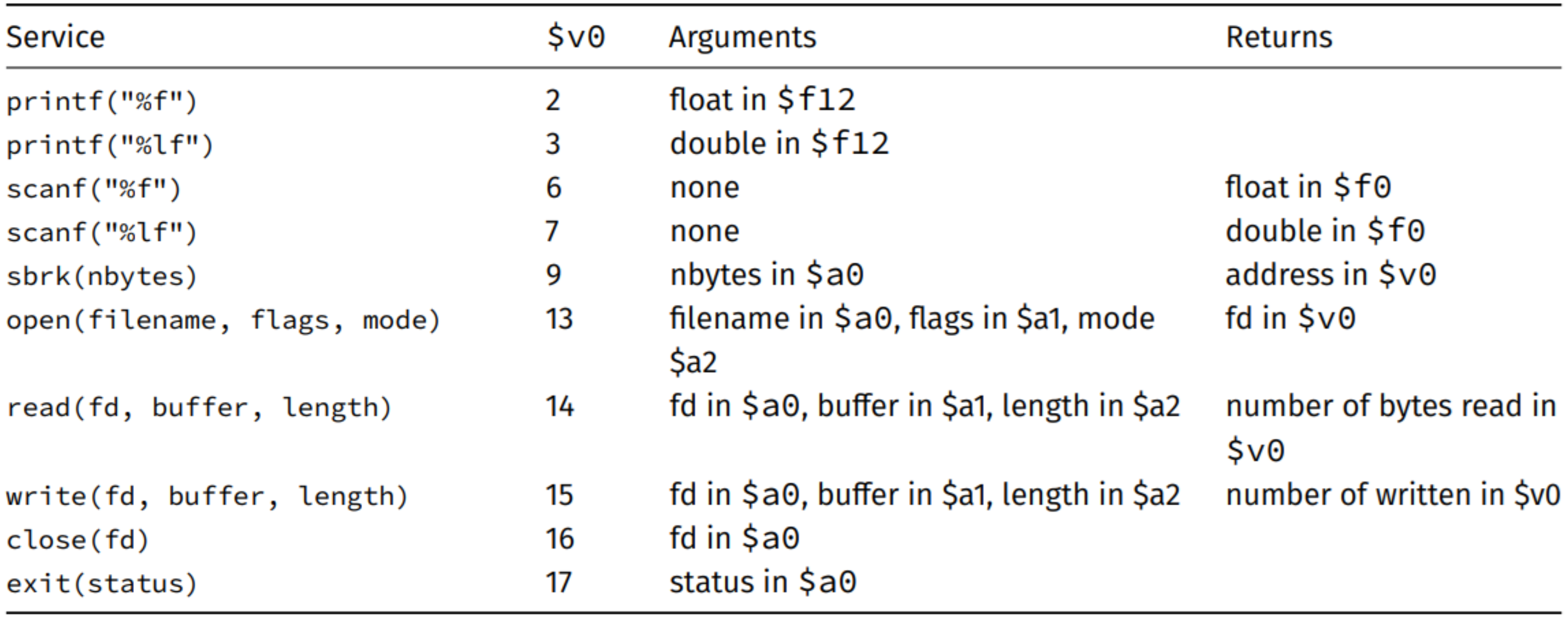
System calls

- None of the instructions we have access to can interact with the outside world (eg. printing, scanning)
- Instead, we request the operating system to perform these tasks for us - this process is called a **system call**
- The operating system can access privileged instructions on the CPU (eg. communicating to other devices)
- *mipsy* simulates a very basic operating system
- We will explore real system calls in the second half of the course

Common mipsy syscalls

Service	$v0	Arguments	Returns
`printf("%d")`	1	int in `$a0`	
`fputs`	4	string in `$a0`	
`scanf("%d")`	5	none	int in `$v0`
`fgets`	8	line in `$a0`, length in `$a1`	
`exit(0)`	10	none	
`printf("%c")`	11	char in `$a0`	
`scanf("%c")`	12	none	char in `$v0`

More ✨advanced✨ syscalls



Service	`$v0`	Arguments	Returns
`printf("%f")`	2	float in `$f12`	
`printf("%lf")`	3	double in `$f12`	
`scanf("%f")`	6	none	float in `$f0`
`scanf("%lf")`	7	none	double in `$f0`
`sbrk(nbytes)`	9	nbytes in `$a0`	address in `$v0`
`open(filename, flags, mode)`	13	filename in `$a0`, flags in $a1, mode $a2	fd in `$v0`
`read(fd, buffer, length)`	14	fd in `$a0`, buffer in $a1, length in $a2	number of bytes read in `$v0`
`write(fd, buffer, length)`	15	fd in `$a0`, buffer in $a1, length in $a2	number of written in $v0
`close(fd)`	16	fd in `$a0`	
`exit(status)`	17	status in `$a0`	

Probably only used for challenge exercises in COMP1521

Let’s try to print out the number 42

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 fulfill our request if it looks sane
    - `syscall`
- Some syscalls may return a value - check syscall table

Let’s try to print out the number 42

“Hello COMP1521!!”

Printing Strings and the Data segment

- We need to define our string in the data section
- Then pass the address of our string to our system call in $a0
- We need to use the `.data` directive so we can create global data in our program
- We need to use the `.asciiz` directive so we can define a string and give the string a label!
- We need to use the **`la`** to load the address of the string!!

Hello COMP1521 revisited

```
        .text
main:
        li      $v0, 4              # syscall 4: print_string
        la      $a0, hello_msg      #
        syscall                     # printf("Hello COMP1521!!\n");

        li      $v0, 0
        jr      $ra                 # return 0;

        .data
hello_msg:
        .asciiz "Hello COMP1521!!\n"
```

Example: Integer Average

```c
// Translate into MIPS
int main(void) {
    int a, b;
    printf("Enter a number: ");
    scanf("%d", &a);
    printf("Enter another number: ");
    scanf("%d", &b);
    printf("The average is %d\n", (a + b)/ 2);
    return 0;
}
```

Assembly Language Syntax Recap

- **li** (load immediate) is loading a **fixed value** into a register
  - `li $t0, 7`
- **la** (load address) is for loading a **fixed address** into a register
  - remember, labels really just represent addresses!
  - `la $t0, my_label`
- **move** is for copying value from a **register** into another register
  - `move $t0, $t1`

Assembly Language Syntax Recap

- Labels
  - Appended with :
  - They represent memory addresses
- Comments
  - Start with #
- Directives
  - Symbols beginning with . eg **`.asciiz .text .data`**
- Constant definitions
  - Like #define in C e.g.
  - **`MAX_NUMBERS = 10`**

MIPS Control

So far

- Our programs only execute **linearly**

- How can we **conditionally** execute code?
- How can we **loop** over code?

Branch Instructions

- We have many conditional branch instructions of the form:
  - "if condition is true, jump to offset"

`BGE` R_s, R_t, Offset_{16}

```
IF Rs >= Rt THEN
```

$\texttt{PC += } \mathit{Offset}_{16} \texttt{ << 2}$

- We have an unconditional branch instruction too

`J` $\mathit{Address}_{26}$

$\texttt{PC = PC[31-28] \&\& } \mathit{Address}_{26} \texttt{ << 2}$

- Thankfully, *offset* and *Address* can be replaced by a label:

Simplified C

- Translating C code directly to MIPS is challenging

- Simplify your C code and then translate it to “simplified C”:
    - Each line of Simplified C to map to one MIPS instruction
    - Compile your simplified C and make sure it still works
    - Translate each line of simplified C to MIPS

COMP1511 staff hid this simple trick!

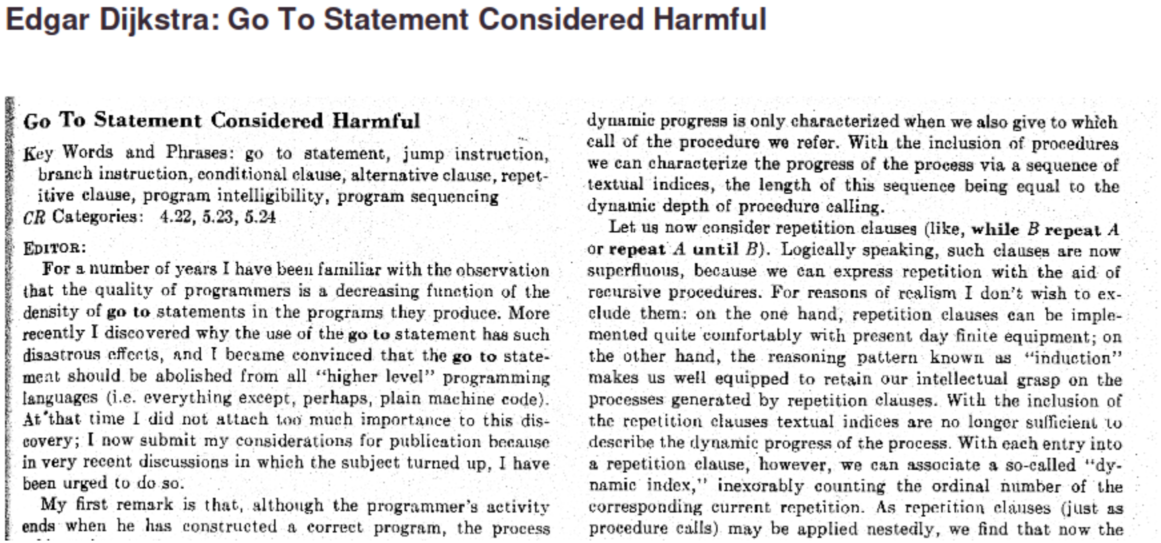
In C, `goto` allows jumping to any arbitrary label within a program.

This means we can effectively jump around within a program however we wish.

What will this code do?

```c
int main(void) {
    goto sleep;
    printf("Please pay close attention\n");
sleep:
    printf("You are getting sleepy\n");
    goto sleep;
    printf("Please wake up now!");
    return 0;
}
```

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
- In general, do not use **goto** without good reason!
- We will use it in this course ONLY for writing simplified C to translate into MIPS.

Simplifying if-else statements

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

    if (n % 2 == 0) {
        printf("even\n");
    }
    return 0;
}
```

```
int main(void){
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);
    int tmp = n % 2;
    if (tmp != 0) goto if_even_end;
        printf("even\n");
if_even_end:
    return 0;
}
```

Now we can write it in MIPS.
Exercise: add an else statement for odd numbers

Style

- Have equivalent C code as *inline comments*
- Huge recommendation: indent with 8-wide tabs
- We generally don't indent to show structure
    - i.e no indenting within loops or if statements, etc.
- Instead:
    - don't indent labels
    - indent instructions by one step
- For this course: focus on readable code, not reducing number of registers used or lines of code

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

```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!");
```

More complex conditionals: &&

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

```c
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)

(A && B) becomes !(!A || !B)

```
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)

(A && B) becomes !(!A || !B)

```
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

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

```c
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

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

```c
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: Counting

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

$\Rightarrow$

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

Simplifying for loops: Counting

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

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

Beware: Don't forget the i++ when converting to a while loop

Counting

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

```c
int i;
loop_i_to_10_init:
    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++;
    goto loop_i_to_10_cond;
loop_i_to_10_end:

    // ...
```

Exercise: Sum 100 squares

Convert to MIPS

```c
int sum = 0;
for (int i = 1; i <= 100; i++) {
    sum += i * i;
}
```

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.

What did we learn today?

- MIPS
  - Recap of basics from lecture 1
  - System calls
    - printing out and reading in integers, and chars
    - printing out strings
  - Branches
  - Simplified C
  - Control
    - goto statements
    - if statements,
    - loops

Reach Out

Content Related Questions:

[Forum]

Admin related Questions email:

cs1521@cse.unsw.edu.au

Student Support | I Need Help With...

My Feelings and Mental Health Managing Low Mood, Unusual Feelings & Depression	**Mental Health Connect**	student.unsw.edu.au/**counselling** Telehealth	**In Australia Call Afterhours UNSW Mental Health Support Line**	1300 787 026 5pm-9am
	Mind HUB	student.unsw.edu.au/**mind-hub** Online Self-Help Resources	**Outside Australia Afterhours 24-hour Medibank Hotline**	+61 (2) 8905 0307
Uni and Life Pressures Stress, Financial, Visas, Accommodation & More	**Student Support Indigenous Student Support**	— student.unsw.edu.au/**advisors**		
Reporting Sexual Assault/Harassment	**Equity Diversity and Inclusion (EDI)**	— edi.unsw.edu.au/**sexual-misconduct**		
Educational Adjustments To Manage my Studies and Disability / Health Condition	**Equitable Learning Service (ELS)**	— student.unsw.edu.au/**els**		
Academic and Study Skills	**Academic Language Skills**	— student.unsw.edu.au/**skills**		
Special Consideration Because Life Impacts our Studies and Exams	Special Consideration	— student.unsw.edu.au/**special-consideration**		