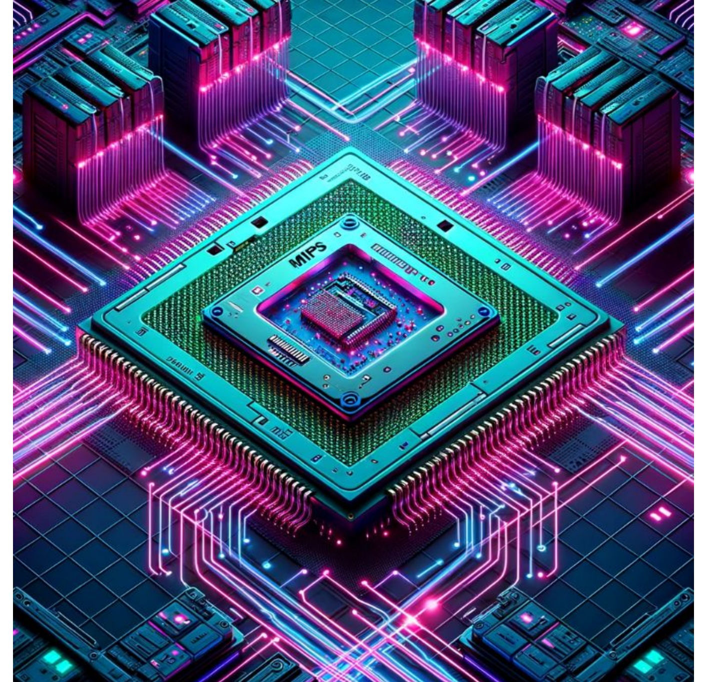
COMP1521 26T2

Week 1 Lecture 2

MIPS: Basics and Control

Today's Lecture

- Announcements
- Aside: Hexadecimal and binary
- Recap intro MIPS
- More about MIPS Instructions
- System Calls
- Style
- Simplified C and goto
- MIPS Control
    - if statements, boolean expressions
    - loops



Week 2 Revision Session

- Time: Week 2 Wednesday, **6-8pm**
- Location: Online Moodle (BlackBoard Collaborate)

Content:

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

DISCLAIMER:

Code written live in lectures may not always have the best style!

Assembly Style Guide

COMP1521 C Style Guide

Also refer to tut/lab and assignment resources and the additional commented and polished code e.g. mips_basics code

Aside: Hexadecimal

0x in C and mipsy means hexadecimal.

Hexadecimal uses 16 digits.

It uses 0-9 then A-F

Decimal	Hexadecimal	Decimal	Hexadecimal
0	0	10	A
1	1	11	B
2	2	12	C
3	3	13	D
4	4	14	E
5	5	15	F
6	6	16	10
7	7	17	11
8	8	18	12
9	9	19	13

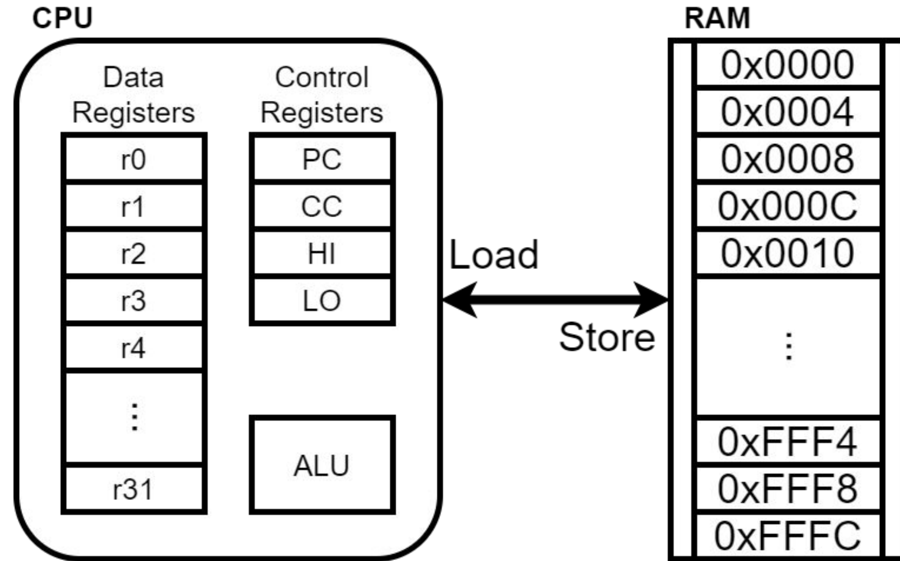
Aside: Hexadecimal to Binary

- Used to represent addresses and other binary data like instructions
- Easier for humans to read than binary
  - 8 hex digits can represent 32 bits
- Maps more nicely to binary than decimal
- Note: We will learn more about hexadecimal and binary later in the course.

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Recap: What's in a MIPS CPU?

- a set of data registers
- a set of control registers
- a control unit
- an arithmetic-logic unit
- a floating-point unit
- caches
- connection to Memory/RAM



Recap: MIPS registers to use for now

- For now we will mainly use **$t0** to **$t9** registers for general purpose calculations
- Will also need `$v0`, `$a0` for certain things too.

- **$zero** (**$0**) is special!
  - Always has the value 0 -> attempts to change it have no effect
- **$ra** is also special!
  - We use it at the end of every program

Recap: Simple Program Template

Here is a bare bones template to put instructions in to run them:

```
main:
        # YOUR CODE GOES IN HERE



        li      $v0             # return 0
        jr      $ra
```

Recap exercise

- Open Mipsy Web
- Store the value **1** in register $t0
- Store the value **6** in register $t1
- Sum these and store in $t2
- Divide the result by 2 and store in $t2

Recap: MIPS and mipsy documentation

Literally your best friend (it'll even be there for you in the exam 🥺)

COMP1521 - 26T1 Outline Forum Submissions

MIPS Instruction Set

An overview of the instruction set of the MIPS32 architecture as implemented by the mipsy and SPIM emulators. Adapted from reference documents from the University of Stuttgart and Drexel University, from material in the appendix of Patterson and Hennessey's *Computer Organization and Design*, and from the MIPS32 (r5.04) Instruction Set reference.

- Registers
- Memory
- Syntax
- Instructions
  - CPU Arithmetic Instructions
  - CPU Logical Instructions

Pseudo Instructions

- Generally a 1:1 mapping from **assembly** instructions to **binary** (CPU) instructions
- Assemblers also provide **pseudo-instructions** for convenience
- Pseudo-instructions turn into 1-3 real CPU instructions
  - Example:
    - **li $t0, 5** expands to
    - **addi $t0, $zero, 5**
  - You will see many more as you write more code in MIPS.

But how can we do input and output?

System calls

- The CPU instructions available to a normal program do not allow access to hardware devices (e.g. printing or reading input).
- Operating system performs these tasks because it has access to privileged CPU instructions
- A program must request the operating system to do the operation
  - this request is called a system call.
- mipsy simulates a very small operating system
- We will explore real system calls later in 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`

We will use all of these, especially 1, 4, 5 and 11

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`	

double / float and file syscalls are not implemented in mipsy

Others might be useful occasionally in some challenge questions.

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 and then 99 on the next line

Let’s print a string!

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 (nul-terminated) 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"
```

Exercise: 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;
}
```

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 maps to one MIPS instruction
    - **Compile your simplified C and make sure it still works**
    - Translate each line of simplified C to MIPS

Summary: Putting data in registers

- **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`

Summary: Assembly Language Syntax

- 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`
  - `QUIT = ‘q’`

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 till 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)

We deserve a quick break now!

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 instruction at a given label” e.g.
  - `ble $t0, $t1, label1  # if ($t0 <= $t1)`
  - `bgt $t0, 5, label1    # if ($t0 > 5)`
- Similar ones that compare register to 0:
  - `blez $t0, label1          # if ($t0 <= 0)`
- An unconditional branch instruction:
  - `b label1`

But how do we simplify our C if statements and while loops

COMP1511/1911 staff hid this from you!

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?

```
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)

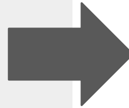
Don’t (ab)use goto

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, resulting in slower programs
- In general, do not use **goto** without good reason!
    - Typically only kernel/embedded programmers use goto
- 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;
}
```

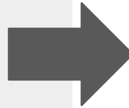


```
int main(void){
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);

    int tmp = n % 2;
    if (tmp != 0) goto if_even_end;
    printf("even\n");
if_even_end:
    return 0;
}
```

We use to **opposite** condition to check if we want to skip over the code inside the if statement

Simplifying if-else statements

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

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



```c
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;
}
```

Let’s code it up in MIPS
Extra exercise: Let’s include an else statement for odd values.

Style

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

Boolean Operators

C uses short-circuit evaluation for **&&** and ||

For example:

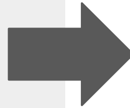
- `if ( x >= 0 && y < 100)`
    - If we know x is -3, then the condition will be false.
    - C does not evaluate the second condition
- `if ( x >= 0 || y < 100)`
    - If we know x is 0, then the condition will be true
    - C does not evaluate the second condition

Boolean Operators &&

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

Boolean Operators &&

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



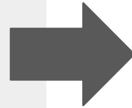
```
if (x < 0) goto else_out_of_bounds;
if (x > 100) goto else_out_of_bounds;

// in bounds
goto if_bounds_end;

else_out_of_bounds:
// out of bounds

if_bounds_end:
```

Boolean Operators ||

```
if (milk_age >= 30 ||
    milk_level < 10) {
    // get new milk
} else {
    // drink milk
}
```

Boolean Operators ||

```
if (milk_age >= 30 ||
    milk_level < 10) {
    // get new milk
} else {
    // drink milk
}
```



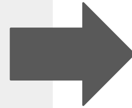
```
if (milk_age >= 30) goto
if_need_new_milk;
if (milk_level < 10) goto
if_need_new_milk;

// drink milk
goto if_milk_end;

if_need_new_milk:
// get new milk

if_milk_end:
```

Your turn

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

Your turn

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



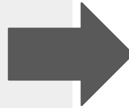
```c
if (y < 10) goto if_cond;
if (z > 50) goto if_cond;

// condition not met
goto if_cond_end;

if_cond:
// condition met

if_cond_end:
```

Your turn... a bit trickier

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

Your turn... a bit trickier

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



```
if (y < 10) goto if_cond;
if (z <= 50) goto else_not_cond;
if (w >= 5) goto else_not_cond;
goto if_cond //unnecessary

if_cond:
// condition met
goto if_cond_end
else_not_cond:
// condition not met
if_cond_end:
```

Simplifying loop structures

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

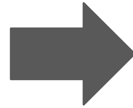
General structure:

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

You can use labels to show structure for style even if they are not used

Simplifying for loops

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

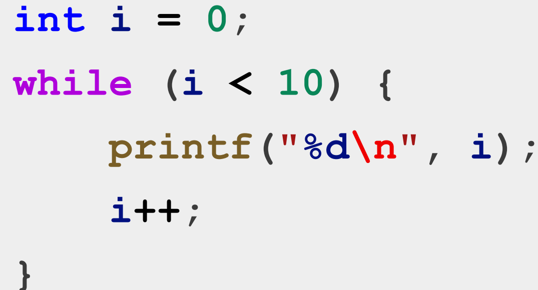


```c
int i = 0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

These are almost the same in C.

What is the difference?

Simplifying while loops




```
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

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

What did we learn today?

- MIPS
  - recap of basics from lecture 1
  - more details about MIPS instructions
  - system calls
    - printing out and reading in integers, chars and printing out strings
  - simplified C and goto statements
  - control
    - if statements,
    - boolean expressions
    - loops

Additional Resources

- MIPS Basics Notes   MIPS Basics Code
- MIPS Control Notes   MIPS Control Code

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.

https://forms.office.com/r/EYPYy0KG5E

Reach Out

Content Related Questions:

[Forum]

Admin related Questions email:

cs1521@cse.unsw.edu.au

Student Support | I Need Help With...

I need help with		Service	Contact
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**