COMP1521 24T2 Lec01

MIPS:

An Introduction

2024
Hammond Pearce
Adapted from Abiram’s slides
MIPS

MIPS?  ... but why?
What is a computer?

- A machine that “computes”
- A machine that executes a program
- How do we make a machine that executes a program?
What is a program? How do they execute?

In COMP1[59]11:

- We run a compiler (dcc?)
- ./hello
- profit ??

What’s going on here? What’s even in hello?
So what is a “program”??

- A program is a set of instructions and data

For example:
So how do we execute the program?

- The program is a set of instructions and data... somewhere
  - Maybe a “hard disk”
  - Long-term, non-volatile

- We load the program into “memory” - RAM!
  - RAM is like a massive 1D array which we divide into sections
  - It has addresses, which are like indexes into that array
  - RAM is volatile
Disks and RAM

or
And then... the CPU “runs” the program!
Programs and Program Memory

● A program contains information on how to set up memory
  ○ What instructions need to be followed?
  ○ What data do we need to load into the memory?
    ▪ Variables?
      ● Globals and locals

● Then, during operation, we might request *more* memory
  ○ malloc
  ○ So greedy

● Where do we put all these things?
A program’s memory map
How does the CPU know what to do?

- There are a finite number of possible instructions
  - We assemble programs by combining the instructions in sequences

- E.g. if we have just “\( x = a + b \)" - how do we get “\( y = a + b + c \)"?
  - \( \text{temp} = a + b \)
  - \( y = \text{temp} + c \)

- The CPU is built to execute all the possible instructions
- i.e. the CPU implements an “Instruction Set Architecture”
Some ISAs

MIPS, ARM, x86, Itanium, x86_64, Power, AVR, PIC, RISC-V ...

Lots of possible implementations

Lots of possible uses/users

E.g. games consoles:

<table>
<thead>
<tr>
<th>Year</th>
<th>Console</th>
<th>Architecture</th>
<th>Chip</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>PS1</td>
<td>MIPS</td>
<td>R3000A</td>
<td>34</td>
</tr>
<tr>
<td>1996</td>
<td>N64</td>
<td>MIPS</td>
<td>R4200</td>
<td>93</td>
</tr>
<tr>
<td>2000</td>
<td>PS2</td>
<td>MIPS</td>
<td>Emotion Engine</td>
<td>300</td>
</tr>
<tr>
<td>2001</td>
<td>xbox</td>
<td>x86</td>
<td>Celeron</td>
<td>733</td>
</tr>
<tr>
<td>2001</td>
<td>GameCube</td>
<td>Power</td>
<td>PPC750</td>
<td>486</td>
</tr>
<tr>
<td>2006</td>
<td>xbox360</td>
<td>Power</td>
<td>Xenon (3 cores)</td>
<td>3200</td>
</tr>
<tr>
<td>2006</td>
<td>PS3</td>
<td>Power</td>
<td>Cell BE (9 cores)</td>
<td>3200</td>
</tr>
<tr>
<td>2006</td>
<td>Wii</td>
<td>Power</td>
<td>PPC Broadway</td>
<td>730</td>
</tr>
<tr>
<td>2013</td>
<td>PS4</td>
<td>x86</td>
<td>AMD Jaguar (8 cores)</td>
<td>1800</td>
</tr>
<tr>
<td>2013</td>
<td>xbone</td>
<td>x86</td>
<td>AMD Jaguar (8 cores)</td>
<td>2000</td>
</tr>
<tr>
<td>2017</td>
<td>Switch</td>
<td>ARM</td>
<td>NVidia TX1</td>
<td>1000</td>
</tr>
<tr>
<td>2020</td>
<td>PS5</td>
<td>x86</td>
<td>AMD Zen 2 (8 cores)</td>
<td>3500</td>
</tr>
<tr>
<td>2020</td>
<td>xboxs</td>
<td>x86</td>
<td>AMD Zen 2 (8 cores)</td>
<td>3700</td>
</tr>
<tr>
<td>2022</td>
<td>steam deck</td>
<td>x86</td>
<td>AMD Zen 2 (4 cores)</td>
<td>3500</td>
</tr>
</tbody>
</table>
What can instructions do?

- **Load/store**: Got data? Need to load it! Need to store it!
- **Computations**: eg. add, subtract, multiply, divide, bitwise
- **Branch**: jump to execute different instructions
  - Can’t have logic (eg. if statements) if our program continues linearly
- **System calls**: phone-a-friend
- **Coprocessor**: Do hard, special things needing special hardware
  - E.g. floating point math
Do we write the instructions directly?

● Not often…
  ○ But sometimes we do!
  ○ (Someone has to!)

● Instead, we tend to write in a higher-level *compiled* language
  ○ C, C++, D, Go, Zig, Rust, Java, Swift, and many more…

● A *compiler* will input programs in these languages and *output* the corresponding assembly instructions
Assembly

Instructions are really just 0s and 1s

- Would be a *pain* to read/write literal instructions
- Instead, we use *assembly* language to form a human-readable representation of each instruction
  - Each instruction we write in assembly language *typically* represents a single CPU instruction
  - An assembler translates this to binary CPU instructions
So, to recap: how do we make a program?

- We have a program in some language (e.g. C)
- We have a processor that runs some ISA (e.g. MIPS)
- We **compile** the program into **assembly** (and a binary)
- The binary is stored to a file

Then...

- The program is loaded into memory
- The CPU is pointed at the memory
- And we are off!
More about CPUs
What’s in there?

- 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
A day in the life of a CPU - as C code

```c
int program_counter = START_ADDRESS;

while (1) {
    // Fetch an instruction from memory
    int instruction = memory[program_counter];
    // Move to the next instruction
    program_counter++;
    // Execute the next instruction
    execute(instruction, &program_counter);
        // ^ note: some instructions may
          // modify the program counter
}
```

It's more fun than it sounds, I swear.
So... writing instructions ourselves?

In this course we write assembly *ourselves* instead of compiling. But why would anyone do that?

- **To optimise** code for performance
  - Less instructions = faster to execute = saving picoseconds!
- **To write for edge cases** not supported by compilers
  - eg. writing code to interact directly with a device (i.e. *drivers*)
- **To learn** how a compiled program executes
  - Primary reason in this course
  - Can be helpful when debugging
  - Also handy to identify security vulnerabilities and exploit binaries
- **And sometimes, someone has to!**
  - E.g. who’s going to make your compiler in the first place?
So why “MIPS”?

● Once used from game consoles to supercomputers
  ○ Still used in routers and TVs

● Considerable learning resources available

● Inspired many other ISAs
  ○ If you know MIPS, you can easily branch to ARM, RISC-V, and others

● All ISAs have tradeoffs
  ○ Some focus on performance and special features

● MIPS is “simple” yet powerful - good foundation for knowledge
# More about MIPS

<table>
<thead>
<tr>
<th>Year</th>
<th>MIPS R2000</th>
<th>MIPS R3000</th>
<th>MIPS R4000</th>
<th>MIPS R5000</th>
<th>MIPS R10000</th>
<th>MIPS R12000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985</td>
<td>MIPS I (32-bit)</td>
<td>MIPS I (32-bit)</td>
<td>MIPS III (64-bit)</td>
<td>MIPS IV (64-bit)</td>
<td>MIPS IV (64-bit)</td>
<td>MIPS IV (64-bit)</td>
</tr>
<tr>
<td>1988</td>
<td>110k</td>
<td>110k</td>
<td>2.3 – 4.8m</td>
<td>3.7m</td>
<td>6.8m</td>
<td>7.15m</td>
</tr>
<tr>
<td>1992</td>
<td>2 µm</td>
<td>1.2 µm</td>
<td>0.35 µm</td>
<td>0.32 µm</td>
<td>0.35 µm</td>
<td>0.25 µm</td>
</tr>
<tr>
<td>1996</td>
<td>80 mm²</td>
<td>40 mm²</td>
<td>84 – 100 mm²</td>
<td>84 mm²</td>
<td>350 mm²</td>
<td>229 mm²</td>
</tr>
<tr>
<td>1998</td>
<td>12 – 33 MHz</td>
<td>20 – 40 MHz</td>
<td>50 – 260 MHz</td>
<td>150 – 266 MHz</td>
<td>180 – 360 MHz</td>
<td>270 – 400 MHz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flagship devices</th>
<th>DECstation 2100 and 3100 workstations</th>
<th>SGI IRIS and Indigo workstations</th>
<th>NASA New Horizons space probe</th>
<th>Carrera Computers and DeskStation Technology PCs (Windows NT)</th>
<th>SGI O2 and Indy workstations</th>
<th>Cobalt Qube servers</th>
<th>SGI Octane 2, Onyx 2, and Origin workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PlayStation game console</td>
<td>Nintendo N64 game console</td>
<td>SGI Onyx, Indigo, Indigo2, and Indy workstations</td>
<td>SGI Octane 2, Onyx 2, and Origin workstations</td>
<td>SGI Octane 2, Onyx 2, and Origin workstations</td>
<td>SGI Octane 2, Onyx 2, and Origin workstations</td>
<td>SGI Octane 2, Onyx 2, and Origin workstations</td>
<td>SGI Octane 2, Onyx 2, and Origin workstations</td>
</tr>
</tbody>
</table>

Hammond Pearce

22
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:

```
0010000110000100110000000000001100

addi $t1, $t0, 12
```
“But I don’t have a MIPS CPU!”

- True (probably).
- We can’t run our MIPS instructions directly on x86_64/ARM.
- But, we can emulate them using *mipsy*
- recreation of the behaviour of a real MIPS CPU
  - written by Zac* (past course admin, now graduated/lecturing COMP6991)
  - can download on your own machine: [https://github.com/insou22/mipsy/](https://github.com/insou22/mipsy/)
  - comes with a **command-line interface** to run in your terminal
- *mipsy_web* runs entirely in your browser
  - by Shrey*, on course website: [https://cgi.cse.unsw.edu.au/~cs1521/mipsy](https://cgi.cse.unsw.edu.au/~cs1521/mipsy)
- **vscode extension**
  - written by Xavier 🎉 - can download the ‘mipsy editor features’ extension

*some contributions from Josh Harcombe, Dylan Brotherston and Abiram*
Can we write some MIPS?
Soon™
All about registers

- Most CPU architectures perform operations over registers.
- They are part of the processor itself, not the memory.
- Speed advantages:
  - Memory is fast, but not as fast as the CPU.
  - Caches store some memory for faster access.
    - Usually not as fast as registers!
- Simplifies processor design considerably.
All about registers

- MIPS specifies 32 general-purpose registers
  - 32-bits each, same size as a typical C integer - coincidence?
  - Floating point registers (not used in COMP1521)
  - Hi/Lo special registers for multiply and divide (not important in this course)
  - Program counter
    - Keeps track of which instruction to fetch and execute next
    - Modified by branch and jump instructions
Almost all of our computations happen between registers!

Want to multiply 2 and 3 and store the result
Load 2 and 3 into registers:

\[
\text{li } $t0, 2 \\
\text{li } $t1, 3
\]

And store the result:

\[
\text{mul } $t2, $t0, $t1
\]
More about registers

Registers are denoted by a $ and can be referred to using a number ($0…$31) or by symbolic names ($zero…$ra)

$zero ($0) is special!
- Always has the value 0 -> attempts to change it have no effect

$ra ($31) is also special!
- Directly affected by two instructions we use in Week 3
More about registers

Can use the other 30 registers however we want, technically, but:
There are conventions to prevent utter chaos and madness

( Discussed more in next week’s tutorials and Week 3 lectures )

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

Convention says $t0$ to $t9$ can be used however you want

Will also need $v0$, $a0$, $ra$ for certain things at the moment

Should not need to use any other registers (yet)

We will cover the other registers when we talk about functions in Week 3
Now let’s make something
Your turn
Our programs are useless
System calls

● We mentioned these earlier
● System call ==
  ○ Hi system friend
  ○ Can you do this thing for me
  ○ Thanks
● What sort of things?

What can instructions do?

● Load/store: Got data? Need to load it! Need to store it!
● Computations: eg. add, subtract, multiply, divide, bitwise
● Branch: jump to execute different instructions
  ○ Can't have logic (eg. if statements) if our program continues linearly
● System calls: phone-a-friend
● Coprocessor: Do hard, special things needing special hardware
  ○ E.g. floating point math
System calls

- None of the instructions we have access to can interact with the outside world (e.g. 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 (e.g. communicating to other devices)
- *mipsy* simulates a very basic operating system
- Will explore real system calls in the second half of the course
Common mipsy syscalls

<table>
<thead>
<tr>
<th>Service</th>
<th>$v0</th>
<th>Arguments</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf(&quot;%d&quot;)</td>
<td>1</td>
<td>int in $a0</td>
<td></td>
</tr>
<tr>
<td>fputs</td>
<td>4</td>
<td>string in $a0</td>
<td></td>
</tr>
<tr>
<td>scanf(&quot;%d&quot;)</td>
<td>5</td>
<td>none</td>
<td>int in $v0</td>
</tr>
<tr>
<td>fgets</td>
<td>8</td>
<td>line in $a0, length in $a1</td>
<td></td>
</tr>
<tr>
<td>exit(0)</td>
<td>10</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;%c&quot;)</td>
<td>11</td>
<td>char in $a0</td>
<td></td>
</tr>
<tr>
<td>scanf(&quot;%c&quot;)</td>
<td>12</td>
<td>none</td>
<td>char in $v0</td>
</tr>
</tbody>
</table>

We don’t use syscalls 8 and 12 much in COMP1521
Most input will be integers
More ✨advanced✨ syscalls

<table>
<thead>
<tr>
<th>Service</th>
<th>$v0</th>
<th>Arguments</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf(&quot;%f&quot;)</td>
<td>2</td>
<td>float in $f12</td>
<td>float in $f0</td>
</tr>
<tr>
<td>printf(&quot;%lf&quot;)</td>
<td>3</td>
<td>double in $f12</td>
<td>double in $f0</td>
</tr>
<tr>
<td>scanf(&quot;%f&quot;)</td>
<td>6</td>
<td>none</td>
<td>address in $v0</td>
</tr>
<tr>
<td>scanf(&quot;%lf&quot;)</td>
<td>7</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>sbrk(nbytes)</td>
<td>9</td>
<td>nbytes in $a0</td>
<td></td>
</tr>
<tr>
<td>open(filename, flags, mode)</td>
<td>13</td>
<td>filename in $a0, flags in $a1, mode $a2</td>
<td>fd in $v0</td>
</tr>
<tr>
<td>read(fd, buffer, length)</td>
<td>14</td>
<td>fd in $a0, buffer in $a1, length in $a2</td>
<td>number of bytes read in $v0</td>
</tr>
<tr>
<td>write(fd, buffer, length)</td>
<td>15</td>
<td>fd in $a0, buffer in $a1, length in $a2</td>
<td>number of written in $v0</td>
</tr>
<tr>
<td>close(fd)</td>
<td>16</td>
<td>fd in $a0</td>
<td></td>
</tr>
<tr>
<td>exit(status)</td>
<td>17</td>
<td>status in $a0</td>
<td></td>
</tr>
</tbody>
</table>

Probably only used for challenge exercises in COMP1521
The system call workflow

● We specify which system call we want in $v0
  ○ eg. print_int is syscall 1:
    \[\text{li } \$v0, 1\]

● We specify arguments (if any)
  \[\text{li } \$a0, 42\]

● We transfer execution to the operating system
  ○ The OS will fulfil our request if it looks sane
    \[\text{syscall}\]

● Some syscalls may return a value - check syscall table
MIPS and mipsy documentation

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

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
  - CPU Shift Instructions
  - CPU Load, Store, and Memory Control Instructions
  - CPU Move Instructions
  - CPU Branch and Jump Instructions
Now we can say hello world