Why Study Assembler?

Useful to know assembly language because ...

- sometimes you are *required* to use it:
  - e.g., low-level system operations, device drivers
- improves your understanding of how compiled programs execute
  - very helpful when debugging
  - understand performance issues better
- performance tweaking ... squeezing out last pico-second
  - re-write that performance-critical code in assembler!
- create games in pure assembler
  - e.g., *RollerCoaster Tycoon*
CPU Components

A typical modern CPU has:
- a set of *data* registers
- a set of *control* registers (including PC)
- a *control unit* (CU)
- an *arithmetic-logic unit* (ALU)
- a *floating-point unit* (FPU)
- caches
  - caches normally range from L1 to L3
    - L1 is the fastest and smallest
  - sometimes separate data and instruction caches
    - eg. L1d and L1i caches
- access to *memory* (RAM)
  - Address generation unit (AGU)
  - Memory management unit (MMU)
- a set of simple (or not so simple) instructions
  - transfer data between memory and registers
  - compute values using ALU/FPU
  - make tests and transfer control of execution

Different types of processors have different configurations of the above
What A CPU Looks Like
## CPU Architecture Families Used in Game 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>xboxes</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>
### MIPS Family

<table>
<thead>
<tr>
<th>Year</th>
<th>MIPS ISA</th>
<th>Transistor count</th>
<th>Process node</th>
<th>Die size</th>
<th>Speed</th>
<th>Flagship devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985</td>
<td>MIPS I (32-bit)</td>
<td>110k</td>
<td>2 μm</td>
<td>80 mm²</td>
<td>12 – 33 MHz</td>
<td>DECstation 2100 and 3100 workstations</td>
</tr>
<tr>
<td>1988</td>
<td>MIPS I (32-bit)</td>
<td>110k</td>
<td>1.2 μm</td>
<td>40 mm²</td>
<td>20 – 40 MHz</td>
<td>SGI IRIS and Indigo workstations</td>
</tr>
<tr>
<td>1992</td>
<td>MIPS III (64-bit)</td>
<td>2.3 – 4.6m</td>
<td>0.35 μm</td>
<td>84 – 100 mm²</td>
<td>50 – 250 MHz</td>
<td>NASA New Horizons space probe</td>
</tr>
<tr>
<td>1996</td>
<td>MIPS IV (64-bit)</td>
<td>3.7m</td>
<td>0.32 μm</td>
<td>84 mm²</td>
<td>150 – 266 MHz</td>
<td>Sony PlayStation game console</td>
</tr>
<tr>
<td>1995</td>
<td>MIPS IV (64-bit)</td>
<td>6.8m</td>
<td>0.35 μm</td>
<td>350 mm²</td>
<td>180 – 360 MHz</td>
<td>Nintendo N64 game console</td>
</tr>
<tr>
<td>1998</td>
<td>MIPS IV (64-bit)</td>
<td>7.15m</td>
<td>0.25 μm</td>
<td>229 mm²</td>
<td>270 – 400 MHz</td>
<td>SGI O2 and Indy workstations</td>
</tr>
</tbody>
</table>

**Figure 3: MIPS Family**
Fetch-Execute Cycle

- typical CPU program execution pseudo-code:

```c
uint32_t program_counter = START_ADDRESS;
while (1) {
    uint32_t instruction = memory[program_counter];

    // move to next instruction
    program_counter++;

    // branches and jumps instruction may change program_counter
    execute(instruction, &program_counter);
}
```

https://www.cse.unsw.edu.au/~cs1521/23T1/COMP1521-23T1—MIPS-Basics
Executing an instruction involves:
- determine what the *operator* is
- determine if/which *register(s)* are involved
- determine if/which *memory location* is involved
- carry out the operation with the relevant operands
- store result, if any, in the appropriate register / memory location

Example instruction encodings
(not from a real machine):

```
<table>
<thead>
<tr>
<th>ADD</th>
<th>$t1</th>
<th>$t2</th>
<th>$t0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>8 bits</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>LOAD</th>
<th>$s7</th>
<th>0x1004</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>8 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
```

Figure 4: Fake Instructions
MIPS Architecture

MIPS is a well-known and simple architecture

- historically used everywhere from supercomputers to game consoles
- still popular in some embedded fields: e.g., modems/routers, TVs
- but being out-competed by ARM and, more recently, RISC-V

COMP1521 uses the MIPS32 version of the MIPS family.

COMP1521 uses simulators, not real MIPS hardware:

- mipsy … command-line-based emulator written by Zac
  - source code: https://github.com/insou22/mipsy
- mipsy-web … web (WASM) GUI-based version of mipsy written by Shrey
  - https://cgi.cse.unsw.edu.au/~cs1521/mipsy/
MIPS has several classes of instructions:

- **load and store** ... transfer data between registers and memory
- **computational** ... perform arithmetic/logical operations
- **jump and branch** ... transfer control of program execution
- **coprocessor** ... standard interface to various co-processors
  - coprocessors implement floating-point operations
  - won’t be covered in COMP1521
- **special** ... miscellaneous tasks (e.g. syscall)
MIPS Instructions

Instructions are simply bit patterns. MIPS instructions are 32-bits long, and specify …

- an **operation** (e.g. load, store, add, branch, ...)
- zero or more **operands** (e.g. registers, memory addresses, constants, ...)

Some possible instruction formats

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>R1</th>
<th>R2</th>
<th>Memory Address Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td><strong>I-type</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>R1</th>
<th>Memory Address Constant Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>J-type</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://www.cse.unsw.edu.au/~cs1521/23T1/
Assembly Language

Instructions are simply bit patterns — on MIPS, 32 bits long.

- Could write **machine code** programs just by specifying bit-patterns
e.g as a sequence of hex digits:

  0x2002000b 0x20040048 0x0000000c 0x20040069 0x0000000c 0x2004000a 0x0000000c

  - unreadable!
  - difficult to maintain!

- adding/removing instructions changes bit pattern for other instructions
  - *branch* and *jump* instructions use relative offsets

- changing variable layout in memory changes bit pattern for instructions
  - *load* and *store* instructions require encoded addresses

Solution: **assembly language**, a symbolic way of specifying machine code

- write instructions using names rather than bit-strings
- refer to registers using either numbers or names
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw</code></td>
<td><code>$t1, address</code></td>
<td># <code>reg[t1] = memory[address]</code></td>
</tr>
<tr>
<td><code>sw</code></td>
<td><code>$t3, address</code></td>
<td># <code>memory[address] = reg[t3]</code></td>
</tr>
<tr>
<td><code>la</code></td>
<td><code>$t1, address</code></td>
<td># <code>reg[t1] = address</code></td>
</tr>
<tr>
<td><code>lui</code></td>
<td><code>$t2, const</code></td>
<td># <code>reg[t2] = const &lt;&lt; 16</code></td>
</tr>
<tr>
<td><code>and</code></td>
<td><code>$t0, $t1, $t2</code></td>
<td># <code>reg[t0] = reg[t1] &amp; reg[t2]</code></td>
</tr>
<tr>
<td><code>add</code></td>
<td><code>$t0, $t1, $t2</code></td>
<td># <code>reg[t0] = reg[t1] + reg[t2]</code></td>
</tr>
<tr>
<td><code>addi</code></td>
<td><code>$t2, $t3, 5</code></td>
<td># <code>reg[t2] = reg[t3] + 5</code></td>
</tr>
<tr>
<td><code>mult</code></td>
<td><code>$t3, $t4</code></td>
<td># <code>(Hi,Lo) = reg[t3] * reg[t4]</code></td>
</tr>
<tr>
<td><code>slt</code></td>
<td><code>$t7, $t1, $t2</code></td>
<td># <code>reg[t7] = (reg[t1] &lt; reg[t2])</code></td>
</tr>
<tr>
<td><code>j</code></td>
<td><code>label</code></td>
<td># <code>PC = label</code></td>
</tr>
<tr>
<td><code>beq</code></td>
<td><code>$t1, $t2, label</code></td>
<td># <code>PC = label if reg[t1]==reg[t2]</code></td>
</tr>
<tr>
<td><code>nop</code></td>
<td></td>
<td># do nothing</td>
</tr>
</tbody>
</table>
MIPS Architecture: Registers

MIPS CPU has

- 32 general purpose registers (32-bit)
- 32/16 floating-point registers (for float/double)
  - pairs of floating-point registers used for double-precision (not used in COMP1521)
- PC ... 32-bit register (always aligned on 4-byte boundary)
  - modified by branch and jump instructions
- Hi, Lo ... store results of mult and div
  - accessed by mthi and mflo instructions only

Registers can be referred to as numbers ($0...$31), or by symbolic names ($zero...$ra)

Some registers have special uses:

- register $0 ($zero) always has value 0, can not be changed
- register $31 ($ra) is changed by jal and jalr instructions
- registers $1 ($at) reserved for mipsy to use in pseudo-instructions
- registers $26 ($k0), $27 ($k1) reserved for operating-system to use in interrupts (exception handling and
# MIPS Architecture: Integer Registers

<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>
MIPS Architecture: Integer Registers ... Usage Convention

- Except for registers zero and ra (0 and 31), these uses are only programmer’s conventions
  - No difference between registers 1..30 in the silicon
  - mipsy follows these conventions so at, k0, k1 can change unexpectedly

- Conventions allow compiled code from different sources to be combined (linked).
  - Conventions are formalized in an Application Binary Interface (ABI)

- Some of these conventions are irrelevant when writing tiny assembly programs
  - Follow them anyway
  - It’s good practice

- For general use, keep to registers t0..t9, s0..s7

- Use other registers only for conventional purposes
  - E.g. only, and always, use a0..a3 for arguments

- Never use registers at, k0, k1
Data and Addresses

All operations refer to data, either

- in a register
- in memory
- a constant that is embedded in the instruction itself

Computation operations refer to registers or constants.

Only load/store instructions refer to memory.

The syntax for constant value is C-like:

```
1 3 -1 -2 12345 0x1 0xFFFFFFFF 0b10101010 0o123
"a string" 'a' 'b' '1' '\n' '\0'
```
Describing MIPS Assembly Operations

Registers are denoted:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_d$</td>
<td>destination register</td>
</tr>
<tr>
<td>$R_s$</td>
<td>source register #1</td>
</tr>
<tr>
<td>$R_t$</td>
<td>source register #2</td>
</tr>
</tbody>
</table>

For example:

```
add $R_d, $R_s, $R_t \implies R_d := R_s + R_t
```
## Integer Arithmetic Instructions

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $r_d, r_s, r_t$</td>
<td>$r_d = r_s + r_t$</td>
<td>0000000ssssstttttdddd00000100000</td>
</tr>
<tr>
<td>sub $r_d, r_s, r_t$</td>
<td>$r_d = r_s - r_t$</td>
<td>0000000ssssstttttdddd00000100010</td>
</tr>
<tr>
<td>mul $r_d, r_s, r_t$</td>
<td>$r_d = r_s \times r_t$</td>
<td>0111000ssssstttttdddd0000000010</td>
</tr>
<tr>
<td>rem $r_d, r_s, r_t$</td>
<td>$r_d = r_s % r_t$</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>div $r_d, r_s, r_t$</td>
<td>$r_d = r_s / r_t$</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>addi $r_t, r_s, I$</td>
<td>$r_t = r_s + I$</td>
<td>001000ssssstttttIIIIIIIIIIIIIIII</td>
</tr>
</tbody>
</table>

- integer arithmetic is 2’s-complement (covered later in COMP1521)
- also: **addu, subu, mulu, addiu** - equivalent instructions which do not stop execution on overflow.
- no **subi** instruction - use **addi** with negative constant
- **mipsy** will translate **add** and of **sub** a constant to **addi**
  - e.g. **mipsy** translates **add $t7, t4, 42** to **addi $t7, t4, 42**
  - for readability use **addi**, e.g. **addi $t7, t4, 42**
- **mipsy** allows $r_s$ to be omitted and will use $r_d$
  - e.g. **mipsy** translates **add $t7, t1** to **add $t7, t7, t1**
  - for readability use the full instruction, e.g. **add $t7, t7, t1**
Integer Arithmetic Instructions - Example

```assembly
addi $t0, $zero, 6   # $t0 = 6
addi $t5, $t0, 2    # $t5 = 8
mul $t4, $t0, $t5   # $t4 = 48
add $t4, $t4, $t5   # $t4 = 56
addi $t6, $t4, -14  # $t6 = 42
```
### Extra Integer Arithmetic Instructions (little used in COMP1521)

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>div</strong> $r_s, r_t$</td>
<td>$hi = r_s % r_t$; $lo = r_s / r_t$</td>
<td>0000000sssssttttt0000000000011010</td>
</tr>
<tr>
<td><strong>mult</strong> $r_s, r_t$</td>
<td>$hi = (r_s \times r_t) \gg 32$</td>
<td>0000000sssssttttt00000000000011000</td>
</tr>
<tr>
<td><strong>mflo</strong> $r_d$</td>
<td>$r_d = lo$</td>
<td>00000000000000000000000001010</td>
</tr>
<tr>
<td><strong>mfhi</strong> $r_d$</td>
<td>$r_d = hi$</td>
<td>00000000000000000000000001001</td>
</tr>
</tbody>
</table>

- **mult** multiplies and provides a 64-bit result
- **mul** instruction provides only 32-bit result (can overflow)
- **mipsy** translates **rem** $r_d, r_s, r_t$ to **div** $r_s, r_t$ plus **mfhi** $r_d$
- **mipsy** translates **div** $r_d, r_s, r_t$ to **div** $r_s, r_t$ plus **mflo** $r_d$
- **divu** and **multu** are unsigned equivalents of **div** and **mult**
Bit Manipulation Instructions (for future reference)

- Instructions explained later when we cover bitwise operators

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and</code> $r_d, r_s, r_t$</td>
<td>$r_d = r_s &amp; r_t$</td>
<td>000000sssssttttttddd00000100100</td>
</tr>
<tr>
<td><code>or</code> $r_d, r_s, r_t$</td>
<td>$r_d = r_s \mid r_t$</td>
<td>000000sssssttttttddd00000100101</td>
</tr>
<tr>
<td><code>xor</code> $r_d, r_s, r_t$</td>
<td>$r_d = r_s ^ r_t$</td>
<td>000000sssssttttttddddd00000100110</td>
</tr>
<tr>
<td><code>nor</code> $r_d, r_s, r_t$</td>
<td>$r_d = \sim (r_s \mid r_t)$</td>
<td>000000sssssttttttddddd00000100111</td>
</tr>
<tr>
<td><code>andi</code> $r_t, r_s, I$</td>
<td>$r_t = r_s &amp; I$</td>
<td>001100sssssttttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><code>ori</code> $r_t, r_s, I$</td>
<td>$r_t = r_s \mid I$</td>
<td>001101sssssttttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><code>xori</code> $r_t, r_s, I$</td>
<td>$r_t = r_s ^ I$</td>
<td>001110sssssttttttIIIIIIIIIIIIIIII</td>
</tr>
<tr>
<td><code>not</code> $r_d, r_s$</td>
<td>$r_d = \sim r_s$</td>
<td>pseudo-instruction</td>
</tr>
</tbody>
</table>

- Mipsy translates `not $r_d, r_s$` to `nor $r_d, r_s, 0$`
Shift Instructions (for future reference)

- instructions explained later when we cover bitwise operators

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sllv</code></td>
<td><code>r_d = r_t &lt;&lt; r_s</code></td>
<td>000000000000000000100</td>
</tr>
<tr>
<td><code>srlv</code></td>
<td><code>r_d = r_t &gt;&gt; r_s</code></td>
<td>000000000000000000110</td>
</tr>
<tr>
<td><code>srav</code></td>
<td><code>r_d = r_t &gt;&gt; r_s</code></td>
<td>000000000000000000111</td>
</tr>
<tr>
<td><code>sll</code></td>
<td><code>r_d = r_t &lt;&lt; I</code></td>
<td>000000000000000000000</td>
</tr>
<tr>
<td><code>srl</code></td>
<td><code>r_d = r_t &gt;&gt; I</code></td>
<td>000000000000000000001</td>
</tr>
<tr>
<td><code>sra</code></td>
<td><code>r_d = r_t &gt;&gt; I</code></td>
<td>000000000000000000001</td>
</tr>
</tbody>
</table>

- `srl` and `srlv` shift zeros into most-significant bit
  - this matches shift in C of `unsigned` value
- `sra` and `srav` propagate most-significant bit
  - this ensure shifting a negative number divides by 2
- `slav` and `sla` don’t exist as arithmetic and logical left shifts are the same
- mipsy provides `rol` and `ror` pseudo-instructions which rotate bits
  - real instructions on some MIPS versions
  - no simple C equivalent
### Miscellaneous Instructions

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>li R_d, value</code></td>
<td>$R_d = \text{value}$</td>
<td>psuedo-instruction</td>
</tr>
<tr>
<td><code>la R_d, label</code></td>
<td>$R_d = \text{label}$</td>
<td>psuedo-instruction</td>
</tr>
<tr>
<td><code>move R_d, R_s</code></td>
<td>$R_d = R_s$</td>
<td>psuedo-instruction</td>
</tr>
<tr>
<td><code>slt R_d, R_s, R_t</code></td>
<td>$R_d = R_s &lt; R_t$</td>
<td>00000000000000000000000000001010</td>
</tr>
<tr>
<td><code>slti R_t, R_s, I</code></td>
<td>$R_t = R_s &lt; I$</td>
<td>00101000000000000000000000000000000000000000100</td>
</tr>
<tr>
<td><code>lui R_t, I</code></td>
<td>$R_t = I \times 65536$</td>
<td>0011110000000000000000000000000000000000000000000001100</td>
</tr>
<tr>
<td><code>syscall</code></td>
<td>system call</td>
<td></td>
</tr>
</tbody>
</table>

1. **MIPSY allows `li` and `la` to be used interchangably**
   - for readability use `li` for constants, e.g. 0, 0xFF, ‘#’
   - for readability use `la` for labels, e.g main
2. **probably not needed in COMP1521, but also similar instruction/psuedo-instructions to `slt/slhi`**
   - `sle/sleui, sge/sgeui, sgt/sgtui, seq/seqi, sne/snei`
   - and unsigned versions `sleu/sleui, sgeu/sgeui, sgtu/sgtui, sequ/sequi, sneu/snei`
3. **mipsy may translate pseudo-instructions to `lui`**

---

For more information, visit: [https://www.cse.unsw.edu.au/~cs1521/23T1/COMP1521-23T1-MIPS-Basics](https://www.cse.unsw.edu.au/~cs1521/23T1/COMP1521-23T1-MIPS-Basics)
Example Use of Miscellaneous Instructions

```
li $t4, 42        # $t4 = 42
li $t0, 0x2a      # $t0 = 42 (hexadecimal @aA is 42 decimal)
lis $t3, 'r'      # $t3 = 42 (ASCII for * is 42)
la $t5, start     # $t5 = address corresponding to label start
move $t6, $t5     # $t6 = $t5
slt $t1, $t3, $t3 # $t1 = 0 ($t3 and $t3 contain 42)
slti $t7, $t3, 56  # $t7 = 1 ($t3 contains 42)
lui $t8, 1        # $t8 = 65536
addi $t8, $t8, 34464 # $t8 = 100000
```
Example Translation of Pseudo-instructions

**Pseudo-Instructions**

- `move $a1, $v0`
- `li $t5, 42`
- `li $s1, 0xdeadbeef`
- `la $t3, label`

**Real Instructions**

- `addi $a1, $0, $v0`
- `ori $t5, $0, 42`
- `lui $at, 0xdead`
- `ori $s1, $at, 0xbeef`
- `lui $at, label[31..16]`
- `ori $t3, $at, label[15..0]`
MIPS vs mipsy

MIPS is a machine architecture, including instruction set

mipsy is an *emulator* for the MIPS instruction set

- reads text files containing instruction + directives
- converts to machine code and loads into “memory”
- provides some debugging capabilities
  - single-step, breakpoints, view registers/memory, ...
- provides mechanism to interact with operating system (syscall)

Also provides extra instructions, mapped to MIPS core set:

- provide convenient/mnemonic ways to do common operations
  - e.g. `move $s0, $v0` rather than `addu $s0, $v0, $0`
Using Mipsy

How to execute MIPS code without a MIPS

- **1521 mipsy**
  - command line tool on CSE systems
  - load programs using command line arguments
  - interact using stdin/stdout via terminal

- **mipsy_web**
  - [https://cgi.cse.unsw.edu.au/~cs1521/mipsy/](https://cgi.cse.unsw.edu.au/~cs1521/mipsy/)
  - runs in web browser, load programs with a button
  - visual environment for debugging

- **spim, xspim, qtspim**
  - older widely used MIPS simulator
  - beware: missing some pseudo-instructions used in 1521 for function calls
Using mipsy Interactively

$ 1521 mipsy

[mipsy] load my_program.s

success: file loaded

[mipsy] step 6

_start:
0x80000000 kernel [0x3c1a0040] lui $k0, 64
0x80000004 kernel [0x375a0000] ori $k0, $k0, 0
0x80000008 kernel [0x0340f809] jalr $ra, $k0

main:
0x00400000 2 [0x20020001] addi $v0, $zero, 1 # li $v0, 1
0x00400004 3 [0x2004002a] addi $a0, $zero, 42 # li $a0, 42
0x00400008 4 [0x0000000c] syscall # syscall

[SYSCALL 1] print_int: 42

[mipsy]
Important System Calls

Our programs can’t really do anything... we usually rely on system services to do things for us. **syscall** lets us make system calls for these services.

mipsy provides a set of system calls for I/O and memory allocation. **$v0** specifies which system call —

<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>puts</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 won’t use system calls 8, 12 much in COMP1521 - most input will be integers
Other System Calls ... Little Used in COMP1521

- for completeness some other system calls provided by mipsy
- probably not needed for COMP1521, except could appear in challenge exercise or provided code

<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></td>
</tr>
<tr>
<td>printf(&quot;%lf&quot;)</td>
<td>3</td>
<td>double in $f12</td>
<td></td>
</tr>
<tr>
<td>scanf(&quot;%f&quot;)</td>
<td>6</td>
<td>none</td>
<td>float in $f0</td>
</tr>
<tr>
<td>scanf(&quot;%lf&quot;)</td>
<td>7</td>
<td>none</td>
<td>double in $f0</td>
</tr>
<tr>
<td>sbrk(nbytes)</td>
<td>9</td>
<td>nbytes in $a0</td>
<td>address in $v0</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>
### Encoding MIPS Instructions as 32 bit Numbers

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $a3, $t0, $zero</td>
<td>000000 sssss ttttt ddddd 00000 100000</td>
</tr>
<tr>
<td>add $d, $s, $t</td>
<td>000000 01000 00000 00111 00000 100000</td>
</tr>
<tr>
<td></td>
<td>0x01003820 (decimal 16791584)</td>
</tr>
<tr>
<td>sub $a1, $at, $v1</td>
<td>000000 sssss ttttt ddddd 00000 100010</td>
</tr>
<tr>
<td>sub $d, $s, $t</td>
<td>000000 00001 00111 00101 00000 100010</td>
</tr>
<tr>
<td></td>
<td>0x00232822 (decimal 2304034)</td>
</tr>
<tr>
<td>addi $v0, $v0, 1</td>
<td>001000 sssss ddddd CCCCCCCCCCCCCCCCC</td>
</tr>
<tr>
<td>addi $d, $s, $C</td>
<td>001000 00010 00010 000000000000000001</td>
</tr>
<tr>
<td></td>
<td>0x20420001 (decimal 541196289)</td>
</tr>
</tbody>
</table>

All instructions are variants of a small number of bit patterns.

...register numbers always in same place.
MIPS assembly language programs contain:

- assembly language instructions
- labels ... appended with :
- comments ... introduced by #
- directives ... symbol beginning with .
- constant definitions, equivalent of #define in C, e.g:

```
MAX_NUMBERS = 1000
```

Programmers need to specify:

- data objects that live in the data region
- instruction sequences that live in the code/text region

Each instruction or directive appears on its own line.
Our First MIPS program

C

```c
int main(void) {
    printf("%s", "I love MIPS\n");
    return 0;
}
```

MIPS

```mips
# print a string in MIPS assembly
# Written by: Andrew Taylor <andrewt@unsw.edu.au>
# Written as a COMP1521 lecture example

main:
    la $a0, string  # ... pass address
    li $v0, 4      # ... 4 is printf
    syscall
    li $v0, 0
    syscall
    jr $ra

.data

string:
.asciiz "I love MIPS\n"
```

source code for i_love_mips.s

[https://www.cse.unsw.edu.au/~cs1521/23T1/COMP1521 23T1 — MIPS Basics](https://www.cse.unsw.edu.au/~cs1521/23T1/COMP1521 23T1 — MIPS Basics)
Writing correct assembler directly is hard.

Recommended strategy:

- write, test & debug a solution in C
- map down to “simplified” C
- test “simplified” C and ensure correct
- translate simplified C statements to MIPS instructions

**Simplified C**

- does *not* have complex expressions
- *does* have one-operator expressions
Adding Two Numbers — C to Simplified C

C

```c
int main(void) {
    int x = 17;
    int y = 25;
    printf("%d\n", x + y);
    return 0;
}
```

Simplified C

```c
int main(void) {
    int x, y, z;
    x = 17;
    y = 25;
    z = x + y;
    printf("%d", z);
    printf("%c", '\n');
    return 0;
}
```
**Simplified C**

```c
int x, y, z;
x = 17;
y = 25;
z = x + y;
printf("%d", z);
printf("%c", '\n');
```

**MIPS**

```mips
# add 17 and 25 then print the result
# Written by: Andrew Taylor <andrewt@unsw.edu.au>
# Written as a COMP1521 lecture example
main:
    # x in $t0
    # y in $t1
    # z in $t2
    li $t0, 17    # x = 17;
    li $t1, 25    # y = 25;
    add $t2, $t1, $t0     # z = x + y
    move $a0, $t2         # printf("%d", z);
    li $v0, 1
    syscall
    li $a0, '\n'         # printf("%c", '\n');
    li $v0, 11
    syscall
    li $v0, 0            # return 0
    jr $ra
```