**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
    - e.g., 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

![Figure 1: A Simple CPU](https://www.cse.unsw.edu.au/~cs1521/24T2/)

https://www.cse.unsw.edu.au/~cs1521/24T2/
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 R3000A</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>1996</td>
<td>N64</td>
<td>MIPS R4200</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>PS2</td>
<td>Emotion Engine</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>xbox</td>
<td>x86</td>
<td>Celeron 733</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>GameCube</td>
<td>Power PPC750</td>
<td>486</td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>xbox360</td>
<td>Power Xenon</td>
<td>3200</td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>PS3</td>
<td>Power Cell BE</td>
<td>3200</td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td>Wii</td>
<td>Power PPC Broadway</td>
<td>730</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>PS4</td>
<td>x86 AMD Jaguar (8 cores)</td>
<td>1800</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>xbone</td>
<td>x86 AMD Jaguar (8 cores)</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Switch</td>
<td>ARM NVidia TX1</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td>PS5</td>
<td>x86 AMD Zen 2 (8 cores)</td>
<td>3500</td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td>xbone</td>
<td>x86 AMD Zen 2 (8 cores)</td>
<td>3700</td>
<td></td>
</tr>
<tr>
<td>2022</td>
<td>steam deck</td>
<td>x86 AMD Zen 2 (4 cores)</td>
<td>3500</td>
<td></td>
</tr>
</tbody>
</table>

MIPS Family

- MIPS R2000
- MIPS R3000
- MIPS R4000
- MIPS R5000
- MIPS R10000
- MIPS R12000

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistor count</td>
<td>110k</td>
<td>110k</td>
<td>2.3 - 4.8m</td>
<td>3.7m</td>
<td>6.8m</td>
<td>7.1m</td>
</tr>
<tr>
<td>Process width</td>
<td>2 μm</td>
<td>1.2 μm</td>
<td>0.35 μm</td>
<td>0.25 μm</td>
<td>0.25 μm</td>
<td></td>
</tr>
<tr>
<td>Die size</td>
<td>80 mm²</td>
<td>40 mm²</td>
<td>84 - 100 mm²</td>
<td>62 mm²</td>
<td>350 mm²</td>
<td>229 mm²</td>
</tr>
<tr>
<td>Speed</td>
<td>12 - 33 MHz</td>
<td>35 - 40 MHz</td>
<td>50 - 200 MHz</td>
<td>180 - 266 MHz</td>
<td>180 - 360 MHz</td>
<td>275 - 420 MHz</td>
</tr>
</tbody>
</table>

Flagship devices:
- Sony PlayStation game consoles
- Sega Genesis and Sega Saturn game consoles
- Sega Dreamcast game consoles
- IBM 9000 workstation
- HP 9000 workstation
- Silicon Graphics Indigo, Indigo2, and Indigo workstations
- NEC RX and octeron supercomputers
- Siemens NDR/5 servers
- Cray C-90 supercomputer
- IBM RS/6000 workstation
- DECstation 3100 and 3100 workstations
- NASA JPL space probe

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);
}
```

- 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>Operator</th>
<th>$t1</th>
<th>$t2</th>
<th>$t0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>8 bits</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>LOAD</td>
<td>8 bits</td>
<td>16 bits</td>
<td></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)

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

- **R-type**

- **I-type**

- **J-type**

![Figure 5: MIPS Instructions](https://www.cse.unsw.edu.au/~cs1521/24T2/)

Assembly Language - why?

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 0x200200

  - 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
Assembly Language - symbolic way of specifying machine code

- write instructions using names rather than bit-strings
- refer to registers using either numbers or names
- allow names (labels) associated with memory addresses

```assembly
li $v0, 11
li $a0, 'H'
syscall
li $a0, 'i'
syscall
li $a0, '
'
syscall
li $v0, 0
jr $ra
```

**Example MIPS Assembler**

```assembly
lw $t1, address # reg[t1] = memory[address]
sw $t3, address # memory[address] = reg[t3]
la $t1, address # reg[t1] = address
lui $t2, const # reg[t2] = const << 16
and $t0, $t1, $t2 # reg[t0] = reg[t1] & reg[t2]
add $t0, $t1, $t2 # reg[t0] = reg[t1] + reg[t2]
   # add signed 2’s complement ints
addi $t2, $t3, 5 # reg[t2] = reg[t3] + 5
   # add immediate, no sub immediate
mult $t3, $t4 # (Hi,Lo) = reg[t3] * reg[t4]
   # store 64-bit result across Hi,Lo
slt $t7, $t1, $t2 # reg[t7] = (reg[t1] < reg[t2])
j label # PC = label
beq $t1, $t2, label # PC = label if reg[t1]==reg[t2]
nop # do nothing
```

**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 system calls)
### MIPS Architecture: Integer Registers - the important ones for COMP1521

<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..15</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 0xFFFFFFF 0b10101010 0o123
"a string" 'a' 'b' '1' '\n' '\0'
```
Describing MIPS Assembly Operations

Registers are denoted:

- $R_d$: destination register where result goes
- $R_s$: source register #1 where data comes from
- $R_t$: source register #2 where data comes from

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><code>add</code> $r_d, r_s, r_t$</td>
<td>$r_d = r_s + r_t$</td>
<td><code>000000ssssstttttddddd00000100000</code></td>
</tr>
<tr>
<td><code>sub</code> $r_d, r_s, r_t$</td>
<td>$r_d = r_s - r_t$</td>
<td><code>000000ssssstttttddddd00000100010</code></td>
</tr>
<tr>
<td><code>mul</code> $r_d, r_s, r_t$</td>
<td>$r_d = r_s \times r_t$</td>
<td><code>011100ssssstttttddddd00000000010</code></td>
</tr>
<tr>
<td><code>rem</code> $r_d, r_s, r_t$</td>
<td>$r_d = r_s % r_t$</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td><code>div</code> $r_d, r_s, r_t$</td>
<td>$r_d = r_s / r_t$</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td><code>addi</code> $r_t, r_s, I$</td>
<td>$r_t = r_s + I$</td>
<td><code>001000ssssstttttIIIIIIIIIIIIIIIIIII</code></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

```
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
```
**Shift 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>\texttt{div} ( r_s, r_t )</td>
<td>( hi = r_s \mod r_t ); ( lo = r_s / r_t )</td>
<td>( 00000000000000001010 )</td>
</tr>
<tr>
<td>\texttt{mul} ( r_s, r_t )</td>
<td>( hi = (r_s \times r_t) \div 32 ); ( lo = (r_s \times r_t) &amp; \text{0xffffffff} )</td>
<td>( 00000000000000001100 )</td>
</tr>
<tr>
<td>\texttt{mflo} ( r_d )</td>
<td>( r_d = lo )</td>
<td>( 00000000000000001010 )</td>
</tr>
<tr>
<td>\texttt{mfhi} ( r_d )</td>
<td>( r_d = hi )</td>
<td>( 00000000000000001010 )</td>
</tr>
</tbody>
</table>

- \texttt{mul} multiplies and provides a 64-bit result.
- \texttt{mul} instruction provides only 32-bit result (can overflow).
- \texttt{mipsy} translates \texttt{rem} \( r_d, r_s, r_t \) to \texttt{div} \( r_s, r_t \) plus \texttt{mfhi} \( r_d \).
- \texttt{mipsy} translates \texttt{div} \( r_d, r_s, r_t \) to \texttt{div} \( r_s, r_t \) plus \texttt{mflo} \( r_d \).
- \texttt{div} and \texttt{mul} are unsigned equivalents of \texttt{div} and \texttt{mul}.

**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>\texttt{and} ( r_d, r_s, r_t )</td>
<td>( r_d = r_s \land r_t )</td>
<td>( 00000000000000001111 )</td>
</tr>
<tr>
<td>\texttt{or} ( r_d, r_s, r_t )</td>
<td>( r_d = r_s \lor r_t )</td>
<td>( 00000000000000001111 )</td>
</tr>
<tr>
<td>\texttt{xor} ( r_d, r_s, r_t )</td>
<td>( r_d = r_s \oplus r_t )</td>
<td>( 00000000000000001111 )</td>
</tr>
<tr>
<td>\texttt{nor} ( r_d, r_s, r_t )</td>
<td>( r_d = \neg (r_s \lor r_t) )</td>
<td>( 00000000000000001111 )</td>
</tr>
<tr>
<td>\texttt{andi} ( r_d, r_s, I )</td>
<td>( r_d = r_s \land I )</td>
<td>( 00000000000000001111 )</td>
</tr>
<tr>
<td>\texttt{ori} ( r_d, r_s, I )</td>
<td>( r_d = r_s \lor I )</td>
<td>( 00000000000000001111 )</td>
</tr>
<tr>
<td>\texttt{xori} ( r_d, r_s, I )</td>
<td>( r_d = r_s \oplus I )</td>
<td>( 00000000000000001111 )</td>
</tr>
<tr>
<td>\texttt{not} ( r_d, r_s )</td>
<td>( r_d = \neg r_s )</td>
<td>pseudo-instruction</td>
</tr>
</tbody>
</table>

- \texttt{mipsy} translates \texttt{not} \( r_d, r_s \) to \texttt{nor} \( r_d, r_s, 0 \).

**Shift 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>\texttt{slv} ( r_d, r_s, r_t )</td>
<td>( r_d = r_{s+1} \land r_{t+1} )</td>
<td>( 00000000000000001100 )</td>
</tr>
<tr>
<td>\texttt{sr} ( r_d, r_s, r_t )</td>
<td>( r_d = r_{s-1} \land r_{t-1} )</td>
<td>( 00000000000000001110 )</td>
</tr>
<tr>
<td>\texttt{sra} ( r_d, r_s, r_t )</td>
<td>( r_d = r_{s+1} \land r_{t+1} )</td>
<td>( 00000000000000001101 )</td>
</tr>
<tr>
<td>\texttt{sll} ( r_d, r_s, I )</td>
<td>( r_d = r_{s+1} \land I )</td>
<td>( 00000000000000001111 )</td>
</tr>
<tr>
<td>\texttt{srl} ( r_d, r_s, I )</td>
<td>( r_d = r_{s+1} \land I )</td>
<td>( 00000000000000001111 )</td>
</tr>
</tbody>
</table>

- \texttt{sr} and \texttt{srl} shift zeros into most-significant bit.
- This matches shift in \texttt{C} of \textit{unsigned} value.
- \texttt{sra} and \texttt{sra} propagate most-significant bit.
- This ensures shifting a negative number divides by 2.
- \texttt{slav} and \texttt{sla} don’t exist as arithmetic and logical left shifts are the same.
- \texttt{mipsy} provides \texttt{rol} and \texttt{ror} pseudo-instructions which rotate bits.
- Real instructions on some MIPS versions.
- No simple C equivalent.
### Example Use of 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><code>R_d = value</code></td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td><code>la R_d, label</code></td>
<td><code>R_d = label</code></td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td><code>move R_d, R_s</code></td>
<td><code>R_d = R_s</code></td>
<td>pseudo-instruction</td>
</tr>
</tbody>
</table>
| `slt R_d, R_s, R_t` | `R_d = R_s < R_t` | `000000ssssstttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
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`

https://www.cse.unsw.edu.au/~cs1521/COMP1521 24T2 — MIPS Basics 28 / 38

Using Mipsy

How to 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/  
  - 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

https://www.cse.unsw.edu.au/~cs1521/COMP1521 24T2 — MIPS Basics 29 / 38

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
Important System Calls

Our programs can't really do anything ... we usually rely on the operating system 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.

$\texttt{sv0}$ specifies which system call —

<table>
<thead>
<tr>
<th>Service</th>
<th>$\texttt{sv0}$</th>
<th>Arguments</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;%d&quot;)</code></td>
<td>1</td>
<td>int in $\texttt{a0}$</td>
<td></td>
</tr>
<tr>
<td><code>fprintf(&quot;%d&quot;)</code></td>
<td>4</td>
<td>string in $\texttt{a0}$</td>
<td></td>
</tr>
<tr>
<td><code>scanf(&quot;%d&quot;)</code></td>
<td>5</td>
<td>none</td>
<td>int in $\texttt{sv0}$</td>
</tr>
<tr>
<td><code>fgets</code></td>
<td>8</td>
<td>line in $\texttt{a0}$, length in $\texttt{a1}$</td>
<td></td>
</tr>
<tr>
<td><code>exit()</code></td>
<td>10</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;%c&quot;)</code></td>
<td>11</td>
<td>char in $\texttt{a0}$</td>
<td></td>
</tr>
<tr>
<td><code>scanf(&quot;%c&quot;)</code></td>
<td>12</td>
<td>none</td>
<td>char in $\texttt{sv0}$</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>$\texttt{sv0}$</th>
<th>Arguments</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;%f&quot;)</code></td>
<td>2</td>
<td>float in $\texttt{f12}$</td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;%lf&quot;)</code></td>
<td>3</td>
<td>double in $\texttt{f12}$</td>
<td></td>
</tr>
<tr>
<td><code>scanf(&quot;%f&quot;)</code></td>
<td>6</td>
<td>none</td>
<td>float in $\texttt{f0}$</td>
</tr>
<tr>
<td><code>scanf(&quot;%lf&quot;)</code></td>
<td>7</td>
<td>none</td>
<td>double in $\texttt{f0}$</td>
</tr>
<tr>
<td><code>sbrk(nbytes)</code></td>
<td>9</td>
<td>nbytes in $\texttt{a0}$</td>
<td>address in $\texttt{sv0}$</td>
</tr>
<tr>
<td><code>open(filename, flags, mode)</code></td>
<td>13</td>
<td>filename in $\texttt{a0}$, flags in $\texttt{a1}$, mode $\texttt{a2}$</td>
<td>fd in $\texttt{sv0}$</td>
</tr>
<tr>
<td><code>read(fd, buffer, length)</code></td>
<td>14</td>
<td>fd in $\texttt{a0}$, buffer in $\texttt{a1}$, length in $\texttt{a2}$</td>
<td>number of bytes read in $\texttt{sv0}$</td>
</tr>
<tr>
<td><code>write(fd, buffer, length)</code></td>
<td>15</td>
<td>fd in $\texttt{a0}$, buffer in $\texttt{a1}$, length in $\texttt{a2}$</td>
<td>number of written in $\texttt{sv0}$</td>
</tr>
<tr>
<td><code>close(fd)</code></td>
<td>16</td>
<td>fd in $\texttt{a0}$</td>
<td></td>
</tr>
<tr>
<td><code>exit(status)</code></td>
<td>17</td>
<td>status in $\texttt{a0}$</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><code>add $a3, $t0, $zero</code></td>
<td>000000 sssss ttttt ddddd 00000 100000</td>
</tr>
<tr>
<td><code>add $d, $s, $t</code></td>
<td>000000 01000 00000 00111 00000 100000</td>
</tr>
<tr>
<td><code>add $7, $8, $0</code></td>
<td>0x01003820 (decimal 16791584)</td>
</tr>
<tr>
<td><code>addi $v0, $v0, 1</code></td>
<td>000000 sssss ttttt ddddd 00000 100010</td>
</tr>
<tr>
<td><code>addi $d, $s, C</code></td>
<td>000000 00001 00101 00000 100010</td>
</tr>
<tr>
<td><code>add $d, $s, $t</code></td>
<td>0x00232822 (decimal 2304034)</td>
</tr>
<tr>
<td><code>addi $v0, $v0, 1</code></td>
<td>000000 sssss ddddd CCCCCCCCCCCCCC</td>
</tr>
<tr>
<td><code>add $d, $s, $t</code></td>
<td>001000 00010 00101 00000000000001</td>
</tr>
<tr>
<td><code>add $d, $s, $t</code></td>
<td>0x20420001 (decimal 541196289)</td>
</tr>
</tbody>
</table>

all instructions are variants of a small number of bit patterns with register numbers always in same place
MIPS Assembly Language

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:

\[
\text{MAX\_NUMBERS} = 1000
\]

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 of
    li $v0, 4       # ... 4 is printf "%s"
    syscall
    li $v0, 0
    jr $ra
.data
string:
    .asciiz "I love MIPS\n"
```

source code for i_love_mips.s

MIPS Programming

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("\n");
    return 0;
}
```

Adding Two Numbers — Simple C to MIPS

Simplified

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

MIPS

```assembly
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("\n");
    li $v0, 11
    syscall
    li $v0, 0  # return 0
    jr $ra
```

source code for add.s

source code for add.simple.c