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!

CPU Components

A typical modern CPU has

- a set of **data** registers
- a set of **control** registers (including PC)
- an **arithmetic-logic unit (ALU)**
- access to **memory** (RAM)
- a set of simple instructions
  - transfer data between memory and registers
  - push values through the ALU to compute results
  - make tests and transfer control of execution

Different types of processors have different configurations of the above
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>R4300</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>xbone</td>
<td>x86</td>
<td>AMD Zen 2 (8 cores)</td>
<td>3700</td>
</tr>
</tbody>
</table>

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 appropriate register

Example instruction encodings
(not from a real machine):

```
ADD       R1       R2       R3
8 bits    8 bits    8 bits    8 bits

LOAD      R4       0x10004
8 bits    8 bits    16 bits
```
MIPS Architecture

MIPS is a well-known and simple architecture

- historically used everywhere from supercomputers to PlayStations, ...
- 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 ... new spim/qtspim replacement written by Zac
- mipsy-web ... web-based (WASM) version of mipsy
- spim ... command-line-based simulator, good for testing
- qtspim ... GUI-based simulator, better for debugging
- xspim ... also GUI-based simulator - 1521 students prefer qtspim

MIPSY source: https://github.com/insou22/mipsy

Spim executables and source: http://spimsimulator.sourceforge.net/

Source code for browsing under /home/cs1521/spim

MIPS Instructions

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
- special ... miscellaneous tasks (e.g. syscall)

And several addressing modes for each instruction:

- between memory and register — direct, indirect
- constant to register — immediate
- register + register + destination register

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

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

Some possible instruction formats

<table>
  <tr>
    <th>OPCODE</th>
    <th>R1</th>
    <th>R2</th>
    <th>R3</th>
    <th>unused</th>
  </tr>
  <tr>
    <td>6 bits</td>
    <td>5 bits</td>
    <td>5 bits</td>
    <td>11 bits</td>
  </tr>
</table>

<table>
  <tr>
    <th>OPCODE</th>
    <th>R1</th>
    <th>Memory Address or Constant Value</th>
  </tr>
  <tr>
    <td>6 bits</td>
    <td>5 bits</td>
    <td>21 bits</td>
  </tr>
</table>
Assembly Language

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

- Could write machine code program just by specifying bit-patterns
  e.g as a sequence of hex digits:
  
  0x3c041001 0x3e020004 0x0000000c 0x03e00008
  
  - unreadable! difficult to maintain!

- adding/removing instructions changes bit pattern for other instructions
- changing variable layout in memory changes bit pattern for instructions

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
- allow names (labels) associated with memory addresses

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]
addi $t2, $t3, 5   # reg[t2] = reg[t3] + 5
mult $t3, $t4      # (Hi,Lo) = reg[t3] * reg[t4]
slt $t7, $t1, $t2  # reg[t7] = (reg[t1] < reg[t2])
addi $t2, $t1, $t2  # reg[t2] = reg[t1] + reg[t2]
```

MIPS Architecture: Registers

MIPS CPU has

- 32 general purpose registers (32-bit)
- 16/32 floating-point registers (for float/double)
- PC ... 32-bit register (always aligned on 4-byte boundary)
- Hi, Lo ... for storing results of multiplication and division

Registers can be referred to as $0...$31, or by symbolic names

Some registers have special uses; e.g.,

- register $0 always has value 0, discards all written values
- registers $1, $26, $27 reserved for use by system

More details on following slides...
### 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 OS kernel</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>Pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>$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 instruction)</td>
</tr>
</tbody>
</table>

- Except for registers 0 and 31, these uses are only programmers conventions
  - no difference between registers 1..30 in the silicon
- Conventions allow compiled code from different sources to be combined (linked).
- Some of these conventions are irrelevant when writing tiny assembly programs ... follow them anyway
- for general use, keep to registers $t0..$t9, $s0..$t7
- use other registers only for conventional purpose
  - e.g. only use $a0..$a3 for arguments
  - never use registers 1, 26, 27 ($at, $k0, $k1)

### MIPS Architecture: Floating-Point Registers

<table>
<thead>
<tr>
<th>Reg</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f0..$f2</td>
<td>hold return value of functions which return floating-point results</td>
</tr>
<tr>
<td>$f4..$f10</td>
<td>temporary registers; not preserved across function calls</td>
</tr>
<tr>
<td>$f12..$f14</td>
<td>used for first two double-precision function arguments</td>
</tr>
<tr>
<td>$f16..$f18</td>
<td>temporary registers; used for expression evaluation</td>
</tr>
<tr>
<td>$f20..$f30</td>
<td>saved registers; value is preserved across function calls</td>
</tr>
</tbody>
</table>

Floating-point registers come in pairs:
- either use all 32 as 32-bit registers,
- or use only even-numbered registers for 16 64-bit registers

COMP1521 will not explore floating point on the MIPS
Data and Addresses

All operations refer to data, either
- in a register
- in memory
- a constant which is embedded in the instruction itself

Computation operations refer to registers or constants.

Only load/store instructions refer to memory.

To access registers, you can also use $\text{name}$
e.g. $\text{zero} == 0$, $\text{t0} == 8$, $\text{fp} == 30$, ...

The syntax for constant value is C-like:
1 3 -1 -2 12345 0x1 0xFFFFFFFF "a string" 'a' 'b' '1' '\n' '\0'

Describing MIPS Assembly Operations

Registers are denoted:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Meaning</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_d$</td>
<td>destination register</td>
<td>where result goes</td>
<td>$R_d$ = $R_s$ + $R_t$</td>
</tr>
<tr>
<td>$R_s$</td>
<td>source register #1</td>
<td>where data comes from</td>
<td>$R_d, R_s, R_t$</td>
</tr>
<tr>
<td>$R_t$</td>
<td>source register #2</td>
<td>where data comes from</td>
<td>$R_d, R_s, R_t$</td>
</tr>
</tbody>
</table>

For example:

$$\text{add } R_d, R_s, R_t \implies R_d := R_s + R_t$$

Integer Arithmetic Instructions

- integer arithmetic is 2’s-complement.
- see also: addu, subu, mulu, addiu:
  - instructions which do not stop execution on overflow.
- mipsy & SPIM allows second operand ($R_t$) to be replaced by a constant, and will generate appropriate real MIPS instructions(s).
## Bit Manipulation Instructions

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>div</strong> $r_d, r_s, r_t$</td>
<td>hi = $r_s \div r_t$; lo = $r_s \mod r_t$</td>
<td>00000000000000000000000000000010</td>
</tr>
<tr>
<td><strong>mult</strong> $r_d, r_s, r_t$</td>
<td>hi = $(r_s \times r_t) \div 32$; lo = $(r_s \times r_t) \mod 0xffffffff$</td>
<td>00000000000000000000000000000000</td>
</tr>
<tr>
<td><strong>mflo</strong> $r_d$</td>
<td>$r_d = \text{lo}$</td>
<td>00000000000000dddd00000000000100</td>
</tr>
<tr>
<td><strong>mfhi</strong> $r_d$</td>
<td>$r_d = \text{hi}$</td>
<td>0000000000000000000000000000000010</td>
</tr>
</tbody>
</table>

- **mult** provides multiply with 64-bit result
- little use of these instructions in COMP1521 except challenge exercises
- pseudo-instruction **rem** $r_d, r_s, r_t$ translated to **div** $r_d, r_s, r_t$ plus **mfhi** $r_d$
- pseudo-instruction **div** $r_d, r_s, r_t$ translated to **div** $r_d, r_s, r_t$ plus **mflo** $r_d$
- **divu** and **multu** are unsigned equivalents of **div** and **mult**

### Assembly

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>and</strong> $r_d, r_s, r_t$</td>
<td>$r_d = r_s &amp; r_t$</td>
<td>0000000000000000000000000000001000</td>
</tr>
<tr>
<td><strong>or</strong> $r_d, r_s, r_t$</td>
<td>$r_d = r_s \lor r_t$</td>
<td>0000000000000000000000000000001001</td>
</tr>
<tr>
<td><strong>xor</strong> $r_d, r_s, r_t$</td>
<td>$r_d = r_s \oplus r_t$</td>
<td>0000000000000000000000000000001000</td>
</tr>
<tr>
<td><strong>nor</strong> $r_d, r_s, r_t$</td>
<td>$r_d = \lnot(r_s \lor r_t)$</td>
<td>0000000000000000000000000000001001</td>
</tr>
<tr>
<td><strong>andi</strong> $r_d, r_s, I$</td>
<td>$r_d = r_s &amp; I$</td>
<td>000000000000000000000000000000001111</td>
</tr>
<tr>
<td><strong>ori</strong> $r_d, r_s, I$</td>
<td>$r_d = r_s \lor I$</td>
<td>000000000000000000000000000000000111</td>
</tr>
<tr>
<td><strong>xor</strong> $r_d, r_s, I$</td>
<td>$r_d = r_s \oplus I$</td>
<td>000000000000000000000000000000000111</td>
</tr>
<tr>
<td><strong>not</strong> $r_d, r_s$</td>
<td>$r_d = \lnot r_s$</td>
<td>pseudo-instruction</td>
</tr>
</tbody>
</table>

- spim translates **not** $r_d, r_s$ to **nor** $r_d, r_s, 0$
- instructions explained later when we cover bitwise operators

### Shift Instructions

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sllv</strong> $r_d, r_t, r_s$</td>
<td>$r_d = r_t \ll r_s$</td>
<td>0000000000000000000000000000001000</td>
</tr>
<tr>
<td><strong>sriv</strong> $r_d, r_t, r_s$</td>
<td>$r_d = r_t \rr r_s$</td>
<td>0000000000000000000000000000001010</td>
</tr>
<tr>
<td><strong>sraiv</strong> $r_d, r_t, r_s$</td>
<td>$r_d = r_t \rr r_s$</td>
<td>0000000000000000000000000000001011</td>
</tr>
<tr>
<td><strong>sll</strong> $r_d, r_t, I$</td>
<td>$r_d = r_t \ll I$</td>
<td>0000000000000000000000000000001100</td>
</tr>
<tr>
<td><strong>sr</strong> $r_d, r_t, I$</td>
<td>$r_d = r_t \rr I$</td>
<td>0000000000000000000000000000001101</td>
</tr>
<tr>
<td><strong>sra</strong> $r_d, r_t, I$</td>
<td>$r_d = r_t \rr I$</td>
<td>0000000000000000000000000000001110</td>
</tr>
</tbody>
</table>

- **srl** and **srlv** shift zeros into most-significant bit
  - this matches shift in C of **unsigned** value
- **sra** and **sraiv** propagate most-significant bit
  - this ensure shifting a negative number divides by 2
- spim provides **rol** and **ror** pseudo-instructions which rotate bits
  - real instructions on some MIPS versions
  - no simple C equivalent
- instructions explained later when we cover bitwise operators
### Example Translation of Pseudo-instructions

<table>
<thead>
<tr>
<th>Pseudo-Instructions</th>
<th>Real Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $a1, $v0</td>
<td>addu $a1, $0, $v0</td>
</tr>
<tr>
<td>li $t5, 42</td>
<td>ori $t5, $0, 42</td>
</tr>
<tr>
<td>li $s1, 0xdeadbeef</td>
<td>lui $at, 0xdead</td>
</tr>
<tr>
<td></td>
<td>ori $s1, $at, 0xbeef</td>
</tr>
<tr>
<td>la $t3, label</td>
<td>lui $at, label[31..16]</td>
</tr>
<tr>
<td></td>
<td>ori $t3, $at, label[15..0]</td>
</tr>
</tbody>
</table>

### MIPS vs mipsy & SPIM

MIPS is a machine architecture, including instruction set.

Mipsy & SPIM are **emulators** for the MIPS instruction set:

- reads text files containing instruction + directives
- converts to machine code and loads into "memory"
- provides (primitive) 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 SPIM

Five ways to execute MIPS code with SPIM...

**mipsy and spim** ... command line tools

- load programs using command line arguments
- interact using stdin/stdout via terminal

**qtspim** ... GUI environment

- load programs via a load button
- interact via a pop-up stdin/stdout terminal

**xspim** ... GUI environment

- similar to qtspim, but not as pretty

Plus we have **mipsy web**... zac & shrey's web-based version of mips which should make 1521 students life better,
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.

SPIM 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>printf(&quot;%s&quot;)</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>status in $a0</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 - any input is mostly integers

Also system calls 13…16 support file I/O: open, read, write, close.

Not used in COMP1521 and rarely used by anyone.

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>0000000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>add $d, $s, $t</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>add $7, $8, $0</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>sub $a1, $at, $v1</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>sub $d, $s, $t</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>sub $5, $1, $3</td>
<td>0000000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>addi $v0, $v0, 1</td>
<td>0010000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>addi $d, $s, $c</td>
<td>0010000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>addi $2, $2, $1</td>
<td>0010000000000000000000000000000000000000000000000000000000000000</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

- comments ... introduced by #
- labels ... appended with :
- directives ... symbol beginning with .
- assembly language instructions

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
int main(void) {
    printf("I love MIPS\n");
    return 0;
}
MIPS
main:
# ... pass address of string as argument
la $a0, string
# ... 4 is printf ”%s” syscall number
li $v0, 4
syscall
li $v0, 0  # return 0
jr $ra
.data
string:
    .asciiz "I love MIPS\n"
```