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 PPC750</td>
<td></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 PPC Broadway</td>
<td></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 NVidia TX1</td>
<td></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>
</tbody>
</table>

---

**Fetch-Execute Cycle**

- typical CPU program execution pseudo-code:

```plaintext
word pc = START_ADDRESS;
while (1) {
  word instruction = memory[pc];
  pc++; // move to next instr
  if (instruction == HALT)
    break;
  else
    execute(instruction);
}
```

- **pc** = Program Counter, a special CPU register which tracks execution
  - note some instructions modify pc (branches and jumps)

---

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):

```
<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>LOAD</th>
<th>R4</th>
<th>0x10004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>
```
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

We consider the MIPS32 version of the MIPS family, running on SPIM
- qtspim ... provides a GUI front-end, useful for debugging
- spim ... command-line based version, useful for testing
- xspim ... GUI front-end, useful for debugging, only in CSE labs

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
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 0x34020004 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

Examples MIPS Assembler

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]
seq $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)
- 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 (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 $name e.g. $zero == $0, $t0 == $8, $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:

- $R_d$ 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 ⟹ $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</td>
<td>$r_d$, $r_s$, $r_t$</td>
<td>$r_d = r_s + r_t$</td>
</tr>
<tr>
<td>sub</td>
<td>$r_d$, $r_s$, $r_t$</td>
<td>$r_d = r_s - r_t$</td>
</tr>
<tr>
<td>mul</td>
<td>$r_d$, $r_s$, $r_t$</td>
<td>$r_d = r_s * r_t$</td>
</tr>
<tr>
<td>rem</td>
<td>$r_d$, $r_s$, $r_t$</td>
<td>$r_d = r_s % r_t$</td>
</tr>
<tr>
<td>div</td>
<td>$r_d$, $r_s$, $r_t$</td>
<td>$r_d = r_s / r_t$</td>
</tr>
<tr>
<td>addi</td>
<td>$r_t$, $r_s$, I</td>
<td>$r_t = r_s + I$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Meaning</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>div r_s, r_t</code></td>
<td>hi = ( r_s % r_t )</td>
<td>000000sssssttttt0000000011010</td>
</tr>
<tr>
<td></td>
<td>lo = ( r_s / r_t )</td>
<td></td>
</tr>
<tr>
<td><code>mult r_s, r_t</code></td>
<td>hi = ( r_s \times r_t ) \times 32</td>
<td>000000sssssttttt0000000011000</td>
</tr>
<tr>
<td></td>
<td>lo = ( r_s \times r_t ) &amp; 0xffffffff</td>
<td></td>
</tr>
<tr>
<td><code>mflo r_d</code></td>
<td>( r_d = \text{lo} )</td>
<td>0000000000000000000001010</td>
</tr>
<tr>
<td><code>mfhi r_d</code></td>
<td>( r_d = \text{hi} )</td>
<td>0000000000000000000001001</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_s, r_t` plus `mfhi r_d`
- Pseudo-instruction `div r_d, r_s, r_t` translated to `div r_s, r_t` plus `mflo r_d`
- `divu` and `multu` are unsigned equivalents of `div` and `mult`

### Bit Manipulation Instructions

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Meaning</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and r_d, r_s, r_t</code></td>
<td>( r_d = r_s &amp; r_t )</td>
<td>000000ssssstttttddddddddddd000001100</td>
</tr>
<tr>
<td><code>or r_d, r_s, r_t</code></td>
<td>( r_d = r_s</td>
<td>r_t )</td>
</tr>
<tr>
<td><code>xor r_d, r_s, r_t</code></td>
<td>( r_d = r_s ^ r_t )</td>
<td>000000ssssstttttddddddddddd000001100</td>
</tr>
<tr>
<td><code>nor r_d, r_s, r_t</code></td>
<td>( r_d = \neg (r_s</td>
<td>r_t) )</td>
</tr>
</tbody>
</table>
| `andi r_t, r_s, I` | \( r_t = r_s \& I \)    | 001100ssssstttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
# examples of miscellaneous instructions...

```plaintext
start:
li $8, 42  # $8 = 42
li $24, 0x2a  # $24 = 42
li $15, 'x'  # $15 = 42
move $8, $9  # $8 = $9
la $8, start  # $8 = address corresponding to start
```

## 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>move $8, $9</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 SPIM

MIPS is a machine architecture, including instruction set

SPIM is an emulator 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 SPIM

Three ways to execute MIPS code with SPIM...

spim ... command line tool
- load programs using -file option
- 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
- requires X Windows

Using SPIM Interactively

$ 1521 spim
(spim) load "myprogram.s"
(spim) step 6
[0x00400000] 0x8fa40000 lw $4, 0($29)
[0x00400004] 0x27a50004 addiu $5, $29, 4
[0x00400008] 0x24a60004 addiu $6, $5, 4
[0x0040000c] 0x00041080 sll $2, $4, 2
[0x00400010] 0x00c23021 addu $6, $6, $2
[0x00400014] 0x0c100009 jal 0x00400024 [main]
(spim) print_all_regs hex

....

General Registers
R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 ...
R1 (at) = 10010000 R9 (t1) = 00000000 R17 (s1) = 00000000 ...
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.

$\$\text{v0}$ specifies which system call —

<table>
<thead>
<tr>
<th>Service</th>
<th>$$\text{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 $\text{v0}$</td>
</tr>
<tr>
<td>fgets</td>
<td>8</td>
<td>$a0$: line, $a1$: length</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 $\text{v0}$</td>
</tr>
</tbody>
</table>

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

**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 dddd0000001000000</td>
</tr>
<tr>
<td>add $d, $s, $t</td>
<td>000000 00111 01000 0000000000100000</td>
</tr>
<tr>
<td>add $7, $8, $0</td>
<td>0x01e80020 (decimal 31981600)</td>
</tr>
<tr>
<td>sub $a1, $at, $v1</td>
<td>000000 sssss ttttt dddd0000001000000</td>
</tr>
<tr>
<td>sub $d, $s, $t</td>
<td>000000 00001 00011 00101000000010000</td>
</tr>
<tr>
<td>sub $5, $1, $3</td>
<td>0x00232822 (decimal 2304034)</td>
</tr>
<tr>
<td>addi $v0, $v0, 1</td>
<td>001000 sssss dddd CCCCCCCCCCCCCCC</td>
</tr>
<tr>
<td>addi $d, $s, $c</td>
<td>01000 0010 0010 000000000000000001</td>
</tr>
<tr>
<td>addi $2, $2, $1</td>
<td>0x20420001 (decimal 54196289)</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"
```

source code for i_love_mips.s source code for i_love_mips.c