Levels of representation

<table>
<thead>
<tr>
<th>High Level Language Program</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Language Program</td>
<td>Assembler</td>
</tr>
<tr>
<td>Machine Language Program</td>
<td></td>
</tr>
<tr>
<td>Control Signal Specification</td>
<td></td>
</tr>
</tbody>
</table>

Software seen by hardware is a sequence of machine code!

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!
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 Components

![CPU components diagram]

### 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>P55</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

A 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);
}
```
Fetch-Execute Cycle

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>Instruction</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>LOAD</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

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 (WASM) GUI-based version of mipsy
- spim ... command-line-based simulator, not as useful anymore
- qtspim ... GUI-based version of spim
- xspim ... also GUI-based simulator - students prefer qtspim over xspim

COMP1521 will use mipsy, but it’s worth knowing that other simulators exist.

mipsy source code: https://github.com/insou22/mipsy
Spim executables and source: http://spimsimulator.sourceforge.net/

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
MIPS Instructions

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

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

Example MIPS Assembler

```assembly
lw    $t1, address
sw    $t3, address
la    $t1, address
lui   $t2, const
and   $t0, $t1, $t2
add   $t0, $t1, $t2
addi  $t2, $t3, 5
mult  $t3, $t4
slt   $t7, $t1, $t2
j     label
beq   $t1, $t2, label
nop
```

- `lw` - load word
- `sw` - store word
- `la` - load absolute
- `lui` - load immediate
- `and` - and
- `add` - add
- `addi` - add immediate
- `mult` - multiply
- `slt` - set less than
- `j` - jump
- `beq` - branch if equal
- `nop` - no operation

- `# reg[t1] = memory[address]`
- `# memory[address] = reg[t3]`
- `# address must be 4-byte aligned`
- `# reg[t1] = address`
- `# reg[t2] = const << 16`
- `# reg[t0] = reg[t1] & reg[t2]`
- `# reg[t0] = reg[t1] + reg[t2]`
- `# add signed 2's complement ints`
- `# reg[t2] = reg[t3] + 5`
- `# add immediate, no sub immediate`
- `# (Hi,Lo) = reg[t3] * reg[t4]`
- `# store 64-bit result across Hi,Lo`
- `# reg[t7] = (reg[t1] < reg[t2])`
- `# PC = label`
- `# PC = label if reg[t1]==reg[t2]`
- `# do nothing`
MIPS integer registers

- R0-R31 or $0-$31
- They are also divided into groups for special uses.

<table>
<thead>
<tr>
<th>0</th>
<th>zero constant 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>at: reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0: expression evaluation &amp;</td>
</tr>
<tr>
<td>3</td>
<td>v1: function results</td>
</tr>
<tr>
<td>4</td>
<td>a0: arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0: temporary: caller saves</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0: callee saves</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8: temporary</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0: reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp: Pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>sp: Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp: Frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra: Return Address (HW)</td>
</tr>
</tbody>
</table>

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 ...

Some Terminology

Bit = Binary Digit
Byte = 8 bits
Word = 2 or more Bytes
usually 2, 4, 8, 16 bytes

Bits can be represented in differing forms

- Inside computers – voltage (1.2V is a 1 and 0 volts is a 0)
- On CDs and DVDs – pits on a surface
- On optical fibers – power of light (high 1 low 0)
- In air (Wi-Fi and Bluetooth)– Electromagnetic waves
There are only 10 types of students ...

- those that understand binary
- those that don't understand binary

### Decimal Representation

- Can interpret decimal number 4705 as:
  \[ 4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0 \]
- The base or radix is 10 ... digits 0 – 9
- Place values:

  \[
  \begin{array}{cccc}
  \cdots & 1000 & 100 & 10 & 1 \\
  \cdots & 10^3 & 10^2 & 10^1 & 10^0 \\
  \end{array}
  \]

- Write number as 4705\textsubscript{10}
  - Note use of subscript to denote base

### Representation in Other Bases

- Base 10 is an arbitrary choice
- Can use any base
- e.g. could use base 7
- Place values:

  \[
  \begin{array}{cccc}
  \cdots & 343 & 49 & 7 & 1 \\
  \cdots & 7^3 & 7^2 & 7^1 & 7^0 \\
  \end{array}
  \]

- Write number as 1216\textsubscript{7} and interpret as:
  \[ 1 \times 7^3 + 2 \times 7^2 + 1 \times 7^1 + 6 \times 7^0 = 454\textsubscript{10} \]
Binary Representation

Modern computing uses binary numbers because digital devices can easily produce high or low level voltages which can represent 1 or 0.

The base or radix is 2.

Digits 0 and 1

Place values:

\[
\begin{array}{cccc}
\cdots & 8 & 4 & 2 & 1 \\
\cdots & 2^3 & 2^2 & 2^1 & 2^0 \\
\end{array}
\]

Write number as 1011_2 and interpret as:

\[1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = = 11_{10}\]

Hexadecimal Representation

Binary numbers hard for humans to read — too many digits!

Conversion to decimal awkward and hides bit values

Solution: write numbers in hexadecimal!

The base or radix is 16... digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Place values:

\[
\begin{array}{cccc}
\cdots & 4096 & 256 & 16 & 1 \\
\cdots & 16^3 & 16^2 & 16^1 & 16^0 \\
\end{array}
\]

Write number as 3AF_{16} and interpret as:

\[3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0 = = 15089_{10}\]

in C, \texttt{0x} prefix denotes hexadecimal, e.g. \texttt{0x3AF1}

Octal & Binary C constants

Octal (based 8) representation used to be popular for binary numbers

Similar advantages to hexadecimal

in C a leading \texttt{0} denotes octal, e.g. \texttt{07563}

standard C doesn’t have a way to write binary constants

some C compilers let you write \texttt{0b}

OK to use \texttt{0b} in experimental code but don’t use in important code

```
printf("%d", 0x2A); // prints 42
printf("%d", 052); // prints 42
printf("%d", 0b101010); // might compile and print 42
```
Binary Constants

In hexadecimal, each digit represents 4 bits

```
0100 1000 1111 1010 1011 1100 1001 0111
0x  4  8  F  A  B  C  9  7
```

In octal, each digit represents 3 bits

```
01 001 000 111 110 101 011 110 010 010 111
0  1  1  0  7  6  5  3  6  2  2  7
```

In binary, each digit represents 1 bit

```
0b01001000111110101011110010010111
```

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>

MIPS Architecture: Integer Registers ... Usage Convention

- **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:

\[
\begin{array}{c c c}
R_d & \text{destination register} & \text{where result goes} \\
R_s & \text{source register #1} & \text{where data comes from} \\
R_t & \text{source register #2} & \text{where data comes from} \\
\end{array}
\]

For example:

\[
\text{add } R_d, R_s, R_t \implies R_d := R_s + R_t
\]
<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>000000sssssttttttddddd000001000000</td>
</tr>
<tr>
<td>sub (r_d, r_s, r_t)</td>
<td>(r_d = r_s - r_t)</td>
<td>000000sssssttttttddddd000001001010</td>
</tr>
<tr>
<td>mul (r_d, r_s, r_t)</td>
<td>(r_d = r_s * r_t)</td>
<td>011100sssssttttttddddd000000001010</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>
</tbody>
</table>
| addi \(r_t, r_s, I\) | \(r_t = r_s + I\) | 001000ssssstttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bitpattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>sllv 𝑟_𝑑, 𝑟_𝑡, 𝑟_𝑠</td>
<td>𝑟_𝑑 = 𝑟_𝑡 « 𝑟_𝑠</td>
<td>000000ssssstttttddddd00000000100</td>
</tr>
<tr>
<td>srlv 𝑟_𝑑, 𝑟_𝑡, 𝑟_𝑠</td>
<td>𝑟_𝑑 = 𝑟_𝑡 » 𝑟_𝑠</td>
<td>000000ssssstttttddddd00000000110</td>
</tr>
<tr>
<td>srav 𝑟_𝑑, 𝑟_𝑡, 𝑟_𝑠</td>
<td>𝑟_𝑑 = 𝑟_𝑡 » 𝑟_𝑠</td>
<td>000000ssssstttttddddd00000000111</td>
</tr>
<tr>
<td>sll 𝑟_𝑑, 𝑟_𝑡, I</td>
<td>𝑟_𝑑 = 𝑟_𝑡 « I</td>
<td>00000000000tttttddddd00000000100</td>
</tr>
<tr>
<td>srl 𝑟_𝑑, 𝑟_𝑡, I</td>
<td>𝑟_𝑑 = 𝑟_𝑡 » I</td>
<td>00000000000tttttddddd00000000110</td>
</tr>
<tr>
<td>sra 𝑟_𝑑, 𝑟_𝑡, I</td>
<td>𝑟_𝑑 = 𝑟_𝑡 » I</td>
<td>00000000000tttttddddd00000000111</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
- **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

### Miscellaneous Instructions

<table>
<thead>
<tr>
<th>assembly</th>
<th>meaning</th>
<th>bitpattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>li 𝑅_𝑑, value</td>
<td>𝑅_𝑑 = value</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>la 𝑅_𝑑, label</td>
<td>𝑅_𝑑 = label</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>move 𝑅_𝑑, 𝑅_𝑠</td>
<td>𝑅_𝑑 = 𝑅_𝑠</td>
<td>pseudo-instruction</td>
</tr>
<tr>
<td>slt 𝑅_𝑑, 𝑅_𝑠, 𝑅_𝑡</td>
<td>𝑅_𝑑 = 𝑅_𝑠 &lt; 𝑅_𝑡</td>
<td>000000ssssstttttddddd00000000101010</td>
</tr>
</tbody>
</table>
| slti 𝑅_𝑡, 𝑅_𝑠, I | 𝑅_𝑡 = 𝑅_𝑠 < I | 001010ssssstttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
**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 (or 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

- mipsy_web
  - runs in web browser, load programs with a button
  - visual environment for debugging
  - still new ... expect some bugs! (report on course forum)

- qtspim ... GUI environment
  - load programs via a load button
  - interact via a pop-up stdin/stdout terminal

- xspim ... similar to qtspim, but not as pretty

---

**Using SPIM**
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 [0x200020001] addi $v0, $zero, 1  # li $v0, 1
0x00400004 3 [0x2000402a] addi $a0, $zero, 42  # li $a0, 42
0x00400008 4 [0x0000000c] syscall # syscall

[SYSCALL 1] print_int: 42

[mipsy]

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.
$\texttt{\$v0}$ specifies which system call —

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

We won't use system calls 8, 12 much in COMP1521 - any input is mostly integers

System Calls ... Little Used in COMP1521

<table>
<thead>
<tr>
<th>Service</th>
<th>$\texttt{$v0}$</th>
<th>Arguments</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf(&quot;%f&quot;)</td>
<td>2</td>
<td>float in $\texttt{$f12}$</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;%lf&quot;)</td>
<td>3</td>
<td>double in $\texttt{$f12}$</td>
<td></td>
</tr>
<tr>
<td>scanf(&quot;%f&quot;)</td>
<td>6</td>
<td>none</td>
<td>float in $\texttt{$f0}$</td>
</tr>
<tr>
<td>scanf(&quot;%lf&quot;)</td>
<td>7</td>
<td>none</td>
<td>double in $\texttt{$f0}$</td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>nbytes in $\texttt{$a0}$</td>
<td>address in $\texttt{$v0}$</td>
</tr>
<tr>
<td>exit(status)</td>
<td>17</td>
<td>status in $\texttt{$a0}$</td>
<td></td>
</tr>
</tbody>
</table>

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>000000 ssrss sssst ddddd000000100000</td>
</tr>
<tr>
<td>add $d, $s, $t</td>
<td>000000 00111 01000 0000000000100000</td>
</tr>
<tr>
<td>sub $a1, $at, $v1</td>
<td>0x01e80020 (decimal 31981600)</td>
</tr>
<tr>
<td>add $d, $s, $t</td>
<td>000000 ssrss sssst ddddd000000100000</td>
</tr>
<tr>
<td>sub $5, $1, $3</td>
<td>000000 00011 00110 0100100000100010</td>
</tr>
<tr>
<td>addi $v0, $v0, 1</td>
<td>0x00232822 (decimal 2304034)</td>
</tr>
<tr>
<td>addi $d, $s, $c</td>
<td>001000 ssrss ddddd CCCCCCCCCCCCCCCC</td>
</tr>
<tr>
<td>addi $2, $2, 1</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

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

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

MIPS

```assembly
main:
    # ... pass address of string as argu
    la $a0, string
    # ... # is printf "%s" syscall numbe
    li $v0, 4
    syscall
    li $v0, 0       # return 0
    jr $ra
.data
string:
    .ascii "I love MIPS\n"
```