We will study the MIPS assembly language as an exemplar of the concept.

MIPS assembly instructions each consist of a single token specifying the command to be carried out, and zero or more operation arguments:

```
<opcode> arg0, arg1, ... argN
```

The tokens are separated by commas and (optionally) whitespace. Indentation is insignificant to the assembler, but is certainly significant to the human reader.

MIPS command tokens are short and mnemonic (in principle). For example:

```
add lw sw jr
```

The MIPS reference card bound in the front of P&H includes a complete listing of all the MIPS commands you will need to understand and use.

MIPS command arguments include:

- hardware registers
- offset and base register
- literal constants (*immediate* arguments)
- labels

Registers are specified either as $k$, where $0 \leq k \leq 31$, or using the symbolic names shown earlier.

Of course, MIPS assembly also allows comments. Simply, all characters from a `#` character to the end of the line are considered a comment.

There are also some special *directives*, but those can wait...
# PROGRAM: Hello, World!

```
.data  # Data declaration section
out_string: .ascii "\nHello, World!\n"

.text  # Assembly language instructions
main:  # Start of code section
    li $v0, 4           # system call code for printing string = 4
    la $a0, out_string # load address of string to be printed into $a0
    syscall            # call operating system to perform operation in $v0
                          # syscall takes its arguments from $a0, $a1, ...
```

This illustrates the basic structure of an assembly language program.
- data segment and text segment
- use of label for data object (which is a zero-terminated ASCII string)
- use of registers
- invocation of a system call

---

**MIPS Assembly Arithmetic Instructions**

All **arithmetic** and **logical** instructions have 3 operands

Operand order is **fixed** (destination first):

```
<opcode> <dest>, <src1>, <src2>
```

**Design Principle: simplicity favors regularity.**

Example:

C code: \[ a = b + c; \]

MIPS 'code': `add $s3, $s1, $s5`

“The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple”
Assembly Arithmetic Instructions

Design Principle: simplicity favors regularity.

Of course this complicates some things...

C code: \[ a = b + c + d; \]

MIPS pseudo-code:

\[
\begin{align*}
\text{add} & \ a, \ b, \ c \\
\text{add} & \ a, \ a, \ d
\end{align*}
\]

Operands must be registers (or immediates), only 32 registers are provided
Each register contains 32 bits

Design Principle: smaller is faster.

Why?

Immediates

In MIPS assembly, immediates are literal constants.

Many instructions allow immediates to be used as parameters.

\[
\begin{align*}
\text{addi} & \ \$t0, \ \$t1, \ 42 \ # \ note \ the \ opcode \\
\text{li} & \ \$t0, \ 42 \ # \ actually \ a \ pseudo-instruction
\end{align*}
\]

Note that immediates cannot be used with all MIPS assembly instructions; refer to your MIPS reference card.

Immediates may also be expressed in hexadecimal: \( 0xFFFFFFFF \)
MIPS Assembly Logical Instructions

Logical instructions also have 3 operands:

\[ \langle \text{opcode} \rangle \quad \langle \text{dest} \rangle, \langle \text{src1} \rangle, \langle \text{src2} \rangle \]

Examples:

- `and $s0, $s1, $s2` # bitwise AND
- `andi $s0, $s1, 42`
- `or $s0, $s1, $s2` # bitwise OR
- `ori $s0, $s1, 42`
- `nor $s0, $s1, $s2` # bitwise NOR (i.e., NOT OR)
- `sll $s0, $s1, 10` # logical shift left
- `srl $s0, $s1, 10` # logical shift right

QTP: MIPS assembly doesn’t include the logical operation `not`. Why?

How would you achieve the effect of a logical not operation in MIPS assembly?

Assembly Load and Store Instructions

Transfer data between memory and registers

Example:

- MIPS code:
  - `lw $t0, 32($s3)` # load word
  - `add $t0, $s2, $t0`
  - `sw $t0, 48($s3)` # store word

Can refer to registers by name (e.g., \(\$s2, \$t2\)) instead of number

Load command specifies destination **first**: opcode <dest>, <address>

Store command specifies destination **last**: opcode <dest>, <address>

Remember arithmetic operands are registers or immediates, not memory!

Can’t write: `add 48($s3), $s2, 32($s3)`
Addressing Modes

In **register** mode the address is simply the value in a register:

\[
\text{lw} \quad $t0, (\$s3)
\]

In **immediate** mode the address is simply an immediate value in the instruction:

\[
\text{lw} \quad $t0, 0
\]

In **base + register** mode the address is the sum of an immediate and the value in a register:

\[
\text{lw} \quad $t0, 100(\$s3)
\]

There are also various **label** modes:

\[
j \quad \text{absval} \\
j \quad \text{absval + 100} \\
j \quad \text{absval + 100}(\$s3)
\]

An Assembly Language Example

Can we figure out the code?

```c
void swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Conventions for procedure calls:
- arg0 is in register $4 (aka $a0)
- arg1 is in register $5 (aka $a1)
- ...
- return address is in register $31 ($ra)

```assembly
swap:     # need label to jump to in call
    sll  $t0, $a1, 2  # calculate offset of v[k]
    add  $t0, $a0, $t0  # add offset to array base address
    lw   $t2, 0($t0)  # load v[k] into register
    lw   $t3, 4($t0)  # load v[k+1] into register
    sw   $t3, 0($t0)  # store register v[k] to v[k+1]
    sw   $t2, 4($t0)  # store register v[k+1] to v[k]
    jr   $ra  # return to caller
```
So far we’ve learned…

MIPS
- loading words but addressing bytes
- arithmetic on registers only

<table>
<thead>
<tr>
<th># Instruction</th>
<th># Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
</tbody>
</table>

Conditional Branch Instructions

Decision making instructions
- alter the control flow,
- i.e., change the “next” instruction to be executed

MIPS conditional branch instructions:

- `bne $t0, $t1, <label>` # branch on not-equal
  # PC = &<label> if $t0 != $t1
- `beq $t0, $t1, <label>` # branch on equal

Labels are strings of alphanumeric characters, underscores and periods, not beginning with a digit. They are declared by placing them at the beginning of a line, followed by a colon character.

```
if ( i == j )
  h = i + j;

      bne $s0, $s1, Miss
      add $s3, $s0, $s1

Miss: ....
```
### Unconditional Branch Instructions

MIPS unconditional branch instructions:

```assembly
j Label    # PC = Label
jr $ra     # PC = $ra
```

```assembly
if ( i != j )
h = i + j;
else
h = i - j;
```

```assembly
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

Can you build a simple for loop?

### Conditional Set Instructions

MIPS conditional set instructions:

```assembly
slt $t0, $s0, $s1    # $t0 = 1 if $s0 < $s1
# $t0 = 0 otherwise
slti $t0, $s0, <imm>  # $t0 = 1 if $s0 < imm
# $t0 = 0 otherwise
```

There are similar unsigned versions.

```assembly
if ( i < j )
goto A;
else
    goto B;
```

```assembly
# $s3 -- i, $s4 -- j
    slt $t1, $s3, $s4
    beq $zero, $t1, B
A:    # code...
    j C
B:    # code...
C:    # code...
```
So far we’ve learned:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1,$s2,$s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1,$s2,$s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1,100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1,100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
<tr>
<td>bne $s4,$s5,L0</td>
<td>Next instr. is at L0 if $s4 ≠ $s5</td>
</tr>
<tr>
<td>beq $s4,$s5,L1</td>
<td>Next instr. is at L1 if $s4 = $s5</td>
</tr>
<tr>
<td>j Label</td>
<td>Next instr. is at Label</td>
</tr>
</tbody>
</table>

Custom Control Flow

We have: beq, bne, what about Branch-if-less-than?

Recall the set-less-than instruction:

```
slt $t0, $s1, $s2
```

If ($s1 < $s2)

```
$t0 = 1
```

else

```
$t0 = 0;
```

We can use this instruction to build "blt $s1, $s2, Label"
— can now build general control structures

Note that the assembler needs a register to do this,
— there are policy of use conventions for registers
Program Termination

Unlike the high-level languages you are accustomed to, MIPS assembly does not include an instruction, or block syntax, to terminate the program execution.

MIPS programs can be terminated by making a system call:

```assembly
## Exit
li $v0, 10  # load code for exit system call in $v0
syscall      # make the system call to exit
```

Without such code, the system would attempt to continue execution into the memory words that followed the final instructions of the program. That rarely produces graceful results.

Policy of Use Conventions

MIPS programmers are expected to conform to the following conventions when using the 29 available 32-bit registers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$st8-$st9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Register 1 ($at) is reserved for the assembler, 26-27 for operating system.
You may have noticed something is odd about a number of the MIPS instructions that have been covered so far. For example:

\[
\text{li } \$t0, 0xFFFFFFFF
\]

Now, logically there's nothing wrong with wanting to place a 32-bit value into one of the registers.

But there's certainly no way the instruction above could be translated into a 32-bit machine instruction, since the immediate value alone would require 32 bits.

This is an example of a pseudo-instruction. A MIPS assembler or interpreter like SPIM may be designed to support such extensions that make it easier to write complex programs.

In effect, the assembler supports an extended MIPS architecture that is more sophisticated than the actual MIPS architecture of the underlying hardware.

Of course, the assembler must be able to translate every pseudo-instruction into a sequence of valid MIPS assembly instructions.

Pseudo-Instruction Examples

\[
\text{move } \$t1, \$t2 \quad # \$t1 <-- \$t2
\]

\[
\text{or } \$t1, \$t2, \$zero \quad # \text{recall: } x \text{ OR } 0 \equiv x
\]

\[
\text{li } \$t1, <\text{imm}> \quad # \$t1 = 32\text{-bit imm value}
\]

---

# e.g., suppose <imm> is 0x23A0FB17

# The assembler sometimes needs a register in which it can
# store temporary values. The register $at is reserved for
# such use.

\[
\text{lui } \$at, 0x23A0 \quad # \text{put upper byte in upper byte of reg,}
\]

# and 0s in the lower byte

\[
\text{ori } \$t1, \$at, 0xFB18 \quad # \text{put lower byte into reg}
\]
We'd like to be able to load a 32-bit constant into a register.
Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 43690       # 1010101010101010
ori
```

low-order bits filled with zeros

```
0101010101010101 0000000000000000
0000000000000000 1001100110011001
```

Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 39321  # 1001100110011001
```

```
0101010101010101 0000000000000000 1010101010101010
1010101010101010 1001100110011001
```

MIPS Directives

Directives are special reserved identifiers used to communicate instructions to the assembler.

Directives begin with a period character.

For example:

```
.data       # mark beginning of a data segment
.text       # mark beginning of a text (code) segment
.align      # specify memory alignment of data
.space      # allocate uninitialized space in memory
.word       # store values in successive words
.byte       # store values in successive bytes
.asciiz     # store zero-terminated character sequence
.globl      # specify global scope for a label
```

Note: technically, directives are not part of the MIPS assembly language, but rather are instructions to the assembler to take certain actions.
## MIPS System Calls

<table>
<thead>
<tr>
<th>Service</th>
<th>Call code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>$a0 = integer to print</td>
<td></td>
</tr>
<tr>
<td>print float</td>
<td>2</td>
<td>$f12 = float to print</td>
<td></td>
</tr>
<tr>
<td>print double</td>
<td>3</td>
<td>$f12 = double to print</td>
<td></td>
</tr>
<tr>
<td>print string</td>
<td>4</td>
<td>$a0 = addr of null-term string</td>
<td></td>
</tr>
<tr>
<td>read integer</td>
<td>5</td>
<td>$v0 contains integer read</td>
<td></td>
</tr>
<tr>
<td>read float</td>
<td>6</td>
<td>$f0 contains float read</td>
<td></td>
</tr>
<tr>
<td>read double</td>
<td>7</td>
<td>$f0 contains double read</td>
<td></td>
</tr>
<tr>
<td>read string</td>
<td>8</td>
<td>$a0 = address of input buffer, $a1 = maximum # of chars to be read</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$v0 contains address of allocated memory</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print character</td>
<td>11</td>
<td>$a0 = character to print</td>
<td>$a0 contains character read</td>
</tr>
<tr>
<td>read character</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### MIPS System Calls

<table>
<thead>
<tr>
<th>Service</th>
<th>Call code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>open file</td>
<td>13</td>
<td>$a0 = addr of null-term string containing file name, $a1 = flags, $a2 = mode</td>
<td>$a0 contains file descriptor</td>
</tr>
<tr>
<td>read from file</td>
<td>14</td>
<td>$a0 = file descriptor, $a1 = addr of input buffer, $a2 = max # of chars to read</td>
<td>$a0 contains number of chars read</td>
</tr>
<tr>
<td>write to file</td>
<td>15</td>
<td>$a0 = file descriptor, $a1 = address of output buffer, $a2 = # of chars to write</td>
<td>$a0 contains number of chars written</td>
</tr>
<tr>
<td>close file</td>
<td>16</td>
<td>$a0 = file descriptor</td>
<td></td>
</tr>
<tr>
<td>exit2</td>
<td>17</td>
<td>$0 = termination result</td>
<td></td>
</tr>
</tbody>
</table>
Basic fact: at the machine language level there are no explicit data types, only contents of memory locations. The concept of type is present only implicitly in how data is used.

Declaration: reserving space in memory, or deciding that a certain data item will reside in a certain register.

Directives are used to reserve or initialize memory:

```
.data  # mark beginning of a data segment
.asciiz "a string"   # declare and initialize a string
.byte 13, 14, -3  # store values in successive bytes
.space 16           # allocate 16 bytes of space
.word 13, 14, -3  # store values in successive words
```

A complete listing of MIPS/SPIM directives can be found in P&H Appendix A.

Arrays

First step is to reserve sufficient space for the array.

Array elements are accessed via their addresses in memory, which is convenient if you’ve given the .space directive a suitable label.

```
.data
list: .word 2, 3, 5, 7, 11, 13, 17, 19, 23, 29
size: .word 10

la $t1, list     # get array address
li $t2, 0       # set loop counter

print_loop:
  beq $t2, $t3, print_loop_end   # check for array end
  lw $a0, ($t1)   # print value at the array pointer
  li $v0, 1
  syscall
  addi $t2, $t2, 1  # advance loop counter
  addi $t1, $t1, 4  # advance array pointer
  j print_loop     # repeat the loop

print_loop_end:
```
Array Example

This is part of the palindrome example from the course website:

```
data
string_space: .space 1024
...
# prior to the loop, $t1 is set to the address of the first
# char in string_space, and $t2 is set to the last one

array_example:
  bge $t1, $t2, is_palind
  # if lower pointer >= upper
  # pointer, yes

  lb $t3, ($t1)          # grab the char at lower ptr
  lb $t4, ($t2)          # grab the char at upper ptr
  bne $t3, $t4, not_palind
  # if different, it's not

  addu $t1, $t1, 1      # advance lower ptr
  subu $t2, $t2, 1      # advance upper ptr
  j test_loop          # repeat the loop
```

MIPS Instruction Summary

```
<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>3 operands, data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>3 operands, data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Load from memory to register</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Store from register to memory</td>
</tr>
<tr>
<td></td>
<td>lb</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Load from register to memory</td>
</tr>
<tr>
<td></td>
<td>sb</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Store from register to memory</td>
</tr>
<tr>
<td></td>
<td>lui</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2^16</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>beq</td>
<td>beq $s1, $s2, 25</td>
<td>($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test, PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>bne $s1, $s2, 25</td>
<td>($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test, PC-relative</td>
</tr>
<tr>
<td></td>
<td>slt</td>
<td>slt $s1, $s2, $s3</td>
<td>($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than, for beq, bne</td>
</tr>
<tr>
<td></td>
<td>slti</td>
<td>slti $s1, $s2, 100</td>
<td>($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>j</td>
<td>j 2500</td>
<td>go to target address</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td>jal 2500</td>
<td>Tpc = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
```
Other Issues

Not discussed yet:
- support for procedures
- linkers, loaders, memory layout
- stacks, frames, recursion
- manipulating strings and pointers
- interrupts and exceptions
- system calls and conventions

Some of these we'll talk more about later