We’ll be working with the MIPS instruction set architecture (ISA)
- similar to other architectures developed since the 1980’s
- almost 100 million MIPS processors manufactured in 2002
- used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …

Registers
- 32 general 32-bit registers, referred to as $0, $1, …, $31, or…
- 16 floating-point 64-bit registers, referred to as $f0, … $f15
- conventions govern the use of the general registers

We will, for now, adopt the view that the underlying computer is a “black box” that understands MIPS machine language.
### MIPS Assembly Language

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 parameters:

```
<opcode> par1 par2 ... parN
```

The tokens are separated by commas and/or 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 Assembly Language

MIPS command parameters include:

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

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:    .asciiz "\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

All arithmetic and logical instructions have 3 operands

Operand order is fixed (destination first):

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

Example:

C code:       a = b + c;
MIPS ‘code’:  add a, b, c

(we’ll talk about register syntax in a bit)

“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:

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

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.

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

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

Immediates may also be expressed in hexadecimal: 0xFFFFFF
Logical instructions also have 3 operands:

\[
\text{<opcode> \text{<dest>, <src1>, <src2>}}
\]

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?

- Operands to arithmetic and logical instructions must be registers or immediates.
- Compiler associates variables with registers
- What about programs with lots of variables?
Memory Organization

Viewed as a large, single-dimension array, with an address.

A memory address is an index into the array

"Byte addressing" means that the index points to a byte of memory.

MIPS Memory Organization

Bytes are nice, but most data items use larger "words"

For MIPS, a word is 32 bits or 4 bytes.

Registers hold 32 bits of data

2^{32} bytes with byte addresses from 0 to 2^{32} - 1
2^{30} words with byte addresses 0, 4, 8, ... 2^{32} - 4

Words are aligned, that is, each has an address that is a multiple of 4.

MIPS can be either big-endian (that is, the address of each word is the address of the "left-most" byte of the word) or little-endian. This is important when viewing the contents of memory.
Assembly Load and Store Instructions

Transfer data between memory and registers

Example:


MIPS code:

\[
\begin{align*}
\text{lw} & \quad \text{\$t0, 32($s3)} & \# \text{ load word} \\
\text{add} & \quad \text{\$t0, \$s2, \$t0} \\
\text{sw} & \quad \text{\$t0, 48($s3)} & \# \text{ store word}
\end{align*}
\]

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:

\[ \text{add} \quad 48($s3), \$s2, 32($s3) \]

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:

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

```plaintext
swap:               # need label to jump to in call
    muli $t0, $a1, 4    # 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>
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
  
- `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;
else
  h = i - j;
```

```
if ( i != j )
  h = i + j;
  bne $s0, $s1, Miss
  add $s3, $s0, $s1
  Miss: ....
else
  h = i - j;
  beq $s4, $s5, Lab1
  add $s3, $s4, $s5
  j Lab2
Lab1:  sub $s3, $s4, $s5
Lab2:  ...
```

Unconditional Branch Instructions

MIPS unconditional branch instructions:

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

```
if ( i != j )
  h = i + j;
  beq $s4, $s5, Lab1
  add $s3, $s4, $s5
  j Lab2
Lab1:  sub $s3, $s4, $s5
Lab2:  ...
else
  h = i - j;
```

Can you build a simple for loop?
**Conditional Set Instructions**

MIPS conditional set instructions:

- `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.

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

```plaintext
# $s3 == i, $s4 == j
slt $t1, $s3, $s4
bne $zero, $t1, B
A:  # code...
j  C
B:  # code...
C:
```

**So far we've learned:**

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

We have: \texttt{beq}, \texttt{bne}, what about Branch-if-less-than?

Recall the set-less-than instruction:

![MIPS code]

We can use this instruction to build \texttt{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-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$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.

Pseudo-Instructions

You may have noticed something is odd about a number of the MIPS instructions that have been covered so far. For example:

```
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 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

move $t1, $t2    #$t1 <- $t2

or $t1, $t2, $zero # recall: x OR 0 == x

li $t1, <imm>     #$t1 = 32-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.
lui $at, 0x23A0     # put upper byte in upper byte of reg,
# and 0s in the lower byte
ori $t1, $at, 0xFB18 # put lower byte into reg

lui and ori Details

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

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

ori $t0, $t0, 39321  # 1001100110011001

ori

low-order bits filled with zeros
### SPIM 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

### SPIM System Calls

<table>
<thead>
<tr>
<th>Service</th>
<th>System call code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0 = integer</td>
<td></td>
</tr>
<tr>
<td>print_flt</td>
<td>2</td>
<td>$f12 = float</td>
<td></td>
</tr>
<tr>
<td>print_dbl</td>
<td>3</td>
<td>$t12 = double</td>
<td></td>
</tr>
<tr>
<td>print_str</td>
<td>4</td>
<td>$a0 = string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td>integer (in $v0)</td>
<td></td>
</tr>
<tr>
<td>read_flt</td>
<td>6</td>
<td>float (in $f0)</td>
<td></td>
</tr>
<tr>
<td>read_dbl</td>
<td>7</td>
<td>double (in $f0)</td>
<td></td>
</tr>
<tr>
<td>read_str</td>
<td>8</td>
<td>$a0 = buffer, $a1 = length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td>address (in $v0)</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print_char</td>
<td>11</td>
<td>$a0 = char</td>
<td></td>
</tr>
<tr>
<td>read_char</td>
<td>12</td>
<td></td>
<td>char (in $a0)</td>
</tr>
<tr>
<td>open</td>
<td>13</td>
<td>$a0 = filename (string), $a1 = flags, $a2 = mode</td>
<td>file descriptor (in $a0)</td>
</tr>
<tr>
<td>read</td>
<td>14</td>
<td>$a0 = file descriptor, $a1 = buffer, $a2 = length</td>
<td>num chars read (in $a0)</td>
</tr>
<tr>
<td>write</td>
<td>15</td>
<td>$a0 = file descriptor, $a1 = buffer, $a2 = length</td>
<td>num chars written (in $a0)</td>
</tr>
<tr>
<td>close</td>
<td>16</td>
<td>$a0 = file descriptor</td>
<td></td>
</tr>
<tr>
<td>exit2</td>
<td>17</td>
<td>$a0 = 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
.ascii "a string"  # declare and initialize a string
.byte 13, 14, -3  # store values in successive bytes
.space 16          # alloc 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
add $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
test_loop:
  bge $t1, $t2, is_palin  # 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_palin # 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
...
```

Assembly Language vs. Machine Language

Assembly provides convenient symbolic representation
- much easier than writing down numbers
- e.g., destination first

Machine language is the underlying reality
- e.g., destination is no longer first

Assembly can provide 'pseudoinstructions'
- e.g., "move $t0, $t1" exists only in Assembly
- would be implemented using "add $t0, $t1, $zero"

When considering performance you should count real instructions
Of course, the hardware doesn’t really execute MIPS assembly language code.

The hardware can only store bits, and so the instructions it executes must be expressed in a suitable binary format.

We call the language made up of those instructions the **machine language**.

Different families of processors typically support different machine languages.

In the beginning, all programming was done in assembly language… very ugly…

Assembly languages were created to make the programming process more human-centric.

Assembly language code is translated into machine language by an **assembler**.

Alas, there is no universal assembly language. In practice, assembly languages are coupled with the underlying machine language and hardware.

Instructions, like registers and words of data, are also 32 bits long

Example:

```
add $t1, $s1, $s2
```

Registers have numbers, $t1 = 9, $s1 = 17, $s2 = 18

Machine language basic arithmetic/logic instruction format:

```
000000 10001 10010 01001 00000 100000
```

Can you guess what the field names stand for?

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>operation code (opcode)</td>
</tr>
<tr>
<td>rs</td>
<td>1st source register</td>
</tr>
<tr>
<td>rt</td>
<td>2nd source register</td>
</tr>
<tr>
<td>rd</td>
<td>destination register</td>
</tr>
<tr>
<td>shamt</td>
<td>shift amount</td>
</tr>
<tr>
<td>funct</td>
<td>opcode variant selector</td>
</tr>
</tbody>
</table>
MIPS Machine Language

Consider the load-word and store-word instructions,
- what would the regularity principle have us do?
- new principle: Good design demands a compromise

Introduce a new type of machine language instruction format
- I-type for data transfer instructions
- other format was R-type for register

Example: `lw $t0, 32($s2)`

Where's the compromise?

Overview of MIPS Machine Language

Simple instructions all 32 bits wide

Very structured, no unnecessary baggage

Only three instruction formats:

<table>
<thead>
<tr>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>16-bit address</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td>26-bit address</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rely on compiler to achieve performance
— what are the compiler's goals?

Help compiler where we can
### MIPS Hardware Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>MIPS operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $fp, $gp, $sp, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
<td></td>
</tr>
<tr>
<td>230 memory words</td>
<td>Memory[0], Memory[4] ... Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilted registers, such as those saved on procedure calls.</td>
<td></td>
</tr>
</tbody>
</table>

### 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>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three 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>load word</td>
<td>lw  $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw  $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb  $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb  $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * $216</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</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>branch on not equal</td>
<td>bne  $s1, $s2, 25</td>
<td>($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>set on less than</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>set less than immediate</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>jump</td>
<td>j    2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr   $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal  2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
### Addresses in Conditional Branch Instructions

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Next instruction is at:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bne $t4, $t5, Label</code></td>
<td><code># Label if $t4 != $t5</code></td>
</tr>
<tr>
<td><code>beq $t4, $t5, Label</code></td>
<td><code># Label if $t4 == $t5</code></td>
</tr>
</tbody>
</table>

**Machine language format:**

| I | op | rs | rt | 16-bit address |

Addresses are less than 32 bits, *necessarily* if the address is immediate within the instruction.

**Why do we care?**

**How do we handle this with load and store instructions?**

If we treat the 16-bit field as the absolute address of its target, then we limit the address space of every MIPS program to no more than $2^{16}$ bytes. That’s only 64 KiB!

### Increasing the Branch Range

**So, how can we eliminate the 64 KiB limitation?**

**Idea:** specify a register that would always be added to the 16-bit address field.
- this would make the address field an offset from the address in the register
- could use the PC (more or less) so we’d address relative to the current instruction
- in general, using a register in this way could increase the range of the branch to $2^{32}$ bytes, which would equal the limit of the memory size for MIPS
- could also interpret the branch distance as a number of words, not bytes

**Observation:** in actuality, conditional branches tend to be to nearby locations.
- PC-relative scheme would allow us to branch to locations $\pm 2^{15}$ bytes from the current instruction, or $\pm 2^{15}$ words.
- MIPS uses this approach, but by the time the branch address is computed the PC has already been incremented (by 4), so it’s relative to the location of what would have been the next instruction if the branch had not occurred

If we need to branch far away:

```
<table>
<thead>
<tr>
<th>bne</th>
<th>$s0, $s1, L1</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>L1</td>
</tr>
</tbody>
</table>
```
Addresses in Jump Instruction

# Instruction Next instruction is at:
\[ j \text{ Label} \quad \# \text{ Label} \]

Machine language format:

\[ J \text{ op} \quad 26\text{-bit address} \]

If the assembler simply replaces the label with its address, that would limit the size of the address space for MIPS programs to \(2^{26}\) words, or 256 MiB.

MIPS Addressing Mode Summary

Can you identify where, if anywhere, each mode is used in MIPS?

1. Immediate addressing

\[ \text{op} \quad \text{immediate} \]

2. Register addressing

\[ \text{op} \quad \text{r3} \quad \text{r2} \quad \ldots \quad \text{funct} \]

3. Base addressing

\[ \text{op} \quad \text{r3} \quad \text{r2} \quad \text{address} \]

4. PC-relative addressing

\[ \text{op} \quad \text{r3} \quad \text{r2} \quad \text{address} \]

5. Pseudodirect addressing

\[ \text{op} \quad \text{address} \]
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