Hardware Operations

- Every computer must be able to perform arithmetic
  \[ \text{add } a, b, c \]
- In order to do \( a = b + c + d + e \)
  \[ \text{add } a, b, c \quad \# \text{ Sum of } b+c \text{ to } a \]
  \[ \text{add } a, a, d \quad \# \text{ Sum of } b+c+d \text{ to } a \]
  \[ \text{add } a, a, e \quad \# \text{ Sum of } b+c+d+e \text{ to } a \]
Hardware operations

- # is used for comments (until the end of line)
- The natural number of operands for an operation like addition is three.

*Design Principle 1: Simplicity favors regularity*
Compiling a complex C assignment into MIPS

\( f = (g+h) - (i+j) \)

add t0, g, h # temporary variable t0 contains \( g+h \)
add t1, i, j # temporary variable t1 contains \( i+j \)
sub f, t0, t1# f gets \( t0 - t1 \)
MIPS Assembly Language

- Arithmetic operations
  - add. Usage: add a, b, c Meaning: a=b+c
  - sub. Usage: sub a, b, c Meaning: a=b-c
Operands of the hardware

- Operands of the instructions are from a limited number of special locations called **registers**
- The size of registers in MIPS is 32 bit.
- The **word** size in MIPS is 32 bit.
- MIPS has 32 registers. The reason of 32 registers in MIPS is

  *Design Principle 2: Smaller is faster*
Compiling a C assignment using registers

\[ f = (g+h)-(i+j) \]

- The variables f, g, h, i and j are assigned to the registers $s0, s1, s2, s3$ and $s4$ respectively.

```
add $t0, $s1, $s2  # temporary variable $t0$ contains $g+h$
add $t1, $s3, $s4  # temporary variable $t1$ contains $i+j$
sub $s0, $t0, $t1  # $f$ gets $t0-t1$
```
Memory operands

- MIPS must include instructions that transfer data between memory and registers:
  - Data transfer instructions
Compiling an assignment when an operand is in memory

- $g = h + A[8]$
- We can add two numbers when they are in registers. So transfer the memory data ($A[8]$) into a register.
- Assume the base address of the array is stored in $s3$
  
  ```
  lw $t0, 8($s3)  # temp reg $t0 gets A[8]
  ```
- $A[8]$ is in $t0$

  ```
  add $s1, $s2, $t0  # g = h + A[8]
  ```
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.
Memory organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

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

- In MIPS, words must start at addresses that are multiple of 4
  - Alignment restriction
- Computers divide into those that use the address of the leftmost (or “big end”) byte as the word address versus that use the rightmost (or “little end”) byte.
- MIPS is big-endian. In order to get A[8] we should load 32\textsuperscript{nd} byte (which is the 8.\textsuperscript{th} word)
Alignment

Alignment: require that objects fall on an address that is a multiple of their size.
Assume h is in $s2. The base address of the array A is in $s3.

Constant or immediate operands

- Adding constants to registers.
- Two way:
  - Load the constant from the memory location
  - Add immediate
    \[ \text{addi } \$s3, \$s3, 4 \]  # $s3 = s3 + 4
  - Constant operands occur frequently

*Design Principle 3: Make the common case fast*

- There is no subtract immediate! Why?
Representing instructions in the computer

- Since all kind of information is stored in computer as binary digits (*bits*) there should be binary representations of instructions.
- Mapping of register names into numbers.
- In MIPS assembly language
  - registers $s0$ to $s7$ map onto register numbers 16 to 23
  - registers $t0$ to $t7$ map onto register numbers 8 to 15
Translating a MIPS Assembly instruction into a machine instruction

- `add $t0, $s1, $s2`
- `registers have numbers, $t0=8, $s1=17, $s2=18`

**Instruction Format:**

- **op**: operation code (opcode)
- **rs**: the first register source operand
- **rt**: the second register source operand
- **rd**: the register destination operand
- **shamt**: shift amount. Used in shift operations
- **funct**: function. Specific variant of the opcode (function code)
Instruction formats

- R-type (for register) or R-format
- I-type (for immediate) or I-format
  - Example: `lw $t0, 32($s2)`

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>18</td>
<td>9</td>
<td>32</td>
</tr>
</tbody>
</table>

- The formats are distinguished by the values in the first field
## Logical operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>and $s1,$s2,$s3</td>
<td>$s1 = $s2 &amp; $s3</td>
</tr>
<tr>
<td>Or</td>
<td>or $s1,$s2,$s3</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td>Nor</td>
<td>nor $s1,$s2,$s3</td>
<td>$s1 = ~(s2</td>
</tr>
<tr>
<td>And immediate</td>
<td>andi $s1,$s2,100</td>
<td>$s1 = $s2 &amp; 100</td>
</tr>
<tr>
<td>Or immediate</td>
<td>ori $s1,$s2,100</td>
<td>$s1 = $s2</td>
</tr>
<tr>
<td>Shift left logical</td>
<td>sll $s1,$s2,10</td>
<td>$s1 = $s2&lt;&lt;10</td>
</tr>
<tr>
<td>Shift right logical</td>
<td>srl $s1,$s2,10</td>
<td>$s1 = $s2&gt;&gt;10</td>
</tr>
</tbody>
</table>
Branch instructions

beq register1, register2, L1
- goes to L1 if register1 == register2
bne register1, register2, L1
- goes to L1 if register1 != register2
if-then-else

- Replace the C code for
  
  ```c
  if (i == j) f = g + h; else f = g - h;
  ```
  
  by equivalent MIPS instructions.

- Assume variables f through j correspond to registers $s0$ through $s4$.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>bne $s3, $s4, Else</td>
<td>if (i != j) goto Else</td>
</tr>
<tr>
<td>add $s0, $s1, $s2</td>
<td>f = g + h</td>
</tr>
<tr>
<td>j Exit</td>
<td>go to Exit</td>
</tr>
<tr>
<td>Else:</td>
<td>sub $s0, $s1, $s2</td>
</tr>
<tr>
<td>Exit:</td>
<td>f = g - h</td>
</tr>
</tbody>
</table>

Jump to Exit
For loop

- Branch instructions end up the way we implement C-style loops

```c
for ( j = 0; j < 10; j++) {
    a = a + j;
}
```

- Assume $s0 == j; s1 == a; t0 == temp;

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $s0, $zero, 0</td>
<td>$j = 0 + 0</td>
</tr>
<tr>
<td>addi $t0, $zero, 10</td>
<td>$temp = 0 + 10</td>
</tr>
<tr>
<td>Loop: beq $s0, $t0, Exit</td>
<td>if ($j == $temp) goto Exit</td>
</tr>
<tr>
<td>add $s1, $s1, $s0</td>
<td>$a = $a + $j</td>
</tr>
<tr>
<td>addi $s0, $s0, 1</td>
<td>$j = $j + 1</td>
</tr>
<tr>
<td>j Loop</td>
<td>goto Loop</td>
</tr>
<tr>
<td>Exit:</td>
<td>exit from loop and continue</td>
</tr>
</tbody>
</table>
Set on less than

```
slt $t0, $s3, $s4
```
- Register $t0 is set to 1 if the value in register $s3 is less than the value in register $s4

```
slti $t0, $s3, 10
```
- Register $t0 is set to 1 if the value in register $s3 is less than the immediate value 10