Machine Language

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 machine 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.

---

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 as an extension to assembly
- would be implemented using "add $t0,$t1,$zero"

When considering performance you should count real instructions
MIPS Machine Language: Arithmetic Instructions

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
op  rs  rt  rd  shamt  funct
```

Can you guess what the field names stand for?

<table>
<thead>
<tr>
<th>op</th>
<th>operation code (opcode)</th>
</tr>
</thead>
<tbody>
<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>

Mapping Assembly to Machine Language

Note how the assembly instruction maps into the machine representation:

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

The three register fields are each 5 bits wide. Why?

For arithmetic-logical instructions, both the `op` field and the `funct` field may be used to specify the particular operation that is to be performed.

If you view memory contents, this would appear as `0x02324820`. 
MIPS Machine Language: Load Instructions

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>16-bit address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td>26-bit address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Arithmetical-logical instructions are R-format.
Load/store instructions are I-format.
Jump/branch instructions are J-format.
# 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># Label if $t4 != $t5</td>
</tr>
<tr>
<td><code>beq $t4, $t5, Label</code></td>
<td># Label if $t4 == $t5</td>
</tr>
</tbody>
</table>

Machine language format:

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

But this would limit the target addresses to 16 bits, necessarily since 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 if we stretch.
- 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:

```
bne $s0, $s1, L1
```

```
bne $s0, $s1, L2
j L1
```

```
L2:
```
### Addresses in Jump Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Next instruction is at:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>j Label</code></td>
<td><code># Label</code></td>
</tr>
</tbody>
</table>

Machine language format:

```
J op 26-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.

So… read the discussion on pages 97-99 in P&H.

Note that "stretching" the jump address also raises an issue of alignment…

### MIPS Addressing Mode Summary

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

1. Immediate addressing
   - `op` `im` `rt` Immediate

2. Register addressing
   - `op` `rs` `rt` `rd` ... `funct` Registers

3. Base addressing
   - `op` `rs` `rt` `addr` Memory

4. PC-relative addressing
   - `op` `rs` `rt` Address Memory

5. Pseudodirect addressing
   - `op` `rt` Address Memory
Example: asum.asm

```assembly
.data
x: .word -37
y: .word -12
z: .word 0

.text
main:  lw $s0, x       # put 1st operand into $s0
       lw $s1, y       # put 2nd operand into $s1

step1: bgez $s0, ispos1  # calculate |$s0| and put it into $t0
       sub $t0, $zero, $s0
       j step2
ispos1: add $t0, $zero, $s0

step2: bgez $s1, ispos2  # calculate |$s0| and put it into $t1
       sub $t1, $zero, $s1
       j step3
ispos2: add $t1, $zero, $s1

step3:  add $s2, $t0, $t1       # put sum into destination register
        sw $s2, z
```

Example: MARS Assembly

```assembly
Example: MARS Assembly

<table>
<thead>
<tr>
<th>B PC</th>
<th>Address</th>
<th>Code</th>
<th>Data</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>17000000</td>
<td>0x10010000</td>
<td>lw $s0, x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17000001</td>
<td>0x10010000</td>
<td>lw $s1, y</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```assembly
Example: MARS Assembly

<table>
<thead>
<tr>
<th>B PC</th>
<th>Address</th>
<th>Code</th>
<th>Data</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>17000002</td>
<td>0x10010000</td>
<td>lw $s0, x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17000003</td>
<td>0x10010000</td>
<td>lw $s1, y</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17000004</td>
<td>0x10010000</td>
<td>sub $s0, $s1, $zero</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17000005</td>
<td>0x10010000</td>
<td>add $t0, $zero, $s0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17000006</td>
<td>0x10010000</td>
<td>add $t1, $zero, $s1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17000007</td>
<td>0x10010000</td>
<td>add $t2, $t0, $t1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17000008</td>
<td>0x10010000</td>
<td>sw $t2, z</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Example: MARS Assembly Results

Here's the translation of a `lw` instruction into basic MIPS instructions:

\[
lw \quad \text{s1, y} \quad \rightarrow \quad lui \quad \text{s1, 4097} \quad \quad \text{y: } \quad 0x \quad \text{1001 0004} \\
\]

\[
lw \quad \text{s17, 4(s1)} \\
\]

\[
\text{at: } \quad 1001 0000 \\
\text{y: } \quad 4097_{10} \quad 4_{10} \\
\text{address: } \quad 1001 0004 \\
\]

Now consider the mapping into an I-format machine language instruction:

\[
lui \quad \text{s1, 4097: } \quad 3c01 \quad 1001 \quad 001111 \quad 00000 \quad 00001 \quad 0001000000000001 \\
\]

\[
lw \quad \text{s17, 4(s1): } \quad 8c31 \quad 0004 \quad 000011 \quad 000001 \quad 0001 \quad 000000000000000100 \\
\]

Example: MARS Assembly Results

Here's the translation of a `j` instruction:

\[
j \quad \text{step2} \quad \rightarrow \quad j \quad 4194336 \\
\]

\[
\text{step2: } \quad 0x00400020 \\
\]

… and here's what it looks like in MIPS machine code:

\[
0x08100008: \quad 000010 \quad 00000100000000000000000001000 \\
0x00400020: \quad 100 \quad 0000 \quad 0000 \quad 0000 \quad 0010 \quad 0000 \\
\]

Note how the literal in the assembly statement is shifted when it's placed into the immediate field of the machine instruction…
Example: MARS Assembly

Finally, here's the translation of a conditional branch instruction:

```
bgez $s0, ispos1 --> bgez $16, 3
```

---

**Test Segment**

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Label</th>
<th>Inst</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400000</td>
<td>0xc0110001</td>
<td>0x10</td>
<td>3. branch</td>
<td>1w 32b, x</td>
</tr>
<tr>
<td>0x0400001</td>
<td>0xc0300000</td>
<td>0x160000</td>
<td>3. main</td>
<td>1w 32b, x</td>
</tr>
<tr>
<td>0x0400002</td>
<td>0xc0110002</td>
<td>0x14</td>
<td>3. main</td>
<td>1w 32b, x</td>
</tr>
<tr>
<td>0x0400003</td>
<td>0xc0300004</td>
<td>0x170000</td>
<td>3. main</td>
<td>1w 32b, x</td>
</tr>
<tr>
<td>0x0400004</td>
<td>0xc0104000</td>
<td>0x16</td>
<td>3. main</td>
<td>1w 32b, x</td>
</tr>
<tr>
<td>0x0400005</td>
<td>0xc0114000</td>
<td>0x17</td>
<td>3. main</td>
<td>1w 32b, x</td>
</tr>
<tr>
<td>0x0400006</td>
<td>0xc0104002</td>
<td>0x16</td>
<td>3. main</td>
<td>1w 32b, x</td>
</tr>
<tr>
<td>0x0400007</td>
<td>0xc0114002</td>
<td>0x17</td>
<td>3. main</td>
<td>1w 32b, x</td>
</tr>
</tbody>
</table>