The binary encoding of an instruction set is called machine code.

1 instruction = 1 word
1 integer = 1 word
1 address = 1 word
1 character = 1 byte

00000100011001000100000000100000

Assembling MIPS instructions

add $t0, $s1, $s2

000000 10001 10010 01000 00000 100000
6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

32 bits or one word

MIPS instruction format (R-type)

Instruction operation or opcode

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td>function code</td>
</tr>
</tbody>
</table>

R = register

Remember those useless numbers we gave each register?
MIPS Register Conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Machine code

addi $t0, $s1, 1024

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>10001</td>
<td>01000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Design Principle 4.
Good design demands good compromises

I-type instruction format

I = immediate  largest immediate value?
Compile all the way to bits.

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Exercise. First compile and then assemble \( a[200] = h + a[300] \), assuming \( t1 \) already holds the base address of \( a \) and \( s2 \) holds \( h \).

*Here’s where that green card in the front of your text comes in handy.*

Dealing with larger constants

\[
lui \ t0, 255 \quad \# \ t0 \ is \ register \ 8
\]

\[
\begin{array}{cccccccc}
001111 & 0000 & 0100 & 0000 & 0000 & 1111 & 1111 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{Register 8} & \\
0000 & 0000 & 1111 & 1111 & 0000 & 0000 & 0000 & 0000 \\
\end{array}
\]

*But what about the lower 16-bits?*

addi \( t0, s1, 16711685 \)

\[
\begin{array}{cccccccc}
\text{op} & \text{rs} & \text{rt} & \text{constant or address} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{op} & \text{rs} & \text{rt} & \text{constant or address} \\
6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 16 \text{ bits} \\
\end{array}
\]

OR immediate to the rescue

\[
lui \ t0, 255 \quad \# \ t0 \ is \ register \ 8
\]

\[
\begin{array}{cccccccc}
001111 & 0000 & 0100 & 0000 & 0000 & 1111 & 1111 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{Register 8} & \\
0000 & 0000 & 1111 & 1111 & 0000 & 0000 & 0000 & 0000 \\
\end{array}
\]

\[
\text{ori} \ t0, t0, 5
\]

\[
\begin{array}{cccccccc}
001101 & 0100 & 0100 & 0000 & 0000 & 0000 & 0000 & 0101 \\
\end{array}
\]

*But what about the lower 16-bits?*
addi $t0, $s1, 16711685

Finally, the newly constructed binary value of 16711685 is added to the contents of $s1 and stored in $t0.

lui $t0, 255    # $t0 is register 8
ori $t0, $t0, 5    # or lower half with upper
add $t0, $t0, $s1    # $t0 <- $s1 + 16711685

The original pseudoinstruction is translated into three actually assembly language instructions at assemble time.

What if low order 16 bits part doesn’t fit in or i7?

Assembler does this for us, uses reserved register $at if needed.

J-type instruction format

<table>
<thead>
<tr>
<th>op</th>
<th>jump address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>goto address 5000</td>
<td></td>
</tr>
</tbody>
</table>

6 bits 26 bits

26-bit destination is concatenated with upper 4 bits of PC plus 2 zeros below.

Conditional branches are more limited

bne $s0, $s1, address

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset sizing</th>
</tr>
</thead>
<tbody>
<tr>
<td>000101</td>
<td>10000</td>
<td>10001</td>
<td>address</td>
</tr>
</tbody>
</table>

6 bits 5 bits 5 bits 16 bits

But they only branch within a single procedure...

PC-relative addressing

bne $t0, $s5, Exit

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset sizing</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>10000</td>
<td>10001</td>
<td>address</td>
</tr>
</tbody>
</table>

6 bits 5 bits 5 bits 16 bits

How big of a loop does this allow? [Trick question.]
**Compare with base addressing**

```plaintext
lw $t0, 12($t1)
```

**Branching offsets in machine language**

```plaintext
while (save[i] == k) {
    i += 1;
    address op  rs  rt  rd  shamt  funct

    Loop:  sll  $t1, $s3, 2  80004 0   9   22   9   0    32
            add  $t1, $t1, $s6  80000 8   8   22   8   0    32
            lw  $t0, 0($t1)  80008 35  9   8        0
            bne $t0, $s5, Exit  80012 5   8   19       2
            addi $s3, $s3, 1  80016 8  19   19
            j  Loop  80020 2              20000
}

Exit:  
```

**5 addressing modes**

- Immediate
- Register
- Base and index
- PC-relative
- Register-base index

**Disassembly!**

```
0x00af8020
```