Reduced Instruction Set Computer (RISC)

- Focuses on reducing the number and complexity of instructions of the machine.

- Reduced number of cycles needed per instruction.
  - Goal: At least one instruction completed per clock cycle.

- Designed with CPU instruction pipelining in mind.

- Fixed-length instruction encoding.

- Load-Store: Only load and store instructions access memory.

- Simplified addressing modes.
  - Usually limited to immediate, register indirect, register displacement, indexed.

- Delayed loads and branches.

- Prefetch and speculative execution.

- Examples: MIPS, UltraSpark, Alpha, PowerPC.
RISC Instruction Set Architecture Example: MIPS R3000

- Memory: Can address $2^{32}$ bytes or $2^{30}$ words (32-bits).
- Instruction Categories:
  - Load/Store.
  - Computational: ALU.
  - Jump and Branch.
  - Floating Point.
    - coprocessor
  - Memory Management.
  - Special.
- 3 Instruction Formats: all 32 bits wide:

<table>
<thead>
<tr>
<th>R-Type</th>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-Type: ALU Load/Store, Branch</td>
<td>OP</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J-Type: Jumps</td>
<td>OP</td>
<td>jump target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Registers

R0 - R31

31 GPRs
R0 = 0

PC
HI
LO
MIPS Memory Addressing & Alignment

- MIPS uses Big Endian operand storage in memory where the most significant byte is in low memory (this is similar to IBM 360/370, Motorola 68k, Sparc, HP PA).

- MIPS requires that all words (32-bit) to start at memory addresses that are multiple of 4.

- In general objects must fall on memory addresses that are multiple of their size.
MIPS Register Usage/Naming Conventions

- In addition to the usual naming of registers by $ followed with register number, registers are also named according to MIPS register usage convention as follows:

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Name</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>Constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>Reserved for assembler</td>
<td>no</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0-$v1</td>
<td>Values for result and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0-$a3</td>
<td>Arguments</td>
<td>yes</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0-$t7</td>
<td>Temporaries</td>
<td>no</td>
</tr>
<tr>
<td>16-23</td>
<td>$s0-$s7</td>
<td>Saved</td>
<td>yes</td>
</tr>
<tr>
<td>24-25</td>
<td>$t8-$t9</td>
<td>More temporaries</td>
<td>no</td>
</tr>
<tr>
<td>26-27</td>
<td>$k0-$k1</td>
<td>Reserved for operating system</td>
<td>yes</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>Global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>Stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>Frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>Return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide

**Register (direct)**

- First Operand
- Second Operand
- Destination

**Immediate**

- First Operand
- Second Operand
- Destination

**Displacement: Base+index**

- First Operand
- Second Operand
- Destination

**PC-relative**

- First Operand
- Second Operand
- Destination
# MIPS Arithmetic Instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; exception possible</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $1,$2,100</td>
<td>$1 = $2 + 100 + constant</td>
<td>+ constant; exception possible</td>
</tr>
<tr>
<td>add unsigned</td>
<td>addu $1,$2,$3</td>
<td>$1 = $2 + $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>subu $1,$2,$3</td>
<td>$1 = $2 – $3</td>
<td>3 operands; no exceptions</td>
</tr>
<tr>
<td>add imm. unsign.</td>
<td>addiu $1,$2,100</td>
<td>$1 = $2 + 100 + constant</td>
<td>+ constant; no exceptions</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>multu$2,$3</td>
<td>Hi, Lo = $2 x $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td>divide</td>
<td>div $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Lo = quotient, Hi = remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>divide unsigned</td>
<td>divu $2,$3</td>
<td>Lo = $2 ÷ $3,</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hi = $2 mod $3</td>
<td></td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi $1</td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo $1</td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>
# MIPS Logic/Shift Instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>and $1,$2,$3</td>
<td>$1 = $2 &amp; $3</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td>or</td>
<td>or $1,$2,$3</td>
<td>$1 = $2</td>
<td>$3</td>
</tr>
<tr>
<td>xor</td>
<td>xor $1,$2,$3</td>
<td>$1 = $2 ⊕ $3</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td>nor</td>
<td>nor $1,$2,$3</td>
<td>$1 = ~($2</td>
<td>$3</td>
</tr>
<tr>
<td>and immediate</td>
<td>andi $1,$2,10</td>
<td>$1 = $2 &amp; 10</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>ori $1,$2,10</td>
<td>$1 = $2</td>
<td>10</td>
</tr>
<tr>
<td>xor immediate</td>
<td>xori $1, $2,10</td>
<td>$1 = ~$2 &amp;~10</td>
<td>Logical XOR reg, constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll $1,$2,10</td>
<td>$1 = $2 &lt;&lt; 10</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>sra $1,$2,10</td>
<td>$1 = $2 &gt;&gt; 10</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sllv $1,$2,$3</td>
<td>$1 = $2 &lt;&lt; $3</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srlv $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td>shift right arithm.</td>
<td>srav $1,$2, $3</td>
<td>$1 = $2 &gt;&gt; $3</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>

**Example**: Example instructions and their meanings are demonstrated in the table above. Each instruction is paired with its corresponding example and meaning, along with a comment to clarify the usage and context.
MIPS data transfer instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw 500($4), $3</td>
<td>Store word</td>
</tr>
<tr>
<td>sh 502($2), $3</td>
<td>Store half</td>
</tr>
<tr>
<td>sb 41($3), $2</td>
<td>Store byte</td>
</tr>
<tr>
<td>lw $1, 30($2)</td>
<td>Load word</td>
</tr>
<tr>
<td>lh $1, 40($3)</td>
<td>Load halfword</td>
</tr>
<tr>
<td>lhu $1, 40($3)</td>
<td>Load halfword unsigned</td>
</tr>
<tr>
<td>lb $1, 40($3)</td>
<td>Load byte</td>
</tr>
<tr>
<td>lbu $1, 40($3)</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>lui $1, 40</td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

```
LUI R5
```

```
R5 0000 ... 0000
```
# MIPS Branch, Compare, Jump Instructions Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch on equal</td>
<td>beq $1,$2,100</td>
<td>if ($1 == $2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Equal test; PC relative branch</td>
</tr>
<tr>
<td>branch on not eq.</td>
<td>bne $1,$2,100</td>
<td>if ($1!=$2) go to PC+4+100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not equal test; PC relative branch</td>
</tr>
<tr>
<td>set on less than</td>
<td>slt $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare less than; 2’s comp.</td>
</tr>
<tr>
<td>set less than imm.</td>
<td>slti $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare &lt; constant; 2’s comp.</td>
</tr>
<tr>
<td>set less than uns.</td>
<td>sltu $1,$2,$3</td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare less than; natural numbers</td>
</tr>
<tr>
<td>set l. t. imm. uns.</td>
<td>sltiu $1,$2,100</td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compare &lt; constant; natural numbers</td>
</tr>
<tr>
<td>jump</td>
<td>j 10000</td>
<td>go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jump to target address</td>
</tr>
<tr>
<td>jump register</td>
<td>jr $31</td>
<td>go to $31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td>jump and link</td>
<td>jal 10000</td>
<td>$31 = PC + 4; go to 10000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Details of The MIPS Instruction Set

• Register zero **always** has the value **zero** (even if you try to write it).
• Branch/jump **and link** put the return addr. PC+4 into the link register (R31).
• All instructions change **all 32 bits** of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
• Immediate arithmetic and logical instructions are extended as follows:
  – logical immediates ops are zero extended to 32 bits.
  – arithmetic immediates ops are sign extended to 32 bits (including addu).
• The data loaded by the instructions lb and lh are extended as follows:
  – lbu, lhu are zero extended.
  – lb, lh are sign extended.
• Overflow can occur in these arithmetic and logical instructions:
  – add, sub, addi
  – it **cannot** occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu
Example: C Assignment To MIPS

- Given the C assignment statement:

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

- Assuming the variables are assigned to MIPS registers as follows:

\[
\begin{align*}
  f & : \$s0, \\
  g & : \$s1, \\
  h & : \$s2, \\
  i & : \$s3, \\
  j & : \$s4
\end{align*}
\]

- MIPS Instructions:

\[
\begin{align*}
  \text{add } \$s0,\$s1,\$s2 & \ # \ \$s0 = g+h  \\
  \text{add } \$t1,\$s3,\$s4 & \ # \ \$t1 = i+j  \\
  \text{sub } \$s0,\$s0,\$t1 & \ # \ f = (g+h)-(i+j)
\end{align*}
\]
Example: C Assignment With Operand In Memory To MIPS

• For the C statement:
  \[ g = h + A[8]; \]
  – Assume the following MIPS register mapping:
    \[ g: \$s1, \quad h: \$s2, \quad \text{base address of } A[ ]: \$s3 \]

• Steps:
  – Add 32 bytes to \$s3 to select \( A[8] \), put into \( \$t0 \)
  – Next add it to \( h \) and place in \( g \)

• MIPS Instructions:
  
  \[
  \text{lw } \$t0, 32(\$s3) \quad \# \, \$t0 \text{ gets } A[8] \\
  \text{add } \$s1, \$s2, \$t0 \quad \# \, \$s1 = h + A[8]
  \]
Example: C Assignment With Variable Index To MIPS

• For the C statement with a variable array index:
  \[ g = h + A[i]; \]

• Assume: \( g: \$s1, \ h: \$s2, \ i: \$s4, \) base address of \( A[\ ]: \$s3 \)

• Steps:
  – Turn index \( i \) to a byte offset by multiplying by four or by addition as done here: \( i + i = 2i, \ 2i + 2i = 4i \)
  – Next add 4i to base address of \( A \)
  – Load \( A[i] \) into a temporary register.
  – Finally add to \( h \) and put sum in \( g \)

• MIPS Instructions:
  
  \[
  \begin{align*}
  \text{add } & \$t1, \$s4, \$s4 \quad \# \ $t1 = 2*i \\
  \text{add } & \$t1, \$t1, \$t1 \quad \# \ $t1 = 4*i \\
  \text{add } & \$t1, \$t1, \$s3 \quad \#$t1 = address of A[i] \\
  \text{lw } & \$t0, 0(\$t1) \quad \# \ $t0 = A[i] \\
  \text{add } & \$s1, \$s2, \$t0 \quad \# \ g = h + A[i]
  \end{align*}
  \]
Example: C If Statement to MIPS

• For The C statement:

```c
if (i == j) f=g+h;
else f=g-h;
```

– Assume the following MIPS register mapping:

```plaintext
f: $s0,  g: $s1,  h: $s2,  i:  $s3,  j:  $s4
```

• Mips Instructions:

```plaintext
beq $s3,$s4, True  # branch if i==j
sub $s0,$s1,$s2     # f = g-h (false)
j Exit
True: add $s0,$s1,$s2  # f = g+h (true)
Exit:
```
Example: Simple C Loop to MIPS

- Simple loop in C:

```
Loop:    g = g + A[i];
         i = i + j;
         if (i != h) goto Loop;
```

- Assume MIPS register mapping:

```
g: $s1,  h: $s2,  i: $s3,  j: $s4,  base of A[ ]: $s5
```

- MIPS Instructions:

```
Loop:    add $t1,$s3,$s3     # $t1= 2*i
         add $t1,$t1,$t1     # $t1= 4*i
         add $t1,$t1,$s5     # $t1=address of A[I]
         lw $t1,0($t1)       # $t1= A[i]
         add $s1,$s1,$t1     # g = g + A[i]
         add $s3,$s3,$s4     # I = i + j
         bne $s3,$s2,Loop    # goto Loop if i!=h
```
Example: C Less Than Test to MIPS

- Given the C statement:
  \[
  \text{if (g < h) go to Less}
  \]

- Assume MIPS register mapping:
  
  \[
  g: \$s0, \quad h: \$s1
  \]

- MIPS Instructions:

  
  \[
  \text{slt } \$t0, \$s0, \$s1 \quad \# \; \$t0 = 1 \text{ if } \$s0 < \$s1 \quad (g < h)
  
  \text{bne } \$t0, \$zero, \text{ Less} \quad \# \; \text{goto Less}
  
  \]

  \[
  \text{Less:}
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]

  
  
  
  
  \[
  \]
Example: While C Loop to MIPS

- While loop in C:
  ```c
  while (save[i]==k)
      i = i + j;
  ```

- Assume MIPS register mapping:
  ```
  i: $s3,    j: $s4,    k: $s5,    base of save[ ]: $s6
  ```

- MIPS Instructions:
  ```
  Loop:  add $t1,$s3,$s3       # $t1 = 2*i
         add $t1,$t1,$t1         # $t1 = 4*i
         add $t1,$t1,$s6         # $t1 = Address
         lw  $t1,0($t1)          # $t1 = save[i]
         bne $t1,$s5,Exit        # goto Exit
                                 # if save[i]!=k
         add $s3,$s3,$s4         # i = i + j
         j   Loop                # goto Loop
  
  Exit:
  ```
Example: C Case Statement To MIPS

- The following is a C case statement called switch:
  
  ```
  switch (k) {
    case 0: f=i+j; break; /* k=0*/
    case 1: f=g+h; break; /* k=1*/
    case 2: f=g–h; break; /* k=2*/
    case 3: f=i–j; break; /* k=3*/
  }
  ```

- Assume MIPS register mapping:
  
  ```
  f: $s0,  g: $s1,  h: $s2,  i: $s3,  j: $s4,  k: $s5
  ```

- Method: Use k to index a jump address table in memory, and then jump via the value loaded.

- Steps:
  - 1st test that k matches one of the cases (0<=k<=3); if not, the code exits.
  - Multiply k by 4 to index table of words.
  - Assume 4 sequential words in memory, base address in $t2, have addresses corresponding to labels L0, L1, L2, L3.
  - Load a register $t1 with jump table entry address.
  - Jump to address in register $t1 using jump register jr $t1.
Example: C Case Statement To MIPS (Continued)

MIPS Instructions:

```
slti $t3,$s5,0  # Test if k < 0
bne $t3,$zero,Exit  # if k < 0, goto Exit
slti $t3,$s5,4  # Test if k < 4
beq $t3,$zero,Exit  # if k >= 4, goto Exit
add $t1,$s5,$s5  # Temp reg $t1 = 2*k
add $t1,$t1,$t1  # Temp reg $t1 = 4*k
add $t1,$t1,$t2  # $t1 = addr JumpTable[k]
lw $t1,0($t1)  # $t1 = JumpTable[k]
jr $t1  # jump based on $t1

L0: add $s0,$s3,$s4  # k=0 so f = i + j
    j Exit  # end case, goto Exit
L1: add $s0,$s1,$s2  # k=1 so f = g + h
    j Exit  # end case, goto Exit
L2: sub $s0,$s1,$s2  # k=2 so f = g - h
    j Exit  # end case, goto Exit
L3: sub $s0,$s3,$s4  # k=3 so f = i - j
    # end of switch statement
    Exit:
```
Example: Single Procedure Call In MIPS

- **C Code:**
  ```c
  ... sum(a,b);... /* a,b:a: $s0, b: $s1 */
  }  
  ...
  int sum(int x, int y) {
    return x+y;
  }
  ```

- **MIPS Instructions:**

  ADDRESS          ADDI    1000 $a0,$s0,$zero # x = a
  1004  add   $a1,$s1,$zero  # y = b
  1008  jal    sum        # $ra=1012, go to sum
  1012  ...
  2000  sum:  add $v0,$a0,$a1
  2004  jr    $ra
C Memory Allocation Seen By MIPS Programs

- **Stack**: Space for saved procedure information
  - $sp$
  - stack pointer

- **Heap**: Explicitly created space, e.g., malloc(); C pointers
  - global pointer
  - $gp$

- **Static**: Variables declared once per program

- **Code**: Program

- **0**: Program start
Example: Nested Procedure Call In MIPS

- **C Code:**

  ```c
  int sumSquare(int x, int y) {
    return mult(x,x)+ y;
  }
  ```

- **MIPS Code:**

  ```mips
  sumSquare:
  subi $sp,$sp,12 # space on stack
  sw  $ra,$ 8($sp) # save return address
  sw  $a0,$ 0($sp) # save x
  sw  $a1,$ 4($sp) # save y
  addi $a1,$a0,$zero # mult(x,x)
  jal  mult # call mult
  lw   $ra,$ 8($sp) # get return address
  lw   $a0,$ 0($sp) # restore x
  lw   $a1,$ 4($sp) # restore y
  add  $vo,$v0,$a1 # mult()+ y
  addi $sp,$sp,12 # => stack space
  jr   $ra
  ```
### MIPS R-Type (ALU) Instruction Fields

**R-Type:** All ALU instructions that use three registers

<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>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **op:** Opcode, basic operation of the instruction.
  - For R-Type \( \text{op} = 0 \)
- **rs:** The first register source operand.
- **rt:** The second register source operand.
- **rd:** The register destination operand.
- **shamt:** Shift amount used in constant shift operations.
- **funct:** Function, selects the specific variant of operation in the op field.

Examples:
- `add $1,$2,$3`
- `sub $1,$2,$3`
- `and $1,$2,$3`
- `or $1,$2,$3`
MIPS ALU I-Type Instruction Fields

I-Type ALU instructions that use two registers and an immediate value
Loads/stores, conditional branches.

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

• op: Opcode, operation of the instruction.
• rs: The register source operand.
• rt: The result destination register.
• immediate: Constant second operand for ALU instruction.

Examples:
- add immediate: addi $1,$2,100
- and immediate andi $1,$2,10

Result register in rt
Source operand register in rs
Constant operand in immediate
MIPS Load/Store I-Type Instruction Fields

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>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>

- **op**: Opcode, operation of the instruction.
  - For load op = 35, for store op = 43.
- **rs**: The register containing memory base address.
- **rt**: For loads, the destination register. For stores, the source register of value to be stored.
- **address**: 16-bit memory address offset in bytes added to base register.

Examples:
- **Store word**: `sw 500($4), $3`
- **Load word**: `lw $1, 30($2)`
**MIPS Branch I-Type Instruction Fields**

<table>
<thead>
<tr>
<th>OP</th>
<th>rs</th>
<th>rt</th>
<th>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>

- **op**: Opcode, operation of the instruction.
- **rs**: The first register being compared
- **rt**: The second register being compared.
- **address**: 16-bit memory address branch target offset in words added to PC to form branch address.

**Examples:**

- Branch on equal: `beq $1,$2,100`
- Branch on not equal: `bne $1,$2,100`
# MIPS J-Type Instruction Fields

J-Type: Include jump j, jump and link jal

<table>
<thead>
<tr>
<th>OP</th>
<th>jump target</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- **op**: Opcode, operation of the instruction.
  - Jump j op = 2
  - Jump and link jal op = 3
- **jump target**: Jump memory address in words.

Jump memory address in bytes equal to instruction field jump target x 4

**Examples:**
- Branch on equal: j 10000
- Branch on not equal: jal 10000