Machine code (encoding)
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
- Register numbers
  - $t0 – t7$ are reg’s $8 – 15$
  - $t8 – t9$ are reg’s $24 – 25$
  - $s0 – s7$ are reg’s $16 – 23$

MIPS R-format Instructions
- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example
- add $t0, $s1, $s2
  - op
  0
  000000
  - rs $s1
  17
  10001
  - rt $s2
  18
  10010
  - rd $t0
  8
  01000
  - shamt 0
  0
  00000
  - funct add
  32
  10000
  - 32-bit code
  00000100011001001000000000100000 = 02324020$_{16}$

MIPS I-format Instructions
- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: $-2^{15}$ to $2^{15} – 1$
  - Address: offset added to base address in rs
Logical Operations

- Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

Shift Operations

- shamt: number of positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - sll by i bits multiplies by $2^i$
- Shift right logical
  - Shift right and fill with 0 bits
  - srl by i bits divides by $2^i$ (unsigned only)

AND Operations

- and $t0$, $t1$, $t2$

```
$t2$ 0000 0000 0000 0000 0000 1101 1100 0000
$t1$ 0000 0000 0000 0000 0011 1100 0000 0000
$t0$ 0000 0000 0000 0000 0000 1100 0000 0000
```

OR Operations

- or $t0$, $t1$, $t2$

```
$t2$ 0000 0000 0000 0000 0000 1101 1100 0000
$t1$ 0000 0000 0000 0000 0011 1100 0000 0000
$t0$ 0000 0000 0000 0000 0011 1101 1100 0000
```
### NOT Operations

- MIPS has NOR 3-operand instruction
  - \( \text{a NOR b} = \text{NOT (a OR b)} \)
- \( \text{nor } $t0, $t1, $zero \)

<table>
<thead>
<tr>
<th>$t1</th>
<th>0000 0000 0000 0000 0011 1100 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0</td>
<td>1111 1111 1111 1111 1100 0011 1111 1111</td>
</tr>
</tbody>
</table>

### Conditional Operations

- Branch to a labeled instruction if a condition is true, else, continue sequentially
  - \( \text{beq rs, rt, L1} \)
    - if \((rs == rt)\) branch to instruction labeled \(L1\)
  - \( \text{bne rs, rt, L1} \)
    - if \((rs != rt)\) branch to instruction labeled \(L1\)
  - \( \text{j L1} \)
    - unconditional jump to instruction labeled \(L1\)

### Compiling If Statements

- C code:
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```
  - \(f, g, \ldots)\) in \($s0, $s1, \ldots\)
- Compiled MIPS code:
  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

### Compiling Loop Statements

- C code:
  ```c
  while (save[i] == k) i += 1;
  ```
  - \(i\) in \($s3\), \(k\) in \($s5\), address of \(\text{save}\) in \($s6\)
- Compiled MIPS code:
  ```
  Loop: sll $t1, $s3, 2
  add $t1, $t1, $s6
  lw $t0, 0($t1)
  bne $t0, $s5, Exit
  addi $s3, $s3, 1
  j Loop
  
  Exit: ...
  ```
### More Conditional Operations
- Set result to 1 if a condition is true
- Otherwise, set to 0
- `slt` rd, rs, rt
  - if (rs < rt) rd = 1; else rd = 0;
- `slti` rt, rs, constant
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with `beq`, `bne`
  
  ```
  slt $t0, $s1, $s2  # if ($s1 < $s2)  
bne $t0, $zero, L  # branch to L
  ```

### Branch Instruction Design
- Why not `blt`, `bge`, etc?
- Hardware for `<`, `≥` ... slower than `=`, `≠`
- Combining with branch involves more work per instruction, requiring a slower clock
- All instructions penalized!
- `beq` and `bne` are the common case

### Signed vs. Unsigned
- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  
  ```
  $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  slt $t0, $s0, $s1  # signed  
  -1 < +1 ⇒ $t0 = 1
  sltu $t0, $s0, $s1  # unsigned  
  +4,294,967,295 > +1 ⇒ $t0 = 0
  ```

### Byte/Halfword Operations
- Could use bitwise operations
- MIPS byte/halfword load/store
  
  ```
  lb rt, offset(rs)  
lh rt, offset(rs)  
  lbu rt, offset(rs)  
lhu rt, offset(rs)  
  sb rt, offset(rs)  
  sh rt, offset(rs)  
  ```
- Sign extend to 32 bits in rt
- Zero extend to 32 bits in rt
- Store just rightmost byte/halfword
String Copy Example

- **C code (naïve):**
  - Null-terminated string
  ```c
  void strcpy (char x[], char y[]) {
    int i;
    i = 0;
    while ((x[i]=y[i])!='\0')
      i += 1;
  }
  ```
  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- **MIPS code:**
  ```mips
  strcpy:
  addi $sp, $sp, -4      # adjust stack for 1 item
  sw $s0, 0($sp)         # save $s0
  addi $s0, $zero, 1     # i = 0
  L1: add $t1, $s0, $a1   # addr of y[i] in $t1
      lbu $t2, 0($t1)     # $t2 = y[i]
      add $t3, $s0, $a0   # addr of x[i] in $t3
      sb $t2, 0($t3)      # x[i] = y[i]
      beq $t2, $zero, L2   # exit loop if y[i] == 0
      addi $s0, $s0, 1     # i = i + 1
      j L1                   # next iteration of loop
  L2: lw $s0, 0($sp)      # restore saved $s0
      addi $sp, $sp, 4     # pop 1 item from stack
      jr $ra                 # and return
  ```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  - **lui rt, constant**
    - Copies 16-bit constant to left 16 bits of rt
    - Clears right 16 bits of rt to 0
  ```mips
  lui $s0, 61
  ```

- **ori $s0, $s0, 2304**
  ```mips
  ```

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
  ```mips
  ```

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
  ```mips
  ```
Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>20 bits</td>
</tr>
</tbody>
</table>

- (Pseudo)Direct jump addressing
  - Target address = PC_{31...28} : (address × 4)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

| Loop: s11 | $t1, $s3, 2 | 80000 | 0 0 19 | 9 4 0 |
| add | $t1, $t1, $s6 | 80004 | 0 9 22 | 9 0 32 |
| lw | $t0, 0($t1) | 80008 | 35 9 8 | 0 0 |
| bne | $t0, $s5, Exit | 80102 | 5 8 21 | 2 2 |
| addi | $s3, $s3, 1 | 80016 | 8 19 19 | 1 1 |
| j | Loop | 80020 | 0 0 0 0 |
| Exit: | ... | 80024 | 2 20000 |

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code

  ```
  beq $s0,$s1, L1
  bne $s0,$s1, L2
  j L1
  L2: ...
  ```

Addressing Mode Summary