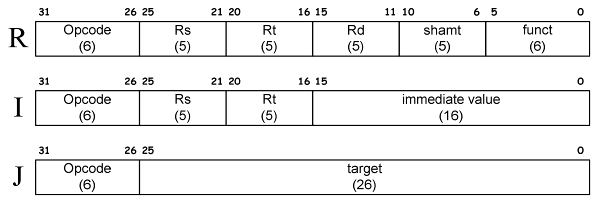
## MIPS32 AL – Instruction Decoding + Extras
(machine language → assembly language)

- For each 32 bits (set of 32 0's and 1's):
  - Use opcode to find out which of R, I, and J formats is involved
    - **0** → **R**-format
    - **2 or 3** → **J**-format
    - **Other** → **I**-format
  - Use instruction format to determine which fields exist



  - Write out MIPS assembly code
    - Convert each field to name, register number/name, or decimal/hex number

## MIPS32 AL – Instruction Decoding + Extras
(decoding example)

- Given:
  - 6 machine instructions in hex:

```
00001025
0005402A
11000003
00441020
20A5FFFF
08100001
```

  - Address of 1st instruction:

```
4194304 (0x00400000)
```

## MIPS32 AL – Instruction Decoding + Extras
(decoding example)

- Convert instructions in hex to *instructions in binary*:

```
00000000000000000001000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000100010000010000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

## MIPS32 AL – Instruction Decoding + Extras
(decoding example)

- Find out opcodes and instruction formats:

```
R 000000 00000 00000 00010 00000 100101
R 000000 00000 00101 01000 00000 101010
I 000100 01000 00000 0000000000000011
R 000000 00010 00100 00010 00000 100000
I 001000 00101 00101 1111111111111111
J 000010 00000100000000000000000001
```

| | | | | | | |
|---|---|---|---|---|---|---|
| R | 0 | Rs | Rt | Rd | shamt | funct |
| I | 1, 4-62 | Rs | Rt | immediate | | |
| J | 2 or 3 | target address | | | | |

## MIPS32 AL – Instruction Decoding + Extras
(decoding example)

- Separate fields based on format/opcode:

| | | | | | | |
|---|---|---|---|---|---|---|
| R | 0 | 0 | 0 | 2 | 0 | 37 |
| R | 0 | 0 | 5 | 8 | 0 | 42 |
| I | 4 | 8 | 0 | +3 | | |
| R | 0 | 2 | 4 | 2 | 0 | 32 |
| I | 8 | 5 | 5 | -1 | | |
| J | 2 | 1048577 (0x100001) | | | | |

## MIPS32 AL – Instruction Decoding + Extras
(decoding example)

- Translate (disassemble) into assembly code:

| | *Address* | *Assembly Instruction* |
|---|---|---|
| given | `0x00400000` | `or $2, $0, $0` |
| | `0x00400004` | `slt $8, $0, $5` |
| | `0x00400008` | `beq $8, $0, 3` |
| | `0x0040000c` | `add $2, $2, $4` |
| | `0x00400010` | `addi $5, $5, -1` |
| | `0x00400014` | `j 0x100001` |

## MIPS32 AL – Instruction Decoding + Extras
(decoding example)

- Use *addressing mode semantics* to "decrypticize" code:

| *Address* | *Assembly Instruction* |
|---|---|
| `0x00400000` | `or $2, $0, $0` |
| `0x00400004` | `slt $8, $0, $5` |
| `0x00400008` | `beq $8, $0, 3` |
| `0x0040000c` | `add $2, $2, $4` |
| `0x00400010` | `addi $5, $5, -1` |
| `0x00400014` | `j 0x100001` |

## MIPS32 AL – Instruction Decoding + Extras
(decoding example)

- Modify code to bring "decrypticization" to bear:
  - Replace branch/jump destinations with labels
  - Replace register numbers with register names

```
            or $v0, $0, $0
J1:         slt $t0, $0, $a1
            beq $t0, $0, B1
            add $v0, $v0, $a0
            addi $a1, $a1, -1
            j J1
B1:
```

## MIPS32 AL – Instruction Decoding + Extras
(decoding example – mystery revealed)

- Actual C code:

```
product = 0;
while (multiplier > 0)
{
   product += multiplicand;
   multiplier -= 1;
}
```

**`$v0`**: product
**`$a0`**: multiplicand
**`$a1`**: multiplier

```
            or $v0, $0, $0
J1:         slt $t0, $0, $a1
            beq $t0, $0, B1
            add $v0, $v0, $a0
            addi $a1, $a1, -1
            j J1
B1:
```

## MIPS32 AL – Instruction Decoding + Extras

Extras

Extras

Extras

## MIPS32 AL – Instruction Decoding + Extras
(how to put arbitrary 32-bit value in register)

- **lui** to the rescue:

```
lui Rt, imm
```

- Example:
  - To load **$a0** with hex value **0xABABCDCD**, do as follows:

```
lui     $a0, 0xABAB
ori     $a0, $a0, 0xCDCD
```

Will **addi** work also?

## MIPS32 AL – Instruction Decoding + Extras
(**lui** used by assembler where needed)

- Example 1:
  - The statement:

```
addi $t0, $t0, 0xABABCDCD
```

is translated by assembler into

```
lui $at, 0xABAB
ori $at, $at, 0xCDCD
add $t0, $t0, $at
```

NOTE: Above statement is therefore a *pseudoinstruction*

## MIPS32 AL – Instruction Decoding + Extras (`lui` used by assembler where needed)

- Example 1a:
  - In general, the statement:

```
addi $t1, $t2, imm_value
```

is translated by assembler into

```
lui $at, upper 16 bits of imm_value
ori $at, $at, lower 16 bits of imm_value
add $t1, $t2, $at
```

*if* **`imm_value`** *cannot fit into 16 bits*

NOTE: So **`addi`** *may be* a pseudoinstruction

## MIPS32 AL – Instruction Decoding + Extras (`lui` used by assembler where needed)

- Example 2a:
  - In general, the statement:

```
lw $t1, displ($t0)
```

is translated by assembler into

```
lui $at, upper 16 bits of displ
ori $at, $at, lower 16 bits of displ
add $t0, $t0, $at
lw $t1, 0($t0)
```

*if* **`displ`** *cannot fit into 16 bits*

NOTE: So **`lw`** *may be* a pseudoinstruction also

## MIPS32 AL – Instruction Decoding + Extras (`lui` used by assembler where needed)

- Example 2b:
  - In general, the statement:

```
sw $t1, displ($t0)
```

is translated by assembler into

```
lui $at, upper 16 bits of displ
ori $at, $at, lower 16 bits of displ
add $t0, $t0, $at
sw $t1, 0($t0)
```

*if* **`displ`** *cannot fit into 16 bits*

NOTE: So **`sw`** *may be* a pseudoinstruction also

## MIPS32 AL – Instruction Decoding + Extras (`lui` used by assembler where needed)

- Example 3:
  - The statement (pseudoinstruction):

```
la $t0, label
```

is translated by assembler into

```
addi $t0, $0, label_value
```

if **`label_value`** can fit into 16 bits, otherwise it's replaced by

```
lui   $t0, upper 16 bits of label_value
ori   $t0, $t0, lower 16 bits of label_value
```

## MIPS32 AL – Instruction Decoding + Extras (`lui` used by assembler where needed)

- Example 4:
  - The statement (pseudoinstruction):

```
li $t0, imm_value
```

is translated by assembler into

```
ori $t0, $0, imm_value
```

if **`imm_value`** is *positive constant less than 32,768* (= $2^{15}$)

```
lui  $at, upper 16 bits of imm_value
ori  $t0, $at, lower 16 bits of imm_value
```

if **`imm_value`** is *negative constant* or *positive constant greater than 32,767*

## MIPS32 AL – Instruction Decoding + Extras (when one must branch very far)

- From empirical studies of real programs:
  - Most branches go to targets less than 32,767 instructions away
    - Branches used mostly in loops and conditionals
    - Programmers taught to make bodies of loops/conditionals short
- If one still must branch further, one can simulate

```
                  beq $s0, $s1, far
```

with

```
                  bne $s0, $s1, next
                  j far
         next:
```

## MIPS32 AL – Instruction Decoding + Extras

## (some pseudoinstructions of note)

- The statement

```
move $t2, $t1
```

can be expanded into

```
add $t2, $t1, $0
```

- The "no operation" statement

```
nop
```

can be expanded into

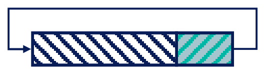
```
sll $0, $0, 0
```

## MIPS32 AL – Instruction Decoding + Extras

## (some pseudoinstructions of note)

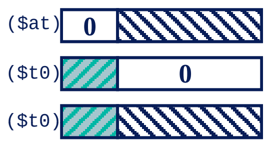
- The statement

```
ror $t0, $t0, imm
```



can be expanded into

```
srl $at, $t0, imm
sll $t0, $t0, 32 - imm
or $t0, $t0, $at
```



- Do it yourself for

```
rol $t0, $t0, imm
```