# MIPS

- Mircroprocessor w/o Interlocked Pipelin Stages
  - A type of RISC (Reduced Instruction Set Computer) as opposite to the CISC such as the X86 micro-archaetecture.
  - RISC philosophy: Simple yet fast instructions, complecated stuff should be done by the assembler.
  - CISC philosophy: Complex instrutions that can do complex things.

## Mips Instructions

- Instructions are written in the format `<instr> <parameter>`
- Each instruction takes its own line
- All instructions are 32 bits long. (that is, 4 bytes)
- Instruction address are measured in bytes, start point is address 0
  - **Thus, all instruction addresses are divisible by 4!**

# Assembly Language Instructions

## Converting from Assemby to Machine Code

Consider the operation `$t3 = $t1 + $t2`, which has assembly equivalent `add $t3, $t1, $t2`. Then, we can convert this into the equivalent machine code.

```
| Opcode |         Operand         |Function|
| 000000 | 01001 01010 01011 XXXXX | 100000 |
```

Notice that here the opcode is 6'b0, since this operation is a R-type operation. The XXXXX in the syntax part is marks don't care values for the `<shamt>`

## R-type VS i-type arithmetic

In general, some intructions are R-type (meaning all operands are registers) and some are I-type (meaning they use an immediate/constant value in their operation). Here are some examples for each flavour

- R-Type: `add, addu, div, divu, mult, multu, sub, subu...`
- I-Type: `addi, addiu...`

## Converting From Assembly to Machine Code, Take II

We should do an example where we use a I-type instruction. Consider `$t2 = $t1 + 42`, which shall translate into `addi $t2, $t1, 42`. Notice that here the actual operation that was carried out is `$t = $s + SignExtend(i = 42)`. The equivalent machine code is

```
| Opcode |   Operand   |     Immediate    |
| 001000 | 01001 01010 | 0000000000101010 |
```

## Shift Instructions

Here is a list of common shift instructions

| Insturction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| `sll` | 000000 | `$d, $t, $s` | `$d = $t << a` |
| `sllv` | 000100 | `$d, $t, $s` | `$d = $t << $s` |
| `sra` | 000011 | `$d, $t, a` | `$d = $t >> a` |
| `srav` | 000111 | `$d, $t, $s` | `$d = $t >> $s` |
| `srl` | 000010 | `$d, $t, a` | `$d = $s >>> a` |
| `srlv` | 000110 | `$d, $t, $s` | `$d = $t >>> $s` |

**Note:**

- `srl` = "shift right logical"
- `sra` = "shift right arithmetic"
- `v` denotes a variable number of bits, specified by `$s`
- `a` is the shift amount, and is stored in the `shamt` when encoding the R-type machine code instructions

## Data Movement Instructions

| Insturction | Opecode/Functions | Syntax | Operation |
|---|---|---|---|
| `mfhi` | 010000 | `$d` | `$d = hi` |
| `mflo` | 010010 | `$d` | `$d = lo` |
| `mthi` | 010001 | `$s` | `hi = $s` |
| `mtlo` | 010011 | `$s` | `lo = $s` |

These are R-type instructions for operating on the HI and LO registers described earlier.

## ALU instructions

For the ALU instructions, most are R-type instructions. Thus, they mostly have the 6'b0 opcodes. **However, not all R-type instructions have I-type equivalent**. This is due to the philosophy of the RISC archaetecture, which states that an operation doesn't need an instruction it it can be performed through multiple existing operations.

## Converting Programs and Assembly - Fibonacci Sequence Example

```
int fib(void) {
    int n = 10;
    int f1 = 1, f2 = -1;

    while (n != 0) {
        f1 = f1 + f2;
        f2 = f1 - f2;
        n = n - 1;
    }
    return f1;
}
```

In assembly code

```
# fib.asm
# register usage: $t3 = n, $t4 = f1, $t5 = f2

FIB:    addi $t3, $zero, 10     # Init to n = 10
            addi $t4, $zero, 1      # Init to f1 = 1
            addi $t5, $zero, -1     # Init to f2 = -1
LOOP:   beq $t3, $zero, END     # Until n = 0, then jump to END
            add $t4, $t4, $t5       # f1 = f1 + f2
            sub $t5, $t4, $t5       # f2 = f1 - f2
            addi $t3, $t3, -1       # n = n - 1
            j LOOP                  # REPEAT
END:    sb $t4, 0($sp)          # store the result
```

# Control Flow in Assembly

- Not all programs follow a linear set of intructions.
    - Some operations require the code to brach to one section of code or another, e.g. IF/ELSE calls
    - Some require the code to jump back and repeat a section of code again (FOR/WHILE)
- For this, we have labels on the left hand side that indicate the points that the program flow might need to jump to.
    - Reference to these points the assembly code are resolved at compile time to offset values for the program counter.

## Jump Instructions

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| `j` | 000010 | `label` | `pc = (pc & 0xF00000000) \| (i << 2)` |
| `jal` | 000011 | `label` | `$31 = pc + 4; pc = (pc & 0xF00000000) \| (i << 2)` |
| `jalr` | 001001 | `$s` | `$31 = pc + 4; pc = $s` |
| `jr` | 001000 | `$s` | `pc = $s` |

- `jal` = "jump and link"
  - Register `$31` (aka `$ra`) stores the address that is used when returning from a subroutine
- **Note:** `jr` and `jalr` are jumps, but not J-type instructions.

## More on `jr` and `jalr` (Jump to Registers)

- For instructions such as `jr $ra` and `jalr $t0`.
- The processor moves the address stored in `$ra` and `$t0` into the program counter.
  - The next instruction to be fetched will be at this new address, and the program will continue from there.

## More on `j` and `jal` (Jump to Label)

- For `j` and `jal` instructions, the address is supplied by the instruction, and this can potentially cause a problem.
  - Problem:

    ```
    | 6'b opcode | 26'b address |
    ```

    If we have 32'b instructions and the first 6'b are occupied by the opcode, the remaining bits ***aren't*** enough for a entire address!
- **Work Around:**
  - Trailing Zeros: Since jump instructions load new addresses into the program counter, the values being loaded must be diisibe by 4.
    - Therefore, the binary values of these addresses will always end in "00".
    - Therefore, we are 100% sure that the last two bits are zero!
    - This, combined with the 26'b provided as input gives in total 28'b which is still not enough.
  - First Four bits: Several solutions exsits, we will dicuss the solution that MIPS adapts.
    - We simply keep the first four bits of the previous PC value, and this gave rise to the update rule we saw earlier.

    ```
    pc = (pc & 0xF00000000) | (i << 2)
    ```

    - Notice that the bitwise and part preserves first four bits of the previous program counter.
    - The last part adds to zeros
    - Bitwise OR them together produces the desired result.

## Branch Instructions

Branching statements are used widely in IF statements and WHILE loops, or generally anything that requires the assembly code to jump around. **Note:** the labels are memory locations, assigned to each label at compile time.

| Instruction | Opcode/Function | Syntax | Operation |
|---|---|---|---|
| `beq` | 000100 | `$s, $t, label` | `if ($s == $t) pc += i<< 2` |
| `bgtz` | 000111 | `$s, label` | `if ($s > 0) pc += i << 2` |
| `blez` | 000110 | `$s, label` | `if ($s <= 0) pc += i << 2` |
| `bne` | 000101 | `$s, $t, label` | `if ($s != $t) oc += i << 2` |

### Branch's Immediate (i) Value

- Branch statements are I-type instructions
- The immediate value (i) is a 16bit offset to add to the current instruction if the brach condition is satisfied (not the absolute address like with jumps)
    - Calculated as the difference between the current PC value and the address of the instruction youa re branchng to.
    - Stored here as number of instructions and not number of bytes
        - Again, not storing the trailing '00' if it is not necessary.
    - The i value can be positive (if you are jumping i instructions forward) or negative (if you are jumping i instructions backward)
    -