# Lecture 11
## Instruction Set Architectures,
## MIPS Instruction Formats & Addressing Modes

**Basic Instruction Set Architectures (ISA)**

- There is a number of different ISA. We have seen two so far.

  - *Accumulator (single register) (JVN Machine)*
    - 1 address    add A    acc ← acc + mem [A]
    - 1 address + x    add x A    acc ← acc + mem [A+x]

  - *Stack*
    - 0 address    add    tos ← tos + next    (tos = 'top of stack')

  - *General Purpose Register*
    - 1 address    add A Rb    mem[A] ← mem[A] + Rb
    - 2 address    add A B Rc    mem[A] ← mem[B] + Rc

  - *Load/Store (MIPS Machine)*
    - Load Ra Rb    Ra ← mem[Rb]
    - Store Ra Rb    mem[Rb] ← Ra

  - *Memory to Memory*
    - All operands and destinations can be memory addresses

- Each ISA will result in a number of different instructions to perform an operation.

  - Ex: C = A + B

| **Stack** | **Accumulator** | **Register (register-memory)** | **Register (load/store)** |
|---|---|---|---|
| Push A | Load A | Load R1, A | Load R1, A |
| Push B | Add B | Add R1, B | Load R2, B |
| Add | Store C | Store C, R1 | Add R3, R1, R2 |
| Pop C | | | Store C, R3 |

- RISC machines (like MIPS) have only load/store instructions that access memory. So they are also called load-store machines.

- CISC machines may even have memory-memory instructions, like mem[C] = mem[A] + mem[B]

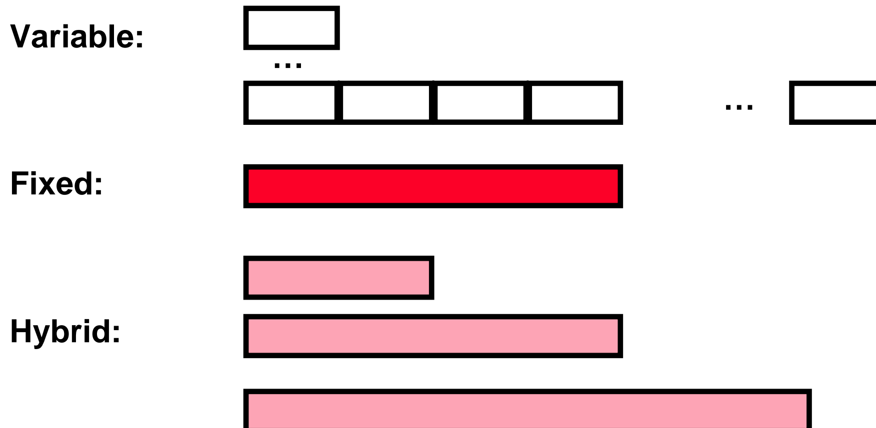
## Addressing Modes

- Every word in memory has an address, similar to the index of an array.
- Early computers numbered words like *C* numbers elements in an array:
    - Memory[0], Memory[1], Memory[2], …
- Today machines address memory as bytes, hence word addresses differ by 4.
    - Memory[0], Memory[4], Memory[8], …
        - 0, 4, 8 is called the address of a word
    - Computers needed to access 8-bit bytes as well as words (4 bytes/word). Called 'byte addressing'.

| Addressing Mode | Example | Meani ng |
|---|---|---|
| Register | Add R4,R3 | R4 ← R4+R3 |
| Immediate | Add R4,#3 | R4 ← R4+3 |
| Displacement | Add R4,100(R1) | R4 ← R4+Mem[100+R1] |
| Register indirect | Add R4,(R1) | R4 ← R4+Mem[R1] |
| Direct or absolute | Add R1,(1001) | R1 ← R1+Mem[1001] |

## Generic Examples of Instruction Formats

- Depending on the hardware architecture and ISA, there are varying types of instruction formats.
- Three types: *variable*, *fixed* and *hybrid.*




- *Variable* instruction format means that each instruction can have a different length.
- *Fixed* instruction format means that all instructions are of the exact same length.
- *Hybrid* instruction format means that classes of instructions have the same instruction length, but different classes can have different lengths.

- If code size is most important, use variable length instructions:
    (1) Difficult control design to compute next address.
    (2) complex operations, so use microprogramming.
    (3) Slow due to several memory accesses.

- If performance is most important, use fixed length instructions:
    (1) Simple to decode, so use hardware
    (2) Wastes code space because of simple operations
    (3) Works well with pipelining

- Little has changed in terms of the typical operations required by our machines. Below is the top 10 80x86 instructions.

| ° Rank | instruction | Integer Average Percent total executed |
|---|---|---|
| 1 | load | 22% |
| 2 | conditional branch | 20% |
| 3 | compare | 16% |
| 4 | store | 12% |
| 5 | add | 8% |
| 6 | and | 6% |
| 7 | sub | 5% |
| 8 | move register-register | 4% |
| 9 | call | 1% |
| 10 | return | 1% |
| | Total | 96% |

° **Simple instructions dominate instruction frequency**

- While we deal with high-level languages, and can talk about complicated addressing modes and instructions, the ones that are actually used in programs are the simple ones => RISC philosophy.

## MIPS Instruction Set Design

- Use general purpose registers with a load-store architecture.

- Provides at 32 general purpose registers plus 32 separate floating-point registers.

- Supports basic addressing modes:
    - displacement (with an address offset size of 16 bits)
    - immediate (16 bits)

- All addressing modes apply to all data transfer instructions.

- Supports these data sizes and types: 8-bit, 16-bit, 32-bit integers; and 32-bit and 64-bit IEEE 754 floating point numbers.

- Supports these simple instructions, since they will dominate the number of instructions executed: *load*, *store*, *add*, *subtract*, *move* register-register, *and*, *shift*, *compare equal*, *compare not equal*, *branch* (with a PC-relative address 16-bits long), *jump*, *call*, and *return*.

- Aims at a minimalist instruction set.

## MIPS Instruction Formats

- MIPS instructions are a fixed length of 32 bits. The 32 bits are split into fields.
- There are 3 types of instruction formats in MIPS: Register, Immediate, Jump

| | | | | | | |
|---|---|---|---|---|---|---|
| R | op | rs | rt | rd | shamt | funct |
| I | op | rs | rt | 16 bit address | | |
| J | op | 26 bit address | | | | |

- ***Register Format*** (R-type)
  - Used for arithmetic and logical AND, OR, shifts
  - Fields have names:

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- op: basic operation of instruction, "opcode"
  - 91 opcodes in MIPS
  - 26 FP opcodes for single and double precision.
    - All FP have 0x11 in opcode field. The actual opcode is in funct field
  - Compared to a CISC machine: 250 opcodes, 8 addressing modes
  - Compared to a JVM Machine: ~250 opcodes
- rs: 1st register source operand.
- rt: 2nd register source operand.
- rd: register destination operand, gets the result.
- shamt: shift amount (we will see how this is used later; assume it is 0 for now).
- funct: function; selects the specific variant of the operation in the op field; sometimes called the *function code*.
  - 6 bits for the opcode means that there are 64 opcodes. However this is not enough, therefore the funct field is used. If the op code is 0 or 1, then the true instruction is determined by either the rt or funct field.
  - If 7 or 8 bits were used for the opcode, we could fit all the opcodes, but then we have to shorten all the remaining fields. This way we only use extra bits when we need them.

- ***Immediate Format*** (I-type)

  - Used for immediate instructions, data transfers, branches, and logical operations .

  - Allows for small operand values (signed, range $\pm 2^{15}$, 1 bit for sign) to be stored in the instruction itself.

  - Avoids a memory access.

  - Fields have names:

\

- op: basic operation of instruction, "opcode".

- rs: 1st register source operand.

- rt: destination operand.
  - To make it easier for the hardware, the first 3 fields are the same in R and I format. However that means that the rt field changes meaning. The hardware knows which format is which based on the distinct values in the opcode field.

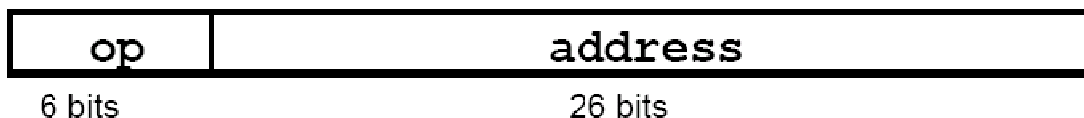
- ***Jump Format*** (J-type)

  - Used for jump instructions.

  - 26 bits is enough to store a word address. Multiply it by 4 to get the byte address [26 bits ⟶ 28 bits].

  - These bits are placed in the lower 28 bits of PC.

  - Fields have names:




  - Since leftmost 4 bits of PC do not change, this kind of addressing is effective/correct if the program is placed within a block of $2^{28}$ = 256 MB properly.

  - If you want to jump beyond that, then use jump register instruction (`jr`).
    - o Place the address in a register before jumping ⟶ full 32 bits.

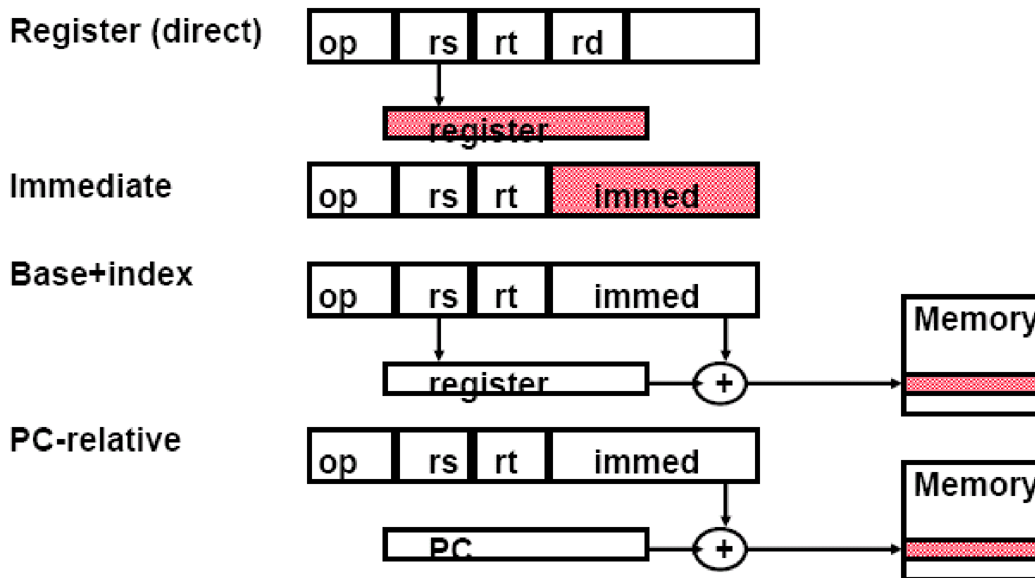
  - When we use 26 bit field the addressing mode is know as *pseudo-direct* (it is not 'direct' because 4 bits are from the existing PC value).

**MIPS Addressing**

- How are the addresses of the operands specified?
    - 3 operand instructions would require 32*3 = 96 bits (just for the operands). This is too long and not desirable.
    - Registers make things easy, only a few bits are necessary to specify a register.
    - When operand is present in register/or when destination is a register this is called *register addressing*.

*MIPS Addressing Modes*

- Depending on the instruction and the instruction format, the memory location or the address/data to be retrieved can vary.
- We have already seen the *pseudo-direct* addressing mode (in the context of the J-type instruction format used for jump instructions).
- There are four more types of addressing modes: *Register*, *Immediate*, *Base+index* and *PC-relative.*




- *Base+index Mode*
    - When a register is specified inside a pair of parenthesis, then its contents are to be treated as an address.
    - Ex: `lw $t0,c($t1)` #load the word at the memory address specified in register $t1 into $t0.
    - An offset, *c*, can be used. It is a signed 16 bit number (range $\pm 2^{15}$). This is in byte quantity.
    - The address in the register is known as the *base address.*

- *PC-Relative*
    - PC-relative is used to specify branch/jump instructions.
    - This is a better approach than storing the absolute address in the instruction. We don't have enough space in our 32 bit instruction.
    - $PC_{branch}$ + 4 + (offset*4) is the target address, where $PC_{branch}$ is the address of the branch instruction (*i.e.*, the value of the PC when we start to *Fetch-Decode-Execute* cycle for the branch instruction). By having the offset in words, we increase the range specified by the offset by 4 times.
    - Conditional branch instructions use this format.
    - Ex:

```
Loop:   add $t1, $s3, $s3
        …..
        bne $t0, $s5, Exit
        add $s3, $s4, $s5
        j Loop
Exit:   ….
```

- `Exit` label is used in the instruction `bne`. What is the offset here? 3
- Why? From the address of `bne` the location `Exit` is 3 instructions/words further down.
- Since the PC has been incremented before we execute `bne`, it is only 2 words or 8 bytes.
- Thus, the assembled instruction for `bne` should look like:
    - opcode: 5, rs: 8, rt: 21, offset: 2
- Target address = address of `Exit` = ($PC_{branch}$ + 4) + 2*4 = $PC_{updated}$ + 2*4, where $PC_{updated}$ is the value of the incremented PC.
- If we store the offset in bytes the range will be reduced by a factor of 4.
- If you examine this using *PCSpim* you will notice that it does not use $PC_{updated}$, but rather $PC_{branch}$. Thus, the target address in *PCSpim* would be $PC_{branch}$ + 3*4 .
- Many machines use the 'old' value of the PC (*i.e.*, the value before it is incremented) – that is (PC-4) or (PC-1) depending on whether the PC contains a byte or a word address.
- Maybe that is why SPIM also uses $PC_{branch}$ = ($PC_{updated}$ - 4), the address of the instruction `bne`.

- Ex: *PC-relative*

```
Loop: slt  $t1,$zero,$a1  # t1=9,a1=5
      beq  $t1,$zero,Exit # no=>Exit
      add  $t0,$t0,$a0    # t0=8,a0=4
      subi $a1,$a1,1      # a1=5
      j    Loop           # goto Loop
Exit: add $v0,$t0,$zero   # v0=2,t0=8
```

Set t1=1 if $zero < $a1

| Address | | | | | | |
|---|---|---|---|---|---|---|
| 80000 | 0 | 0 | 5 | 9 | 0 | 42 |
| 80004 | 4 | 9 | 0 | 3 | | |
| 80008 | 0 | 8 | 4 | 8 | 0 | 32 |
| 80012 | 8 | 5 | 5 | -1 | | |
| 80016 | 2 | 20000 | | | | |
| 80020 | 0 | 8 | 0 | 2 | 0 | 32 |

80020 = 80004 + 4 + 3*4

.22 1998©UCB

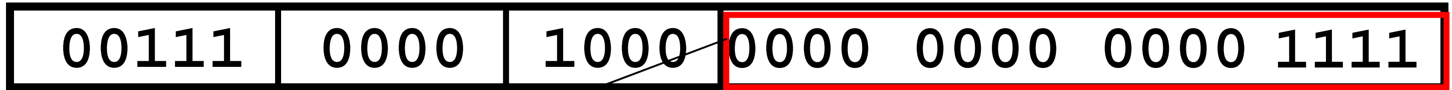
## Instructions for Dealing with Characters

- MIPS (and most other instruction sets) include 2 instructions to move bytes:
  - Load byte (`lb`) loads a byte from memory, placing it in rightmost 8 bits of a register.
  - Store byte (`sb`) takes a byte from rightmost 8 bits of register and writes it to memory.
- Declare byte variables in *C* as *char*.
- Assume *x*, *y* are declared *char*, *y* in memory at 0(`$gp`) and *x* at 1(`$gp`).
  What is MIPS code for `x = y;` ?

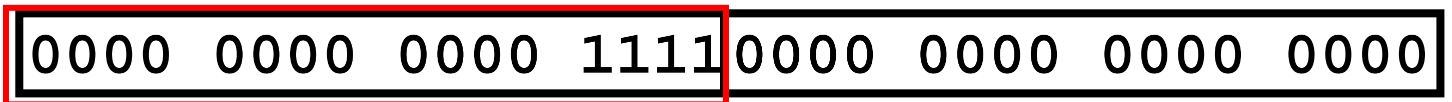
```
lb $t0,0($gp)     # Read byte y
sb $t0,1($gp)     # Write byte x
```

**What happens when constant values are bigger than 16 bits**

- Must keep instructions same size, but immediate field (addi) only 16 bits.
- So, we use one add instruction to load upper 16 bits, then another instruction gives lower 16 bits
    - load upper immediate (`lui`) sets the upper 16 bits of a constant in a register.
- Machine language version of `lui $s0,15`




- Contents of $s0 after executing `lui $s0,15`




Ex: Loading a large constant value

° `C: i = 80000; /* i:$s1 */`

° **MIPS Asm:**

- $80000_{ten}$ =
  $0000\ 0000\ 0000\ 0001\ 0011\ 1000\ 1000\ 0000_{two}$
- `lui $s1, 1`
  `addi $s1,$s1,14464` *# 0011 1000 1000 0000*

° **MIPS Machine Language**