MIPS 3

Assembly to Binary.

- MIPS Instructions has a fixed length of 32 bits
- To reduce complexity of processor design → regular instruction encoding.
  ∴ small no. of formats, as few variations as possible.

Instruction Formats.

- Instr. classified according to their operands (some operands have same encoding)

R-format op $r1, $r2, $r3

- 2 source registers
- 1 destination register
- eg. add, sub, and, or, nor, slt.
- special: srl, sll.

I-format op $r1, $r2, Immd

- 1 source register
- 1 destination register
- 1 immediate value
- eg. addi, andi, ori, slti, lw, sw, beq, bne.

J-format op Immd

- 1 immediate value
- j instructions.

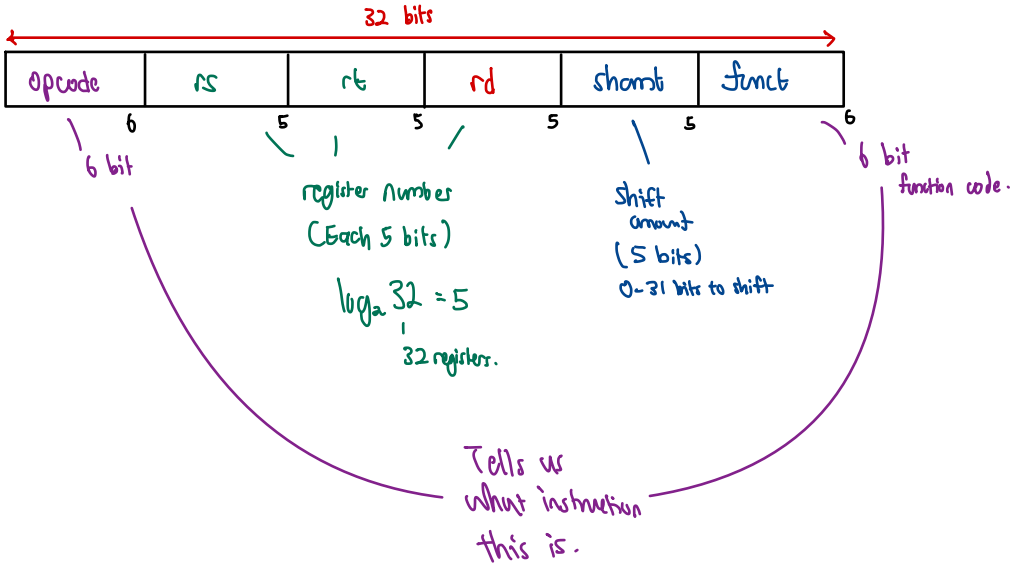
Name	Register number	Usage
$zero	0	Constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Program variables

Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

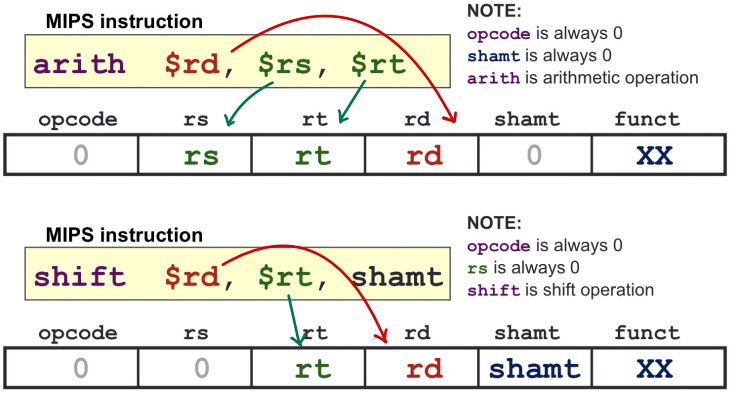
do not touch

$at (register 1) is reserved for the assembler.
$k0-$k1 (registers 26-27) are reserved for the operation system.

R-format



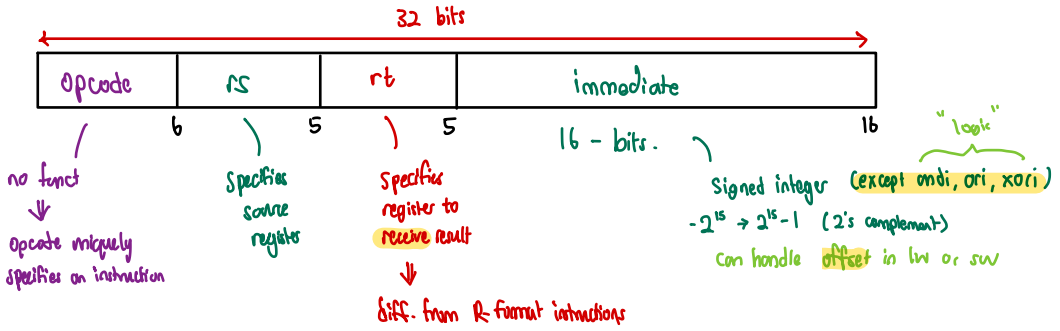
Fields	Meaning
opcode	- Partially specifies the instruction - Equal to 0 for all R-Format instructions
funct	- Combined with opcode exactly specifies the instruction
rs (Source Register)	- Specify register containing first operand
rt (Target Register)	- Specify register containing second operand
rd (Destination Register)	- Specify register which will receive result of computation
shamt "Shift amt"	- Amount a shift instruction will shift by - 5 bits (i.e. 0 to 31) - Set to 0 in all non-shift instructions



Conversions → Decimal → Binary → Hexadecimal.

I-format

- 5 bit shamt field can only represent 0 to 31
- Immediates may be much larger (eg. lw, sw)
- partially consistent with R-format.

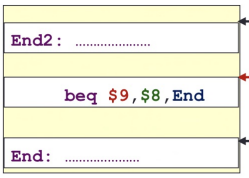


Instruction Address

- Instructions in memory have addresses
  - used by Control Flow instr.
- Instructions are word-aligned (multiple of 4) as they are 32-bits long.
- Program Counter (PC)
  - special register that keeps address of next instruction being executed in the processor.

PC relative addressing

- bne, beq → change PC/addr. of next instr.
- 16 bits for immediate → not enough for 32 bit memory address.
- Loops are generally small. (<50)
- only for branch (not jump) usually large.
  - specify target address relative to PC
  - target addr = PC + 16-bits immediate field.
    ~ PC
    signed 2's complement.



Can branch to $\pm 2^{15}$ bytes from PC

- no. of bytes added to PC is always a multiple of 4.
- Interpret the immediate as no. of words (multiplied by $4_{10}(100_2)$)

⇓ $\pm 2^{15}$ words from PC

extend range to $\pm 2^{17}$ bytes from PC

branch 4 times further.

In branching, immediate value is no. of instr. to jump if branch is taken.

count start from next instruction (forward)

till if branch taken. (may be backwards).

Branch Calculation:

If branch is not taken:

PC = PC + 4

addr. of next instr. (next word).

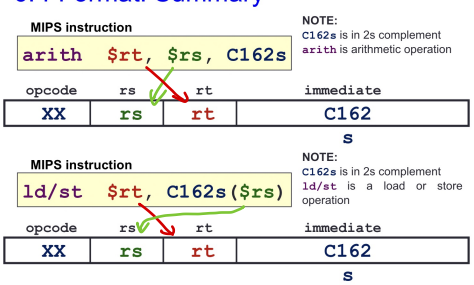
If branch is taken:

PC = (PC+4) + (immediate × 4)

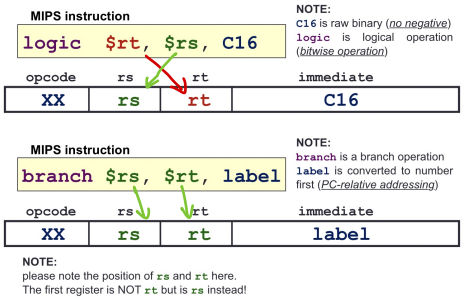
hardware design.

Address of instr. are sequential.

6. I-Format: Summary

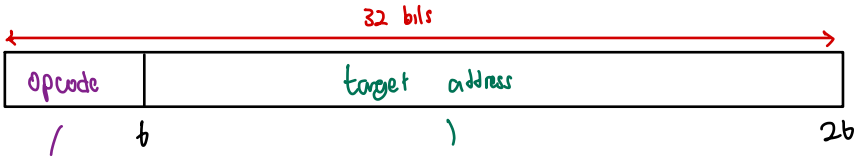


6. I-Format: Summary



J-format

- Jumps anywhere in the memory.
- eg. calls OS function (diff. part of memory)
- Cannot specify 32 bit memory address in Jump instruction.



kept identical to R-format and I-format for consistency

26-bits

Optimisation

- Since jump will only jump to word-aligned addresses, last 2 bits always 00. (x 4)
- Assume address ends with "00" and leave them out.

Specify 28 bits out of 32-bits address.

jumps to 2^{26} instructions / 2^{28} bytes.

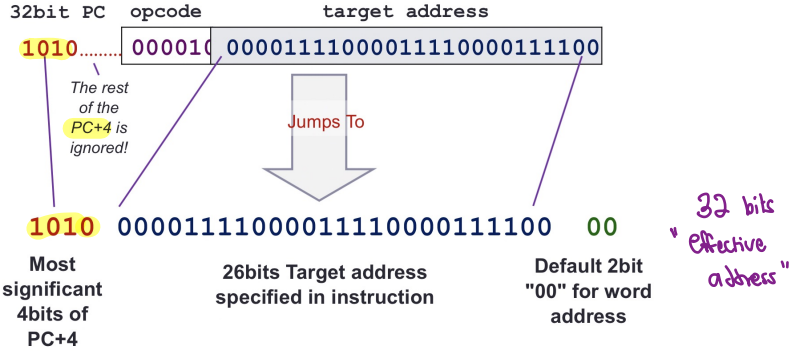
MIPS choose to take the 4 most significant bits from PC+4.
(next instruction after the jump instruction).

- Cannot jump anywhere in memory
- Should be sufficient most of the time.
- maximum jump range = 256 MB boundary. i.e. 2^{28} bytes

"jr" instruction. (if straddles boundary)

target address is specified through a register.

- **Summary:** Given a **Jump** instruction

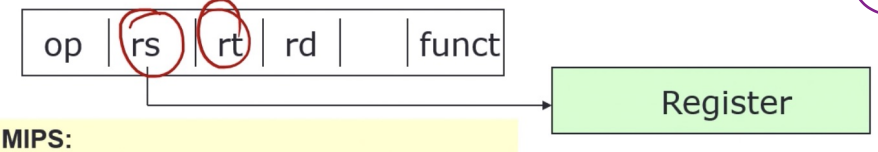


32 bits "effective address"

8. Addressing Modes (1/3)

Ways to calculate final address of the operand.

1. **Register addressing**: operand is a register (shamt)



except this. "shift amt"

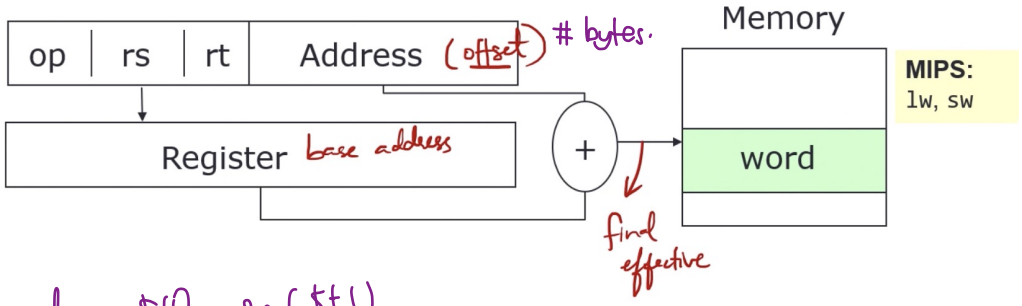
MIPS:
add, sub, and, or, xor, nor, slt, sll, srl

2. **Immediate addressing**: operand is a constant within the instruction itself (`addi`, `andi`, `ori`, `slti`)

16-bits — embedded in instruction set.

MIPS:
addi, andi, ori, xori, slti

3. **Base addressing (displacement addressing)**: operand is at the memory location whose address is sum of a register and a constant in the instruction (`lw`, `sw`)

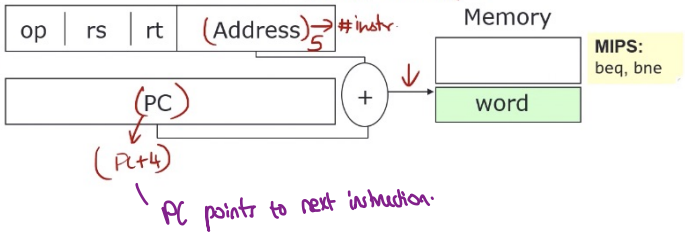


MIPS:
lw, sw

`lw $t0, 20($t1)`
offset · base addr.

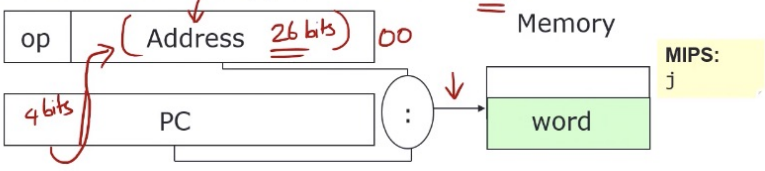
4. **PC-relative addressing**: address is sum of PC and constant in the instruction (`beq`, `bne`)

1. base address → PC
2. displacement or offset → # instr.
PC + 5 x 4 = 20 bytes



MIPS:
beq, bne

5. **Pseudo-direct addressing**: 26-bit of instruction concatenated with upper 4-bits of PC (`j`)



MIPS:
j

Summary (1/2)

- MIPS Instruction:
  32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

  - Branches and load/store are both I-format instructions; but branches use PC-relative addressing, whereas load/store use base addressing
  - Branches use ***PC-relative*** addressing
    Jumps use ***pseudo-direct*** addressing
  - Shifts use R-format, but other immediate instructions (`addi`, `andi`, `ori`) use I-format

IN OTHER WORDS:
beq, bne: ignore PC, count displacement (from PC)
`j`: use PC, ignore displacement (uses 4 MSB of PC).

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
Data transfer	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
	load w ord	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100	Word from memory to register
	store w ord	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi- tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For sw itch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call