Practice problems and solutions: MIPS and Instructions set Architecture

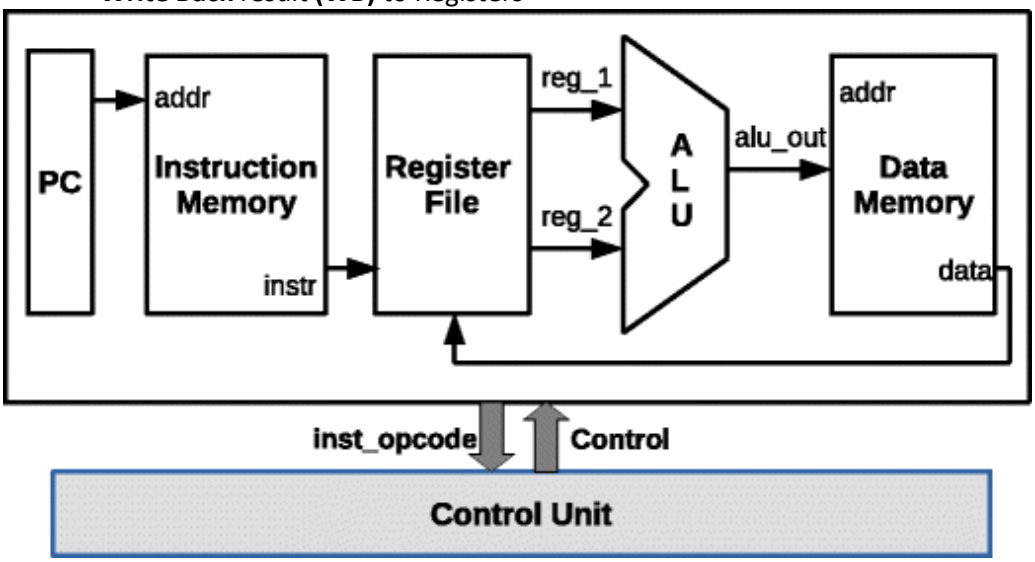
	Discuss the history of RISC The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy. IBM RISC technology originated in 1974 in a project to design a large telephone-switching network capable of handing an average of three hundred calls per second. With an approximate 20 000 instructions per call and Stringent real-time response requirements, the performance target was 12 million instructions per second (MIPS). This specialized application required a very fast processor, but did not have to perform computed instructions and had little demand for floating-point calculations. Other than moving data between registers and memory, the machine had to be able to add, combine fields extracted from several registers, perform branches, and carry out input/output operations. When the telephone project was terminated in 1975, the machine itself had not been built, but the design had progressed to the point where it seemed to be an excellent basis for a general-purpose, high-performance miniprocessor. The attractiveness of the processor design stemmed from projections that it would be able to compute at high speed relative to its cost in a variety of application areas. The most important features of the telephone switching machine which contributed to its low cost/performance ratio were 1) separate instruction and data caches, allowing a much higher bandwidth between memory and CPU; 2) no arithmetic operations to storage, which greatly simplified the pipeline; and 3) uniform instruction length and simplicity of design, making possible a very short cycle time: ten levels of logic. (For example, all register-to-register operations executed in one cycle.) John Cocke and his colleagues developed simpler ISAs and compilers for minicomputers. As an experiment, they retargeted their research compilers to use only the simple register-register operations and load-store data transfers of the IBM 360 ISA, avoiding the more complicated instructions. They found that programs ran up to three times faster using the simple subset. Emer and Clark found 20% of the VAX instructions needed 60% of the microcode and represented only 0.2% of the execution time.
	List the design features of RISC processors. • *one cycle execution time*: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called pipelining; • *pipelining*: a technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions; • *large number of registers*: RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory
	Discuss MIPS architecture and its development. MIPS, an acronym for Microprocessor **without Interlocked Pipeline Stages**, was a research project conducted by John L. Hennessy at Stanford University between 1981 and 1984. The MIPS processor was developed as part of a VLSI research program at Stanford University in the early 80s. Professor John Hennessy started the development of MIPS with a brainstorming class for graduate students. The readings and idea sessions helped launch the development of the processor which

became one of the first RISC processors, with IBM and Berkeley developing processors at around the same time.

- MIPS processor implemented a smaller, simpler instruction set. Each of the instructions included in the chip design ran in a single clock cycle. The processor used a technique called pipelining to more efficiently process instructions.
- MIPS used 32 registers, each 32 bits wide (a bit pattern of this size is referred to as a *word*).

Instruction cycle of MIPS processor was subdivided into **five stages**:

- **Instruction Fetch (IF)**
- **Instruction Decode (ID) and Register Read**
- **Execution (EXE)**
- **Memory read/write(MEM)**
- **Write Back** result **(WB)** to Registers



Discuss MIPS I instruction set architecture.

The instruction set consists of about 111 total instructions. A variety of basic instructions, including:

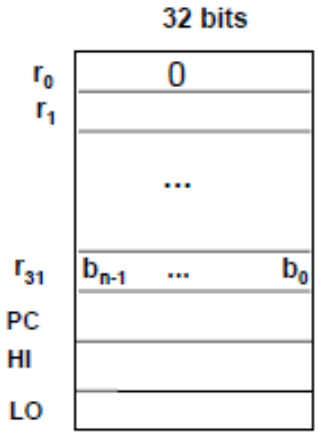
21 - Arithmetic instructions (+, -, *, /, %)
8- Logic instructions (&, |, ~)
8- Bit manipulation instructions
12- Comparison instructions (>, <, =, >=, <=, ¬)
25 - Branch/jump instructions
15 - Load instructions
10 - Store instructions
8 - Move instructions
4 - Miscellaneous instructions

Describe the register architecture of MIPS.

MIPS has 32 registers Each register is 32 bits wide.

MIPS Registers (R2000/R3000)

- 32x32-bit GPRs (General purpose registers)
  - $0 = $zero (therefore only 31 GPRs)
  - $1 = $at (reserved for assembler)
  - $2 - $3 = $v0 - $v1 (return values)
  - $4 - $7 = $a0 - $a3 (arguments)
  - $8 - $15 = $t0 - $t7 (temporaries)
  - $16 - $23 = $s0 - $s7 (saved)
  - $24 - $25 = $t8 - $t9 (more temporaries)
  - $26 - $27 = $k0 - $k1 (reserved for OS)
  - $28 = $gp (global pointer)
  - $29 = $sp (stack pointer)
  - $30 = $fp (frame pointer)
  - $31 = $ra (return address)



- 32x32-bit floating point registers (paired double precision)
- HI, LO, PC
- Status, Cause, BadVAddr, EPC

Register	Alias	Usage	Register	Alias	Usage
$0	$zero	constant 0	$16	$s0	saved temporary
$1	$at	used by assembler	$17	$s1	saved temporary
$2	$v0	function result	$18	$s2	saved temporary
$3	$v1	function result	$19	$s3	saved temporary
$4	$a0	argument 1	$20	$s4	saved temporary
$5	$a1	argument 2	$21	$s5	saved temporary
$6	$a2	argument 3	$22	$s6	saved temporary
$7	$a3	argument 4	$23	$s7	saved temporary
$8	$t0	unsaved temporary	$24	$t8	unsaved temporary
$9	$t1	unsaved temporary	$25	$t9	unsaved temporary
$10	$t2	unsaved temporary	$26	$k0	reserved for OS kernel
$11	$t3	unsaved temporary	$27	$k1	reserved for OS kernel
$12	$t4	unsaved temporary	$28	$gp	pointer to global data
$13	$t5	unsaved temporary	$29	$sp	stack pointer
$14	$t6	unsaved temporary	$30	$fp	frame pointer
$15	$t7	unsaved temporary	$31	$ra	return address

Name the registers that hold programmer variables
Registers $16-$23 which are indicated by $s0-$s7 in low level language/instructions.

Name the registers that hold temporary variables.
Registers $8-$15 which are indicated by $t0-$t7 in low level language/instructions.
Registers $24-$25 which are indicated by $t8-$t9 in low level language/instructions.

Show the format and syntax of ALU instruction using register addressing mode. Give example

Instruction Syntax of ALU instruction using register addressing mode is:
op dst, src1, src2,
Here 3 operands are used.

- op = operation name ("operator")
- dst = register getting result ("destination")
- src1 = first register for operation ("source 1")
- src2 = second register for operation ("source 2")

Example: **add $s1, $s2, $s3**
sub $s1, $s2, $s3

Suppose, a = $s0, b = $s1, c = $s2, d = $s3, and e = $s4. Convert the following C statement to MIPS: a = (b + c) – (d + e);

```
add $t1, $s3, $s4
add $t2, $s1, $s2
sub $s0, $t2, $t1
```

What is Register zero ($0 or $zero) register?
Register $0 or $zero always has the value 0 and cannot be changed.
Example Uses:

```
add  $s3,  $0,  $0
add  $s1, $s2,  $0
```

Show the format and syntax of ALU instructions using immediate mode.

- Numerical constants are called immediates
- Separate instruction syntax for immediates:
  opi dst, src, imm

Operation names end with 'i', replace 2nd source register with an immediate
Example Uses:

```
addi $s1, $s2, 5  # a=b+5
addi $s3, $s3, 1  # c++
```

Show the format and syntax for Data Transfer instructions with examples.
Instruction set of RISC processors has LOAD and STORE instructions to transfer data between Register and RAM.
Using LOAD instruction, CPU reads data from RAM and saves/copies to a register. CPU saves the contents of a register to RAM using STORE instruction.
Syntax for data transfer (load/store)instruction syntax for data transfer between Register and Memory is as follows:

opcode	reg	Off(bAddr)

- op = operation name ("operator")
- reg = register for operation source or destination
- bAddr = register with pointer to memory ("base address")
- off = address offset (immediate) in bytes ("offset")

Accesses memory at address: bAddr + off

Instruction for Load Word (32 bits/4 Bytes):

lw $t0, 12($s3)

CPU reads 4 bytes data starting from memory address = 12 + [$s3] and saves it into register $t0.

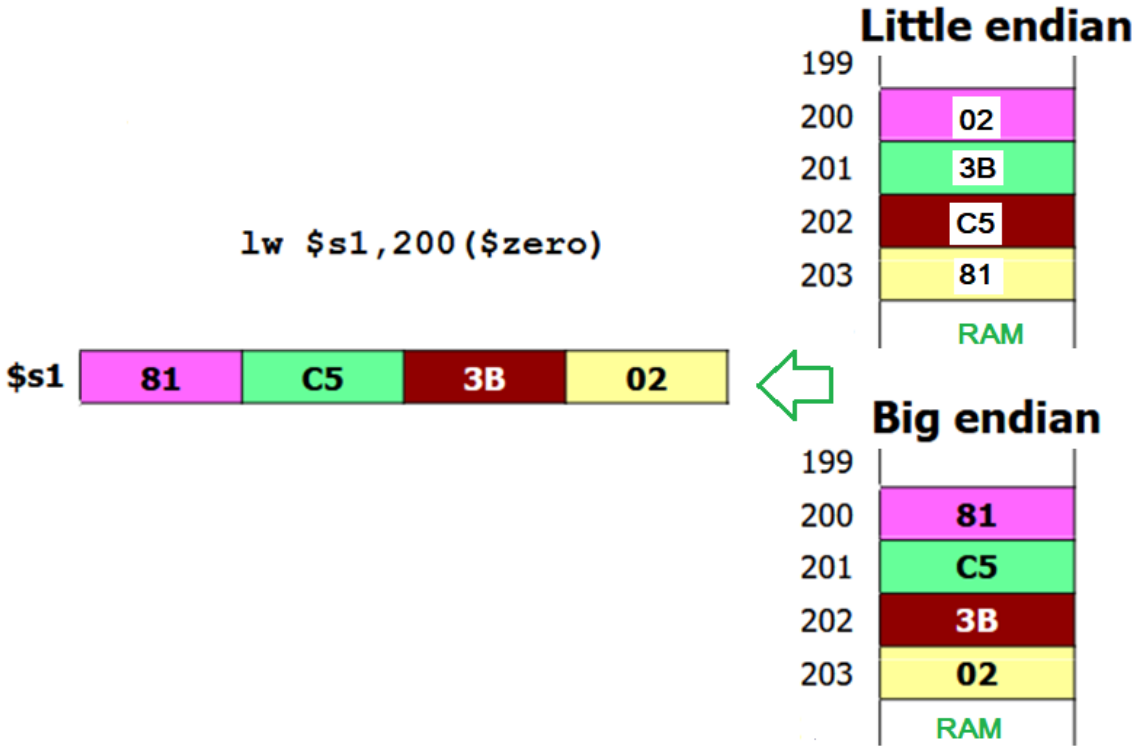
Instruction for Store Word (32 bits/4 Bytes):

sw $t0, 40($s3)

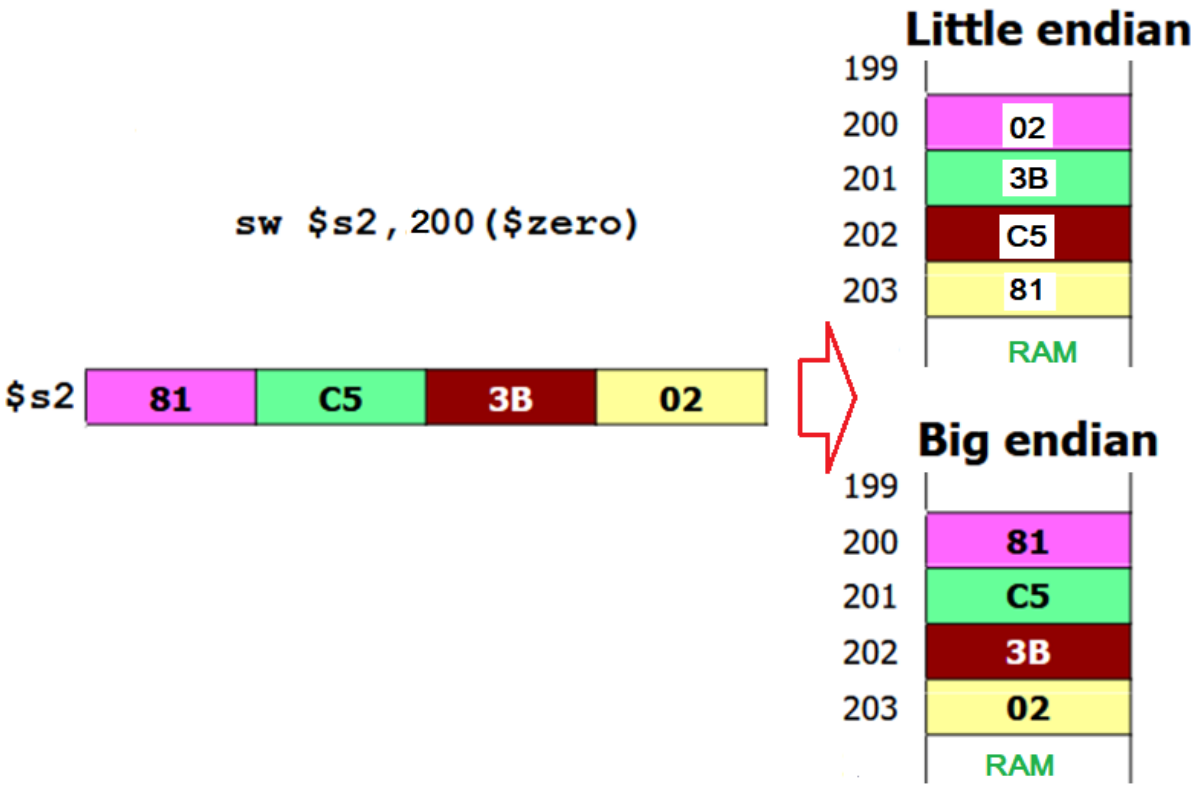
CPU saves contents of $t0 (32 bits/4 bytes) to memory, starting at address = 40 + [$s3].

How 32-bits/4bytes data is loaded to a register in load instruction?
MIPS uses byte addressable memory. In case of LOAD WORD (lw) instruction, CPU reads from 4 consecutive locations of memory and saves to a register.
There are two standards
Big Endian: Most-significant byte at least address
Little Endian: Least-significant byte at least address
Examples:
lw $s1, 200($zero)
sw $s2, 200($zero)

In both the instruction, memory address pointed by 200($zero) = 200 + 0 = 200.
A 32-bit data 0x81C53B02 is assumed to be saved in either little endian or big endian as shown.
Using lw $s1, 200($zero), CPU will read the 32-bit/4bytes data, starting from memory address 200 and saves to register $s1.



Here it is assumed that the content of register $s2 is 0x81C53B02.
Using sw $s2, 200($zero), CPU will save the 32-bit/4bytes data, starting from memory address 200 and saves to register $s1.



Show the format and syntax of decision making/conditional branch instructions with examples.
The general format of decision making/conditional branch instructions is as follows

Opcode	Reg1	Reg2	offset

Opcode holds the condition to be tested between Reg1 and Reg2. If the condition is evaluated TRUE, CPU will jump to memory address: PC + 4 + offset and read/fetch instruction from that location. If the condition is evaluated FALSE, CPU will read/fetch next instruction from memory address: PC + 4.

Example: beq $1, $2, 100
bne $1, $2, 100
In the first instruction, CPU will check whether contents of $1 and $2 are EQUAL, if found equal , CPU will jump to memory address PC + 4 + 100 and read/fetch instruction.
In the 2nd instruction, CPU will check whether contents of $1 and $2 are UNEQUAL, if found unequal, CPU will jump to memory address PC + 4 + 100 and read/fetch instruction.

branch on equal Test if registers are equal	`beq $1,$2,100`	if($1==$2) go to PC+4+100
branch on not equal Test if registers are not equal	`bne $1,$2,100`	if($1!=$2) go to PC+4+100

In conditional branch instructions, labels are also used instead of offset to point next instruction to be read/fetch if condition is evaluated true.
The general format of decision making/conditional branch instructions is as follows

Opcode	Reg1	Reg2	label

Example: beq $t0, $0, NSU
Here the CPU will check whether $t0 = 0. If $t0 is found zero, CPU will jump to instruction lb $t0, 0($s0) as pointed by user defined label 'NSU'.
If the condition is evaluated FALSE, the CPU will process next instruction that appears in the list, it means, instruction: addi $s0, $s0, 1

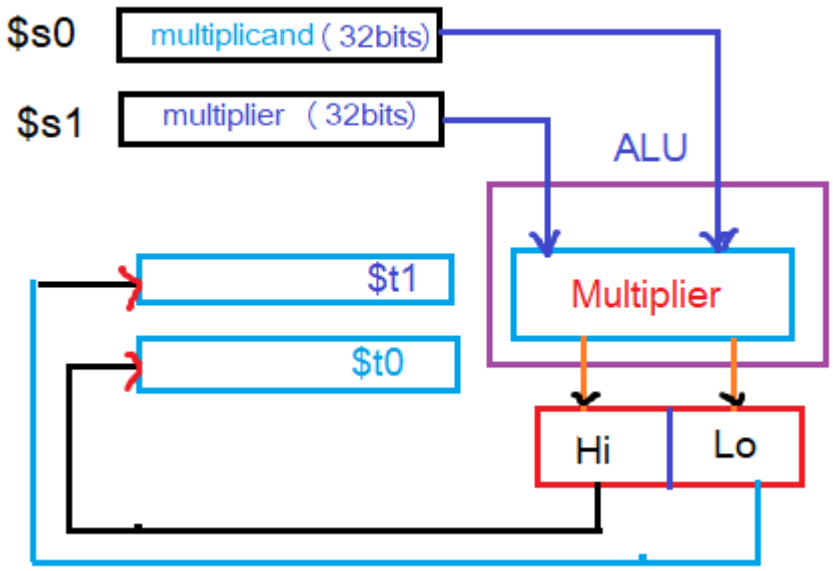
```
NSU: lb   $t0,0($s0)
     sb   $t0,0($s1)
     addi $s1,$s1,1
     beq  $t0,$0,NSU
     addi $s0,$s0,1
```

Show the format and syntax of unconditional jump instruction.
The general format of unconditional jump instruction is

Jump	label

The syntax is
J label

Explain the multiplication operation with instruction format, syntax and example.
In MIPS multiplication instructions, two 32 bit registers are used to hold multiplicand and multiplier
mult $s0, $s1
Here $s0 and $s1 hold multiplicand and multiplier respectively, each of 32 bits. The product is 64 bits: higher 32 bits are saved in Hi and lower 32-bits are saved in Lo register by default.
Two halves of the products can be loaded to other registers using mfhi reg1 and mflo reg2 instructions following multiply instruction.
If it is intended to save higher and lower 32 bits of the product to $t0 and $t1 registers, following instructions would be used
mfhi $t0
mflo $t1
The following diagram shows the multiplication operation as discussed.



Explain the division operation with instruction format, syntax and example.

In MIPS division instructions, two 32 bit registers are used to hold dividend and divisor

div $s0, $s1

Here $s0 and $s1 hold dividend and divisor respectively, each of 32 bits. The remainder is saved ito Hi and quotient is saved to Lo register by default.

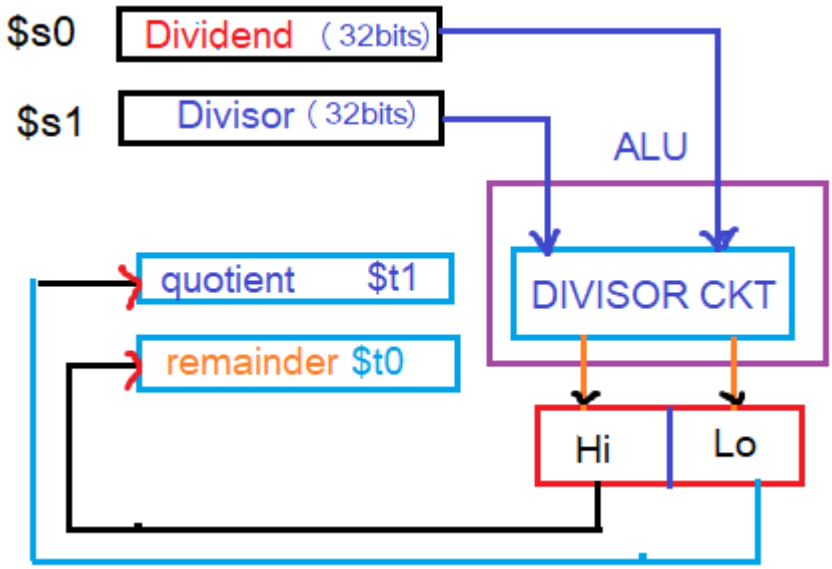
However, remainder and quotient can be loaded to other registers using mfhi reg1 and mflo reg2 instructions following div instruction.

If it is intended to save remainder to $t0 and quotient to $t1 register, following instructions would be used

mfhi $t0

mflo $t1

The following diagram shows the division operation as discussed.



Show formats of machine codes /binary instruction formats of MIPS processor.

There are three types of instruction formats: R-format, I-format and J-format. All formats use 32 bits binary to represent instructions.

Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

Instruction fields
op: operation code (opcode)
Rs: first source register number
Rt: second source register number
Rd: destination register number
Shift: shift amount (00000 for now)
Function: function code (extends opcode)

R-format is used in ALU instructions supporting register addressing mode.
I-format is used in following instructions and addressing modes

a) ALU instructions in immediate mode
b) LOAD and STORE instructions
c) Conditional Branch instructions

J-format is used in unconditional branch/jump instructions

Explain R-format instruction with an example.

R-format is used in ALU instructions supporting register addressing mode. Here there are three operand fields: rs, rt and rd, referring to first source register number, second source register number and destination register number respectively. funct field in 6-bits represent operation code whereas shamt and op fields are set to zeros, each of 5-bits in binary. Register numbers are represented in 5-bits binary. Following example shows the R-format in steps:

Example: Instruction `add $t0, $s1, $s2`

General R-format	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Operand Fields	op	$s1	$s2	$t0	0	add
Codes in decimal	0	17	18	8	0	32
codes in binary	000000	10001	10010	01000	00000	100000

Show another example of R-format instruction.

Example: Instruction `Sub $t0, $s1, $s2`

General R-format	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Operand Fields	op	$s1	$s2	$t0	0	sub
Codes in decimal	0	17	18	8	0	34
codes in binary	000000	10001	10010	01000	00000	100100

Explain I-format instruction with an example.

I-format is used in following instructions and addressing modes

a) ALU instructions in immediate mode:
b) LOAD and STORE instructions:
c) Conditional Branch instructions:

The general I-format is as follows

op	rs	rt	constant/Immediate
6 bits	5 bits	5 bits	16 bits

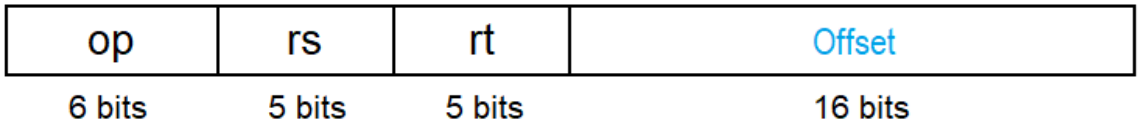
For ALU instructions in immediate mode:
rt: destination register
Constant/immediate: second operand/data
rs: first source register in ALU instruction

Example: Instruction **`addi $s1,$s2,16`**

General I-format	op	rs	rt	constant/Immediate
	6 bits	5 bits	5 bits	16 bits
Operand Fields	addi	$s2	$s1	16
Codes in decimal	8	18	17	16
codes in binary	001000	10010	10001	0000000000010000

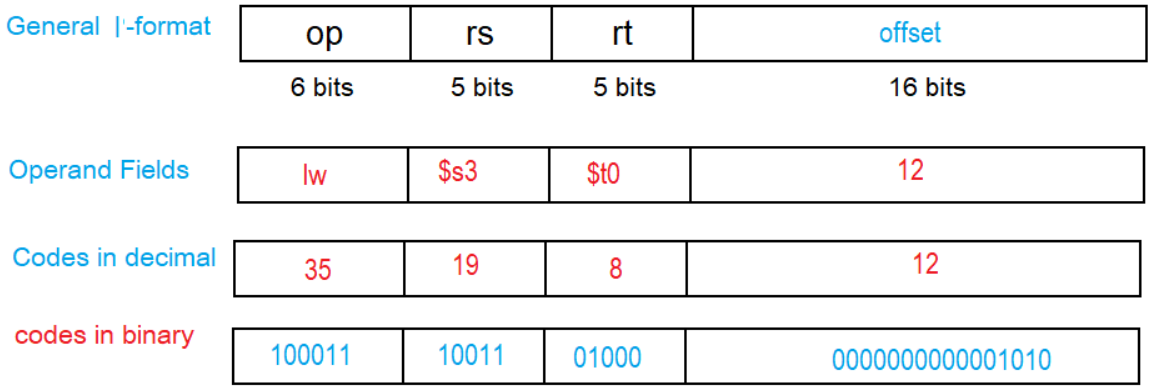
Explain I-format instruction format for LOAD instruction with an example.

The general I-format is as follows



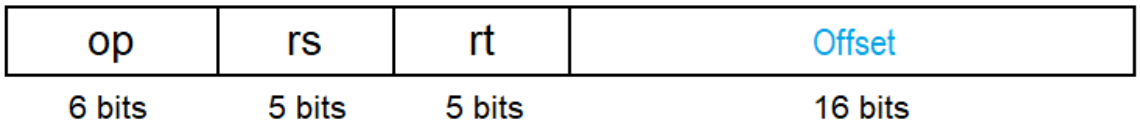
For LOAD instruction:
rt: destination register
Offset: this number is to be added to the content of Base register and memory address is formed
rs: Base register

Example: Instruction `lw $t0,12($s3)`



Explain I-format instruction for STORE instruction with an example.

The general I-format is as follows



For STORE instruction:
rt: source register
Offset: this number is to be added to the content of Base register and memory address is formed
rs: Base register

Example: Instruction SW $t0,12($s3)

General I-format

op	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

Operand Fields	SW	$s3	$t0	12
Codes in decimal	43	19	8	12
codes in binary	101011	10011	01000	0000000000001010

Explain I-instruction format for conditional branch instruction with an example.

The general I-format is as follows

op	rs	rt	Offset
6 bits	5 bits	5 bits	16 bits

For conditional branch instruction:
rt: second register
Offset: this number is to be added to (PC +4) and PC is loaded with PC + 4 + 64
rs: first register

Example Instruction: beq $t0, $s5, 64
Here CPU will check the contents of $t0 and $s5. If the contents are found to be equal, PC is loaded with (PC + 4 +64). It means that the CPU will jump to memory address PC + 4 + 64 and read/fetch next instruction. Binary format of above instruction is shown below in some steps:

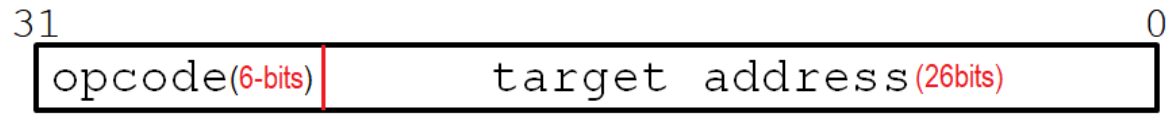
Example: Instruction beq $t0, $s5, 64

General I-format

op	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

Operand Fields	beq	$t0	$s5	64
Codes in decimal	4	8	21	64
codes in binary	000100	01000	10101	0000000001000000

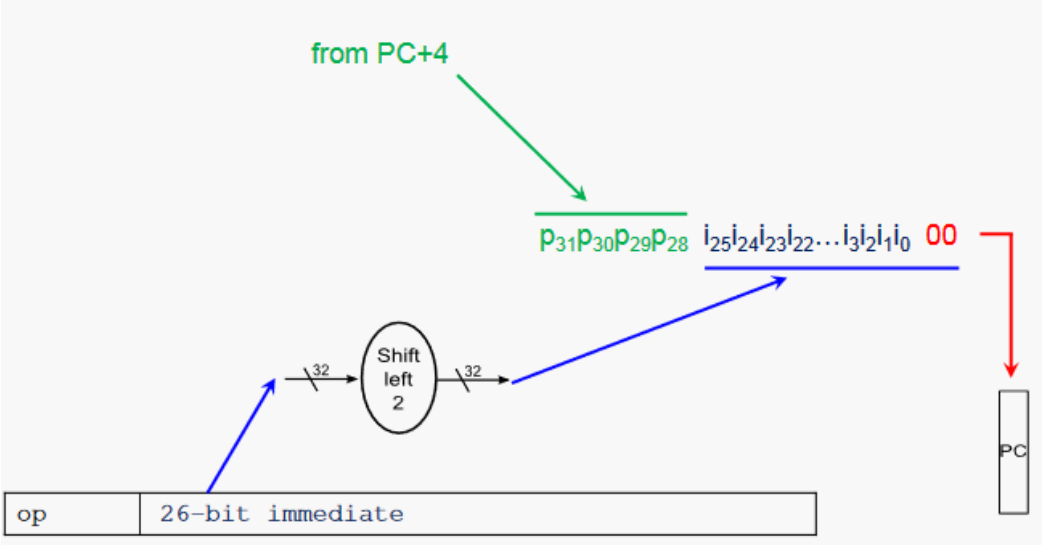
Show J-format of instruction.
The general j-format for instruction is



This format is used for unconditional jump instruction.

Example:

J	10100000001100001111000010



The 26-bits target address is converted to 32 bits, shown above. 26 bits target address is shifted left by 2 bits and thereby a 28-bits binary pattern is formed. Four Most significant bits are extracted from binary representation of PC + 4 and appended at the MSB position of 28-bit patterns. This value is loaded to program counter.