Difference Between Von Neumann and Harvard Architecture

Parameters	Von Neumann Architecture	Harvard Architecture
Definition	The Von Neumann Architecture is an ancient type of computer architecture that follows the concept of a stored-program computer.	Harvard Architecture is a modern type of computer architecture that follows the concept of the relay-based model by Harvard Mark I.
Physical Address	It uses one single physical address for accessing and storing both data and instructions.	It uses two separate physical addresses for storing and accessing both instructions and data.
Buses (Signal Paths)	One common signal path (bus) helps in the transfer of both instruction and data.	It uses separate buses for the transfer of both data and instructions.
Number of Cycles	It requires two clock cycles for executing a single instruction.	It executes any instruction using only one single cycle.

Cost	It is comparatively cheaper in cost than Harvard Architecture.	It is comparatively more expensive than the Von Neumann Architecture.
Access to CPU	The CPU is not able to read/write data and access instructions at the same time.	The CPU can easily read/write data as well as access the instructions at any given time.
Uses	This method comes to play in the case of small computers and personal computers.	This architecture is best for signal processing as well as microcontrollers.
Requirement of Hardware	As compared to Harvard Architecture, Von Neumann Architecture requires lesser architecture. It is because it only needs to reach one common memory.	This one requires more hardware. It is because it requires separate sets of data as well as address buses for individual memory.
Requirement of Space	This architecture basically requires less space.	This architecture comparatively requires more space.
Usage of Space	This architecture does not waste any space. It is because the instruction memory can utilize the left space of the data memory. It can also happen vice-versa.	This type of architecture can result in space wastage. It is because the instruction memory cannot utilize the leftover space in the data memory. It also cannot happen vice-versa.

Execution Speed	The speed of execution of the Von Neumann Architecture is comparatively slower. It is because it is not capable of fetching the instructions and data both at the same time.	The overall speed of execution of Harvard Architecture is comparatively faster. It is because the processor, in this case, is capable of fetching both instructions and data at the very same time.
Controlling	The process of controlling becomes comparatively simpler with this architecture. It is because it fetches either instructions or data at any given time.	The process of controlling becomes comparatively complex with this architecture. It is because it basically fetches both instructions and data simultaneously at the very same time.

4.	Implementation of complex instructions is enabled through memory units.	RISC lacks special memory and thus utilizes specialized hardware to execute instructions.
5.	CISC devices are installed with a microprogramming unit.	RISC devices are embedded with a hardwired programming unit.

6.	CISC uses a variety of instructions to accomplish complex tasks.	RISC is provided with a reduced instruction set, which is typically primitive in nature.
7.	CISC processors are generally micro-coded, thereby allowing ROM-based CPU control. However, modern CISC processors also use hardwired units for easy CPU control.	RISC processors use hardwired units to control CPUs.

8.	A CISC processor works with 16 bits to 64 bits to execute each instruction.	A RISC processor utilizes 32 bits to execute each instruction.
9.	A CISC architecture uses one cache to store data as well as instructions. However, recent CISC designs employ split caches to divide data and instructions.	The RISC architecture relies on split caches, one for data and the other for instructions.

10.	CISC processors use a memory-to-memory framework to execute instructions such as ADD, LOAD, and even STORE.	RISC processors rely on a register-to-register mechanism to execute ADD, STORE, and independent LOAD instructions.
11.	The CISC architecture uses only one register set.	The RISC architecture utilizes multiple registers sets.

12.	Since CISC devices operate in a multi-clock environment, it supports addressing modes in the range of 12 to 24.	Since RISC machines operate on single clock cycles, it has limited addressing modes.
13.	CISC processors are capable of processing high-level programming language statements more efficiently, thanks to the support of complex addressing modes.	Since RISC processors support a limited set of addressing modes, complex instructions are synthesized through software codes.

14.	CISC does not support parallelism and pipelining. As such, CISC instructions are less pipelined.	RISC processors support instruction pipelining.
15.	CISC complexity is embedded in microprograms.	RISC complexity is pinned with compilers that execute the software program.
16.	CISC instructions require high execution time.	RISC instructions require less time for execution.

17.	CISC supports code expansion, which is similar to macro expansion, wherein a copy of inline functions is added in each place wherever it is called. Such inline functions run faster than normal functions.	RISC does not support code expansion.
18.	In the CISC architecture, the task of decoding instructions is quite complex.	In RISC processors, instruction decoding is simpler than in CISC.

19.	Some examples of CISC processors include Intel x86 CPUs, System/360, VAX, PDP-11, Motorola 68000 family, and AMD.	Examples of RISC processors include Alpha, ARC, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture, and SPARC.
20.	CISC processors are used for low-end applications such as home automation devices, security systems, etc.	RISC processors are suitable for high-end applications, including image and video processing, telecommunications, etc.

- A **complex instruction set computer** (**CISC**) is a computer architecture in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions

- Specific instruction set architectures that have been retroactively labeled CISC are System/360 through z/Architecture, the PDP-11 and VAX architectures, and many others. Well known microprocessors and microcontrollers that have also been labeled CISC in many academic publications include the Motorola 6800, 6809 and 68000 families; the Intel 8080, iAPX 432 and x86 family; the Zilog Z80, Z8 and Z8000 families; the National Semiconductor NS320xx family; the MOS Technology 6502 family; the Intel 8051 family; and others.

- Microchip Technology PIC has been labeled RISC in some circles and CISC in others.

Most RISC designs use uniform instruction length for almost all instructions, and employ strictly separate load and store instructions.

Reduced instruction set computer (**RISC**) is a computer architecture designed to simplify the individual instructions given to the computer to accomplish tasks. Compared to the instructions given to a complex instruction set computer (CISC), a RISC computer might require more instructions (more code) in order to accomplish a task because the individual instructions are written in simpler code.

The varieties of RISC processor design include the ARC processor, DEC Alpha, the AMD Am29000, the ARM architecture, the Atmel AVR, Blackfin, Intel i860, Intel i960, LoongArch, Motorola 88000, the MIPS architecture, PA-RISC, Power ISA, RISC-V, SuperH, and SPARC. RISC processors are used in supercomputers, such as the Fugaku.

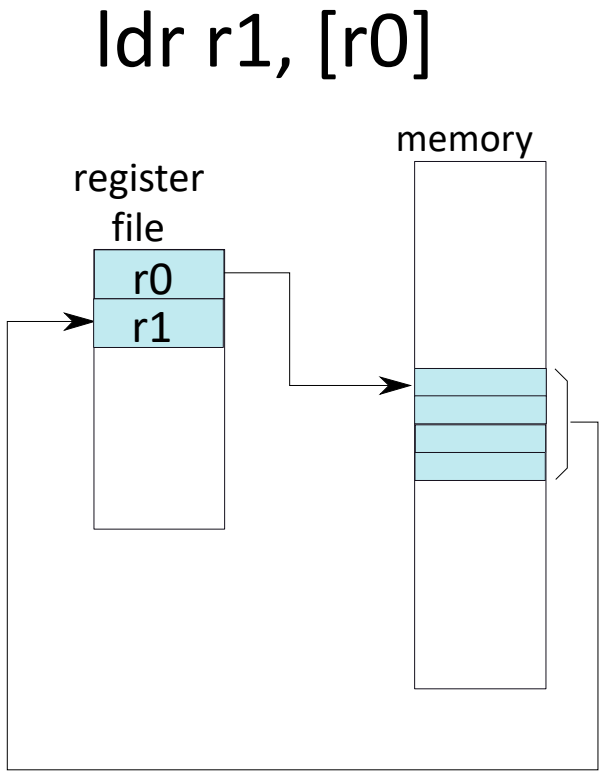
ARM (Advanced RISC Machine) is a well-known example of the RISC framework.

Its processors are observed in desktops, laptops, smartphones, gaming consoles, and several other smart IoT devices that are battery-operated where energy efficiency is essential.

ARM Assembly language

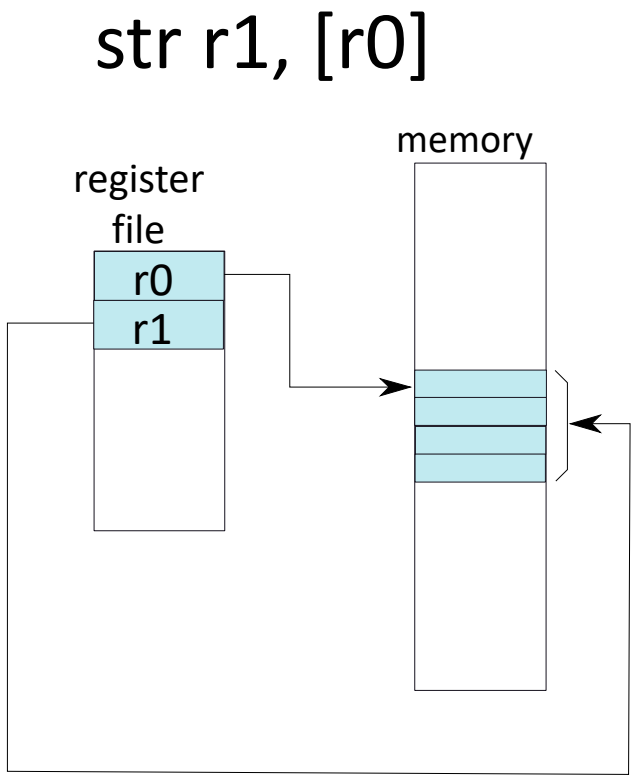
Basic Load Instruction

- ldr r1, [r0]



Basic Store Instruction

- str r1, [r0]



Load and Store Instructions with an Offset

* ldr r1, [r0, #4]
  * r1 ← mem[r0 + 4]
* ldr r1, [r0, r2]
  * r1 ← mem[r0 + r2]

Semantics of Load and Store Instructions

Semantics	Example	Explanation	Addressing Mode
ldr *reg*, [*reg*]	ldr r1, [r0]	$r1 \leftarrow [r0]$	register-indirect
ldr *reg*, [*reg*, *imm*]	ldr r1, [r0, #4]	$r1 \leftarrow [r0 + 4]$	base-offset
ldr *reg*, [*reg*, *reg*]	ldr r1, [r0, r2]	$r1 \leftarrow [r0 + r2]$	base-index
ldr *reg*, [*reg*, *reg*, shift *imm*]	ldr r1, [r0, r2, lsl #2]	$r1 \leftarrow [r0 + r2 << 2]$	base-scaled-index
str *reg*, [*reg*]	str r1, [r0]	$[r0] \leftarrow r1$	register-indirect
str *reg*, [*reg*, *imm*]	str r1, [r0, #4]	$[r0 + 4] \leftarrow r1$	base-offset
str *reg*, [*reg*, *reg*]	str r1, [r0, r2]	$[r0 + r2] \leftarrow r1$	base-index
str *reg*, [*reg*, *reg*, shift *imm*]	str r1, [r0, r2, lsl #2]	$[r0 + r2 << 2] \leftarrow r1$	base-scaled-index

* Note the base-scaled-index addressing mode

Load and Store for Bytes and Half words

Semantics	Example	Explanation	Meaning
ldrb *reg*, [*reg, imm*]	ldrb r1, [r0, #2]	$r1 \leftarrow [r0 + 2]$	1 unsigned byte
ldrh *reg*, [*reg*, *imm*]	ldrh r1, [r0, #2]	$r1 \leftarrow [r0 + 2]$	2 unsigned bytes
ldrsb *reg*, [*reg*, *imm*]	ldrsb r1, [r0, #2]	$r1 \leftarrow [r0 + 2]$	1 signed byte
ldrsh *reg*, [*reg*, *imm*]	ldrsh r1, [r0, #2]	$r1 \leftarrow [r0 + 2]$	2 signed bytes
strb *reg*, [*reg, imm*]	strb r1, [r0, #2]	$[r0+2] \leftarrow r1$	1 unsigned byte
strh *reg*, [*reg*, *imm*]	strh r1, [r0, #2]	$[r0 + 2] \leftarrow r1$	2 unsigned bytes

* Note there is no ARM instruction to extend sign of the operand while saving it in memory

An example with arrays

C

```
void addNumbers(int a[100]) {
        int idx;
        int sum = 0;
        for (idx = 0; idx < 100; idx++){
                sum = sum + a[idx];
        }
}
```

ARM assembly

```
/* base address of array a in r0 */
mov r1, #0 /* sum = 0 */
mov r2, #0 /* idx = 0 */

.loop:
        ldr r3, [r0, r2, lsl #2]
        add r2, r2, #1   /* idx ++ */
        add r1, r1, r3   /* sum += a[idx] */
        cmp r2, #100     /* loop condition */
        bne .loop
```

Advanced Memory Instructions

* Consider an array access again
  * ldr r3, [r0, r2, lsl #2] /* access array */
  * add r2, r2, #1 /* increment index */
* Can we fuse both into one instruction
  * ldr r3, [r0], r2, lsl #2
  * Equivalent to :
    * r3 = [r0]
    * r0 = r0 + r2 << 2

Post-indexed addressing mode

Pre-Indexed Addressing Mode

- Consider
  - ldr r0, [r1, #4]!
- This is equivalent to:
  - r0 ← mem [r1 + 4]
  - r1 ← r1 + 4

Post-Indexed Addressing Mode

- Consider
  - ldr r3, [r0], #4
- This is equivalent to:
  - r0 ← mem [r0]
  - r0 ← r + 4

Example: Adding 100 Integer arrays

C

```
void addNumbers(int a[100]) {
        int idx;
        int sum = 0;
        for (idx = 0; idx < 100; idx++){
                sum = sum + a[idx];
        }
}
```

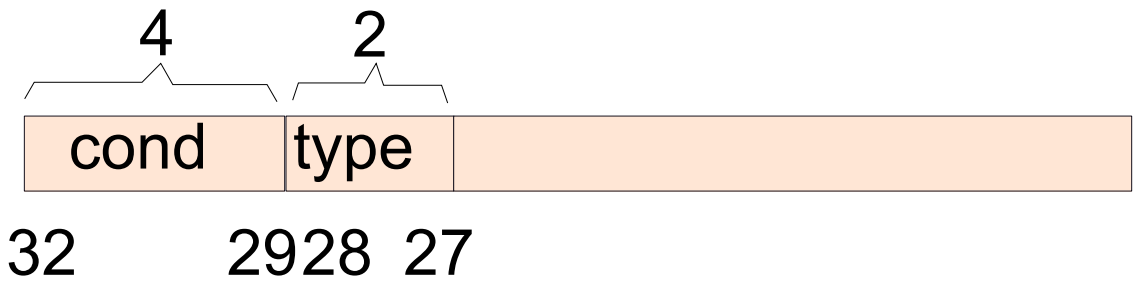
Answer:

ARM assembly

```
/* base address of array a in r0 */
mov r1, #0          /* sum = 0 */
add r4, r0, #400    /* set r4 to address of a[100] */

.loop:
        ldr r3, [r0], #4
        add r1, r1, r3   /* sum += a[idx] */
        cmp r0, r4       /* loop condition */
        bne .loop
```

Encoding in ARM Assembly

Instruction Encoding

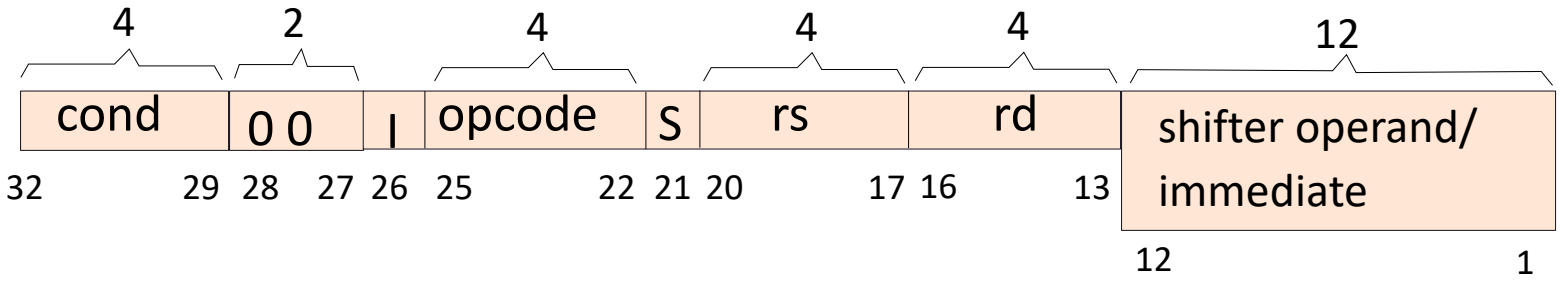


* cond → instruction condition (eq, ne, …)
* type → instruction type

Condition Field

Bits (32:29)	Suffix	Meaning	Flag State
0000	eq	equal	Z = 1
0001	ne	notequal	Z = 0
0010	cs	carry set	C = 1
0011	cc	carry clear	C = 0
0100	mi	negative/ minus	N = 1
0101	pl	positive or zero/ plus	N = 0
0110	vs	overflow	V = 1
0111	vc	no overflow	V = 0
1000	hi	unsigned higher	(C = 1) ∧ (Z = 0)
1001	ls	unsigned lower or equal	(C = 0) ∨ (Z = 1)
1010	ge	signed greater than or equal	(N = 0) ∨ (Z = 1)
1011	lt	signed less than	N = 1
1100	gt	signed greater than	(Z = 0) ∧ (N = 0)
1101	le	signed less than or equal	(Z = 1) ∨ (N = 1)
1110	al	always	
1111	–	reserved	

Data Processing Instructions



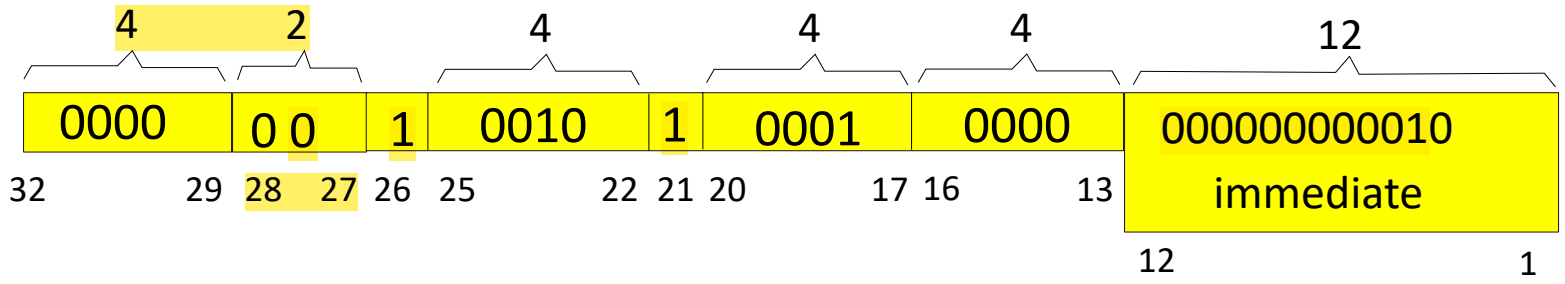
* Data processing instruction type : 00
* I → Immediate bit
* opcode → Instruction code
* S → 'S' suffix bit (for setting the CPSR flags)
* rs, rd → source register, destination register

Opcodes for Data Processing Instructions

Instruction	Opcode	Instruction	Opcode
AND	0000	TST	1000
EOR	0001	TEQ	1001
SUB	0010	CMP	1010
RSB	0011	CMN	1011
ADD	0100	ORR	1100
ADC	0101	MOV	1101
SBC	0110	BIC	1110
RSC	0111	MVN	1111

Example (with condition code)

Encode the Instruction **SUB EQ S R0, R1, #2**



Opcode for EQ is 0000

Opcode for SUB 0010

Output 0x02510002

Conditional Variants of Normal Instructions

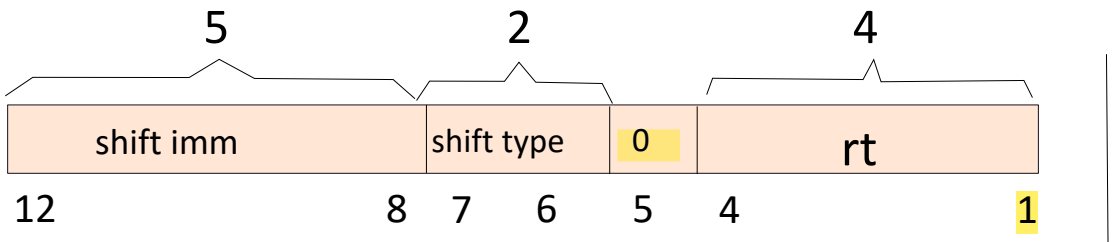
* Normal Instruction + <condition>
  * **Examples** : addeq, subne, etc.
* Also known as predicated instructions
* If the condition is true
  * Execute instruction **normally**
* Otherwise
  * Do not execute at all

Example

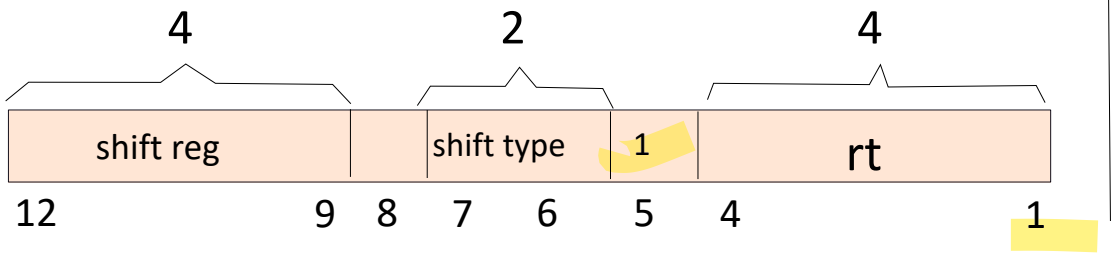
Write a program in ARM assembly to count the number of 1s in a 32 bit number stored in r1. Save the result in r4.
Answer:

```
mov r2, #1 /* idx = 1 */
mov r4, #0 /* count = 0 */
/* start the iterations */
.loop:
    /* extract the LSB and compare */
    and r3, r1, #1
    cmp r3, #1

    /* increment the counter */
    addeq r4, r4, #1

    /* prepare for the next iteration */
    mov r1, r1, lsr #1
    add r2, r2, #1

    /* loop condition */
    cmp r2, #32
    ble .loop
```

Encoding the Shifter Operand



(a)



(b)

Shift type	
lsl	00
lsr	01
asr	10
ror	11

(c)

A shifter operand is of the form rt (lsl | lsr | asr |ror) (shift reg/shift imm)

Encoding Immediate Values

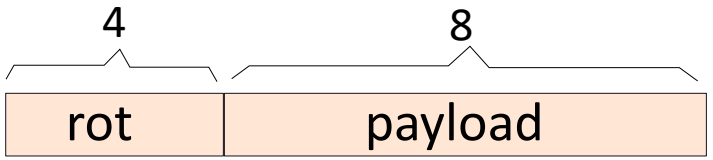
* ARM has 12 bits for immediates

* What do we do with 12 bits ?
  * It is not 1 byte, nor is it 2 bytes

* Let us divide 12 bits into two parts
  * 8 bit payload + 4 bit rot

Encoding Immediate Values Contd.

* The value of the immediate is equal to : payload ror (2 * rot)



* The processor decodes (expands) 12 bits → 32 bits
* Advanced versions of ARM processors (64 bit M/C) support 32 bit immediates

Example

Encode the decimal number 42.

Answer:

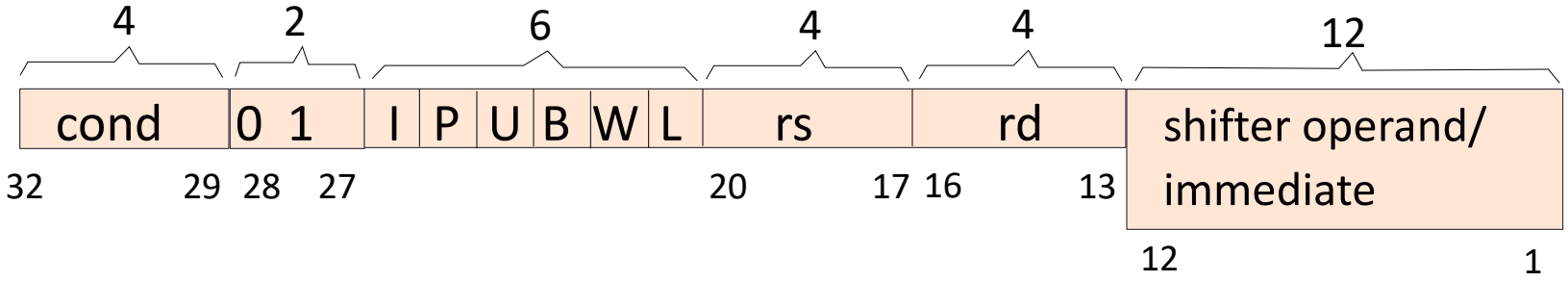
42 in the hex format is 0x2A, or alternatively 0x 00 00 00 2A. There is no right rotation involved. Hence, the immediate field is 0x02A.

Encode the number 0x2A 00 00 00.

Answer:

The number is obtained by right rotating 0x2A by 8 places. Note that we need to right rotate by 4 places for moving a hex digit one position to the right. We need to now divide 8 by 2 to get 4. Thus, the encoding of the immediate: 0x42A

Load and Store Instruction

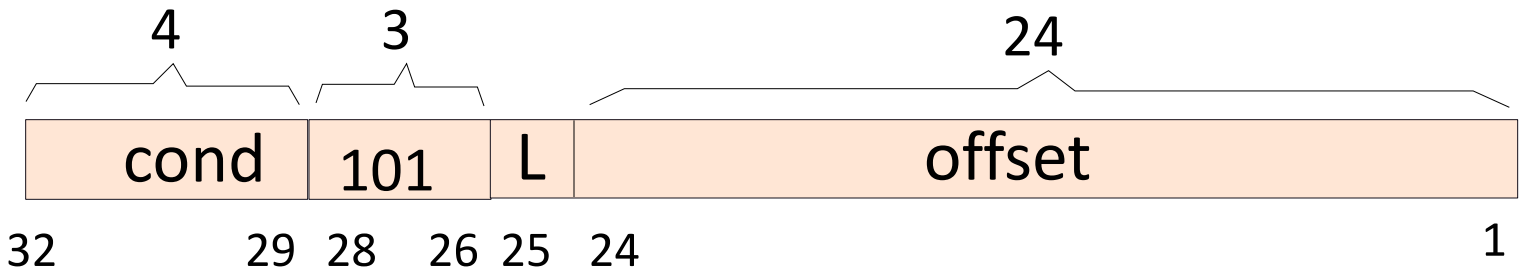


* Memory instruction type : 01
* rs, rd, shifter operand
  * Connotation remains the same
* Immediates are not in (rot + payload format) : They are standard 12 bit unsigned numbers

I, P, U, B, W, and L bits

Bit	Value	Semantics
I	0	Offset is an immediate value
	1	Offset is a register or shifter operand
P	0	add offset after transfer
	1	add offset before transfer
U	0	subtract offset from base
	1	add offset to base
B	0	transfer word
	1	transfer byte
W	0	do not write back to the base
	1	Write back to the base
L	0	store to memory
	1	load from memory

Branch Instructions



* **L** bit → Link bit
* **offset** → branch offset (similar to SimpleRisc)