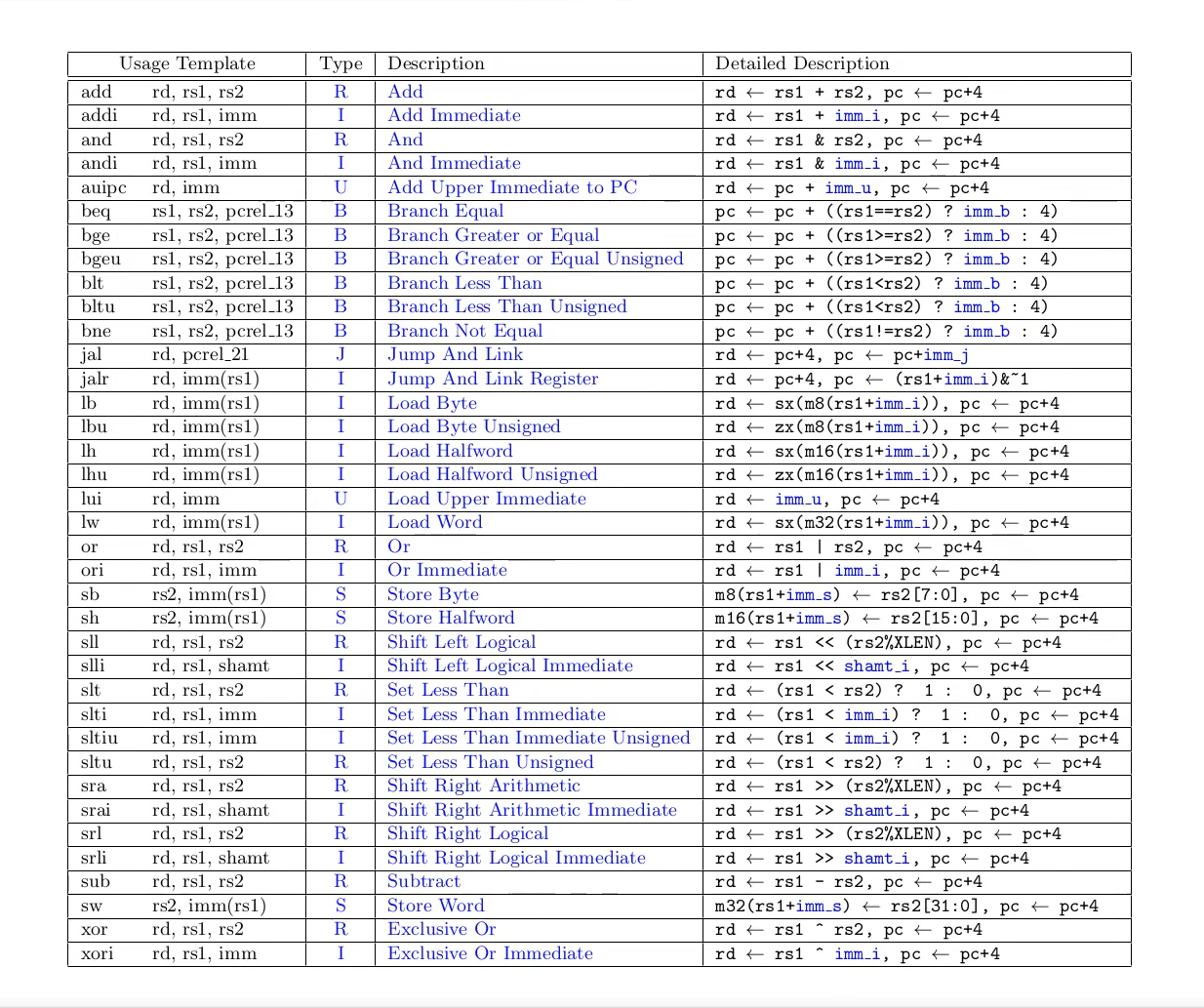
**RISC-V Multicycle Design Specification**

**Proposed by: Navin Kumar Singh**

**Introduction:**

RISC stands for "reduced instruction set computing". RISC-V is the fifth in a series of RISC ISAs from UC Berkeley. It is an open-source ISA specification developed by UC Berkley and the specification and working group is hosted at [www.riscv.org](http://www.riscv.org).

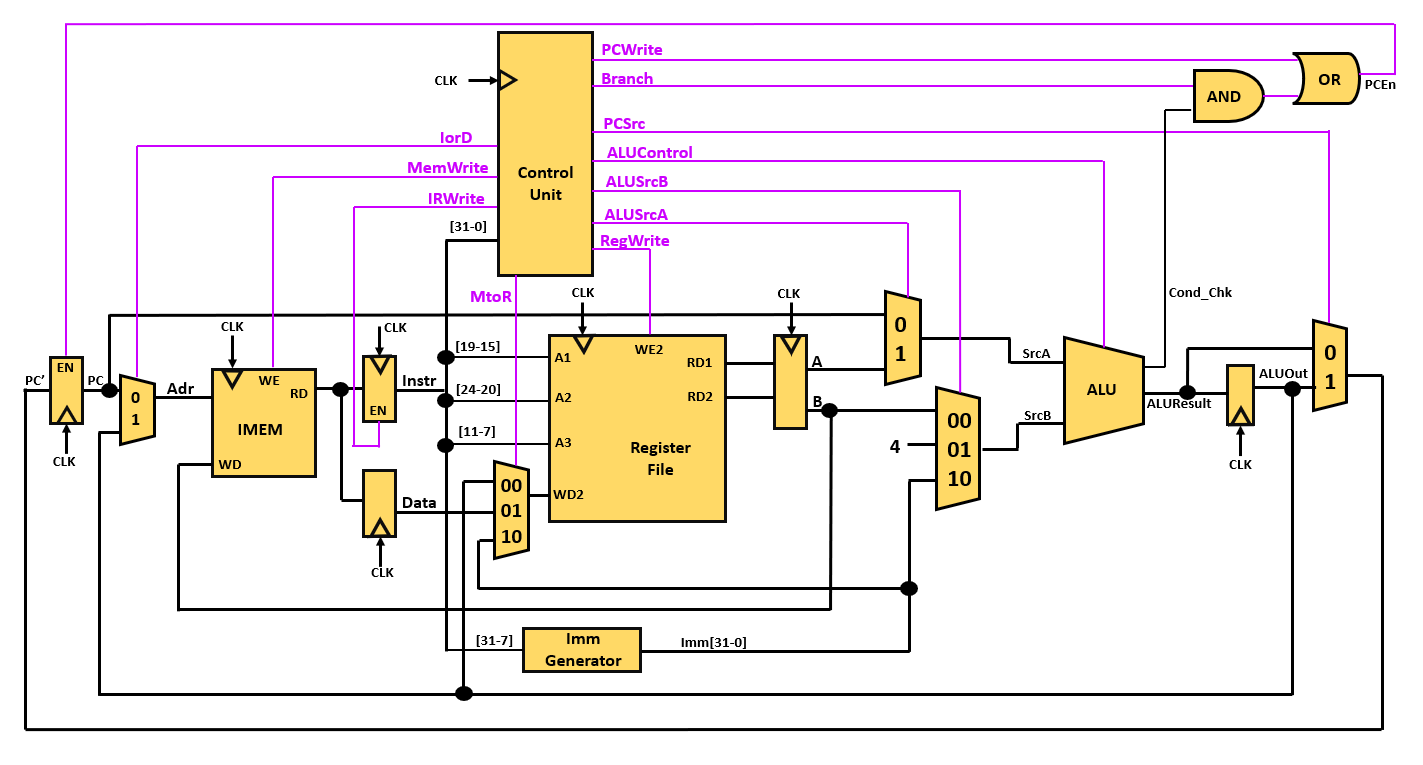
We will implement the core instructions of the base RISC-V instruction set (RV32I), which contains just 47 instructions as listed below:



RISC-V is a *load-store architecture*. The 32-bit implementation of RISCV supports 32 registers each 32 bits wide.

**RISC-V Pipeline:**

A RISC-V pipeline processor is a CPU architecture that processes instructions in a sequential manner, where each instruction passes through five stages of execution. The five stages are: **Instruction Fetch (IF), Instruction Decode (ID),** **Execute (EX),** **Memory (MEM),** and **Writeback (WB).** In this design document, we will outline the high-level design of a 5 Stage RISC-V Pipeline Processor.



1. **Instruction Fetch (IF) Stage:**

The first stage of the pipeline is the instruction fetch (IF) stage. This stage fetches the instruction from memory and stores it in the instruction register (IR).

* PC register holds the address of the next instruction to be fetched.
* The instruction cache stores the instructions to be fetched.
* A multiplexer selects the next instruction to be fetched based on the current value of the PC register.
* The instruction is fetched from the instruction cache and stored in the instruction register (IR).
* The program counter (PC) is incremented by four after each instruction fetch.

1. **Instruction Decode (ID) Stage:**

The second stage is the instruction decode (ID) stage.

* The instruction register (IR) holds the fetched instruction.
* The instruction is decoded to determine the operation to be performed.
* The operands are extracted from the register file.
* The operand values are sent to the execute (EX) stage.

1. **Execute (EX) Stage:**

The third stage is the execute (EX) stage. This stage performs the operation specified by the instruction. For example, if the instruction is an add operation, the EX-stage will perform the addition operation on the operands.

* The operation specified by the instruction is performed.
* The result of the operation is sent to the memory (MEM) stage.

1. **Memory (MEM) Stage:**

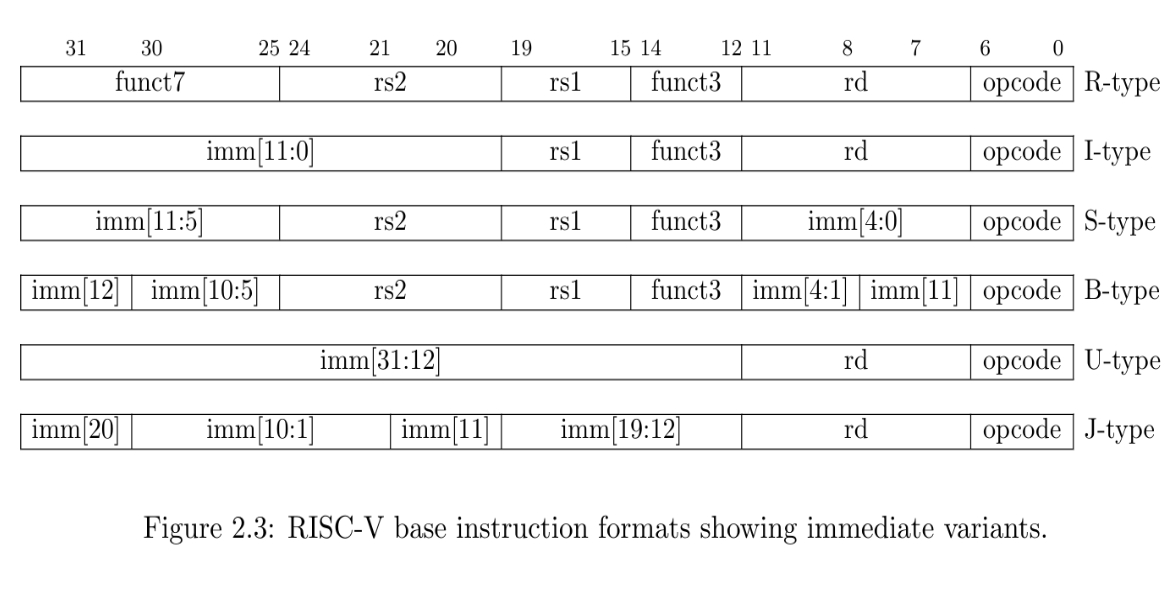
The fourth stage is the memory (MEM) stage. This stage accesses memory to read or write data. For example, if the instruction is a load operation, the MEM stage will read data from memory and store it in a register.

* The memory address is computed based on the result of the execute (EX) stage.
* The data is read from or written to memory.

1. **Writeback (WB) Stage:**

The fifth and final stage is the writeback (WB) stage. This stage writes the results of the operation back to the register file. For example, if the instruction is an add operation, the WB stage will write the result of the addition back to a register.

**RISC-V INSTRUCTION FORMAT:**

****

**Fig: RISC-V Base Instructions types and their formats**

RISC-V instructions provide the following fields:

* **opcode**  
  gives a general classification of the instruction and determines which of the remaining fields are needed, and how they are encoded in the remaining instruction bits.
* **functionfield** **(funct3/funct7)**  
  specifies the exact function performed by the instruction, if not fully specified by the opcode.
* **source\_registers(rs1/rs2)**  
  identifying the register(s) in the register file containing the source operand values on which the instruction operates.
* **destination\_register(rd)**  
  identifying the register into which the instruction’s result is to be written.
* **immediate**  
  value contained within the instruction bits themselves. This value may provide an offset for indexing into memory or a value upon which to operate (in place of the register value indexed by rs2).

All instructions are 32 bits in length. The R-type encoding provides a general layout of the instruction fields used by all instruction types. R-type instructions have no immediate value. Other instruction types use a subset of the R-type fields and provide an immediate value in the remaining bits.

**Instruction Fetch Logic:**

The Instruction Fetch Unit (IFU) is the first block in a Central Processing Unit (CPU) that processes instructions. It is responsible for organizing program instructions to be fetched from memory and executed in the correct order.

The operation of this unit is as follows:

* if **PCSel** is **HIGH** then the **PC** value is updated by **PC+Imm\_Data**.
* If **PCSel** is **LOW** then the **PC** is updated by **PC+4**.

**PCSel** is set to HIGH if any JUMP instruction is encountered. If BRANCH instructions are encountered, **PCSel** is set as HIGH if condition is fulfilled and LOW if not.

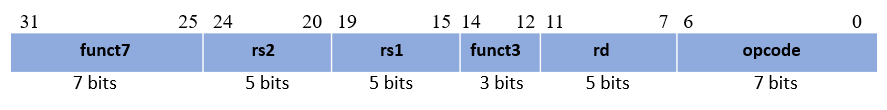
**Instruction Decode Logic:**

It is a part of the pipelining in the CPU Design. It basically takes the instructions from the **Instruction Memory** block. The instruction Memory takes input from the **Fetching Unit** and stores the address of the instruction, reads it and passes the 32-bit instruction code to the decoding block.

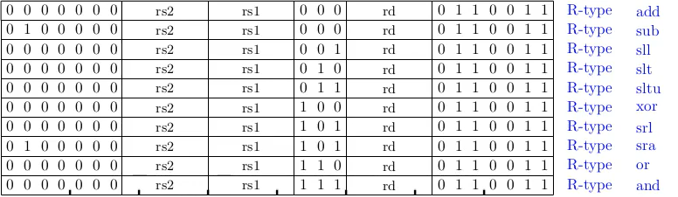
This address is provided by the **Program Counter** which points to the next instruction to be subsequently decoded and executed. Decode Unit accesses the instructions pointed by the program counter and performs the decoding as directed by the control unit.

The information on one example of each type of instruction is given below:

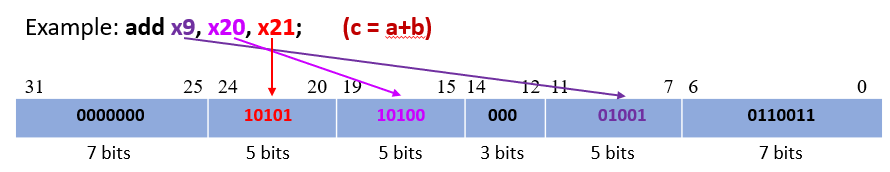
* **R-Type:**



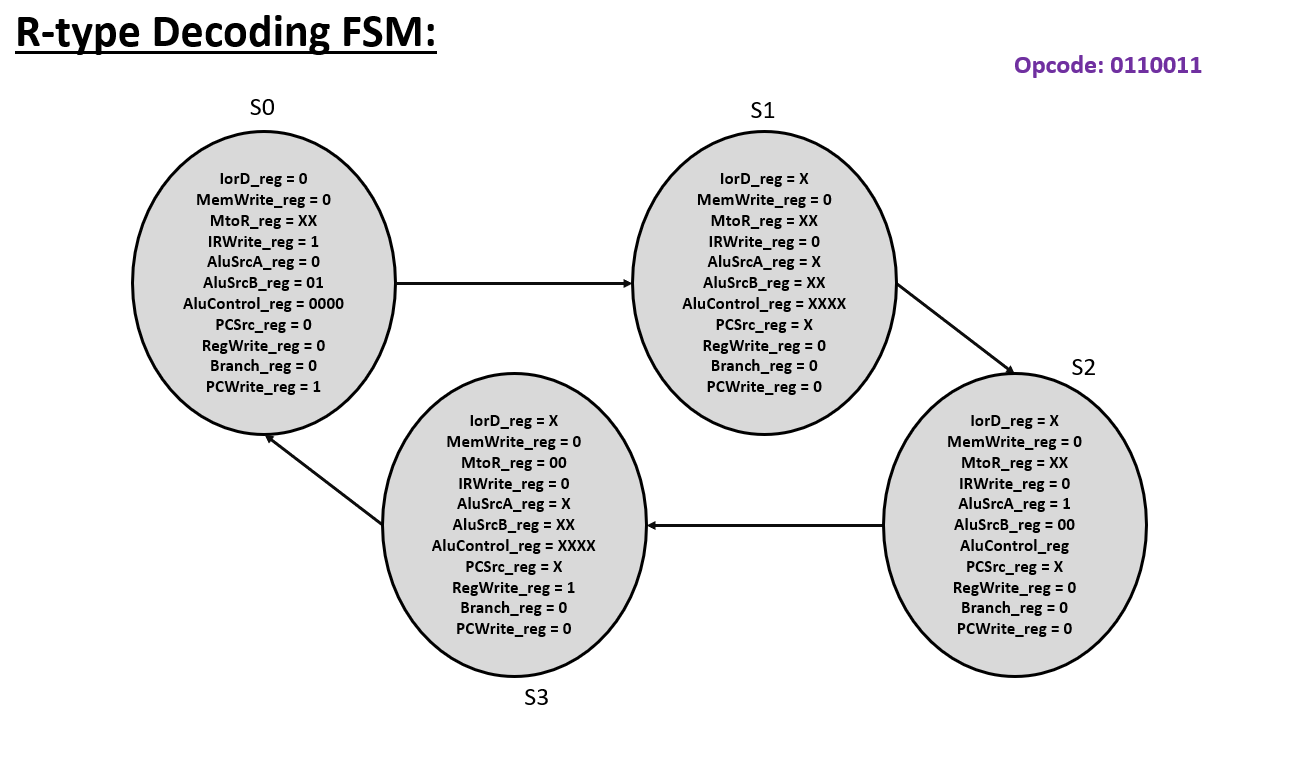
R-type of instructions have their unique **opcode[6-0]** as “0110011”. Next up, based on the values of the **funct7[31-25]** & **funct3[14-12],** the exact operation to be done is identified. A few are shown below.



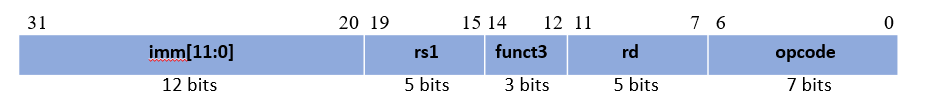
After recognising the exact operation to be done, the index of the source registers is decoded from the instruction fields i.e. here **rs2[24-20]** & **rs1[19-15]** contains the address of the source registers where the data is stored on which, the operation is to be done. Similarly, we decode the destination register’s index i.e. **rd[11-7]** where the result is to be stored after operation completion.



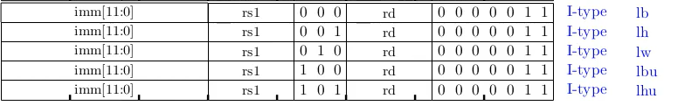
After decoding all the input, it is forwarded to **ALU** which does the operation on the data and store it.

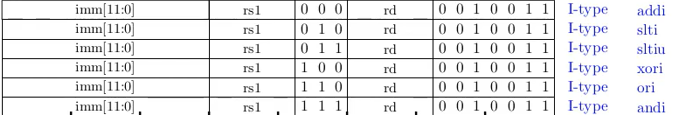


* **I-Type:**

****

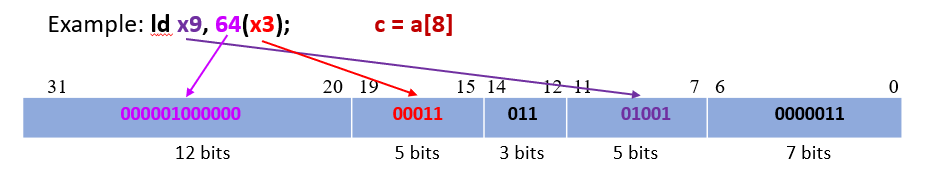
Here opcode values are different according to their respective purpose. Like for **loading instruction** it is “**0000011**”, for **register jump** it is “**1100111**”, for **immediate arithmetic**, **logical operation** and **shift operation** it is “**0010011**” so similarly like R-type we extract values in the **funct3[14-12]** and find the specific operation to be done.



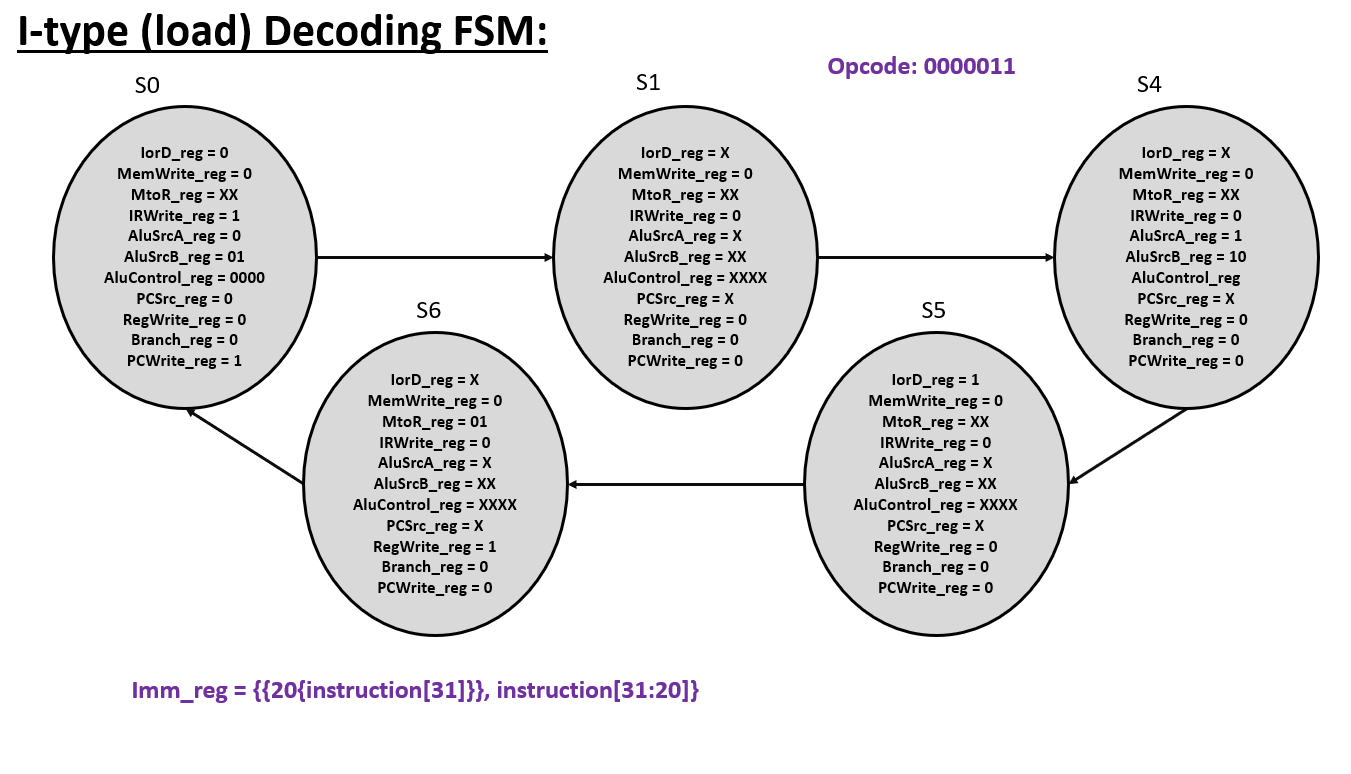


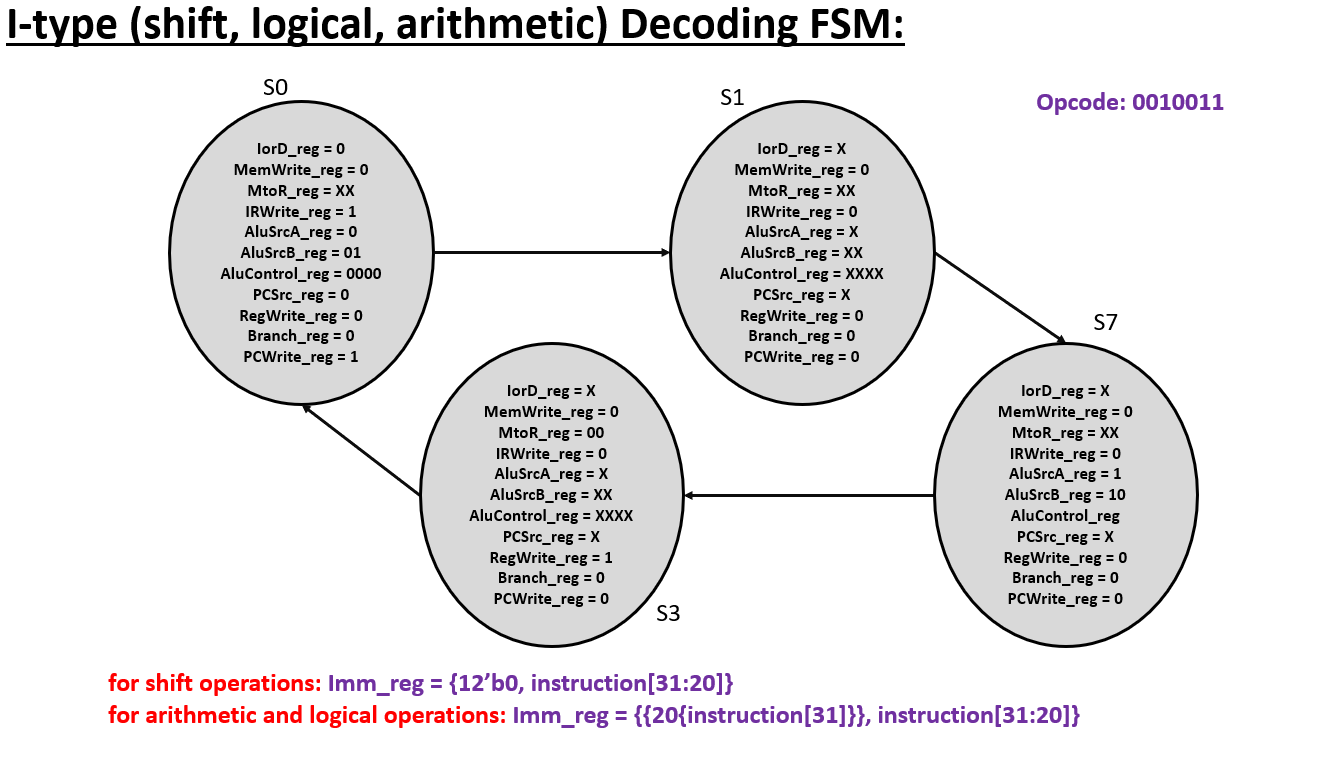


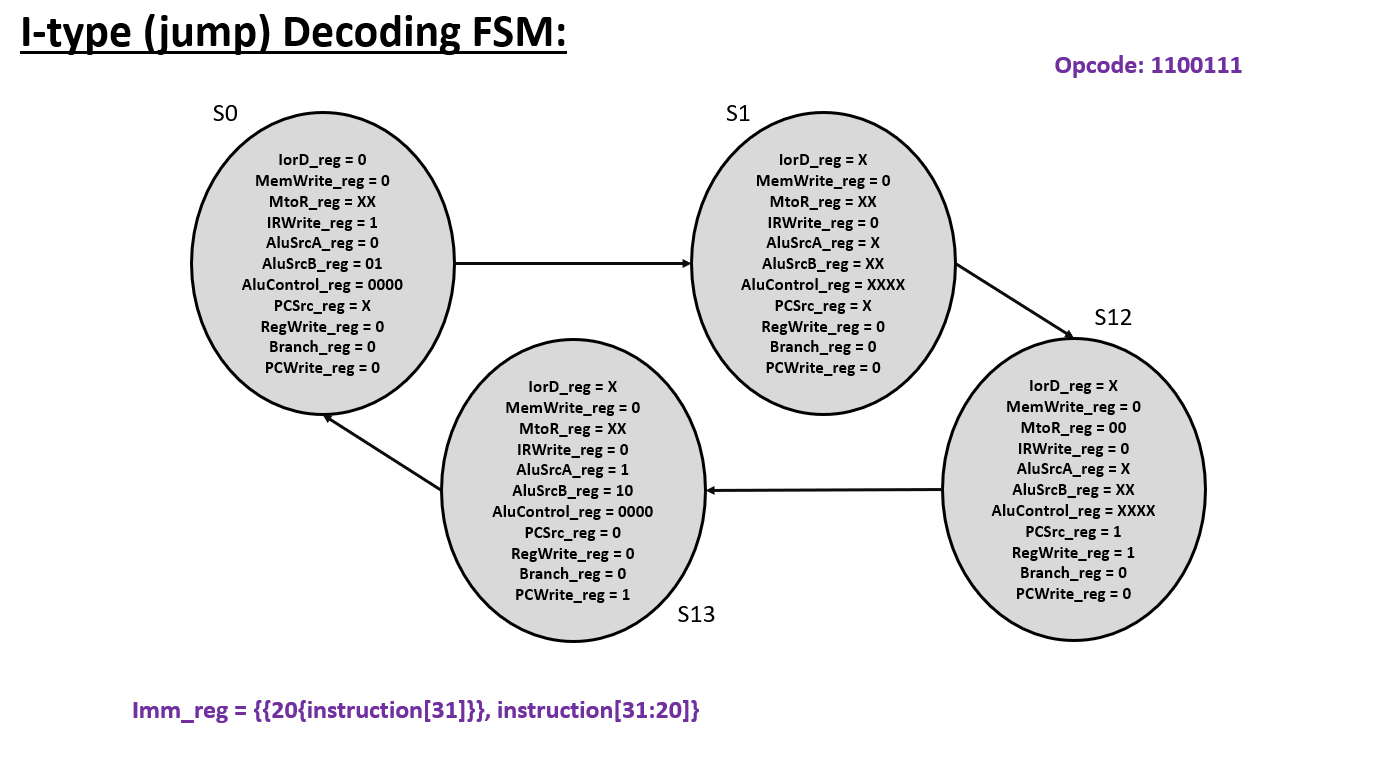
Here, for each instruction the immediate input (data provided in the instruction itself), lies in the position [31-20]. In case of shift operation, it is of 5 bits size depicting the **shift amount[24-20]** by which the data is to be shifted right or left.



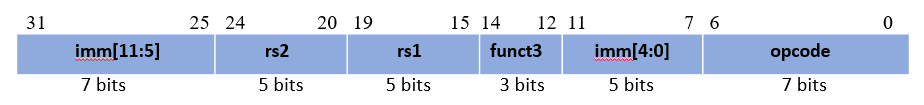
Similar to R-type instructions, we will decode the opcode and find what type of immediate instruction this is and analyse the immediate data accordingly. For the example instruction above, the data present in the address location specified by the sum of “address stored in register **x3** andthe **offset**(64)” has to be copied to the register **x9**. So this specified address is passed to the **ALU** block which will do the operation and copy the data present in that location to the register **x9**.



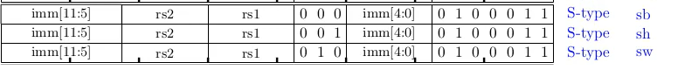


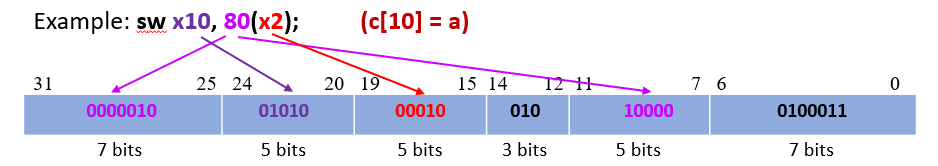


* **S-Type:**

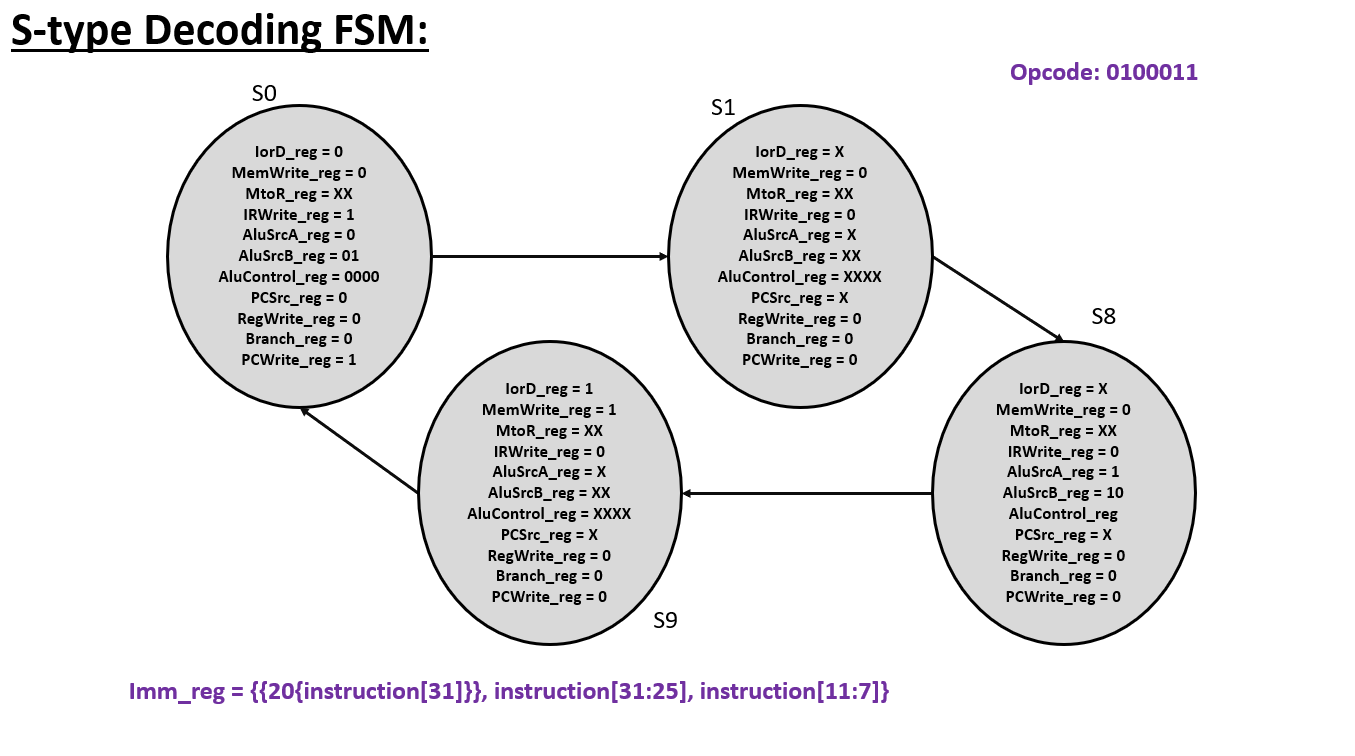
****

S-type are used for storing the data in the immediate destination which is provided in the instruction itself. Its opcode is given by “**0100011**”. Here again opcode is decoded and discriminated. The immediate data part is divided into two blocks i.e. **{imm[31-25], imm[11-7]}** but works as a whole which we can access by compounding the values stored in it.

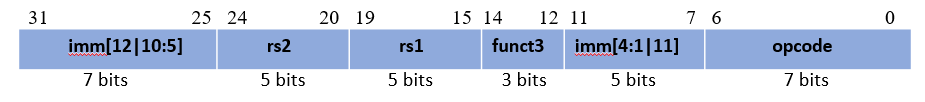




In the example given, the data stored in register **x10** is being copied to the address location pointed by the sum of the address stored in register **x2** and **offset**(80). So based on the **funct3**, we will find the exact operation of store to be done and then, pass the data to the **ALU** block which will operate and store the result in the destination specified accordingly.

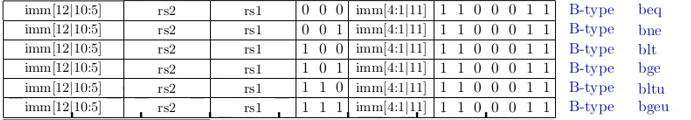


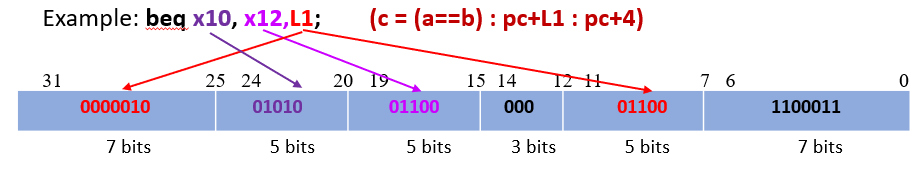
* **B-Type:**

****

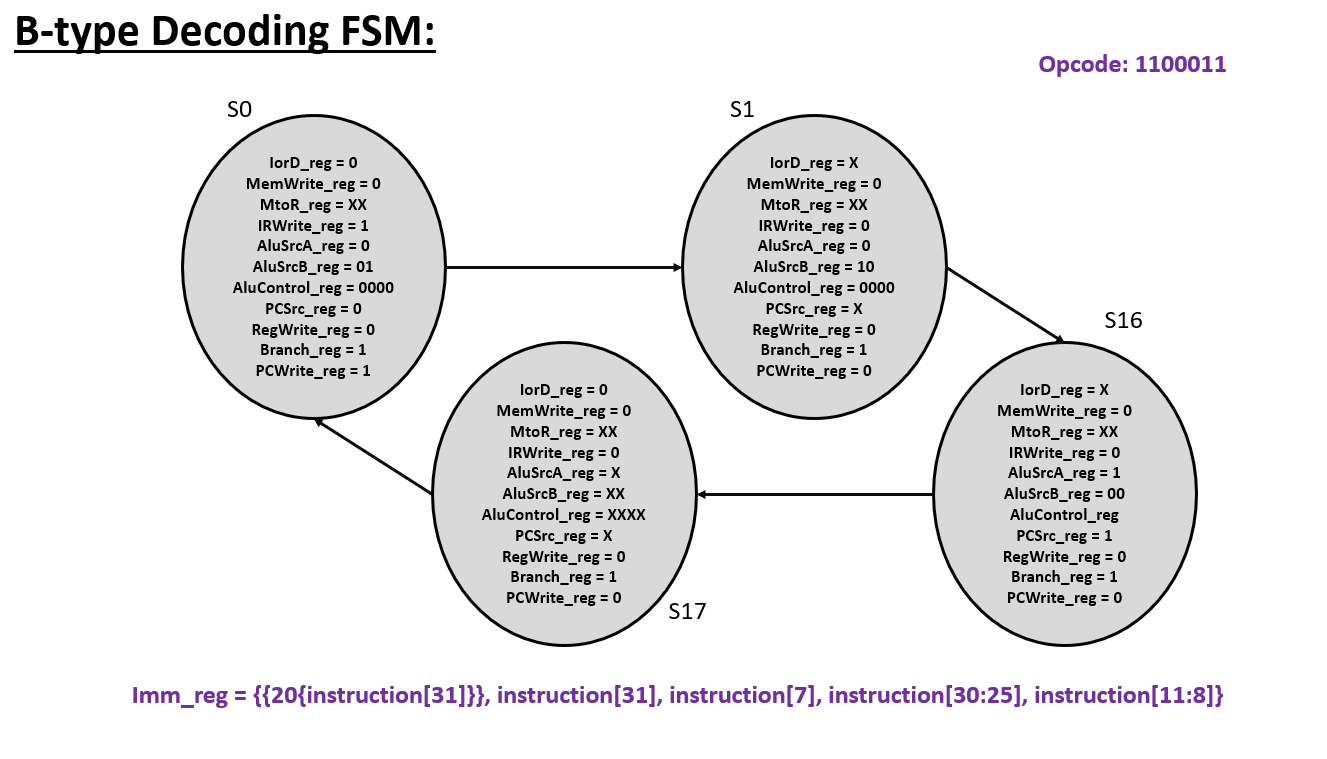
These instructions are used for conditional jump. The order of the immediate data is present in the way represented above. These instructions holds the opcode “**1100011**”.

Applying the same procedure, we decode each of the blocks, like **rs1**, **rs2**, and **funct3** for specific comparison to be made between operands. The immediate data is the address where the **PC** should point for the next instruction, so this offset is provided back to the **program counter** for branching.

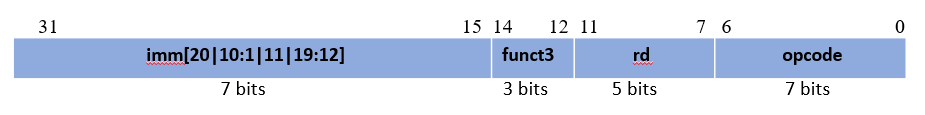




Here, in this example, if data stored in **x10** and **x12** registers are same then the program counter is required to branch to the instruction pointed by the immediate address (**L1**) given in the instruction itself. Based on the **funct3** the exact type of branch instruction is to be decoded and implemented.



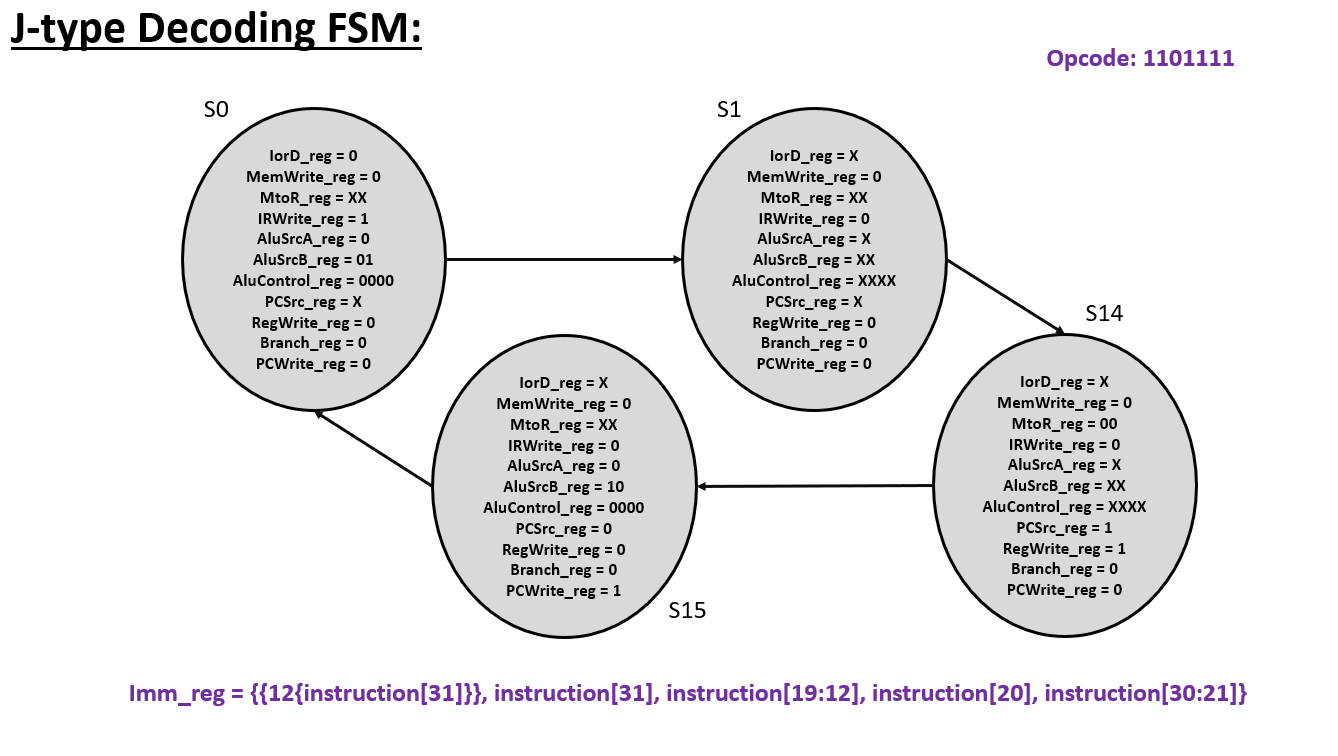
* **J-Type:**

****

This type of instruction is used for unconditional jump. Here opcode value is “**1101111**”. No **funct3** is present as this instruction has no different types. The order of the immediate data is present in the way represented above. This instruction will increment **program counter** by the immediate value present in the instruction. The result is stored in the address pointed by **initial** **PC value + 4**.



So, we record the immediate data part and pass it on to the next stage for incrementing the PC to desired address i.e. **(PC+imm\_data)**.



* **U-Type:**

****

This type of instruction is used for a specific purpose of **“adding upper immediate”** and **“loading upper immediate”** for which opcode values are “**0010111**” or **“0110111”** respectively. No **funct3** is present as this instruction has no different types. The order of the immediate data is present in the way represented above. The **add immediate** instruction will add **program counter** value with the immediate value present in the instruction and save it to the mentioned destination register **rd**. The **load immediate** instruction will save **program counter** value to the mentioned destination register **rd**. The **program counter value** is then changed by **initial** **PC value + 4**.

