Tutorial – 2

Intro to Processor Architecture

- Kaamya Dasika
- Krrish Goenka
- Siddarth Gottumukkula
- Vedant Tejas
- M P Samartha
- Vedant Pahariya

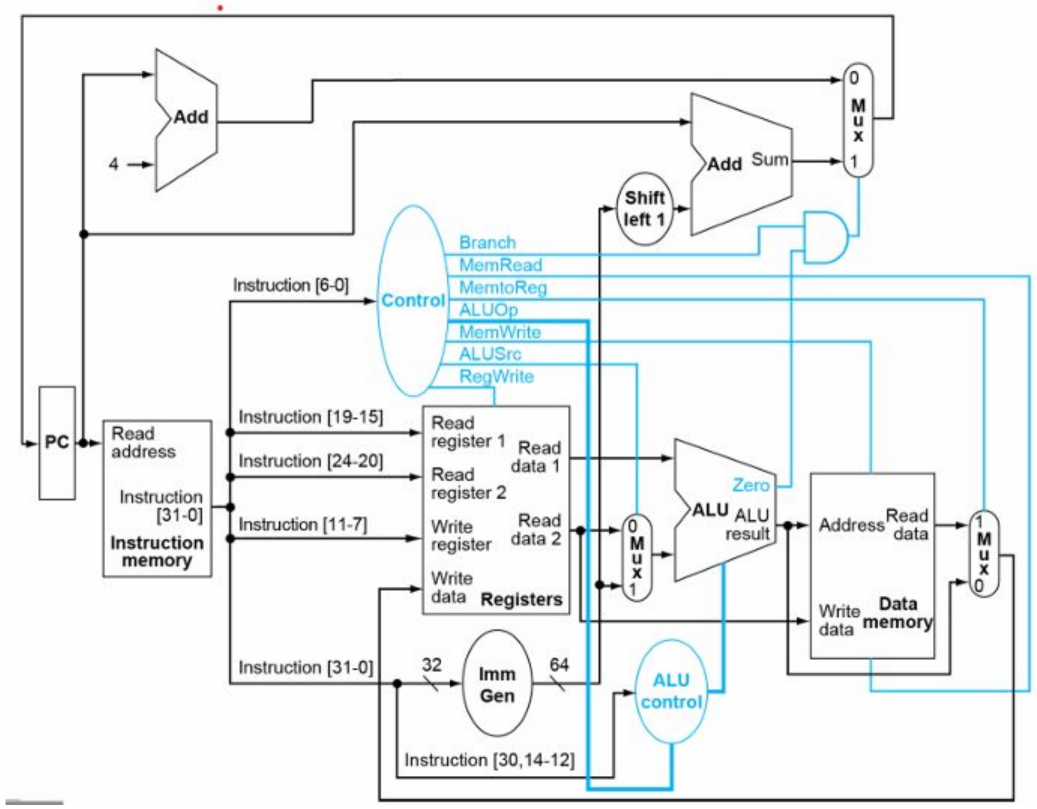
*Order of Names is decided by Mario Kart

Overview

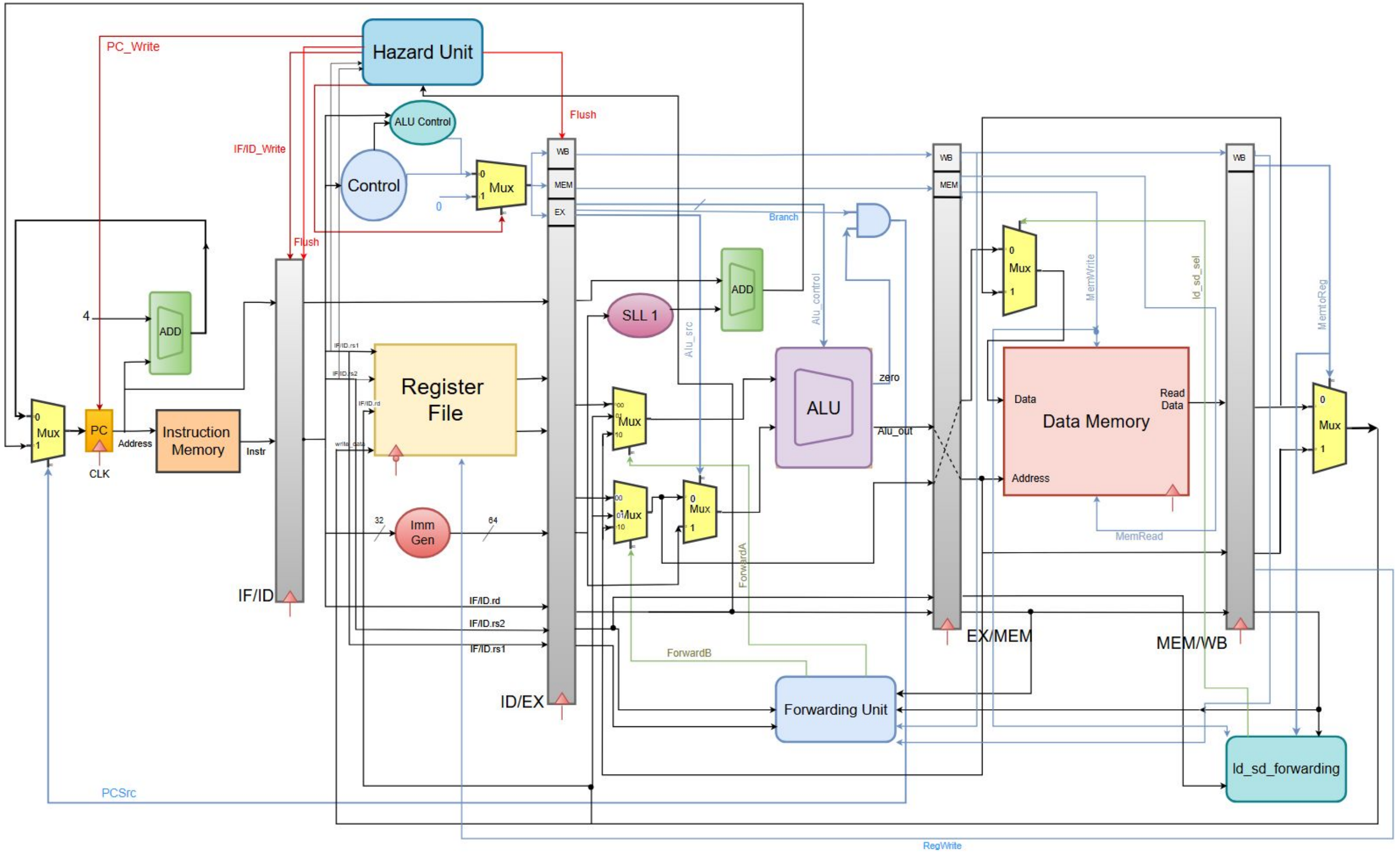
- In the previous tut..
- RISC vs CISC
- Instructions and various formats
- ALU implementation
- Carry vs Overflow flags
- Assembly language

In the previous tut..

Sequential Design

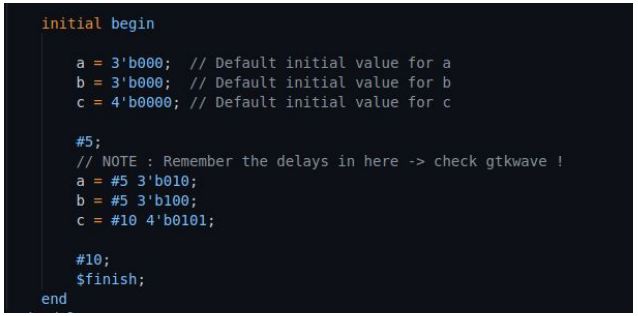


Pipelined Processor

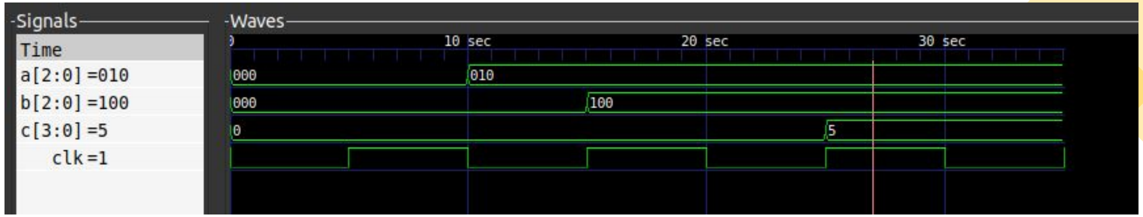


Blocking vs Non-blocking statement {Example}

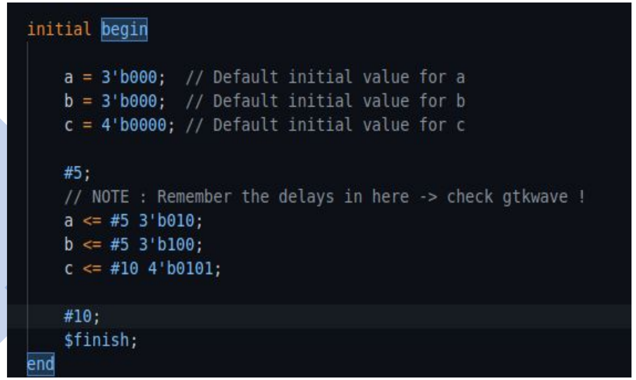
1. Blocking



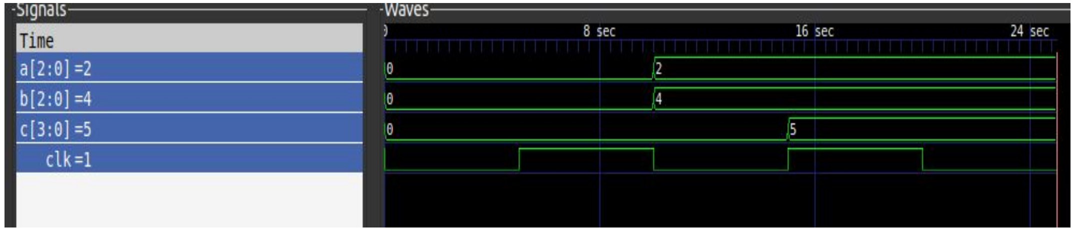
```
initial begin

    a = 3'b000;  // Default initial value for a
    b = 3'b000;  // Default initial value for b
    c = 4'b0000; // Default initial value for c

    #5;
    // NOTE : Remember the delays in here -> check gtkwave !
    a = #5 3'b010;
    b = #5 3'b100;
    c = #10 4'b0101;

    #10;
    $finish;
end
```



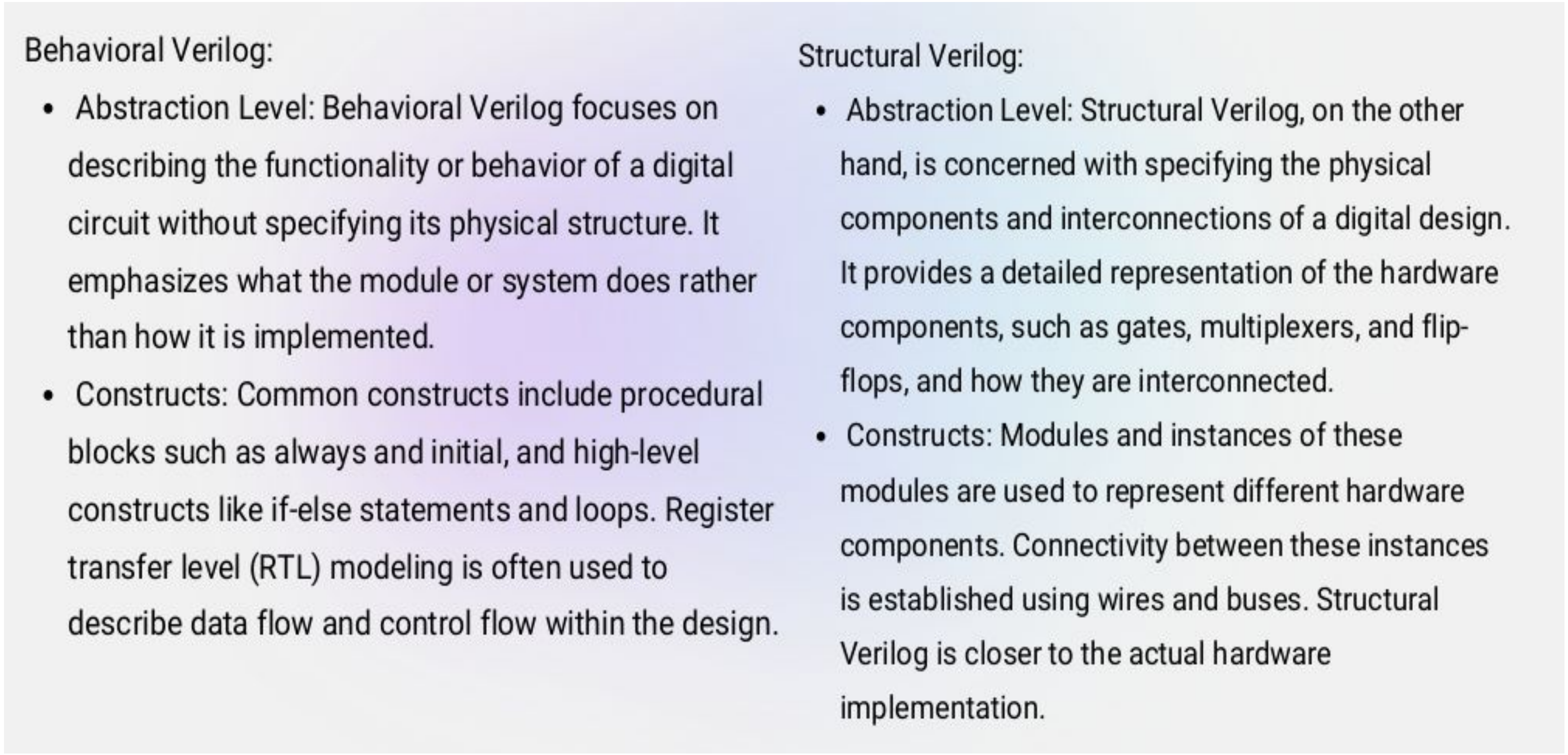
2. Non blocking



```
initial begin

    a = 3'b000;  // Default initial value for a
    b = 3'b000;  // Default initial value for b
    c = 4'b0000; // Default initial value for c

    #5;
    // NOTE : Remember the delays in here -> check gtkwave !
    a <= #5 3'b010;
    b <= #5 3'b100;
    c <= #10 4'b0101;

    #10;
    $finish;
end
```



Behavioural vs Structural Code



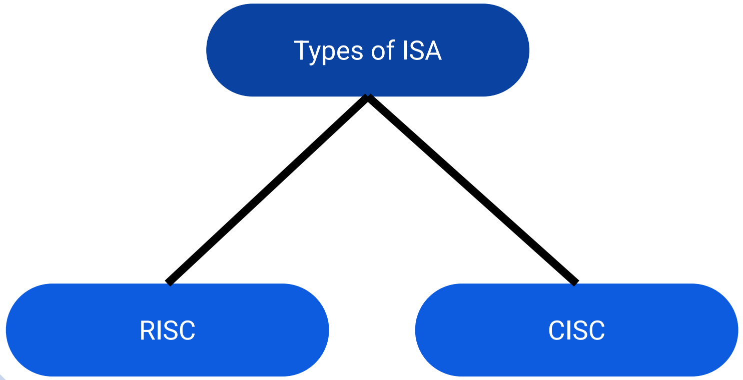
Behavioral Verilog:

- Abstraction Level: Behavioral Verilog focuses on describing the functionality or behavior of a digital circuit without specifying its physical structure. It emphasizes what the module or system does rather than how it is implemented.
- Constructs: Common constructs include procedural blocks such as always and initial, and high-level constructs like if-else statements and loops. Register transfer level (RTL) modeling is often used to describe data flow and control flow within the design.

Structural Verilog:

- Abstraction Level: Structural Verilog, on the other hand, is concerned with specifying the physical components and interconnections of a digital design. It provides a detailed representation of the hardware components, such as gates, multiplexers, and flip-flops, and how they are interconnected.
- Constructs: Modules and instances of these modules are used to represent different hardware components. Connectivity between these instances is established using wires and buses. Structural Verilog is closer to the actual hardware implementation.

Instruction Set Architecture (ISA)



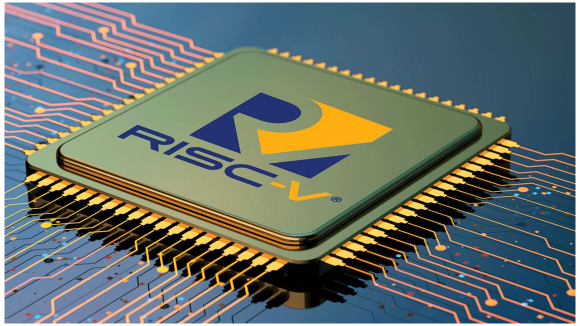
What is an ISA?

Just an interface between hardware and software defining a set of instructions a processor can execute

Components:

Operations, data types, registers, addressing modes and memory access mechanisms used by the CPU

Reduced Instruction Set Computer (RISC)



- **Fixed-length instructions:** Instruction fetching and decoding simpler and faster.

- **Load/store architecture:** Only ld/store instructions access memory, arithmetic operations occur only between registers.

- **Compiler is simpler:** Since fewer instruction types and a more straightforward design, the compiler can be simpler and more efficient

Complex Instruction Set Computer (CISC)

- **Variable-length instructions:** More flexibility but complex instruction decoding process.

- **Memory-to-memory operations:** Instructions can directly access memory locations, reducing the number of instructions required for complex operations.

- **Compiler is complex:** The compiler for CISC architectures tends to be more complex compared to RISC.

Instructions & its formats

Registers in RISC V

x0: the constant value 0

x1: return address

x2: stack pointer

x3: global pointer

x4: thread pointer

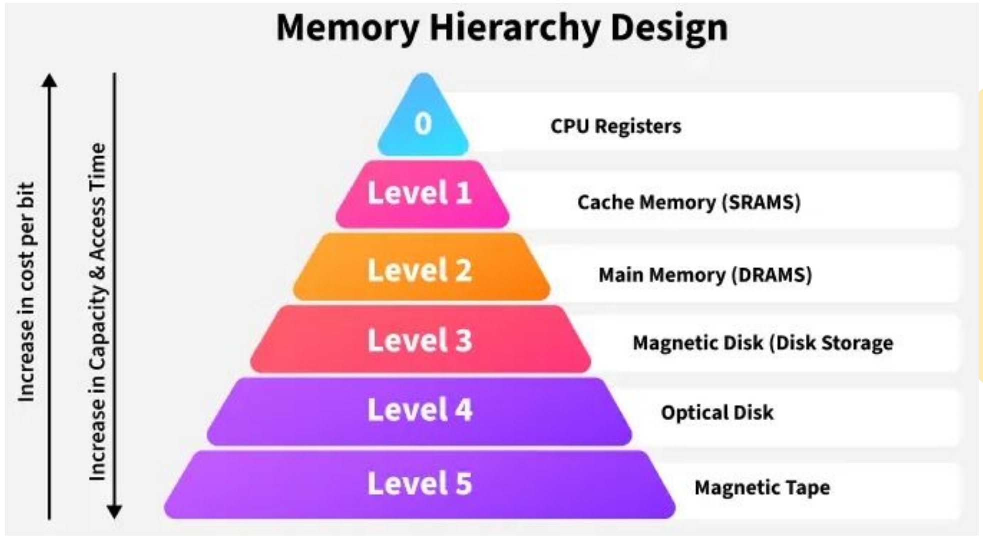
x5 – x7, x28 – x31: temporaries

x8: frame pointer

x9, x18 – x27: saved registers

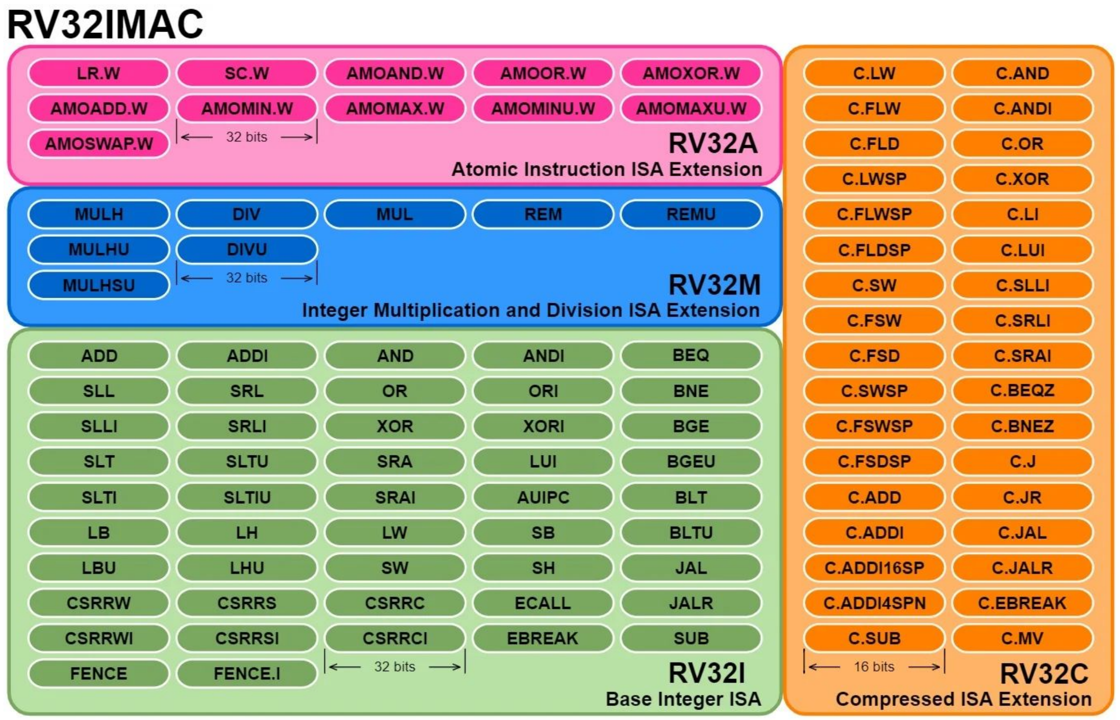
x10 – x11: function arguments/results

x12 – x17: function arguments



- Registers are faster to access than memory.
- For operating in memory we need load and stores but between registers, we don't need to (less instructions to be executed)
- Better to include more frequently used variables inside registers

Instruction field



opcode: operation code

rd: destination register

funct3: 3-bit function code

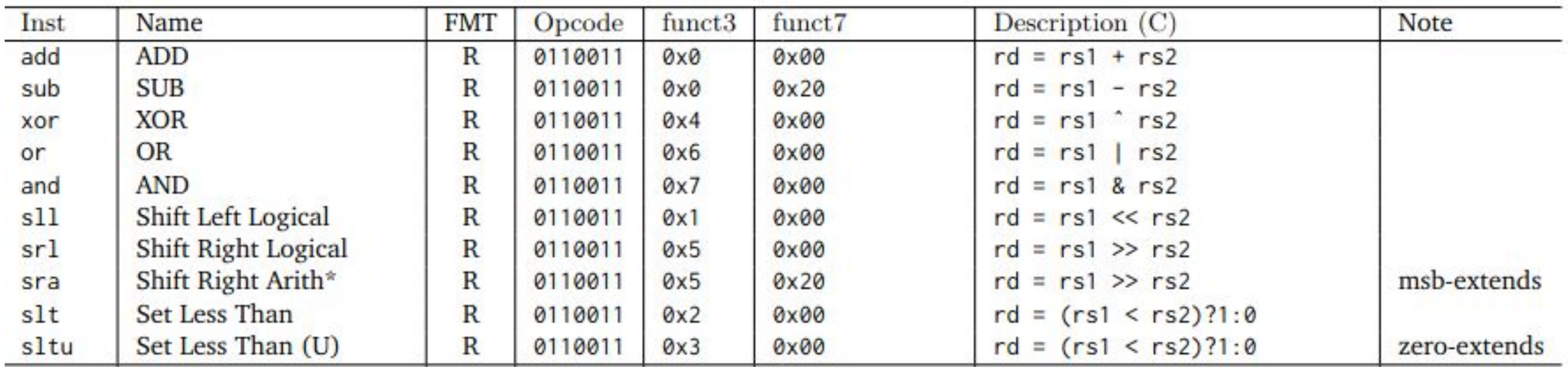
rs1: the first source register

rs2: the second source register

funct7: 7-bit function code

R format instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

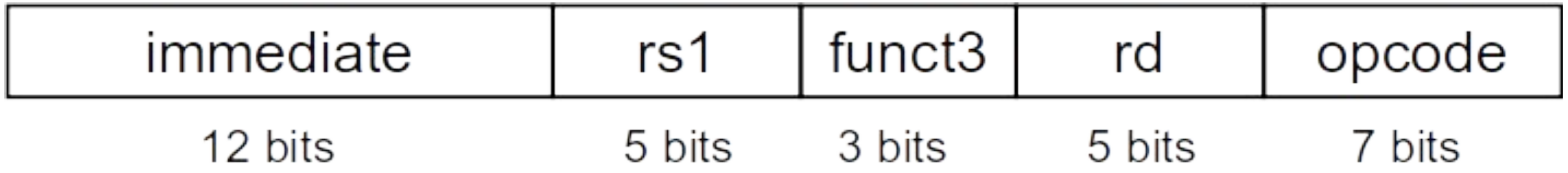


Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

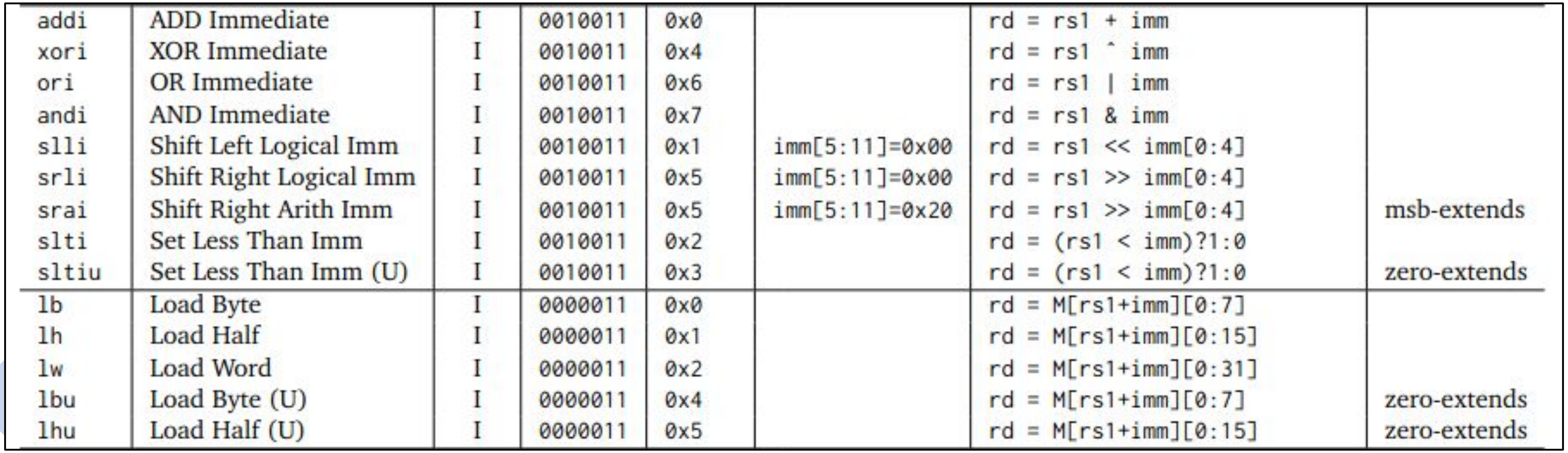
Also called register type instruction because all the operands involved are in registers.

These instructions perform arithmetic and logical operations entirely within the CPU without direct memory access.

I format instructions



immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits



addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends

I format instructions

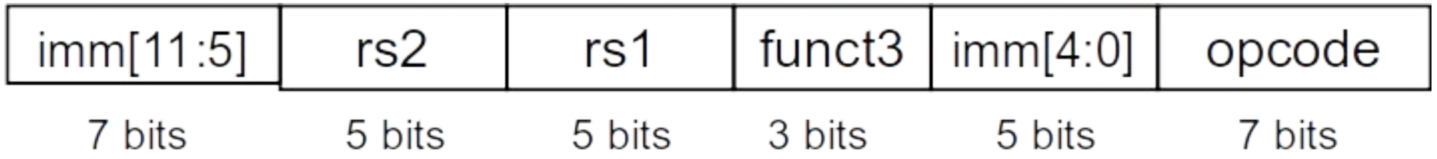
immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

I format instructions are called immediate type instructions because one of the operand is just an immediate constant.

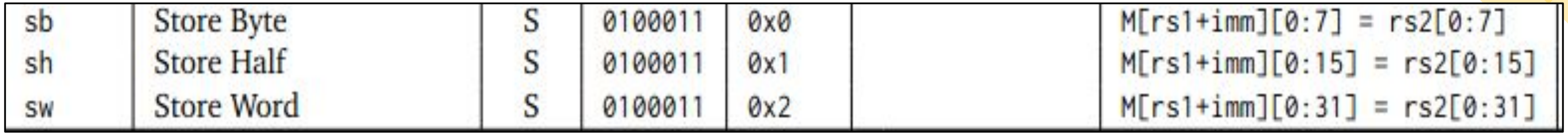
Unlike R type instructions here we only use 1 source register, one destination register and a 12 bit immediate value

The advantage of i type instruction is that they reduce the need of an extra register and therefore additional instructions are not required.

S format instructions



imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits



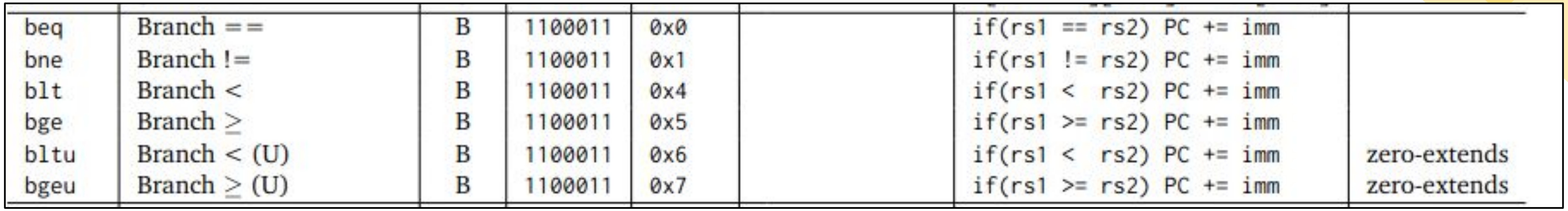
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]

S type instructions are also called store type because they are used to store the data from register into the memory.

They don’t write the result to rd. They just specify rs1(base mem address) rs2 (data to be stored) and the 12 bit immediate value that holds the offset that has to be added to the base address.

B type instructions

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits



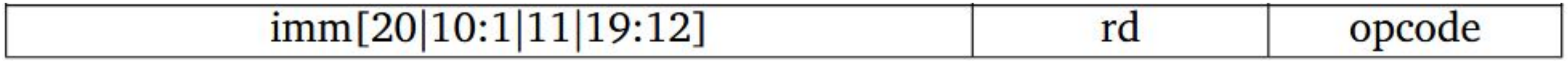
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends

B type instructions are called branch type instructions because they are used to change the flow of the program based on a condition

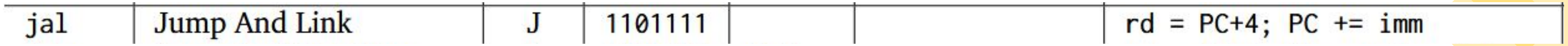
Takes rs1 and rs2 and depending on the condition they branch to a target instruction

We specify the target instruction based on the immediate offset

J type instructions



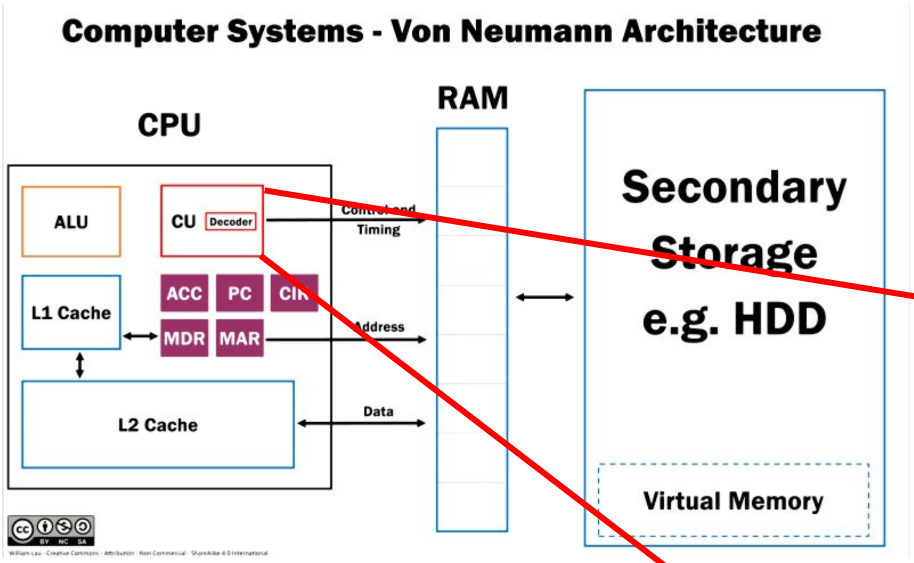
imm[20|10:1|11|19:12]	rd	opcode

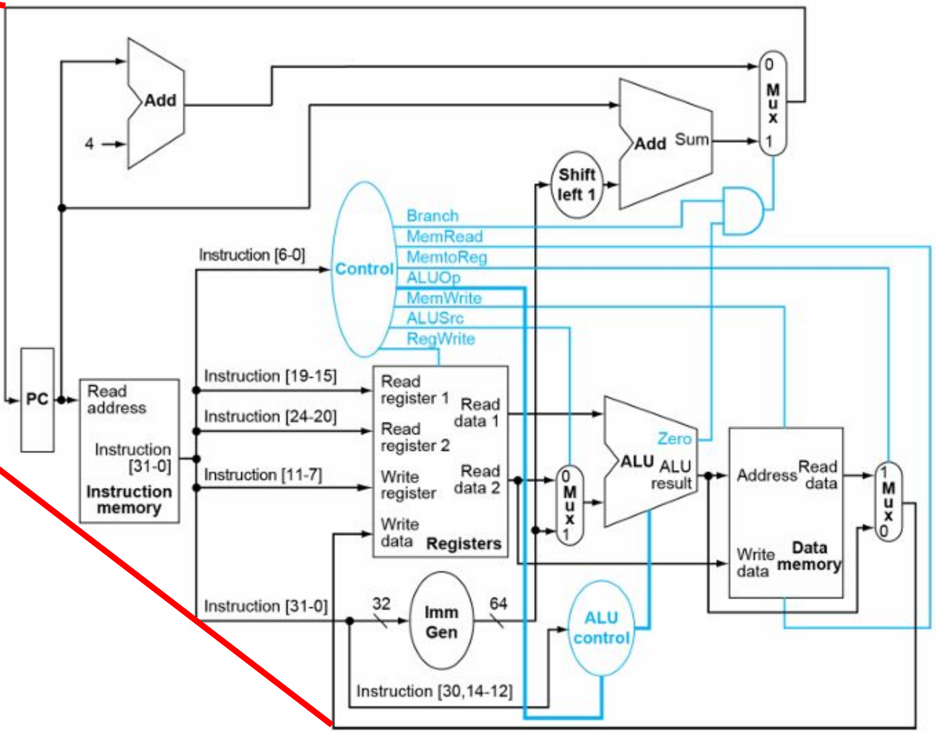


jal	Jump And Link	J	1101111			rd = PC+4; PC += imm

- Only jal uses J-type format (Jump And Link)
- Purpose: unconditional PC-relative jump + optionally save return address (rd = PC + 4)
- Immediate size: 20-bit signed immediate
- Common usage: "*jal ra, label*" for function calls, and "*jal x0, label"* for plain jump (no link)

ALU Implementation


Computer Systems - Von Neumann Architecture
CPU
ALU
CU
Decoder
Control and Timing
L1 Cache
ACC
PC
CIR
MDR
MAR
Address
L2 Cache
Data
RAM
Secondary Storage e.g. HDD
Virtual Memory
William Lau - Creative Commons - Attribution - Non Commercial - ShareAlike 4.0 International


Add
4
Add
Sum
Shift left 1
Mux
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [6-0]
PC
Read address
Instruction [31-0]
Instruction memory
Instruction [19-15]
Instruction [24-20]
Instruction [11-7]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
Instruction [31-0]
32
Imm Gen
64
ALU control
Instruction [30,14-12]

Carry

- Only relevant for unsigned arithmetic.
- Broadly two cases where this flag is set -

  1. Carry when numbers are added :
  >>1111 + 0001 = 1 (carry) + 0000

  2. Carry when numbers are subtracted :
  >>0000 - 0001 = 1(borrow) + 1111
- Otherwise, the carry flag is turned off -
  0111 + 0001 = 1000
  1000 - 0001 = 0111
- Is irrelevant for signed arithmetic.
- **How do you determine carry flag?**

Overflow

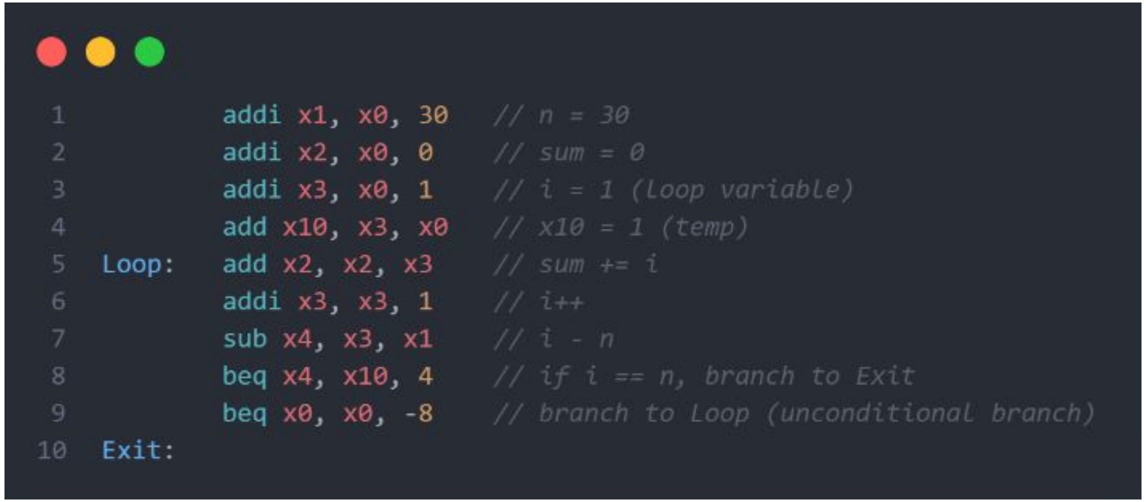
- Only relevant for signed arithmetic.
- Broadly two cases where this flag is set -

  1. Addition of two positive numbers resulted in a negative number :
  >>0100 + 0100 = 1000

  2. Addition of two negative numbers resulted in a positive number :
  >>1000 + 1000 = 0000
- Otherwise the overflow flag is turned off. Equivalently, irrelevant for unsigned arithmetic.
- **How do you determine overflow flag?**

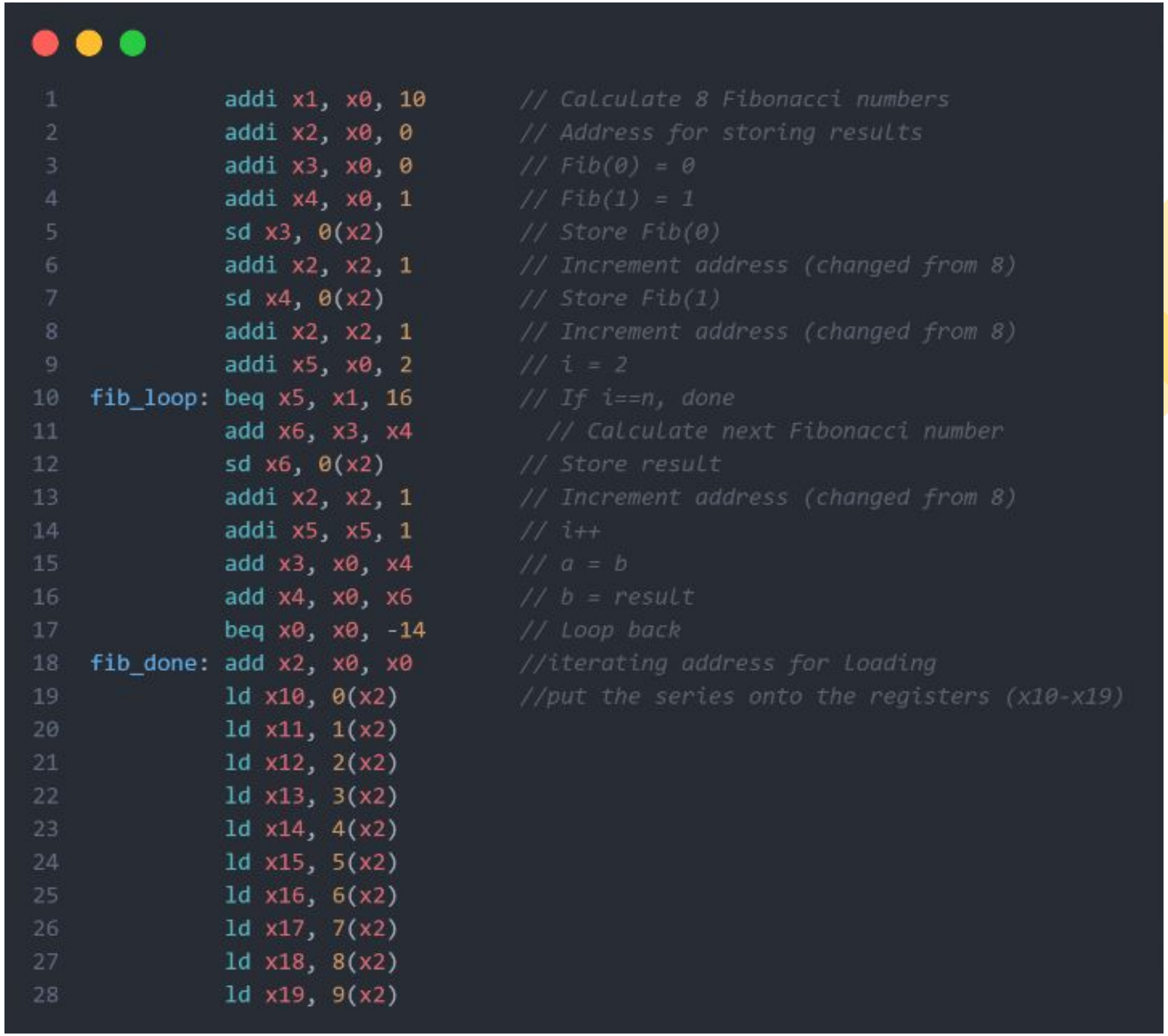
Assembly language

1. Sum of first N natural numbers



```
        addi x1, x0, 30    // n = 30
        addi x2, x0, 0     // sum = 0
        addi x3, x0, 1     // i = 1 (Loop variable)
        add x10, x3, x0    // x10 = 1 (temp)
Loop:   add x2, x2, x3     // sum += i
        addi x3, x3, 1     // i++
        sub x4, x3, x1     // i - n
        beq x4, x10, 4     // if i == n, branch to Exit
        beq x0, x0, -8     // branch to Loop (unconditional branch)
Exit:
```

2. Fibonacci Seq.

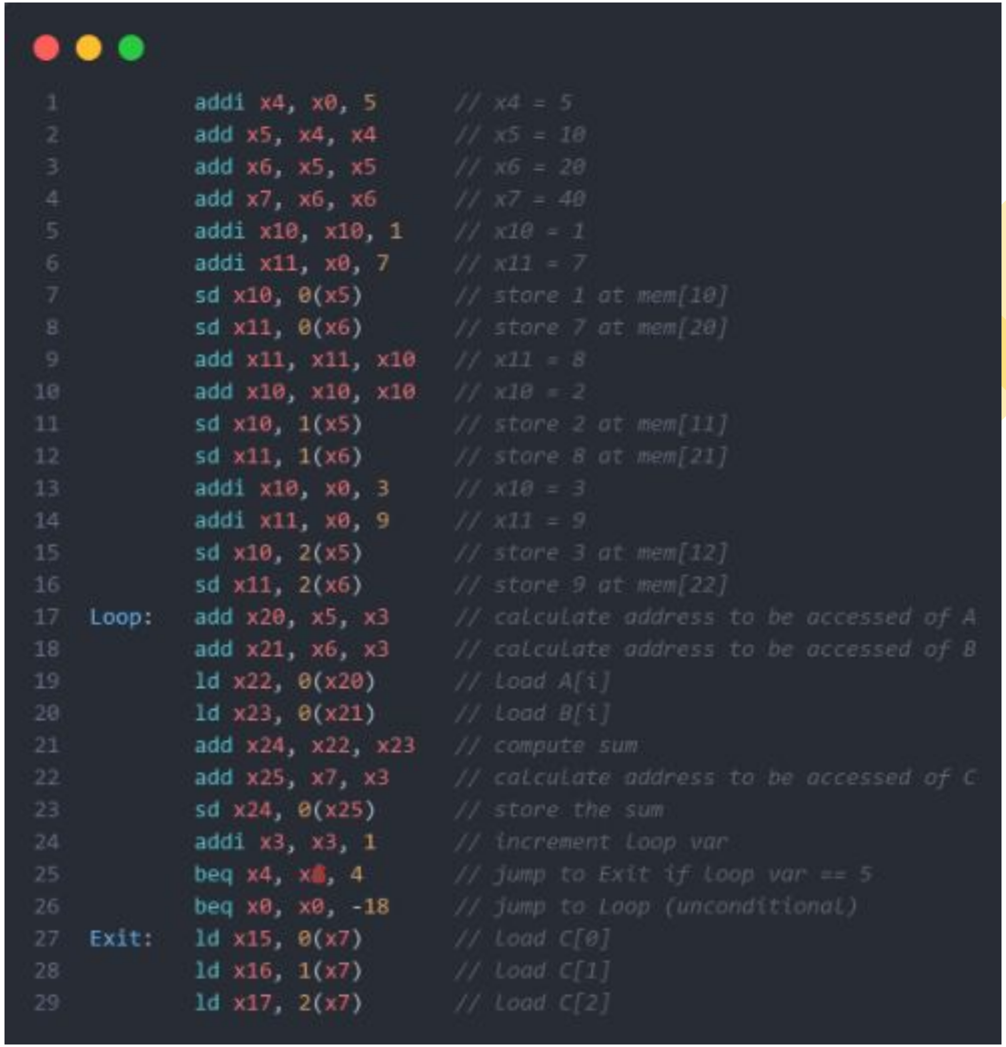


```
          addi x1, x0, 10       // Calculate 8 Fibonacci numbers
          addi x2, x0, 0        // Address for storing results
          addi x3, x0, 0        // Fib(0) = 0
          addi x4, x0, 1        // Fib(1) = 1
          sd x3, 0(x2)          // Store Fib(0)
          addi x2, x2, 1        // Increment address (changed from 8)
          sd x4, 0(x2)          // Store Fib(1)
          addi x2, x2, 1        // Increment address (changed from 8)
          addi x5, x0, 2        // i = 2
fib_loop: beq x5, x1, 16        // If i==n, done
          add x6, x3, x4          // Calculate next Fibonacci number
          sd x6, 0(x2)          // Store result
          addi x2, x2, 1        // Increment address (changed from 8)
          addi x5, x5, 1        // i++
          add x3, x0, x4        // a = b
          add x4, x0, x6        // b = result
          beq x0, x0, -14       // Loop back
fib_done: add x2, x0, x0        //iterating address for loading
          ld x10, 0(x2)         //put the series onto the registers (x10-x19)
          ld x11, 1(x2)
          ld x12, 2(x2)
          ld x13, 3(x2)
          ld x14, 4(x2)
          ld x15, 5(x2)
          ld x16, 6(x2)
          ld x17, 7(x2)
          ld x18, 8(x2)
          ld x19, 9(x2)
```

3. Vector Addition

N = 5
A = [1, 2, 3, 0, 0] @ 10
B = [7, 8, 9, 0, 0] @ 20

Sum = [8, 10, 12, 0, 0] @ 40



```
        addi x4, x0, 5          // x4 = 5
        add x5, x4, x4          // x5 = 10
        add x6, x5, x5          // x6 = 20
        add x7, x6, x6          // x7 = 40
        addi x10, x10, 1        // x10 = 1
        addi x11, x0, 7         // x11 = 7
        sd x10, 0(x5)           // store 1 at mem[10]
        sd x11, 0(x6)           // store 7 at mem[20]
        add x11, x11, x10       // x11 = 8
        add x10, x10, x10       // x10 = 2
        sd x10, 1(x5)           // store 2 at mem[11]
        sd x11, 1(x6)           // store 8 at mem[21]
        addi x10, x0, 3         // x10 = 3
        addi x11, x0, 9         // x11 = 9
        sd x10, 2(x5)           // store 3 at mem[12]
        sd x11, 2(x6)           // store 9 at mem[22]
Loop:   add x20, x5, x3         // calculate address to be accessed of A
        add x21, x6, x3         // calculate address to be accessed of B
        ld x22, 0(x20)          // load A[i]
        ld x23, 0(x21)          // load B[i]
        add x24, x22, x23       // compute sum
        add x25, x7, x3         // calculate address to be accessed of C
        sd x24, 0(x25)          // store the sum
        addi x3, x3, 1          // increment loop var
        beq x4, x3, 4           // jump to Exit if loop var == 5
        beq x0, x0, -18         // jump to Loop (unconditional)
Exit:   ld x15, 0(x7)           // load C[0]
        ld x16, 1(x7)           // load C[1]
        ld x17, 2(x7)           // load C[2]
```

References

1. Computer Architecture: A Quantitative Approach by Patterson, Hennessy and Kozyrakis 7th edition
2. Carry vs Overflow
3. Immediate format in instructions

Any Questions ?