Language of the Computer

Debiprasanna Sahoo
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Roorkee

Content

Book

Computer Organization and Design: The Hardware/Software Interface- RISC-V Edition, 5th Edition, 2017

Chapter-2

Computer Architecture: A Quantitative Approach

Appendix-A

David A. Patterson and John L. Henessey

Manual

The RISC-V Instruction Set Manual

Volume I: User-Level ISA

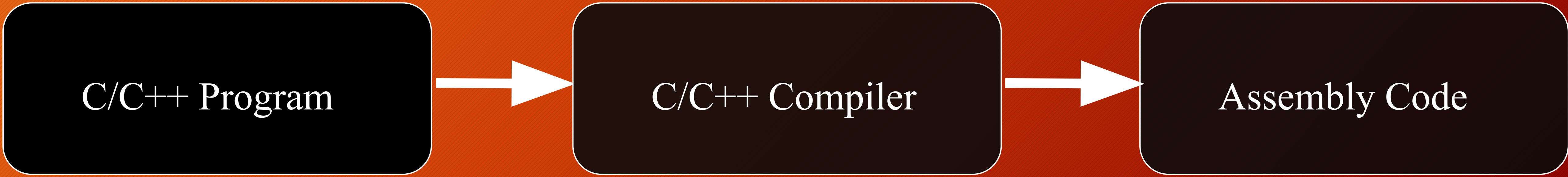
Document Version 2.2

Andrew Waterman and Krste Asanovi

*Image from the book and manual unless specified

Instruction Set

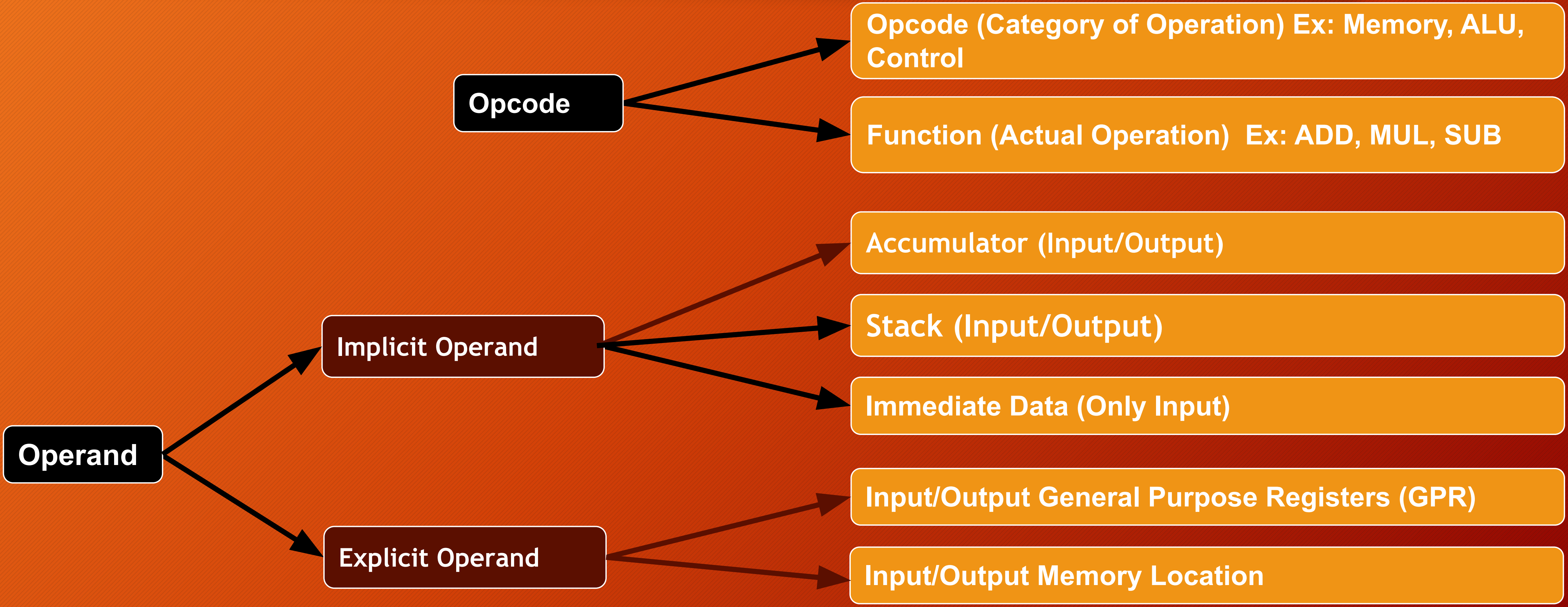
The vocabulary of commands understood by a given architecture



Stored Program Concept

The idea that instructions and data of many types can be stored in memory as numbers and thus be easy to change, leading to the stored-program computer.

Components of Instruction



Opcode
Opcode (Category of Operation) Ex: Memory, ALU, Control
Function (Actual Operation) Ex: ADD, MUL, SUB
Operand
Implicit Operand
Accumulator (Input/Output)
Stack (Input/Output)
Immediate Data (Only Input)
Explicit Operand
Input/Output General Purpose Registers (GPR)
Input/Output Memory Location

Main Memory

- Storage is external to the processor
- CPU performs read and write operations on the main memory
- Who allocates data in main memory?OS
- How many of the operands are memory addresses in an instruction?

General Purpose Registers

Storage is internal to the processor and is faster than memory

Registers are more efficiently allocated by compilers

How many registers are sufficient? Depends on the smartness of the compiler.

How many of the operands are memory addresses in an instruction?

RISC-V General Purpose Register (GPR)

32 GPRs numbered X0 to X31

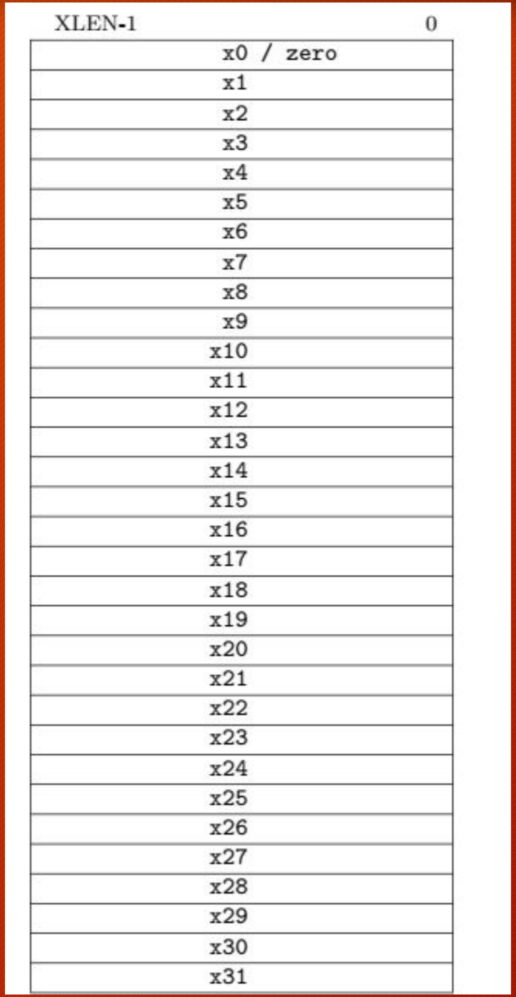
X0 is always set to 0

RV32 ISA has 32 bit registers

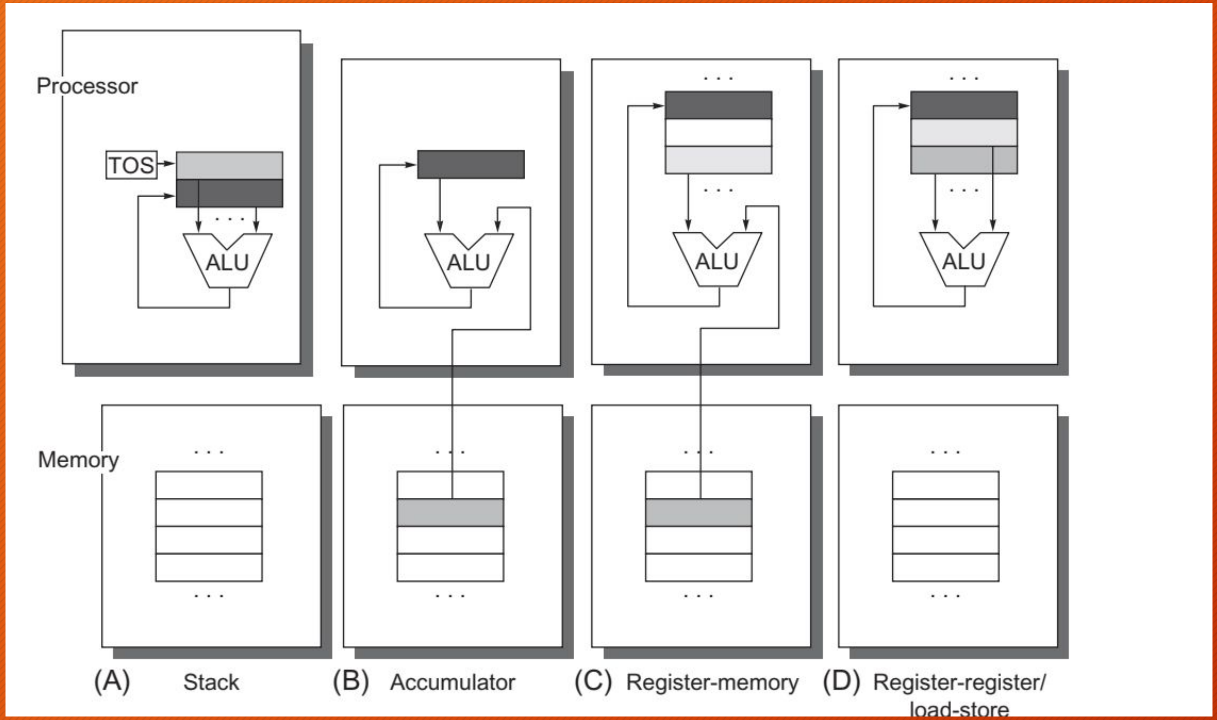
RV64 ISA has 64 bit registers

What is PC? Same size as GPRs

What is Instruction Register?



Classification of Instruction Set



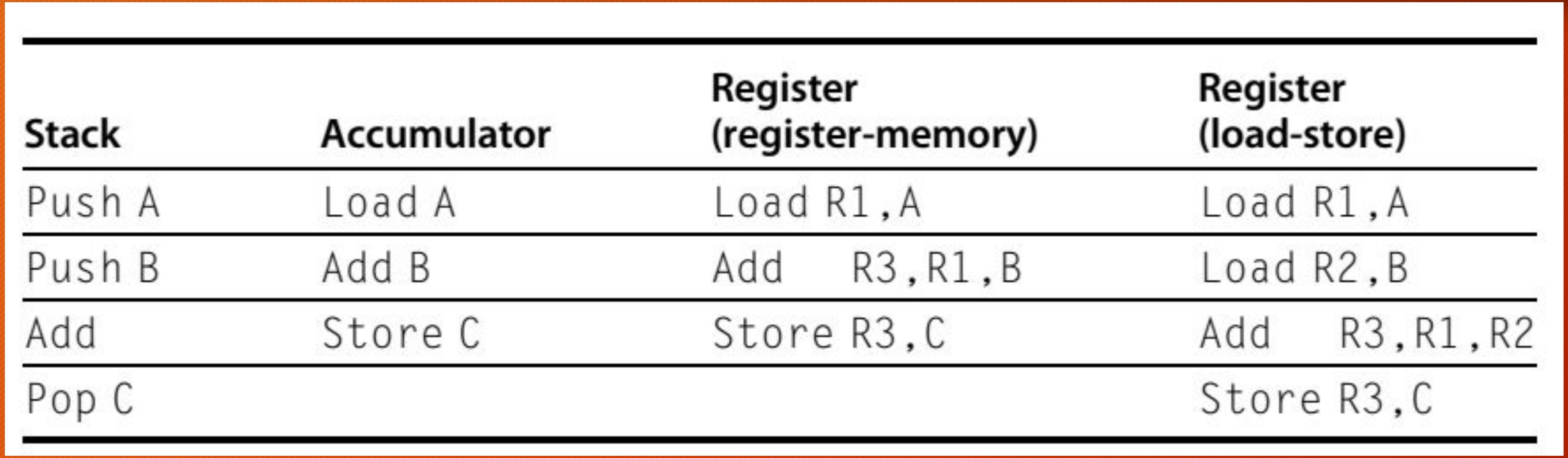
Stack: The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result.

Accumulator: The accumulator is open of the operand and the result.

Register-Memory: One input operand is a register, one is in memory, and the result goes to a register.

Register-Register: All operands are registers.

Classification of Instruction Set (Cont.). How is C = A + B Executed?



Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Stack: A data structure for spilling registers organized as a last-in-first-out queue.

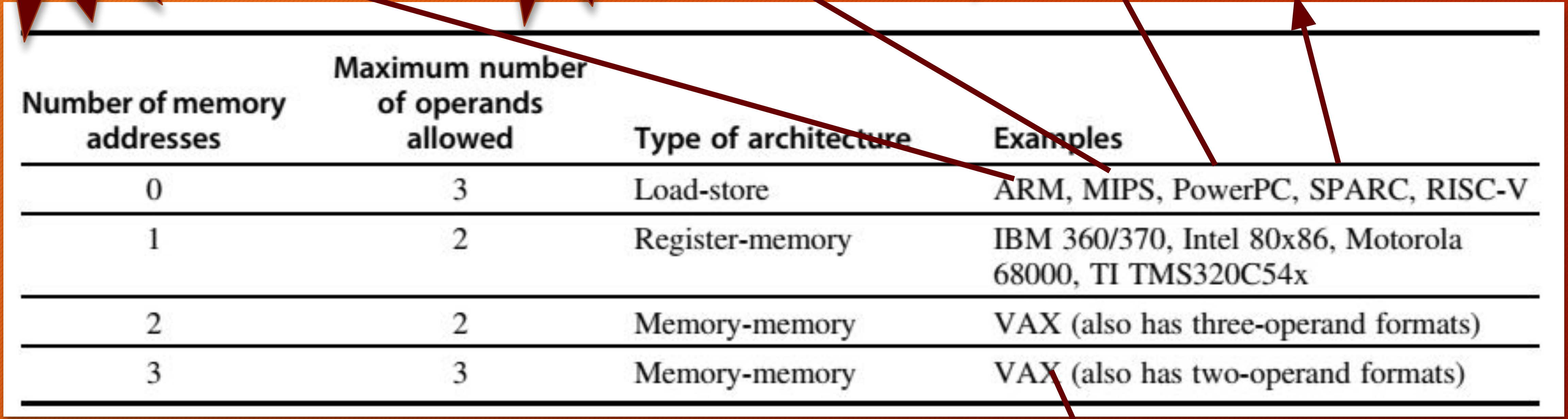
Stack Pointer: A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found.

Advanced RISC Machines

Million Instructions Per Second

Apple-IBM-Motorola (NXP)

Sun Microsystems (Oracle)



Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Combination for memory operands

Digital Equipment Corporations

Register-Register Operands

Advantages

- Simple, fixed-length instruction encoding
- Simple code generation model.
- Instructions take similar numbers of clocks to execute

Disadvantages

- Higher instruction count than architectures with memory references in instructions.
- More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects

What is Instruction Encoding?

$\text{ADD R1, R2, R3} \longrightarrow \text{R1 = R2 + R3}$

What is Instruction Density?

Register-Memory Operands

Advantages

- Data can be accessed without a separate load instruction first.
- Lower Instruction count
- Higher Instruction Density

Disadvantages

- Clocks per instruction vary by operand location
- Complex Code Generation Process
- Variable Length Encoding

$$\begin{tikzcd}\text{ADD R1, R2, 10[R3]} \arrow[d] \\ \text{R1 = R2 + M[10+R3]}\end{tikzcd}$$

TYPES AND SIZE OF OPERANDS

Word: Integer, Float (4)

Double Word: Long Int, Double (8)

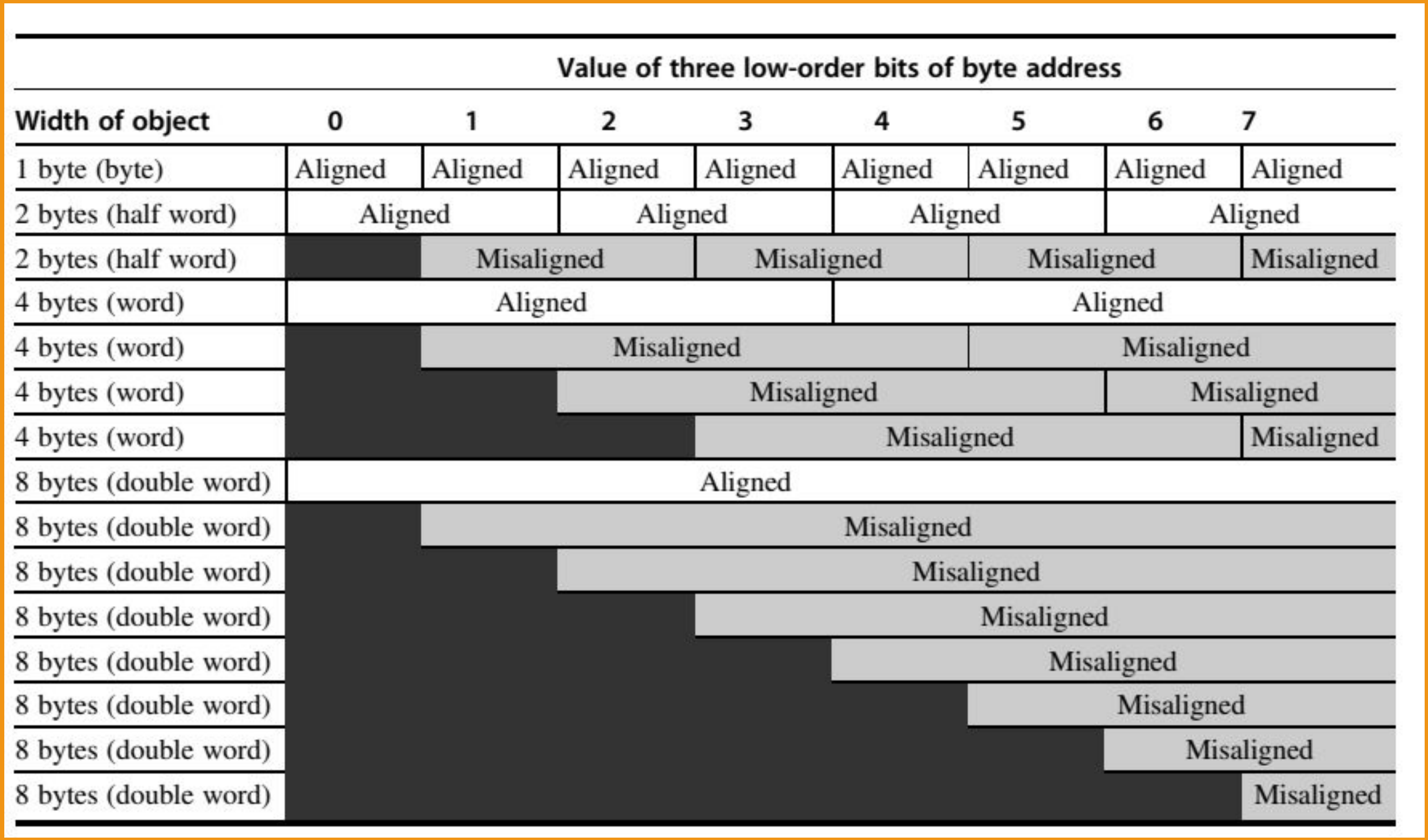
Half Word: Short Integer (2)

Quarter Word: Character (1)

RISC does not support misaligned addresses

Aligned and Misaligned addresses

X86 supports misaligned addresses



	Value of three low-order bits of byte address							
Width of object	0	1	2	3	4	5	6	7
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

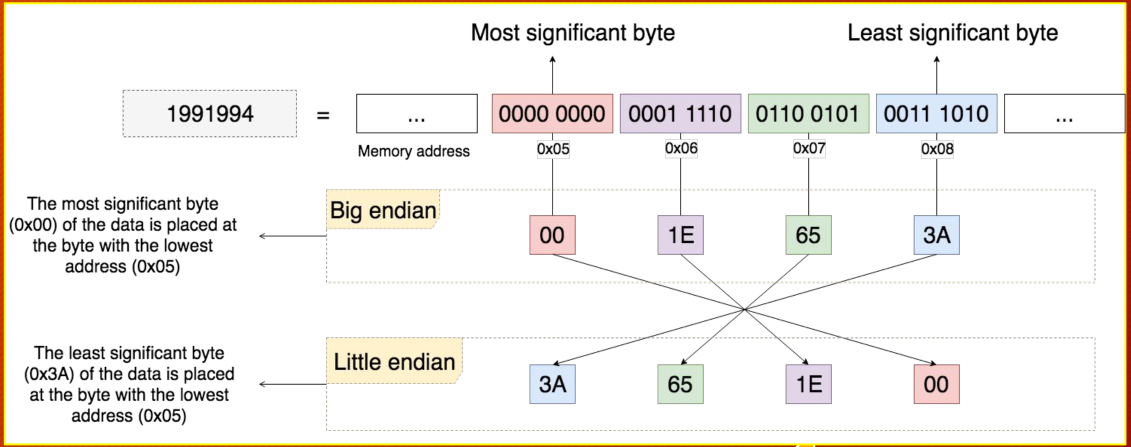
Interpreting Memory Address

Little Endian

- Last byte of binary representation of multi-byte data-type is stored first
- 0x01234567 is stored in the memory as 67 45 23 01
- It's easy to cast the value to a smaller type like from int16_t to int8_t since int8_t is the byte at the beginning of int16_t.

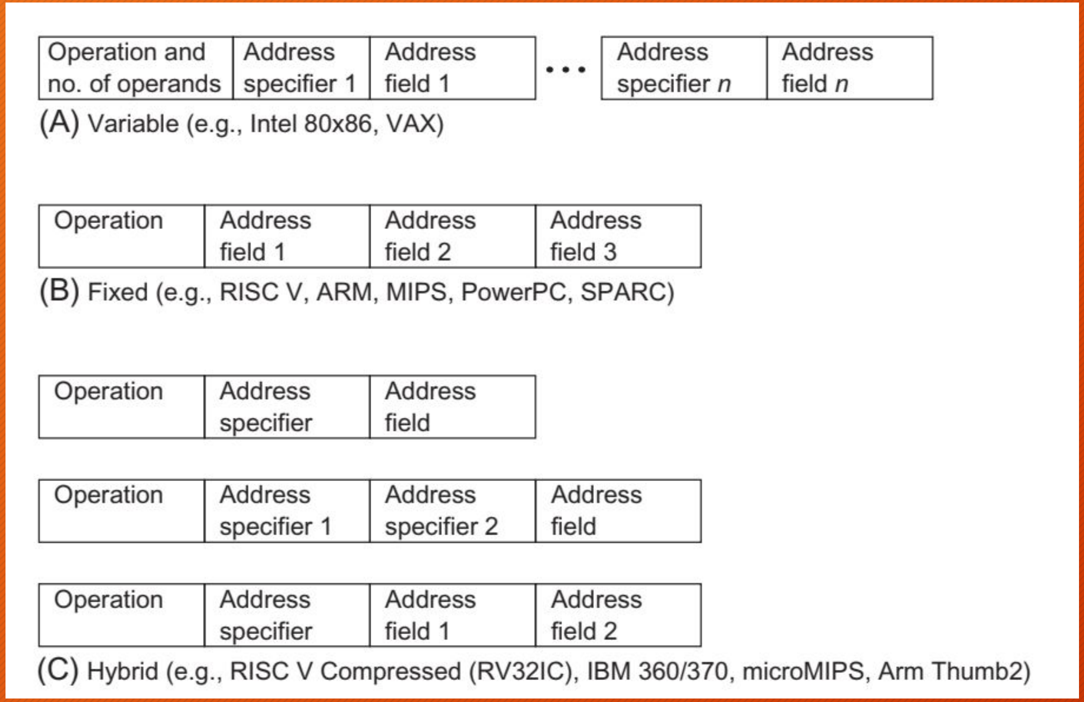
Big Endian

- First byte of binary representation of multi-byte data-type is stored first
- 0x01234567 is stored in the memory as 01 23 45 57
- More human understandable
- Effective for signed numbers because 1st bit is signed bit.


Most significant byte
Least significant byte
1991994
=
...
0000 0000
0001 1110
0110 0101
0011 1010
...
Memory address
0x05
0x06
0x07
0x08
The most significant byte (0x00) of the data is placed at the byte with the lowest address (0x05)
Big endian
00
1E
65
3A
The least significant byte (0x3A) of the data is placed at the byte with the lowest address (0x05)
Little endian
3A
65
1E
00

How to find if a system is little-endian or big-endian?

Encoding Instruction Set



(A) Variable (e.g., Intel 80x86, VAX)

(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)

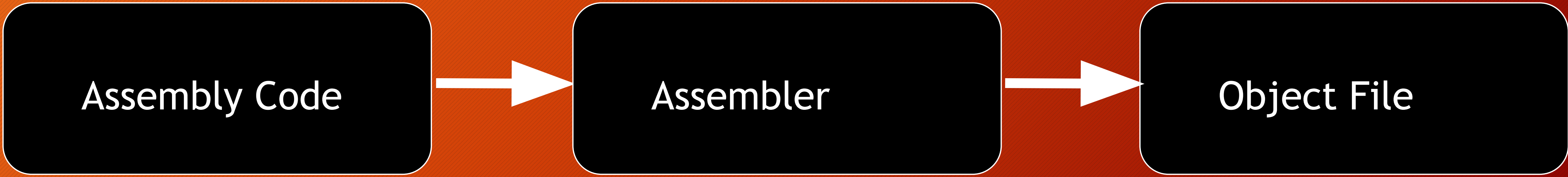
(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

Factors influencing encoding instruction set

- Desire to have as many instruction set or addressing modes as possible
- Impact of size of registers and addressing mode fields on average instruction size and hence average program size
- A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation.

Assemblers

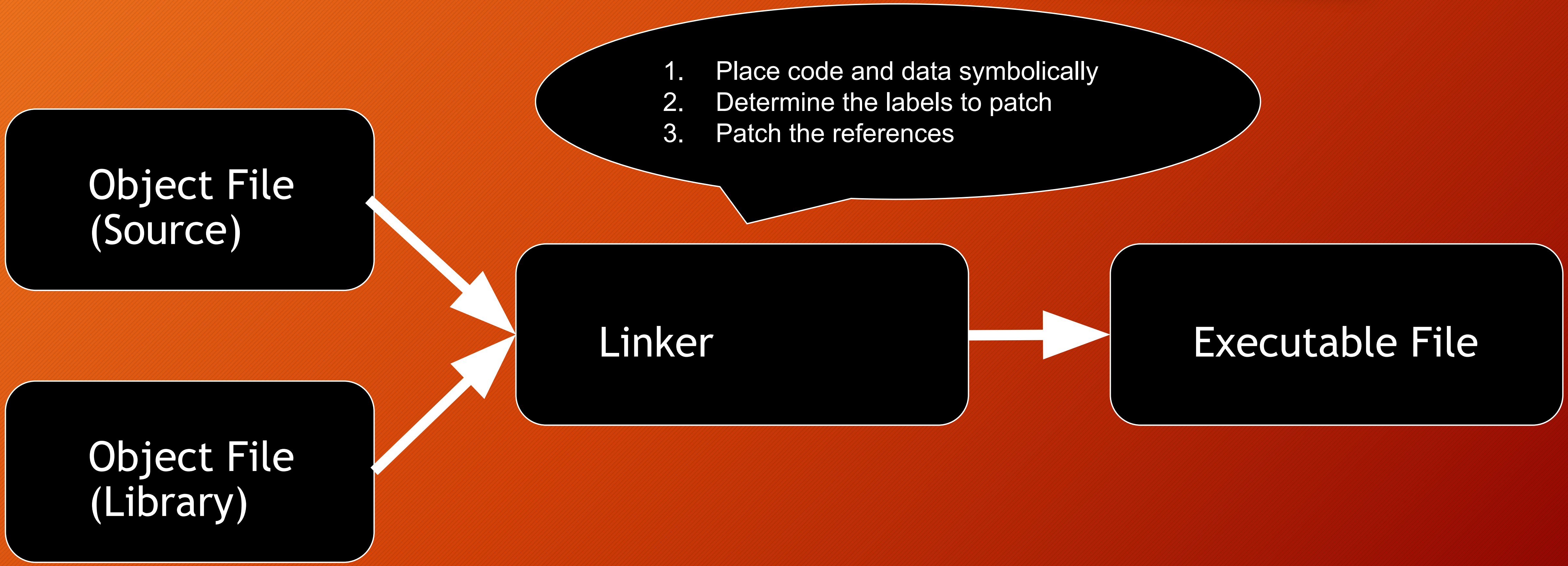
- The assembler turns the assembly language program into an object file, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.
- Assemblers keep track of labels used in branches and data transfer instructions in a symbol table.
- Symbol table matches names of labels to the addresses of the memory words that instructions occupy



Object File

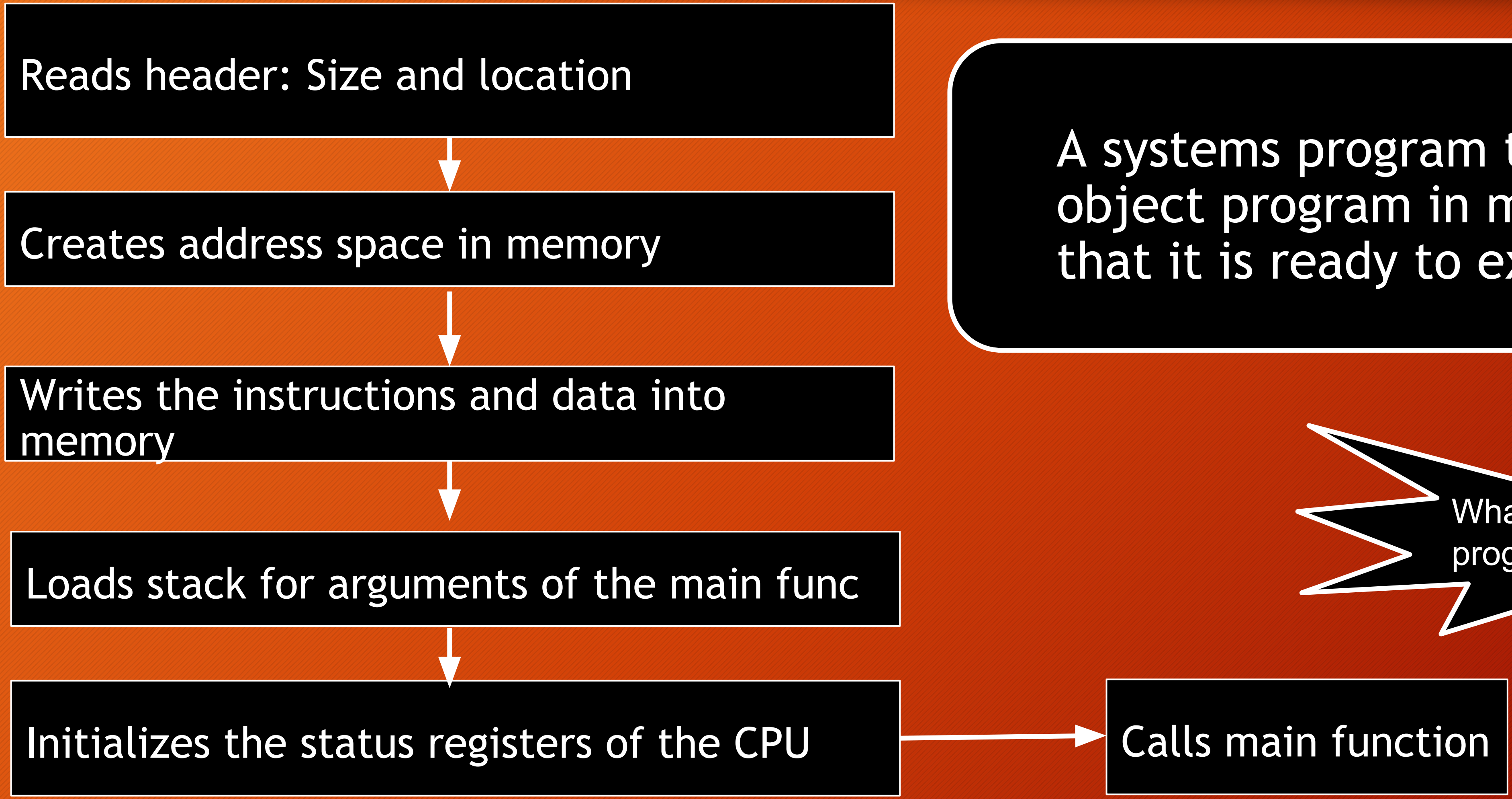
- Header: Specifies size and position
- Code in form of machine language
- Static and Extern Data
- Relocation information: Move code anywhere
- Symbol Table
- Debugging Information: Did you use gdb?

Linker



1. Place code and data symbolically
2. Determine the labels to patch
3. Patch the references

Loader



Reads header: Size and location

Creates address space in memory

Writes the instructions and data into memory

Loads stack for arguments of the main func

Initializes the status registers of the CPU

Calls main function

A systems program that places an object program in main memory so that it is ready to execute.

What happens on program termination?

RISC-V Instruction Set

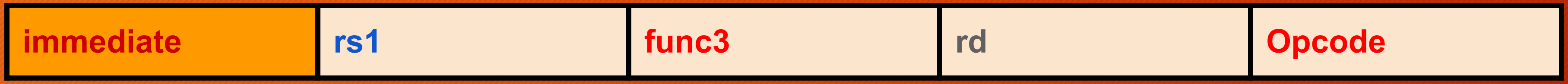
- Immediate Type (I-Type)
- Upper Immediate (U-Type)
- Register Type (R-Type)
- Branch Type (B-Type)
- Store Type (S-Type)
- Jump Type (J-Type)

Addressing Mode

Definition: One of several addressing regimes delimited by their varied use of operands and/or addresses.

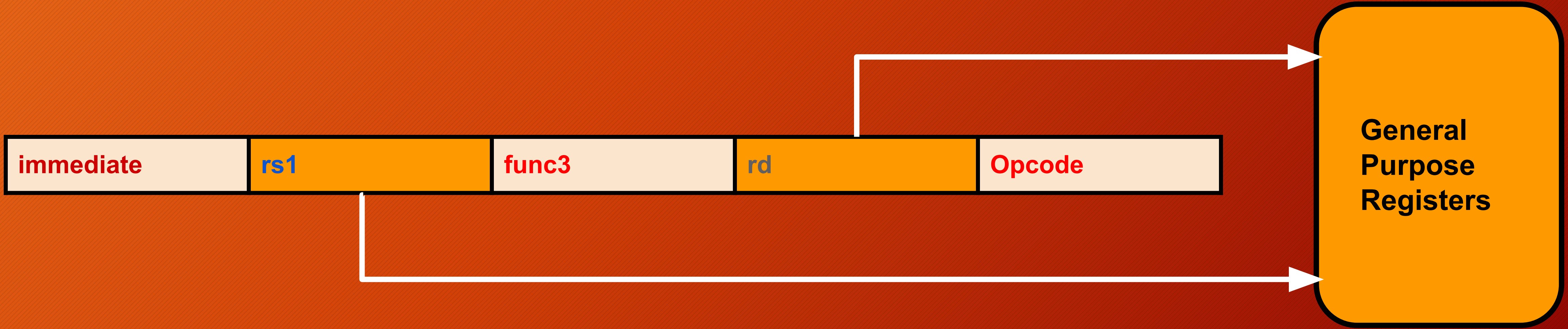
Meaning: The way in which the operand of an instruction is specified

Immediate and Register Addressing Mode



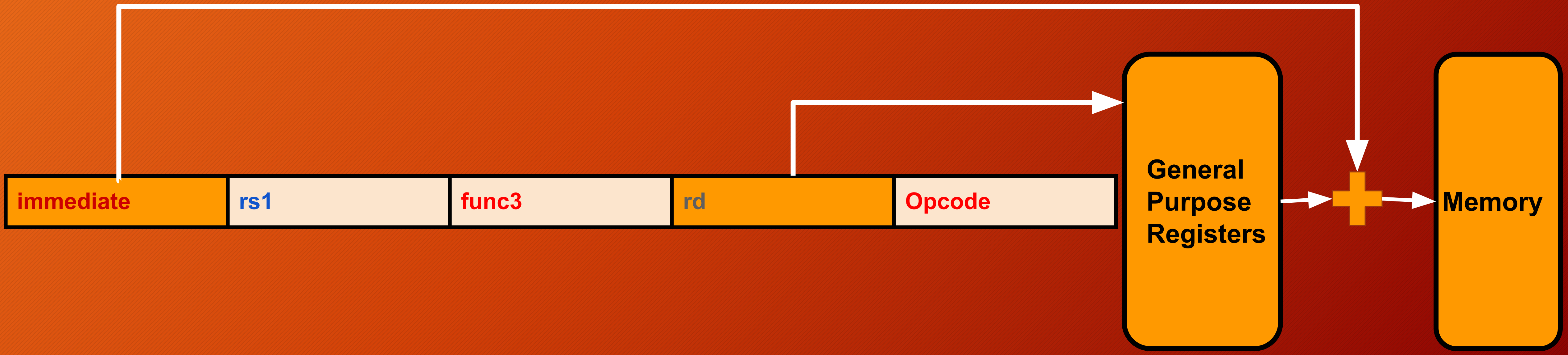
Immediate: Operand is a constant within the instruction itself

Register: Operand is a register



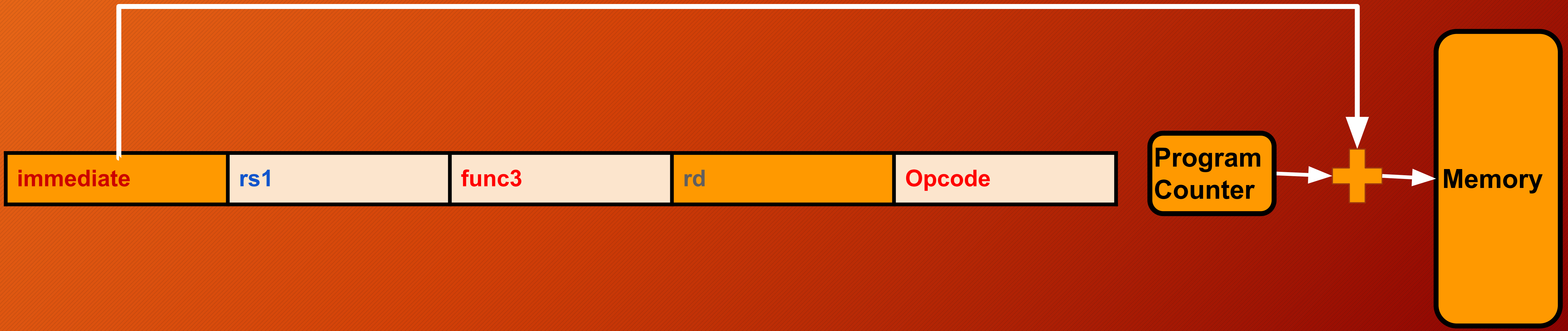
Displaced Addressing Mode

The operand is at the memory location whose address is the sum of a register and a constant in the instruction.

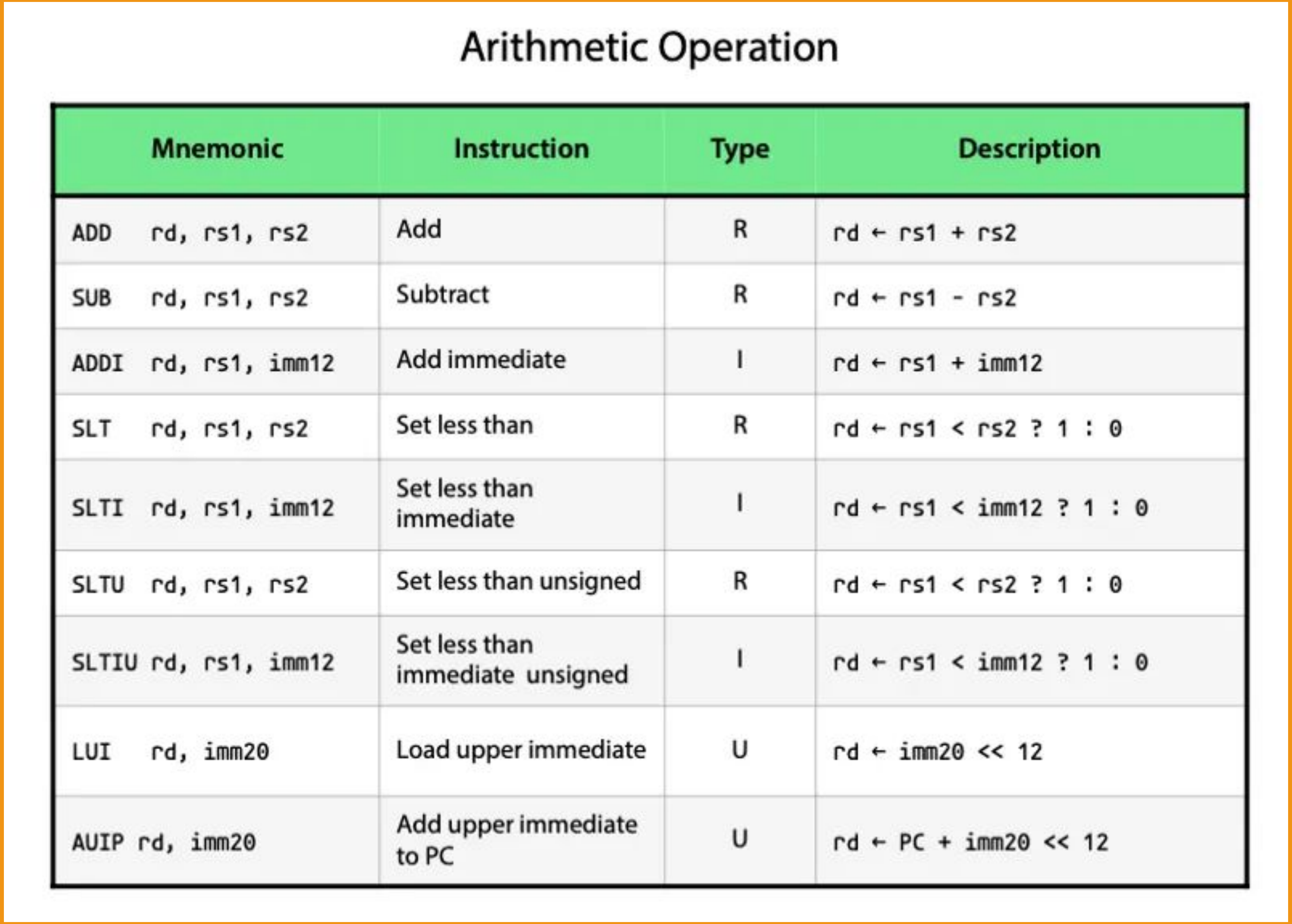


PC-relative Addressing Mode

The branch address is the sum of the PC and a constant in the instruction



RV32I Instruction Set: Arithmetic

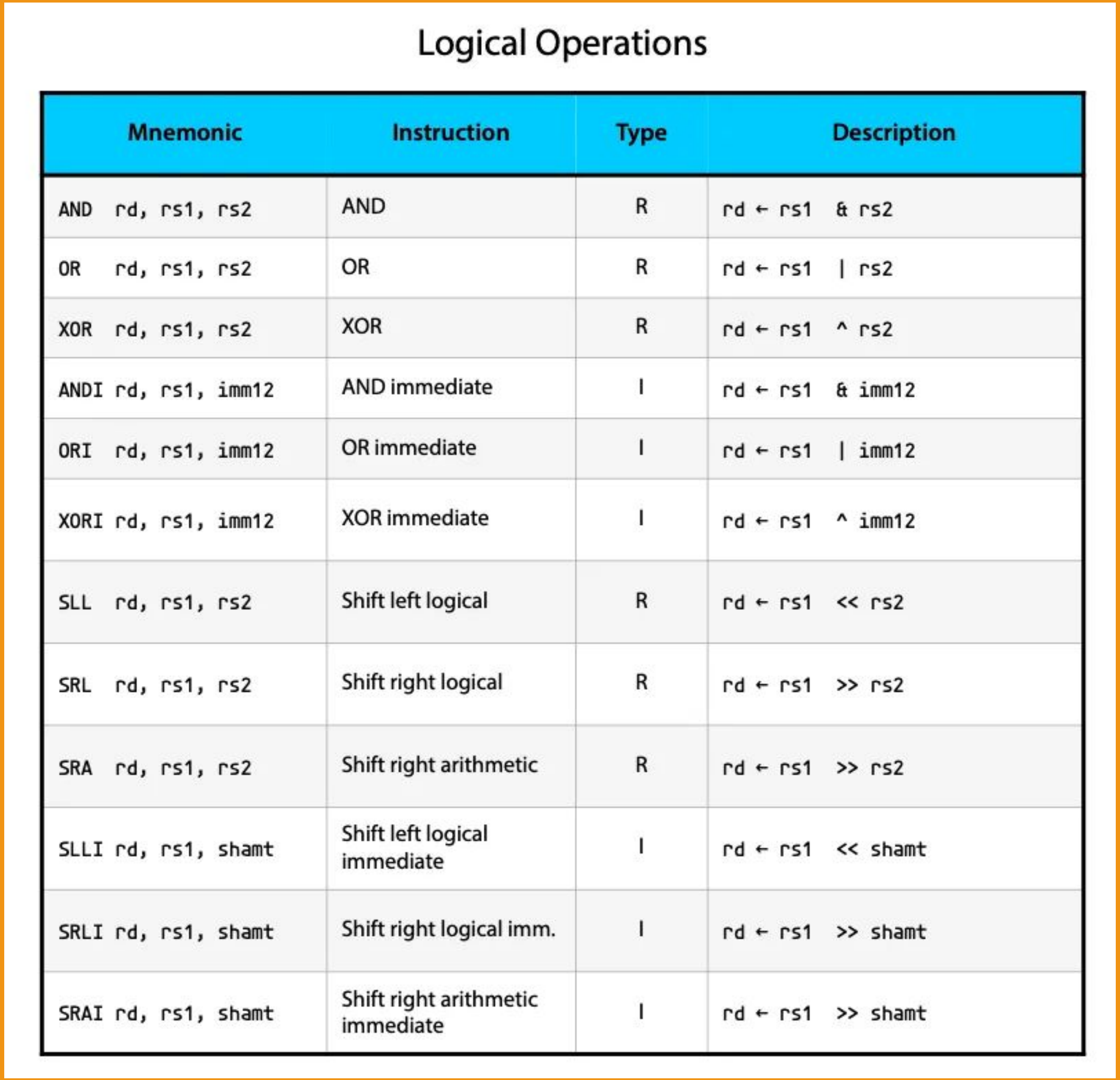


Arithmetic Operation

Mnemonic	Instruction	Type	Description
`ADD rd, rs1, rs2`	Add	R	`rd ← rs1 + rs2`
`SUB rd, rs1, rs2`	Subtract	R	`rd ← rs1 - rs2`
`ADDI rd, rs1, imm12`	Add immediate	I	`rd ← rs1 + imm12`
`SLT rd, rs1, rs2`	Set less than	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTI rd, rs1, imm12`	Set less than immediate	I	`rd ← rs1 < imm12 ? 1 : 0`
`SLTU rd, rs1, rs2`	Set less than unsigned	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTIU rd, rs1, imm12`	Set less than immediate unsigned	I	`rd ← rs1 < imm12 ? 1 : 0`
`LUI rd, imm20`	Load upper immediate	U	`rd ← imm20 << 12`
`AUIP rd, imm20`	Add upper immediate to PC	U	`rd ← PC + imm20 << 12`

https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

RV32I Instruction Set: Arithmetic

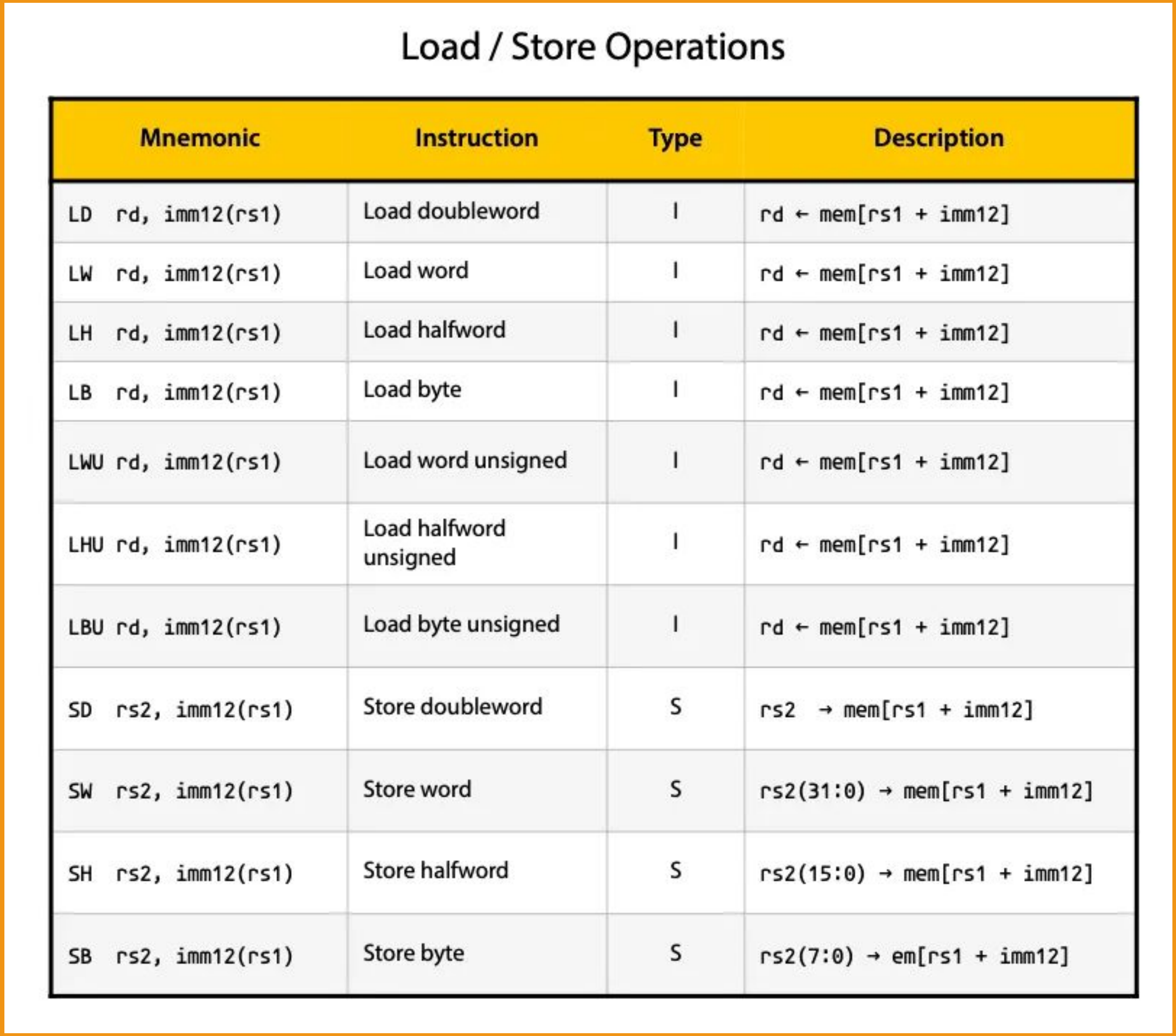


Logical Operations

Mnemonic	Instruction	Type	Description
`AND rd, rs1, rs2`	AND	R	`rd ← rs1 & rs2`
`OR rd, rs1, rs2`	OR	R	`rd ← rs1 | rs2`
`XOR rd, rs1, rs2`	XOR	R	`rd ← rs1 ^ rs2`
`ANDI rd, rs1, imm12`	AND immediate	I	`rd ← rs1 & imm12`
`ORI rd, rs1, imm12`	OR immediate	I	`rd ← rs1 | imm12`
`XORI rd, rs1, imm12`	XOR immediate	I	`rd ← rs1 ^ imm12`
`SLL rd, rs1, rs2`	Shift left logical	R	`rd ← rs1 << rs2`
`SRL rd, rs1, rs2`	Shift right logical	R	`rd ← rs1 >> rs2`
`SRA rd, rs1, rs2`	Shift right arithmetic	R	`rd ← rs1 >> rs2`
`SLLI rd, rs1, shamt`	Shift left logical immediate	I	`rd ← rs1 << shamt`
`SRLI rd, rs1, shamt`	Shift right logical imm.	I	`rd ← rs1 >> shamt`
`SRAI rd, rs1, shamt`	Shift right arithmetic immediate	I	`rd ← rs1 >> shamt`

https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

RV32I Instruction Set: Load Store

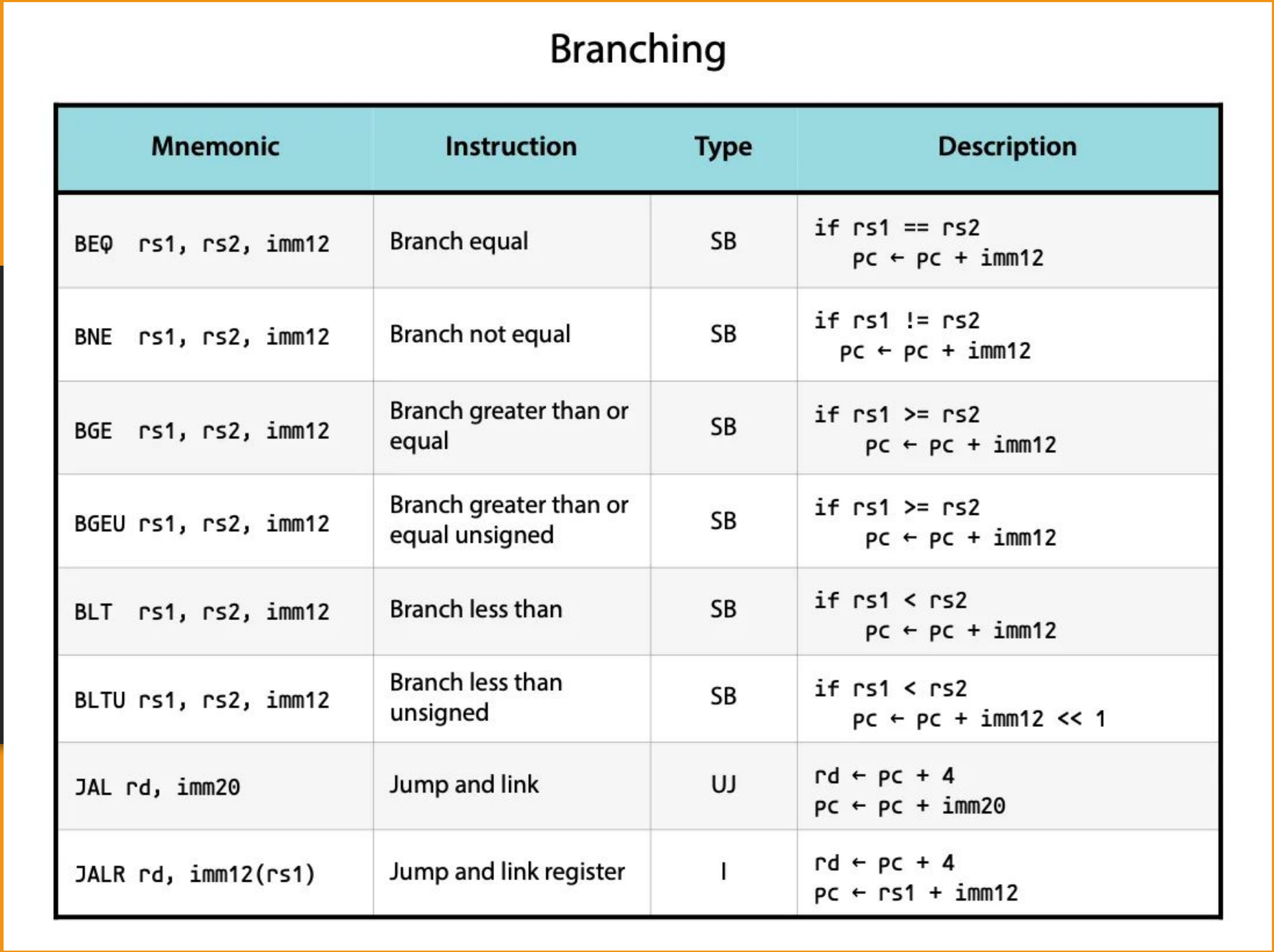


Load / Store Operations

Mnemonic	Instruction	Type	Description
`LD rd, imm12(rs1)`	Load doubleword	I	`rd ← mem[rs1 + imm12]`
`LW rd, imm12(rs1)`	Load word	I	`rd ← mem[rs1 + imm12]`
`LH rd, imm12(rs1)`	Load halfword	I	`rd ← mem[rs1 + imm12]`
`LB rd, imm12(rs1)`	Load byte	I	`rd ← mem[rs1 + imm12]`
`LWU rd, imm12(rs1)`	Load word unsigned	I	`rd ← mem[rs1 + imm12]`
`LHU rd, imm12(rs1)`	Load halfword unsigned	I	`rd ← mem[rs1 + imm12]`
`LBU rd, imm12(rs1)`	Load byte unsigned	I	`rd ← mem[rs1 + imm12]`
`SD rs2, imm12(rs1)`	Store doubleword	S	`rs2 → mem[rs1 + imm12]`
`SW rs2, imm12(rs1)`	Store word	S	`rs2(31:0) → mem[rs1 + imm12]`
`SH rs2, imm12(rs1)`	Store halfword	S	`rs2(15:0) → mem[rs1 + imm12]`
`SB rs2, imm12(rs1)`	Store byte	S	`rs2(7:0) → em[rs1 + imm12]`

https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

RV32I Instruction Set: Branching

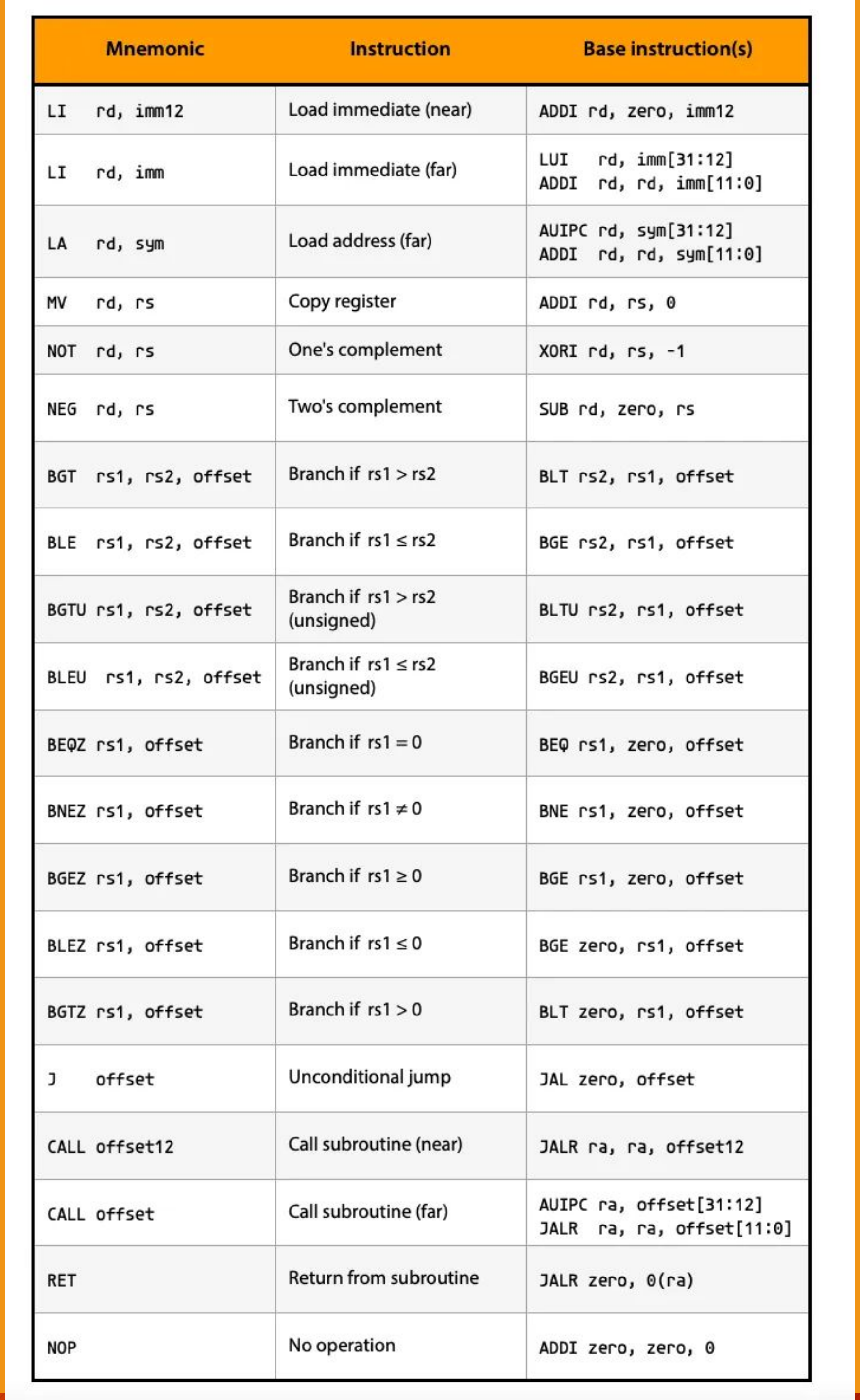


Branching

Mnemonic	Instruction	Type	Description
`BEQ  rs1, rs2, imm12`	Branch equal	SB	`if rs1 == rs2` `pc ← pc + imm12`
`BNE  rs1, rs2, imm12`	Branch not equal	SB	`if rs1 != rs2` `pc ← pc + imm12`
`BGE  rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 >= rs2` `pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2` `pc ← pc + imm12`
`BLT  rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2` `pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2` `pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4` `pc ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4` `pc ← rs1 + imm12`

https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

RV32I Instruction Set: Pseudo-Instruction



Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`

`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]`
		`ADDI rd, rd, imm[11:0]`

`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]`
		`ADDI rd, rd, sym[11:0]`

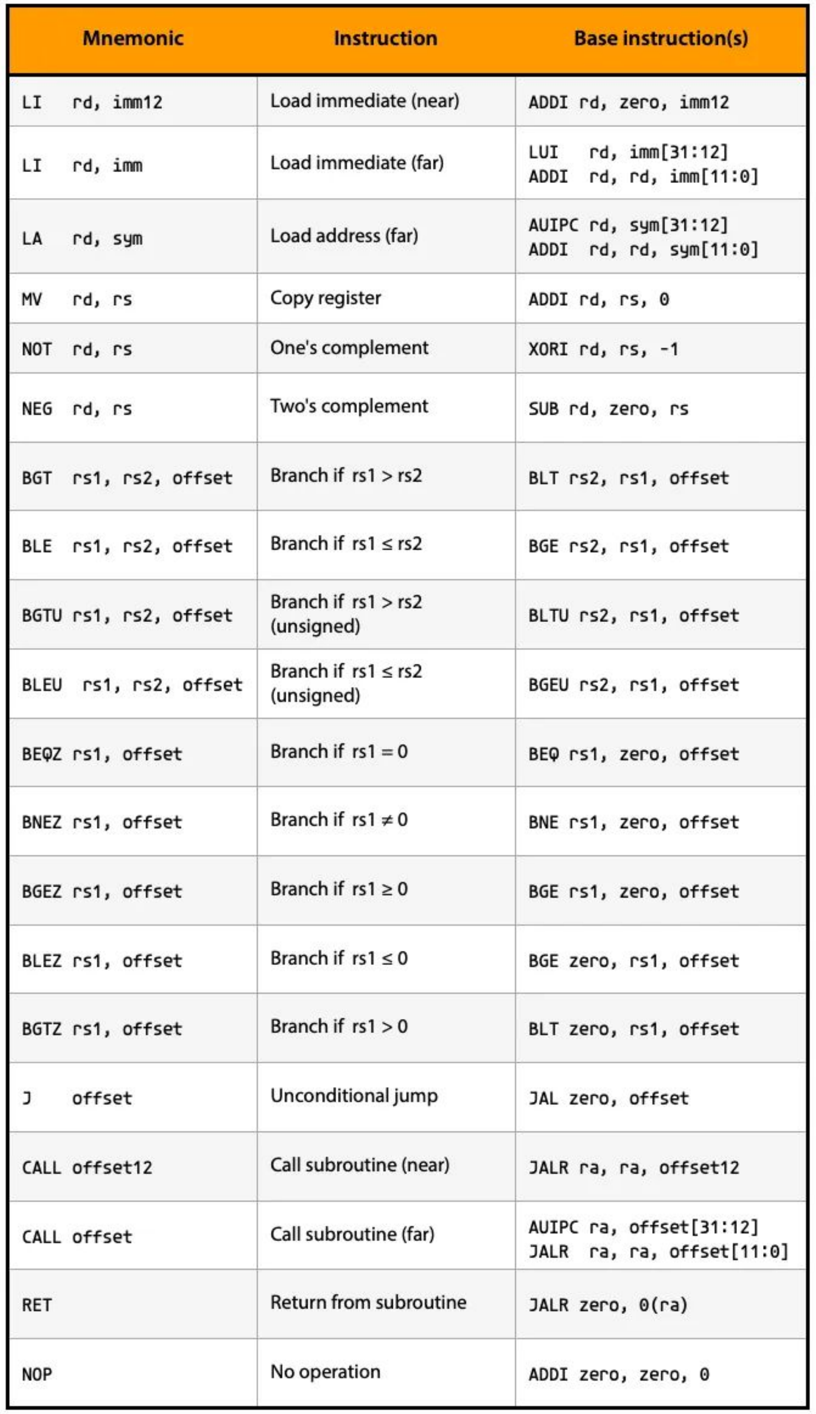
`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`

`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]`
		`JALR ra, ra, offset[11:0]`

`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

RV32I Instruction Set: Pseudo-Instruction



Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`

`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]`
		`ADDI rd, rd, imm[11:0]`

`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]`
		`ADDI rd, rd, sym[11:0]`

`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`

`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]`
		`JALR ra, ra, offset[11:0]`

`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4b

Immediate Type (I-Type)

immediate[31-20]	rs1 [19-15]	func3 [14-12]	rd [11-7]	Opcode [6-0]
immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

func3 bits selects between the 6 immediate ALU operations: Add, And, Or, Xor, set less than immediate, set less than immediate unsigned

c = a + 10 $\longrightarrow$ ADDI dest, src, imm $\longrightarrow$ rd = rs + imm12

Immediate Type (I-Type)

immediate[31-25]	immediate[24-20]	rs1 [19-15]	func3 [14-12]	rd [11-7]	Opcode [6-0]
0000000	shamt[4:0]	src	SLLI/SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

func3 bits selects rest 2 ALU operations: Shift Left, Shift Right

All func3 bits are exhausted, so bit 30 will be set to have extensions of I-Type instructions

a = b << 10; $\longrightarrow$ SLLI dest, src, imm $\longrightarrow$ rd = rs << shamt5

Immediate Type (I-Type): Load Instructions

immediate[31-20]	rs1 [19-15]	func3 [14-12]	rd[11-7]	Opcode [6-0]
offset[11-0]	src	width(LW/LD/LH/LB/LWU/LHU/LBU)	dest	LOAD

func3 bits selects between 7 Load operations: Load Word, Load Double Word, Load Half Word, Load Byte, Load Word Unsigned, Load Half Word Unsigned, Load Byte Unsigned

c = a[i] + b[i]; $\longrightarrow$ LW rd, imm12[rs] $\longrightarrow$ rd = Mem[rs + imm12]

Register Type (R-Type)

func7[31-25]	rs2 [24-20]	rs1 [19-15]	func3 [14-12]	rd [11-7]	Opcode [6-0]
0000000	**src2**	**src1**	**ADD/SLT/SLTU/AND /OR/XOR/SLL/SRL**	**dest**	**OP**
0100000	**src2**	**src1**	**SUB/SRA**	**dest**	**OP**

- Operands are attempted to be placed in same location for easy decoding
- funct3 determines the actual operation to be executed
- funct7 are the extension bits for existing and new extensions
- 30th bit is used to distinguish SUB and Arithmetic Right Shift from other ALU operations

$c = a + b \longrightarrow$ ADD rd, rs1, rs2 $\longrightarrow$ rd = rs1 + rs2

Store Type (S-Type)

immediate[31-20]	rs2[24-20]	rs1 [19-15]	func3 [14-12]	imm[11-7]	Store[6-0]
Offset[11:5]	src	base	width (SD/SW/ SHW/SB)	Offset[4:0]	STORE

func3 bits selects between 3 Store Operations: Store Double, Store Word, Store Half Word, Store Byte

arr[i] = X; $\longrightarrow$ SW rd, imm12[rs] $\longrightarrow$ Mem[rs + imm12] = rd

Conditional Branch Type (B-Type)

immediate[31-25]	rs2[24-20]	rs1 [19-15]	func3 [14-12]	immediate[11-7]	Opcode [6-0]
Offset[12,10:5]	src2	src1	BEQ/BNE/BLT[U] /BGE[U]	Offset[11,4:1]	BRANCH

func3 bits selects between 6 branch operations: branch equal, branch not equal, branch less than, branch less than unsigned, branch greater equal, branch greater equal unsigned

BEQ src1, src2, imm12 $\longrightarrow$ if src1 == src2 then pc = pc + imm12

Unconditional Branch: Jump Type (J-Type)

Immediate[31:12]	rd [11-7]	Opcode [6-0]
Offset[20,10:1,11,19:12]	**dest**	**JAL**

- The offset/imm is sign-extended and added to the pc to form the jump target address.
- JAL stores the address of the next instruction into register rd, i.e., rd = pc + 4
- Used in switch case and procedure call or return
- With JAL, pc = pc + imm20
- With JALR, pc = rs + imm12

immediate[31-20]	rs1 [19-15]	func3 [14-12]	rd[11-7]	Opcode [6-0]
offset[11-0]	**base**	**0**	**dest**	**JALR**

Upper Immediate Type (U-Type)

Immediate[31:12]	rd [11-7]	Opcode [6-0]
U-immediate[31-12]	**dest**	**LUI/AUIPC**

- LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.
- With LUI, rd = imm << 20
- AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd
- With AUIPC, rd = pc + (imm << 20)

NOP Instruction

immediate[31-20]	rs1 [19-15]	func3 [14-12]	rd[11-7]	Opcode [6-0]
0	0	ADDI	0	OP-IMM

- The NOP instruction does not change any user-visible state, except for advancing the pc.
- NOP is encoded as ADDI x0, x0, 0.
- Remember! X0 is set to 0

Procedure Call and Return

Put parameters in a place where the procedure can access them

↓

Transfer control to the procedure

↓

Acquire the storage resources needed for the procedure

↓

Perform the desired task

↓

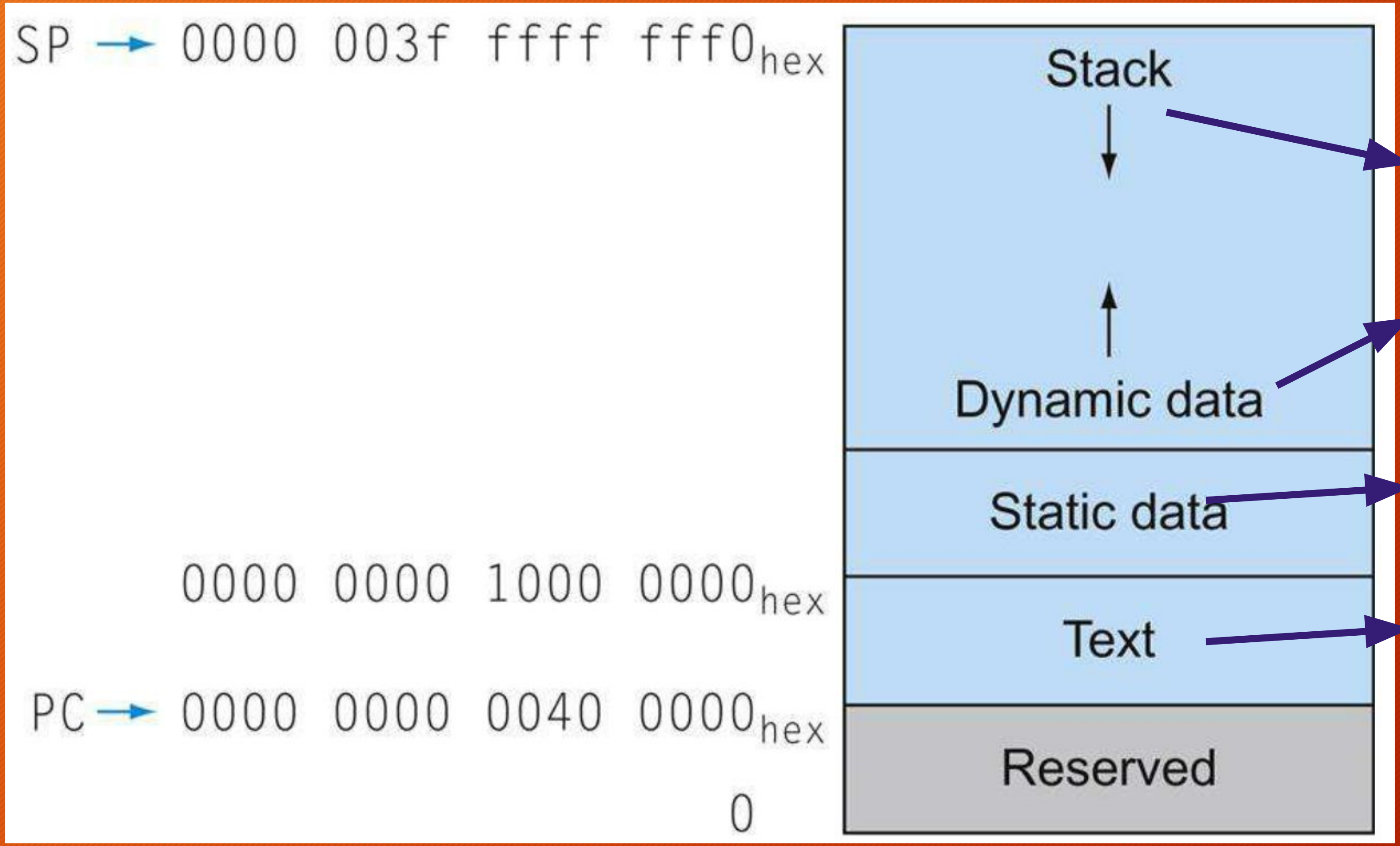
Put the result value in a place where the calling program can access it

→

Return control to the point of origin, since a procedure can be called from several points in a program

- X10-X17 are the parameter registers to pass value or reference or return values
- X1 is used for return address
- jal x1, ProcedureAddress => Used by caller
- jalr x0, 0(x1) => Used by callee
- What if there are more than 8 parameters to pass?
  - Use Stack
  - Stack Pointer (SP) is X2 register
  - Stack grows from higher address to lower address
- Assignment
  - What about nested procedure?
  - Which language supports nested procedure?

Memory Segments


SP → 0000 003f ffff fff0$_{hex}$
Stack
Dynamic data
Static data
0000 0000 1000 0000$_{hex}$
Text
PC → 0000 0000 0040 0000$_{hex}$
Reserved
0

Stack: Local Variable and Parameters
Stack Pointer (x2) points to the stack

Heap: Dynamic Memory Allocation

Data: Static and Extern Variables
Global Pointer (x3) points to the stack

Code

"Good Programmers understand what is the machine, they are Programming!"
"Good Architects understand for whom they are Building!"

Which one are you?