Tikrit University
Electrical Engineering Department

EE-317
Computer Engineering
2024-2025

Instructions: Procedure Calling

Jalal Nazar Abdulbaqi, Ph.D.

jalal.abdulbaqi@tu.edu.iq

Outline

- Procedure Calling
- Stack (Memory Layout)
- Character Data (String)
- Addressing

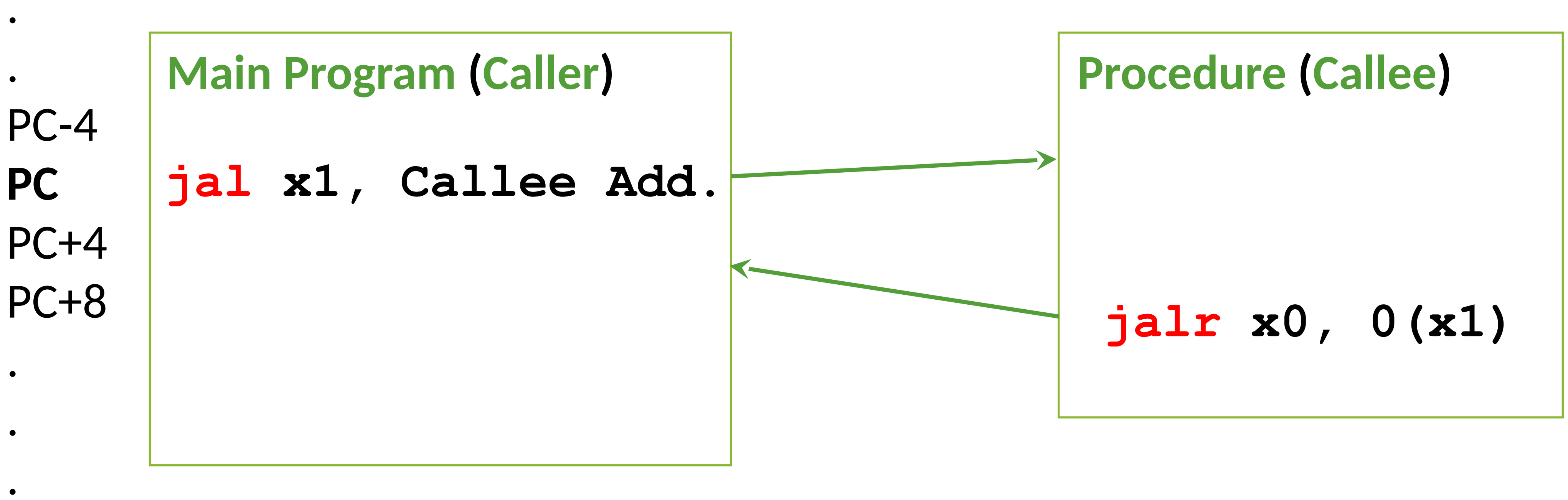
Procedure Call Instructions

- Procedure call: jump and link

  `jal x1, ProcedureLabel`
  - Address of following instruction put in `x1`
  - Jumps to target address
- Procedure return: jump and link register

  `jalr x0, 0(x1)`
  - Like `jal`, but jumps to `0` + address in `x1`
  - Use `x0` as `rd` (`x0` cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Procedure Calling



Address: x1

Data: x10-x17

x10, x11 – arguments/ return

x12 - x17 – arguments

Program Counter (PC)

Register hold the address current instruction being executed

Procedure Calling

Steps required

1. Place parameters in registers **`x10`** to **`x17`**
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call (address in **`x1`**)

Register Usage

- **`x5–x7, x28–x31`**: temporary registers
  - Not preserved by the callee

- **`x8–x9, x18–x27`**: saved registers
  - If used, the callee saves and restores them

Stack

- it is a last-in-fist-out (LIFO) queue saved in the memory.
- Stack Pointer (sp)

  register hold the most recent address in stack – **x2** in RISC-V
- Why we need **Stack** for **Procedure Calling**?
  - we must save the values of the registers that we will use in the procedure calculation.
  - in case we need more than 8 arguments.

Leaf Procedure Example - C code

```
int leaf_example (int g, int h, int i, int j){
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

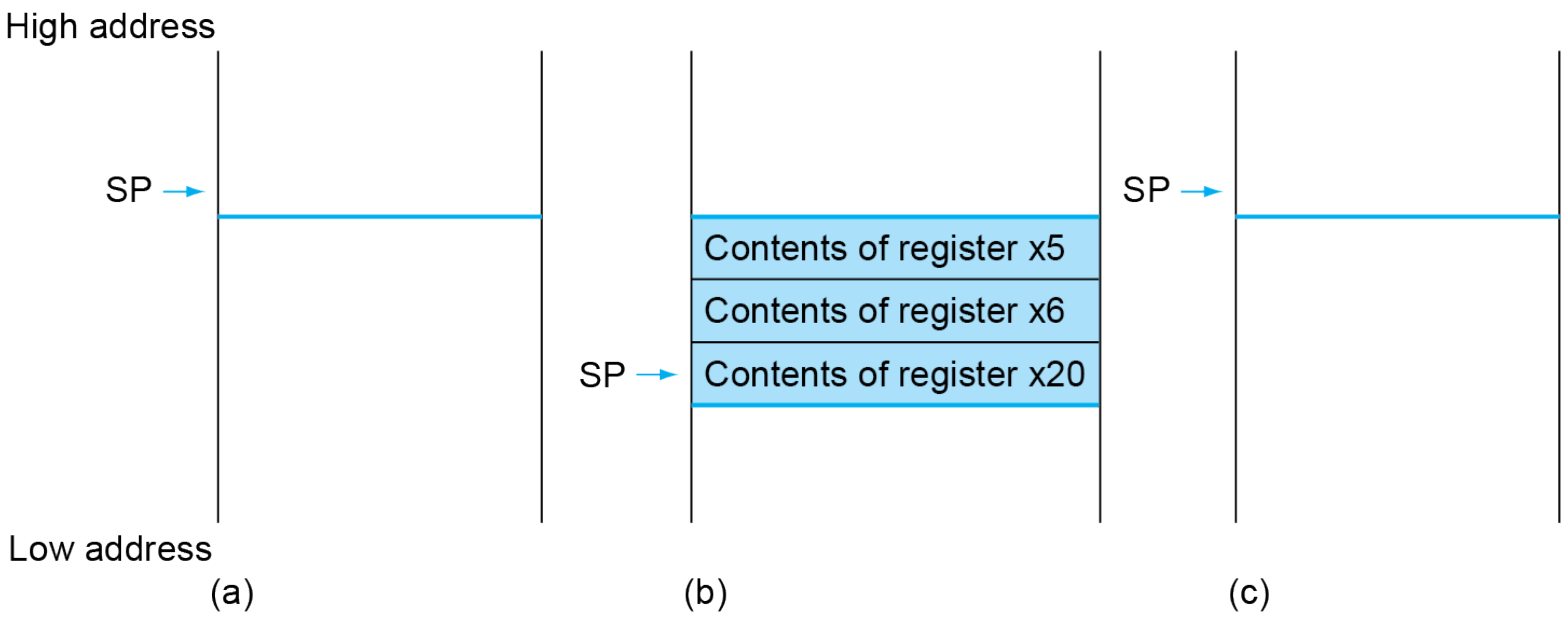
Arguments **`g, …, j`** in **`x10, …, x13`**

- **`f`** in **`x20`**
- temporaries **`x5, x6`**
- Need to save **`x5, x6, x20`** on stack

Leaf Procedure Example - RISC-V code

```
addi x2,x2,-12
sw   x5,8(x2)        # Save x5, x6, x20 on stack
sw   x6,4(x2)
sw   x20,0(x2)
add  x5,x10,x11      # x5 = g + h
add  x6,x12,x1       # x6 = i + j
sub  x20,x5,x6       # f = x5 – x6
addi x10,x20,0       # copy f to return register (x10)
lw   x20,0(x2)
lw   x6,4(x2)        # Restore x5, x6, x20 from stack
lw   x5,8(x2)
addi x2,x2,12
jalr x0,0(x1)        # Return to caller
```

Local Data on the Stack



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example - C code

```
int fact (int n)
{
  if (n < 1) return 1;
  else return (n * fact(n - 1));
}
```

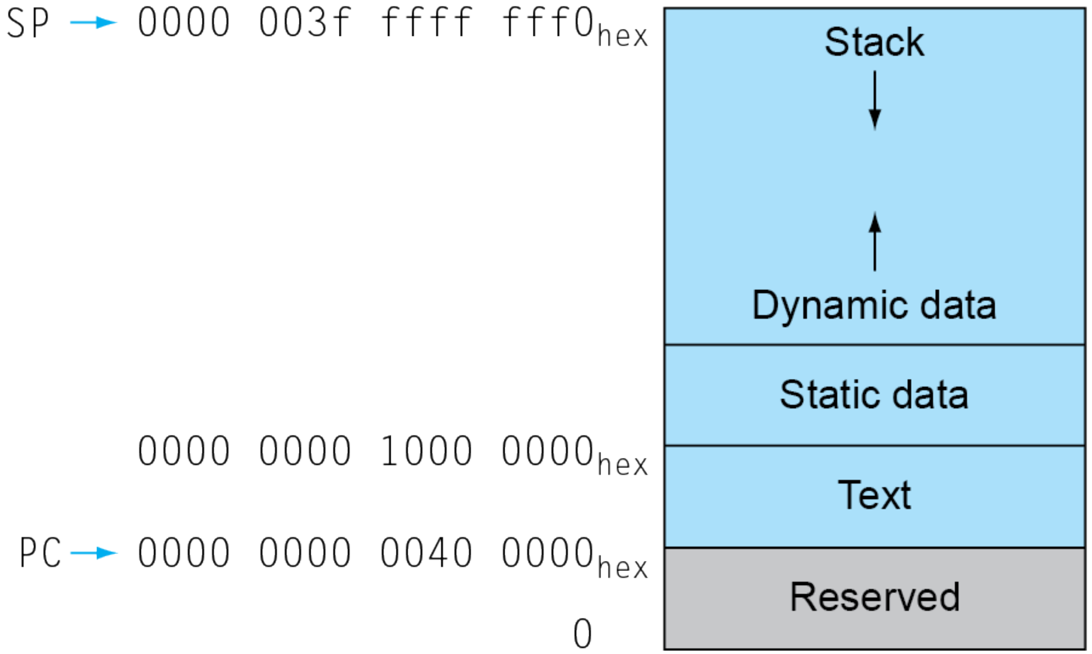
- Argument **`n`** in **`x10`**
- Result in **`x10`**

Non-Leaf Procedure Example - RISC-V code

```
Fact:   addi sp,sp,-8
        sw   x1,4(sp)       # Save return address and n on stack
        sw   x10,0(sp)
        addi x5,x10,-1      # x5 = n - 1
        bge  x5,x0,L1       # if n-1 >= 0, go to L1
        addi x10,x0,1       # Else, set return value to 1
        addi sp,sp,8        # Pop stack, don't bother restoring values
        jalr x0,0(x1)       # Return
L1:     addi x10,x10,-1     # n = n - 1
        jal  x1,fact        # call fact(n-1)
        addi x6,x10,0       # move result of fact(n - 1) to x6
        lw   x10,0(sp)      # Restore caller's n
        lw   x1,4(sp)       # Restore caller's return address
        addi sp,sp,8        # Pop stack
        mul  x10,x10,x6     # return n * fact(n-1)
        jalr x0,0(x1)       # return
```

Memory Layout

- **Text**: program code
- **Static data**: global variables
  - e.g., static variables in C, constant arrays and strings
  - **`x3`** (global pointer) initialized to address allowing ±offsets into this segment
- **Dynamic data**: heap
  - E.g., malloc in C, new in Java
- **Stack**: automatic storage



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

RISC-V Byte / Halfword / Word Operations

- **Load** byte/halfword/word: Sign extend to 32 bits in `rd`

```
lb rd, offset(rs1)
lh rd, offset(rs1)
lw rd, offset(rs1)
```

- **Load** byte/halfword/word unsigned: Zero extend to 32 bits in `rd`

```
lbu rd, offset(rs1)
lhu rd, offset(rs1)
lwu rd, offset(rs1)
```

- **Store** byte/halfword/word: Store rightmost 8/16/32 bits

```
sb rs2, offset(rs1)
sh rs2, offset(rs1)
sw rs2, offset(rs1)
```

String Copy Example - C code

- Null-terminated string

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i] = y[i]) != '\0')
    i += 1;
}
```

Base address of arrays `x` and `y` are found in `x10` and `x11`,

while `i` is in `x19`

String Copy Example - RISC-V code

```
strcpy:
        addi  sp,sp,-4       # adjust stack for 1 word
        sw    x19,0(sp)      # push x19
        add   x19,x0,x0      # i=0
L1:     add   x5,x19,x11     # x5 = addr of y[i]
        lbu   x6,0(x5)       # x6 = y[i]
        add   x7,x19,x10     # x7 = addr of x[i]
        sb    x6,0(x7)       # x[i] = y[i]
        beq   x6,x0,L2       # if y[i] == 0 then exit
        addi  x19,x19,1      # i = i + 1
        jal   x0,L1          # next iteration of loop
L2:     lw    x19,0(sp)      # restore saved x19
        addi  sp,sp,4        # pop 1 word from stack
        jalr  x0,0(x1)       # and return
```

32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient

- For the occasional 32-bit constant

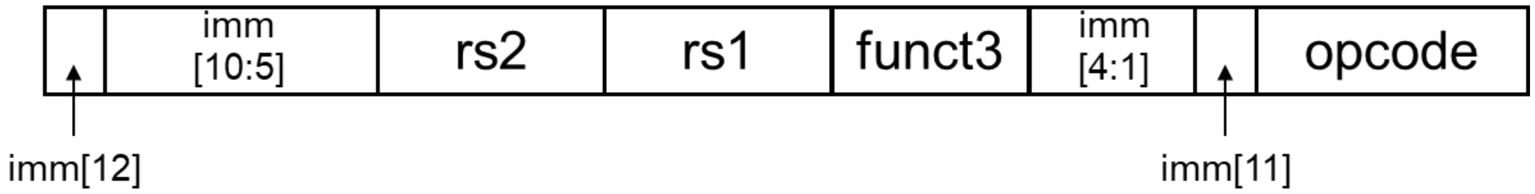
`lui rd, constant`

  - Copies 20-bit constant to bits [31:12] of **`rd`**
  - Clears bits [11:0] of **`rd`** to 0

bits	Data & Instructions	Hexa	Decimal
32	`00000000 00111101 00000101 00000000`	3D0500	3998976
	`lui x19, 0x3D0`		
20	`00000000 00111101 00000101 00000000`	3D0	976
	`addi x19, x19, 0x500`		
12	`00000000 00111101 00000101 00000000`	500	1280

Branch Addressing

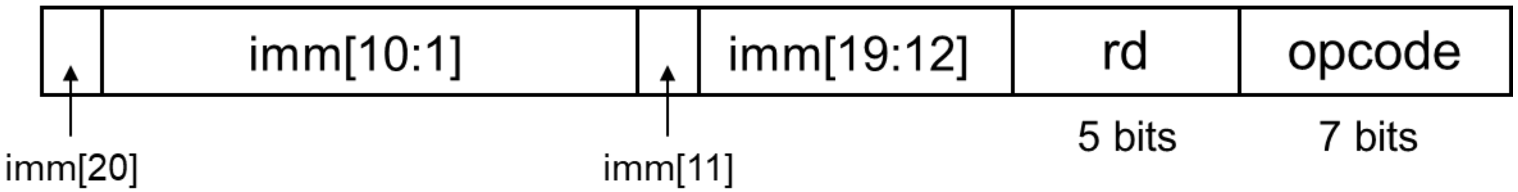
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:



- PC-relative addressing
  - Target address = PC + immediate × 2

Jump Addressing

- Jump and link (**`jal`**) target uses 20-bit immediate for larger range
- UJ format:



- For long jumps, e.g., to 32-bit absolute address
  - **`lui`**: load address[31:12] to temp register
  - **`jalr`**: add address[11:0] and jump to target

RISC-V Addressing Summary

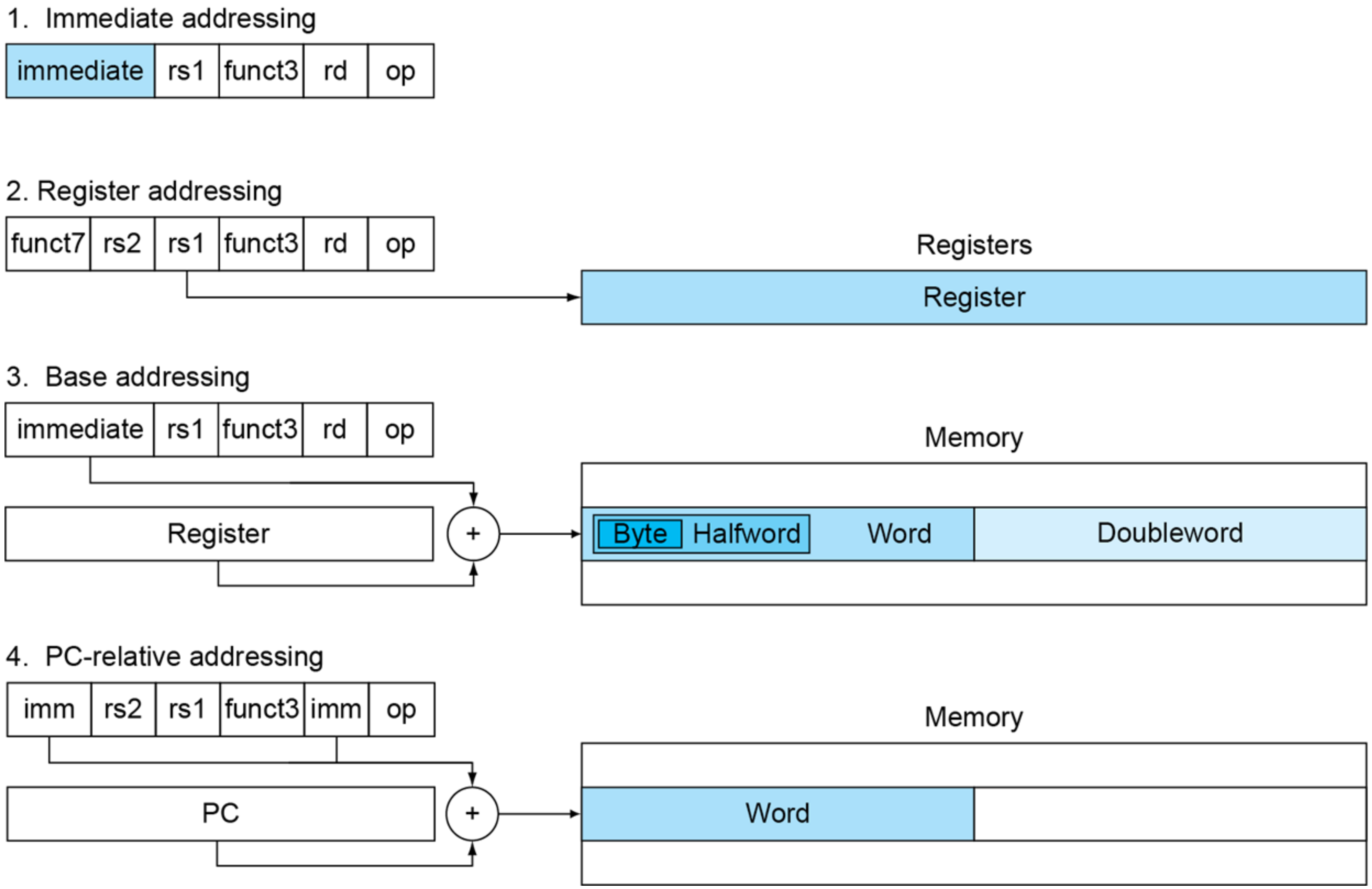
1. ***Immediate addressing***, where the operand is a constant within the instruction itself.

2. ***Register addressing***, where the operand is a register.

3. ***Base* or *displacement addressing***, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.

4. ***PC-relative addressing***, where the branch address is the sum of the PC and a constant in the instruction.

RISC-V Addressing Summary



RISC-V Encoding Summary

Name	Field						Comments
(Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format