Instructions: The Language of Computers

Dr. Pavitra Y J

Introduction

- The “instruction set architecture” (ISA) is an abstract interface between the hardware and the lowest level software that encompasses all the information necessary to write a machine language program that will run correctly
- The words of a computer’s language are called instructions, and its vocabulary is called an instruction set

Introduction

- RISCV
- MIPS(RISCV follows similar design)
- X86

Introduction

- It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations.... The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.
- Burks, Goldstine, and von Neumann, 1947

Instruction set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
- But with many aspects in common
- Early computers had very simple instruction sets

-Simplified implementation

- Many modern computers also have simple instruction sets

Introduction

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.

Operations of the computer

Hardware

- There must certainly be instructions for performing the fundamental arithmetic operations.
- Burks, Goldstine, and von Neumann, 1947
- The RISC-V assembly language notation
- add a, b, c instructs a computer to add the two variables b and c and to put their sum in a.

Operations of the computer

How to add 4 variables

add a, b, c // The sum of b and c is placed in a

add a, a, d // The sum of b, c, and d is now in a

add a, a, e // The sum of b, c, d, and e is now in a

Design Principle 1: Simplicity favors regularity

Compiling Two C Assignment Statements into RISC-V

- This segment of a C program

```
a = b + c;
d = a −e;
```

- This segment of a RISCV program

```
add a, b, c
sub d, a, e
```

Design Principle 1: Simplicity favors regularity

- Compiling a Complex C Assignment into RISC-V

f = (g + h) −(i + j); What might a C compiler produce?

- RISCV

add t0, g, h // temporary variable t0 contains g + h

add t1, i, j // temporary variable t1 contains i + j

sub f, t0, t1 // f gets t0 −t1, which is (g + h) −(i + j)

Operands of the Computer Hardware

- Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called registers
- The size of a register in the RISC-V architecture is 64 bits; groups of 64 bits occur so frequently that they are given the name doubleword in the RISC-V architecture.
- Registers are limited

Design Principle 2: Smaller is faster

- A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther

Compiling a C Assignment Using Registers

- f = (g + h) −(i + j);

add x5, x20, x21// register x5 contains g + h

add x6, x22, x23// register x6 contains i + j

sub x19, x5, x6// f gets x5 –x6, which is (g + h)−(i + j)

Memory Operands

- Processor can keep only small amount of data in registers
- computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.
- data transfer instructions- transfer data between memory and registers
- Memory is just a large, single-dimensional array, with the address acting as the index to that array

Memory Operands

- Data transfer instruction

-A command that moves data between memory and registers

- Address

-A value used to delineate the location of a specific data element within a memory array

Memory Operands

- The data transfer instruction that copies data from memory to a register is traditionally called load

Let's assume that A is an array of 100 doublewords and that the compiler has associated the variables g and h with the registers x20 and x21 as before. Let's also assume that the starting address, or base address, of the array is in x22. Compile this C assignment statement: g = h + A[8];

ld x9, 8(x22) // Temporary reg x9 gets A[8]

add x20, x21, x9 // g = h + A[8]

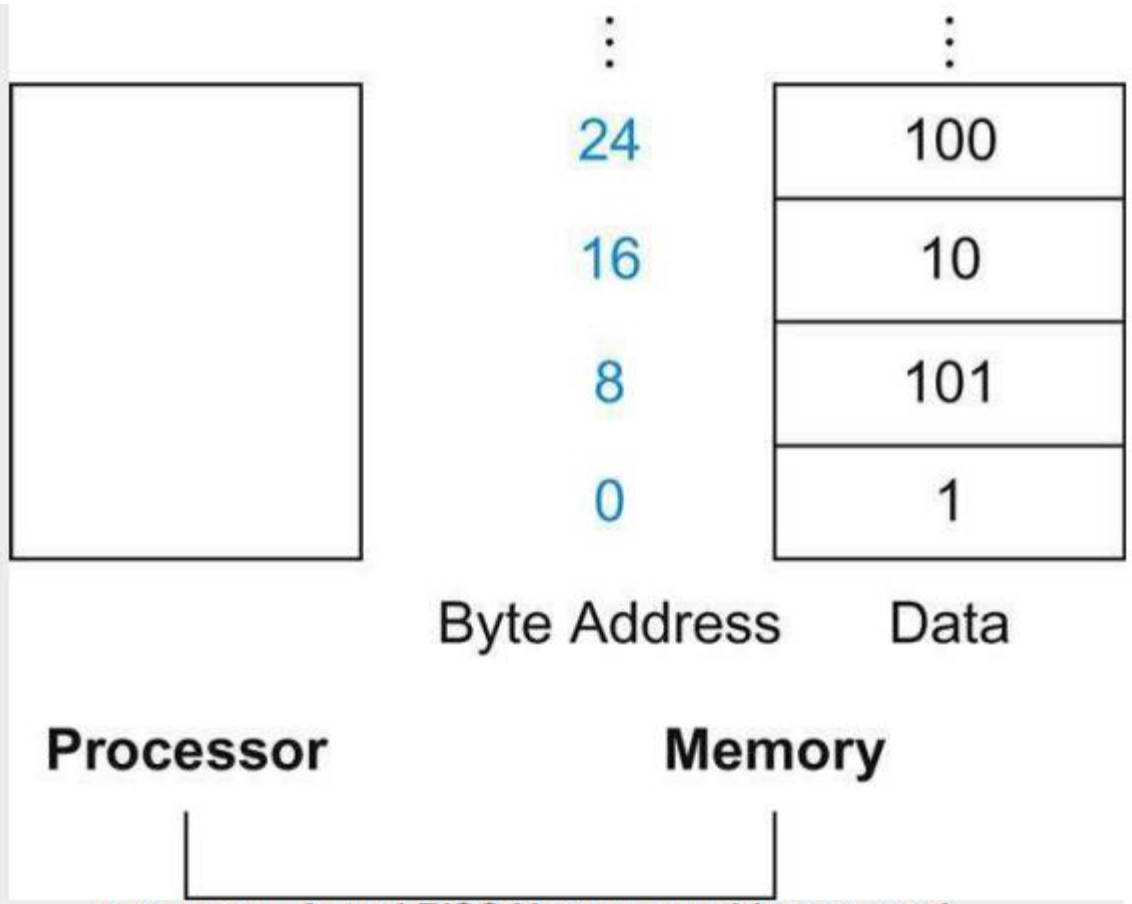


FIGURE 2.3 Actual RISC-V memory addresses and contents of memory for those doublewords.

Endianness

- Little endian
- Big endian

Memory Operands

- Store copies data from a register to memory
- In many architectures, words must start at addresses that are multiples of 4 and doublewords must start at addresses that are multiples of 8. This requirement is called an alignment restriction

Compiling Using Load and Store

- Assume variable h is associated with register x21 and the base address of the array A is in x22. What is the RISC-V assembly code for the C assignment statement below? A[12] = h + A[8];

```
ld x9, 64(x22) // Temporary reg x9 gets A[8]

add x9, x21, x9 // Temporary reg x9 gets h + A[8]

sdx9, 96(x22)// Stores h + A[8] back into A[12]
```

Constant or Immediate Operands

ld x9, AddrConstant4(x3) // x9 = constant 4

add x22, x22, x9 // x22 = x22 + x9 (where x9 == 4)

or

addi x22, x22, 4 // x22 = x22 + 4

- Although the RISC-V registers are 64 bits wide, the RISC-V architects conceived multiple variants of the ISA. In addition to this variant, known as RV64, a variant named RV32 has 32-bit registers, whose reduced cost make RV32 better suited to very low-cost processors

Signed and Unsigned numbers

- Binary digit
- Also called bit. One of the two numbers in base 2, 0 or 1, that are the components of information

$$1011_{two}$$

represents

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten}$$
$$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten}$$
$$= 8+0+2+1_{ten}$$
$$= 11_{ten}$$

Signed and Unsigned numbers

(64 bits wide, split into two 32-bit rows)

- least significant bit- The rightmost bit in an RISC-V doubleword.
- most significant bit- The leftmost bit in an RISC-V doubleword.

Hardware Software Interface

- Hardware can be designed to add, subtract, multiply, and divide the binary bit patterns
- If the number that is the proper result of such operations cannot be represented by the rightmost hardware bits, overflow is said to have occurred
- It’s up to the programming language, the operating system, and the program to determine what to do if overflow occurs

Hardware Software Interface

- Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative
- sign and magnitude- separate bit for sign
- Shortcomings
- First, it’s not obvious where to put the sign bit. To the right? To the left? Early computers tried both
- Second, adders for sign and magnitude may need an extra step to set the sign because we can’t know in advance what the proper sign will be
- Finally, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems for inattentive programmers

Hardware Software Interface

- Final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative.
- This convention for representing signed binary numbers is called two's complement representation
- hardware needs to test only MS bit to see if a number is positive or negative (with the number 0 is considered positive). This bit is often called the sign bit

$$(x_{63} \times -2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Binary to Decimal Conversion

- What is the decimal value of this 64-bit two's complement number?
- 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111100two

$$
\begin{aligned}
&(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \cdots + (1 \times 2^{1}) + (0 \times 2^{1}) + (0 \times 2^{0}) \\
&= -2^{63} + 2^{62} + 2^{61} + \cdots + 2^{2} + 0 + 0 \\
&= -9,223,372,036,854,775,808_{\text{ten}} + 9,223,372,036,854,775,804_{\text{ten}} \\
&= -4_{\text{ten}}
\end{aligned}
$$

Binary to Decimal Conversion

- Signed versus unsigned applies to loads as well as to arithmetic
- Lbu-load byte unsigned
- Lb- load byte(signed)
- Negate 2ten, and then check the result by negating −2ten.

Binary to Decimal Conversion

2_{ten}=00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010$_{two}$

Negating this number by inverting the bits and adding one,

$$\begin{array}{rl} & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} \\ + & 1_{two} \\ \hline = & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} \\ = & -2_{ten} \end{array}$$

Going the other direction,

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111110$_{two}$

is first inverted and then incremented:

$$\begin{array}{rl} & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} \\ + & 1_{two} \\ \hline = & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} \\ = & 2_{ten} \end{array}$$

Sign Extension Shortcut

- Convert 16-bit binary versions of 2ten and −2ten to 64-bit binary numbers

The 16-bit binary version of the number 2 is

$00000000\ 00000010_{two} = 2_{ten}$

It is converted to a 64-bit number by making 48 copies of the value in the most significant bit (0) and placing that in the left of the doubleword. The right part gets the old value:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

Creating a 64-bit version of the negative number means copying the sign bit 48 times and placing it on the left:

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -2_{ten}$

Sign Extension

Check Yourself

What is the decimal value of this 64-bit two's complement number?

11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111000$_{two}$

1) -4_{ten}
2) -8_{ten}
3) -16_{ten}
4) $18,446,744,073,709,551,609_{ten}$

Signed numbers

one's complement

A notation that represents the most negative value by $10 \ldots 000_{two}$ and the most positive value by $01 \ldots 11_{two}$, leaving an equal number of negatives and positives but ending up with two zeros, one positive ($00 \ldots 00_{two}$) and one negative ($11 \ldots 11_{two}$). The term is also used to mean the inversion of every bit in a pattern: 0 to 1 and 1 to 0.

Signed numbers

biased notation

A notation that represents the most negative value by $00 \ldots 000_{two}$ and the most positive value by $11 \ldots 11_{two}$, with 0 typically having the value $10 \ldots 00_{two}$, thereby biasing the number such that the number plus the bias has a non-negative representation.

Representing Instructions in the Computer

- A form of representation of an instruction composed of fields of binary numbers
- RISC-V instruction takes exactly 32 bits—a word, or one half of a doubleword
- RISC-V instructions are all 32 bits long
- Binary representation used for communication within a computer system- machine language

Instruction format

RISC-V – It's Instruction Format

32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

R-format Instruction Layout

RISC-V fields are given names to make them easier to discuss:

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

Here is the meaning of each name of the fields in RISC-V instructions:

- *opcode:* Basic operation of the instruction, and this abbreviation is its traditional name.
- *rd:* The register destination operand. It gets the result of the operation.
- *funct3:* An additional opcode field.
- *rs1:* The first register source operand.
- *rs2:* The second register source operand.
- *funct7:* An additional opcode field.

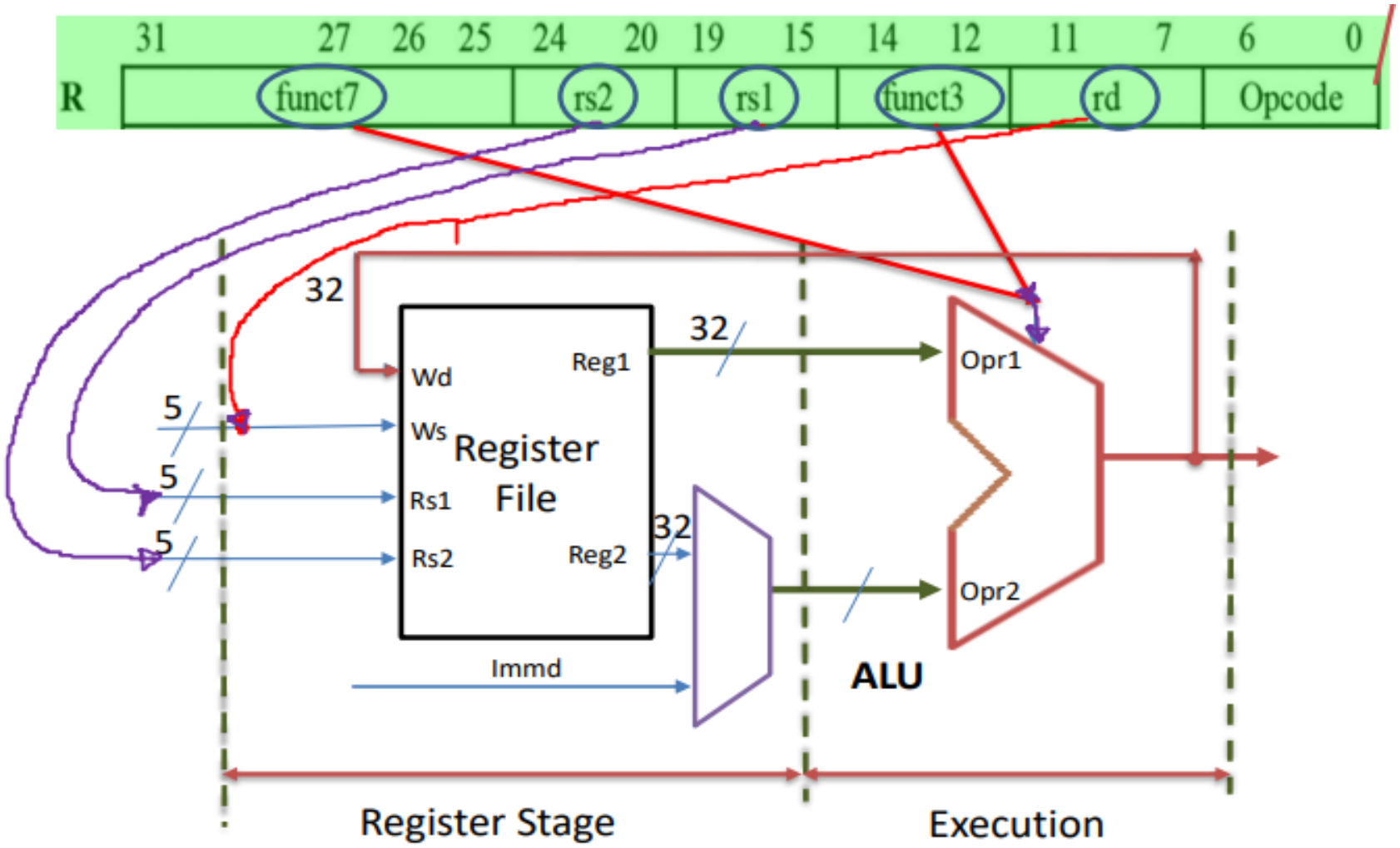
R-Type instructions

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

- `add rd, rs1, rs2 # rd = rs1 + rs2`
- `sub rd, rs1, rs2 # rd = rs1 - rs2`
- `sll rd, rs1, rs2 # rd = rs1 << rs2`
- `xor rd, rs1, rs2 # rd = rs1 ^ rs2`
- `srl rd, rs1, rs2 # rd = rs1 >> rs2`
- `sra rd, rs1, rs2 # rd = rs1 >>> rs2`
- `or rd, rs1, rs2 # rd = rs1 | rs2`
- `and rd, rs1, rs2 # rd = rs1 & rs2`

- Lr.d – load reserved doubleword and store reserved doubleword used during atomic operations

R-type instruction data path



- **funct7+ funct3 (10):** combined with opcode, these two fields describe what operation to perform
- How many R-format instructions can we encode?
- with opcode fixed at 0b0110011, just funct varies: $(2^7) \times (2^3) = (2^{10}) = 1024$.
- **rs1 (5):** 1st operand ("source register 1")
- **rs2 (5):** 2nd operand (second source register)
- **rd (5):** "destination register" — receives the result of computation

Design Principle 3: Good design demands good compromises

- All instruction representation to be of same length

Immediate format

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

Format	Instruction	Opcode	Funct3	Funct6/7
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.

Encoding of I –format(Different formats)

imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	slti
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
0000000	shamt	rs1	001	rd	0010011	slli
0000000	shamt	rs1	101	rd	0010011	srli
0100000	shamt	rs1	101	rd	0010011	srai

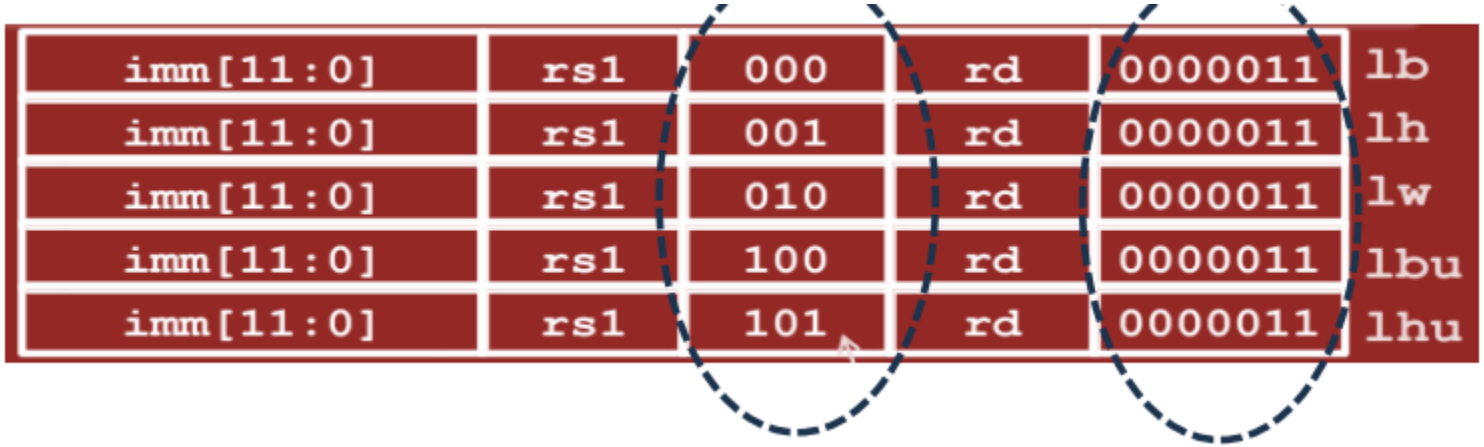
One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

NOT operation in RISC V

- A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1
- Designers of RISC V have included XOR instead of NOT
- XORin with ‘1’ complements the other source operand

I-format for load



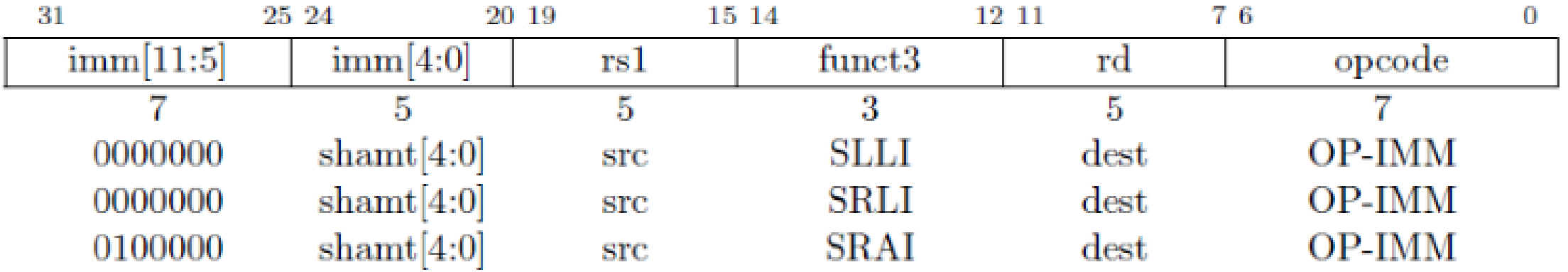
Syntax: mnemonics rd, imm(rs1)

Set less than immediate

- SLTI：SLTI rd,rs1,imm[11:0]
- S stands for Set, and its role is to set rd to 0 or 1
- If condition less than w.r.t immediate is satisfied rd is set with 1 else to 0
- RS1 holds 32 bit value and immediate is 12 bit, hence sign extended and then compared
- SLTIU is unsigned(12 bit extended with 0’s)

Shift instructions

Shift instruction



31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

Shift Instructions

1. SLLI rd,rs1,shamt[4:0]

SLLI： shift left logical by shamt(shift amount)， The number of shifts is determined by imm "4:0". This instruction means to shift the value of rs1 to the left in shamt[4:0], use 0 to fill the low bits of rs1, and write the result to Rd.

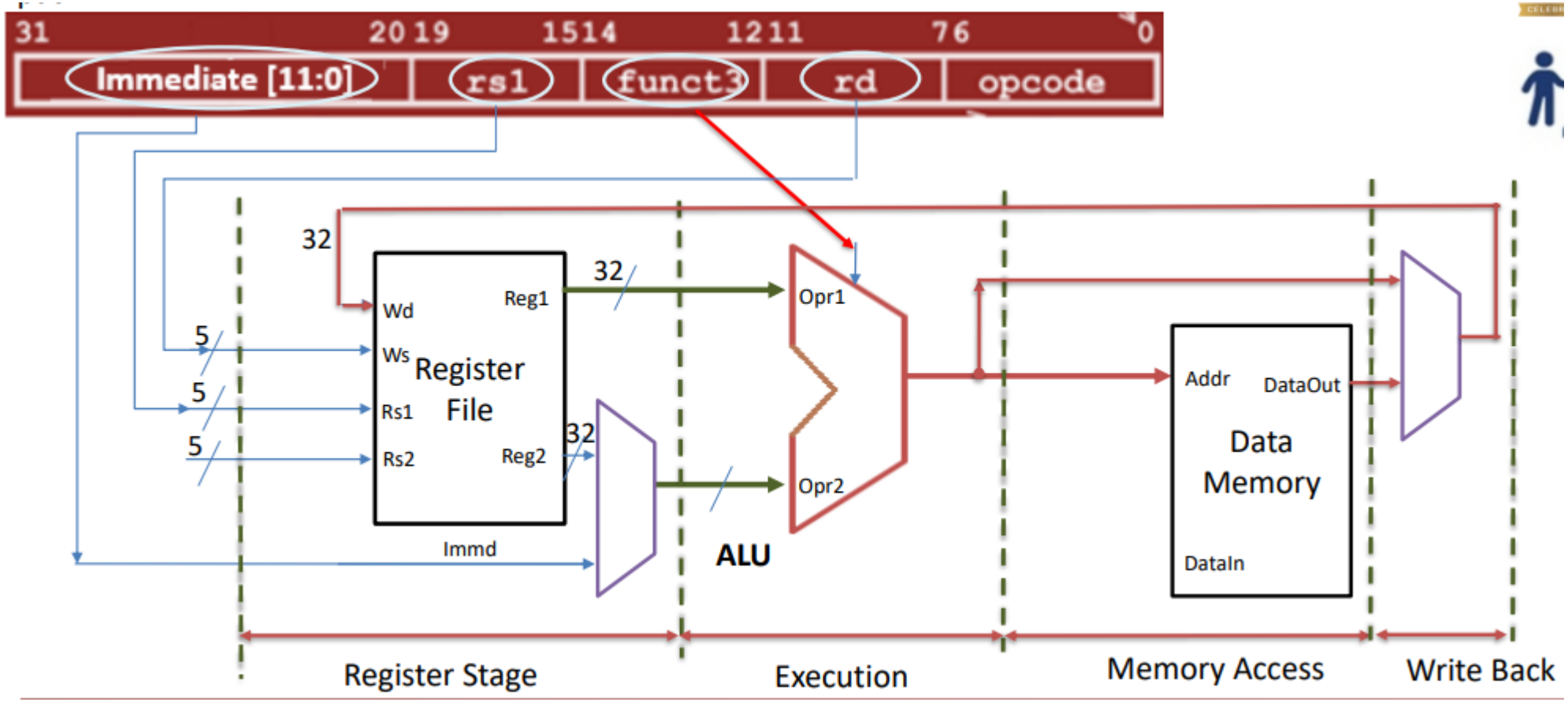
2. SRLI rd,rs1,shamt[4:0]

SRLI： shift right logical .Shift the value in rs1 to the right by shamt[4:0] bits, add zero to the high bits of rs1, and write the result to rd.

3. SRAI rd,rs1,shamt[4:0]

SRAI： Arithmetic shift right, shift the value in rs1 to right by shamt[4:0] bits, the high bits of rs1 are filled with the original rs1[31], and the result is written into rd.

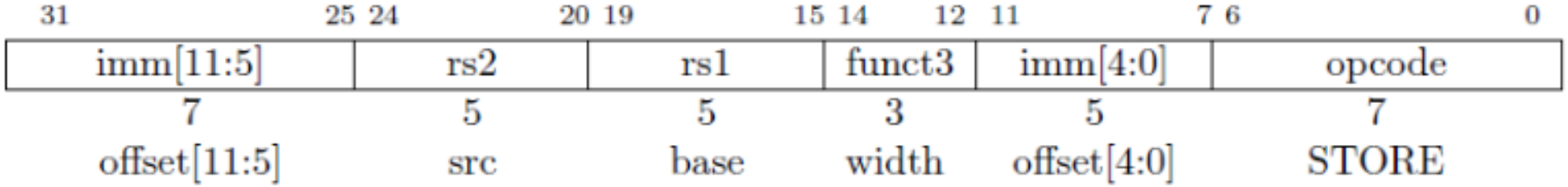
I-type instruction data path



S-type instruction layout

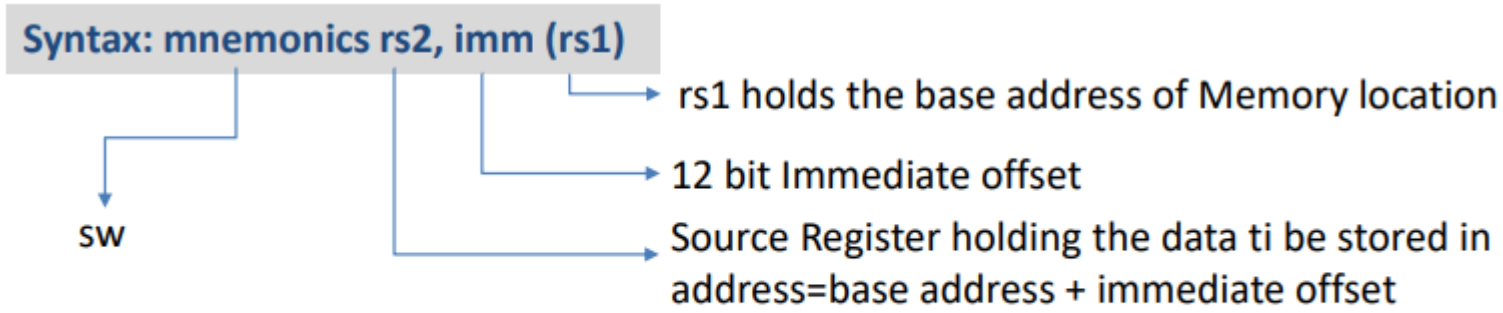
Format	Instruction	Opcode	Funct3	Funct6/7
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.

STORE



S-type instruction layout

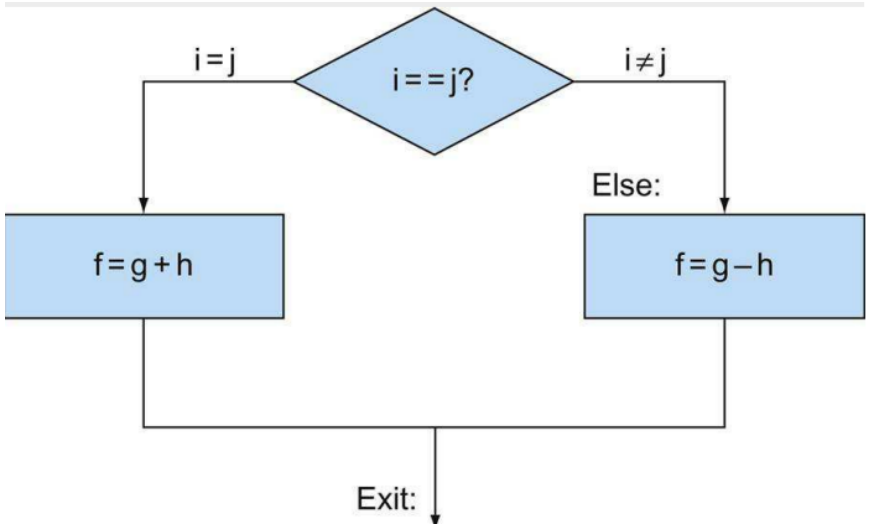
- Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!
- Can’t have both rs2 and immediate in same place as other instructions
- Note that stores doesn’t write a value to the register file, no rd!
- RISC-V design decision is to move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place



Instructions for making decisions

- Decision making is commonly represented in programming languages using the if statement, sometimes combined with go to statements and labels
- RISC-V assembly language includes two decision-making instructions
- beq rs1, rs2, L1 (branch if equal)
- bne rs1, rs2, L1 (branch if not equal)

- In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers x19 through x23, what is the compiled RISC-V code for this C if statement?
- if (i == j) f = g + h; else f = g −h;



```
bne x22, x23, Else // go to Else if i ≠ j
add x19, x20, x21 // f = g + h (skipped if i ≠ j)
beq x0, x0, Exit // if 0 == 0, go to Exit
Else:sub x19, x20, x21 // f = g −h (skipped if i = j)
Exit:
```

Loops

A traditional loop in C:

```
while (save[i] == k)
 i += 1;
```

Assume that i and k correspond to registers x22 and x24 and the base of the array save is in x25. What is the RISC-V assembly code corresponding to this C code?

Loops : for iterating a computation

Here is a traditional loop in C:

```
while (save[i] == k)
i += 1;
```

Assume that **i and k correspond to registers x22 and x24** and the **base of the array save is in x25.** What is the RISC-V assembly code corresponding to this C code?

Address	Instruction
0x8000	loop: slli x10, x22, 2 // x10 = x22 * 4 = i * 4
0x8004	add x10,x10,x25
0x8008	lw x9, 0(x10) // x9=save[i]
0x800C	bne x9,x24,exit
0x8010	addi x22,x22,1
0x8014	beq x0,x0,loop
0x8018	exit

If i=3

Then, address of

save[i=3] = base address +i * 4; where 4 is offset for a 32 bit data

save[i] address = x25+ i* 4 = x25+12

Branch

- The full set of comparisons is less than (<), or less than equal (<=), greater than(>) or greater than equal (≥), equal (=), and not equal (≠).
- Separate branch to handle signed and unsigned numbers (bgtu, bltu, bgeu)
- An alternative to providing these additional branch instructions is to set a register based upon the result of the comparison, then branch on the value in that temporary register with the beq or bne instructions- MIPS (datapath simple but more instructions to express program)
- Condition code/ flags in register- ARM (difficult for pipelined execution)

Case/Switch Statement

- The simplest way to implement switch is via a sequence of conditional tests, turning the switch statement into a chain of if-then-else statements
- the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a branch address table or branch table, and the program needs only to index into the table and then branch to the appropriate sequence.
- In RISC-V, the jump-and-link register instruction (jalr) serves this purpose

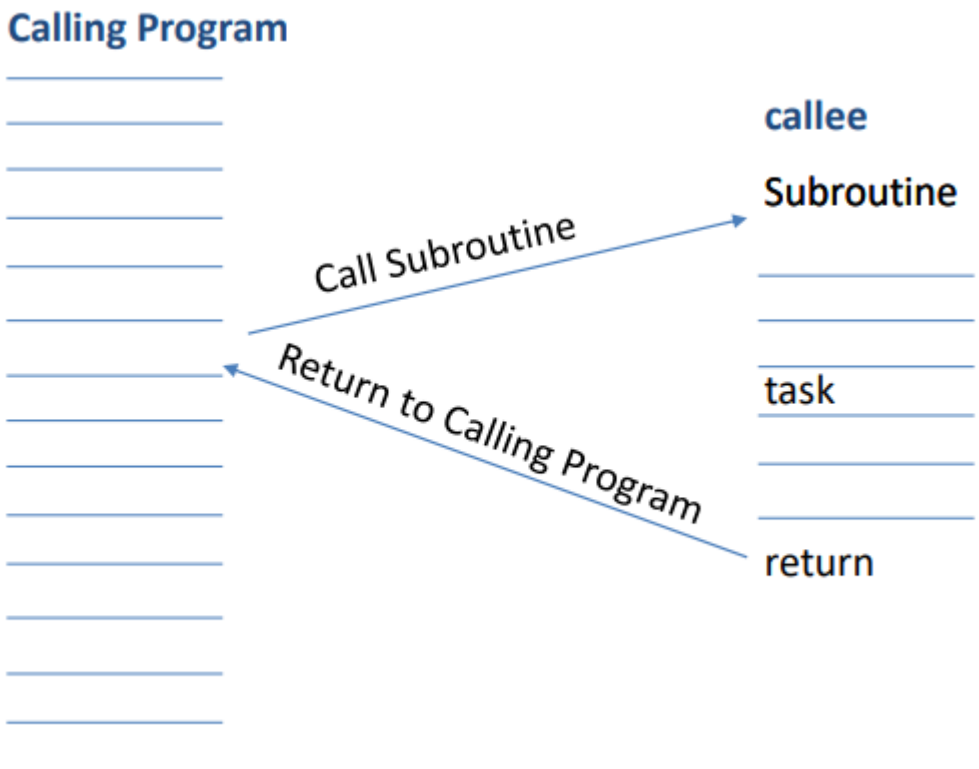
branch address table

- Also called branch table. A table of addresses of alternative instruction sequences called as branch address table or branch table.
- Program needs to index into the branch table and then branch to appropriate sequence
- RISC V supports this operation using indirect jump instruction which performs unconditional branch to the specified address in the register.
- Ex: jalr- jump and link register instruction

Supporting Procedures in Computer Hardware

- Procedure is a stored subroutine that performs a specific task based on the parameters with which it is provided
- Procedures allow the programmer to concentrate on just one portion of the task at a time
- parameters act as an interface between the procedure and the rest of the program and data (they can pass values and return results)

Procedure



- A procedure basically
  - ✓ **Acquires resources,**
  - ✓ **performs the task,**
  - ✓ **covers his or her tracks, and**
  - ✓ **then returns to the point of origin with the desired result.**

- RISC-V software follows the following convention for procedure calling in allocating its 32 registers
- x10–x17: eight parameter registers in which to pass parameters or return values.
- x1: one return address register to return to the point of origin
- RISC-V assembly language includes an instruction just for the procedures: it branches to an address and simultaneously saves the address of the following instruction to the destination register rd

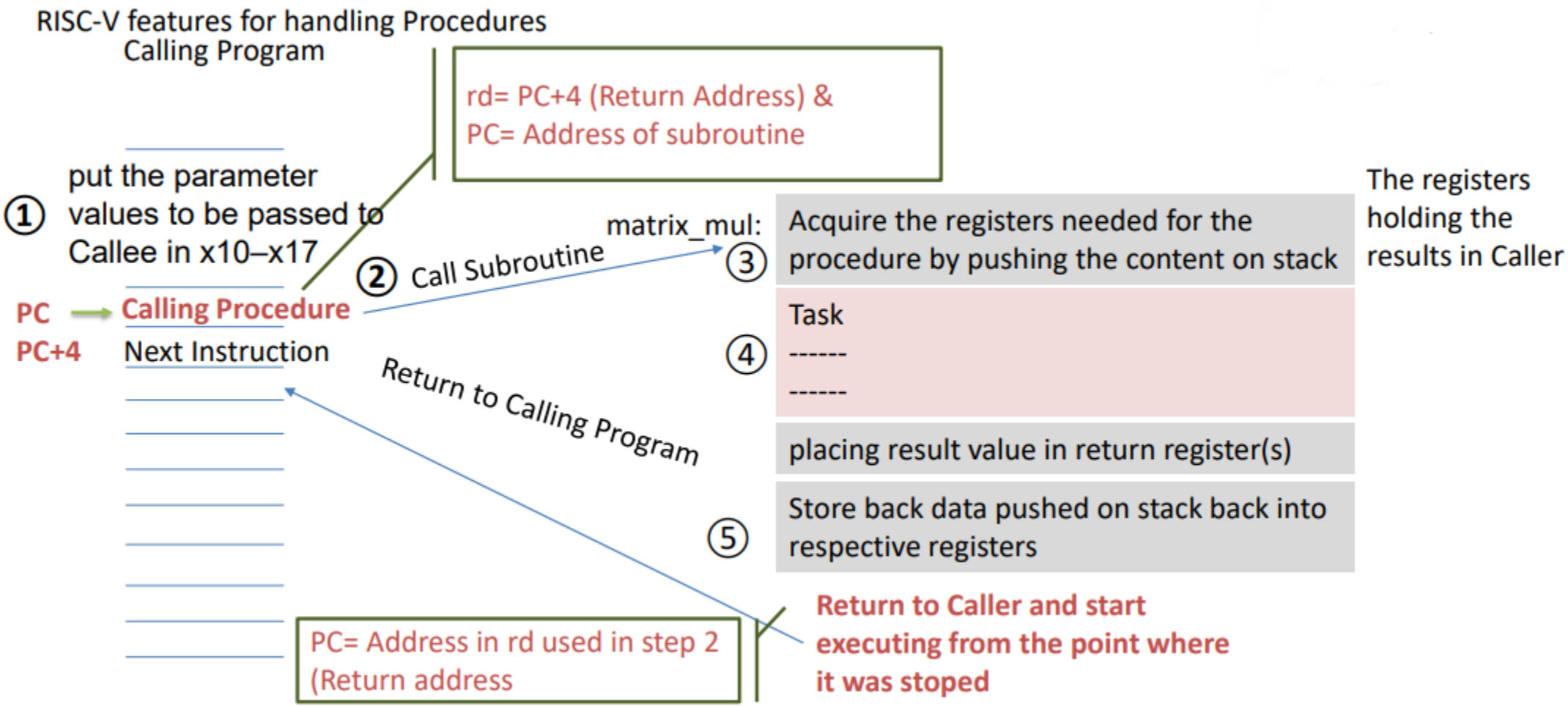
jump-and-link instruction

- An instruction that branches to an address and simultaneously saves the address of the following instruction in a register (usually x1 in RISC-V)
- jal x1, ProcedureAddress // jump to ProcedureAddress and write return address to x1
- return address- A link to the calling site that allows a procedure to return to the proper address; in RISC-V it is stored in register x1
- jalr x0, 0(x1)

Procedure

- Caller- The program that instigates a procedure and provides the necessary parameter values
- Callee- A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller
- program counter (PC)- The register containing the address of the instruction in the program being executed

Procedure


RISC-V features for handling Procedures
Calling Program
rd= PC+4 (Return Address) &
PC= Address of subroutine
① put the parameter values to be passed to Callee in x10–x17
② Call Subroutine
matrix_mul:
③ Acquire the registers needed for the procedure by pushing the content on stack
The registers holding the results in Caller
PC
Calling Procedure
PC+4
Next Instruction
④ Task

---
placing result value in return register(s)
Return to Calling Program
⑤ Store back data pushed on stack back into respective registers
PC= Address in rd used in step 2
(Return address
Return to Caller and start executing from the point where it was stoped

Using more registers

- When a procedure needs more registers than eight argument registers
- Spill registers to memory
- The ideal data structure for spilling registers is a stack—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. In RISC-V, it is register sp, or x2

- push Add element to stack
- pop Remove element from stack

- Write a RISC V assembly code for the following C procedure call

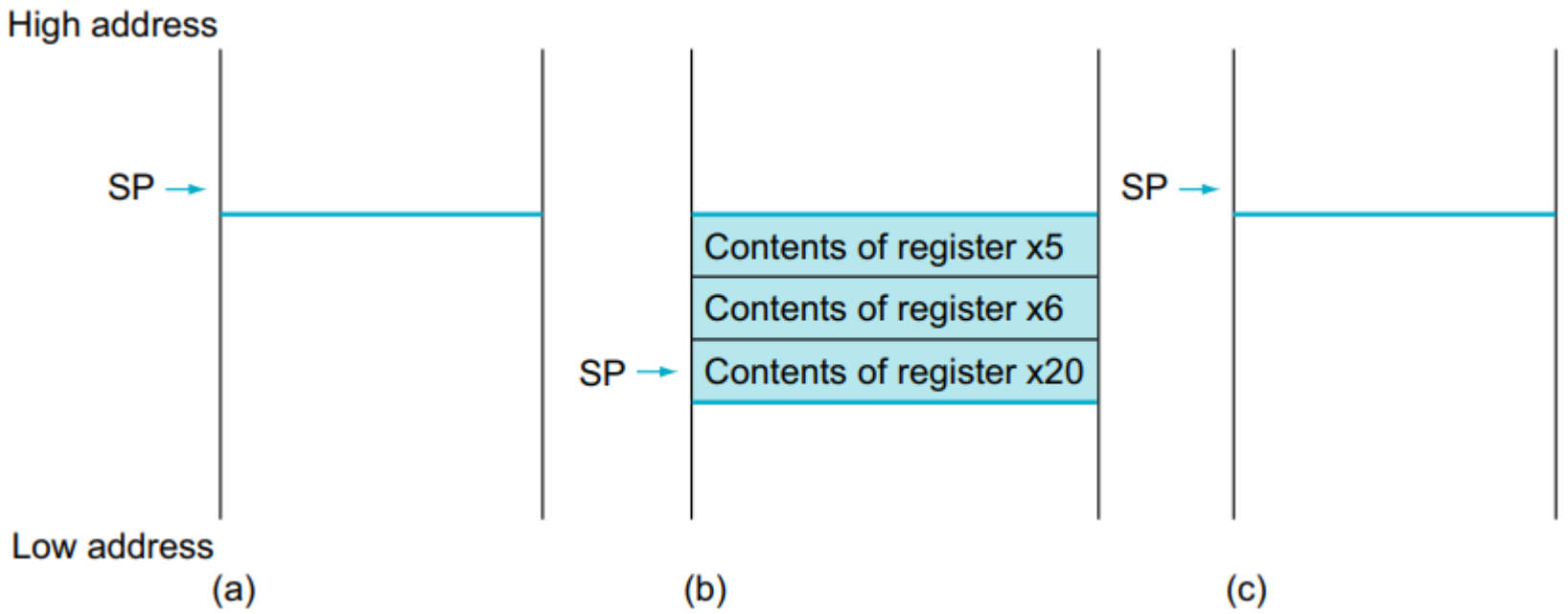
```
int leaf_example (int g, int h, int i, int j)
{
        int f;

        f = (g + h) – (i + j);
        return f;
}
```

The parameter variables g, h, i, and j correspond to the argument registers x10, x11, x12, and x13, respectively, and f corresponds to x20

- leaf example:

```
addi sp, sp, -12          // adjust stack to make room for 3 items
sw   x5, 8(sp)     // save register x5 for use afterwards
sw   x6, 4(sp)            // save register x6 for use afterwards
sw   x20, 0(sp)           // save register x20 for use afterwards
add x5, x10, x11   // register x5 contains g + h
add x6, x12, x13   // register x6 contains i + j
sub x20, x5, x6    // f = x5 - x6, which is (g + h) - (i + j)
addi x10, x20, 0 // returns f (x10 = x20 + 0)
lw x20, 0(sp)     // restore register x20 for caller
lw x6, 4(sp)      // restore register x6 for caller
lw x5, 8(sp)      // restore register x5 for caller
addi sp, sp, 12  // adjust stack to delete 3 items
jalr x0, 0(x1)        // branch back to calling routine
```



The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the “top” of the stack

- To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, RISC-V software separates 19 of the registers into two groups:
- x5–x7 and x28–x31: temporary registers that are not preserved by the callee (called procedure) on a procedure call
- x8–x9 and x18–x27: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

Nested Procedures

- Procedures that do not call others are called leaf procedures
- procedures invoke other procedures- Nested procedures
- If not taken care properly, caller will not be able to return to callee

Nested Procedure

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

What is the RISC-V assembly code?

fact:

```
   addi sp, sp, -8    // adjust stack for 2 items
     sw x1, 4(sp)     // save the return address
     sw x10, 0(sp)    // save the argument n
     addi    x5, x10, -1        // x5 = n - 1
     bge     x5, x0, L1         // if (n - 1) >= 0, go to L1
     addi   x10, x0, 1   // return 1
     addi   sp, sp, 8    // pop 2 items off stack
     jalr   x0, 0(x1)    // return to caller
L1:  addi x10, x10, -1  // n >= 1: argument gets (n – 1)
     jal  x1, fact      // call fact with (n – 1)
     addi  x6, x10, 0   // return from jal: move result of fact
                           (n - 1) to x6:
     lw    x10, 0(sp)   // restore argument n
     lw    x1, 4(sp)    // restore the return address
     addi  sp, sp, 8    // adjust stack pointer to pop 2 items
     mul     x10, x10, x6     // return n * fact (n – 1)
     jalr     x0, 0(x1)     // return to the caller
```

- frame pointer A value denoting the location of the saved registers and local variables for a given procedure.

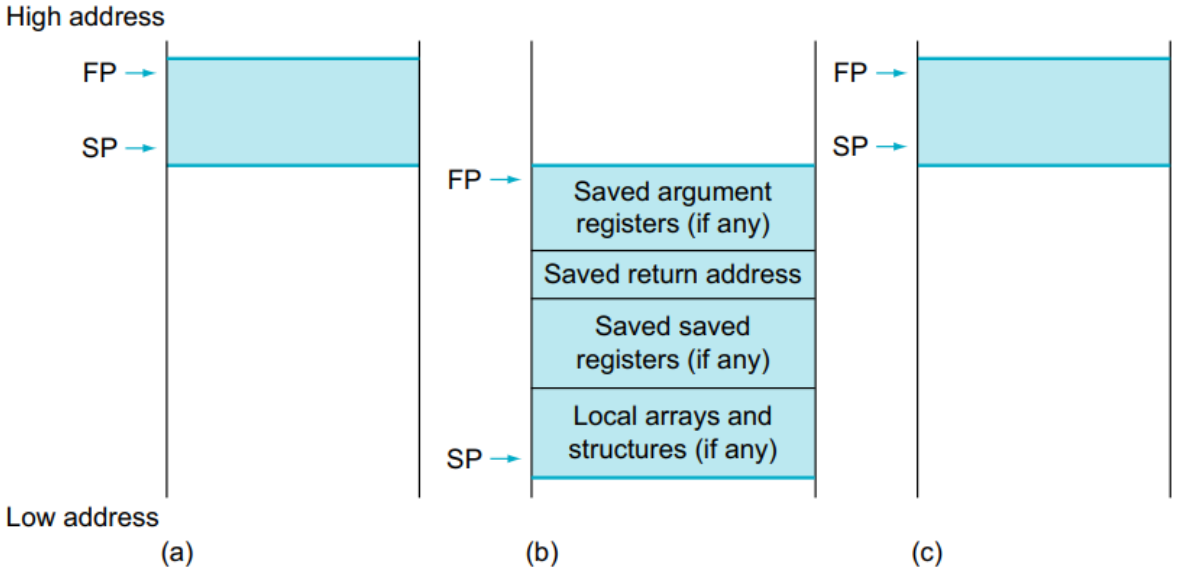


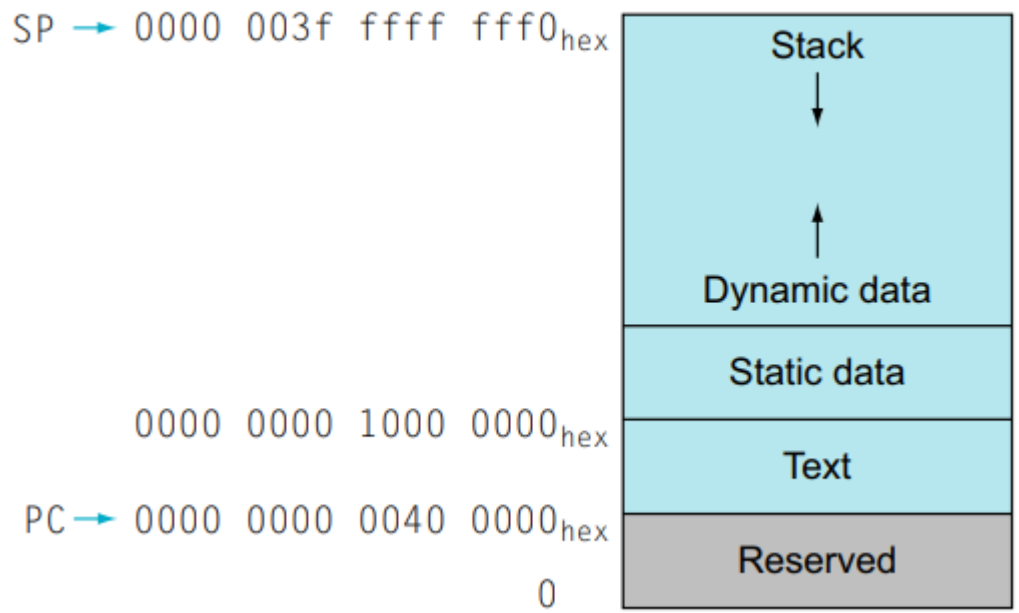
Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call

What is and what is not preserved across a procedure call

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

Allocating Space for New Data on the Heap

- The first part of the low end of memory is reserved, followed by the home of the RISC-V machine code, traditionally called the text segment
- text segment The segment of a UNIX object file that contains the machine language code for routines in the source file
- static data segment, which is the place for constants and other static variables
- data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the heap, and it is placed next in memory


SP → 0000 003f ffff fff0hex
Stack
Dynamic data
Static data
0000 0000 1000 0000hex
Text
PC → 0000 0000 0040 0000hex
Reserved
0

- What if there are more than eight parameters? The RISC-V convention is to place the extra parameters on the stack just above the frame pointer. The procedure then expects the first eight parameters to be in registers x10 through x17 and the rest in memory, addressable via the frame pointer

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

Recursion can be replaced with iteration

- Consider a procedure used to accumulate sum

```
int sum (int n, int acc) {
     if (n > 0)
          return sum(n - 1, acc + n);
     else
          return acc;
}
```

- assume x10 = n, x11 = acc, and the result goes into x12

```
sum: ble x10, x0, sum_exit   // go to sum_exit if n <= 0
     add x11, x11, x10       // add n to acc
     addi x10, x10, -1       // subtract 1 from n
     jal x0, sum             // jump to sum
sum_exit:
     addi x12, x11, 0        // return value acc
     jalr x0, 0(x1)          // return to caller
```

RISC-V Addressing for Wide Immediates and Addresses

- How to handle 32 bit constants?
- The RISC-V instruction set includes the instruction Load upper immediate (lui) to load a 20-bit constant into bits 12 through 31 of a register.
- The rightmost 12 bits are filled with zeros
- lui uses a new instruction format, U-type, as the other formats cannot accommodate such a large constant

- What is the RISC-V assembly code to load this 32-bit constant into register x19?

00000000 00111101 00000101 00000000

lui x19, 976 // 976decimal = 0000 0000 0011 1101 0000

addi x19, x19, 1280 // 1280decimal = 00000101 00000000

- The final value in register x19 is the desired value: 00000000 00111101 00000101 00000000

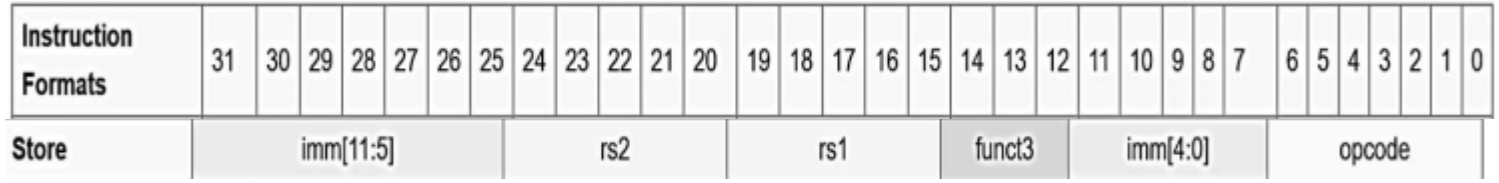
- bit 11 of the constant was 0. If bit 11 had been set, there would have been an additional complication: the 12-bit immediate is sign-extended, so the addend would have been negative. This means that in addition to adding in the rightmost 11 bits of the constant, we would have also subtracted 2^12. To compensate for this error, it suffices to add 1 to the constant loaded with lui, since the lui constant is scaled by 2^12

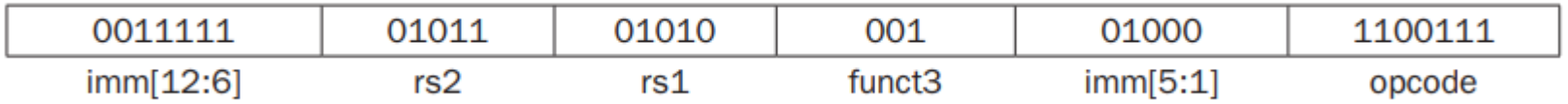
Addressing in Branches

- The RISC-V branch instructions use an RISC-V instruction format with a 12-bit immediate(branch addresses from −4096 to 4094, in multiples of 2)

bne x10, x11, 2000 // if x10 != x11, go to location 2000ten = 0111 1101 0000

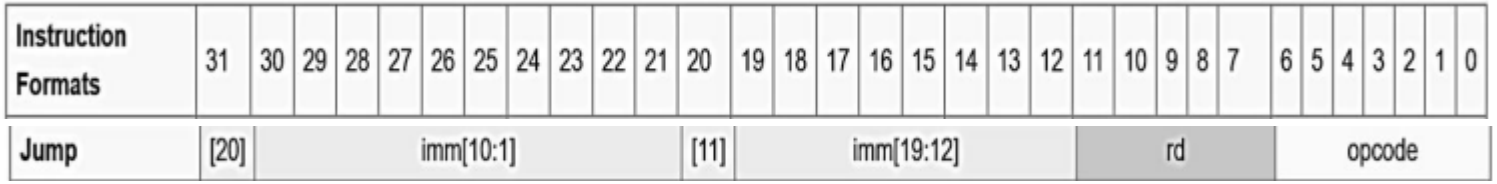
- With S-type





- The address uses an unusual encoding, which simplifies datapath design but complicates assembly

- The unconditional jump-and-link instruction (jal) uses an instruction format with a 12-bit immediate
- jal x0, 2000 // go to location 2000ten = 0111 1101 0000



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode							

00000000001111101000	00000	1101111
imm[20:1]	rd	opcode

- If addresses of the program had to fit in this 20-bit field, it would mean that no program could be bigger than 2^20, which is far too small to be a realistic option
- An alternative would be Program counter =Register+ Branch offset
- This sum allows the program to be as large as 2^32 and still be able to use conditional branches, solving the branch address size problem
- PC is the register to be used as it can handle all loops and if statements. This form of branch addressing is called PC-relative addressing

- To support branches upto 2^32, RISC V allows very long jumps to any 32- bit address with a two-instruction sequence: lui writes bits 12 through 31 of the address to a temporary register, and jalr adds the lower 12 bits of the address to the temporary register and jumps to the sum
- Even though PC relative address represent number of words, RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long, so the branch instructions represent the number of halfwords between the branch and the branch target

- Thus, the 20-bit address field in the jal instruction can encode a distance of ±2^19 halfwords, or ±1 MiB from the current PC. Similarly, the 12-bit field in the conditional branch instructions is also a halfword address, meaning that it represents a 13-bit byte address

I and k correspond to x22 and x24 save is x25

```
while (save[i] == k)
        i += 1;

Loop:slli x10, x22, 2     // Temp reg x10 = i * 4
     add  x10, x10, x25   // x10 = address of save[i]
     lw   x9, 0(x10)      // Temp reg x9 = save[i]
     bne  x9, x24, Exit   // go to Exit if save[i] != k
     addi x22, x22, 1     // i = i + 1
     beq  x0, x0, Loop    // go to Loop
Exit:
```

Address	Instruction					
80000	0000000	00010	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

- The bne instruction on the fourth line adds 3 words or 12 bytes to the address of the instruction, specifying the branch destination relative to the branch instruction (12 + 80012) and not using the full destination address (80024).
- The branch instruction on the last line does a similar calculation for a backwards branch (−20 + 80020), corresponding to the label Loop.

Branching Far Away

- Given a branch on register x10 being equal to zero,

beq x10, x0, L1

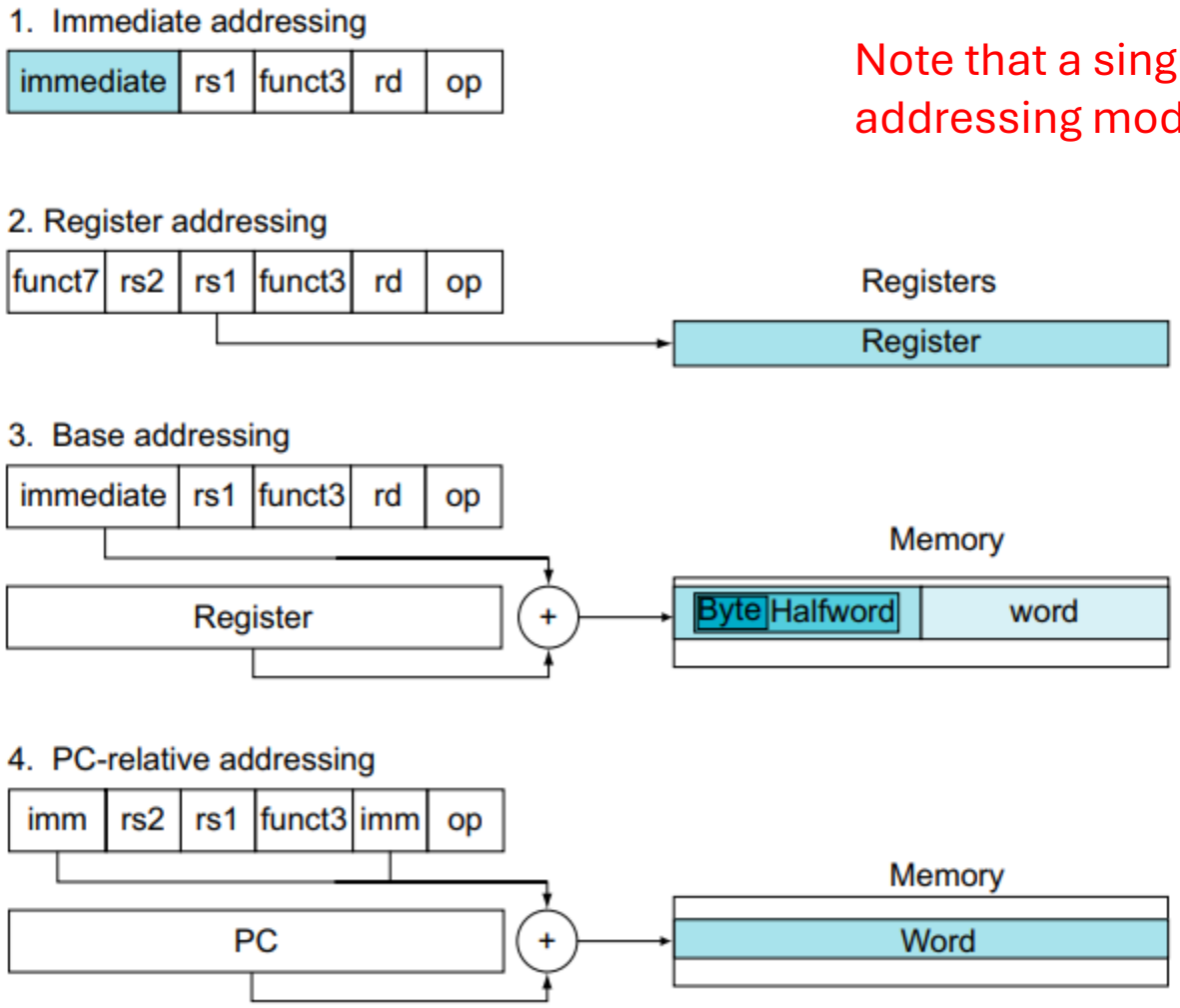
- replace it by a pair of instructions that offers a much greater branching distance. These instructions replace the short-address conditional branch:

bne x10, x0, L2

jal x0, L1

L2:

Illustration of 4 RISC V addressing



Note that a single operation can use more than one addressing mode

Decoding an instruction

- What is the assembly language statement corresponding to this machine instruction? 00578833hex

funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

- add x16, x15, x5
- To know instruction format knowing opcode is necessary