RISC V Architecture

Mahesh Awati
Department of Electronics and Communication Engg.

RISC V ARCHITECTURE

UNIT 2 – Instructions: The Language of Computer

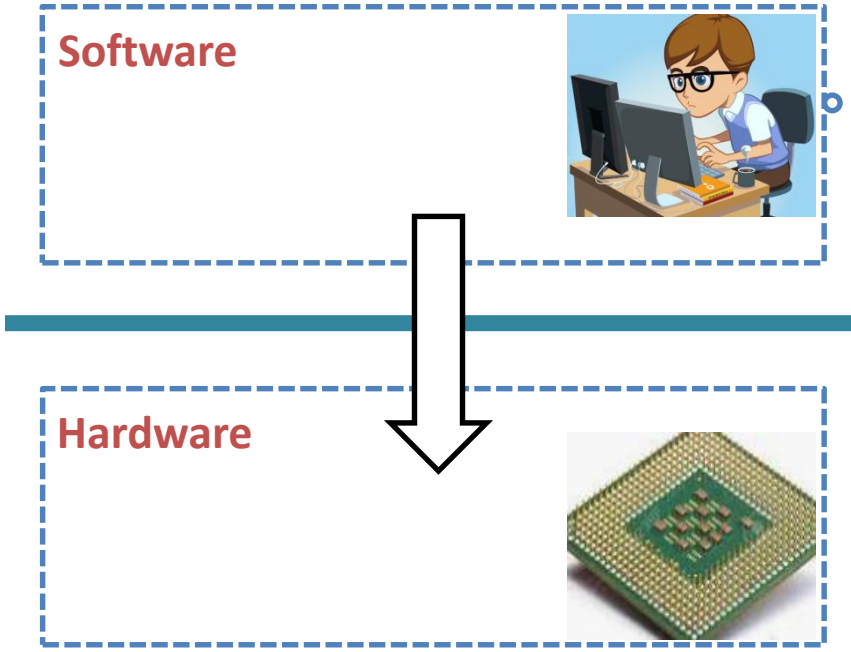
Mahesh Awati
Department of Electronics and Communication Engineering

Instructions – Language of Computer

Introduction

Computer hardware understands computer language called **Instructions**, and it's vocabulary is called an **Instruction set**

Starting from a **notation that looks like a restricted programming** language (written by people), we refine it step-by-step until you see the **actual language** of **a real computer** (read by the computer)



- Computers are slaves to our commands

Can we Imagine of writing commands in binary ?

- What computers understand is numbers in base 2, or *binary numbers*. **1001010100101110 – Function x**

Instructions – Language of Computer

Introduction

Computer hardware understands computer language called **Instructions**, and it's vocabulary is called an **Instruction set**

- **Different computers have different instruction sets.** But with **many aspects in common / similar**
- This **similarity** of instruction sets occurs because **all computers are constructed from hardware technologies based on similar underlying principles** and because there are a few basic operations that all computers must provide.
- Early computers had very **simple instruction sets**
  - Simplified implementation
- Many modern computers also have simple instruction sets

Computer designers have a common goal:

✓ To find a **language that makes** it **easy to build the hardware and the compiler** while maximizing performance and minimizing cost and energy.

Instructions – Language of Computer

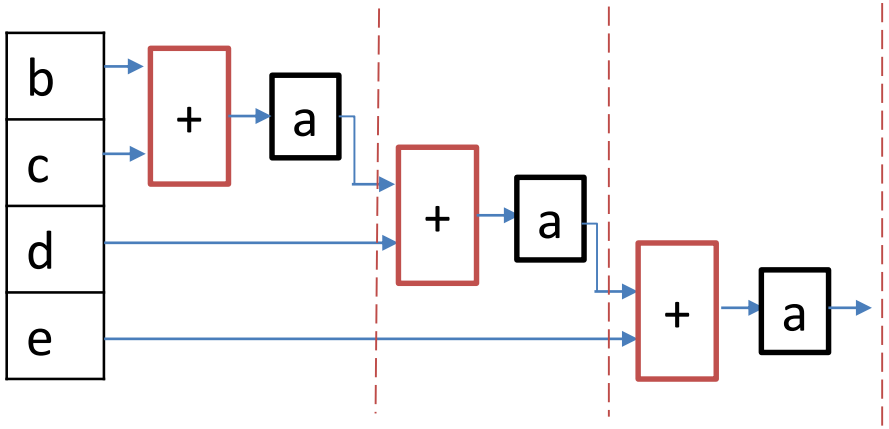
Operations of the Computer

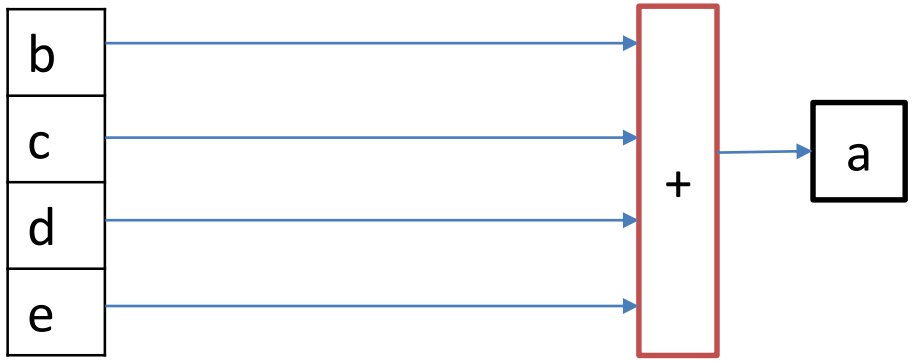
Hardware

- There must certainly be instructions for performing the fundamental arithmetic operations
- Every computer must be able to perform arithmetic. The RISC-V assembly language notation

  add a, b, c // Adds the two variables b and c and stores the sum in a
- Each RISC-V **arithmetic instruction performs only one operation and must always have exactly three variables.**
- Suppose we want to place the sum of four variables **b, c, d, and e** into **variable a**. How it can be done ?

Does RISC V perform
add a, b, c, d, e ?????





RISC V can't perform operation on more than two variables

Instructions – Language of Computer

Operations of the Computer

Hardware

- The natural number of operands for **an operation like addition is three:** the two numbers being added together and a place to put the sum.

Why Not more than two variables ???

- **Conforms to the philosophy of keeping the hardware simple**: hardware for a **variable number of operands is more complicated** than hardware for a fixed number.

- This situation illustrates the **first of three underlying principles of hardware design:**

Design Principle 1: Simplicity favours Regularity.

Instructions – Language of Computer

Operations of the Computer

Hardware

Relationship of programs written in higher-level programming languages to programs in this more primitive notation.

Compiling Two C Assignment Statements into RISC-V

Example

This segment of a C program contains the five variables `a`, `b`, `c`, `d`, and `e`. Since Java evolved from C, this example and the next few work for either high-level programming language:

```
a = b + c;
d = a -e;
```

The *compiler* translates from C to RISC-V assembly language instructions. Show the RISC-V code produced by a compiler.

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

Instructions – Language of Computer

Operations of the Computer

Hardware

Relationship of programs written in higher-level programming languages to programs in this more primitive notation.

Compiling a Complex C Assignment into RISC-V

Example

A somewhat complicated statement contains the five variables `f`, `g`, `h`, `i`, and `j`:

```
f = (g + h) −(i + j);
```

What might a C compiler produce?

```
f = (g + h) −(i + j);
```

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

Does Compiler generate Single Assembly Instruction ????

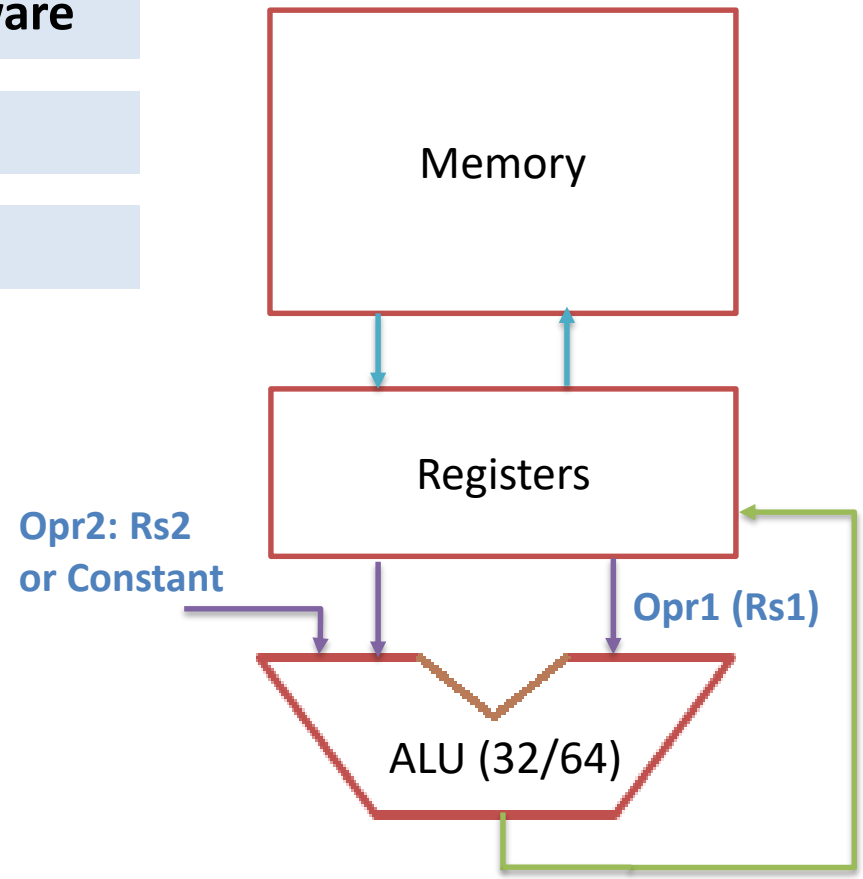
- The compiler must break this statement into several assembly instructions, since only one operation is performed per RISC-V instruction.
- The compiler creates a temporary variable, called t0,t1.. so on to store intermediate results

Instructions – Language of Computer

Operands of the Computer Hardware

Operand location: physical location in computer

1. Registers – Special Location built directly in Hardware
2. Memory Operands
3. Constants (also called immediate)



Instructions – Language of Computer

Operands of the Computer Hardware

1. Registers – Special Location built directly in Hardware

- Programming languages have **simple variables that contain single/few** data elements
- Operands in arithmetic instructions must be **limited number** of **special locations built directly in hardware called registers.**
- Registers are faster than memory

In RISC V

- The size of a register in the RISC-V architecture is 64 bits; known as double-word or 32 bit size known as word.
- Variables in programming Language can be of any number but **registers in a hardware are always limited**.
- Ex: In RISC V 64 bit processor – There are 32 , 64 bit registers.
- Ex: In RISC V 32 bit processor – There are 32 , 32 bit registers.

Instructions – Language of Computer

Operands of the Computer Hardware

Why there is a limit on number of registers ?

Design Principle 2: "Smaller is faster".

1) A very large number of registers may **increase the clock cycle time simply because it takes electronic signals longer time when they must travel farther.**
2) **The number of bits it would take in the instruction format to address a register.**
   Example: If number of registers is 32, then the instruction format will have 5 bit field to address the registers. i.e., if there are 3 operands then the Instruction format will have 3, 5 bit fields reserved.
   Increasing Number of registers means increase in bits required by these fields. This intern will increase size of Instructions

Design Principle 1: Simplicity favours Regularity.

Design Principle 2: "Smaller is faster".

R type Instruction Format

The RISC-V convention is **x followed by the number of the register**, except for a few register names that we will cover later

Instructions – Language of Computer

Operands of the Computer Hardware

RISC V Registers

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10-11	Function arguments / return values
a2–7	x12-17	Function arguments
s2–11	x18-27	Saved registers
t3–6	x28-31	Temporaries

- Registers can be used either name (i.e., ra, zero) or x0, x1, etc. Using name is preferred
- Registers used for specific purposes:
  - **Zero Register:** always holds the constant value 0.
  - **Saved Registers, s0-s11**: used to hold variables
  - **Temporary registers, t0-t6 :**used to hold intermediate values during a larger computation
  - **Function arguments/return values**
  - **Pointers** – Stack, Global, Thread

Instructions – Language of Computer

Operands of the Computer Hardware

Why there is a limit on number of registers ?

Design Principle 2: "Smaller is faster".

Compiling a Complex C Assignment into RISC-V

Example

A somewhat complicated statement contains the five variables `f`, `g`, `h`, `i`, and `j`:

```
f = (g + h) −(i + j);
```

What might a C compiler produce?

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Saved Registers	x8, x9, x18-x27
Temporary registers	x5-x7, x28-x31

Instructions – Language of Computer

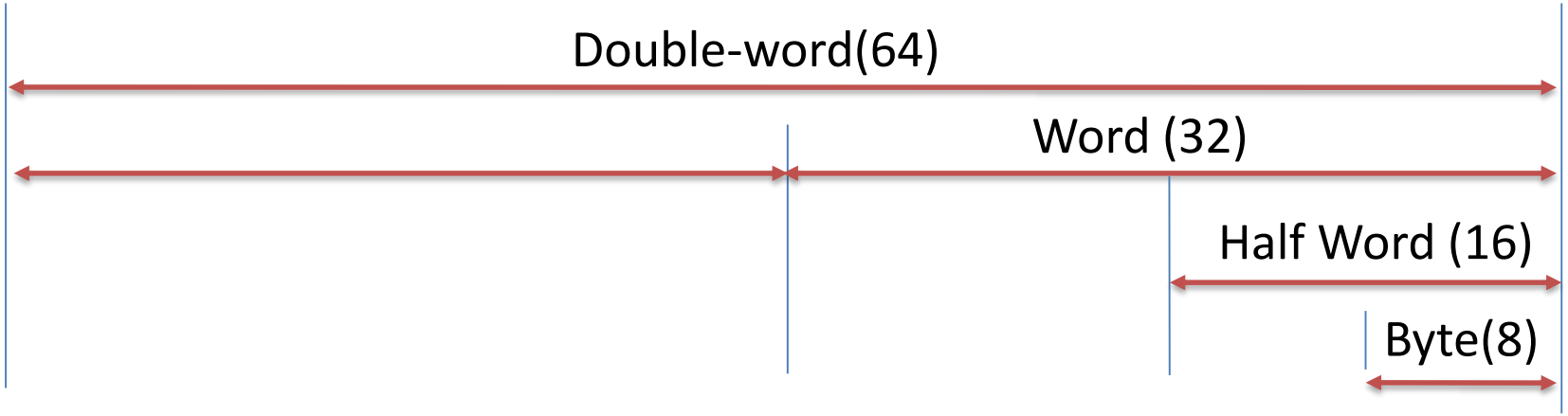
Operands of the Computer Hardware

Data types – Double word, Word, Half word and byte

0x0F	0x0E	0x0D	0x0C	0x0B	0x0A	0x09	0x08
15	14	13	12	11	10	9	8
63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0

← Address in Hexadecimal

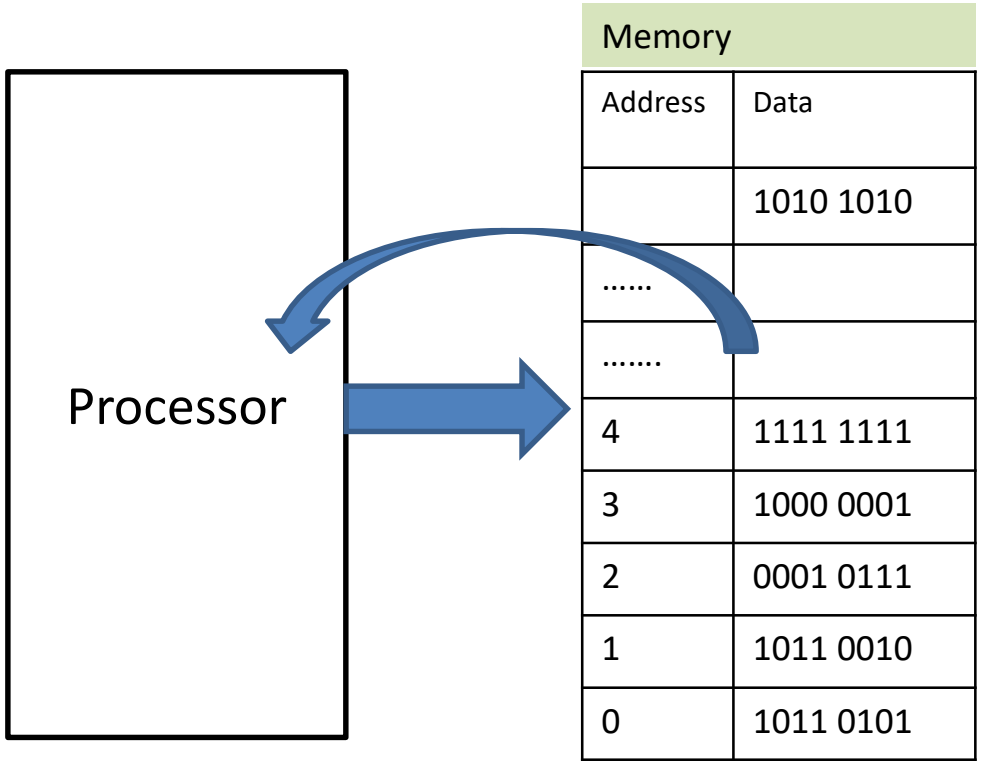
← Address in decimal



Operands of the Computer Hardware

2. Memory Operands

- If Programming languages have more complex data **structures i.e., arrays and structures.**
- The number of elements in these data structure **may be more than the registers available** in a computer hardware (to much to fit in 32 registers).
- **Memory is Large** but **slow**
- Memory is just a large, 1D array, with the address acting as the index to that array, starting at 0.
- Memory locations are addressed to access the data.



Instructions – Language of Computer

Operands of the Computer Hardware

How the memory is addressed ?

Word Addressable: What is Word addressable Memory?

- Each 32-bit data word has a unique address
- Using the unique address , a 32 bit data can be read/written and Next 32 bit data will be there is next successive location

Word No	Word Address	Data (32 bits = 4bytes) 31 24 23 16 15 8 7 0			
10	0x000A	AB	FD	12	34
	.....				
2	0x0002	12	03	DF	2D
1	0x0001	DD	FF	F2	2A
0	0x0000	12	AA	BB	B9h 1011 1001b

Instructions – Language of Computer

Operands of the Computer Hardware

How the memory is addressed ?

Byte Addressable: What is byte addressable Memory?

- Each data byte has a unique address (Individual bytes have individual address)
- A 32-bit word = 4 bytes, so **word address increments by 4**

Word No	Word Address	Data (32 bits = 4bytes) 31 24 23 16 15 8 7 0			
4	0x0010	AB	FD	12	34
3	0x000C				
2	0x0008	12	03	DF	2D
1	0x0004	DD	FF	F2	2A
0	0x0000	12	AA	BB	B9h

Order of Data Storage:

Little Endian: Base address holding byte0 of a word

Ex: Address 0x0004 = 0xDDFFF22A

Big Endian: Base address holding byte3 of a word

Ex: Address 0x0004 = 0x2AF2FFDD

Byte Address

0X0013	0X0012	0X0011	0X0010
0X000F	0X000E	0X000D	0X000C
0X000B	0X000A	0X0009	0X0008
0X0007	0X0006	0X0005	0X0004
0X0003	0X0002	0X0001	0X0000

Base Address of words

Instructions – Language of Computer

Operands of the Computer Hardware

How the memory is addressed ?

Byte Addressable: What is byte addressable Memory?

- Each data byte has a unique address (Individual bytes have individual address)
- A 64-bit double-word = 8 bytes, so **double-word address increments by 8**

D-Word No	Word Address	Data (32 bits = 4bytes) 64 15 8 7 0							
4	0x0020	AB	45	FD	ED	12	AA	34	13
3	0x0018								
2	0x0010	12	DA	03	55	DF	76	2D	AA
1	0x0008	DD	44	FF	AE	F2	0F	2A	11
0	0x0000	12	11	AA	A1	BB	F2	B9h	DE

Byte Address

0X27	0X26	0X25	0X24	0X23	0X22	0X21	0X20
0X1F	0X1E	0X1D	0X1C	0X1B	0X1A	0X19	0X18
0X17	0X16	0X15	0X14	0X13	0X12	0X11	0X10
0X0F	0X0E	0X0D	0X0C	0X0B	0X0A	0X09	0X08
0X07	0X06	0X05	0X04	0X03	0X02	0X01	0X00

Operands of the Computer Hardware

2. Memory Operands

How a computer represent and access such complex data **structures?**

- The **complex data structure (array and structure)** is not stored in registers , instead the compiler **allocates complex data structure to locations in memory**.
- The compiler then place the **starting address into the data transfer instructions**.
- Let us consider array A of 100 double-words stored in memory by a compiler. The starting address 0x10 is referred as base address in which A[0] is stored.

Starting Address of Array which is referred as base address

Element	Address	Data
99	0x330	A[99]
….		A[……]
8	0x58	A[8]
….	…….	Data (64 bit)
2	0x20	A[2]
1	0x18	A[1]
0	0x10	A[0]
	0x08	Data (64 bit)
0	0x00	Data (64 bit)

Instructions – Language of Computer

Operands of the Computer Hardware

2. Memory Operands

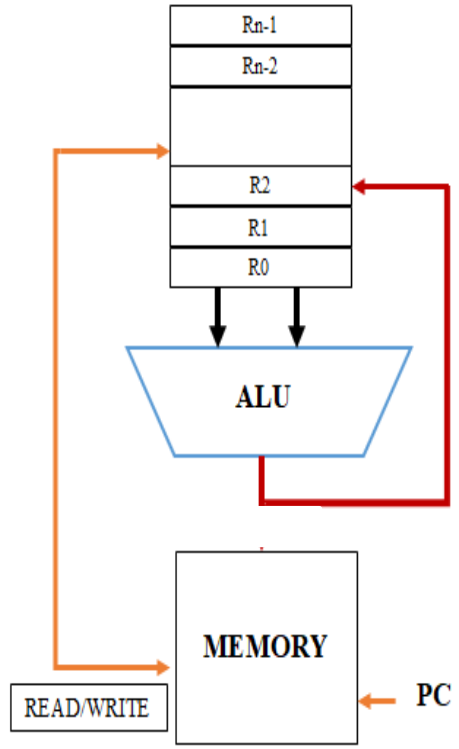
How RISC V represent and access such large structures?

- In RISC V supports The **load-store computer architecture** : Separates instructions into two types of operations.
  - Memory access operations (load and stores) and
  - operations that operate on the data in the register file (register to register or register to immediate).

- **By not combining memory accesses with data manipulation operations, the processor's complexity is reduced which enables making the common case fast,** one of the eight great ideas in computer architecture.

- Therefore, RISC-V must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions.**

- **Example : lw, sw, ld, sd and so on**

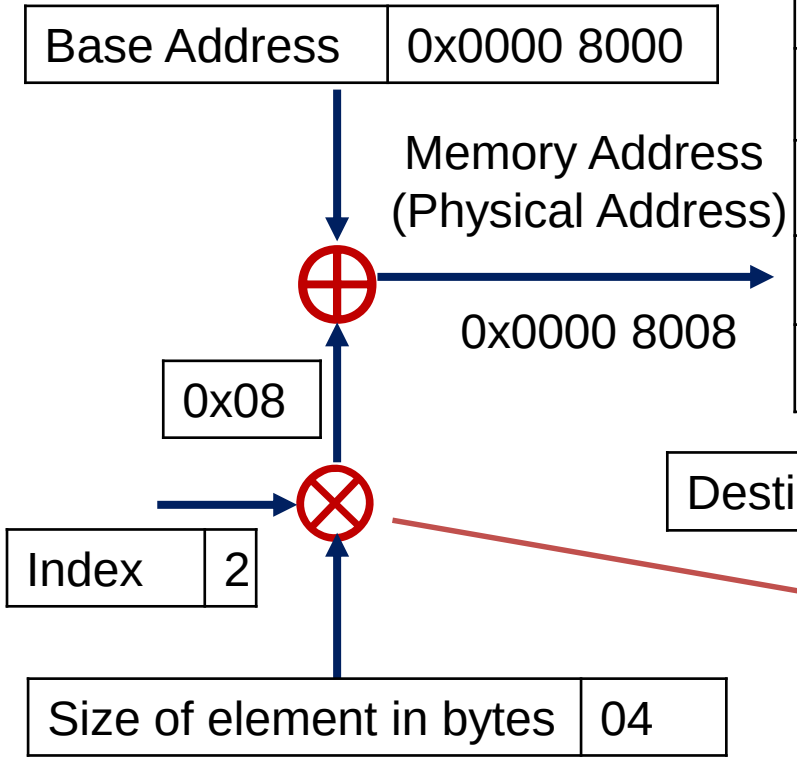


Instructions – Language of Computer

Operands of the Computer Hardware

2. Memory Operands

How a computer represent and access such complex data **structures?**



Bits → Address	31 24	23 16	15 8	7 0	Element
	Byte3	Byte2	Byte1	Byte0	
0x0000 8000	8F	34	3D	DD	0
0x0000 8004	67	DF	ED	11	1
0x0000 8008	77	66	FF	3D	2
0x0000 800C	11	22	DD	EE	3

Destination Register	77	66	FF	3D

This multiplication can be done by **shifting Index (if known) by 2 times (if data of 32 bit size) towards left.**

a) (0x02) <<2 = (0000 0010)b << 2 = (0000 1000)b= 0x08

Or Manually calculated and then taken as imm12 offset provident element number in the array is known

Instructions – Language of Computer

Operands of the Computer Hardware

Saved Registers	x8, x9, x18-x27
Temporary registers	x5-x7, x21-x31

2. Memory Operands

Load Instruction (lw) : Syntax - lw rd,imm12(rs1)

X22 (Base Address)	0x0000 8000

Memory Address (Physical Address)

0x0000 8020

Imm12=32d = 0x20

lw x9, 32 (x22)

Bits → Address	31 24	23 16	15 8	7 0	Element
	Byte3	Byte2	Byte1	Byte0	
0x0000 801B	8F	34	3D	DD	7
0x0000 8020	67	DF	ED	11	8
0x0000 8024	77	66	FF	3D	9
0x0000 8028	11	22	DD	EE	10

x9	67	DF	ED	11

The offset can be taken as imm12 by manually calculating
Imm12 = offset = Element Number x 4 = 8 x 4 =32=0x20

2. Memory Operands

Store Instruction (sw)

Store (sw) : It copies data from a register to memory.

Syntax: sw source reg, offset(base reg)

- Once again, the RISC-V address is specified in part by a constant and in part by the contents of a register
- sw stands fore store word

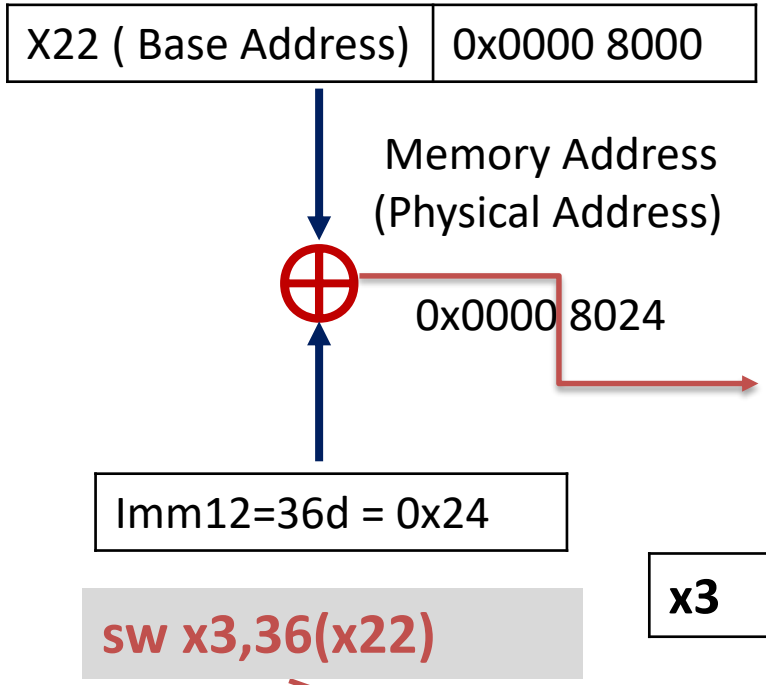
- Example: sw x9,36(x22)

Instructions – Language of Computer

Operands of the Computer Hardware

2. Memory Operands

Store Instruction (sw) : Syntax: sw rs2,imm12(rs1)



Bits → Address	31 24	23 16	15 8	7 0	Element
	Byte3	Byte2	Byte1	Byte0	
0x0000 801C	8F	34	3D	DD	7
0x0000 8020	67	DF	ED	11	8
0x0000 8024	67	DF	ED	11	9
0x0000 8028	11	22	DD	EE	10

x3	67	DF	ED	11

sw x3,36(x22)

The offset can be taken as imm12 by manually calculating
Imm12 = offset = Element Number x 4 = 8 x 4 =32=0x20

Operands of the Computer Hardware

2. Memory Operands

What is the RISC-V assembly code for the C assignment statement below?

A[12] = h + A[8];

Assume variable

h	Associated with x21
Base Address of Array A	Associated with x22

x21	h
x22	Base Address of array A

Address Hexadei	Address Decimal	Data (32bits)
0X2020	2032	A[8]
0X201B	2028	A[7]
0X2018	2024	A[6]
0X2014	2020	A[5]
0X2010	2016	A[4]
0X200B	2012	A[3]
0X2008	2008	A[2]
0X2004	2004	A[1]
0X2000	2000	A[0]

Operands of the Computer Hardware

2. Memory Operands

What is the RISC-V assembly code for the C assignment statement below?

g = h + A[8];

Assume variable

g	Associated with x20
h	Associated with x21
Base Address of Array A	Associated with x22

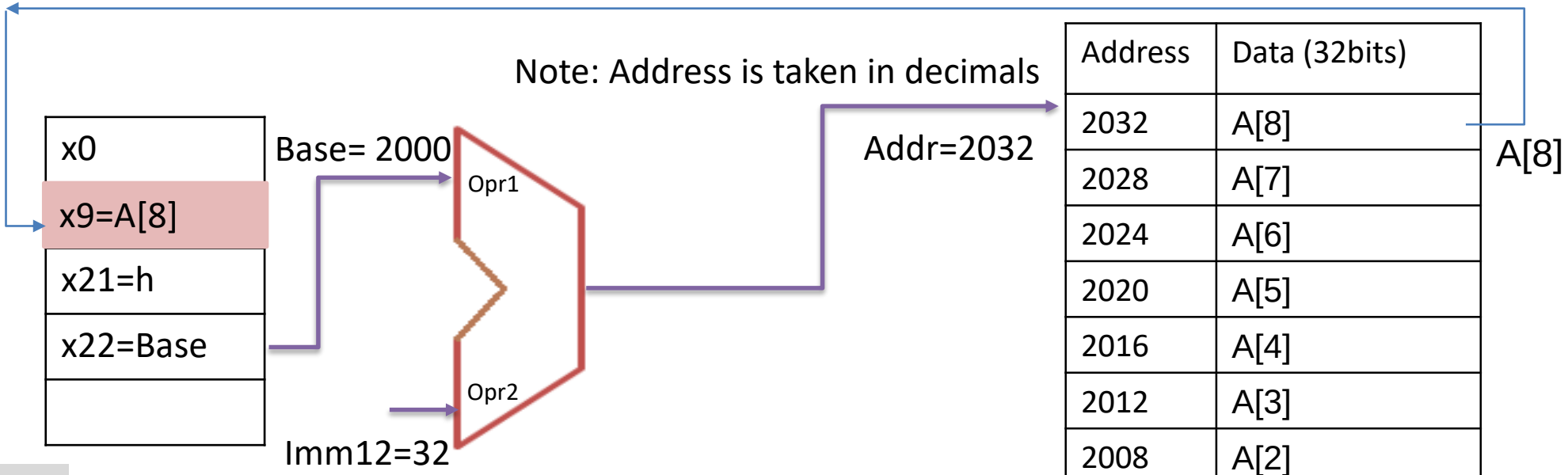
Address Hexadei	Address Decimal	Data (32bits)
0X2020	2032	A[8]
0X201B	2028	A[7]
0X2018	2024	A[6]
0X2014	2020	A[5]
0X2010	2016	A[4]
0X200B	2012	A[3]
0X2008	2008	A[2]
0X2004	2004	A[1]
0X2000	2000	A[0]

Operands of the Computer Hardware

2. Memory Operands

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];



Address	Data (32bits)
2032	A[8]
2028	A[7]
2024	A[6]
2020	A[5]
2016	A[4]
2012	A[3]
2008	A[2]
2004	A[1]
2000	A[0]

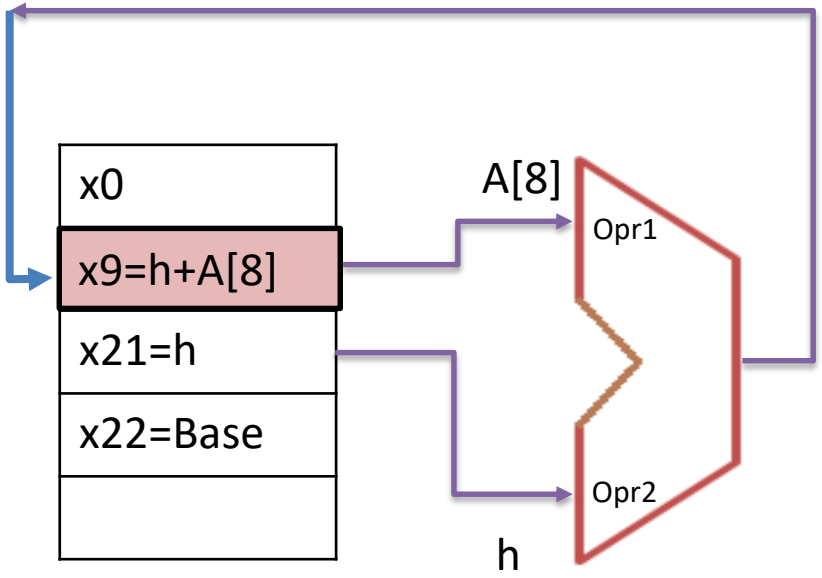
lw x9, 32(x22)

As element Number is know, offset can be calculated manually (8 x 4 =32) and taken as immediate offset in lw Instruction.

2. Memory Operands

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];




```
add     x9, x21,x9
```

Base= 2000

x22

Address	Data (32bits)
2032	A[12]
2028	A[11]
2024	A[10]
2020	A[9]
2032	A[8]
2028	A[7]
2024	A[6]
2020	A[5]
2016	A[4]
2012	A[3]
2008	A[2]
2004	A[1]
2000	A[0]

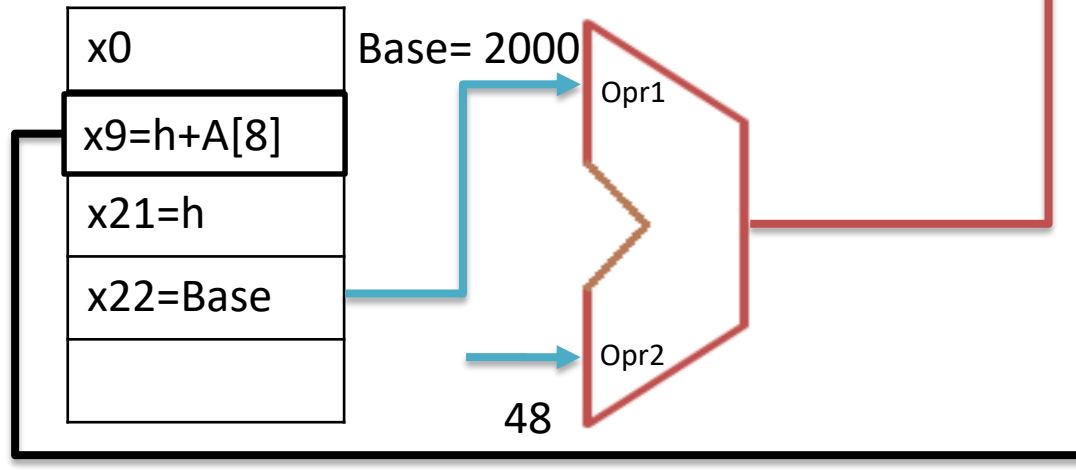
Instructions – Language of Computer

Operands of the Computer Hardware

2. Memory Operands

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];



Address	Data (32bits)
2048	A[12]=Content of x9
2044	A[11]
2040	A[10]
2036	A[9]
2032	A[8]
2028	A[7]
2024	A[6]
2020	A[5]
2016	A[4]
2012	A[3]
2008	A[2]
2004	A[1]
2000	A[0]

```
lw      x9, 32(x22)
add     x9, x21,x9
sw      x9, 48(x22)
```

As element Number is know, offset can be calculated manually (12 x 4 =48) and taken as immediate offset in lw Instruction.

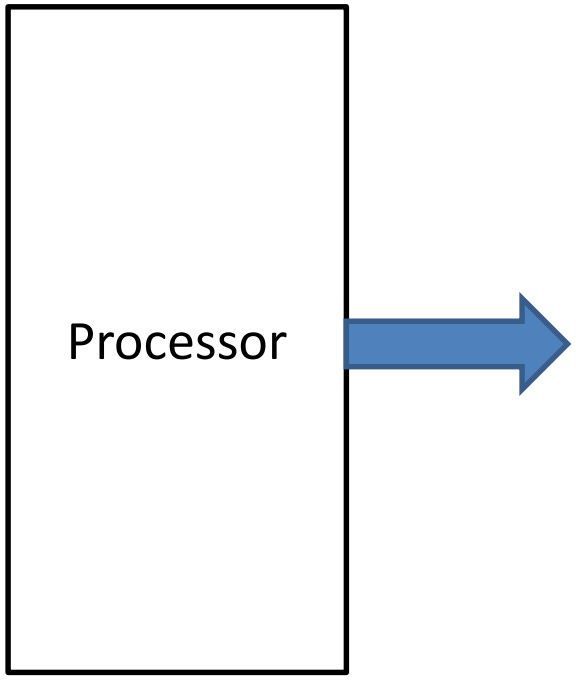
Base= 2000

x22

2. Memory Operands

How RISC V access double word from the Memory?

- RISC-V actually uses byte addressing, with **each double-word representing 8 bytes.**
- **Therefore, the addressing will have an offset of 8 for accessing a double word**



Address	Data
	1010 1010…… 10101111
……	
…….	
……	1111 1111…… 10101011
24	1000 0001…… 11111111
16	0001 0111…… 11111111
8	1011 0010…… 10101010
0	1011 0101……..0000 1111 (64 bits =8 bytes)

Instructions – Language of Computer

Operands of the Computer Hardware

2. Memory Operands

How RISC V access array of 'n' double-word from the Memory?

Compiling an Assignment When an Operand Is in Memory

Example

Let's assume that A is an array of 100 double-words and that the compiler has associated the variables g and **h** with the registers x20 and **x21**. 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[4];

x20	g
x21	h
x22	Base Address of array A

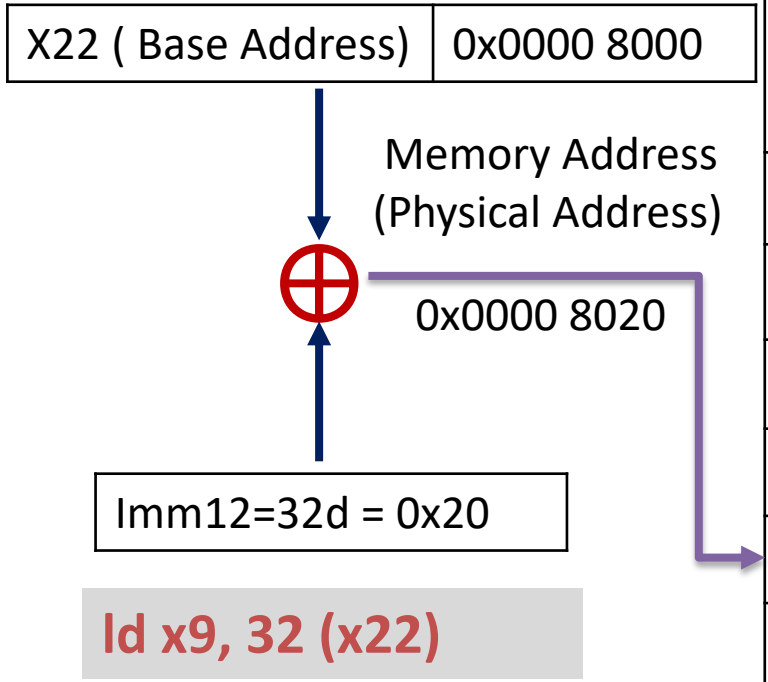
Address (decimal)	Address (Hexadecimal)	Data (64 bits)
2048	2030	A[6]
2040	2028	A[5]
2032	2020	A[4]
2024	2018	A[3]
2016	2010	A[2]
2008	2008	A[1]
2000	2000	A[0]

Instructions – Language of Computer

Operands of the Computer Hardware

2. Memory Operands

Load Instruction (ld) : Syntax - ld rd,imm12(rs1)



ld x9, 32 (x22)

Bits → Address	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	E
	Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0	
0x2000	8F	34	3D	DD	12	78	95	9F	0
0x2008	67	DF	ED	11	8F	34	3D	DD	1
0x2010	77	66	FF	3D	11	22	44	DE	2
0x2018	11	22	DD	EE	05	87	DD	AA	3
0x2020	11	22	DD	EE	FF	3D	11	22	4
0x2028	DD	EE	05	87	8F	34	11	77	5

x9	11	22	DD	EE	FF	3D	11	22

Offset = 4 x 8 = 32

Operands of the Computer Hardware

2. Memory Operands

How RISC V access array of ‘n’ double-word from the Memory?

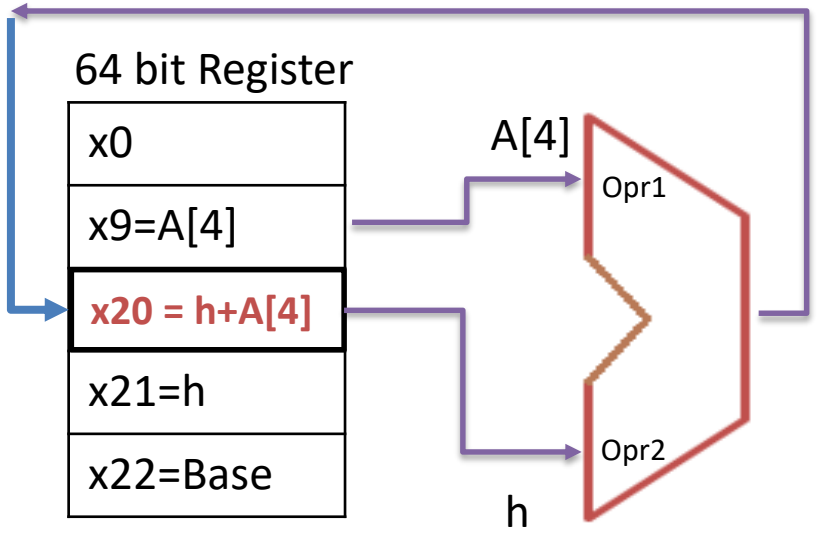
Compiling an Assignment When an Operand Is in Memory

Example

Compile this C assignment statement: **g = h + A[4];**

ld x9, 32(x22)
add x20, x21, x9

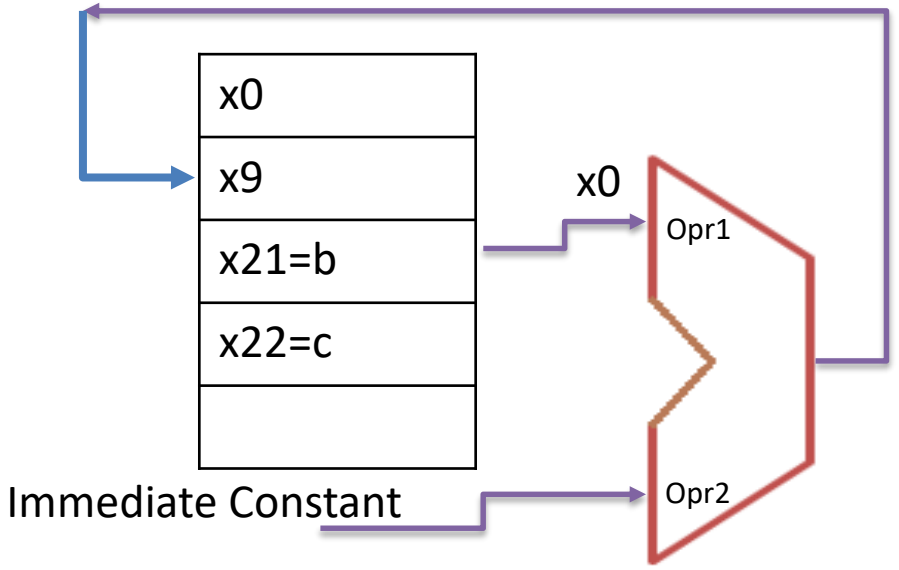
x20	g
x21	h
x22	Base Address of array A



Saved Registers	x8, x9, x18-x27
Temporary registers	x5-x7, x21-x31

2. Constants – Immediate Operand

- **Constant data specified in an instruction**
- **Make the common case fast**
  - ✓ **Small constants are common**
  - ✓ Immediate **operand avoids use of Register for a variable / Memory to hold the constants**



C Code

```
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

RISC-V assembly code

```
# x0 = a, x9 = b
addi x8, zero, -372
addi x9, x8, 6
```

Note: Any immediate that needs **more than 12 bits** can not use this method.

Instructions – Language of Computer

Operands of the Computer Hardware

Example : Add two numbers present in the memory and store the result Next

```
.data
Array1: .word 0x80000000,0x8000000f,00,0xFFFFFFFF
.text
la x18, Array1 # loads the location address of the specified SYMBOL
lw x8,0(x18)
lw x9,4(x18)
add x19,x8,x9
sw x6,8(x18)
```

Address Hexadeicimal	Data (32bits)
0X1000 000B	Array1[3]
0X1000 0008	Array1[2]
0X1000 0004	Array1[1]
0X1000 0000	Array1[0]

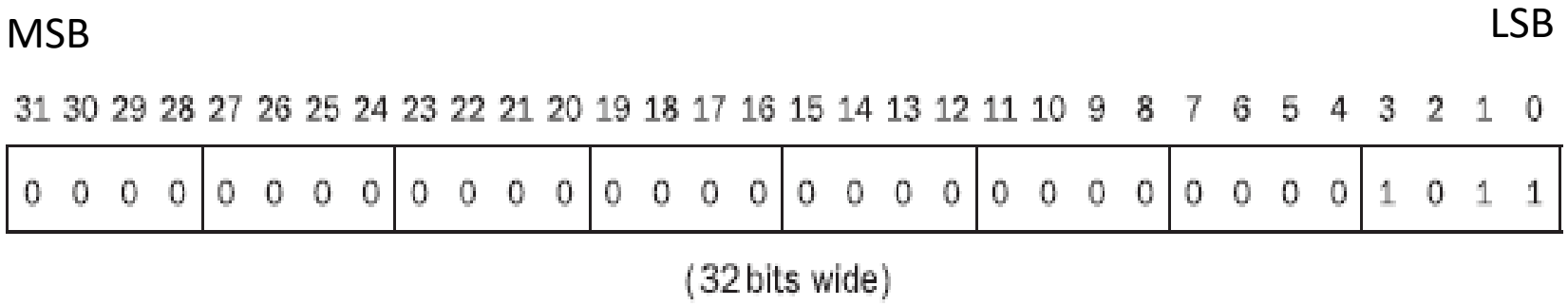
Memory Mapping

0x0000 0000 0x0FFF FFFF	Code Memory
0x1000 0000 0x1FFF FFFF	Data Memory

Instructions – Language of Computer

Signed and Unsigned Numbers

Let us consider a 32 bit (word) number representation



Why didn't computers use decimal?

How many bit pattern it can detect ?
What number they represent in decimal ?

$00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$
$00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$
$00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$
...
$11111111\ 11111111\ 11111111\ 11111101_{two} = 4,294,967,293_{ten}$
$11111111\ 11111111\ 11111111\ 11111110_{two} = 4,294,967,294_{ten}$
$11111111\ 11111111\ 11111111\ 11111111_{two} = 4,294,967,295_{ten}$

Instructions – Language of Computer

Signed and Unsigned Numbers

Let us understand Unsigned Number Reprentation using 32 bit binary number

Binary Number	Hexa-Decimal	
31 0		
0000 0000 0000 0000 0000 0000 0000 0000	(0x 0000 0000)	0_{ten}
0000 0000 0000 0000 0000 0000 0000 0001	(0x 0000 0001)	1_{ten}
0000 0000 0000 0000 0000 0000 0000 0010	(0x 0000 0002)	2_{ten}
…		
…		
0111 1111 1111 1111 1111 1111 1111 1111	(0x 7FFF FFFF)	
1000 0000 0000 0000 0000 0000 0000 0010	(0x 8000 0000)	
1000 0000 0000 0000 0000 0000 0000 0001	(0x 8000 0001)	
…		
….		
1111 1111 1111 1111 1111 1111 1111 1110	(0x FFFF FFFE)	$4,294,967,293_{ten}$
1111 1111 1111 1111 1111 1111 1111 1111	(0x FFFF FFFF)	$4,294,967,294_{ten}$ $4,294,967,295_{ten}$

Instructions – Language of Computer

Signed and Unsigned Numbers

How Signed Number Reprentation is different from Unsigned Number Representation?

31	30	0
Sign	Magnitude	

2's complement representation indicates negative numbers have a 1 in the MSB. Thus, hardware needs to test only MSB to decide whether the Number is positive or negative.

31	30							0	
0	000	0000	0000	0000	0000	0000	0000	0000	(0x 0000 0000) 0
0	000	0000	0000	0000	0000	0000	0000	0001	(0x 0000 0001) 1
0	000	0000	0000	0000	0000	0000	0000	0010	(0x 0000 0002) 2
…									
…									
0	111	1111	1111	1111	1111	1111	1111	1111	(0x 7FFF FFFF) ..
1	000	0000	0000	0000	0000	0000	0000	0010	(0x 8000 0000) ..
1	000	0000	0000	0000	0000	0000	0000	0001	(0x 8000 0001)
…									
….									
1	111	1111	1111	1111	1111	1111	1111	1110	(0x FFFF FFFE)=-2
1	111	1111	1111	1111	1111	1111	1111	1111	(0x FFFF FFFF) =-1

Positive Number Range

Negative Number Range & They are in 2's complement form

Instructions – Language of Computer

Signed and Unsigned Numbers

How Signed Number Reprentation is different from Unsigned Number Representation?

31	30	0
Sign	Magnitude	

2's complement representation indicates negative numbers have a 1 in the MSB. Thus, hardware needs to test only MSB to decide whether the Number is positive or negative.

2s-Complement Signed Integers

Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

Range: -2^{n-1} to $+2^{n-1} - 1$

Example

$1111\ 1111 \ldots 1111\ 1100_2$

$= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$

$= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

Using 64 bits: −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Instructions – Language of Computer

Signed and Unsigned Numbers

How Signed Number Reprentation is different from Unsigned Number Representation?

31	30	0
Sign	Magnitude	

2s-Complement Signed Integers

Some specific numbers

0: 0000 0000 … 0000

–1: 1111 1111 … 1111

Most-negative: 1000 0000 … 0000

Most-positive: 0111 1111 … 1111

Instructions – Language of Computer

Signed and Unsigned Numbers

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) $18,446,744,073,709,551,608_{ten}$

What is the decimal value if it is instead a 64-bit unsigned number?

Instructions – Language of Computer

Load and Store Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory

Instructions – Language of Computer

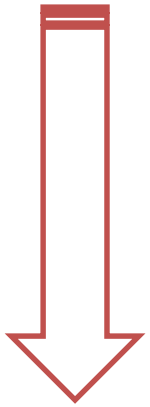
Signed and Unsigned Numbers

Sign extension

- **Representing a number using more bits**
  Ex: Reading a variable of 16 bit (half word) size and loading into a 32 bit register
- The variable may be a Unsigned / signed number

Memory

15 12	11 8	7 4	3 0
0111	0101	1011	0001



Loading 16 bit data from memory into a 32 bit register i.e., **Representing a number using more bits**

- ✓ **Where is the 16 bit data from memory is loaded in destination register ???**
- ✓ **What will happen to remaining 16 bits of the destination register i.e. reg[31:16]???**
- ✓ **reg[31:16] will change the content and It differs based on whether variable is Signed and Unsigned ?**

	31 28				15 12	11 8	7 4	3 0
Register	1010	1111	1011	1000	0111	0101	1011	0001

Data in Register in Hexa

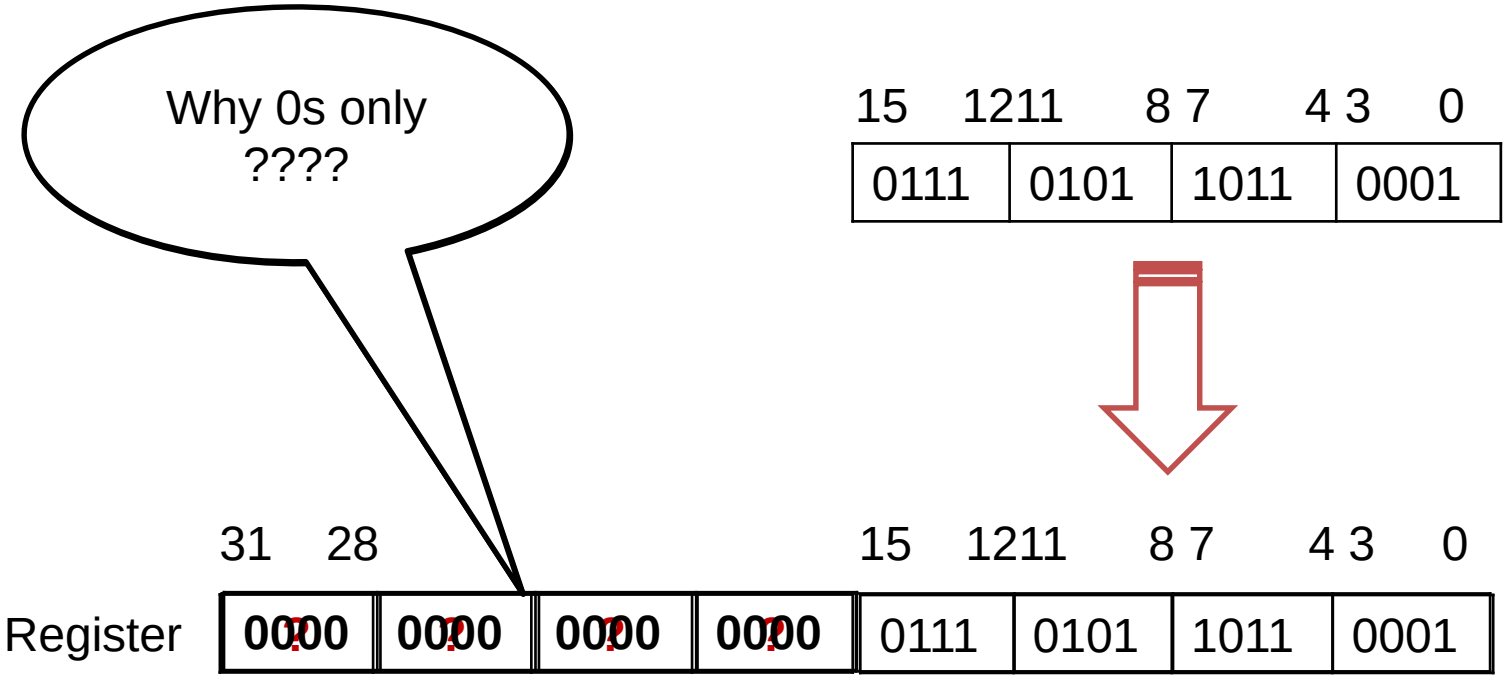
0xAFB875B1

Signed and Unsigned Numbers

Sign extension

How a Signed and Unsigned Load into a Register differs ?

Let us consider loading 32 bit register with a **16 bit unsigned value** ?

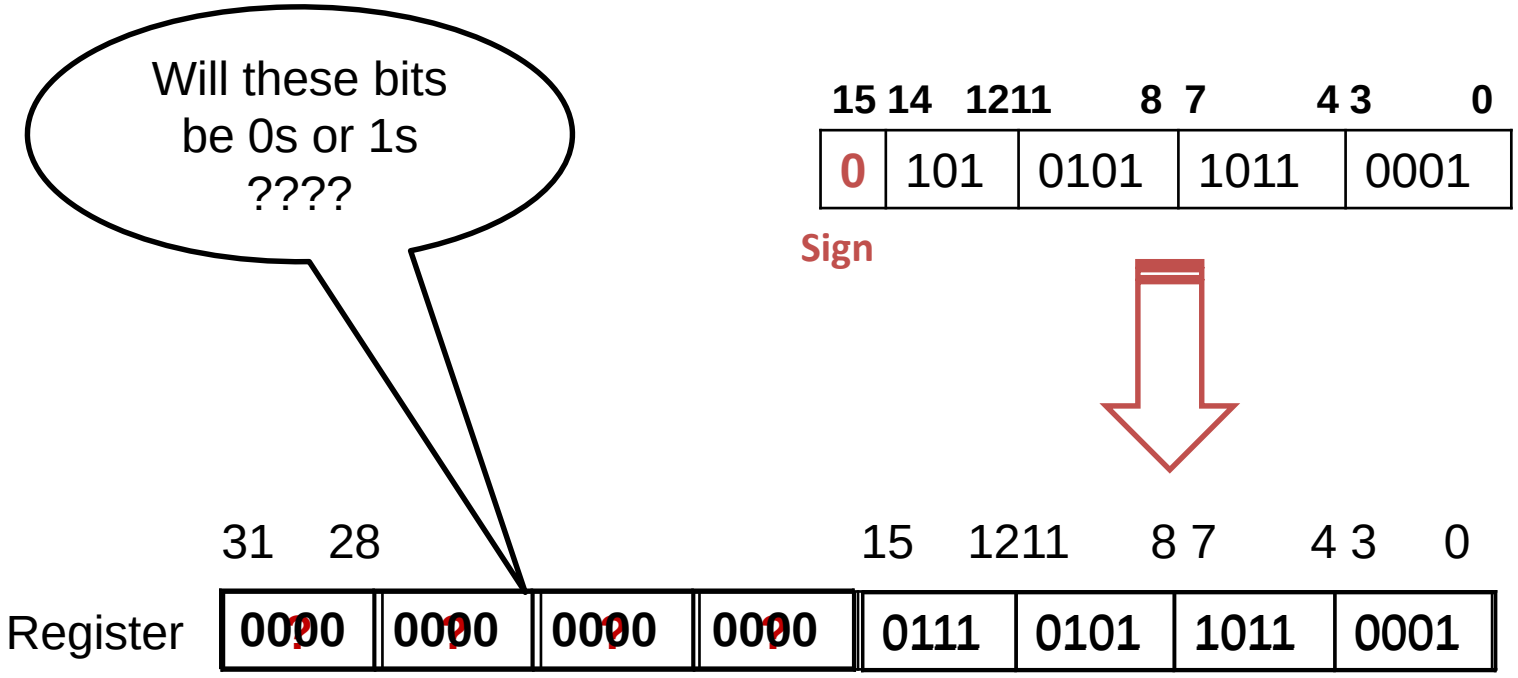


Signed and Unsigned Numbers

Sign extension

How a Signed and Unsigned Load into a Register differs ?

Let us consider loading 32 bit register with a **16 bit Signed value** ?

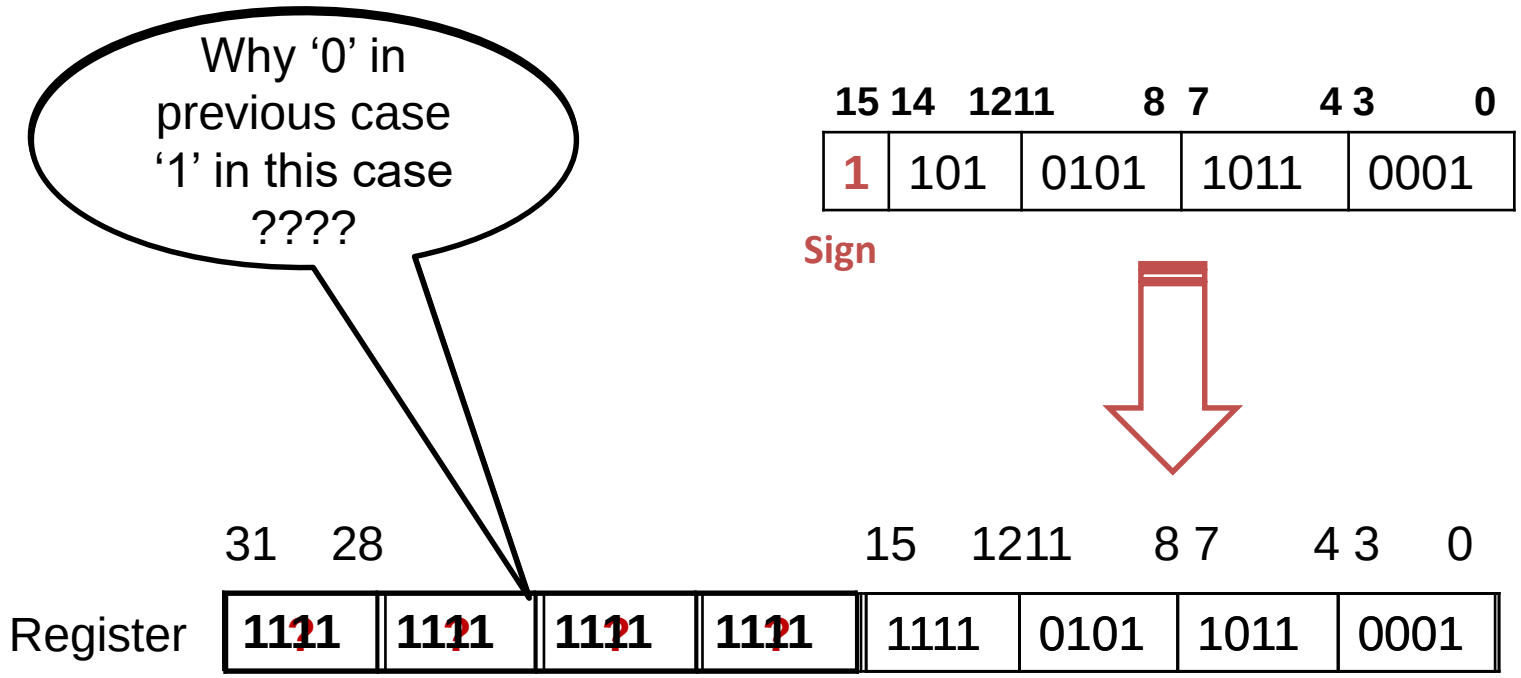


Signed and Unsigned Numbers

Sign extension

How a Signed and Unsigned Load into a Register differs ?

Let us consider loading 32 bit register with a **16 bit Signed value** ?



- When loading a 32-bit word into a 32-bit register, the point is moot (Matter of No importance); signed and unsigned loads are identical.
- When loading 16 bit half word into a 32 bit register, the signed and unsigned loads are not identical.

Signed and Unsigned Numbers

Sign extension

- RISC-V does offer two flavors of byte loads:
  1. ***load byte unsigned*** **(lbu)** treats the byte as an unsigned number and thus zero-extends to fill the leftmost bits of the register,
  2. ***load byte*** **(lb)** works with signed integers.

  Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, lbu is used practically exclusively for byte loads.
- The binary representation can be used as data and Address

  Does Signed Number Representation makes any sense when it is used to address and data ????

Instructions – Language of Computer

Signed and Unsigned Numbers

Working with two's complement numbers.

Negate a two's complement binary number ?

Ex: Negation of (-1) is (+1)

X= (-1) = 1111 1111 1111 1111 1111 1111 1111 1111 (0xFFFF FFFF)

(+1)= 0000 0000 0000 0000 0000 0000 0000 0001 (0x0000 0001)

How can you find it quickly ????

Solution :

a) X' + 1 = -X

b) This shortcut is based on the observation that the **sum of a number and its inverted representation must be (111 … 111)$_2$**, which represents −1. [X+X' =-1]

X= (-1) = 1111 1111 1111 1111 1111 1111 1111 1111 (0xFFFF FFFF)

X' = 0000 0000 0000 0000 0000 0000 0000 0000 (0x0000 0000)

(-1) = 1111 1111 1111 1111 1111 1111 1111 1111

Since *x* + *x'*= -1 , therefore *x* + *x'*+ 1= 0 or *x'*+ 1= -*x* .

Instructions – Language of Computer

Representing Instructions in the Computer

how do we represent instructions?

We know that Computer only understands 1s and 0s.

- A assembler string like “add x10,x11,x0” is meaningless to hardware as hardware can understand only machine code.
  - RISC-V seeks simplicity: since data is in words, make instructions of same 32 bit words. As **RISC-V seeks simplicity, so define six basic types of instruction formats:**
    - **R-format** for register-register arithmetic operations
    - **I-format** for register-immediate arithmetic operations and loads
    - **S-format** for stores
    - **B-format** for branches (minor variant of S-format)
    - **U-format** for 20-bit upper immediate instructions
    - **J-format** for jumps (minor variant of U-format)
- **All these Instruction formats use 32 bit Instruction word and 32 bit word is divided into “fields”**
- Each field tells processor something about instruction.

Machine Code/Language : Binary representation used for communication within a computer system.

Instruction format: A form of representation of an instruction **composed of fields of binary numbers.**

Instructions – Language of Computer

Representing Instructions in the Computer

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						

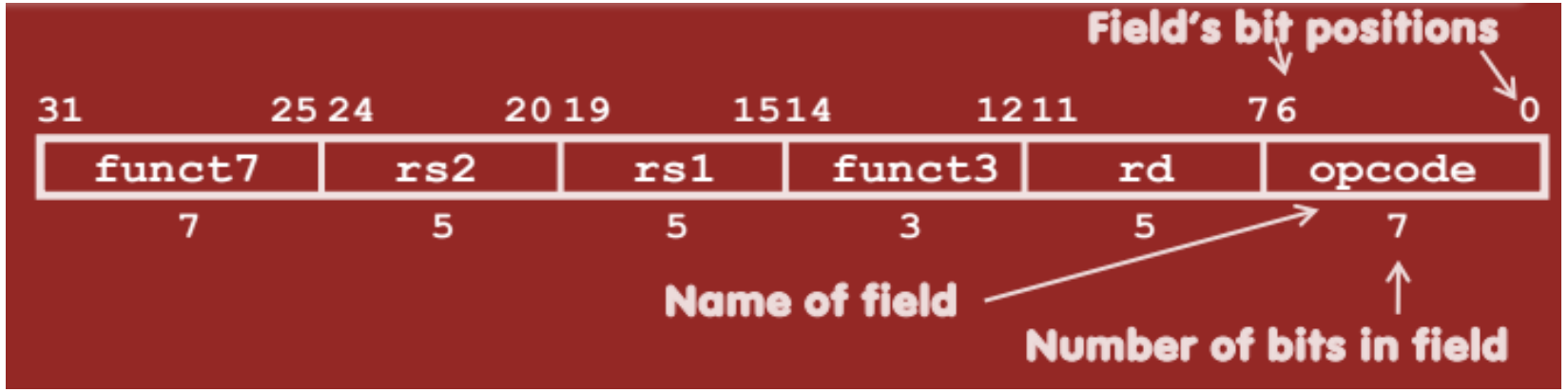
- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register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

Instructions – Language of Computer

Representing Instructions in the Computer

R-Format Instruction Layout:

Syntax: mnemonics rd,rs1,rs2



Opcode: Basic operation of the instruction like I,r,s,u and so on,

Note: This field is equal to **0110011_2 for all R-Format** register-register arithmetic instructions

funct7 + funct3: combined with opcode, these two fields describe what operation to perform

rs1 (Source Register #1): specifies register containing first operand

rs2 (Source Register #2): specifies register containing Second operand

rd(Destination Register):specifies register which will receive result of computation

Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0-x31)

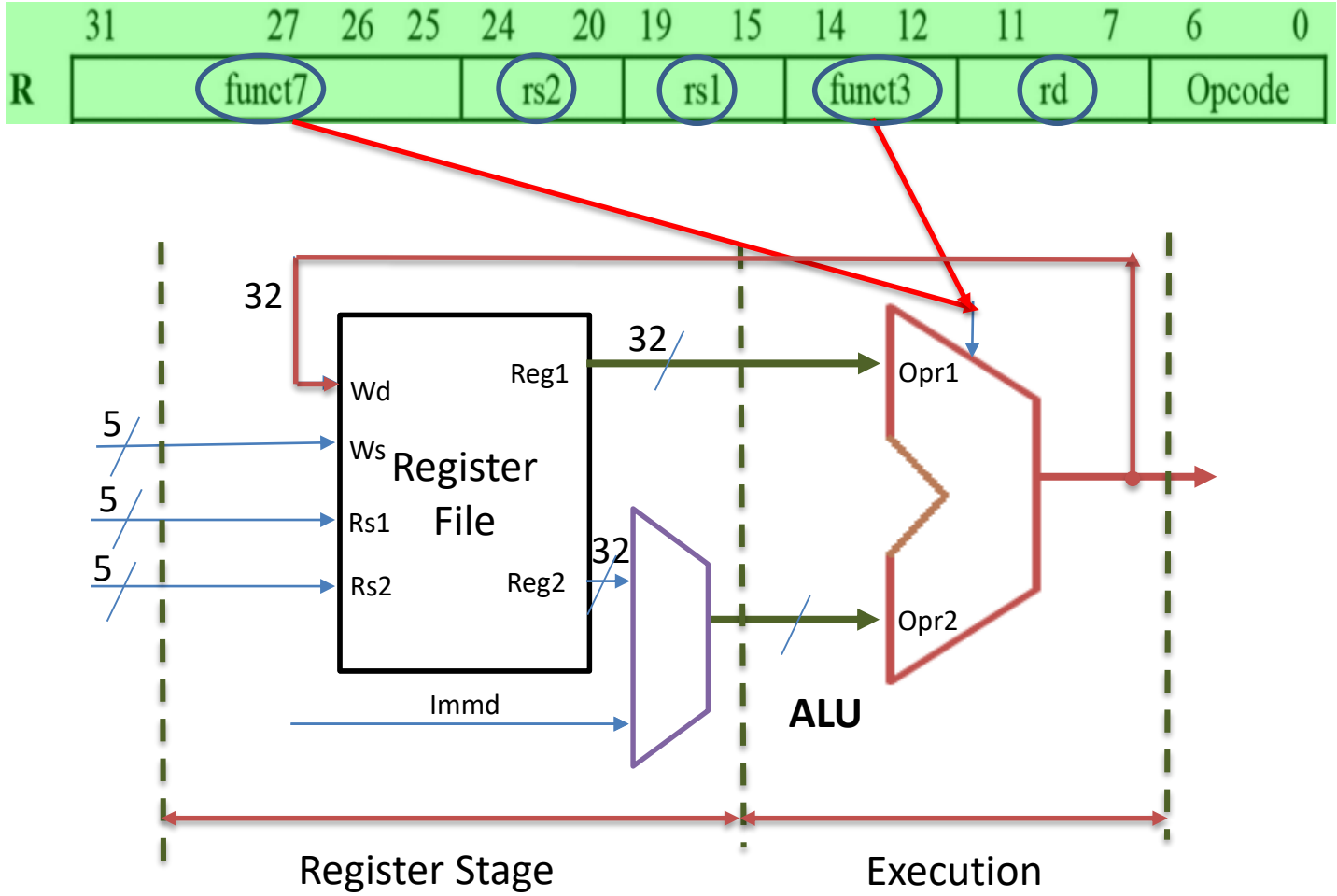
Question: why aren't opcode, funct7 and funct3 a single 17-bit field?

Register	b4 b3 b2 b1 b0
r0	00000
r1	00001
r2	00010
r30	11110
r31	11111

Instructions – Language of Computer

R-type Instruction Data Path

Anything that stores data or operates on data within a processor is called 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) x (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

Instructions – Language of Computer

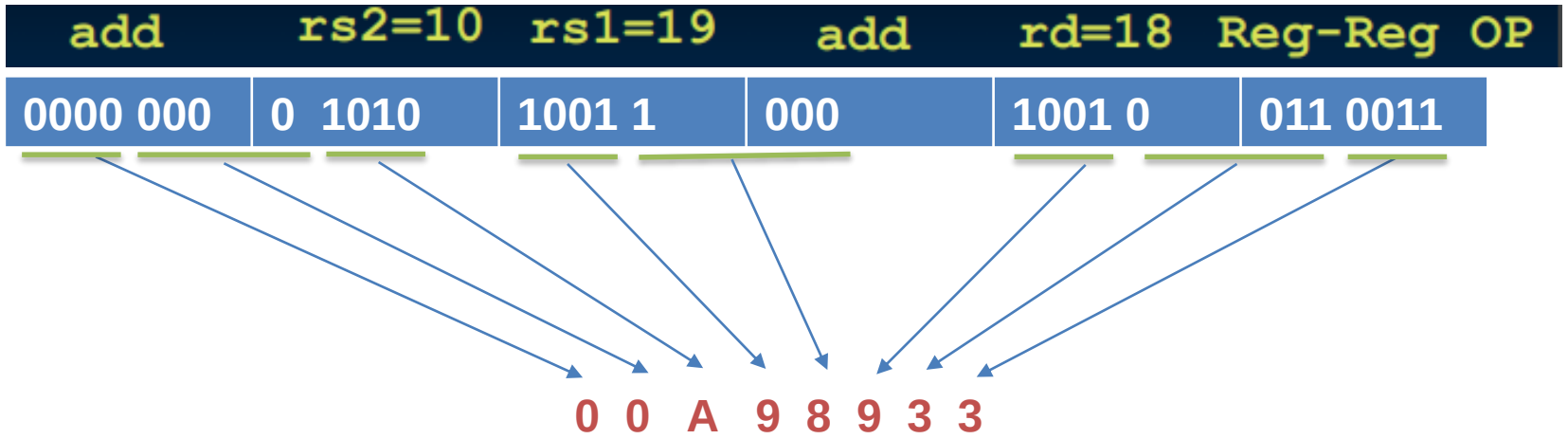
Representing Instructions in the Computer

How R-Format Instruction are Encoded ?

Syntax: mnemonics rd,rs1,rs2

add x18,x19,x10

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011



Hexadecimal Representation: 0x00A9 8933

Register	b4b3b2b1b0
r0	00000
r1	00001
r2	00010
	.....
r10	01010
	.....
r18	10010
r19	10011
	.....
r31	11111

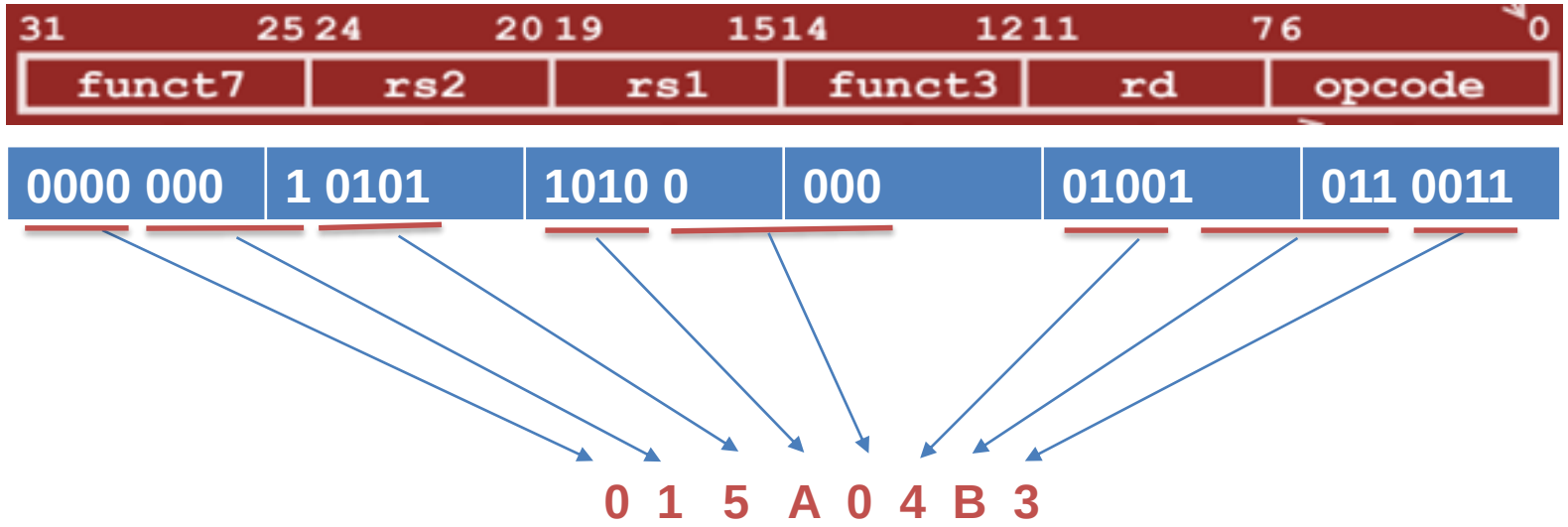
Instructions – Language of Computer

Representing Instructions in the Computer

How R-Format Instruction are Encoded ?

Syntax: mnemonics rd,rs1,rs2

add x9, x20, x21



Hexadecimal Representation: 0x015A 04B3

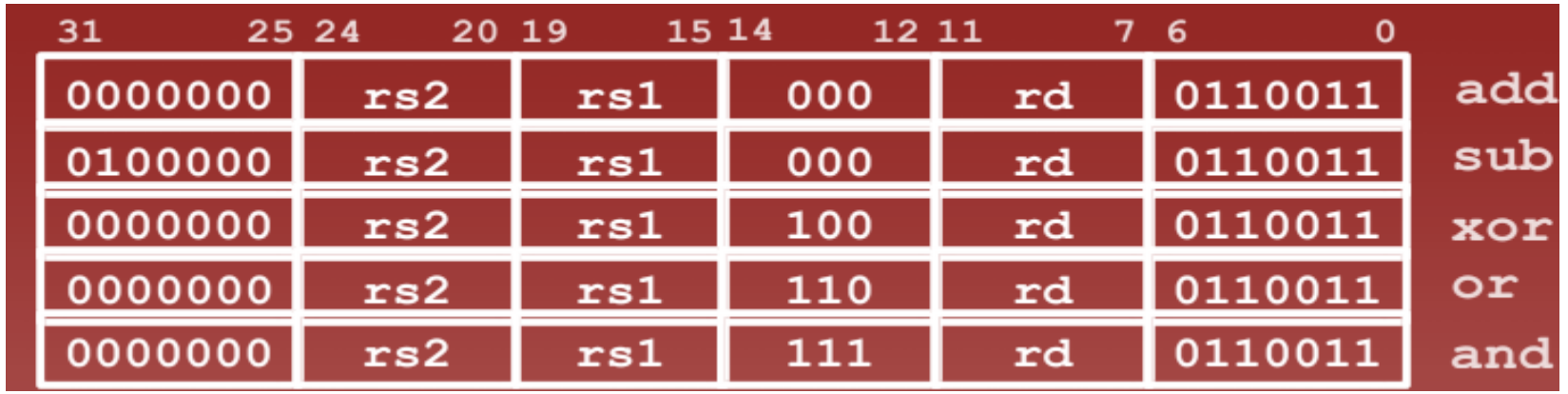
Register	b4b3b2b1b0
r0	00000
r1	00001
r2	00010
	.....
r10	01010
	.....
r18	10010
r19	10011
	.....
r31	11111

Instructions – Language of Computer

Representing Instructions in the Computer

How R-Format Instruction are Encoded ?

Syntax: mnemonics rd,rs1,rs2



31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

What is correct encoding of add x4, x3, x2 ?

1) 0x4021 8233
2) 0x0021 82b3
3) 0x4021 82b3
4) 0x0021 8233
5) 0x0021 8234

0000 000	0 0010	0001 1	000	0010 0	011 0011

Hexadecimal Representation: 0x0021 8233

Register	b4b3b2b1b0
r0	00000
r1	00001
r2	00010
	.....
r10	01010
	.....
r18	10010
r19	10011
	.....
r31	11111

Instructions – Language of Computer

Representing Instructions in the Computer

I-type Instruction Format

Syntax: mnemonics rd,rs1,imm

- In these Instructions data is part of Instruction i.e., Immediate data.
- What maximum size of immediate data can be used ?????
  - Can it be 32 bit size ?
  - Can it be 5 bit size ?
- Ideally, RISC-V would have **only one instruction format** (for simplicity): unfortunately, we need to compromise
- Define **new instruction format that is mostly consistent with R-format**
- **Immediate data can be as large as possible i.e., 32 bits but there is restriction as the instruction format is of maximum 32 bit size. It should be including fields Opcode, rdes, rsrc1, funct and field for immediate data also**
- Notice if instruction has immediate, then uses at most 2 registers (one source, one destination) Ex: **addi rd, rs1, imm**

- **5-bit field only represents numbers up to the value 31: immediate may be much larger than this.**

Compare:

- ✓ **add rd, rs1, rs2**
- ✓ **addi rd, rs1, imm**

Instructions – Language of Computer

Representing Instructions in the Computer

How I-Format Instruction are Encoded ? Syntax: mnemonics rd,rs1,imm

- As Opcode field defines Instruction type and there are limited instructions which support immediate operand func3 field is sufficient.
- Therefore funct7 and rs2 fields of r-type can be used immediate field in i-type at maximum. Therefore, immediate data can be of maximum 12 bit size.
- The 12-bit immediate is interpreted as a two's complement value, so it can represent integers from -2^{11} to $2^{11}-1$.

0	000	0000	0000
0	000	0000	0001
0	111	1111	1111
1	000	0000	0000
1	000	0000	0001
1	111	1111	1111

+ve

-ve

Instructions – Language of Computer

Representing Instructions in the Computer

How I-Format Instruction are Encoded ?

Syntax: mnemonics rdes,rsrc1,imm

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

All these are different Instructions supporting I Format

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

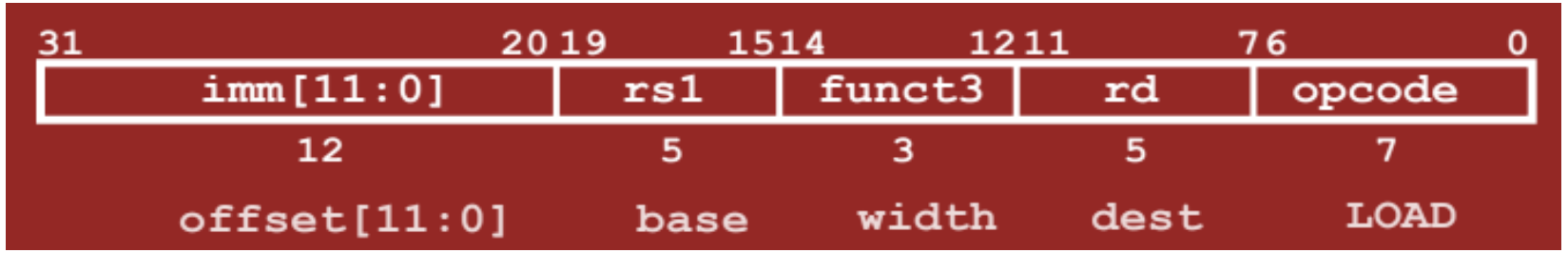
“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

Instructions – Language of Computer

Representing Instructions in the Computer

How I-Format Instruction used for load

Syntax: mnemonics rd, imm(rs1)



Syntax: mnemonics rd, imm (rs1)

lw

rs1 holds the base address of Memory location

Destination Register in which data from Memory address=base address + immediate offset

12 bit Immediate offset

Instructions – Language of Computer

Representing Instructions in the Computer

How I-Format Instruction used for load

Syntax: mnemonics rd, imm(rs1)

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

imm[11:0]	rs1	010	rd	0000011	lw
0000 0010 0000	1011 0	010	0100 1	000 0011	

Hexadecimal Representation: 0x020B2483

lw x9, 32(x22) - Load word

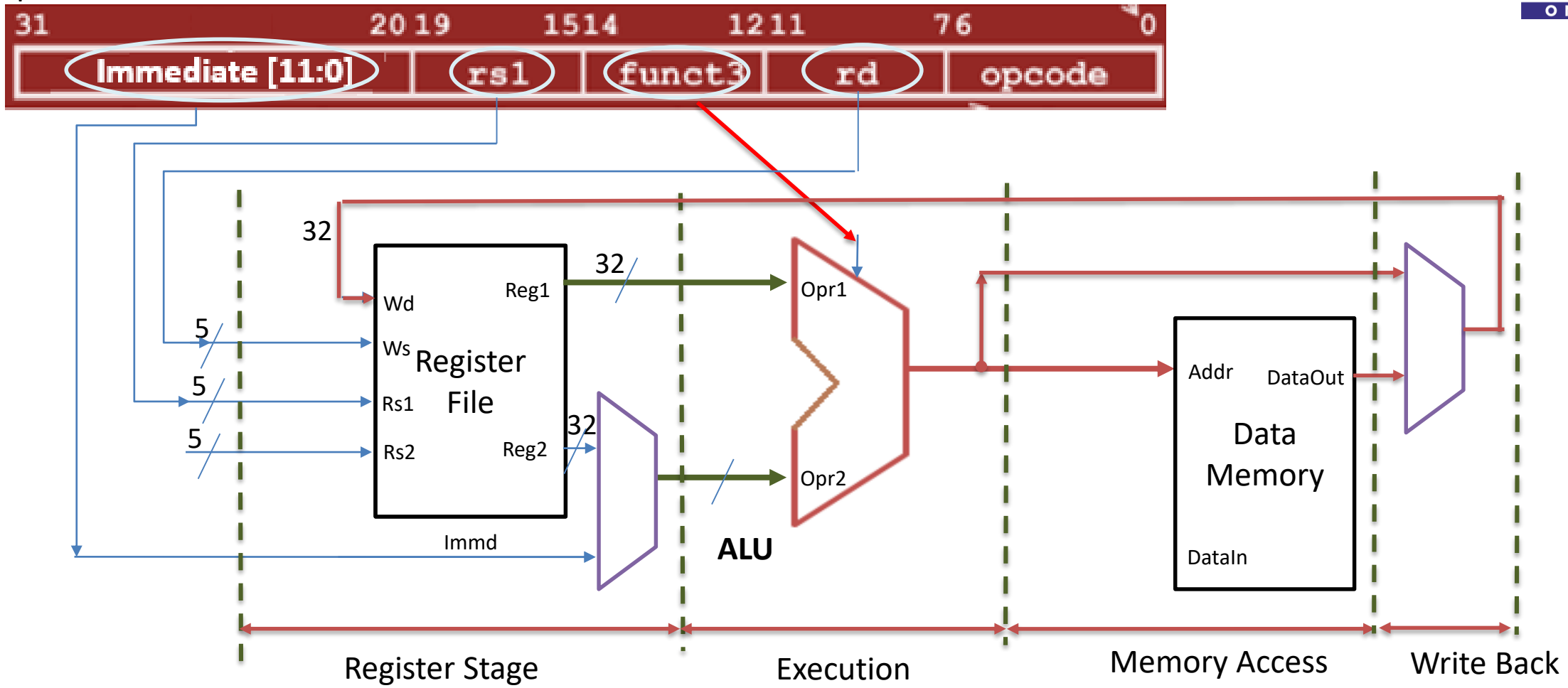
Here,

- 22 (for x22) is placed in the **rs1 field**,
- 32 is placed in the **immediate field**, and
- 9 (for x9) is placed in the **rd field.**

Instructions – Language of Computer

I-type Instruction Data Path

Anything that stores data or operates on data within a processor is called data path.



Instructions – Language of Computer

Representing Instructions in the Computer

Write the assembly code using 3 registers (x5, x6, and x7) to calculate the a) 2+3	
Using addi and add	Using only addi
addi x6,x0,2	addi x6,x0,2
addi x7,x0,23	addi x7,x6,3
add x5,x6,x7	

Write the assembly code using 3 registers (x5, x6, and x7) to calculate the a) 2-3	
Using addi and add	Using only addi and sub
addi x6,x0,2	addi x6,x0,2
addi x7,x0,-3	addi x7,x0,3
add x5,x6,x7	sub x5, x6, x7

Instructions – Language of Computer

Representing Instructions in the Computer

Division by a power of 2, to calculate for the following a) 88/8 b) -88/8	
Using **srli**	Using **srli**
addi x6, x0, 88	addi x6, x0, -88
srli x5, x6, 3	srli x5, x6, 3
x5 t0 0x000000000000000B = 11 x6 t1 0x0000000000000058 = 88	x5 t0 0x1FFFFFFFFFFFFFF5 = 2305843009213693941 x6 t1 0xFFFFFFFFFFFFFFA8 -88
b) -88/8 : The obtained result in the **x5 register shown above is obviously wrong**. The problem is **created by the 0 bits fed into the MSB part of the number during the shift**. Indeed, the expected result is -11, which is a negative number that must have 1 as a most significant bit.	
Problem is easily solved by using the srai (shift right arithmetic immediate) instruction	
addi x6, x0, -88	
srai x5, x6, 3	
x5 t0 0xFFFFFFFFFFFFFFF5 = -11 x6 t1 0xFFFFFFFFFFFFFFA8 = -88	

Instructions – Language of Computer

Representing Instructions in the Computer

The following source code will, for example, move bits [7:4] of the value in x6 to the 4 least significant bits [3:0] of x5 nullifying all its other bits:	
Instuction	Comment
addi x6, x0, 0x123	x6=0x00000123=0000 0000 0000 0000 0000 0001 0010 0011b
slli x7, x6, 24	x7=0x24000000=0010 0100 0000 0000 0000 0000 0000 0000b
srli x5, x7, 28	x5=0x00000002=0000 0000 0000 0000 0000 0000 0000 0010b

Instructions – Language of Computer

Representing Instructions in the Computer

1	Calculate the value of the expression (888/8-123*4)*2 and store the result in x5.
2	Store the value of 0xffffffff00000000 in x5 using only addi and slli instructions.
3	Store the value of 0x0000123400000000 in x5 using only addi and slli instructions.

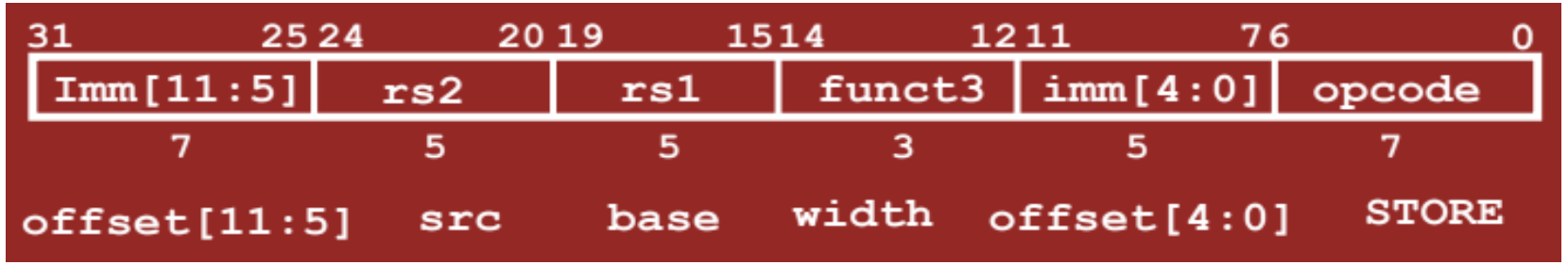
Instructions – Language of Computer

Representing Instructions in the Computer

S-type Instruction Format

Syntax: mnemonics rs2,imm(rs1)

- 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 don'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



Syntax: mnemonics rs2, imm (rs1)

sw

rs1 holds the base address of Memory location

12 bit Immediate offset

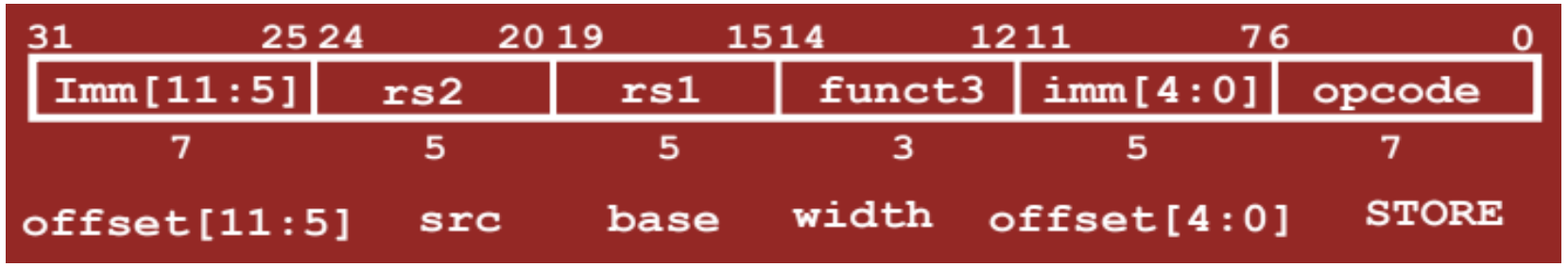
Source Register holding the data ti be stored in address=base address + immediate offset

Instructions – Language of Computer

Representing Instructions in the Computer

S-type Instruction Format

Syntax: mnemonics rs2,imm(rs1)



Imm[11:5]	rs2	rs1	010	Imm[4:0]	010 0011

sw x14, 8(x2)

000 0000	01110	00010	010	0 1000	010 0011

Imm_{12}: 0000 000 0 1000 (08)

Hexadecimal Representation: **0x00E12423**

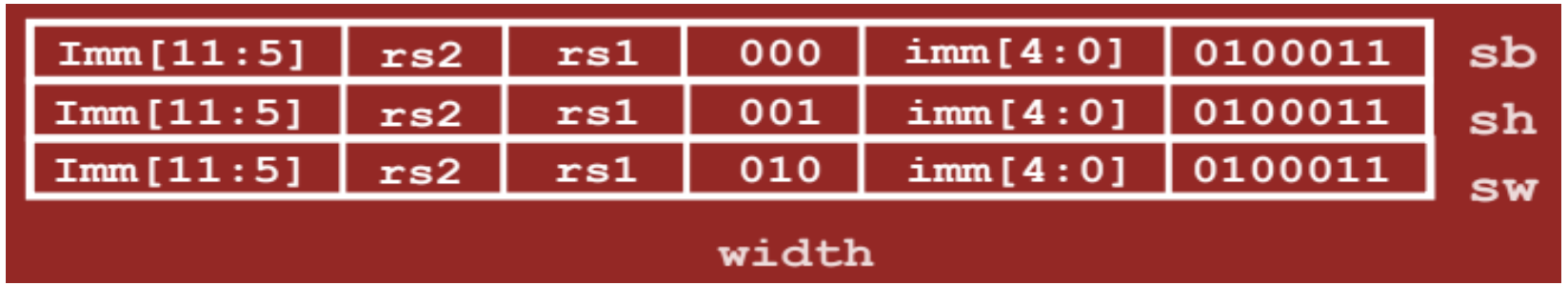
Instructions – Language of Computer

Representing Instructions in the Computer

S-type Instruction Format

Syntax: mnemonics rs2,imm(rs1)

- Store byte, half-word, word



Instructions – Language of Computer

Representing Instructions in the Computer

- Translating RISC-V Assembly Language into Machine Language
- If x10 has the base of the array A and x21 corresponds to h, the assignment statement

  A[30] = h + A[30] + 1; is compiled into

```
lw x9, 120(x10)          // Temporary reg x9 gets A[30]
add x9, x21, x9          // Temporary reg x9 gets h+A[30]
addi x9, x9, 1           // Temporary reg x9 gets h+A[30]+1
sw x9, 120(x10)          // Stores h+A[30]+1 back into A[30]
```

What is the RISC-V machine language code for these three instructions?

Since $120_{ten} = 0000011\ 11000_{two}$, the binary equivalent to the decimal form is:

immediate	rs1	funct3	rd	opcode
000011110000	01010	010	01001	0000011

funct7	rs2	rs1	funct3	rd	opcode
0000000	01001	10101	000	01001	0110011

immediate	rs1	funct3	rd	opcode
000000000001	01001	000	01001	0010011

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
0000011	01001	01010	010	11000	0100011

Instructions – Language of Computer

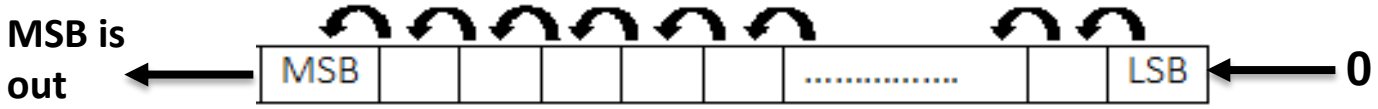
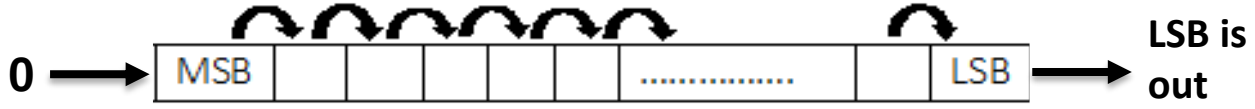
Logical Operations

Logical operations	C operators	Java operators	RISC-V Instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

Instructions – Language of Computer

Logical Operations

SHIFT Operations

Mnemonic	Descriptions
Logical Shift Left **sll , slli**	Content of Source Operand is shifted Left by Count times. Count is either an immediate value (#0-31 or Register Rs 
Logical Shift Right **srl , srli**	Content of Source Operand (Unsigned Number) is shifted Right by Count times. Count is either a immediate value (#0-31 or Register Rs. 
Arithmetic Right Shift **sra , srai**	Content of Source Operand (Signed Number) is shifted Right by Count times. It means in this status of MSB bit is shifted to right as well as copied into MSB itself. Count is either a immediate value (#0-31 or Register Rs. 

Instructions – Language of Computer

Logical Operations

Shift Left Logically Left

Syntax	slli rd, rs1,imm
Operation:	It logically shifts left the content of rs1 by immediate time (Note: Immediate value can be max of 32) and stores the shifted result in rd. Vacated bits on the right(lsb) is filled with zeros. i.e., rd=rs1<<imm
Example:	slli x11, x19, 4; x11=x19<<4
Before Execution	
x19=9_{ten} = **x11= xxxxxxxx** **Imm =4**	0000 0000 0000 0000 0000 0000 0000 1001
After Execution	
x19=9_{ten} =	0000 0000 0000 0000 0000 0000 0000 1001 b
x11= 144_{ten} =	**0000 0000 0000 0000 0000 0000 1001 0000 b [9x16=144]**
Imm =	**4**

Shifting left by *i* bits gives the identical result as multiplying by 2^i

sll x11,x19,x20

- In this shift count is basically the value held in x20.
- Functionally it does the same operation
- X11=x19<<x20 (count)

Instructions – Language of Computer

Representing Instructions in the Computer

How I-Format Instruction are Encoded ?

Syntax: mnemonics rd,rs1,imm

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

These shift instructions use the I-type format. Since it isn't useful to shift a 32-bit register by more than 31 bits, only the lower bits of the I-type format's 12-bit immediate are actually used. The remaining bits are repurposed as an additional opcode field, funct7.

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)

Instructions – Language of Computer

Representing Instructions in the Computer

I-type Instruction Format

Syntax: shift rd,rs1,imm

0000000	shamt	rs1	001	rd	0010011	slli
0000000	shamt	rs1	101	rd	0010011	srli
0100000	shamt	rs1	101	rd	0010011	srai

slli x11, x19, 4

19d = 13h

Funct7	shamt	rs1	funct3	rd	001 0011
0000 000	0 0100	1001 1	001	0101 1	0010011

Hexadecimal Representation: 0x00499593

Instructions – Language of Computer

Logical Operations

Shift Left Logically Left

Syntax	sll rd, rs1,rs2
Operation:	It logically shifts left the content of rs1 by rs2 times (Note: rs2 should be holding the count for shifting) and stores the shifted result in rd. Vacated bits on the right (lsb) is filled with zeros. i.e., rd=rs1<<rs2
Example:	sll x11, x19,x20; x11=x19<<x20
Before Execution	
x19 = 9_{ten} =	0000 0000 0000 0000 0000 0000 0000 1001
x11	**= xxxxxxxx**
x20	**=** 4_{ten}
After Execution	
x19=9_{ten} =	0000 0000 0000 0000 0000 0000 0000 1001 b
x11= 144_{ten} =	**0000 0000 0000 0000 0000 0000 1001 0000 b [9x16=144]**
x20 =	**4**$_{ten}$

Shifting left by *i* bits gives the identical result as multiplying by 2^i

Instructions – Language of Computer

Logical Operations

Shift Left Logically Left

Syntax	srli rd, rs1,imm
Operation:	It logically shifts right the content of rs1 by immediate time (Note: Immediate value can be max of 32) and stores the shifted result in rd. Vacated bits on the left(msb) is filled with zeros i.e., rd=rs1>>imm
Example:	srli x11, x19, 4; x11=x19>>4
Before Execution	
x19=9_{ten} = **x11= xxxxxxxx** **Imm =4**	0000 0000 0000 0000 0000 1000 0000 0000
After Execution	
x19=2048_{ten} = **x11=** 128_{ten} = **Imm** =	0000 0000 0000 0000 0000 1000 0000 0000 b **0000 0000 0000 0000 0000 0000 1000 0000 b [2048/16=128]** **4**

Shifting left by *i* bits gives the identical result as dividing by 2^i

srl x11,x19,x20

- In this shift count is basically the value held in x20.
- Functionally it does the same operation as that of srli but count is value stored in rs2
- X11=x19<<x20 (count)

Instructions – Language of Computer

Logical Operations

Shift Left Logically Left

Syntax	srai rd, rs1,imm
Operation:	It logically shifts right the content of rs1 by immediate time (Note: Immediate value can be max of 32) and stores the shifted result in rd. Vacated bits on the left is filled them with copies of the old sign bit.
Example:	srli x11, x19, 4; x11=x19>>4
Before Execution	
x19=9_{ten} = **x11= xxxxxxxx** **Imm =4**	1000 0000 0000 0000 0000 1000 0000 0000
After Execution	
x19=2048_{ten} = **x11=** 128_{ten} = **Imm =**	1000 0000 0000 0000 0000 1000 0000 0000 b **1111 1000 0000 0000 0000 0000 1000 0000 b** **4**

Shifting left by *i* bits gives the identical result as signed division by 2^i

Shift Left Logically Left

Examples:

1) Write the assembly code to **load the value of 17 in x6**, and then to **multiply it by 4** and store the result in x5. Use sll/slli
2) Write the assembly code to calculate a) (88 / 8) use srl/srli b) (-88/3) use srl/srli c) (-88/3) use sra/srai. Comment on result obtained for b) and c)
3) Shift instructions are often used for extracting subsequences of bits. Write the assembly code to move bits [7:4] of the value in x6 to the 4 least significant bits [3:0] of x5 nullifying all its other bits. Use addi and slli,srli
4) Store the value of 0x0000123400000000 in x5 using only addi and slli instructions.

Instructions – Language of Computer

Logical Operations

Logical AND,OR,XOR and NOT – R and I type

Syntax	and rd, rs1,rs2
Operation:	AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. In this case, the content of source register rs2 is ANDed with rs1 logically and the result is stored in destination register rd
Example:	and x9, x10,x11; x9=x10 & x11
Before Execution	
x10= =	00000000 00000000 00001101 11000000b
X11 =	00000000 00000000 00111100 00000000b
x9 =	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
After Execution	
x10= =	00000000 00000000 00001101 11000000b
X11 =	00000000 00000000 00111100 00000000b
x9 =	00000000 00000000 00001100 00000000b

Use: AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern.
This called as Masking

Instructions – Language of Computer

Logical Operations

Logical AND,OR,XOR and NOT

Syntax	or rd, rs1,rs2
Operation:	OR is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. In this case, the content of source register rs2 is ORed with rs1 logically and the result is stored in destination register rd
Example:	or x9, x10,x11; x9= x10 | x11
Before Execution	
x10= =	00000000 00000000 00001101 11000000b
X11 =	00000000 00000000 00111100 00000000b
x9 =	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
After Execution	
x10= =	00000000 00000000 0000**1101 11**000000b
X11 =	00000000 00000000 00**1111**00 00000000b
x9 =	00000000 00000000 00111101 11000000b

Use: Setting bits

Instructions – Language of Computer

Logical Operations

Logical AND,OR,XOR and NOT

Syntax	xor rd, rs1,rs2
Operation:	A logical bit-by bit operation with two operands that calculates the exclusive OR of the two operands. That is, it calculates a 1 only if the values are different in the two operands. In this case, the content of rs2 is XORed with rs1 logically and the result is stored in destination register rd
Example:	xor x9, x10,x11; x9= x10 ^ x11
Before Execution	
x10= =	00000000 00000000 00001101 11000000b
X11 =	00000000 00000000 00111100 00000000b
x9 =	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
After Execution	
x10= =	00000000 00000000 00001101 11000000b
X11 =	00000000 00000000 00111100 00000000b
x9 =	00000000 00000000 00110001 11000000b

Logical AND,OR,XOR and NOT

RISC-V also provides the instructions *and immediate* (andi), *or immediate* (ori), and *exclusive or immediate* (xori).

- AND x9, x10,imm_{12}
- OR x9, x10, imm_{12}
- XOR x9, x10, imm_{12}

Instructions – Language of Computer

Examples

Shift Left Logically Left

Examples:

1) Extract bits [7:4] of the value in x6 by directly masking out all other bits using the andi (and immediate) instruction and then move the bits to the LSB part.
2) Write a assembly code to combine the middle hexadecimal digit of the value 0x123 with the 1st and the 3rd hexadecimal digits of the value 0x456 and store in result 0x423 in x5.
3) Rotate right by 4 bits the value of 0x0000000000000123. The expected result is 0x3000000000000012, i.e. all hexadecimal digits move right by one position while the rightmost one moves to the front.

Instructions – Language of Computer

Examples

For the following C statement, write the corresponding RISC-V assembly code. Assume that the C variables f, g, and h, have already been placed in registers x5, x6, and x7 respectively. Use a minimal number of RISC-V assembly instructions.
f = g + (h − 5);

```
addi x5, x7,-5
add x5, x5, x6
```

Variables	Registers
f	x5
g	x6
h	x7

Instructions – Language of Computer

Examples

For the following C statement, write the corresponding RISC-V assembly code. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively.

B[8] = A[i–j];

```
sub x30, x28, x29 // compute i-j  (x28 – x29)
slli x30, x30, 2      // 32 bit data at [i-j]th element of Array A is at offset 4x[i-j]
add x3, x30, x10  // physical address of A[i-j] = Base Address + 4 x [i-j]
lw x30, 0(x3)         // load A[i-j]
sw x30, 32(x11)  // store in the content of x30 into B[8] =B[BA + 4x8] = B[x11 + 32]
```

Memory	0x0000	0x0004	0x0008	……	(i-j)x4	[(i-j)+1]x4	[(i-j)+2]
Element	0	1	2	……	(i-j)	[(i-j)+1]	[(i-j)+2]
A[]	A[0]	A[1]	A[2]	……	A[i-j]	A[(i-j)+1]	A[(i-j)+2]

Variables	Registers
f	x5
g	x6
h	x7
i	x28
j	x29
Base Address of A	x10
Base Address of B	x11

Instructions – Language of Computer

Examples

Translate the following C code to RISC-V. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively. Assume that the elements of the arrays A and B are 4-byte words:
B[8] = A[i] + A[j];

```
slli x28, x28, 2        // x28 = i*4 ( Address of ith element)
add x10,x10,x28         // x10 = &A[i] – Physical Address
lw x28, 0(x10)          // x28 = A[i]
slli x29, x29, 2        // x29 = j*4
add x12,x11,x29         // x12 = &B[j] – Physical Address
lw x29,0(x12)           // x29 = B[j]
add x29, x28, x29       // x29 = A[i]+B[j]
sw x29, 32(x11)         // B[8] = B[i]+B[j]
```

Variables	Registers
f	x5
g	x6
h	x7
i	x28
j	x29
Base Address of A	x10
Base Address of B	x11

Instructions – Language of Computer

Examples

Translate the following RISC-V code to C. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively.

```
addi x30, x10, 8
addi x31, x10, 0
sw x31, 0(x30)
lw x30, 0(x30)
add x5, x30, x31
```

```
f = 2*(& A)
addi x30, x10, 8          // x30 = & A[1] ; Address of A[1]
addi x31, x10, 0          // x31 = &A ; Base Address of A
sw x31, 0(x30)            // A[1] = &A ; A[1] = Base address of A
lw x30, 0(x30)            // x30 = A[1] = & A =Base address of A
add x5, x30, x31          // f = &A + &A = 2*(&A)
```

Variables	Registers
f	x5
g	x6
h	x7
i	x28
j	x29
Base Address of A	x10
Base Address of B	x11

Instructions – Language of Computer

Examples

Assume that registers x5 and x6 hold the values x80000000 and 0xD0000000, respectively.

a) What is the value of x30 for the following assembly code?

```
add x30, x5, x6
```

b) Is the result in x30 the desired result, or has there been overflow?

c) For the contents of registers x5 and x6 as specified above, what is the value of x30 for the following assembly code?

```
sub x30, x5, x6
```

d) Is the result in x30 the desired result, or has there been overflow?

e) For the contents of registers x5 and x6 as specified above, what is the value of x30 for the following assembly code?

```
add x30, x5, x6
add x30, x30, x5
```

f) Is the result in x30 the desired result, or has there been overflow?

Variables	Registers
f	x5
g	x6
h	x7
i	x28
j	x29
Base Address of A	x10
Base Address of B	x11

a) 0x50000000
b) overflow
c) 0xB0000000
d) no overflow
e) 0xD0000000
f) overflow

Instructions – Language of Computer

Instructions for Making Decisions

RISC-V assembly language includes two decision-making instructions, similar to an ***if*** **statement with a *go-to*.**

PC-Relative Addressing: Use the **`immediate`** field as a 2's complement offset (signed offset) to PC Branches generally change the PC by a small amount Can specify $\pm 2^{11}$'unit' addresses from the PC

Why not use byte as a unit of offset from PC?
Instructions are of 32-bits (4-bytes) size and a offset of byte means branch into middle of instruction. Branching into middle of the instruction does not make any sense.

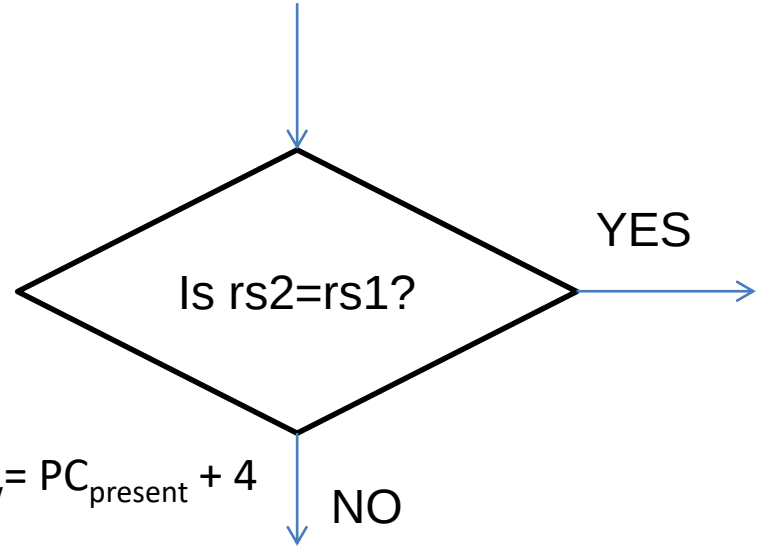
Instructions – Language of Computer

Instructions for Making Decisions

RISC-V assembly language includes two decision-making instructions, similar to an ***if* statement with a *go-to*.**

Conditional branches : An instruction that tests a value and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the test.

beq rs1, rs2, L1



PC= Address of Instruction where branching need to be done. The Label L1 is replaced by the address by assembler.

$PC_{new} = PC_{present} + Signed\ offset$

where Signed offset = Immediate($\pm 2^{11}$) * 4

immediate is number of instructions to jump either forward (+) or backwards (–)

$PC_{new} = PC_{present} + 4$

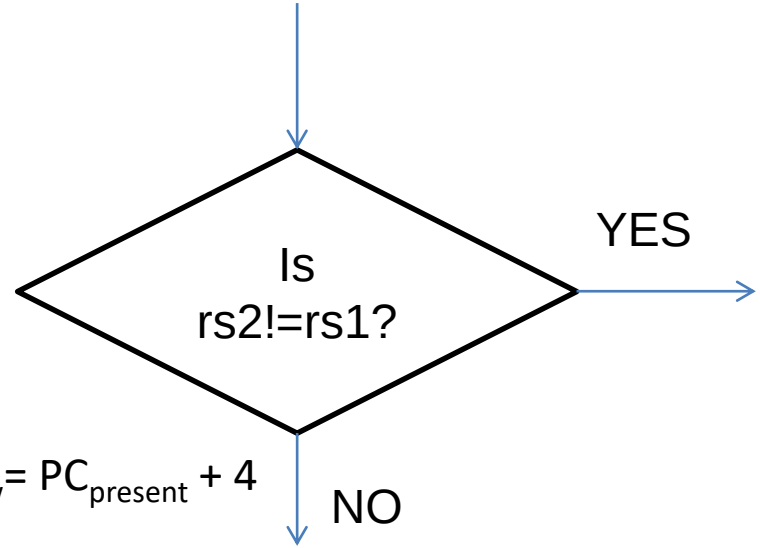
PC – NO change As a result, No change in Sequence of execution i.e., executes next Instruction after branch

Instructions for Making Decisions

RISC-V assembly language includes two decision-making instructions, similar to an ***if*** **statement with a *go-to*.**

Conditional branches : An instruction that tests a value and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the test.

bne rs1, rs2, L1



PC= Address of Instruction where branching need to be done. The Label L1 is replaced by the address by assembler.

$PC_{new} = PC_{present}$ + Signed offset

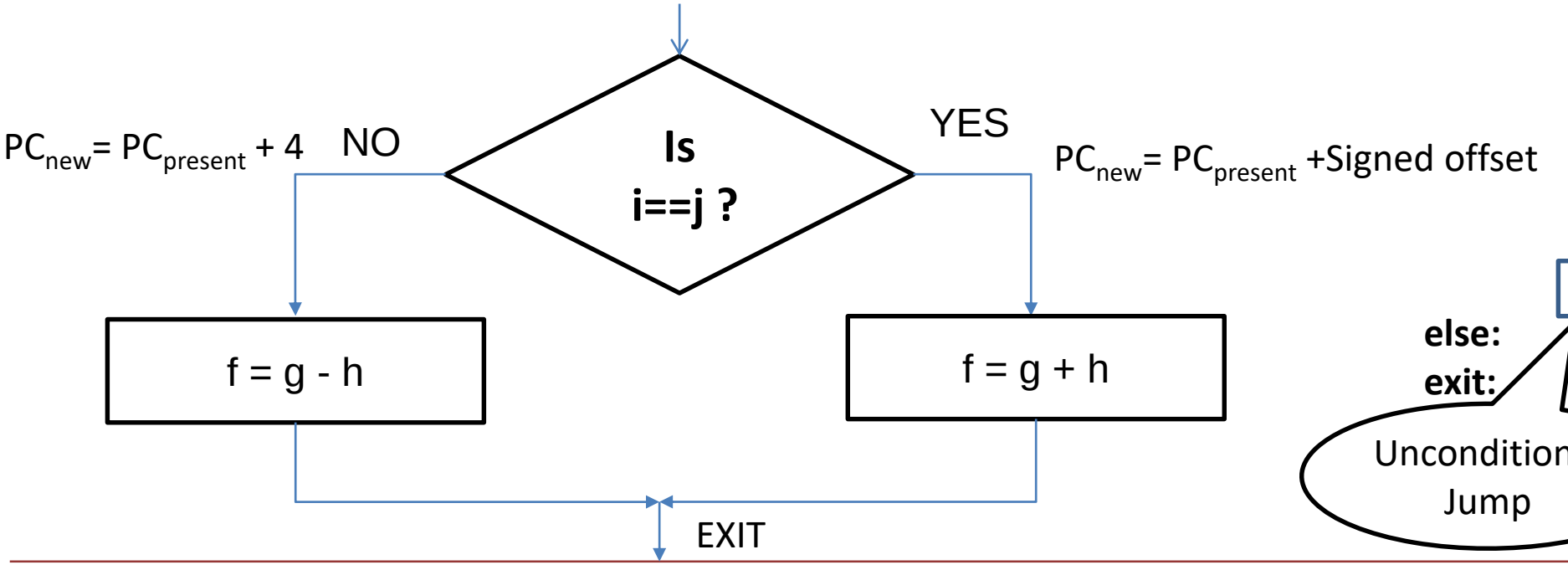
where Signed offset = Immediate($\pm 2^{11}$) * 4

PC – NO change As a result, No change in Sequence of execution i.e., executes next Instruction after branch

Compiling *if-then-else* into Conditional Branches

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;

f	x19
g	x20
h	x21
i	x22
j	x23




```
    bne x22,x23,else
    add x19,x20,x21
    beq x0,x0,exit
else:
    sub x19,x20,x21
exit:
```

Unconditional Jump

Instructions – Language of Computer

Instructions for Making Decisions

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?

```
loop:   slli x10, x22, 2    //   x10 = x22 * 4 = i * 4
        add  x10,x10,x25
        lw x9, 0(x10)    // x9=save[i]
        bne x9,x24,exit
        addi x22,x22,1
        beq x0,x0,loop
exit:
```

	..
	..
addr16	save[4]
addr12	save[i=3]
addr8	save[2]
addr4	save[1]
addr0	save[0]

x25 →

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

Instructions – Language of Computer

Instructions for Making Decisions

Loops : for iterating a computation

Basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

Instructions – Language of Computer

Instructions for Making Decisions

- The full set of comparisons is **less than (<), less than or equal (≤), greater than (>), greater than or equal (≥)**, equal (=), and not equal (≠).
- Comparison of bit patterns must also deal with the dichotomy **between signed and unsigned numbers**.
  Signed Number : bit pattern with a 1 in the most significant bit represents a negative number and, of course, **is less than any positive** number, which must have a 0 in the most significant bit.
  0x8000 FFFF(Negative Number) < 0x7FFF 0000(Positive Number) in signed Number and In Unsigned Number 0x8000 FFFF > 0x7FFF 0000
- RISC-V provides instructions that handle both cases.

Instructions – Language of Computer

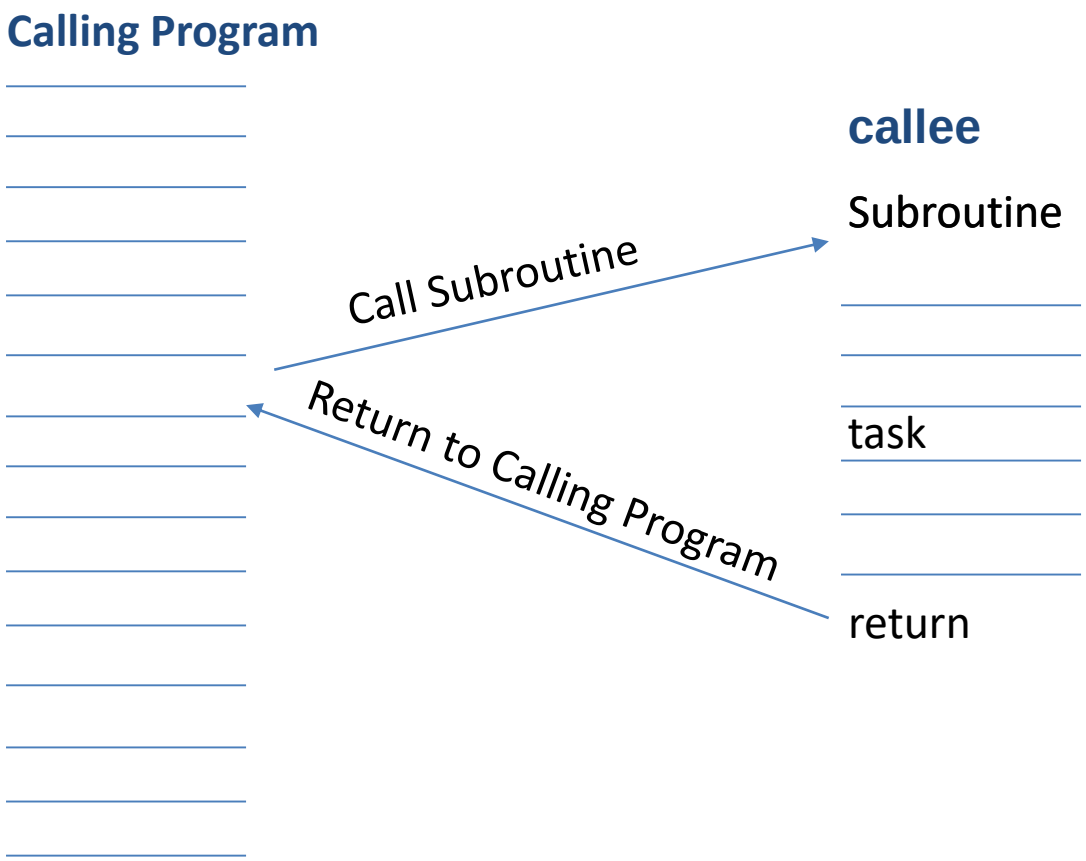
Instructions for Making Decisions

Instruction	Function
blt	The branch if less than (blt) instruction compares the values in registers rs1 and rs2 and takes the branch if the value in rs1 is smaller, when they are treated as two's complement numbers.
bge	Branch if greater than or equal (bge) takes the branch in the opposite case, that is, if the value in rs1 is at least the value in rs2..
bltu	Branch if less than, unsigned (bltu) takes the branch if the value in rs1 is smaller than the value in rs2 when the values are treated as unsigned numbers.
bgeu	branch if greater than or equal, unsigned (bgeu) takes the branch in the opposite case

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

- **Procedure:** A stored subroutine that performs a specific task based on the parameters with which it is provided.



- Parameters act as an interface between the procedure and the rest of the program and data, since they can pass values and return results.
- 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.

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

In the execution of a procedure, the program must follow these six steps:

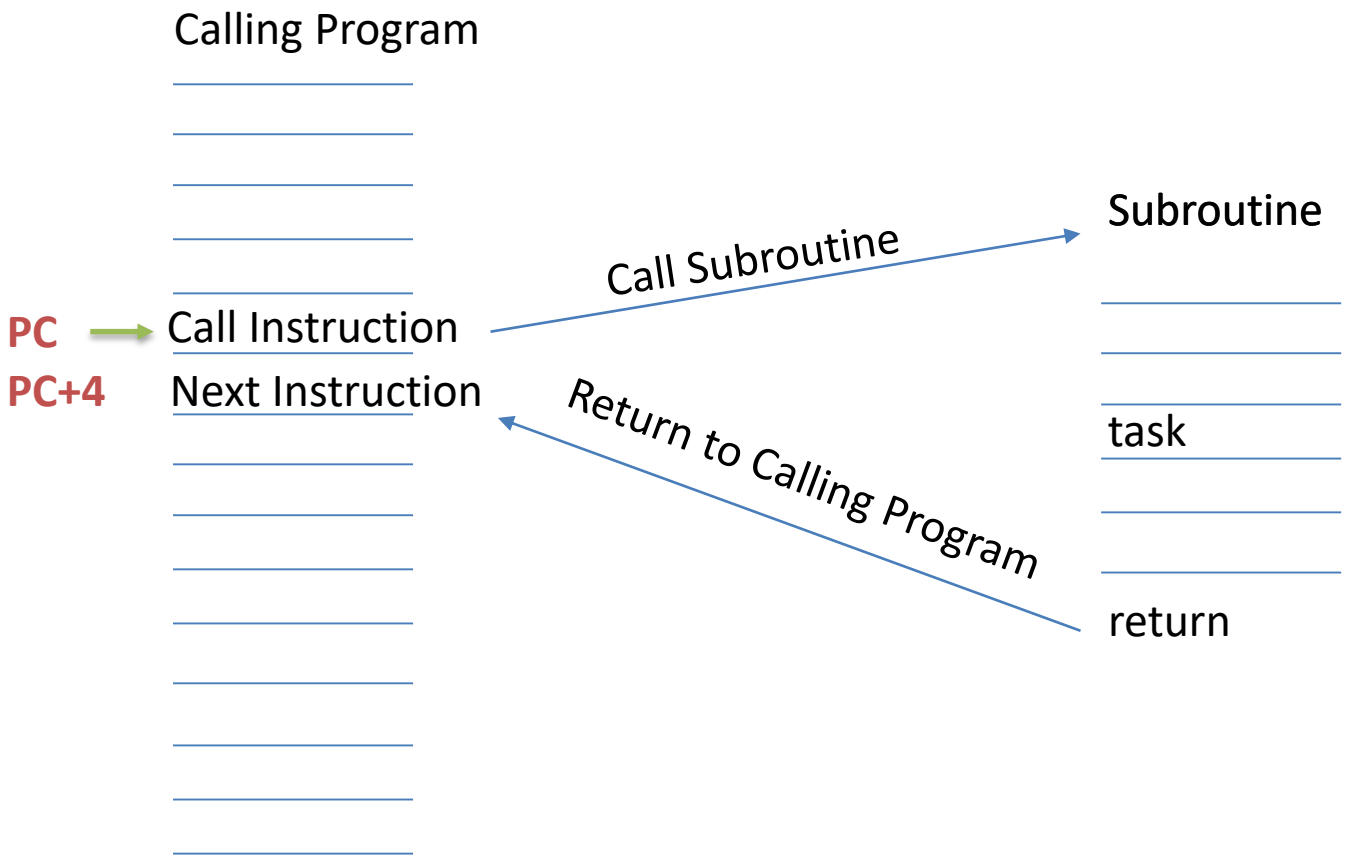
1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

RISC-V features for handling Procedures

Case a) Leaf Procedure: Procedures that do not call others.



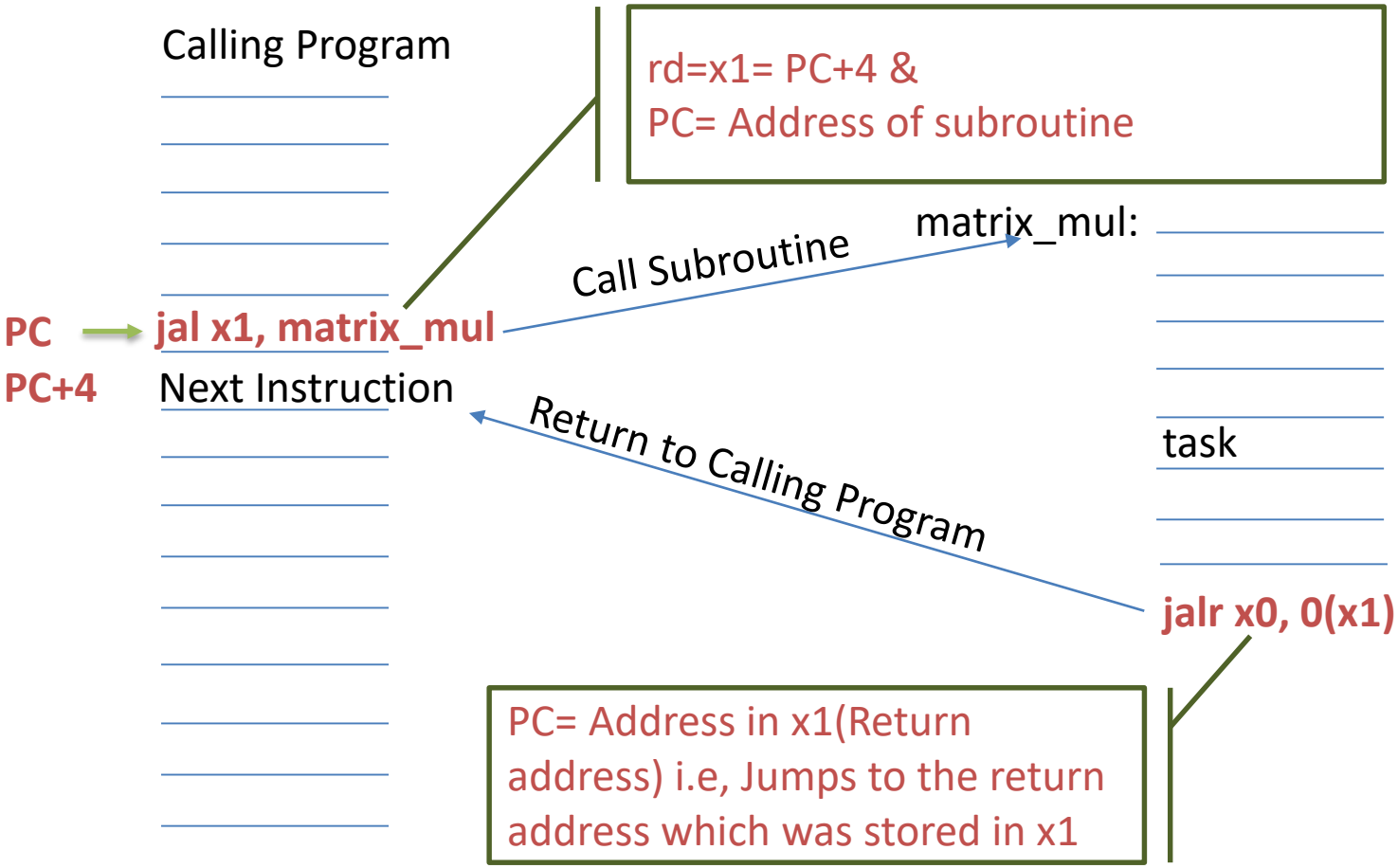
Requirement to handle the procedure

1. Passing & returning parameters:
   x10–x17: eight parameter registers
2. Instruction to call procedure:
   jal x1, ProcedureAddress
3. Making Note of the return address:
   x1:one return address register to return to the point of origin.
4. Instruction to return from subroutine - which should make sure that calling program should start from the point where it was stopped:
   jalr x0, 0(x1)

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

RISC-V features for handling Procedures



Instructions – Language of Computer

Supporting Procedures in Computer Hardware

Using More Registers

Since we must cover our tracks after our mission(procedure is completed) is complete.

What is that we need to do with content of these the register needed by the Caller ????

Any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory

In RISC-V , X10-X17 are used as argument registers.

What if a compiler needs more registers for a procedure than the eight argument Registers ?????

Need to use other registers in addition to X10-X17. Again, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory

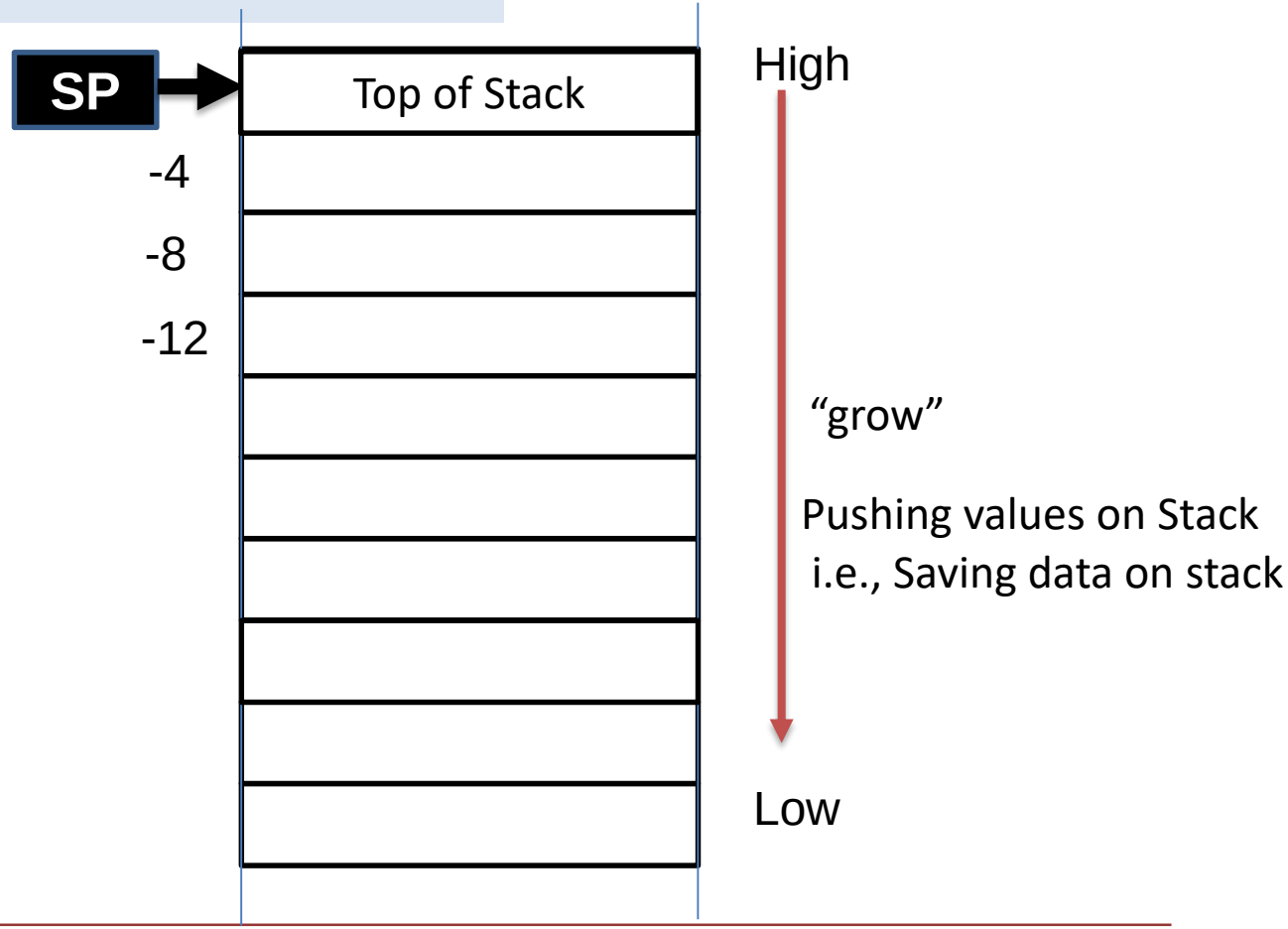
Instructions – Language of Computer

Supporting Procedures in Computer Hardware

Using More Registers

Stack—a last-in-first-out queue

- A data structure for spilling registers organized as a last-in-first-out queue.
- **Stack Pointer (SP) –** Holds the most recently allocated address in a stack. Basically SP shows where registers should be spilled or where old register values can be found.
- In RISC-V, it is register x2 which plays role of SP.
- The SP is adjusted by one word for each register that is saved (Push- Saving data on stack) or restored (Pop- Removing data on stack) .
- Stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by decrementing SP by 4.
- Adding to the stack pointer shrinks the stack, thereby popping values off the stack.



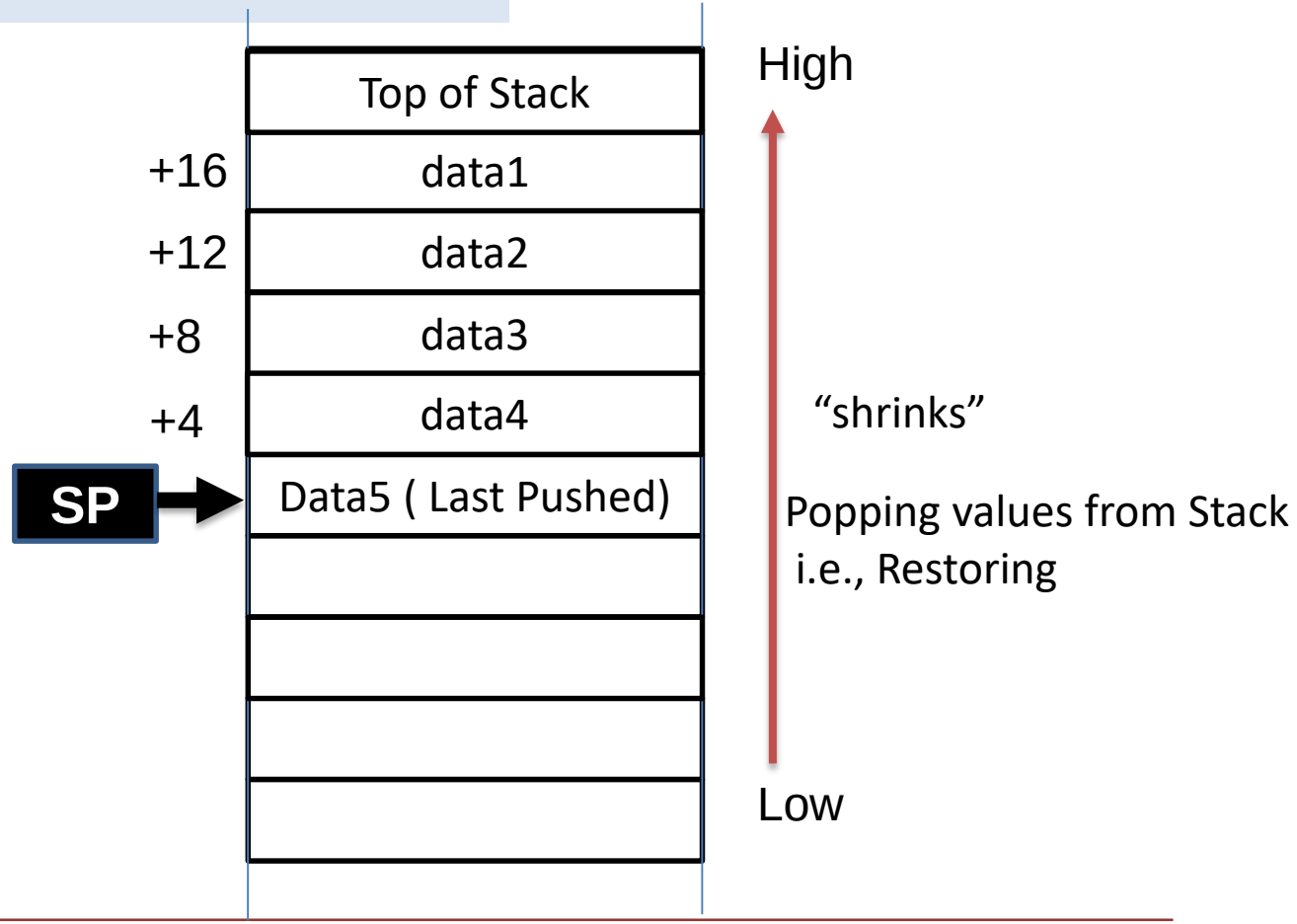
Instructions – Language of Computer

Supporting Procedures in Computer Hardware

Using More Registers

Stack—a last-in-first-out queue

- Stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by decrementing SP by 4.
- Adding to the stack pointer shrinks the stack, thereby popping values off the stack.



Instructions – Language of Computer

Supporting Procedures in Computer Hardware

Compiling a C Procedure that Doesn't Call Another Procedure

The C procedure:

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

What is the compiled RISC-V assembly code?

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

Compiled RISC-V assembly

Assembly procedure should involve

1. Save the registers used by the procedure.
2. Body of the procedure to perform the task.
3. The value of f, should be in parameter register
4. Restore the old values of the registers we saved by "popping" them from the stack.
5. Return from procedure

Note: In RISC-V don't have exclusive Instructions for PUSH data and POP data from stack

leaf_example:

```
addi sp, sp, -12
sw x5, 8(sp)
sw x6, 4(sp)
sw x20, 0(sp)

add x5, x10, x11 //x5 =g + h
add x6, x12, x13 // x6= i + j
sub x20, x5, x6

addi x10, x20, 0

lw x20, 0(sp)
lw x6, 4(sp)
lw x5, 8(sp)
addi sp, sp, 12
jalr x0, 0(x1)  // Return
```

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

The values of Stack and Stack Pointer

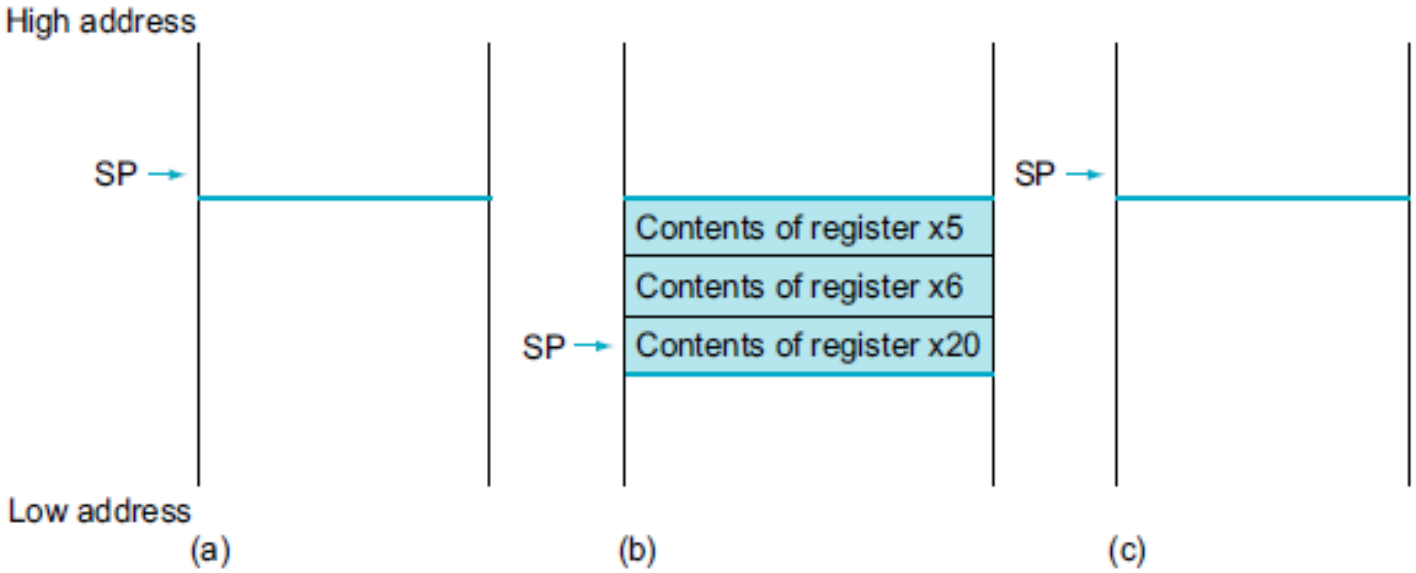


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

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:

1. x5–x7 and x28–x31: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
2. x8–x9 and x18–x27: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

Name	Alias
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5	t0
x6	t1
x7	t2
x8	s0
x9	s1
x10	a0
x11	a1
x12	a2
x13	a3
x14	a4
x15	a5

Name	Alias
x16	a6
x17	a7
x18	s2
x19	s3
x20	s4
x21	s5
x22	s6
x23	s7
x24	s8
x25	s9
x26	s10
x27	s11
x28	t3
x29	t4
x30	t5
x31	t6

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

RISC-V features for handling Procedures

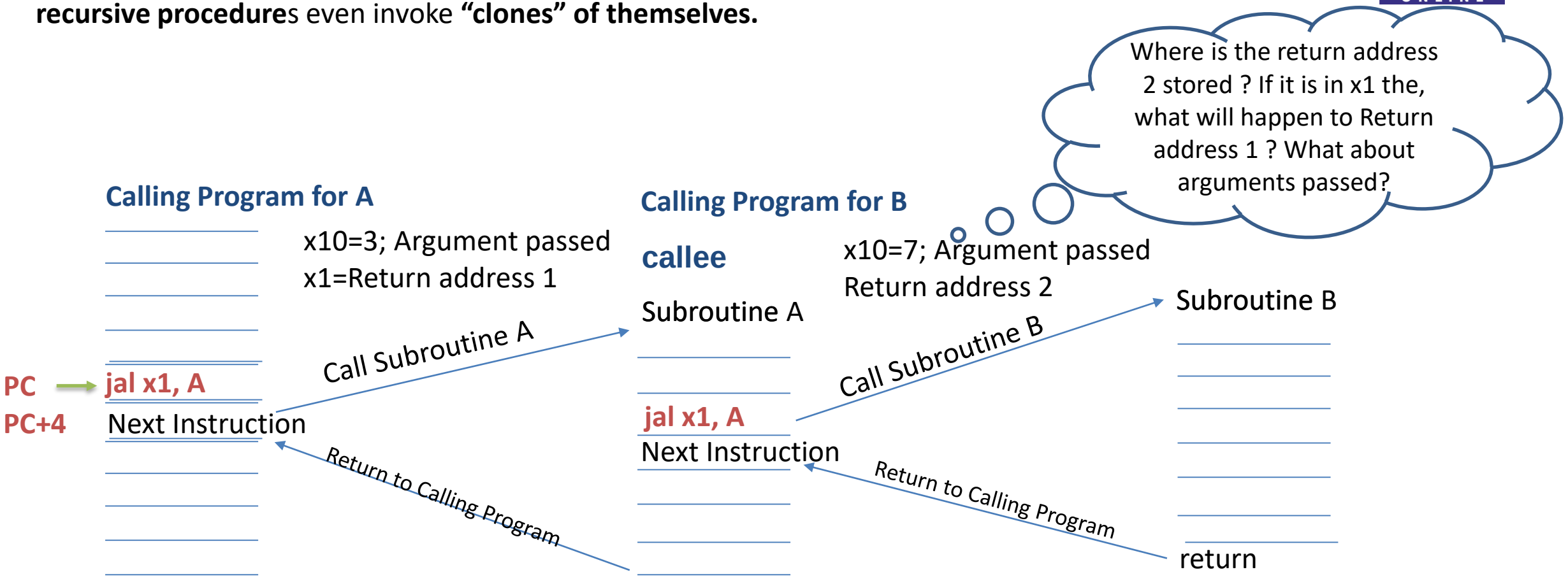
Case b) Nested Procedures: Procedures that invoke other procedures. Moreover, **recursive procedure**s even invoke **“clones” of themselves.**

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

RISC-V features for handling Procedures

Case b) Nested Procedures: Procedures that invoke other procedures. Moreover, **recursive procedure**s even invoke **"clones" of themselves.**



Instructions – Language of Computer

Supporting Procedures in Computer Hardware

RISC-V features for handling Procedures

Case b) Nested Procedures: Procedures that invoke other procedures. Moreover, **recursive procedure**s even invoke **"clones" of themselves.**

Steps to prevent the problem

- ✓ The caller pushes any argument registers (x10–x17) or temporary registers (x5-x7 and x28-x31) that are needed after the call.
- ✓ The callee pushes the return address register x1 and any saved registers (x8- x9 and x18-x27) used by the callee.
- ✓ The stack pointer sp is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory, and the stack pointer is readjusted.

Nested Procedures

Compiling a Recursive C Procedure, Showing Nested Procedure Linking
Ex: Let's tackle a recursive procedure that calculates factorial:

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

What is the RISC-V assembly code?

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

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

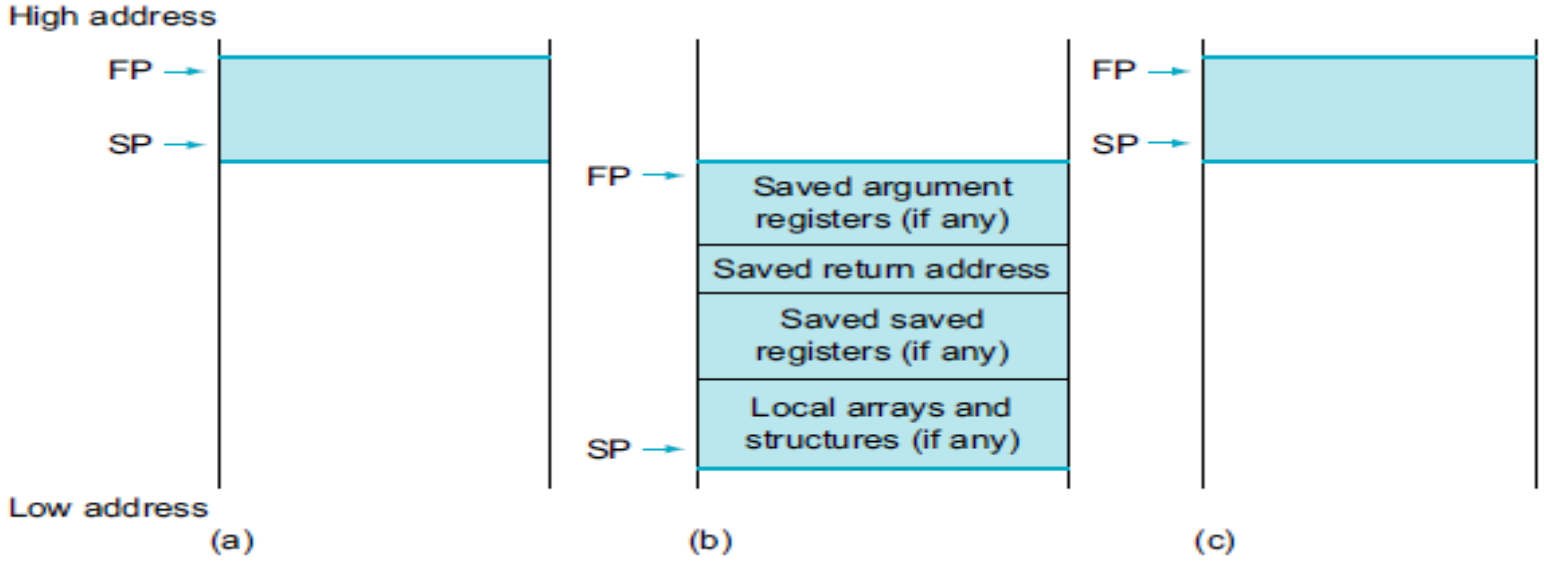
Instructions – Language of Computer

Supporting Procedures in Computer Hardware

Allocating Space for New Data on the Stack

- Stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures.
- **Procedure frame** or **Activation record : It is** the segment of the stack containing a procedure's saved registers and local variables

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



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

Instructions – Language of Computer

Supporting Procedures in Computer Hardware

RISC-V register conventions for assembly language.

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

Instructions – Language of Computer

RISC-V Addressing for wide immediate and addresses (Unit3)

How do we Initialize 32-bit or larger constants or addresses in RISC-V ?

- RISC-V I-type Instruction support 12 bit immediate values.
- Although constants are frequently short and fit into the 12-bit fields, sometimes they are bigger.
- The RISC-V instruction set includes the instruction ***Load upper immediate(lui)***

Syntax		lui rd, imm_{20}	
Operation:		load a 20-bit constant into bits 12 through 31 of a destination register. Rd[31:12] = imm_{20}	
Example:		lui x19, 3D0H;	
Before Execution			
x19=	=	0000 0000 0000 0000 0000	0000 0000 0000b
After Execution			
x19=	=	**0000 0000 0011 1101 0000**	0000 0000 0000b (0x003D0 000)

Instructions – Language of Computer

RISC-V Addressing for wide immediate and addresses

Upper Format for Upper Immediate

- ✓ Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- ✓ One destination register, rd
- ✓ This format is used for lui and **auipc**

Instructions – Language of Computer

RISC-V Addressing for wide immediate and addresses

How do we Initialize 32-bit or larger constants or addresses in RISC-V ?

Write a assembly code to initialize 0x003D0500 in register X19

lui X19,0x003D0
addi X19,X19,0X500

x19	0x0000 0000	0000 0000 0000 0000 0000	0000 0000 0000

Before execution of lui X19,0x003D0

x19	0x003D0000	0000 0000 0011 1101 0000	0000 0000 0000

After execution of lui X19,0x003D0

imm12	0x500	0000 0000 0000 0000 0000	**0**101 0000 0000

x19	0x003D0000	0000 0000 0011 1101 0000	0101 0000 0000

After execution of addi X19,X19,0x500

Note: addi12-bit immediate is always **sign-extended**

Instructions – Language of Computer

RISC-V Addressing for wide immediate and addresses

How do we Initialize 32-bit or larger constants or addresses in RISC-V ?

Write a assembly code to initialize **0xDEADBEEF** in register X19

lui X19,0xDEADB
addi X19,X19,0XEEF

x19	0x0000 0000	0000 0000 0000 0000 0000	0000 0000 0000

Before execution of lui X19,0x003D0

x19	0xDEADB000	1101 1110 1010 1101 1011	0000 0000 0000

After execution of lui X19,0x003D0

imm12	0x500	1111 1111 1111 1111 1111	1110 1110 1111

x19	0xDEADAEEF	1101 1110 1010 1101 1010	1110 1110 1111

After execution of addi X19,X19,0x500

x19	0xDEADAEEF - It is Not a expected Initialization. What has to be done to initialize x19 with the correct result?

Addressing in Branches

- The RISC-V branch instructions use an RISC-V instruction format with a 12-bit immediate. This format can represent branch addresses from −4096 to 4094, in multiples of 2 as it is only possible to branch to even addresses.

bne x10, x11, 2000 rs1=x10, rs2=x11 and imm= 0x7D0 = 0111 11 01000 0

The above instruction can be assembled into the S format

0011111	01011	01010	001	01000	1100111
imm[12:6]	rs2	rs1	funct3	imm[5:1]	opcode

where the **opcode** for conditional branches is **1100111**two and **bne's funct3** code is $\mathbf{001}_{\mathbf{two}}$.

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	beq

Instructions – Language of Computer

RISC-V Addressing for wide immediate and addresses

Addressing in Branches

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	bgeu

Instructions – Language of Computer

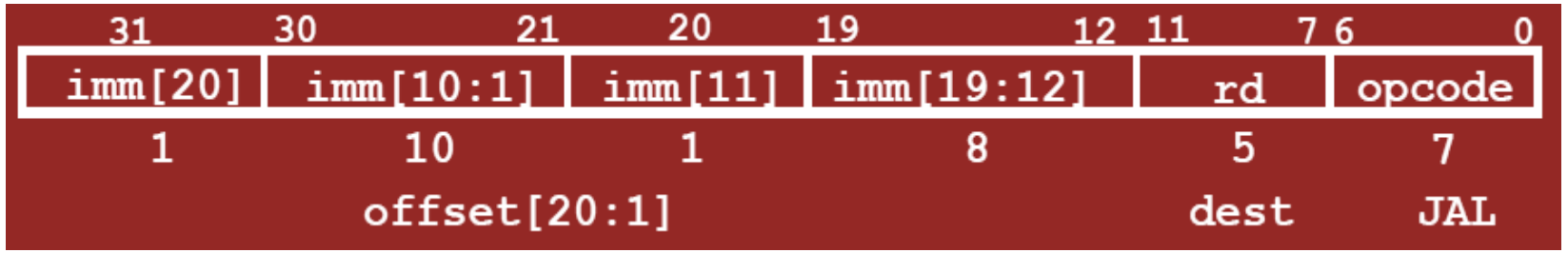
RISC-V Addressing for wide immediate and addresses

Addressing in Branches

What do we do if destination is > 2^{10} instructions away from branch?

2^{10} =1024 words = 1024*4=4092

UJ Format for Jump Instructions



The UJ-type format's address operand uses an unusual immediate encoding, and it cannot encode odd addresses.

jal x0, 2000 // go to location 2000_{ten} = 0111 1101 0000

00000000001111101000	00000	1101111
imm[20:1]	rd	opcode

Addressing in Branches

PC- Relative Addressing

- 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 today.

  Program counter Register Branch offset

- This sum allows the program to be as large as 232 and still be able to use **conditional branches**, solving the branch address size problem. Then the question is, which register?

  jalr x0, 0(x1)

- **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.
- Refer to the number of w*ords* between the branch and the target instruction, rather than the number of bytes.

Addressing in Branches

Showing Branch Offset in Machine Language
The *while* loop on page 100 was compiled into this RISC-V assembler code:

```
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

If we assume we place the loop starting at location 80000 in memory, what is the RISC-V machine code for this loop?

Instructions – Language of Computer

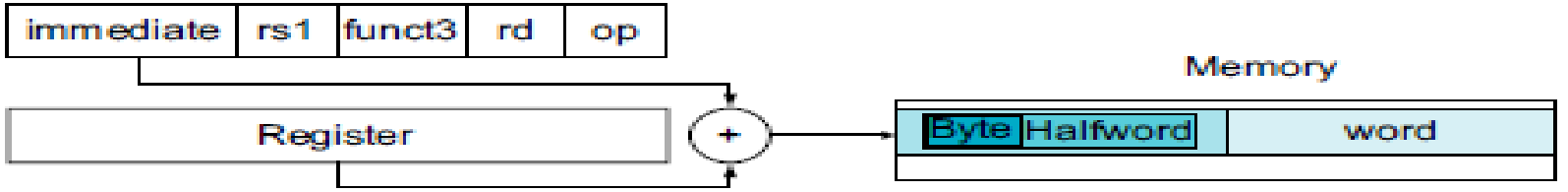
Illustration of four RISC-V addressing modes

Addressing in Branches

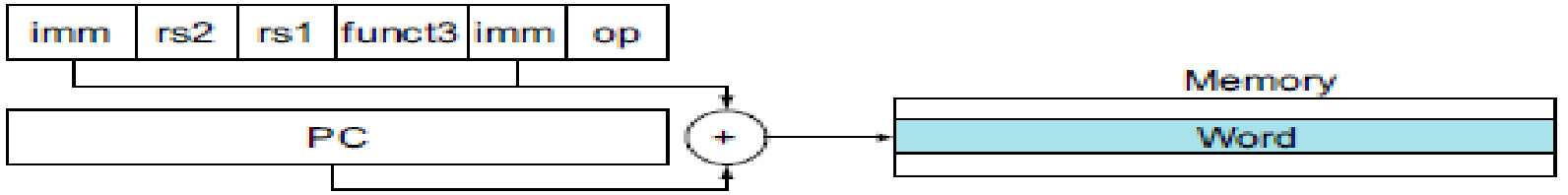
1. Immediate addressing

2. Register addressing

3. Base addressing



4. PC-relative addressing



Instructions – Language of Computer

RISC-V instruction encoding

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
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.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

Decoding Machine Code

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

00578833hex

0000 0000 0101 0111 1000 1000 0011 0011b

funct7	rs2	rs1	funct3	rd	opcode
0000 000	00101	01111	000	10000	0110011 b
	x5	x15		x16	

THANK YOU

Mahesh Awati
Department of Electronics and Communication

mahesha@pes.edu

+91 9741172822

RISC V Instruction Set Architecture (ISA)

RISC-V instruction encoding

How Integer overflow on addition of two unsigned values is detected

Integer overflow on addition of two unsigned values can be detected by **comparing the result to one of the operands to see** if it's smaller, which can probably be done in one additional instruction. I'm sure there's suitably clever tricks for signed addition as well as subtraction. All stuff that a compiler can implement appropriately.

Maybe this limits the relative usefulness of risc-v silicon for programming languages that do a lot of safety checking. If your CPU spends 10% of its time doing additional runtime "safety" instructions vs. other architectures, but takes up 10% less space or runs 10% faster due to the simplicity, it seems like a wash though. Programs that don't need the runtime safety can then run 10% faster. Additionally, nothing is stopping someone from designing hardware that accelerates common instruction sequences. A CPU could still internally implement status flags and decode common compiler instruction sequences faster than otherwise

RISC V Instruction Set Architecture (ISA)

RISC-V instruction encoding

Assume that registers x5 and x6 hold the values x80000000 and 0xD0000000, respectively.

a) What is the value of x30 for the following assembly code?

```
add x30, x5, x6
```

b) Is the result in x30 the desired result, or has there been overflow?

c) For the contents of registers x5 and x6 as specified above, what is the value of x30 for the following assembly code?

```
sub x30, x5, x6
```

d) Is the result in x30 the desired result, or has there been overflow?

e) For the contents of registers x5 and x6 as specified above, what is the value of x30 for the following assembly code?

```
add x30, x5, x6
add x30, x30, x5
```

f) Is the result in x30 the desired result, or has there been overflow?

Variables	Registers
f	x5
g	x6
h	x7
i	x28
j	x29
Base Address of A	x10
Base Address of B	x11

a) 0x50000000
b) overflow
c) 0xB0000000
d) no overflow
e) 0xD0000000
f) overflow