Computer Organization and Architecture

COSC 2425

Lecture – 6

Sept 7th, 2022

Acknowledgement: Slides from Edgar Gabriel & Kevin Long

Chapter 2

Instructions: Language of the Computer

Review

Instruction Set

- Add
- Multiply
- Divide
- Load Data

Instruction Set

Computer 1

ISA1

Computer 2

ISA2

A manual to instruct he computer.

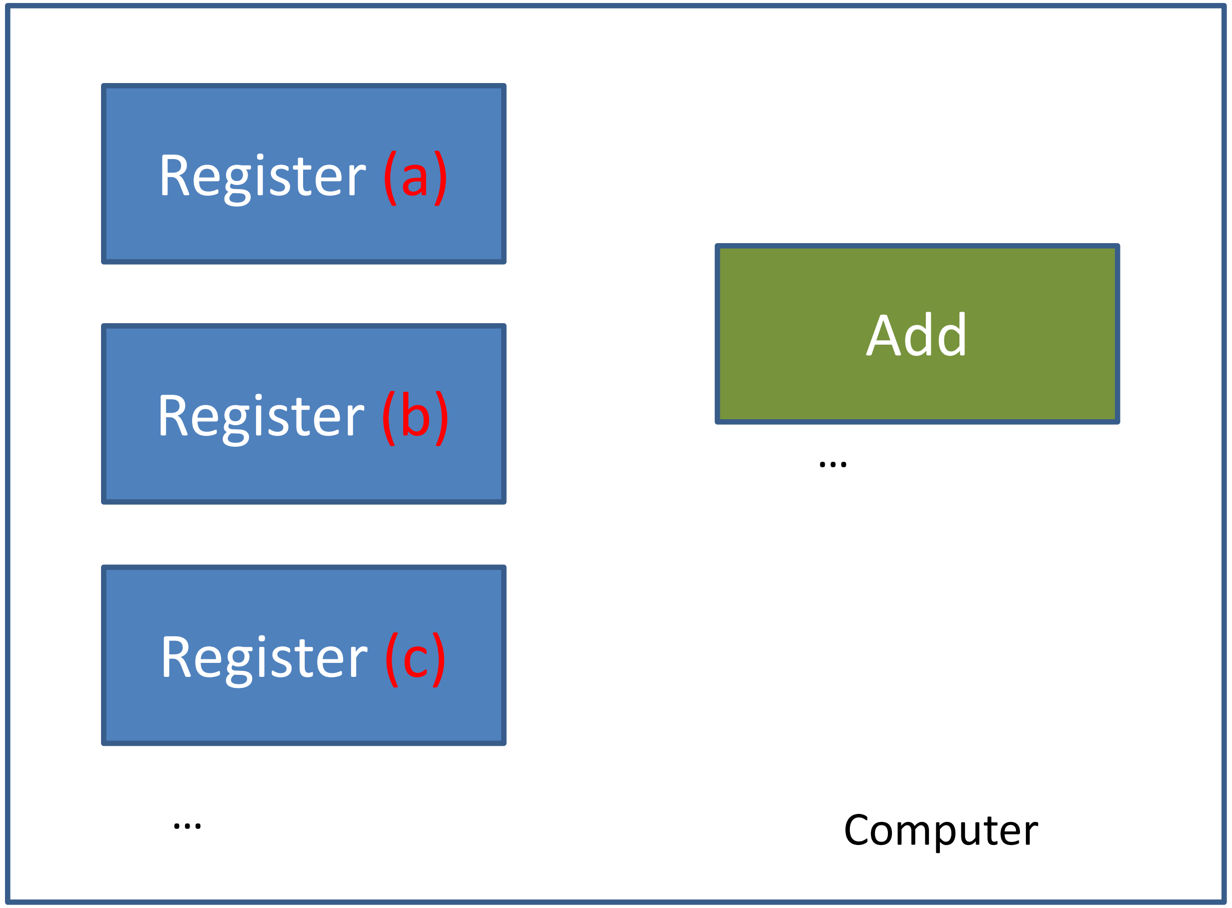
Review

The ARMv8 Instruction Set

- A subset, called LEGv8, used as the example throughout the book
- Commercialized by ARM Holdings (www.arm.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See ARM Reference Data tear-out card

Review

Operations of Computer Hardware



1. Has multiple registers, and logic gates to perform operations. E.g. add
2. Registers contain/store data.
3. Operators (like Add), can only access data in the registers.

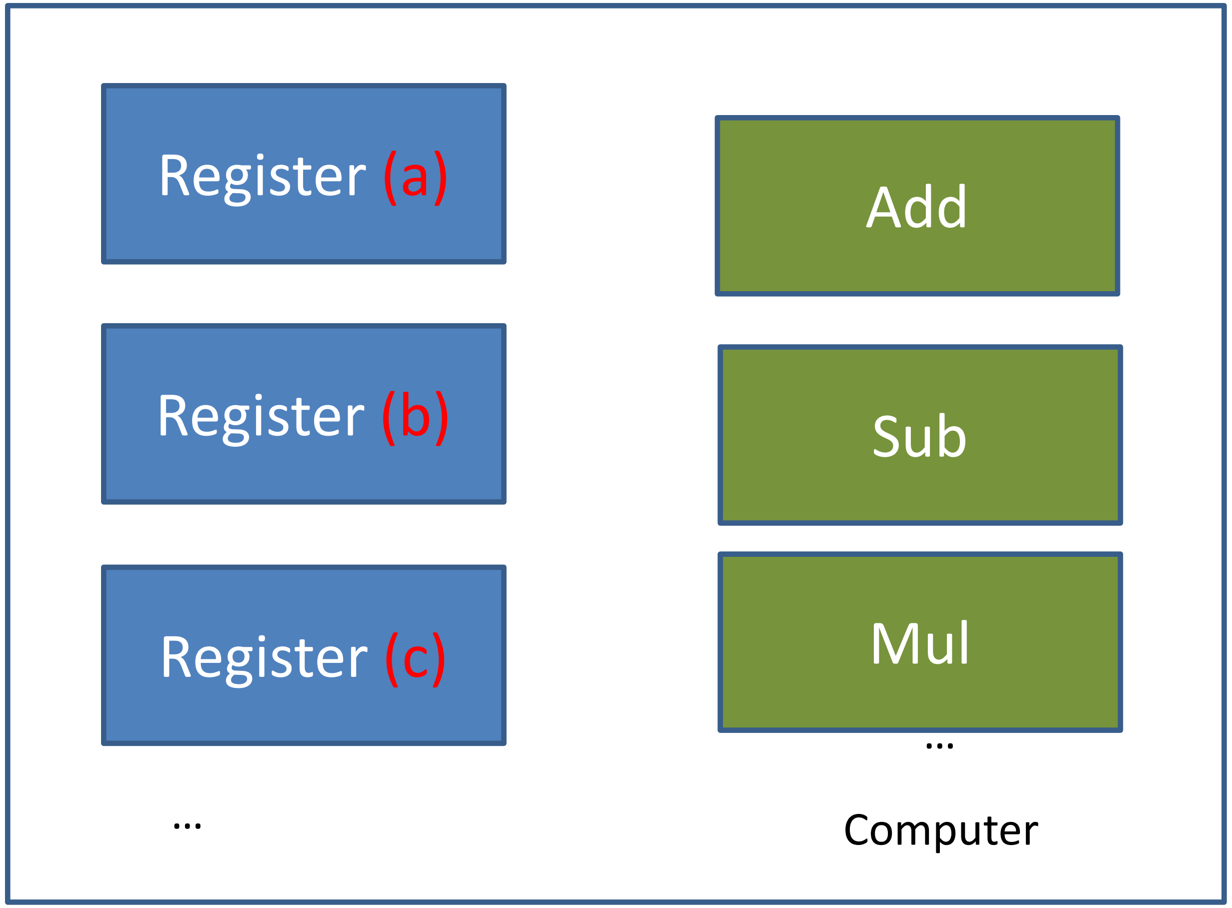
1. To instruct computer to
   1. **Add** (operation)
   2. **Values** in register **b and c** (Source Variables)
   3. Store the **result in a** (Destination Variable)

LEGv8 Instruction:

$$\boldsymbol{ADD\ a, b, c}$$

Review

Operations of Computer Hardware



1. To instruct computer to
   1. **Add** (operation)
   2. **Values** in register **b and c** (Source Variables)
   3. Store the **result in a** (Destination Variable)

LEGv8 Instruction:

$$\underbrace{\boldsymbol{ADD}}_{\text{One operation}}\ \underbrace{\boldsymbol{a, b, c}}_{\text{Has three variables}}$$

Design Principle 1: Simplicity favors regularity

All LEGv8 **Arithmetic Instructions** perform only one operation and always has exactly three variables

$SUB\ a, b, c$ // subtract instrcution $(a = b\ - c)$
$MUL\ a, b, c$ // multiply instruction $(a = b\ * c)$

Review

Example - 1

$$a = b + c + d + e$$

$$\boxed{\begin{array}{ll} ADD\ a, b, c & // a = b + c \\ ADD\ a, a, d & // a = a + d \\ ADD\ a, a, e & // a = a + e \end{array}}$$

3 instruction to sum 4 variables

Review

Example - 3

$$f = (g + h) - (i + j)$$

$$ADD\ t0, g, h\ \ //\ t0 = g + h$$
$$ADD\ t1, i, j\ \ //\ t1 = i + j$$
$$SUB\ f, t0, t1\ \ //\ f = t0 - t1$$

t0, t1: temporary variables created by the compiler

Review

Example - 3

$$f = (g + h) - (i + j)$$

$$\begin{array}{ll} ADD\ t0, g, h & // t0 = g + h \\ ADD\ t1, i, j & // t1 = i + j \\ SUB\ f, t0, t1 & // f = t0 - t1 \end{array}$$

Variables $\underbrace{\text{t0, t1, f, g, h, i, j}}_{\textbf{Stored in registers}}$

Review

Bits, Bytes, Words, and Double Words

For our course!!!

0 ➔ 1 **bit** of data
1 ➔ 1 **bit** of data

10011101 (8 bits) ➔ 1 **byte** of data

Overtime this became a basic unit of data.
Older system represented letters using bytes
As a results most memory hardware

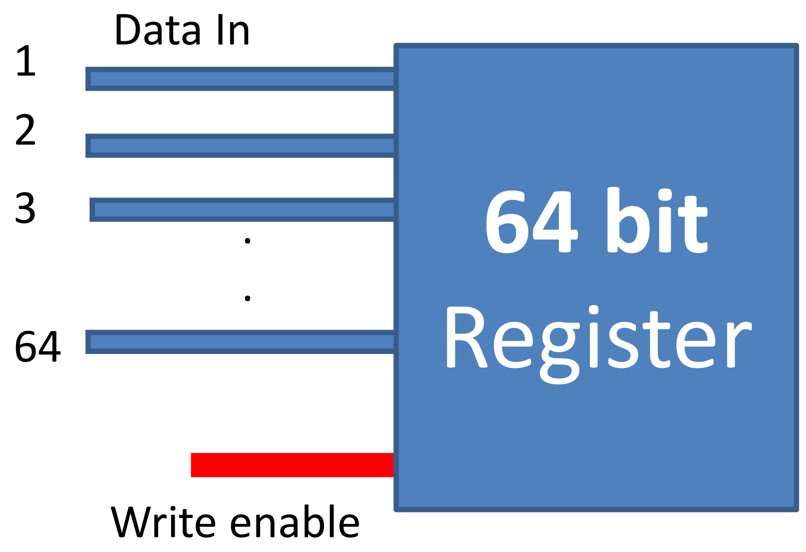
10011101	10010001	10010101	10010101 ➔ **4 bytes** is a **word** (32 bits)
1 byte	**2 byte**	**3 byte**	**4 byte**

10011101	10010001	10010101 …	10010101 ➔ **8 bytes** is a **Doubleword** (64 bits)
1 byte	**2 byte**	**3 byte**	**7 byte**

Review

Operands of the Computer Hardware

- LEGv8 Register size – **64 Bits**
  - **Double words**



Review

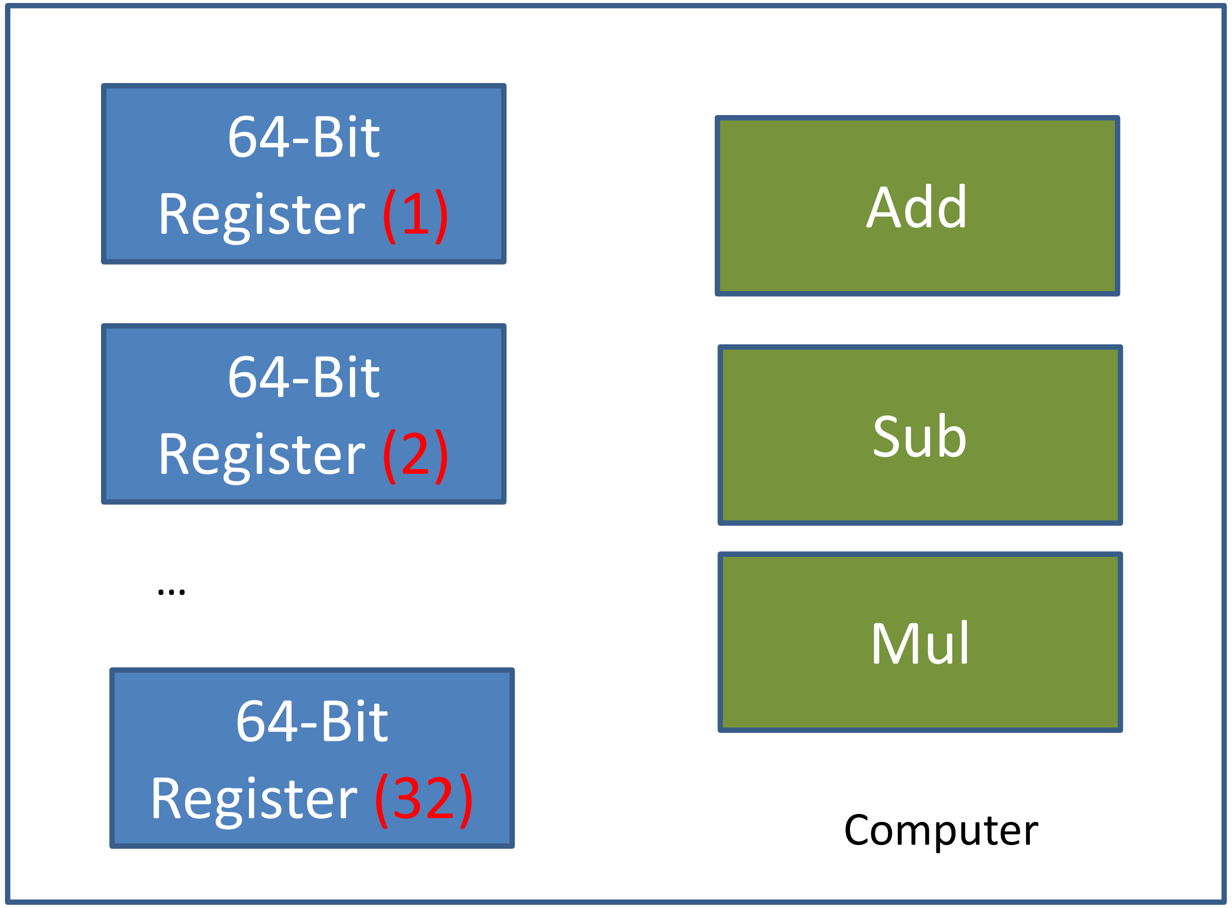
Operands of the Computer Hardware

- LEGv8 Register size – 64 Bits
- Total of **32 registers** (64-bit)

Why only 32??

Design Principle 2: Smaller is faster

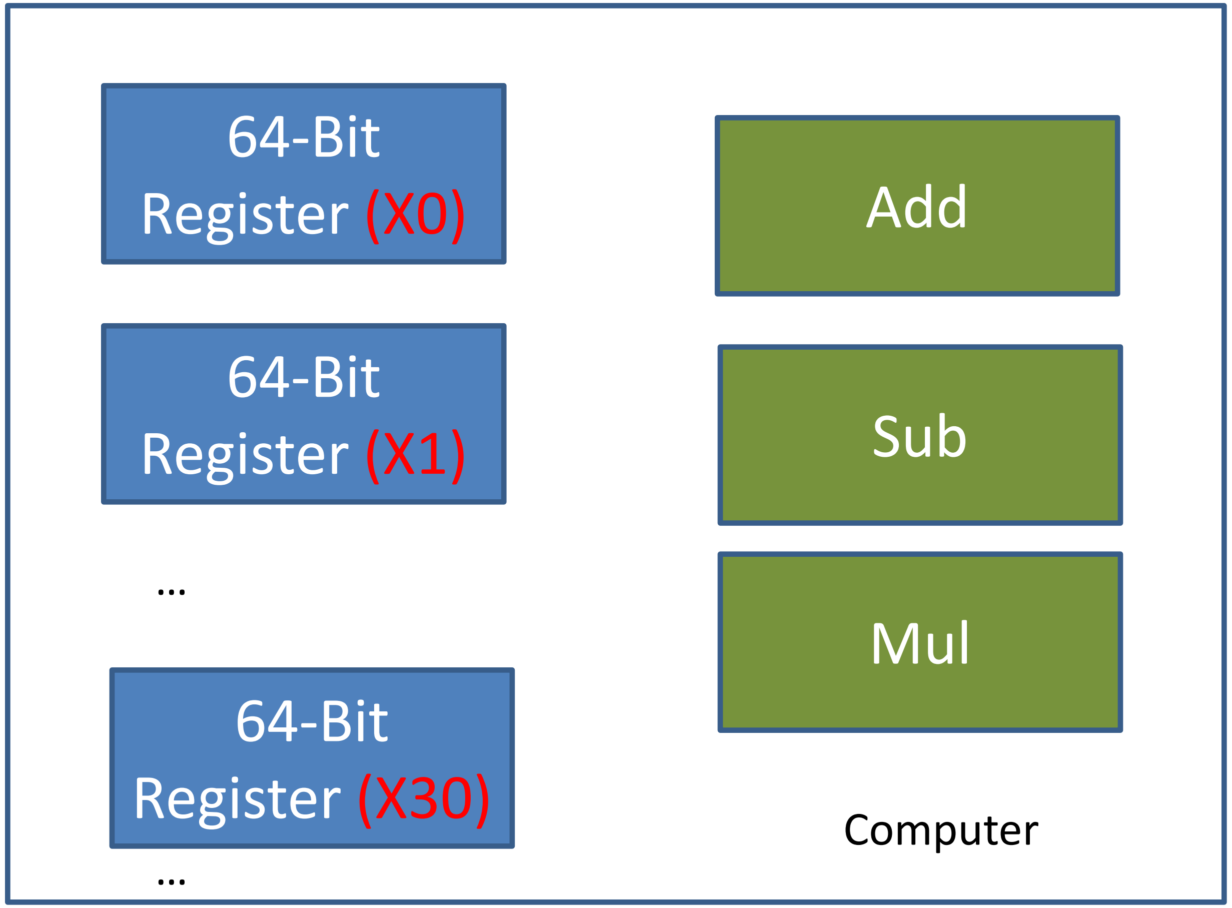
1. Having more registers may increase the clock cycle time (longer for electronic signals to travel)
2. Size of instructions (number of bits) is predefined and same for all instructions. More register requires more bits to specify registers.
   1. 32 registers – require 5 bits max
   2. 64 registers may require 6 bits.



Review

Operands of the Computer Hardware

- LEGv8 Register size – 64 Bits
- Total of **32 registers** (64-bit)
- Register name convention use **X** as prefix.
- Registers are names
  - X0
  - X1
  - …
  - X30
  - XZR(X31) (Exception, more on this later…!)



Review

Example – 3 (Again)

$$f = (g + h) - (i + j)$$

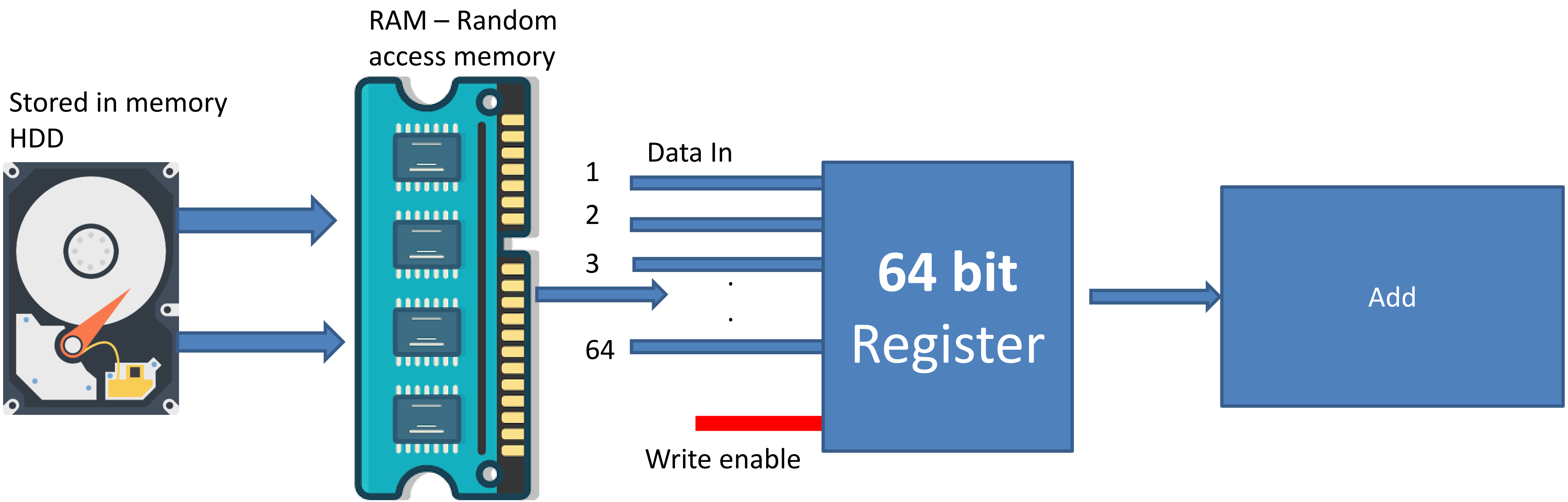
$$f, \ldots, j \; store \; in \; registers \; X19, X20, \ldots, X23$$

$$Two \; temporary \; registers \; are \; availabe \; X9 \,\&\, X10$$

$$\boxed{\begin{array}{l} ADD \; X9, X20, X21 \quad // \, X9 = g + h \\ ADD \; X10, X22, X23 \, // \, X10 = i + j \\ SUB \; X19, X9, X10 \;\; // \; f = X9 \, - X10 \end{array}}$$

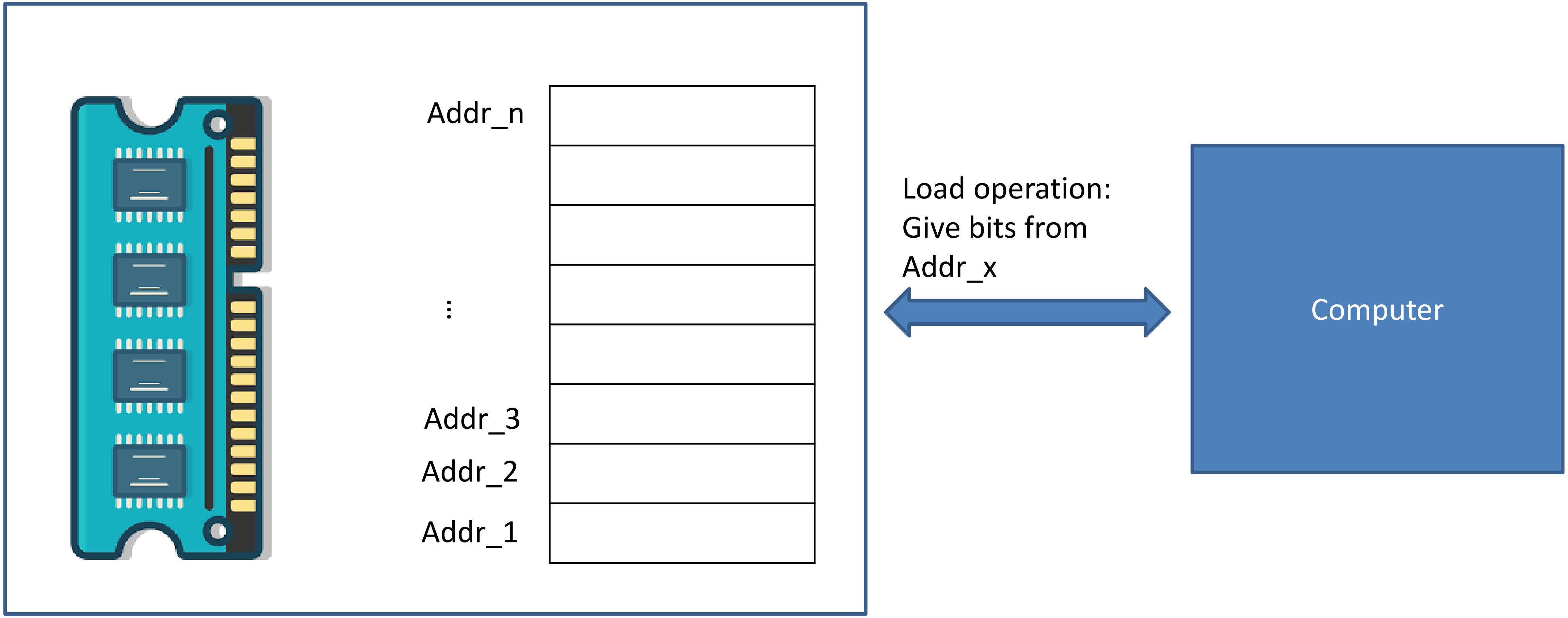
Review

Review: Half-Adder with manual input



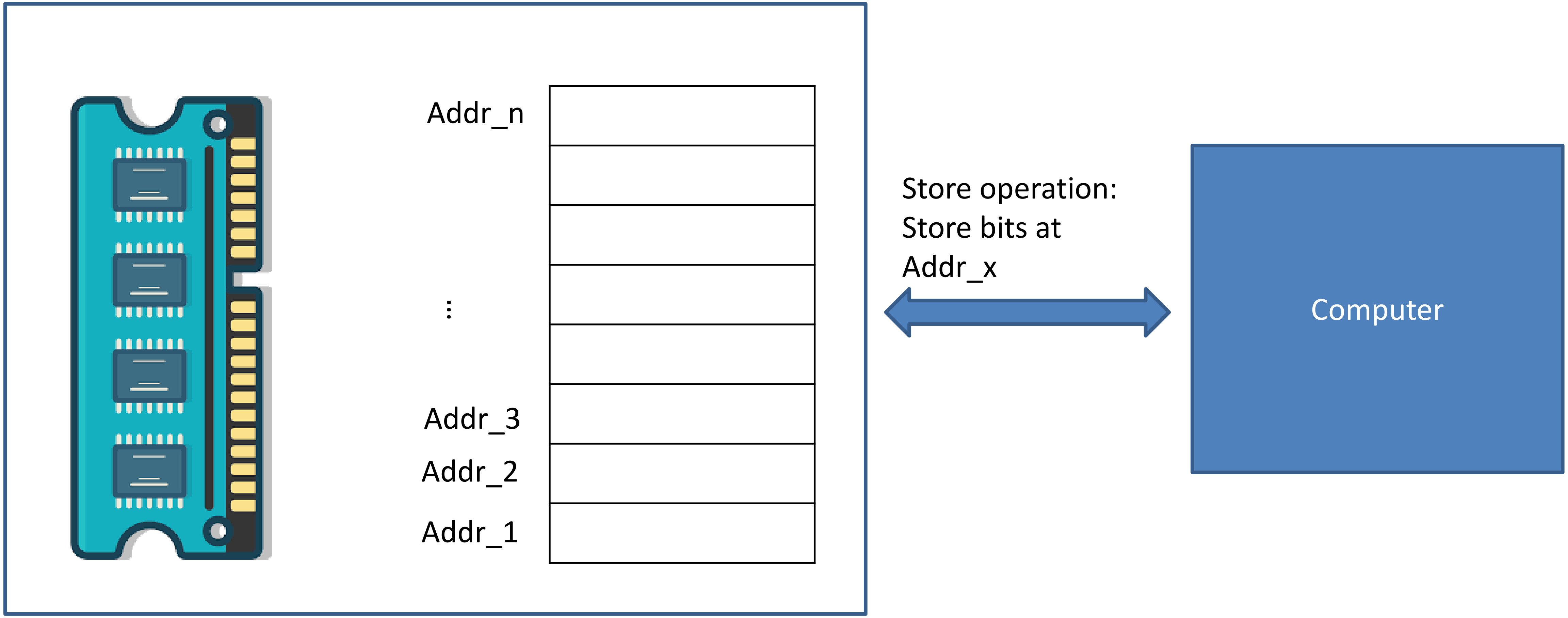
Review

Load Operation



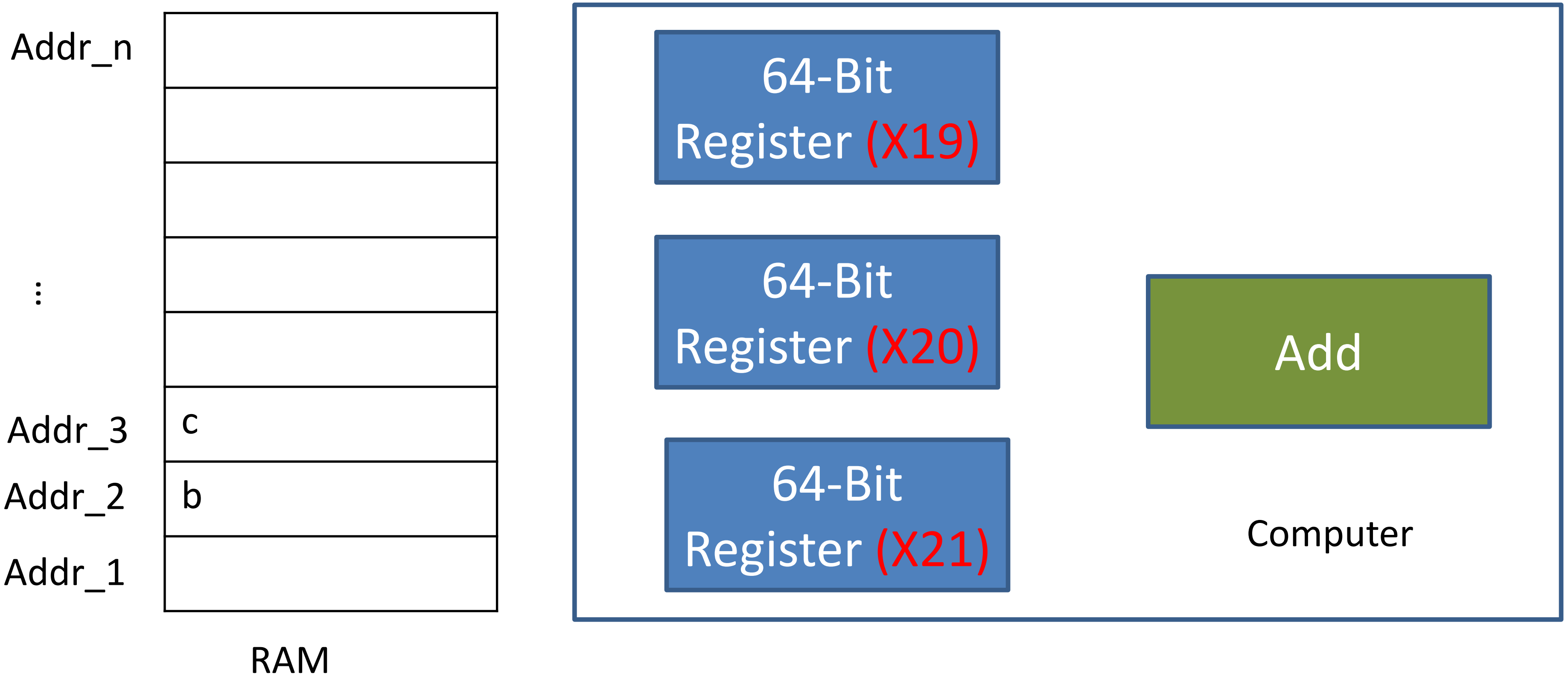
Review

Store Operation



Review

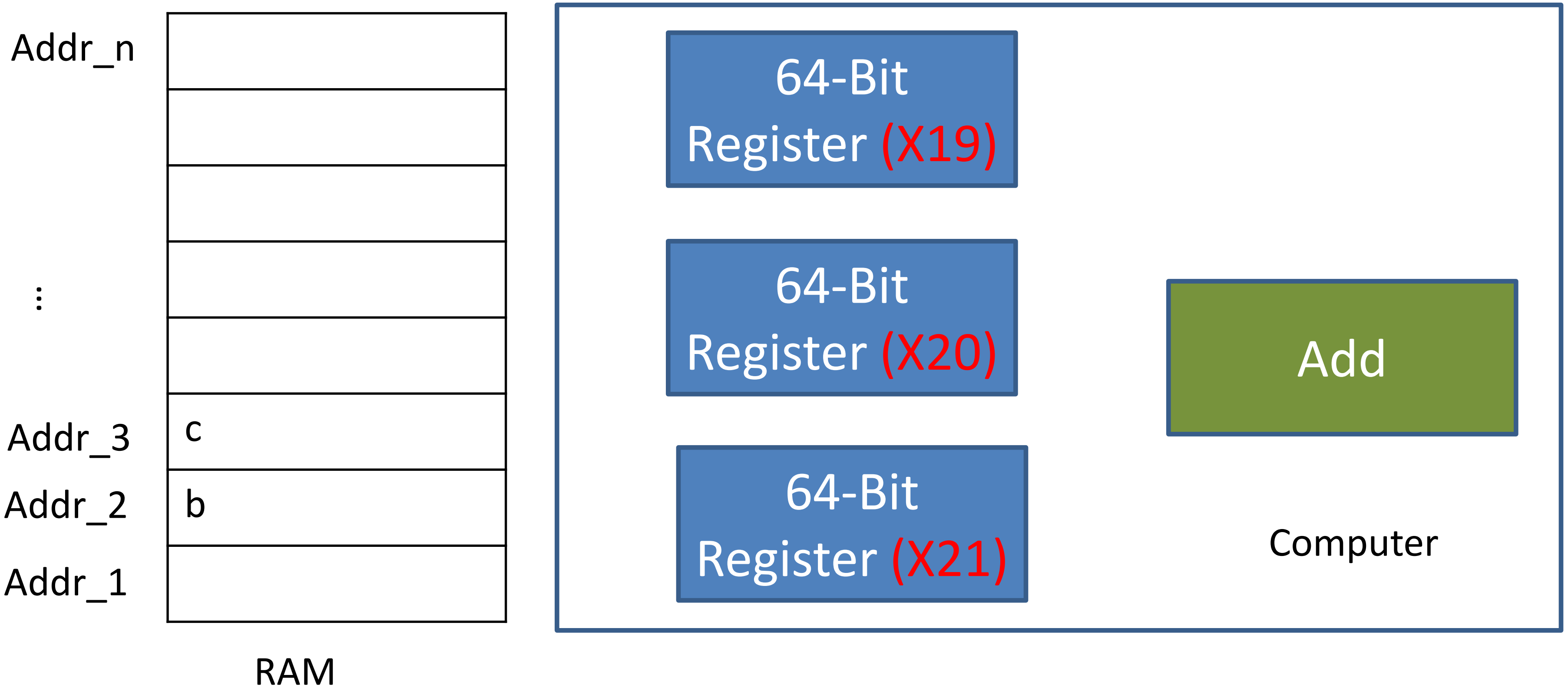
Memory Operand, LOAD



$a = b + c$

Review

Memory Operand , LOAD



$a = b + c$

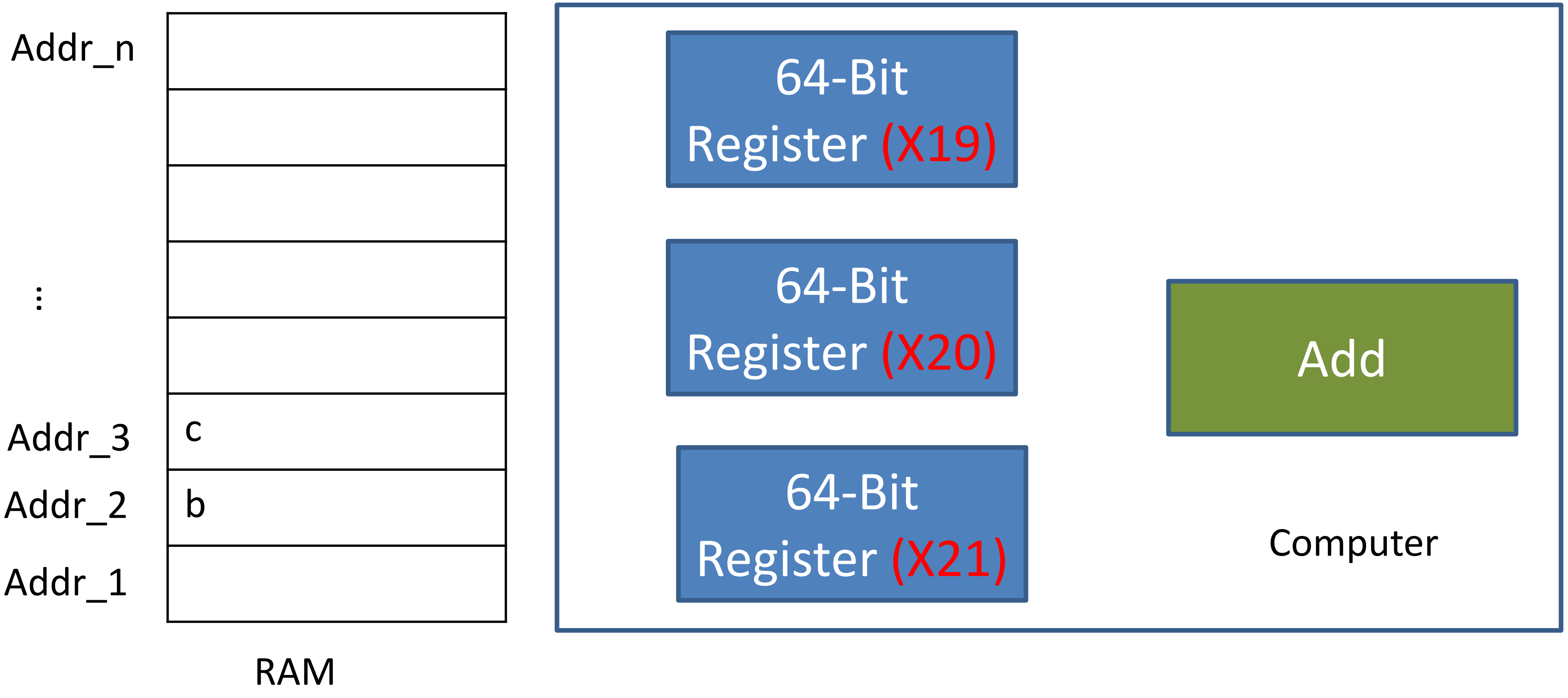
Load Addr_3 (c) to register X19
Load Addr_2 (b) to register X20
ADD X21, X19, X20

For Load:
Need to specify the ram memory address, and the register to load the value into.

memory address-> also in bits, and needs to be stored in another register.

Review

Memory Operand , LOAD



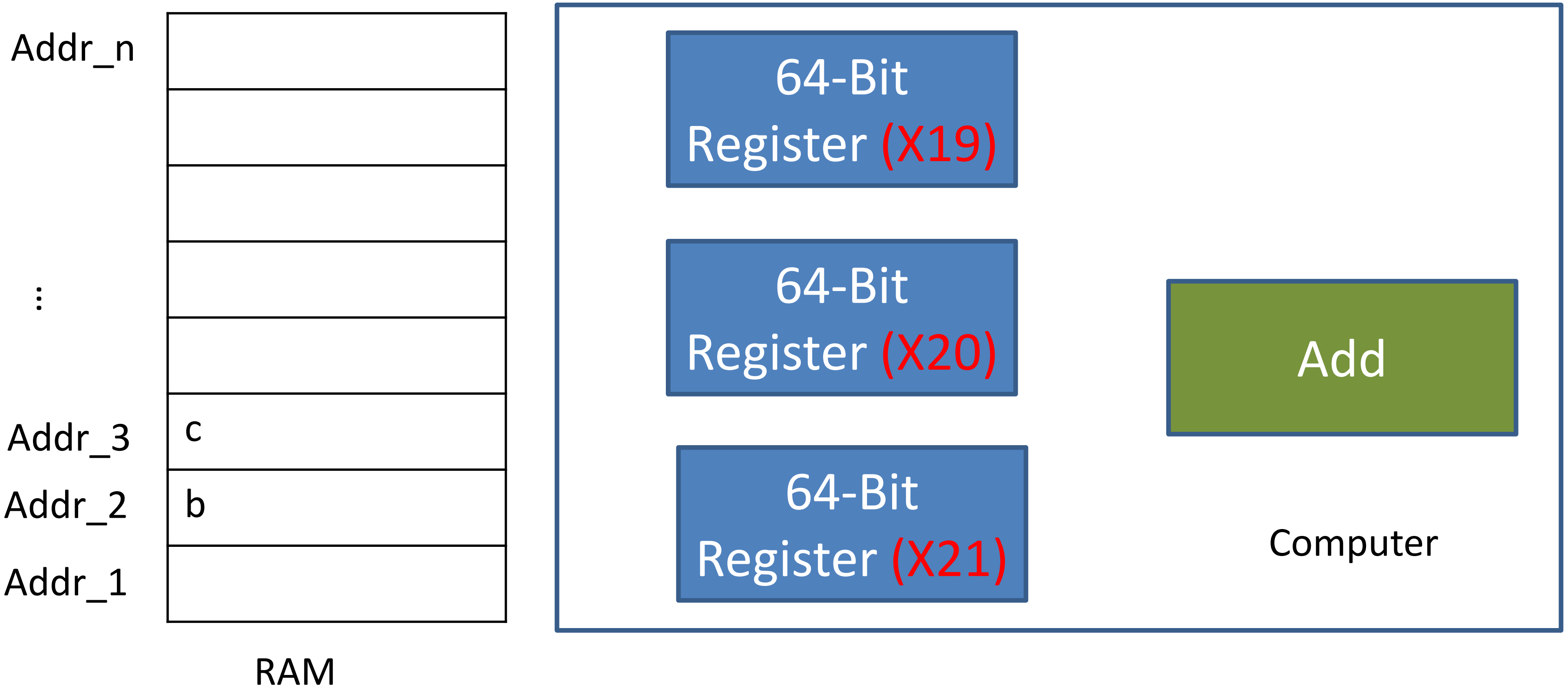
$$a = b + c$$

Load Addr_3 (c) to register X19
Load Addr_2 (b) to register X20
ADD X21, X19, X20

Lets assume memory address is stored in X22

Review

Memory Operand , LOAD



$a = b + c$

Load Addr_3 (c) to register X19
Load Addr_2 (b) to register X20
ADD X21, X19, X20

Lets assume memory address is stored in X22

The **LEGv8 instruction** to:
Load Addr_2 (a) to register X19

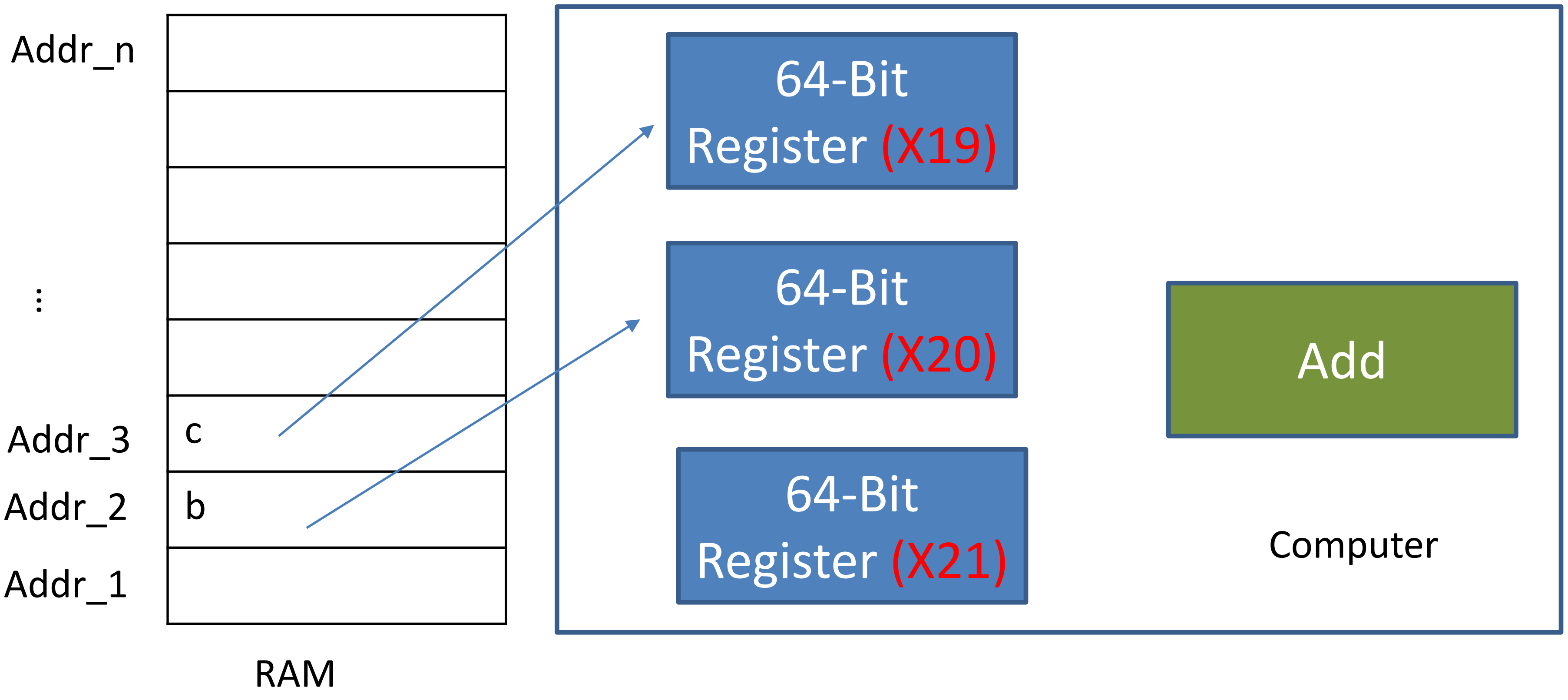
$LDUR\ X19, [X22, \#const]$

Destination register

Ram address

Review

Memory Operand , LOAD



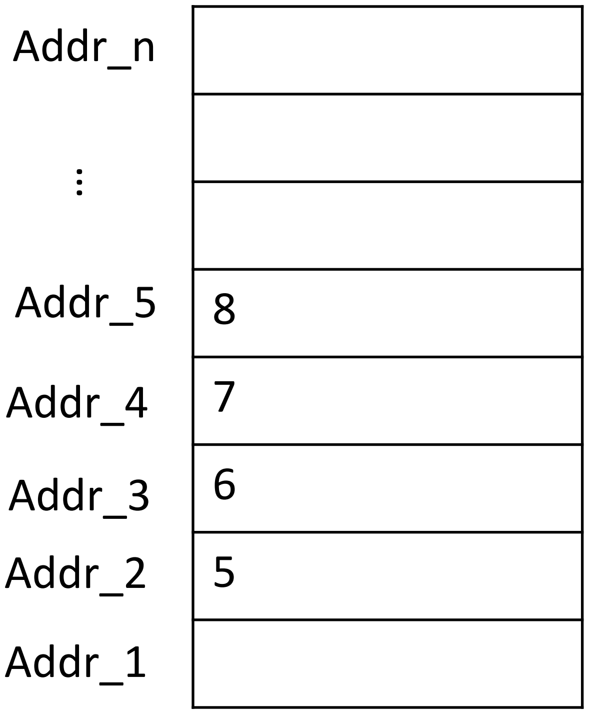
$$a = b + c$$

Assuming Addr_3 is store in X22 and Addr_2 is store in X23 Assembly code **(LEGv8 instruction)**

$$LDUR\ X19, [X22, \#\mathbf{0}]$$
$$LDUR\ X20, [X23, \#\mathbf{0}]$$
$$ADD\ X21, X19, X20$$

Review

Arrays in RAM



RAM

$$int\ a[4] = \{5, 6, 7, 8\};$$

1. Arrays are stored in contiguous memory

Let a start from Addr_2

a[0] ➔Addr_2

a[1] ➔ Addr_3

a[2] ➔Addr_4

a[3] ➔Addr_5

To load $a[1]$, we would have to specify where a starts in the memory, the offset (which is 1) and the destination register (d_register) to load it to.

Instruction

Load d_register, [addr_2, offset(1)]

A constant is needed to specify the offset to load arrays in the LDUR instruction

Review

Bits, Bytes, Words, and Double Words

For our course!!!

0 ➔ 1 **bit** of data

1 ➔ 1 **bit** of data

10011101 (8 bits) ➔ 1 **byte** of data

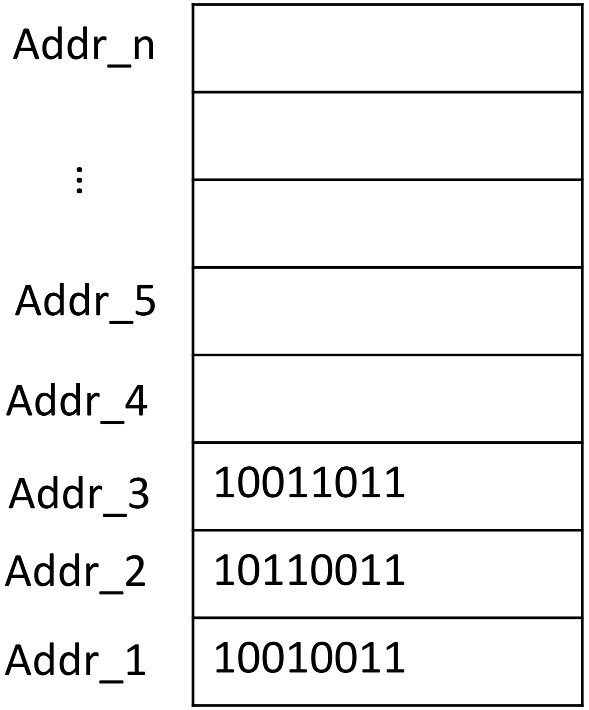
Overtime this became a **basic unit of data.**
Older system represented letters using bytes
As a results most memory hardware

10011101	10010001	10010101	10010101 ➔ **4 bytes** is a **word**
1 byte	**2 byte**	**3 byte**	**4 byte** (32 bits)

10011101	10010001	10010101 …	10010101 ➔ **8 bytes** is a **Doubleword**
1 byte	**2 byte**	**3 byte**	**7 byte** (64 bits)

Review

RAMS and Byte Addresses

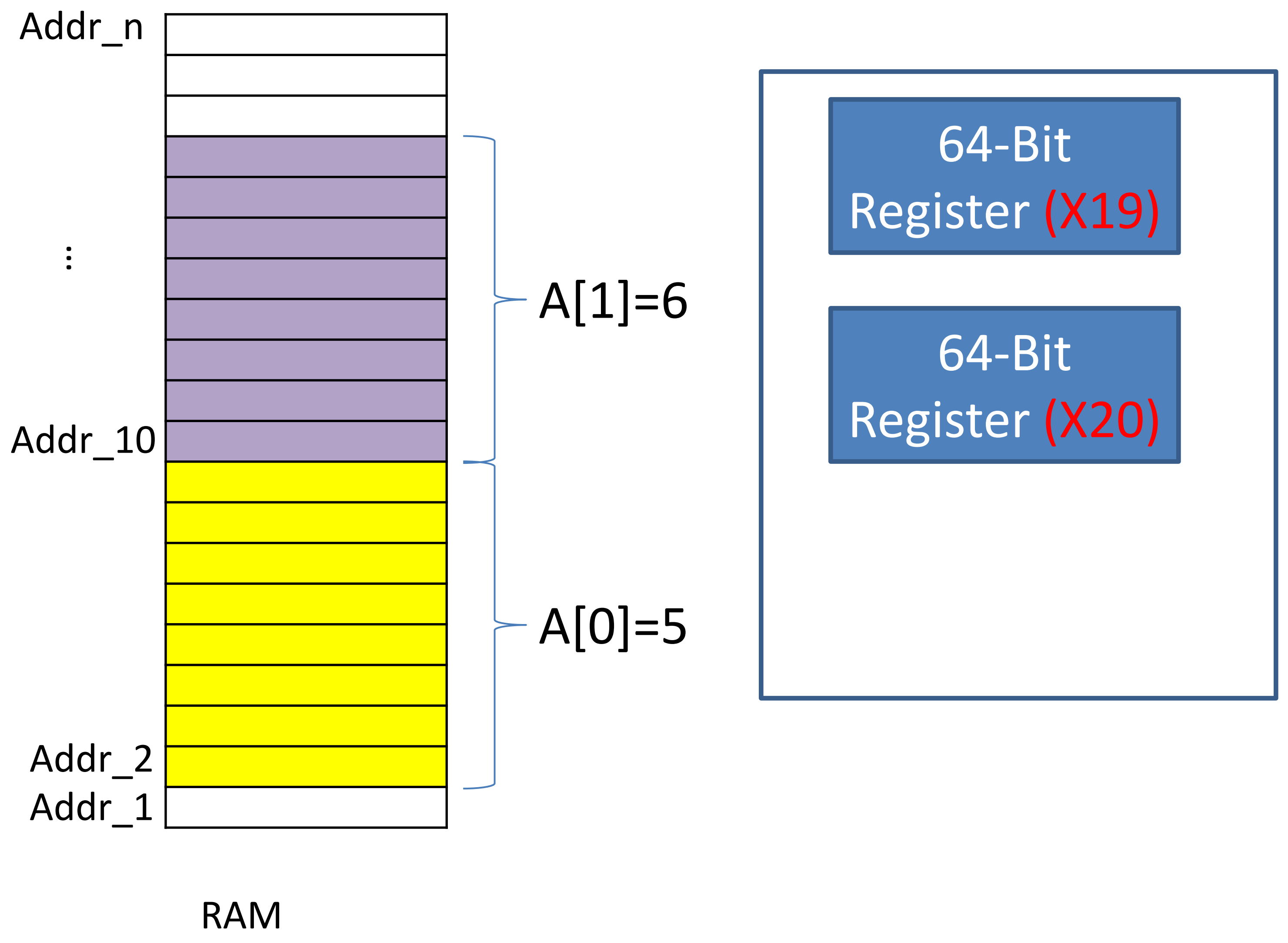


RAM

1. Byte is considered a basic unit of data.
2. Most memory hardware store 1 byte of data at each address.
3. Each address is referred to as a **byte address**, as 8 bits are stores.

Review

Memory Operand , LOAD



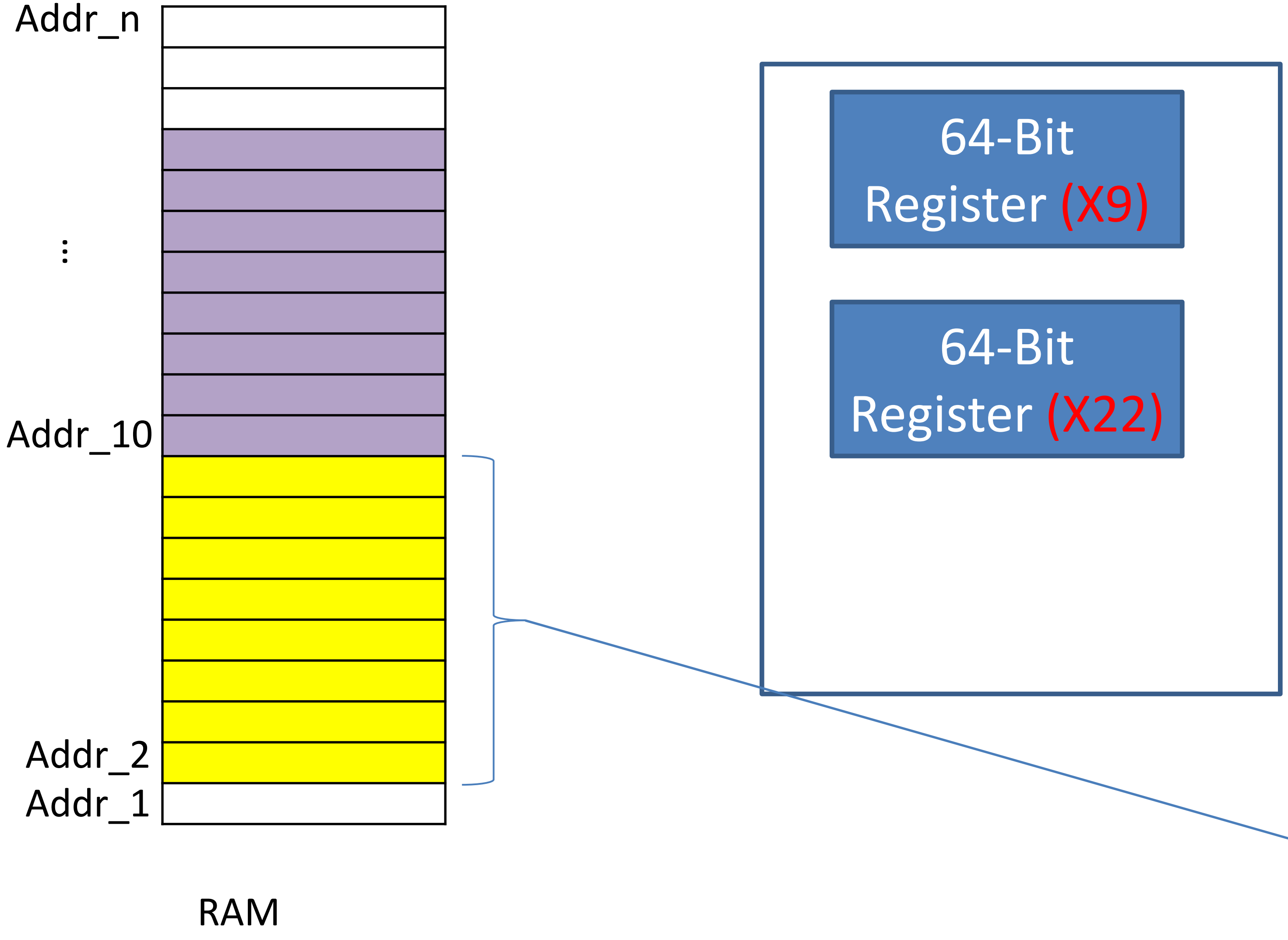
$$int\ a[4] = \{5, 6, 7, 8\};$$

Arrays are store in contiguous memory.
So
5 is stored using 64 bits
6 is stored using 64 bits

Let start address be Addr_2
Let Addr_2 be stored in register X22
What is the LEGv8 instruction to load a[0] into register X19?

Review

Memory Operand , LOAD



$int\ a[4] = \{5, 6, 7, 8\};$

Let start address be Addr_2
Let Addr_2 be stored in register X22
What is the LEGv8 instruction to load a[0] into register X9?

$LDUR\ X9, [X22, \#\mathbf{0}]$

What is the LEGv8 instruction to load a[1] into register X9?

$LDUR\ X9, [X22, \#8]$

What is the LEGv8 instruction to load a[3] into register X9?

$LDUR\ X9, [X22, \#\mathbf{24}]$

3 X 8 = 24

Review

Memory Operand Example

- C code:

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

  – h in X21, base address of A (i.e. A[0]) in X22

- LEGv8 code:

```
LDUR    X9,[X22,#64]
ADD     X9,X21,X9
STUR    X9,[X22,#96]
```

Constant or Immediate Operands

- Using a constant in operation.
- More than half of arithmetic instructions have constant (SPEC CPU2006).

x = x + 4

Let X be stored in register X22
If X20 is some base register, and the number 4 is stored in the memory at location AddrConst4

Constant or Immediate Operands

- Using a constant in operation.
- More than half of arithmetic instructions have constant (SPEC CPU).

x = x +4

Let X be stored in register X22
If X20 is some base register, and the number 4 is stored in the memory at location AddrConst4

Load 4 into register
Add

Constant or Immediate Operands

- Using a constant in operation.
- More than half of arithmetic instructions have constant (SPEC CPU2006).

x = x +4

Let X be stored in register X22
If X20 is some base register, and the number 4 is stored in the memory at location AddrConst4 (offset from X20)

LEGv8 Instructions:
$LDUR\ X9, [X20, \#AddrConst4]$
$ADD\ X22, X22, X9$

Very common to use constants.
Too much time to load constants from memory.
Why not make a version of Add with one operand fixed to a specific value.
One operand is always 4.

Constant or Immediate Operands

- Using a constant in operation.
- More than half of arithmetic instructions have constant (SPEC CPU2006).

x = x +4

Let X be stored in register X22
If X20 is some base register, and the number 4 is stored in the memory at location AddrConst4 (offset from X20)

LEGv8 Instructions:
$LDUR\ X9, [X20, \#AddrConst4]$
$ADD\ X22, X22, X9$

Very common to use constants.
Too much time to load constants from memory.
Why not make a version of Add with one operand fixed to a specific value.
One operand is always 4.

LEGv8 Instructions:
$\boldsymbol{ADDI}\ X22, X22, \#4$
Add immediate

Computer Architecture: Great Ideas

1. Use ***abstraction*** to simplify design

2. Make the **common case faster**

    1. Enhance performance than trying to optimize the rare case.
    2. Usually simpler and easier to enhance

COMMON CASE FAST

Constant or Immediate Operands

- Using a constant in operation.
- More than half of arithmetic instructions have constant (SPEC CPU2006).

Constant operands are common,
Immediate versions make it faster, and
use less energy.

x = x +4

Let X be stored in register X22
If X20 is some base register, and the number 4 is stored in the memory at location AddrConst4 (offset from X20)

LEGv8 Instructions:
$\boldsymbol{ADDI}\ X22, X22, \#4$

Add immediate

Number System

1. Decimal System
    1. Integers
    2. Fractions
    3. Positional number system
2. Binary representation
    1. Integers
    2. Fractions
    3. Addition
3. Conversion
    1. Binary to Decimal
    2. Decimal to Binary
4. Hexadecimal
5. Signed Integers
    1. 2's complement representation

Decimal Numbers (Integers)

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers → 10 numbers (**base** 10, also called as **radix**)
- Represented using a positional number system
- Example 4728_{ten}

Position	3	2	1	0
	4 (digit)	7	2	8

The decimal system is said to have a ***base***, or ***radix***, of 10. This means that each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position:

Decimal Numbers (Integers)

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers ➔ 10 numbers (**base** 10, also called as **radix**)
- Represented using a positional number system
- Example 4728_{ten}

Position	3	2	1	0
	4 (digit)	7	2	8

4728 = (4 * 10^3) +

Decimal Numbers (Integers)

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers → 10 numbers (**base** 10, also called as **radix**)
- Represented using a positional number system
- Example 4728_{ten}

Position	3	2	1	0
	4 (digit)	7	2	8

4728 = $(4 * 10^3) + (7 * 10^2) + (2 * 10^1) + (8 * 10^0)$

Generalizing, the value of i^{th} digit is $d \times \mathrm{Base}^i$

Decimal Numbers (Integers)

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers → 10 numbers (**base** 10, also called as **radix**)
- Represented using a positional number system
- Example 4728_{ten}

Position	3	2	1	0
	4 (digit)	7	2	8

4728 = (4 * 10^3) + (7 * 10^2) + (2 * 10^1) + (8 * 10^0)

= 4000 + 700 + 20 + 9

Generalizing, the value of i^{th} digit is $d \times \mathrm{Base}^i$

Decimal Numbers (Fractions)

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers ➔ 10 numbers (**base** 10, also called as **radix**)
- Represented using a positional number system
- The same principle holds for decimal fractions, but negative powers of 10 are used.
- Example decimal fraction $\mathbf{0.256}_{\boldsymbol{ten}}$ stands for

Decimal Numbers (Fractions)

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers → 10 numbers (**base** 10, also called as **radix**)
- Represented using a positional number system
- The same principle holds for decimal fractions, but negative powers of 10 are used.
- Example decimal fraction $\mathbf{0.256}_{ten}$ stands for

Position	-1	-2	-3
	2	5	6

Decimal Numbers (Fractions)

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers ➔ 10 numbers (**base** 10, also called as **radix**)
- Represented using a positional number system
- The same principle holds for decimal fractions, but negative powers of 10 are used.
- Example decimal fraction $\mathbf{0.256_{ten}}$ stands for

Position	-1	-2	-3
	2	5	6

$$\mathbf{0.256 = 2 * 10^{-1} + 5 * 10^{-2} + 6 * 10^{-3}}$$

Decimal Numbers

- A number with both an integer and fractional part has digits raised to both positive and negative powers of 10:

$$442.256 \\ = (4 * 10^2) + (4 + 10^1) + (2 * 10^0) + (2 * 10^{-1}) + (5 * 10^{-2}) + (6 * 10^{-3})$$

- ***Most significant digit***
  – The leftmost digit (carries the highest value)
- ***Least significant digit***
  – The rightmost digit

Positional Number Systems

- Each number is represented by a string of digits in which each digit position *i* has an associated weight r^i, where *r* is the *radix*, or *base,* of the number system.
- The general form of a number in such a system with radix *r* is

$$(\ldots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3} \ldots)_r$$

where the value of any digit a_i *is an integer in the range* $0 \leq a_i < r$.

The dot between a_0 *and* a_{-1} is called the ***radix point.***

Positional Interpretation of a Number in Base 7

32621.5_{seven}

Position	4	3	2	1	0	−1
Value in exponential form	7^4	7^3	7^2	7^1	7^0	7^{-1}
Decimal value	2401	343	49	7	1	1/7

Number System

1. Decimal System
    1. Integers
    2. Fractions
    3. Positional number system
2. Binary representation
    1. Integers
    2. Fractions
    3. Addition
3. Conversion
    1. Binary to Decimal
    2. Decimal to Binary
4. Hexadecimal
5. Signed Integers
    1. 2's complement representation

The Binary System (Integers)

- Base 2: Only two digits (0 and 1)
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_{two} = 0_{ten}$$

$$1_{two} = 1_{ten}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

Position	1	0
10_{two}	1	0

The Binary System (Integers)

- Base 2: Only two digits (0 and 1)
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_{two} = 0_{ten}$$

$$1_{two} = 1_{ten}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

Position	1	0
	1	0

$$10_{two} = (1 * 2^1) + (0 * 2^0) = 2_{ten}$$

The Binary System (Integers)

- Base 2: Only two digits (0 and 1)
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_{two} = 0_{ten}$$

$$1_{two} = 1_{ten}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

$$10_{two} = (1 * 2^1) + (0 * 2^0) = 2_{ten}$$

$$11_{two} = ?$$

The Binary System (Integers)

- Base 2: Only two digits (0 and 1)
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_{two} = 0_{ten}$$
$$1_{two} = 1_{ten}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

$$10_{two} = (1 * 2^1) + (0 * 2^0) = 2_{ten}$$
$$11_{two} = (1 * 2^1) + (1 * 2^0) = 3_{ten}$$
$$100_{two} = ?$$

The Binary System (Integers)

- Base 2: Only two digits (0 and 1)
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_{two} = 0_{ten}$$
$$1_{two} = 1_{ten}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

$$10_{two} = (1 * 2^1) + (0 * 2^0) = 2_{ten}$$
$$11_{two} = (1 * 2^1) + (1 * 2^0) = 3_{ten}$$
$$100_{two} = (1 * 2^2) + (0 * 2^1) + (0 * 2^0) = 4_{1ten}$$

The Binary System (Fractions)

- Base 2: Only two digits (0 and 1)
- Fractional values are represented with negative powers of the radix:

$$1001.101_{two}$$

The Binary System (Fractions)

- Base 2: Only two digits (0 and 1)
- Fractional values are represented with negative powers of the radix:

$$1001.101_{two} = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625_{10}$$

Number System

1. Decimal System
   1. Integers
   2. Fractions
   3. Positional number system
2. Binary representation
   1. Integers
   2. Fractions
   3. Addition
3. Conversion
   1. Binary to Decimal
   2. Decimal to Binary
4. Hexadecimal
5. Signed Integers
   1. 2’s complement representation

Conversion Base 10 → Base 3

- Division-remainder method
- Convert 104 to base 3

Conversion Base 10 → Base 3

- Division-remainder method
- Convert 104 to base 3

$$\frac{104}{\mathbf{3}}$$

Conversion Base 10 → Base 3

- Division-remainder method
- Convert 104 to base 3

$$\frac{104}{\mathbf{3}} = 34 \Rightarrow remainder\ 2$$

Conversion Base 10 ➔ Base 3

- Division-remainder method
- Convert 104 to base 3

$$\frac{104}{\mathbf{3}} = 34 \Rightarrow remainder\ 2$$

$$\frac{34}{\mathbf{3}} =$$

Conversion Base 10 → Base 3

- Division-remainder method
- Convert 104 to base 3

$$\frac{104}{\mathbf{3}} = 34 \Rightarrow remainder\ 2$$

$$\frac{34}{\mathbf{3}} = 11\ remainder\ 1$$

$$\frac{11}{\mathbf{3}}$$

Conversion Base 10 ➔ Base 3

- Division-remainder method
- Convert 104 to base 3

$$\frac{104}{\mathbf{3}} = 34 \Rightarrow remainder\ 2$$

$$\frac{34}{\mathbf{3}} = 11\ remainder\ 1$$

$$\frac{11}{3} = 3\quad remainder\ 2$$

$$\frac{3}{3} = 1\quad remainder\ 0$$

$$\frac{1}{3} = 0\quad remainder\ 1$$

1	0	2	1	2

Conversion Base 10 → Base 3

- Division-remainder method
- Convert 104 to base 3

$$\frac{104}{\mathbf{3}} = 34 \Rightarrow remainder\ 2$$

$$\frac{34}{\mathbf{3}} = 11\ remainder\ 1$$

$$\frac{11}{3} = 3\ \ remainder\ 2$$

$$\frac{3}{3} = 1\ \ remainder\ 0$$

$$\frac{1}{3} = 0\ \ remainder\ 1$$

1	0	2	1	2

$$10212_{three} = 104_{10}$$

Conversion Base 3 → Base 10

- $10212_3 = 1 * 3^4 + 0 * 3^3 + 2 * 3^2 + 1 * 3^1 + 2 * 3^0$

  $= 1 * 81 + 0 + 2 * 9 + 1 * 3 + 2 * 1$

  $= 81 + 18 + 3 + 2 = 104_{10}$

Conversion Base 10 → Binary

Convert 11_{10} to Binary notation:

Conversion Base 10 → Binary

Convert 11_{10} to Binary notation:

11/2	=	5 →	Remainder 1
5/2	=	2 →	Remainder 1
2/2	=	1 →	Remainder 0
1/2	=	0 →	Remainder 1

Thus $11_{10} = 1011_2$

Conversion Base 10 ➔ Binary

Convert 0.375_{10} to Binary notation:

$0.375 \times 2 = 0.75\ (0.75 + 0)$

Thus $0.375_{10} = 0.$

Conversion Base 10 → Binary

Convert 0.375_{10} to Binary notation:

$0.375 \times 2 = 0.75$ ($0.75 + \mathbf{0}$)

Thus $0.375_{10} = 0.\mathbf{0}$

Conversion Base 10 ➔ Binary

Convert 0.375_{10} to Binary notation:

$0.375 \times 2 = 0.75$ ($0.75 +$ **0**)

0.75×2

Thus $0.375_{10} = 0.$**01**

Conversion Base 10 ➔ Binary

Convert 0.375_{10} to Binary notation:

$0.375 \times 2 = 0.75\ (0.75 + \mathbf{0})$

$0.75 \times 2 = 1.50\ (0.50 + \mathbf{1})$

Thus $0.375_{10} = 0.\mathbf{01}$

Conversion Base 10 ➔ Binary

Convert 0.375_{10} to Binary notation:

$0.375 \times 2 = 0.75\ (0.75 + 0)$
$0.75 \times 2 = 1.50\ (0.50 + 1)$
$0.50 \times 2 = 1.00\ (\mathbf{0.00} + 1)$

Thus $0.375_{10} = 0.011_2$

Note: not every decimal number can be represented in binary with a finite number of digits

Conversion Base 10 → Binary

Convert 0.81_{10} to Binary notation:

Conversion Base 10 ➔ Binary

Convert 0.81_{10} to Binary notation:

$0.81 \times 2 = 1.62$ $(0.62 + 1)$
$0.62 \times 2 = 1.24$ $(0.24 + 1)$
$0.24 \times 2 = 0.48$ $(0.48 + 0)$
$0.48 \times 2 = 0.96$ $(0.96 + 0)$
$0.96 \times 2 = 1.92$ $(0.92 + 1)$
and so on

Thus $0.81_{10} = 0.11001_2$

Note: not every decimal number can be represented in binary with a finite number of digits

Pay attention to the bit significance

Convert 0.375_{10} to Binary notation:

$0.375 \times 2 = 0.75\ (0.75 + 0)$

$0.75 \times 2 = 1.50\ (0.50 + 1)$

$0.50 \times 2 = 1.00\ (\mathbf{0.00} + 1)$

$0.375_{10} = 0.011_2$

Convert 11_{10} to Binary notation:

11/2	=	5 →	Remainder 1
5/2	=	2 →	Remainder 1
2/2	=	1 →	Remainder 0
1/2	=	0 →	Remainder 1

Thus $11_{10} = 1011_2$

Number System

1. Decimal System
   1. Integers
   2. Fractions
   3. Positional number system
2. Binary representation
   1. Integers
   2. Fractions
   3. Addition
3. Conversion
   1. Binary to Decimal
   2. Decimal to Binary
4. Hexadecimal
5. Signed Integers
   1. 2’s complement representation

Hexadecimal Notation

- Easier to represent long binary data, by grouping them
- Binary digits are grouped into sets of four bits, called a *nibble*
- *Double word → 64 bits, can be represented using 16 hexadecimal characters*

Hexadecimal Notation

- Easier to represent long binary data
- Binary digits are grouped into sets of four bits, called a *nibble*
- Each possible combination of four binary digits is given a symbol, as follows:

0000 = 0	0100 = 4	1000 = 8	1100 = C (12_{ten})
0001 = 1	0101 = 5	1001 = 9	1101 = D (13_{ten})
0010 = 2	0110 = 6	1010 = A (10_{ten})	1110 = E (14_{ten})
0011 = 3	0111 = 7	1011 = B (11_{ten})	1111 = F (15_{ten})

- Because 16 symbols are used, the notation is called *hexadecimal* and the 16 symbols are the *hexadecimal digits*

Hexadecimal Notation

- Easier to represent long binary data
- Binary digits are grouped into sets of four bits, called a *nibble*
- Each possible combination of four binary digits is given a symbol, as follows:

0000 = 0	0100 = 4	1000 = 8	1100 = C (12_{ten})
0001 = 1	0101 = 5	1001 = 9	1101 = D (13_{ten})
0010 = 2	0110 = 6	1010 = A (10_{ten})	1110 = E (14_{ten})
0011 = 3	0111 = 7	1011 = B (11_{ten})	1111 = F (15_{ten})

- Because 16 symbols are used, the notation is called *hexadecimal* and the 16 symbols are the *hexadecimal digits*
- Thus

$$2C_{16} = (2_{16} * 16^1) + (C_{16} * 16^0)$$
$$= (2_{10} * 16^1) + (12_{10} * 16^0) = 44$$

Decimal vs. Binary vs. Hexadecimal

- Hexadecimal is more compact than binary or decimal
- Binary data occupies typically multiple of 4 bits, and hence some multiple of a single hexadecimal digit
- Easy to convert between binary and hexadecimal notation

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
15	1110	E
16	1111	F
17	0001 0000	10
18	0001 0001	11
19	0001 0010	12
31	0001 1111	1F
100	0110 0100	64
255	1111 1111	FF

Binary Arithmetic: Additions

- Similar to Decimal System

0011010 + 001100

Number System

1. Decimal System
    1. Integers
    2. Fractions
    3. Positional number system
2. Binary representation
    1. Integers
    2. Fractions
    3. Addition
3. Conversion
    1. Binary to Decimal
    2. Decimal to Binary
4. Hexadecimal
5. Signed Integers
    1. 2’s complement representation

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

$$\begin{array}{rl} 0\,0\,1\,1\,0\,1\,0 & = 26_{10} \\ +\,0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\ \hline \end{array}$$

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

$$
\begin{array}{rl}
0\,0\,1\,1\,0\,1\,0 & = 26_{10} \\
+\,0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\
\hline
0 &
\end{array}
$$

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

$$
\begin{array}{rl}
0\,0\,1\,1\,0\,1\,0 & = 26_{10} \\
+\,0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\
\hline
1\,0 &
\end{array}
$$

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

$$
\begin{array}{rl}
0\,0\,1\,1\,0\,1\,0 & = 26_{10} \\
+\,0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\
\hline
1\,1\,0 &
\end{array}
$$

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

	1	carry
	0 0 1 1 0 1 0	$= 26_{10}$
+	0 0 0 1 1 0 0	$= 12_{10}$
	0 1 1 0	

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

1 1	carry
0 0 1 1 0 1 0	$= 26_{10}$
+ 0 0 0 1 1 0 0	$= 12_{10}$
0 0 1 1 0	

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

1 1	carry
0 0 1 1 0 1 0	$= 26_{10}$
+ 0 0 0 1 1 0 0	$= 12_{10}$
0 1 0 0 1 1 0	

Binary Arithmetic: Additions

- Similar to decimal system

0011010 + 001100

26_{10} 12_{10}

$$
\begin{array}{rl}
1\,1 & \text{carry} \\
0\,0\,1\,1\,0\,1\,0 & = 26_{10} \\
+\,0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\
\hline
0\,1\,0\,0\,1\,1\,0 & = 38_{10}
\end{array}
$$

https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Binary Arithmetic: Subtraction

- Similar to decimal system

0011010 - 001100

https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Binary Arithmetic: Subtraction

- Similar to decimal system

0011010 - 001100

$$
\begin{array}{rl}
0\,0\,1\,1\,0\,1\,0 & = 26_{10} \\
-\,0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\
\hline
\end{array}
$$

https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Binary Arithmetic: Subtraction

- Similar to decimal system

0011010 - 001100

$$
\begin{array}{rl}
0011010 & = 26_{10} \\
-0001100 & = 12_{10} \\
\hline
10 &
\end{array}
$$

https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Binary Arithmetic: Subtraction

- Similar to decimal system

0011010 - 001100

$$\begin{array}{rl}
1 & \text{borrow} \\
0\,0\,1\,\cancel{1}\,0\,1\,0 & = 26_{10} \\
-\,0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\
\hline
1\,0 &
\end{array}$$

https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Binary Arithmetic: Subtraction

- Similar to decimal system

0011010 - 001100

$$
\begin{array}{r l}
1 & \text{borrow} \\
0\,0\,1\,\cancel{1}\,0\,1\,0 & = 26_{10} \\
-\,0\,0\,0\,1\,1\,0\,0 & = 12_{10} \\
\hline
1\,1\,0 &
\end{array}
$$

https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Binary Arithmetic: Subtraction

- Similar to decimal system

0011010 - 001100

```
     1 1       borrow
 0 0 1 1 0 1 0  = 26₁₀
-0 0 0 1 1 0 0  = 12₁₀
---------------
       1 1 1 0
```


https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Binary Arithmetic: Subtraction

- Similar to decimal system

0011010 - 001100 = 00001110

1 1	borrow
0 0 1 1 0 1 0	$= 26_{10}$
- 0 0 0 1 1 0 0	$= 12_{10}$
0 0 0 1 1 1 0	$= 14_{10}$

https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Binary Arithmetic: Multiplication

- Similar to decimal system

Example:

0011010 x 001100 = 100111000

```
      0 0 1 1 0 1 0   = 26₁₀
    x 0 0 0 1 1 0 0   = 12₁₀
  -------------------
      0 0 0 0 0 0 0
    0 0 0 0 0 0 0
  0 0 1 1 0 1 0
0 0 1 1 0 1 0
  -------------------
0 1 0 0 1 1 1 0 0 0   = 312₁₀
```


https://www.tutorialspoint.com/computer_logical_organization/binary_arithmetic.htm

Signed integers

- We have been dealing so far with unsigned integers

Slide based on a lecture at: http://people.sju.edu/~ggrevera/arch/slides/binary-arithmetic.ppt

Signed integers

- We have been dealing so far with unsigned integers

- Multiple ways for representing signed integers:
  1. Sign and magnitude
  2. 2's complement

Slide based on a lecture at: http://people.sju.edu/~ggrevera/arch/slides/binary-arithmetic.ppt

Sign and Magnitude

- Lets consider three bits to represent numbers.
- Number from 0 – 7 can be represented

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Sign and Magnitude

- Lets consider three bits to represent numbers.
- Number from 0 – 7 can be represented
- Sign and magnitude: Uses **one additional bit** to represent positive/negative, called sign bit.
- 0 → positive number
- 1 → negative numbers

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Sign and Magnitude

- Lets consider three bits to represent numbers.
- Number from 0 – 7 can be represented
- Sign and magnitude: Uses one additional bit to represent positive/negative, called sign bit.
- 0 → positive number
- 1 → negative numbers

0	000	+0
0	001	+1
0	010	+2
0	011	+3
0	100	+4
0	101	+5
0	110	+6
0	111	+7

1	000	-0
1	001	-1
1	010	-2
1	011	-3
1	100	-4
1	101	-5
1	110	-6
1	111	-7

Sign and Magnitude

- Lets consider three bits to represent numbers.
- Number from 0 – 7 can be represented
- Sign and magnitude: Uses one additional bit to represent positive/negative, called sign bit.
- 0 ➔ positive number
- 1 ➔ negative numbers

0	000	+0
0	001	+1
0	010	+2
0	011	+3
0	100	+4
0	101	+5
0	110	+6
0	111	+7

1	000	-0
1	001	-1
1	010	-2
1	011	-3
1	100	-4
1	101	-5
1	110	-6
1	111	-7

Sign and Magnitude

- Lets consider three bits to represent numbers.
- Number from 0 – 7 can be represented
- Sign and magnitude: Uses one additional bit to represent positive/negative, called sign bit.
- 0 ➔ positive number
- 1 ➔ negative numbers

0	000	+0
0	001	+1
0	010	+2
0	011	+3
0	100	+4
0	101	+5
0	110	+6
0	111	+7

1	000	-0
1	001	-1
1	010	-2
1	011	-3
1	100	-4
1	101	-5
1	110	-6
1	111	-7

Sign and Magnitude

- Sign and magnitude: Uses one additional bit to represent positive/negative, called sign bit.
- Shortcomings:
  - Where to put the sign bit (left/right)
  - Adders may need extra step to set the sign bit

0	000	+0
0	001	+1
0	010	+2
0	011	+3
0	100	+4
0	101	+5
0	110	+6
0	111	+7

1	000	-0
1	001	-1
1	010	-2
1	011	-3
1	100	-4
1	101	-5
1	110	-6
1	111	-7

Addition w/ signed magnitude algorithm

- For A - B, change the sign of B and perform addition of A + (-B) (as in the next step)
- For A + B:

```
if  (A_sign==B_sign) {
        R = |A| + |B|;    R_sign = A_sign;   }
else if (|A|>|B|) {
        R = |A| - |B|;     R_sign = A_sign; }
else if (|A|==|B|) {
        R = 0;             R_sign = 0;      }
else  {
        R = |B| - |A|;     R_sign = B_sign; }
```

- Complicated???

Sign and Magnitude

- Sign and magnitude: Uses one additional bit to represent positive/negative, called sign bit.
- Shortcomings:
  - Where to put the sign bit (left/right)
  - Adders may need extra step to set the sign bit
  - Both a positive and negative zero

0	000	+0
0	001	+1
0	010	+2
0	011	+3
0	100	+4
0	101	+5
0	110	+6
0	111	+7

1	000	-0
1	001	-1
1	010	-2
1	011	-3
1	100	-4
1	101	-5
1	110	-6
1	111	-7

2’s Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half → Positive
  - Second half → Negative

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

2’s Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2’s Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)

Most negative number

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)
  - Range -8, -7 … 6, 7

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Most negative number

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)
- Many advantages:
  - Leading 0 ➔ Positive, Leading 1 ➔ Negative

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)
- Many advantages:
  - Leading 0 ➔ Positive, Leading 1 ➔ Negative
  - Test only one bit to check positive/negative

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)
- Many advantages:
  - Leading 0 ➔ Positive, Leading 1 ➔ Negative
  - Test only one bit to check positive/negative
  - Made hardware implementation simple

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2's Complement

- Conversion to decimal is straight forward

1	0	1	1
	2^2	2^1	2^0

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2's Complement

- Conversion to decimal is straight forward

1	0	1	1
$-\mathbf{2^3}$	2^2	2^1	2^0

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2's Complement

- Conversion to decimal is straight forward

1	0	1	1
-2^3	2^2	2^1	2^0

$$= 1X\left(-2^3\right) + 0X\left(2^2\right) + 1\,X\,(2^1) + 1\,X\,(2^0)$$
$$= -8 + 0 + 2 + 1 = -5$$

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2's Complement

- Conversion to decimal is straight forward

0	0	1	1
-2^3	2^2	2^1	2^0

$$= 0X\left(-2^3\right) + 0X\left(2^2\right) + 1\,X\,(2^1) + 1\,X\,(2^0)$$
$$= 0 + 0 + 2 + 1 = 3$$

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Shortcut to Negate

- Determine the binary value of -27 in 2's complement representation using 8 bits

+27 in binary is: 0001 1011

Shortcut to Negate

- Determine the binary value of -27 in 2's complement representation using 8 bits

+27 in binary is:	0001 1011
Bitwise complement:	1110 0100

Shortcut to Negate

- Determine the binary value of -27 in 2's complement representation using 8 bits

+27 in binary is: 0001 1011

Bitwise complement: 1110 0100

Add 1: + 1

Shortcut to Negate

- Determine the binary value of -27 in 2’s complement representation using 8 bits

+27 in binary is: 0001 1011

Bitwise complement: 1110 0100
Add 1: + 1

1110 0101

2’s complement for -27

Shortcut to Negate

- Determine the binary value of -27 in 2's complement representation using 8 bits

+27 in binary is: 0001 1011

Bitwise complement: 1110 0100

Add 1: + 1

1110 0101

2's complement for -27

Verify:

$$= 1\ X\ (-2^7) + 1X2^6 + 1\ X\ 2^5 + 0\ X2^4 + 0\ X\ 2^3 + 1\ X\ 2^2 + 0\ X\ 2^1 + 1X\ 2^0$$

$$= -128 + 64 + 32 + 0 + 0 + 4 + 0 + 1$$

$$= -27$$

Shortcut to Negate

Negate -7

-7= 1001

Bitwise complement: 0110

Add 1: + 1

0111

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Range Extension

- Range of numbers that can be expressed is extended by increasing the bit length
- Sign extension shortcut
  - Rule is to move the sign bit to the new leftmost position and fill in with copies of the sign bit
  - For positive numbers, fill in with zeros, and for negative numbers, fill in with ones
  - This is called *sign extension*

Sign Extension

- Example: show the representation of +4 and -4 for 4 bits and 8 bits

+4: 0100

0000 0100

-4: 1100

1111 1100