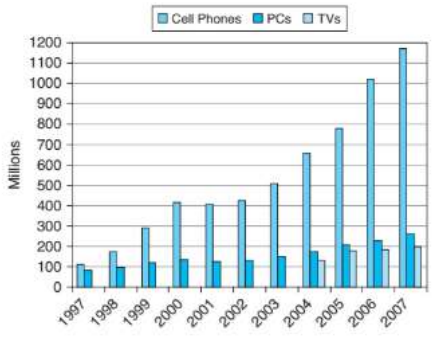
Almost 10% of the gross national product of the United States

Moore's law

- Moore predicted that the number of transistors that can be integrated on a die would grow exponentially with time.
- Amazingly visionary – million transistor/chip barrier was crossed in the 1980's.
- 16 M transistors (Ultra Sparc III)
- 140 M transistor (HP PA-8500)
- 1.7B transistor (Intel Montecito)

5 Commendments

- Moore's Law : The number of transistors on a chip doubles annually
- Rock's Law : The cost of semiconductor tools doubles every four years
- Machrone's Law: The PC you want to buy will always be $5000
- Metcalfe's Law : A network's value grows proportionately to the number of its users squared



What you will learn in this chapter

1.How to convert high level language into assembly language as all the program we are writing is high level language. Means the language that is not directly understandable by computer.

2. What is the interface between the software and the hardware, and how does software instruct the hardware to perform needed functions?

3.How the programmers can improve the efficiency of the program.

4.What technique can be used by hardware designers to improve performance.

5. What are the reasons for and the consequences of the recent switch from sequential processing to parallel processing?

Algorithm	Determines both the number of source-level statements and the number of I/O operations executed
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement
Processor and memory system	Determines how fast instructions can be executed
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed

the software and hardware affect the performance of a program.

1.The algorithm chosen: -This is what you are seeing and one of the most emerging areas. suppose you want to sort large set of data set and you are using bubble sort instead of quick sort or merge sort then it might be possible you will not get the result or TLE will come.

2. The Programming Language or Compiler: Every programming language is designed for different purposes like Opp's work can be best done in Java or c++ instead of other. Ex: -using a dynamically typed language like Python for heavy numerical computations is inefficient in doing this.

3.The operating system: - If your application relies heavily on system calls or interacts frequently with the file system, and the operating system is unable to efficiently manage these system calls or disk I/O operations, the OS can become a bottleneck.

Hardware: -

4.The processor: -If your application required high amount of parallelism and you are using processor with limited numbers of cores or outdated architecture then it will be bottling neck.

5.The I/O System and Devices: -Example: If you have an application that relies heavily on reading and writing data from/to disk, and you're using slow or overloaded storage devices

★ Hardware and software interaction

Application code written by user may contain thousands - millions of line while hardware understand and operate on simple lines of code.

Q How Hardware then able to interact with application software

Ans: System software is one which provide bridge inbetween.

Application software ⟷ System software ⟷ Hardware

There are many system software like compiler & OS.

Example of operating system:- Linux, MacOS, Window, ubuntu ---

work of operating system

1. Handling I/o operation.
2. Allocation of memory & storage
3. providing the protection of computer during various application.

   It means when multiple application is going at same time then each will have independly accen to computer and other can't take data of another application.

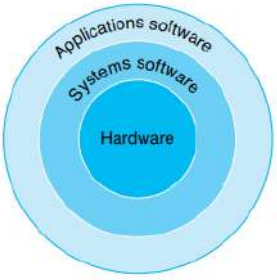
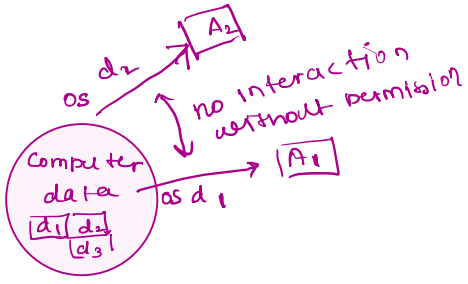


FIGURE 1.2 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost. In complex applications, there are often multiple layers of application software as well. For example, a database system may run on top of the systems software hosting an application, which in turn runs on top of the database.



From a High-Level Language to the Language of Hardware

Computers are slaves to our commands, which are called instructions. letters. Just as the 26 letters of the English alphabet do not limit how much can be written, the two letters of the computer alphabet do not limit what computers can do.

You can also directly write assembly language instead of high-level language. Why we not use?

Ans: - First thing is high level language are easy to write while writing assembly language is a tedious work. Also for each architecture you have to write different assembly code. suppose you are writing code on one system and want to open on another system then you must change & changing is tedious while when you use high level language then the complier already takes care of that. They also allow the programmer to think in a more natural language, using English words and algebraic notation, resulting in programs that look much more like text than like tables of cryptic symbols. Conciseness is a clear advantage of high-level languages over assembly language means require fewer line of code to express idea.

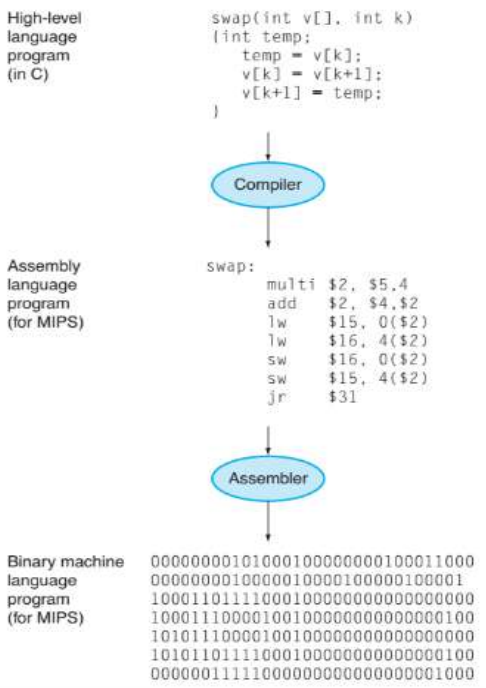
Example of high-level language:-

Fortran was designed for scientific computation.

Cobol for business data processing.

Lisp for symbol manipulation.

Word: -- Using the computer to help program the computer, the pioneers invented programs to translate from symbolic notation to binary.



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

Assembly language: -A symbolic representation of machine instructions.

Machine language:- A binary representation of machine instructions.

Instruction :-A command that computer hardware understands and obeys.

Assembler:- A program that translates a symbolic version of instructions into the binary version.

High-level programming language:- A portable language such as C, C++, Java, or Visual Basic that is composed of words and algebraic notation that can be translated by a compiler into assembly language.

C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly.

Understanding the MIPS (microprocessor without interlocked pipeline stages) mips is also the name of company name which design this.

$ → used for register

swap:

① multi $2, $5, 4 Take the value from register '$5' and multiply with 4 and store into register $2.

② add $2, $4, $2 take the value from register '$4' and add the value at register $2, store into register $2

③ lw $15, 0($2) load the word on the register $2 + 0 and store to $15.

④ sw $15, 4($2) store the value in $16 to memory at address in $2 + 4.

⑤ jr $31 Jump to address stored in $31 (which is often use to return address in subroutin cell. This effectively return from the swap subroutine

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. Figure 1.4 shows the standard organization of a computer. This organization is independent of hardware technology: you can place every piece of every computer, past and present, into one of these five categories. To help you keep all this in perspective, the five components of a computer are shown on the front page of each of the following chapters, with the portion of interest to that chapter highlighted.

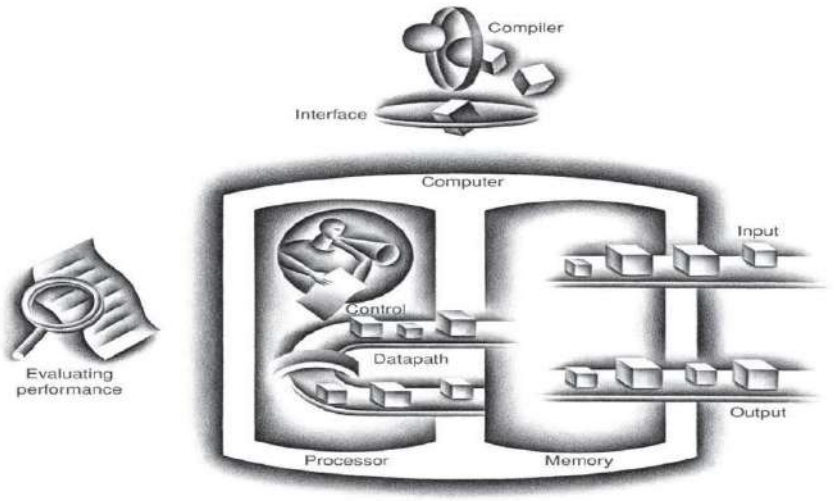


FIGURE 1.4 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

What we are feeding to the computer is input and what computer gives is output.

Some devices such as networks and disk provide both input as well as output.

Disk drive: -input: -we are taking the data from already stored from disk drive to computer.

Output: -we are saving some files from computer to Disk drive.

Network: Input- when it receives data from remote server or another device via network connection

Output: -sending data from your computer to another device or remote server like cloud or Instagram.

Fact:-Initially we used to use electromechanical mouse now we are using electro-optical mouse .

Through the looking glass

Liquid crystal display: - A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.

active-matrix display: - A liquid crystal display using a transistor to control the transmission of light at each individual pixel.

Pixel: -The smallest individual picture element. Screens are composed of hundreds of thousands to millions of pixels, organized in a matrix.

How LED works

LED is not a light sourcing element. It is what that controls transmission of the light coming from back or less often from reflected light .It include rod shaped molecule in al liquid that form a twisting helix that bend the light. The rid straighten out when current is applied and no longer bend the light. Since the liquid crystal material is between two screens polarized at 90 degrees, the light cannot pass through unless it is bent.

Active matrix :-that has a tiny transistor switch at each pixel to precisely control current and make sharper images. A red-green-blue mask associated with each dot on the display determines the intensity of the three colour components in the final image; in a colour active matrix LCD, there are three transistor switches at each point.

Pixels:- which can be represented as a matrix of bits, called a bit map. Depending on the size of the screen and the resolution, the display matrix ranges in size from 640 × 480 to 2560 × 1600 pixels in 2008. A colour display might use 8 bits for each of the three colours (red, blue, and green), for 24 bits per pixel, permitting millions of different colours to be displayed. Amount of pixel = 2^24 . Approx 2 million

The computer hardware support for graphics consists mainly of a *raster refresh buffer*, or *frame buffer*, to store the bit map. The image to be represented onscreen is stored in the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate. Figure 1.6 shows a frame buffer with a simplified design of just 4 bits per pixel.

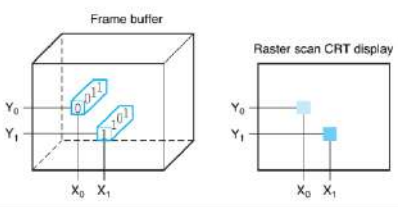
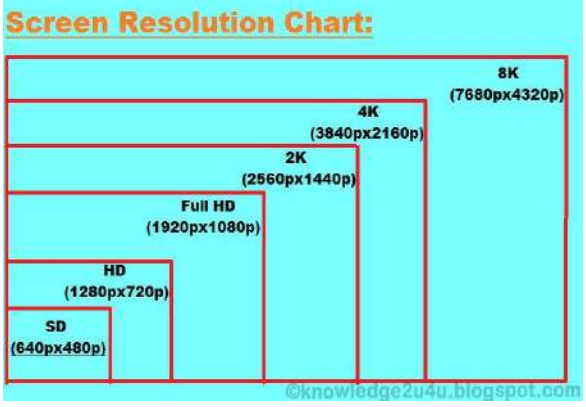


FIGURE 1.6 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade on the screen than the bit pattern 1101 in pixel (X_1, Y_1).



Something more about pixel:

My screen is full HD means 1920 x 1080 p. If you open the YouTube it will show as 1080 p not full .so it is understandable that it is talking about 1920 x 1080 p resolution.

More the number of pixels clearer image will be there. In static image number of pixels on screen will be

= 1920 X 1080 =2073600 i.e., 2 million pixel. but in video at least 30 frames should be there to get good video qualitu 30 frames means in one second 30 times static image will change to show motion.

Total pixel in a second = 30 * 2 million = 60 million

Note:- Higer the frame and resolution higher will no of pixels.

Difference between refresh rate and number of frames in second

The refresh rate of a monitor dictates how many times it can display frames in a second, but the actual number of frames sent to the monitor (FPS) depends on the capabilities of the GPU and the software being run. In many cases, if the FPS is lower than the monitor's refresh rate, you may not see the full benefit of the higher refresh rate as the monitor will display duplicate frames until a new frame is available from the GPU.

Opening the box

motherboard A plastic board containing packages of integrated circuits or chips, including processor, cache, memory, and connectors for I/O devices such as networks and disks.

integrated circuit Also called a chip. A device combining dozens to millions of transistors.

memory The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

dynamic random-access memory (DRAM) Memory built as an integrated circuit; it provides random access to any location. the RAM portion of the term DRAM means that memory accesses take basically the same amount of time no matter what portion of the memory is read. As is reading is random.

Dual inline memory module (DIMM) A small board that contains DRAM chips on both sides. (SIMMs have DRAMs on only one side.)SIMM means single inline memory module.

central processor unit (CPU) Also called processor. The active part of the computer, which contains the Datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

The processor comprises two main component.

1.Datapath 2.Control

Respective are known as brawn(muscle) and brain of processor.

Datapath The component of the processor that performs arithmetic operations.

control the component of the processor that commands the Datapath, memory, and I/O devices according to the instructions of the program.

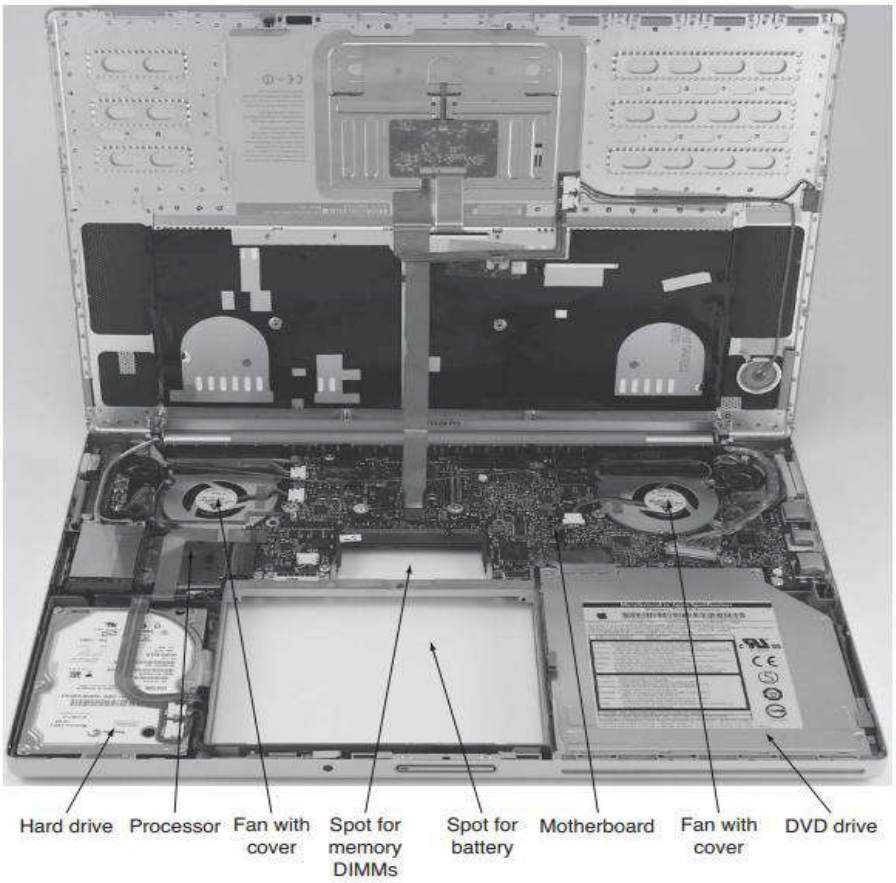


FIGURE 1.7 Inside the laptop computer of Figure 1.5. The shiny box with the white label on the lower left is a 100 GB SATA hard disk drive, and the shiny metal box on the lower right side is the DVD drive. The hole between them is where the laptop battery would be located. The small hole above the battery hole is for memory DIMMs. Figure 1.8 is a close-up of the DIMMs, which are inserted from the bottom in this laptop. Above the battery hole and DVD drive is a printed circuit board (PC board), called the *motherboard*, which contains most of the electronics of the computer. The two shiny circles in the upper half of the picture are two fans with covers. The processor is the large raised rectangle just below the left fan. Photo courtesy of OtherWorldComputing.com.

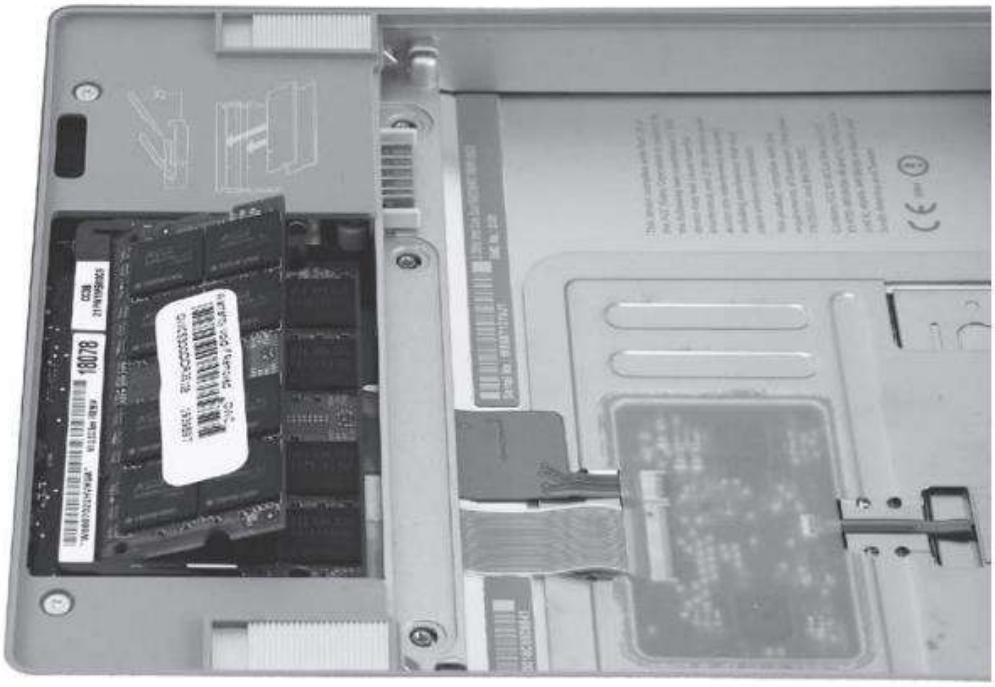


FIGURE 1.8 Close-up of the bottom of the laptop reveals the memory. The main memory is contained on one or more small boards shown on the left. The hole for the battery is to the right. The DRAM chips are mounted on these boards (called DIMMs, for dual inline memory modules) and then plugged into the connectors. Photo courtesy of OtherWorldComputing.com.

cache memory A small, fast memory that acts as a buffer for a slower, larger memory.

static random-access memory (SRAM) Also memory built as an integrated circuit, but faster and less dense than DRAM.

Abstraction A model that renders lower-level details of computer systems temporarily invisible to facilitate design of sophisticated systems. This we have also seen oops. The person who has designing the software has nothing to do with how semiconductor implanted works or other way round also. This is like all level of development are unrevealed from other and have no need to be known.

Why silicon for chips?

Silicon is a semiconductor that is sweet spot between insulator and conductor. Thus, conductivity can be altered by

Adding impurity called doping to meet the electronic need.

This is also the semiconductor present in abundant thus used on larger scale.

The top five semiconductor chip companies in the world by revenue in 2021 are.

1.Samsung Electronics,

2.Intel Corporation,

3. Taiwan Semiconductor Manufacturing Company (TSMC),

4. SK Hynix 5. Micron Technology

Why is wafer manufactured circular not square or rectangular even though the loss of chips at the corner will be more in comparison to other?

Ans: -Manufacturing and etching: - As manufacturing are done by robots and due to symmetrical shape and radius it is easier and more convenient to manufacture and etching.

Working as a mask: - Even though the chips at corner are partially circular means it has no use due to partially manufactured and uneven shape that will not fit into designed place. But these are act as mask for robot and help in patterning.

Accumulation of impurity at the corner: -rectangular and square chips due to corner have high chances of impurity allocation.

Historical aspects: -Also the designed we are using at different level is on circular itself and changing it would lead to change at different level of manufacturing.

A safe place for data

volatile memory: -Storage, such as DRAM, that retains data only if it is receiving power. E.g.- SRAM, DRAM, Register, cache memory

Non-volatile memory: - A form of memory that retains data even in the absence of a power source and that is used to store programs between runs.

Magnetic disk is non-volatile. (That uses magnet to store the data). Optical disc like CD (compact disc) & DVD (Digital versatile disc) & blue ray disc is non-volatile (these uses light to store data)

Main memory Also called primary memory. Memory used to hold programs while they are running; typically consists of DRAM in today's computers. DRAM-Dynamic random-access memory.

secondary memory: - Non-volatile memory used to store programs and data between runs; typically consists of magnetic disks in today's computers. magnetic disk Also called hard disk. A form of non-volatile secondary memory composed of rotating platters coated with a magnetic recording material.

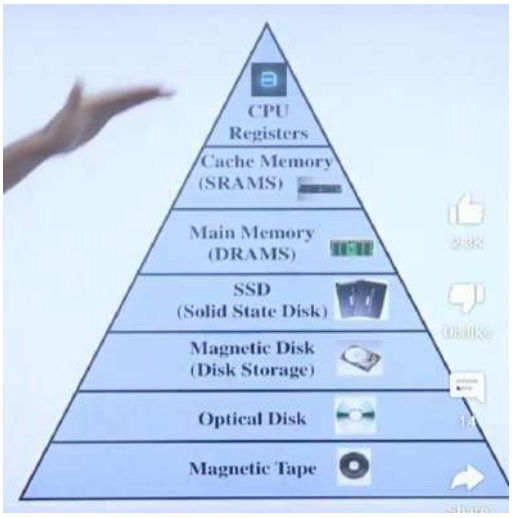
Flash memory A non-volatile semiconductor memory. It is cheaper and slower than DRAM but more expensive and faster than magnetic disks.

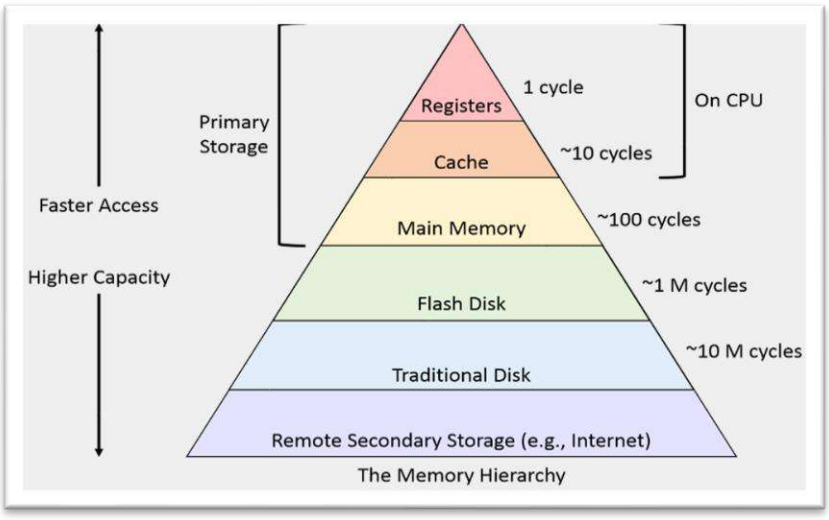
Flash memory is semiconductor memory can be used as an alternative to secondary memory if less storage is needed as these are faster but costly. E.g.: - SSD, USB(pen drive or thumb drive or memory stick)

Semiconductor memory are non-volatile. Flash memory are used in mobile phones instead of disk.

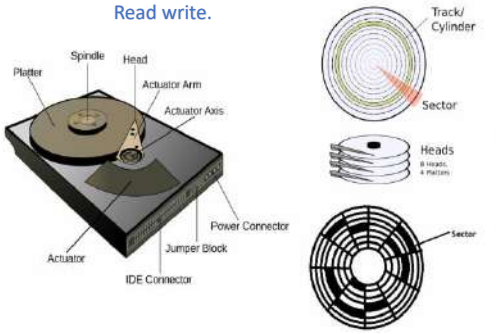
SSDs (Solid-State Drives) are a type of storage device that primarily use NAND flash memory for data storage.

Name Flash: -Flash memory got its name from the way it erases data in a "flash" or all at once, unlike older EEPROM (Electrically Erasable Programmable Read-Only Memory) technology that erased data one byte





Magnetic hard disc



Spindle rotates at 5400 to 15000 revolution per second.

Dia: - 1 inch to 3.5 inch.

Most hard drive appears inside the computer, but it can also be attached from outside using USB (Universal serial Bus). An actuator is **a component of a machine that is responsible for moving and controlling a mechanism or system, for example by opening a valve.**

DRAM provide 100,000 faster speeds than disk but per unit storage cost is also high and about 30 to 100 times as cost of IC manufacturing is more.

Thus, there are three primary differences between magnetic disks and main memory: disks are non-volatile because they are magnetic; they have a slower access time because they are mechanical devices; and they are cheaper per gigabyte because they have very high storage capacity at a modest cost.

Flash memory a challenger of Magnetic disc

Flash memory has same band width but provide more speed than magnetic disc, but it has smaller capacity at the same cost. Camera also uses this

	magnetic disc	Flash memory
cost :-	1	6-10

Unlike disks and DRAM, flash memory bits wear out after 100,000 to 1,000,000 writes. Thus, systems must keep track of the number of writes and have a strategy to avoid wearing out storage, such as by moving popular data

response time Also called execution time. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

Throughput Also called band width. Another measure of performance, it is the number of tasks completed per unit time.

Individual computer user will take care and worry about Response time or execution time.

Datacentre managers are often interested in increasing throughput or bandwidth— the total amount of work done in a given time.

In discussing the performance of computers, we will be primarily concerned with response time for the first few chapters. To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase "X is n times faster than Y"—or equivalently "X is n times as fast as Y"—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times faster than Y, then the execution time on Y is n times longer than it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more floating-point operations, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program *execution time* is measured in seconds per program. However, time can be defined in different ways, depending on what we count. The most straightforward definition of time is called *wall clock time*, *response time*, or *elapsed time*. These terms mean the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything.

Computers are often shared, however, and a processor may work on several programs simultaneously. In such cases, the system may try to optimize throughput rather than attempt to minimize the elapsed time for one program. Hence, we often want to distinguish between the elapsed time and the time that the processor is working on our behalf. **CPU execution time** or simply **CPU time**, which recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**. Differentiating between system and user CPU time is difficult to

because it is often hard to assign responsibility of operating system(system cpu time) and user program (that govern user cpu time).

Elaboration: Although you might expect that the minimum CPI is 1.0, as we'll see in Chapter 4, some processors fetch and execute multiple instructions per clock cycle. To reflect that approach, some designers invert CPI to talk about *IPC*, or *instructions per clock cycle*. If a processor executes on average 2 instructions per clock cycle, then it has an IPC of 2 and hence a CPI of 0.5.

To avoid the potential confusion between the terms increasing and decreasing, we usually say “improve performance” or “improve execution time” when we mean “increase performance” and “decrease execution time.”

Manufacturing and Benchmarking

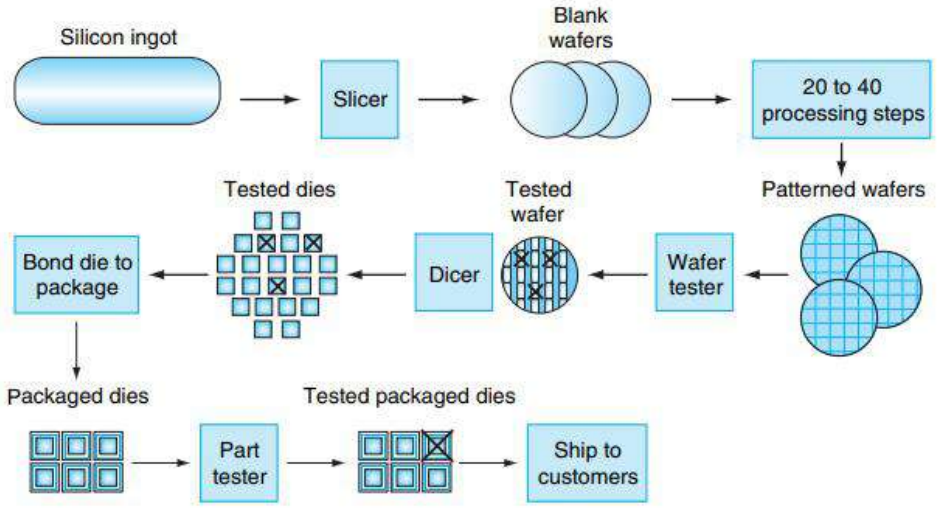


FIGURE 1.18 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.19). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Then, the wafers are diced into dies (see Figure 1.9). In this figure, one wafer produced 20 dies, of which 17 passed testing. (*X* means the die is bad.) The yield of good dies in this case was 17/20, or 85%. These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

Yield of each wafer may be different.

Silicon crystal ingot A rod composed of a silicon crystal that is between 8 and 12 inches in diameter and about 12 to 24 inches long.

wafer A slice from a silicon ingot no more than 0.1 inch thick, used to create chips.

The manufacture of a chip begins with silicon, a substance found in sand. Because silicon does not conduct electricity well, it is called a semiconductor. With a special chemical process(doping), it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

■ Excellent conductors of electricity (using either microscopic copper or aluminium wire)

■ Excellent insulators from electricity (like plastic sheathing or glass)

■ Areas that can conduct or insulate under special conditions (as a switch)

Read this carefully why wafer is not delivered directly and why dies are developed also with consequence of large dies along with how dies are connected for delivery.

A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can result in that area of the wafer failing. These **defects**, as they are called, make it virtually impossible to manufacture a perfect wafer. To cope with imperfection, several strategies have been used, but the simplest is to place many independent components on a single wafer. The patterned wafer is then chopped up, or *diced*, into these components, called **dies** and more informally known as **chips**. Figure 1.19 is a photograph of a wafer containing microprocessors before they have been diced; earlier, Figure 1.9 on page 20 shows an individual microprocessor die and its major components.

Dicing enables you to discard only those dies that were unlucky enough to contain the flaws, rather than the whole wafer. This concept is quantified by the **yield** of a process, which is defined as the percentage of good dies from the total number of dies on the wafer.

The cost of an integrated circuit rises quickly as the die size increases, due both to the lower yield and the smaller number of dies that fit on a wafer. To reduce the cost, a large die is often "shrunk" by using the next generation process, which incorporates smaller sizes for both transistors and wires. This improves the yield and the die count per wafer.

Once you've found good dies, they are connected to the input/output pins of a package, using a process called *bonding*. These packaged parts are tested a final time, since mistakes can occur in packaging, and then they are shipped to customers.

As mentioned above, an increasingly important design constraint is power. Power is a challenge for two reasons. First, power must be brought in and distributed around the chip; modern microprocessors use hundreds of pins just for power and ground! Similarly, multiple levels of interconnect are used solely for power and ground distribution to portions of the chip. Second, power is dissipated as heat and must be removed. An AMD Opteron X4 model 2356 2.0 GHz burns 120 watts in 2008, which must be removed from a chip whose surface area is just over 1 cm^2!

Elaboration: The cost of an integrated circuit can be expressed in three simple equations:

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

The first equation is straightforward to derive.

The second is an approximation, since it does not subtract the area near the border of the round wafer that cannot accommodate the rectangular dies

The final equation is based on empirical observations of yields at integrated circuit factories, with the exponent related to the number of critical processing steps.

Hence, depending on the defect rate and the size of the die and wafer, costs are generally not linear in die area.

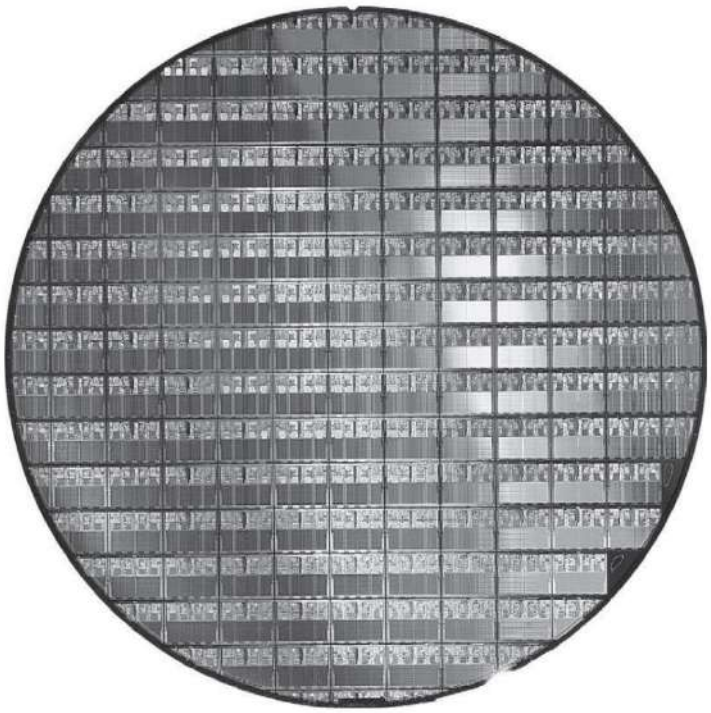
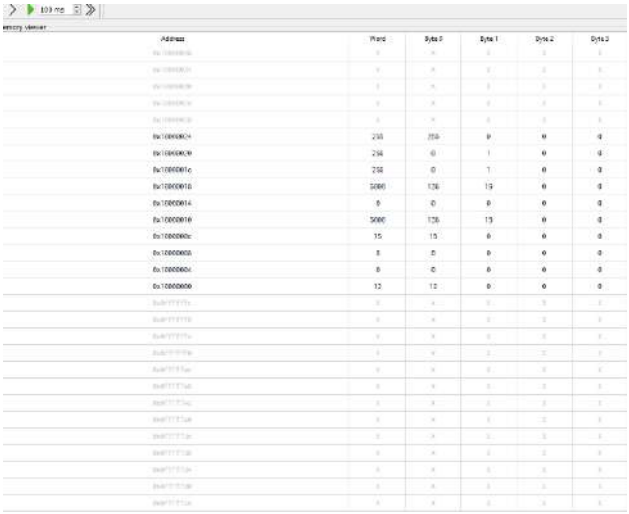


FIGURE 1.19 A 12-inch (300mm) wafer of AMD Opteron X2 chips, the predecessor of Opteron X4 chips (Courtesy AMD). The number of dies per wafer at 100% yield is 117. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it's easier to create the masks used to pattern the silicon. This die uses a 90-nanometer technology, which means that the smallest transistors are approximately 90 nm in size, although they are typically somewhat smaller than the actual feature size, which refers to the size of the transistors as "drawn" versus the final manufactured size.

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle



What does it mean:

We want to store 5000.

Byte0 have 136.

Byte1 have 19.

It means that when you will write binary format.

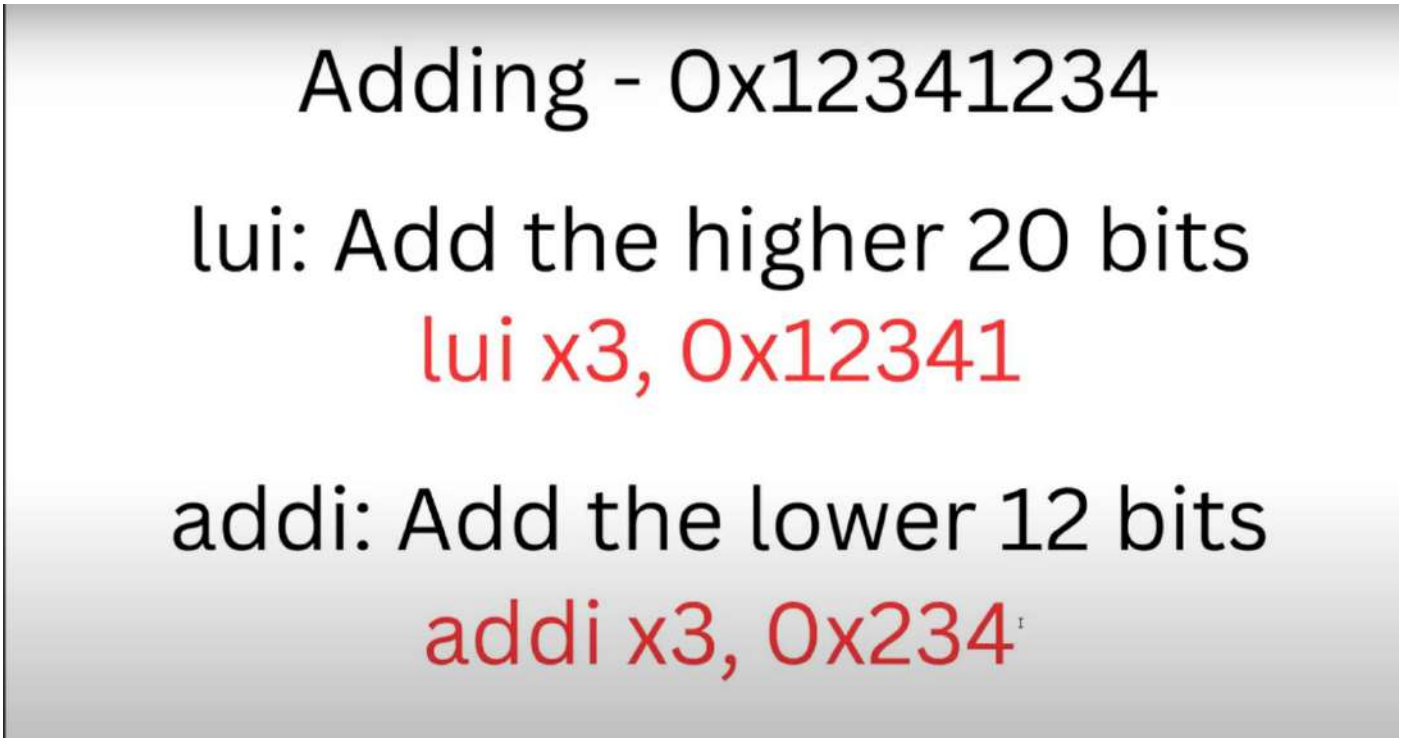
As 5000 is in between 2^8 to 2^16 so it will take 2 bytes.

Byte 0 & 1.

136 denotes the value /sum in Byte0.

While 19 denotes in 2nd Byte i.e., Byte1

1. **Decimal (Base-10):** Decimal numbers are the standard numbers we use in everyday life and don't require a specific prefix. For example, "42" is a decimal number.
2. **Binary (Base-2):** Binary numbers are often preceded by "0b" or "0B" to indicate that they are in base-2. For example, "0b1101" is a binary number.
3. **Octal (Base-8):** Octal numbers are typically preceded by a "0" to indicate that they are in base-8. For example, "037" is an octal number.
4. **Hexadecimal (Base-16):** Hexadecimal numbers are often preceded by "0x" or "0X" to indicate that they are in base-16. For example, "0x1A" is a hexadecimal number.
5. **Base-n (Custom Base):** In some contexts, you might encounter numbers in a custom base, and they are typically preceded by the base itself. For example, "16#2A" could represent a number in base-16 (hexadecimal).



The memory address `0×7ffffff0` is 32 bits long. In hexadecimal notation, it's represented by eight characters (0×7ffffff0), and each character represents 4 bits.

To convert this to binary, you would get:

yaml Copy code

```
0111 1111 1111 1111 1111 1111 1111 0000
```

This is 32 bits in total. Each 0 or 1 in this sequence represents a single bit.

The memory address `0×7ffffff0` appears to be a 64-bit memory address. In a 64-bit system, memory addresses are typically 64 bits (8 bytes) in length.

Here's a breakdown of the memory address:

- `0x`: The "0x" prefix indicates that the address is in hexadecimal notation.
- `7ffffff0`: This hexadecimal number represents the memory address.

In a 64-bit system, memory addresses are represented using 64 bits, allowing for a much larger addressable memory space compared to 32-bit systems. This larger address space is one of the advantages of 64-bit architectures, as it enables systems to address more memory, which is crucial for handling large datasets and running complex applications.

#this is the way to comment out

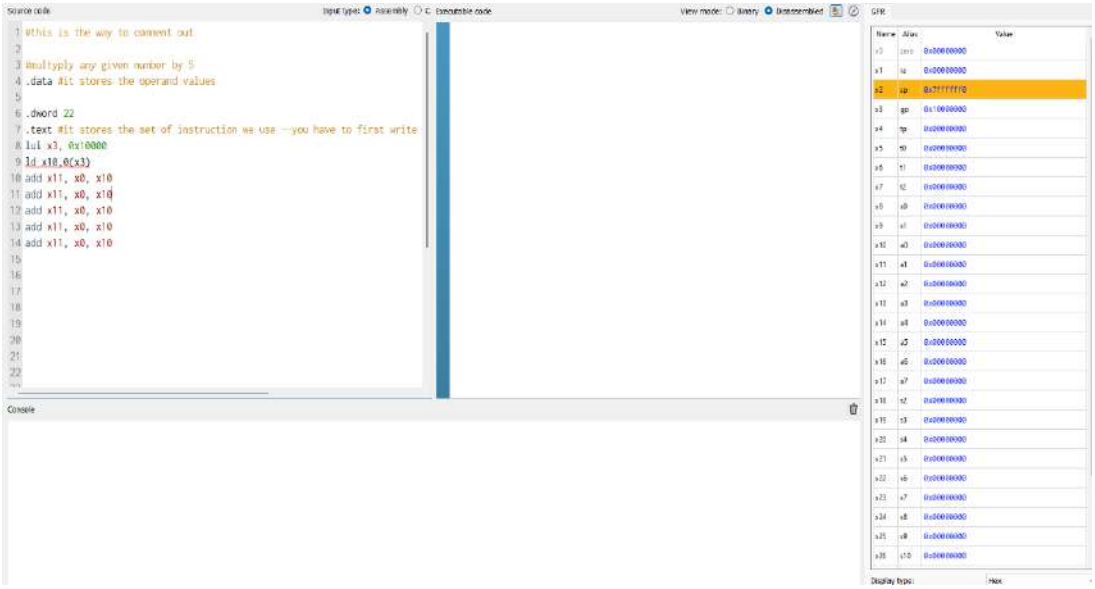
#multyply any given number by 5
.data #it stores the operand values

.dword 22
.text #it stores the set of instruction we use --you have to first write .text then then value will start loading
lui x3, 0x10000
ld x10, x(x3)
add x11, x0, x10
add x11, x0, x10
add x11, x0, x10
add x11, x0, x10

remember “,” must be there .here ripes can work without comma also but actual RISC V you have to put comma

Note :- you can use ld when your architecture is 64 bit as ld takes 64 bit—it mean you cant use in 32 bit .To load the data in 32 bit you should use lw takes 32 bit .

if you see line 9 then it is throwing error because we are using in 32 bit configuration(initially my ripes was in 32 bit architecture) and ld is the instruction set of 64 bit configuration



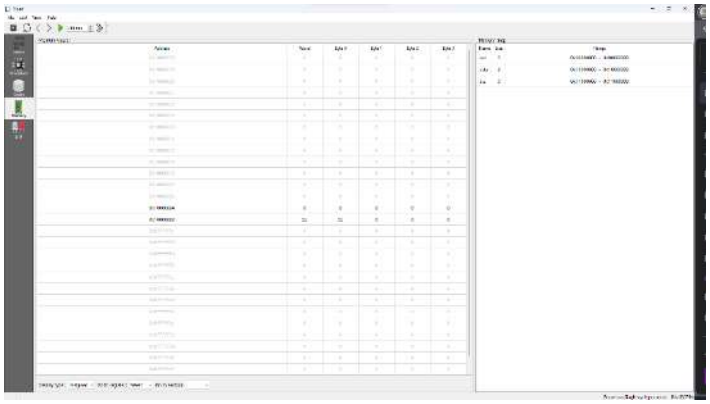


figure 1 How double word are treated in the 32 bits

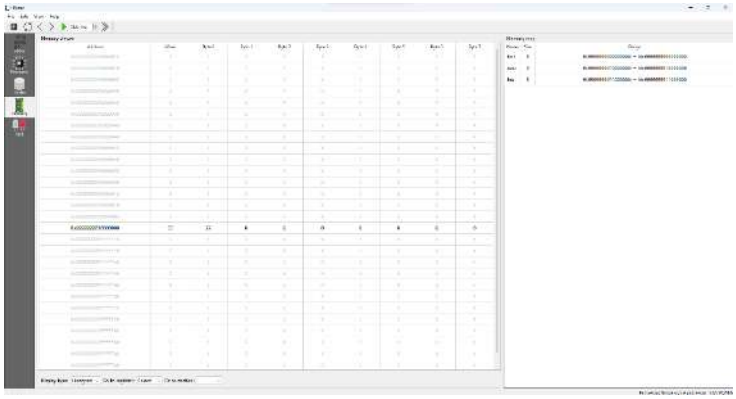


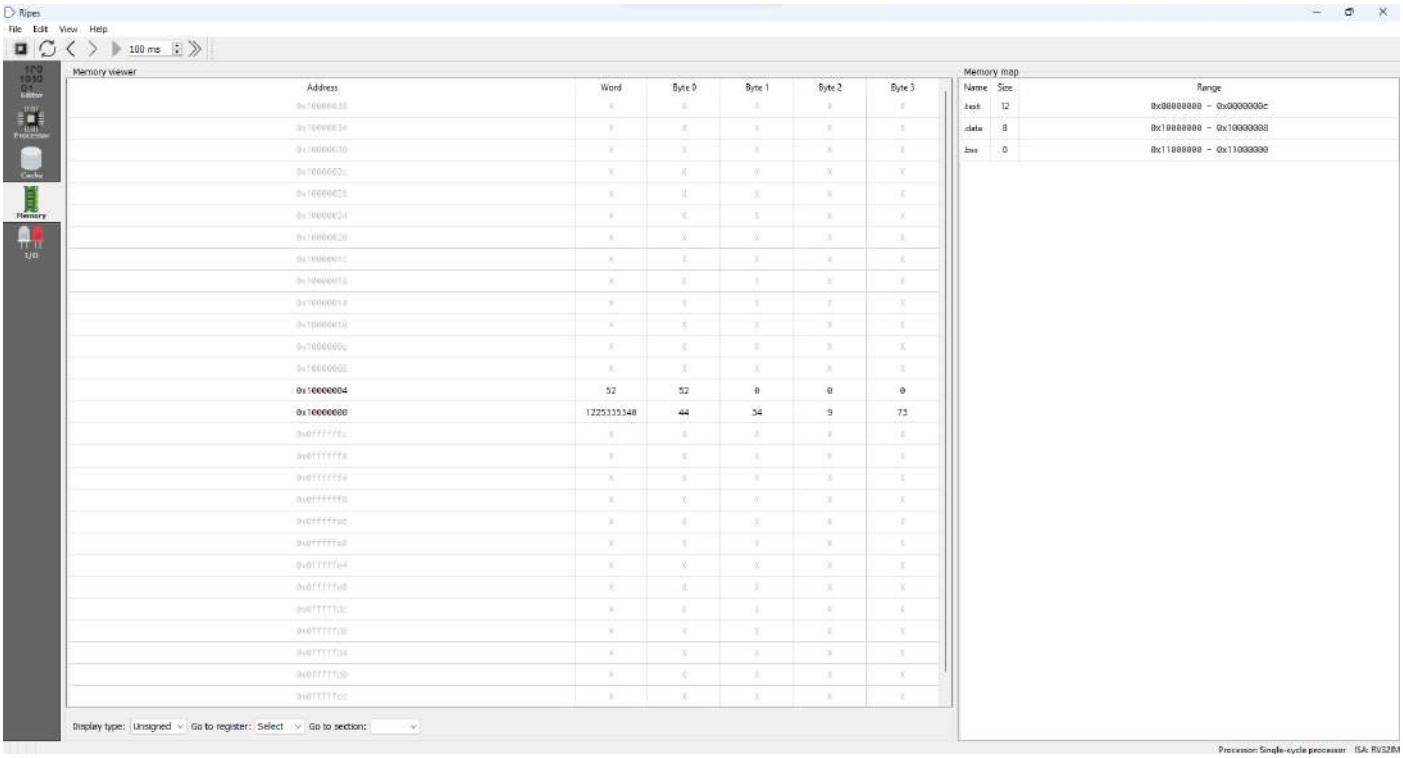
figure 1 How double word are treated in the 64 bit

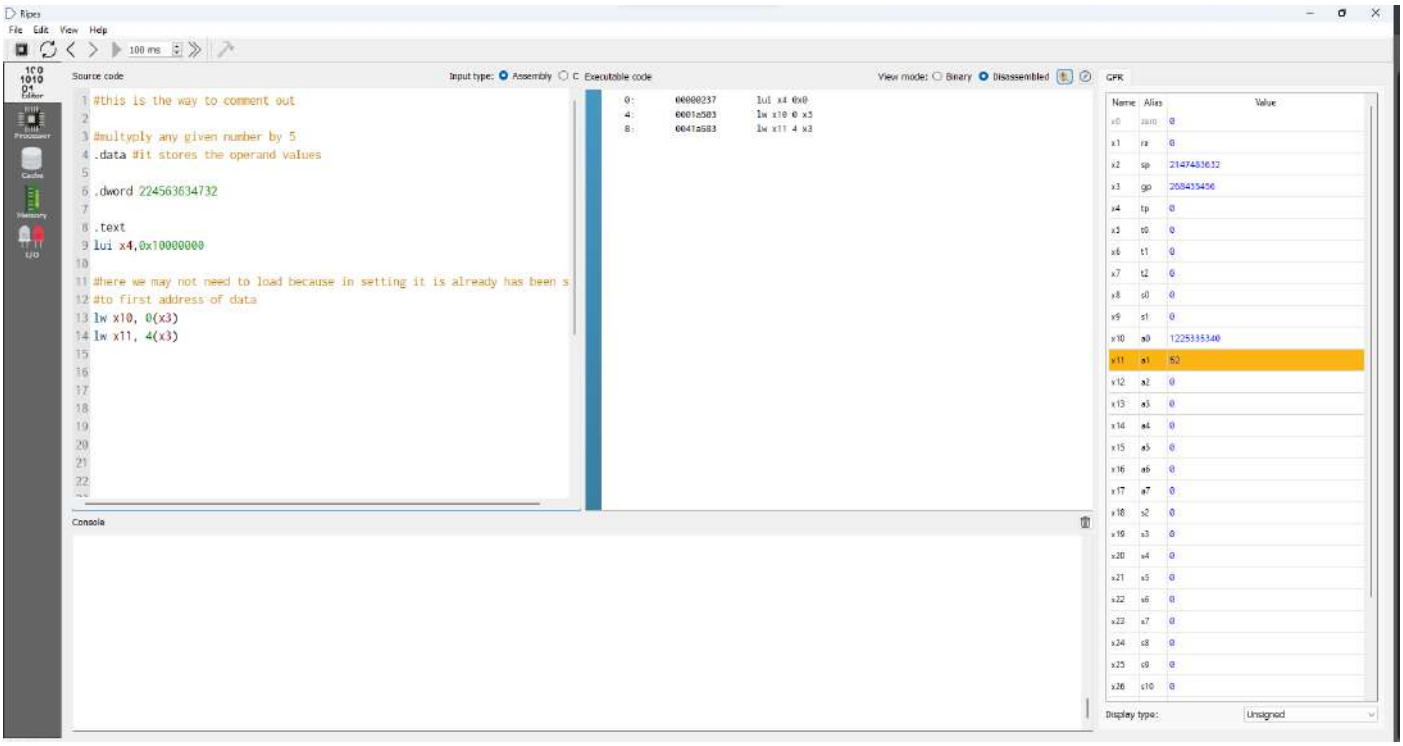
as this is 32 bits so word max can handle 32 bits but

double have 64 bits so 32-bit architecture uses two double

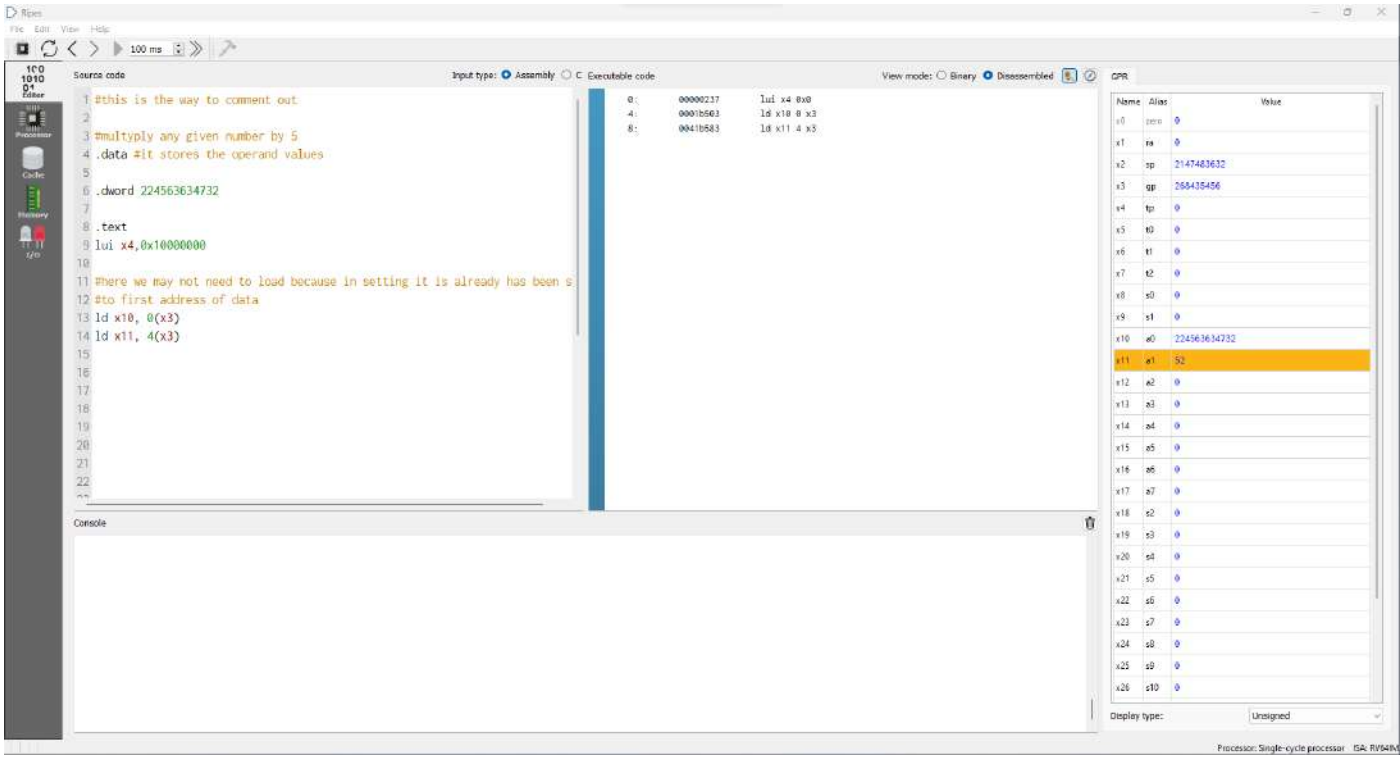
word.

0b00000000000000000000000000110100



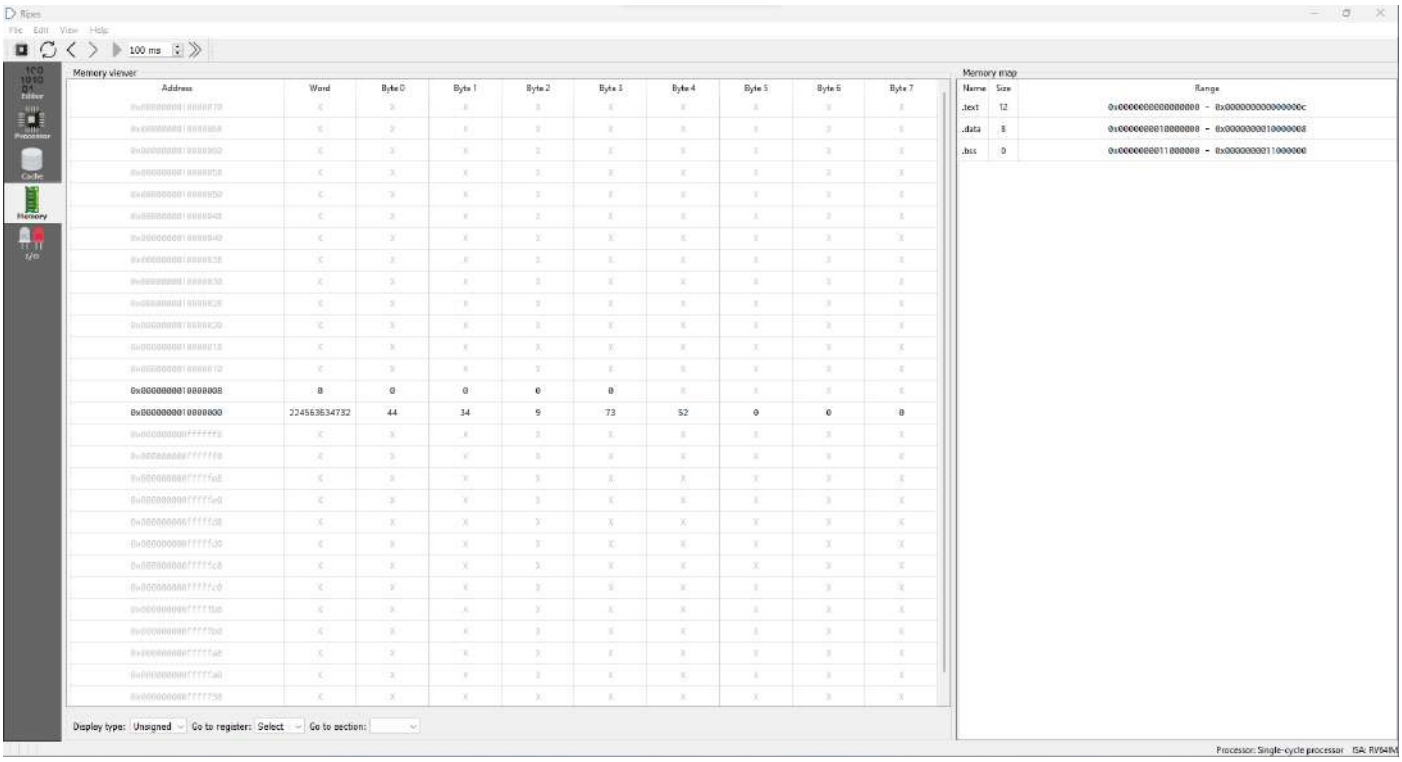


The above one is 32 bit architecture so .dword is valid but ld function is not valid .The above shows the way to load dword in 32 bit arch.



Here we are using 64 bit architecture so while dword is now came in one one register and only one one address of 8 bit is allocated instead two address of 4 bit used in 32 bit

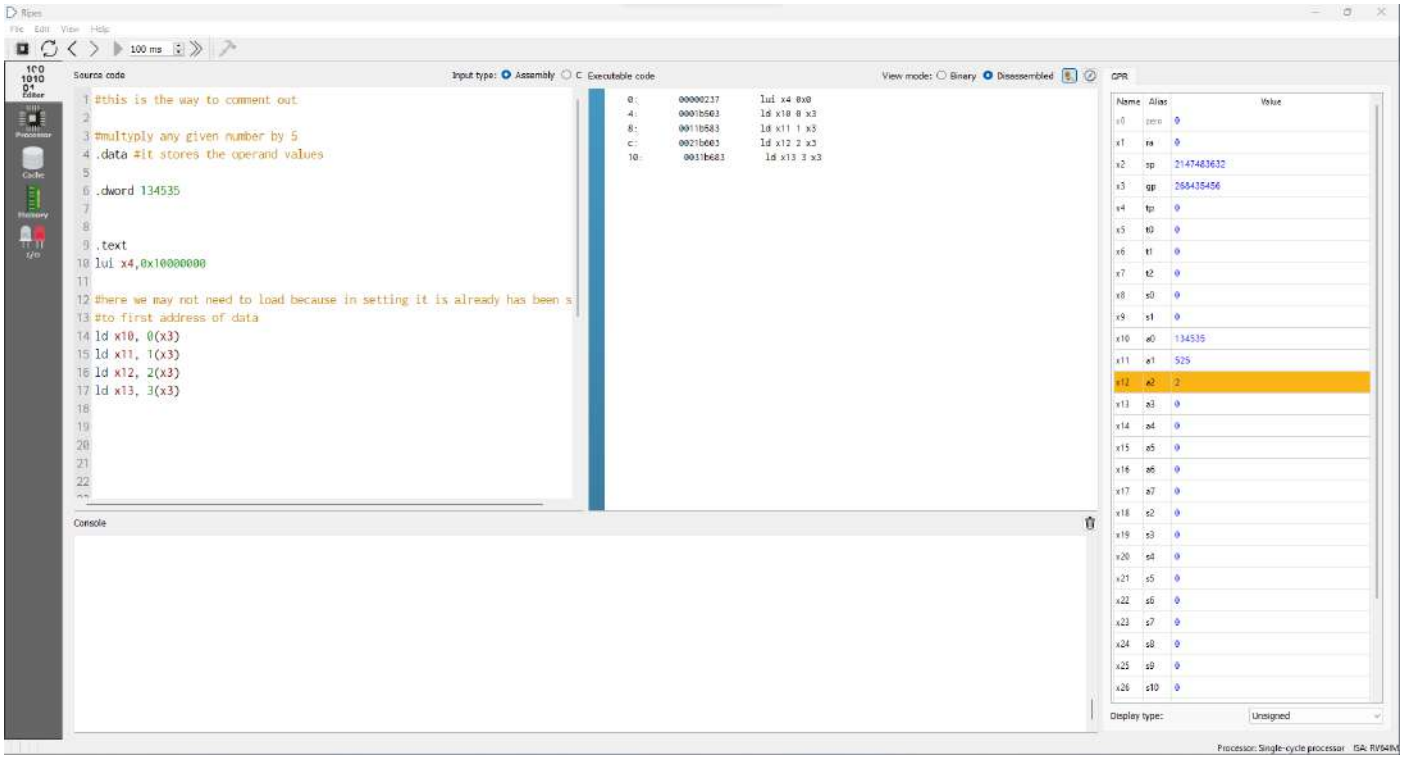
Also if we are loading 4(x3) then it will start loading from byte 4 and load 8 byte from here i.e byte 4 ,5,6,7 and in next address it will load rest 4 byte byte 0,1,2,3(in 0x10000008)



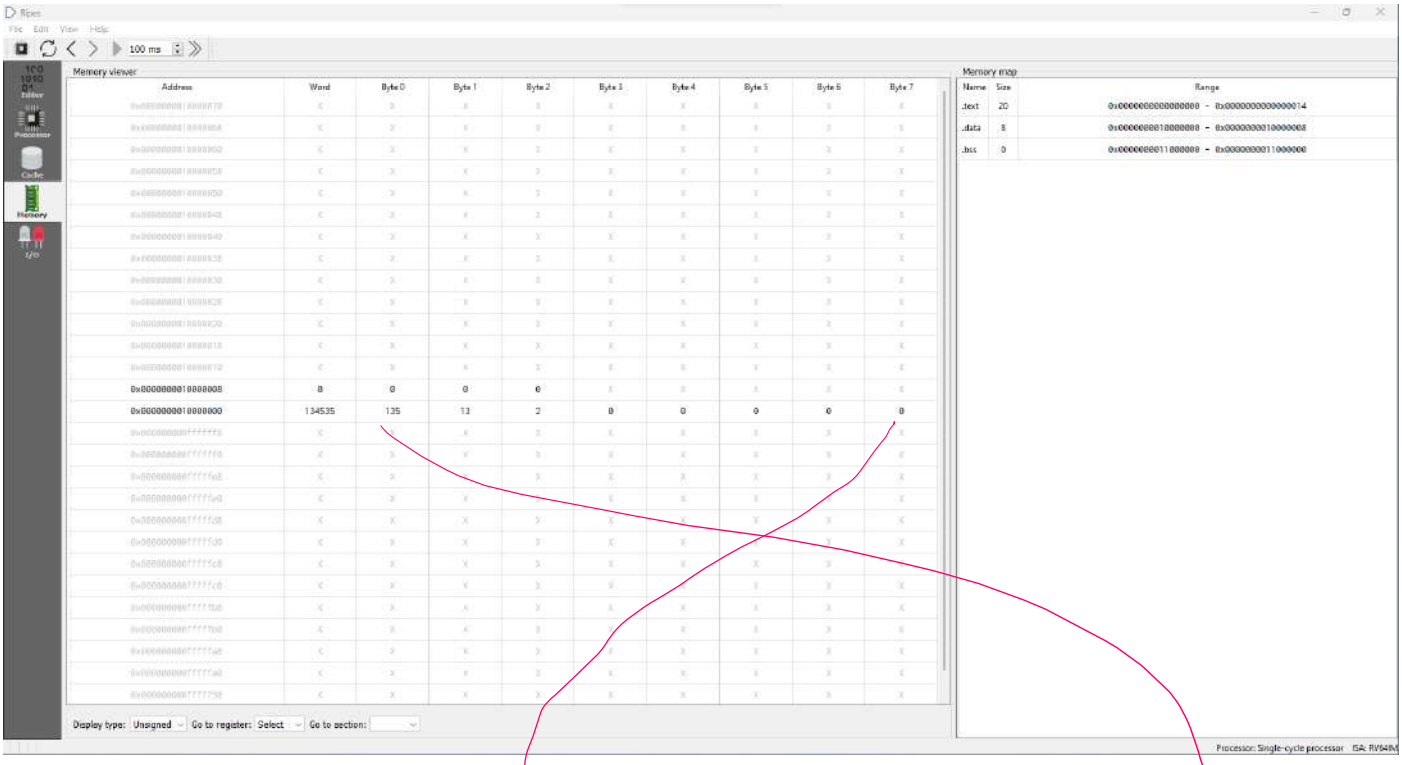
Q. we cants use ld in 32 bit but can we use lw in 64 bit ?????

Sol:-I think its answer should be yes because more than amplified amount of byte is present only thing is it might not load correct value . and ya answer is yes

see and observ:-



memory alignment of above: -



Also, as risk 5 works on little endian: 00000000 00000000 00000000 00000000 00000010 00001101 10000111

MSB

LSB

As this is 64 bit architecture so total 8 Byte will be in one register

Byte 0 — Byte 7

As RISC-V follow little endian format so LSB → min address

Byte 0

Q How unsigned is represented as above?

$1\times2^7 + 0\times2^6 + 0 + 0\times2^4 + 0 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 128 + 4 + 2 + 1 = 135$

Storing Arrays in Memory

- An array of 64-bit elements: P[10]
  - Stored in contiguous locations
  - Offset of each element from base is 8*index
  - Total size: 10*8 Byte = 80 Bytes
- Ordering the 8 bytes in each element?

If 32-bit architecture will be there, then each offset will be of 4 bytes as each memory block is for word i.e. 4*8 →32 bits while in 64 bit each memory block is sufficient to store a double word.

Little Endian Format

- Little Endian (LE)
  - Least-significant byte stored at least address of a word
- Big Endian (BE)
  - Most-significant byte stored at least address
- RISC-V uses Little Endian

	LE	BE
7	B7	B0
6	B6	B1
5	B5	B2
4	B4	B3
3	B3	B4
2	B2	B5
1	B1	B6
0	B0	B7

- Example: Two 64-bit data P and L
  - P contains bytes P7 P6 P5 P4 P3 P2 P1 P0
  - L contains bytes L7 L6 L5 L4 L3 L2 L1 L0

Addr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	P0	P1	P2	P3	P4	P5	P6	P7	L0	L1	L2	L3	L4	L5	L6	L7

It does not make difference in which order you are storing only things to be kept in mind is the order of traversal or access should be same as it was entered.

Memory Operand Example

- C code:

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

  - h in x21, base address of A in x22
- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

load doubleword

```
addi    x23, x22, 64    //x23 ← x22+64
ld      x9, 0(x23)      //x9 ← *(x23)
add     x9, x21, x9
addi    x23, x22, 96    //x23 ← x22+96
sd      x9, 0(x23)      //*(x23) ← x9
```

store doubleword

Memory Operand Example

- C code:

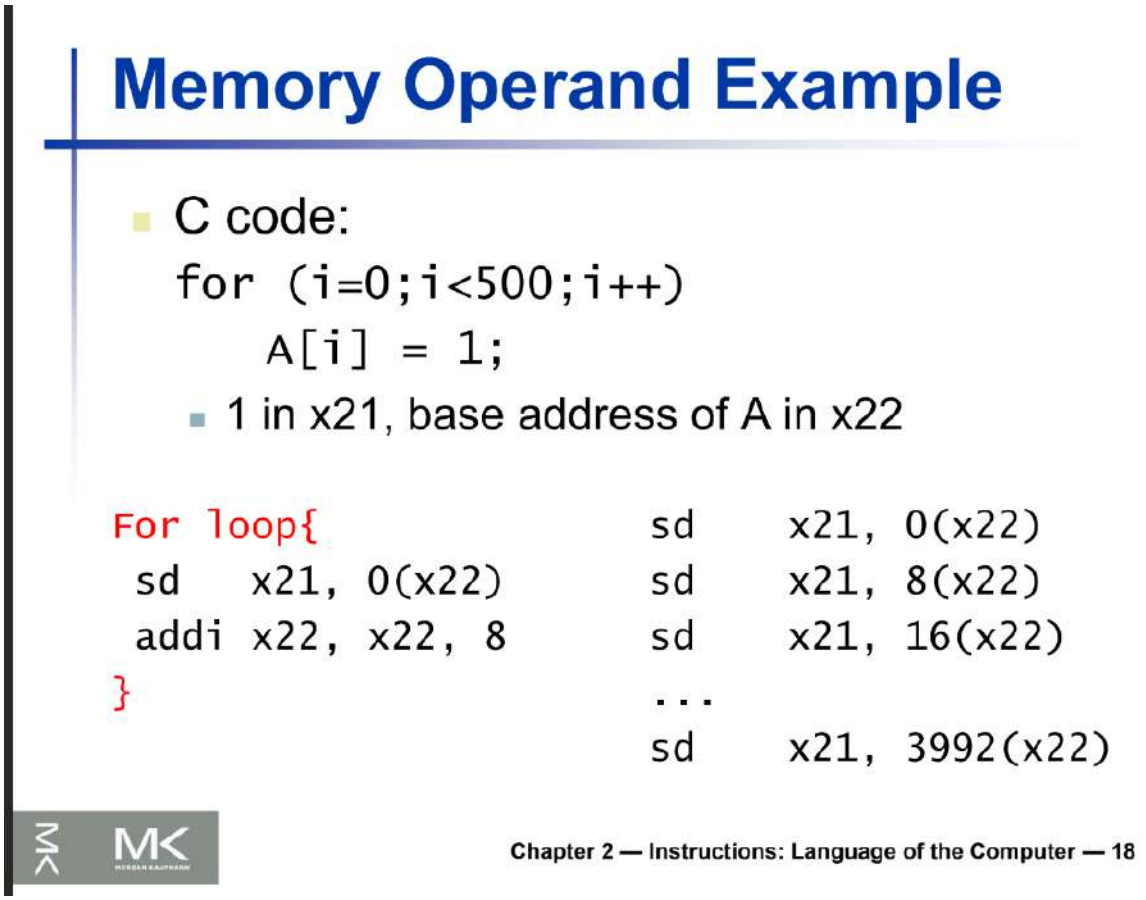
```
A[12] = h + A[8];
```

  - h in x21, base address of A in x22
- Compiled RISC-V code: (another variant)

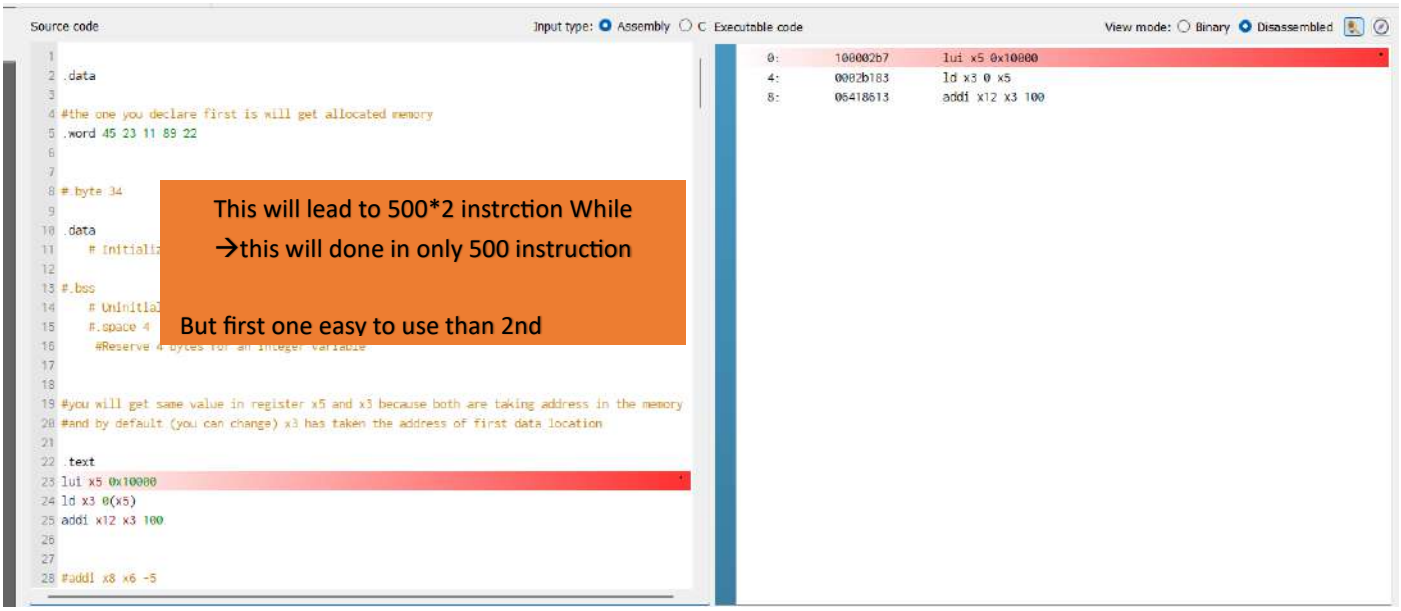
```
ld      x9, n(x23)      //x9 ← *(x23+n)
sd      x9, n(x23)      //*(x23+n) ← x9

ld      x9, 64(x22)     //x9 ← *(x22+64)
add     x9, x21, x9
sd      x9, 96(x22)     //*(x22+96) ← x9
```

Is this always better than previous case?

No, it will good when you are writing few lines of code and no loop is required but when you use loop it is not possible to write every offset. Even though this is more efficient but lead to too much line of assembly code that may be tedious job for programmer.



It may be possible that second code is more efficient even though it is not possible for user to write that much long code. So both should be taken care of.



Memory viewer

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x0000000010000070	X	X	X	X	X	X	X	X	X
0x0000000010000068	X	X	X	X	X	X	X	X	X
0x0000000010000060	X	X	X	X	X	X	X	X	X
0x0000000010000058	X	X	X	X	X	X	X	X	X
0x0000000010000050	X	X	X	X	X	X	X	X	X
0x0000000010000048	X	X	X	X	X	X	X	X	X
0x0000000010000040	X	X	X	X	X	X	X	X	X
0x0000000010000038	X	X	X	X	X	X	X	X	X
0x0000000010000030	X	X	X	X	X	X	X	X	X
0x0000000010000028	X	X	X	X	X	X	X	X	X
0x0000000010000020	X	X	X	X	X	X	X	X	X
0x0000000010000018	X	X	X	X	X	X	X	X	X
0x0000000010000010	22	22	0	0	0	X	X	X	X
0x0000000010000008	382252089355	11	0	0	0	89	0	0	0
0x0000000010000000	98784247853	45	0	0	0	23	0	0	0
0x00000000ffffff8	X	X	X	X	X	X	X	X	X
0x00000000ffffff0	X	X	X	X	X	X	X	X	X
0x00000000fffffe8	X	X	X	X	X	X	X	X	X
0x00000000fffffe0	X	X	X	X	X	X	X	X	X
0x00000000fffffd8	X	X	X	X	X	X	X	X	X
0x00000000fffffd0	X	X	X	X	X	X	X	X	X
0x00000000fffffc8	X	X	X	X	X	X	X	X	X

...000, ...008, ...016 ✗

+8 +8

Because this is a hexadecimal representation not decimal

$(10000000)_{16}$

≡

Adding +8 →

$(0001\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2$

$(0001\ 0000\ 0000\ \cdots\ 1000)_2$

≡ $(1000001)_{16}$

Again adding +8

$(0001\ \cdots\ 0001\ 1000)_2$ (16)

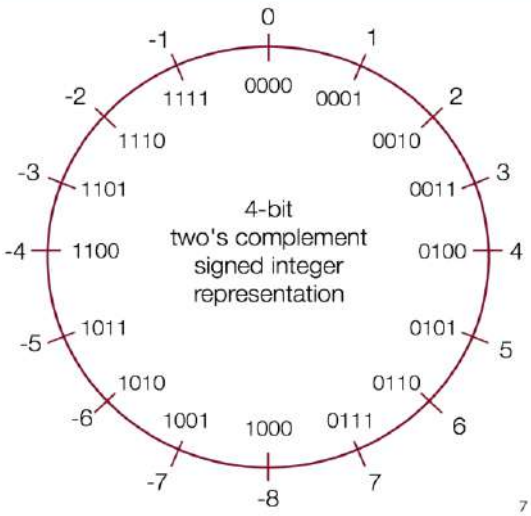
≡ $(1000010)_{16}$

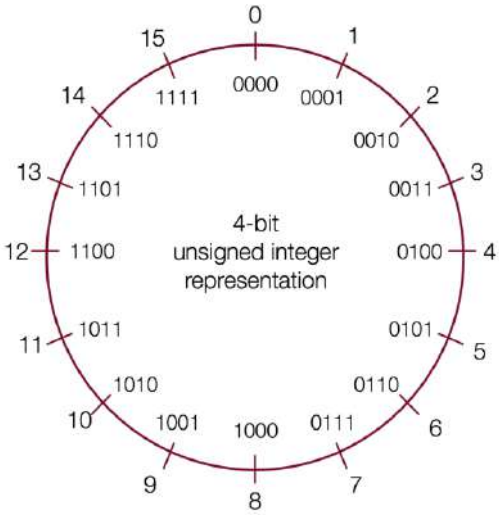
14 → 1110
5 → 0101
9 → 1001

12 ← 1100
5 ← 0101
7 = 0111

24 = 11000
1 = 00001
25 = 11001

24 = 11000
15 = 01111
9 = 01001





7

Overflow and Underflow

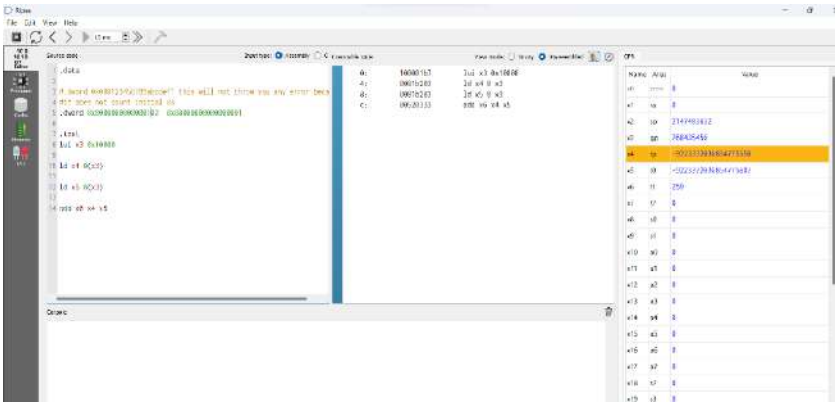
- If you exceed the **maximum** value of your bit representation, you *wrap around* or *overflow* back to the **smallest** bit representation.

```
0b1111 + 0b1 = 0b0000
```

- If you go below the **minimum** value of your bit representation, you *wrap around* or *underflow* back to the **largest** bit representation.

```
0b0000 - 0b1 = 0b1111
```

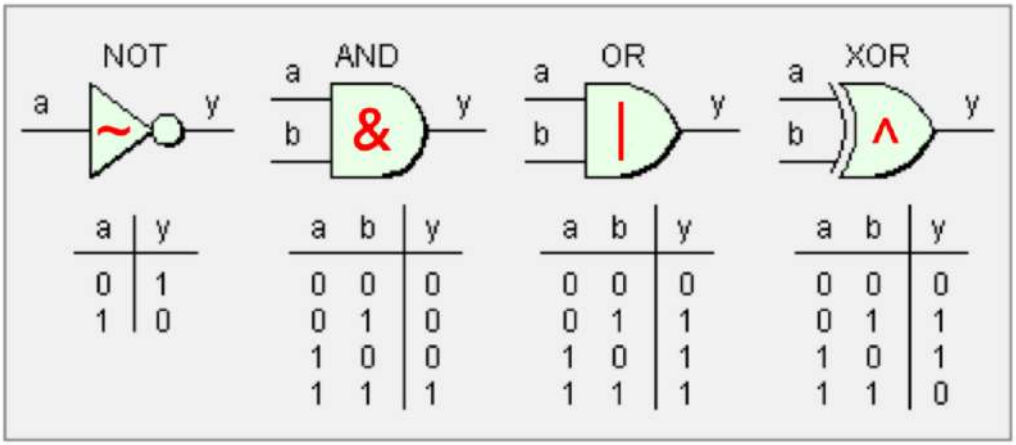
9



Here the overflow is happening .point to be noted that when we are adding to large number it is getting overflow but not giving max or min of that range instead giving a first 64 bit of the number.

An Aside: Boolean Algebra

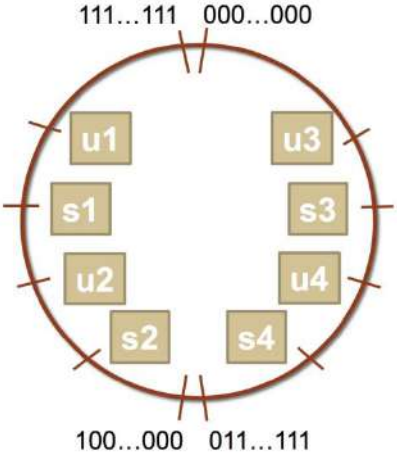
- These operators are not unique to computers; they are part of a general area called **Boolean Algebra**. These are applicable in math, hardware, computers, and more!



Comparisons Between Different Types

Which of the following statements are true? (*assume that variables are set to values that place them in the spots shown*)

1. `s3 > u3 - true`
2. `u2 > u4 - true`
3. `s2 > s4 - false`
4. `s1 > s2 - true`
5. `u1 > u2 - true`
6. `s1 > u3 - true`



Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = 53191;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit int), 53191:

0000 0000 0000 0000 1100 1111 1100 0111

When we cast x to a short, it only has 16-bits, and C *truncates* the number:

1100 1111 1100 0111

This is -12345! And when we cast sx back an int, we sign-extend the number.

1111 1111 1111 1111 1100 1111 1100 0111 // still -12345

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = -3;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit int), -3:

1111 1111 1111 1111 1111 1111 1111 1101

When we cast x to a short, it only has 16-bits, and C *truncates* the number:

1111 1111 1111 1101

This is -3! **If the number does fit, it will convert fine.** y looks like this:

1111 1111 1111 1111 1111 1111 1111 1101 // still -3

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
unsigned int x = 128000;
unsigned short sx = x;
unsigned int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit unsigned int), 128000:

0000 0000 0000 0001 1111 0100 0000 0000

When we cast x to a short, it only has 16-bits, and C *truncates* the number:

1111 0100 0000 0000

This is 62464! **Unsigned numbers can lose info too.** Here is what y looks like:

0000 0000 0000 0000 1111 0100 0000 0000 // still 62464

Expanding Bit Representations

- Sometimes, we want to convert between two integers of different sizes (e.g. short to int, or int to long).
- We might not be able to convert from a bigger data type to a smaller data type, but we do want to always be able to convert from a **smaller** data type to a **bigger** data type.
- For **unsigned** values, we can add *leading zeros* to the representation ("zero extension")
- For **signed** values, we can *repeat the sign of the value* for new digits ("sign extension"
- Note: when doing <, >, <=, >= comparison between different size types, it will *promote to the larger type*.

Expanding Bit Representation

```
unsigned short s = 32772;
// short is a 16-bit format, so                    s = 1000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 1000 0000 0000 0100b
```


Expanding Bit Representation

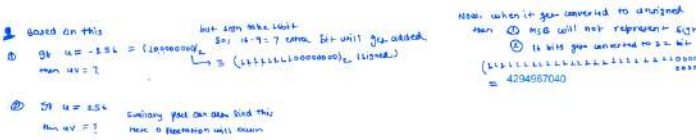
```
unsigned short s = 32772;
// short is a 16-bit format, so                    s = 1000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 1000 0000 0000 0100b
```

— or —

```
short s = -4;
// short is a 16-bit format, so                    s = 1111 1111 1111 1100b

int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```

Casting

- What happens at the byte level when we cast between variable types? **The bytes remain the same! This means they may be interpreted differently depending on the type.**



```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

This prints out: "v = -12345, uv = 4294954951". **Why?**

Casting

- What happens at the byte level when we cast between variable types? **The bytes remain the same! This means they may be interpreted differently depending on the type.**

```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

-12345 in binary is **1111 1111 1111 1111 1100 1111 1100 0111.**

If we treat this binary representation as a positive number, it's *huge*!

Right Shift (>>)

There are *two kinds* of right shifts, depending on the value and type you are shifting:

- **Logical Right Shift:** fill new high-order bits with 0s.
- **Arithmetic Right Shift:** fill new high-order bits with the most-significant bit.

Unsigned numbers are right-shifted using **Logical Right Shift**.

Signed numbers are right-shifted using **Arithmetic Right Shift.**

This way, the sign of the number (if applicable) is preserved!

Shift Operation Pitfalls

1. *Technically*, the C standard does not precisely define whether a right shift for signed integers is logical or arithmetic. However, **almost all compilers/machines** use arithmetic, and you can most likely assume this.
2. Operator precedence can be tricky! For example:

   `1<<2 + 3<<4` means `1 << (2+3) << 4` because addition and subtraction have higher precedence than shifts! Always use parentheses to be sure:

   ```
   (1<<2) + (3<<4)
   ```


Bit Operator Pitfalls

- The default type of a number literal in your code is an **int**.
- Let's say you want a long with the index-32 bit as 1:

```
long num = 1 << 32;
```

- This doesn't work! 1 is by default an **int**, and you can't shift an int by 32 because it only has 32 bits. You must specify that you want 1 to be a **long**.

```
long num = 1L << 32;
```

operator precedence: -

Precedence	Operator	Description	Associativity
1	++ -- ( ) [ ] . -> (*type*){*list*}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement[note 1] Unary plus and minus Logical NOT and bitwise NOT Cast Indirection (dereference) Address-of Size-of[note 2] Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-left
14[note 4]	= += -= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

left_shift.cpp > main()

```cpp
#include <iostream>
using namespace std;
int main()
{

int a =1;
// cout<< (1<<32) <<endl;
int x=5;

//we have to find multiplication of 5 with 7 using bi
//x*6=x*(8-2)=x*8 -x*1 =x<<3 - x<<1
int ans= x<<3 - x<<1;
cout <<ans<<endl;   //give -2147483648 not correct

//remember arithmetic have higer preference than shift operaqtor so
//x*6=x*(8-2)=x*8 -x*1 =(x<<3) - (x<<1) ..this will give correct answer

int ans1= (x<<3) - (x<<1); // give 30  correct
cout <<ans1<<endl;

int b =1;
// cout<< (1<<31) <<endl;  //this will also not work as sign int have limit 2^31-1

//this will not work becasue b is still int then right shift by 32 will not work
long long num2 ;
num2 = b<<32;
cout<<num2<<endl;

long long num;

num= ((long long) b)<<32;
// num=static_cast<long long>(b)<<32;
// long num2=  b<<32;

cout<<num<<endl; //ans will be 4294967296 or 00000000000000000000000000000001000000000000000000000000000000000

//here the size of long is same as int i.e. 4byte so no benifit of using
//cout<<sizeof(long)<<endl;
// cout<<num2<<endl;

return 0;
}
```

Example-1: while and if

```
while (i < j){
  if (p >= q)
     a++;
  b=b-2;
}
Exit:
```

i: x10
j: x11
p: x12
q: x13
a: x14
b: x15

```
L1: bge x10, x11, Exit
    blt x12, x13, L2
    addi x14, x14, 1
L2: addi x15, x15, -2
    beq x0, x0, L1
Exit: ...
```


Example-2: while vs do-while

```
while (i < j){
  if (p >= q)
     a++;
  b=b-2;
}
```

```
L1: bge x10, x11, Exit
    blt x12, x13, L2
    addi x14, x14, 1
L2: addi x15, x15, -2
    beq x0, x0, L1
Exit: ...
```

```
do{
  if (p >= q)
     a++;
  b=b-2;
} while (i < j);
```

```
L1: blt x12, x13, L2
    addi x14, x14, 1
L2: addi x15, x15, -2
    blt x10, x11, L1
Exit: ...
```

Example-3: Complex if

```
if (p >= q ||
    m == n)
        a++;
  b=b-2;
}
m: x10
n: x11
p: x12
q: x13
a: x14
b: x15
```

```
C1: blt x12, x13, C2
    addi x14, x14, 1
    beq x0, x0, B1
C2: bne x10, x11, B1
    addi x14, x14, 1
B1: addi x15, x15, -2
Exit: ...
```


Example-4: Using Pointers

```
long long *p, *q;
while (p != NULL){
     *q = *p;
     q++;
     p++;
}
p: x6
q: x7
```

```
L1: beq x6, x0, Exit
    ld x8, 0(x6)
    sd x8, 0(x7)
    addi x6, x6, 8
    addi x7, x7, 8
    beq x0, x0, L1
Exit: ...
```

Example 7: Large Immediates

- How to load a large (32-bit) immediate value (0x20112f4d) in a register?
- Option-1: Store the value in memory and load using ld
- Option-2: Use logical operations
  - ```
    addi x6, x6, 0x201
    slli x6, x6, 12
    addi x6, x6, 0x12f
    slli x6, x6, 8
    addi x6, x6, 0x4d
    ```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000	0000 0010 0000 0001
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0010 0000	0001 0000 0000 0000
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0010 0000	0001 0001 0010 1111
0000 0000 0000 0000	0000 0000 0000 0000	0010 0000 0001 0001	0010 1111 0000 0000
0000 0000 0000 0000	0000 0000 0000 0000	0010 0000 0001 0001	0010 1111 0100 1101

Example 7: Large Immediates

- Option-3: Using lui

  `lui rd, immediate`
  - Copies 20-bit immediate to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0
- Loading 0x20112f4d using lui:

```
lui x19, 0x20112  // 0x20112
```

0000 0000 0000 0000	0000 0000 0000 0000	0010 0000 0001 0001 0010	0000 0000 0000

```
addi x19,x19,0xf4d  // 0xf4d
```

0000 0000 0000 0000	0000 0000 0000 0000	0010 0000 0001 0001 0010	1111 0100 1101
```

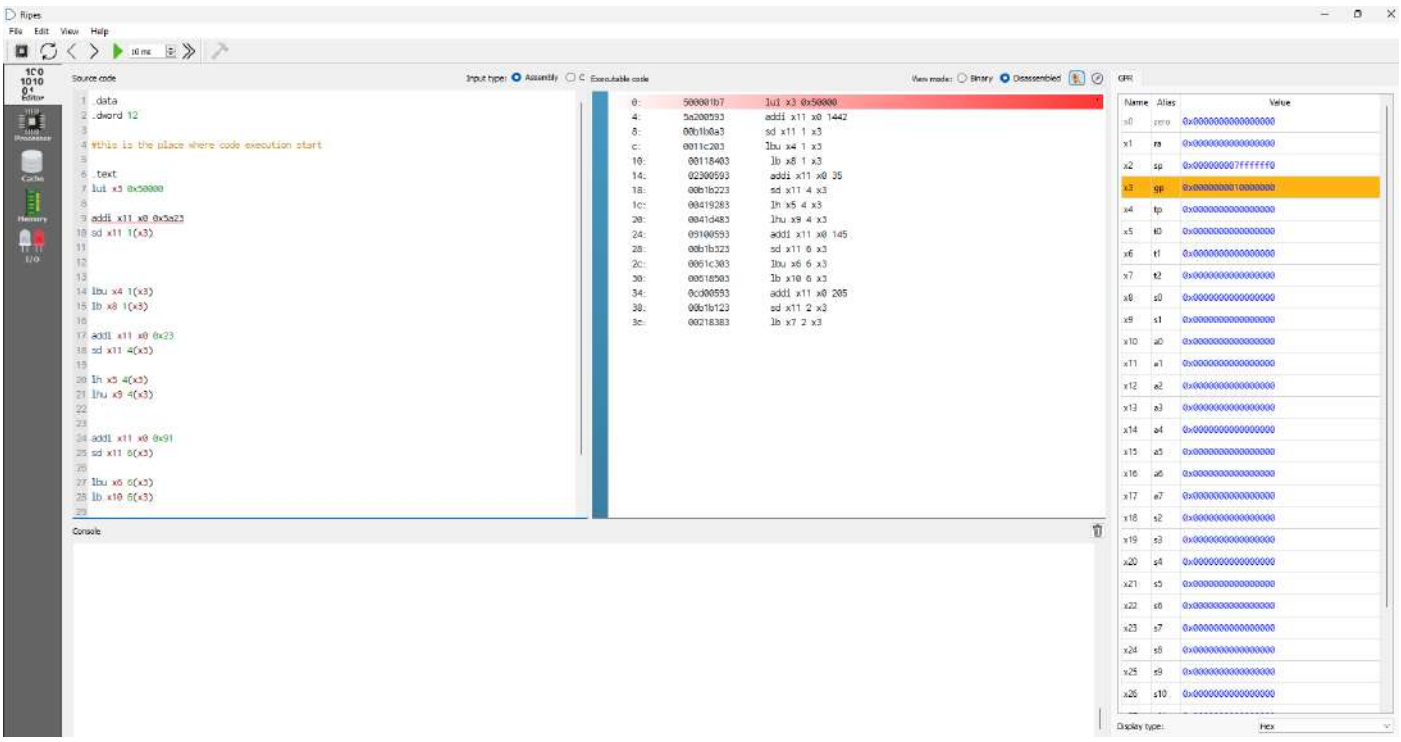
Q. why these register haven been specified some specifier?

RISC-V Registers

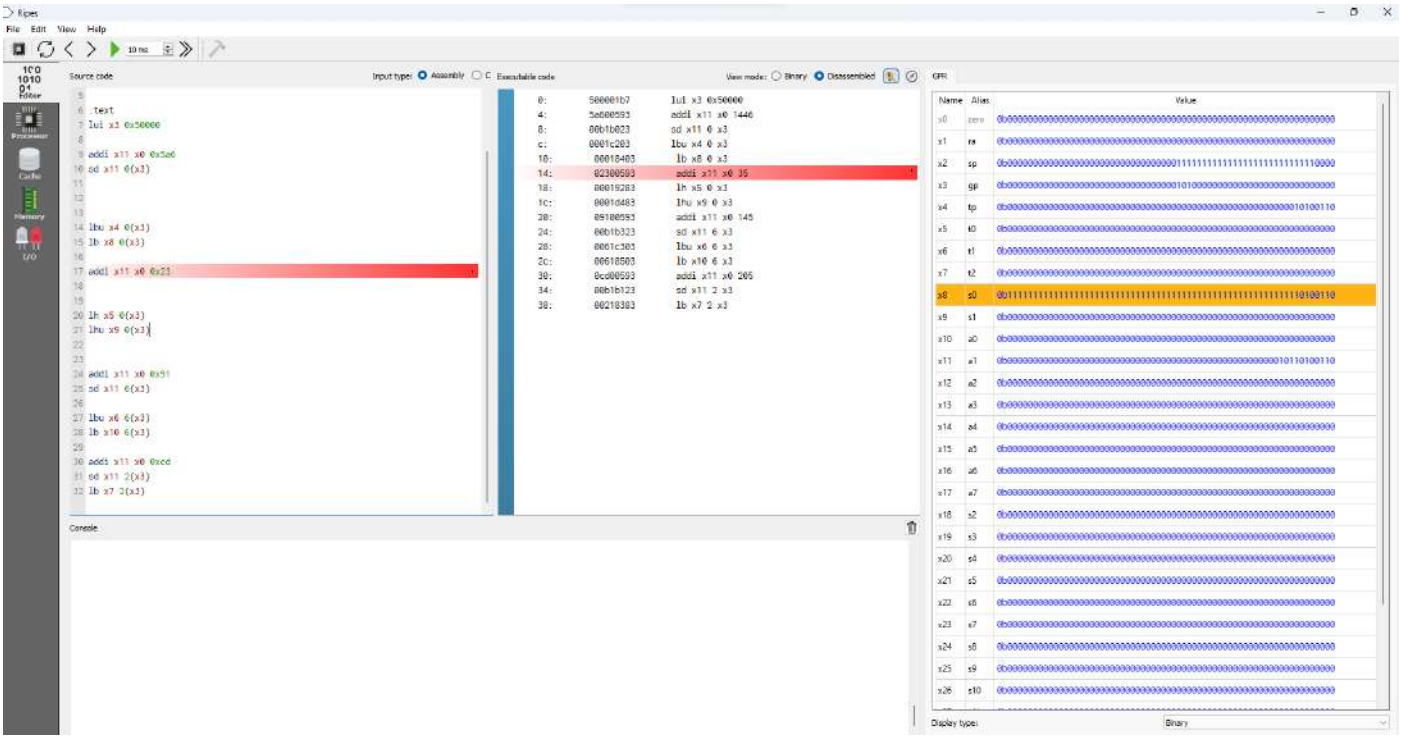
- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

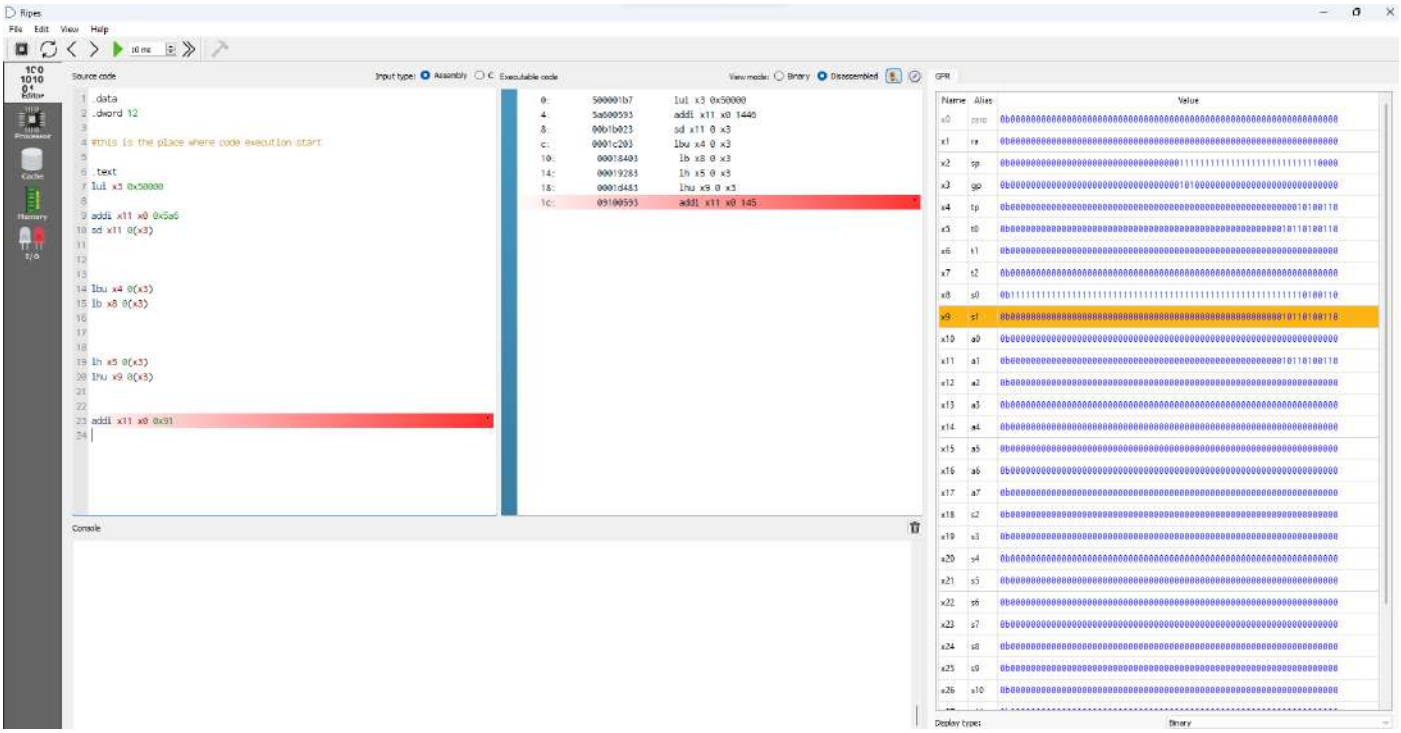
Practice Problems

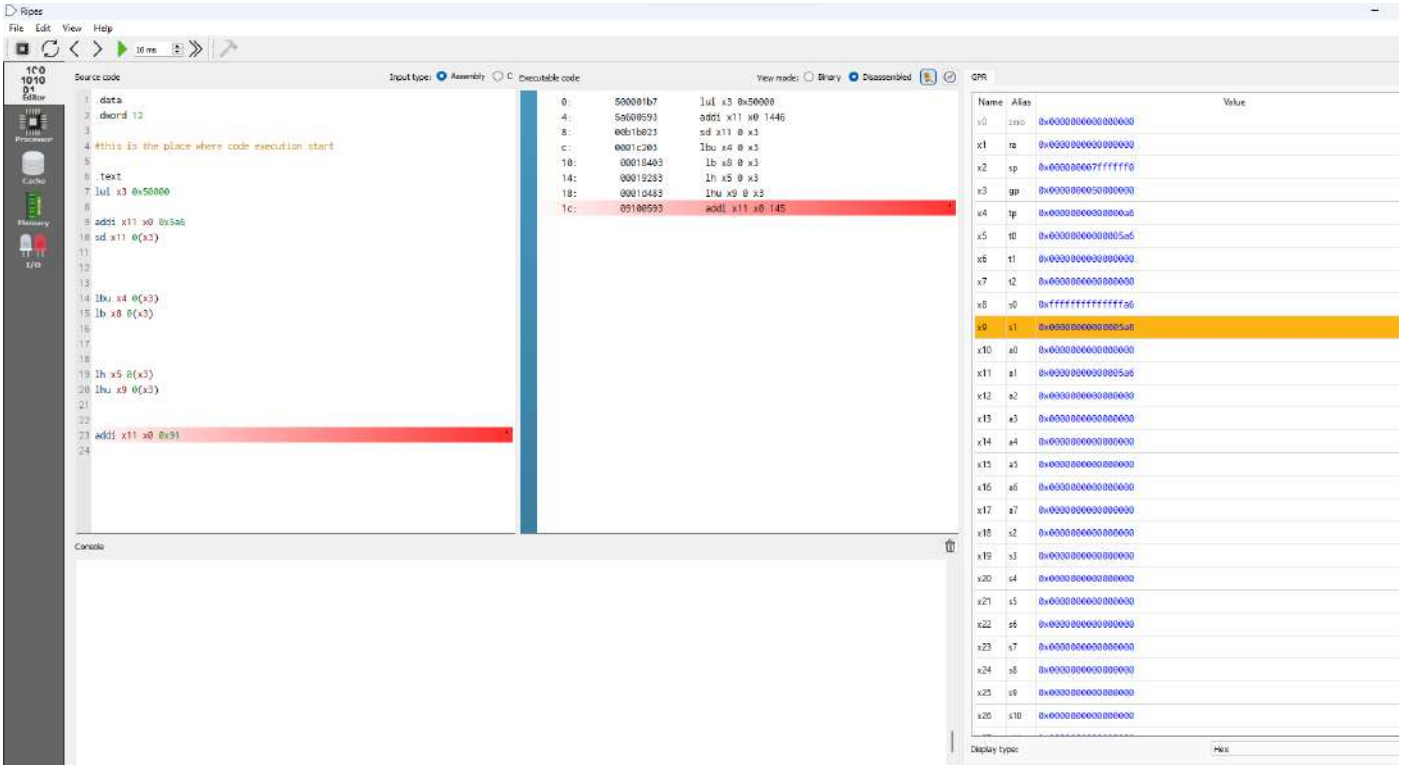
- Multiply two numbers by repeated addition
- Divide two numbers by repeated subtraction - find the quotient and remainder
- Find the largest among n given numbers
- Check if a given number is even or odd
- Element-wise sum of two 1-D arrays (**C** = **A** + **B**)
- Find the GCD (Greatest common divisor) of two numbers
- Transpose of a 2-dimensional array/matrix
- Generate bit-patterns using logical operations (and/or/shift)



you cant add more than 12 bit in addi instruction as constant


Ripes
File Edit View Help
Source code
Input type: Assembly C
Executable code
View mode: Binary Disassembled
GPR
Name Alias Value
.text
lui x3 0x50000
addi x11 x0 0x5e6
sd x11 0(x3)
lbu x4 0(x3)
lb x8 0(x3)
addi x11 x0 0x23
lh x5 0(x3)
lhu x9 0(x3)
addi x11 x0 0x51
sd x11 6(x3)
lbu x6 6(x3)
lb x10 6(x3)
addi x11 x0 0xcd
sd x11 2(x3)
lb x7 2(x3)
0: 500001b7 lui x3 0x50000
4: 5e600593 addi x11 x0 1446
8: 00b1b023 sd x11 0 x3
c: 0001c203 lbu x4 0 x3
10: 00018403 lb x8 0 x3
14: 02300593 addi x11 x0 35
18: 00019283 lh x5 0 x3
1c: 0001d483 lhu x9 0 x3
20: 05100593 addi x11 x0 145
24: 00b1b323 sd x11 6 x3
28: 0061c303 lbu x6 6 x3
2c: 00618503 lb x10 6 x3
30: 0cd00593 addi x11 x0 205
34: 00b1b123 sd x11 2 x3
38: 00218383 lb x7 2 x3
Console
Display type: Binary


Ripes
File Edit View Help
Source code
Input type: Assembly C Executable code
View mode: Binary Disassembled
GPR
.data
.dword 12
#this is the place where code execution start
.text
lui x3 0x50000
addi x11 x0 0x5a6
sd x11 0(x3)
lbu x4 0(x3)
lb x8 0(x3)
lh x5 0(x3)
lhu x9 0(x3)
addi x11 x0 0x91
0: 500001b7 lui x3 0x50000
4: 5a600593 addi x11 x0 1446
8: 00b1b023 sd x11 0 x3
c: 0001c203 lbu x4 0 x3
10: 00018403 lb x8 0 x3
14: 00019283 lh x5 0 x3
18: 00018483 lhu x9 0 x3
1c: 09100593 addi x11 x0 145
Name Alias Value
Console
Display type: Binary


Ripes
File Edit View Help
Source code
Input type: Assembly C Executable code
View mode: Binary Disassembled
GPR
.data
.dword 12
#this is the place where code execution start
.text
lui x3 0x50000
addi x11 x0 0x5a6
sd x11 0(x3)
lbu x4 0(x3)
lb x8 0(x3)
lh x5 0(x3)
lhu x9 0(x3)
addi x11 x0 0x91
0: 500001b7 lui x3 0x50000
4: 5a600593 addi x11 x0 1446
8: 00b1b023 sd x11 0 x3
c: 0001c203 lbu x4 0 x3
10: 00018403 lb x8 0 x3
14: 00019283 lh x5 0 x3
18: 00018483 lhu x9 0 x3
1c: 09100593 addi x11 x0 145
Name Alias Value
Console
Display type: Hex

RISC-V R-format Instructions

Remember Instruction will be always of 32 bit irrespective of architecture.

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

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

R-format Example

For all R format instruction opcode will remain same only funct3 & funct7 will very .of course operands also

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

Opcode have 7 bits it does not mean that 2^7 operation are there. Q. Then why it have been given so much of space?

Ans:-The point is it maybe utilize in the future suppose if user want to add its own set of operation then it can be added and according to the need software are being modified and more operations are coming

```
add  x9,x20,x21
```

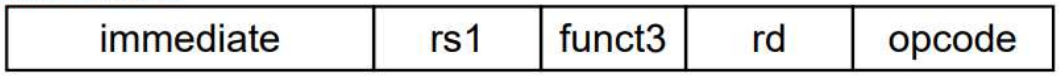
0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

note that assembly code and binary code have one to one Mapping

RISC-V I-format Instructions

as this is immediate so 2nd operand will come in the form of immediate. Also funct7 will be same for all immediate instruction so no funct7



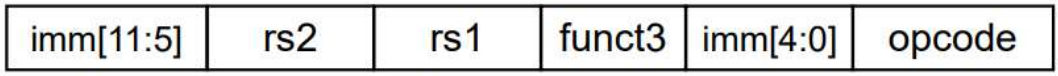
immediate is always considered as signed and also note that it cant take more than 12 bits so essentially range should be -2048 to 2047

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

RISC-V S-format Instructions

ex:- lw,sw

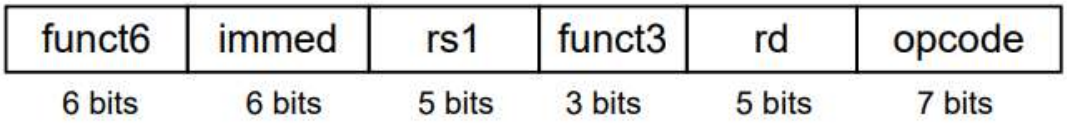
funct7 is still not present as it will be same here also



- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

As there is no storage in register as it will be stored in memory so field for rd(destination)

Shift Operations



- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by 2^i (unsigned only)

Note:- Irrespective of the type of architecture instruction will be of 32 bits in MIPS.

R-format Example

The field that denotes the operation and format of an Instruction.

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

add x9,x20,x21

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

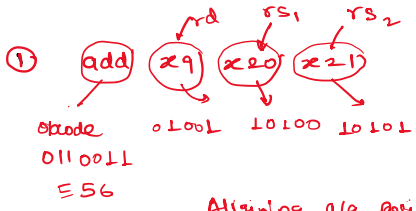
$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} =$
$015A04B3_{16}$

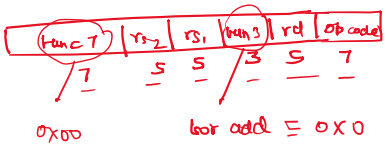
Examples: Ripes Simulator

Executable code View mode

```
0:                  015a04b3        add x9 x20 x21
4:                  3fe7ca13        xori x20 x15 1022
8:                  00310463        beq x2 x3 8 <L1>
c:                  005201b3        add x3 x4 x5

0000000000000010 <L1>:
   10:                  00000033        add x0 x0 x0
```





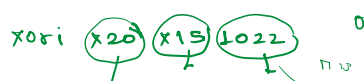
Aligning a/q position

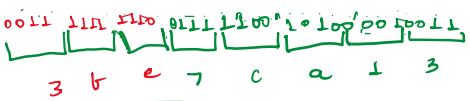
0X 015a04b3

Q2 xori

keep in mind this is case sensitive

xori x20 x15 1022





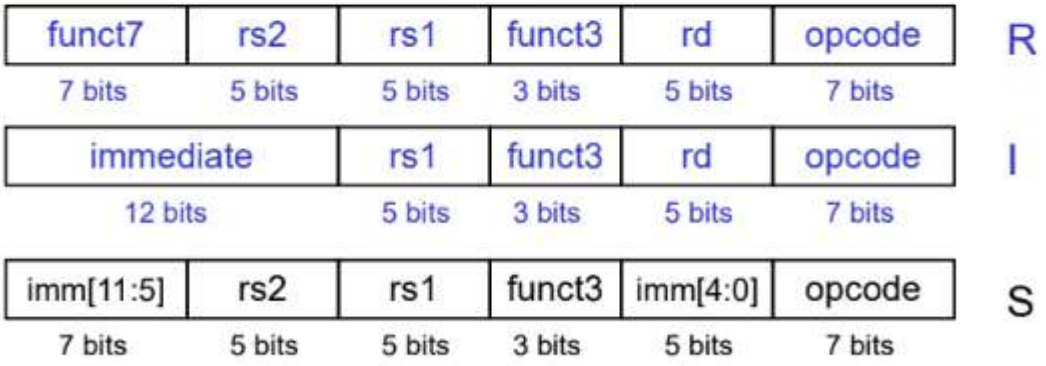
0X 3fe7ca13

I-format Examples

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

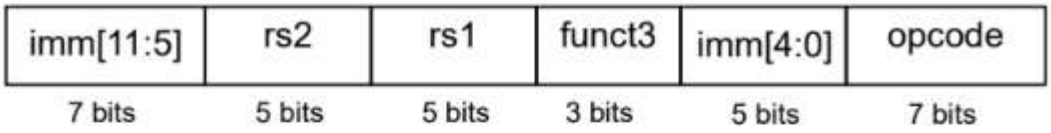
- addi x12, x3, -51
  - rd = x12 (01100), rs1 = x3 (00011)
  - imm = -51 (111111001101)
  - 111111001101 00011 000 01100 0010011
    immediate rs1 func3 rd opcode
  - 0xFCD18613
- ld x28, 100(x1)
  - rd = x28 (11100), rs1 = x1 (00001)
  - imm = 100 (000001100100)
  - 000001100100 00001 011 11100 0000011
    immediate rs1 func3 rd opcode
  - 0x0640BE03

RISC-V S-format Instructions



- Different immediate format for store instructions
  - sd rs2, imm(rs1)
  - rs1: base address register number
  - rs2: source operand register number
  - imm: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

S-format Example

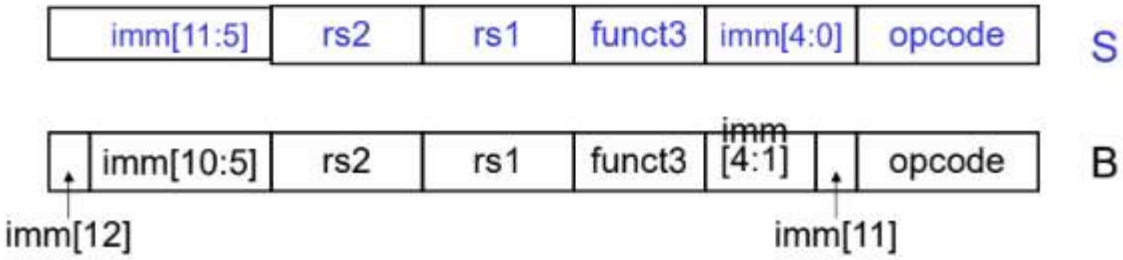


- sd x28, 100(x1)
  - rs1 = x1 (00001), rs2 = x28 (11100)
  - imm = 100 (0000011 00100)
  - 0000**011** **1**1100 **0000**1 011 **0010**0 010**0011**
    imm[11:5] rs2 rs1 funct3 imm[4:0] opcode
  - 0x07C0B223

0	0000	**4**	0100	**8**	1000	**c**	1100
1	0001	**5**	0101	**9**	1001	**d**	1101
2	0010	**6**	0110	**a**	1010	**e**	1110
3	0011	**7**	0111	**b**	1011	**f**	1111

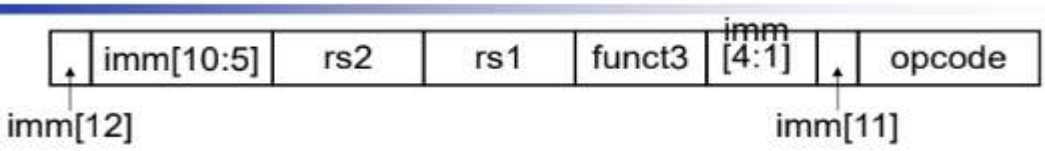
Branch Addressing

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



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

B-format Example



- beq x28, x1, -96
  - rs1 = x28 (11100), rs2 = x1 (00001)
    12 11 10:5 4:1 0
  - imm = -96 (1 1 111101 0000 0)
  - **1** **111101** **00001** 11100 **000** 0000 **1** **1100011**
    imm[12,10:5] rs2 rs1 funct3 imm[4:1,11] opcode
  - 0xFA1E00E3

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

Decoding Machine Code

- Convert 0x00578833 to assembly code
  - 0000 0000 0101 0111 1000 1000 **0011 0011**
  - Determine opcode: 0110011 (R-type)
  - Match bits to their fields

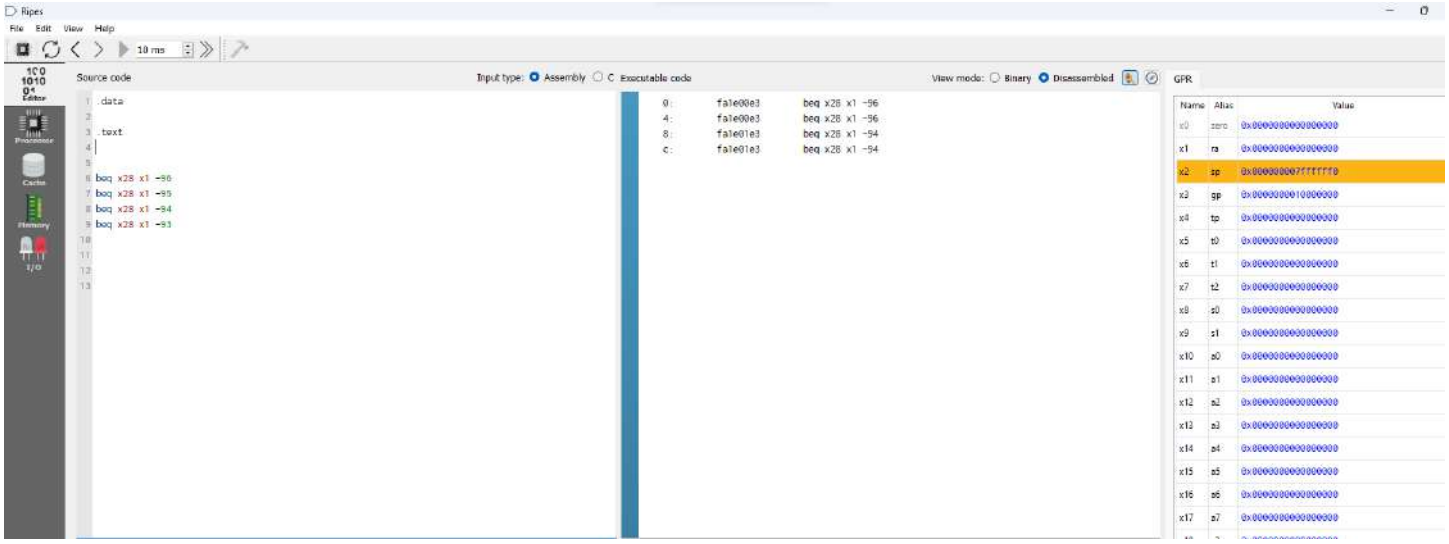
funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

  - add x16, x15, x5

In the above question:-

what if :→ beq X28 X1 -95. → Does it same answer as only immediate is changing in one bit and we are not taking imm[0] anyway.

Ans:- yes.



Summary of Instructions

- Arithmetic: add, sub, addi,
- Logical: and, or, xor, andi, ori, xori
- Data movement: ld/w/h/b, sd/w/h/b
- Shift: sll, srl, sra, slli, srli, srai
- Branch: beq, bne, bge, blt, bgeu, bltu
- Jump: jal, jalr
- Others: lui, auipc

Pseudo Instructions

- Enables simpler assembly, but not implemented in machine code
- Translated to other instructions
  - e.g., mv rd, rs is a pseudo instruction
    addi rd, rs, 0 is actual implementation
  - neg rd, rs == sub rd, x0, rs
  - j offset == jal x0, offset
- Better to use real instructions for a 1-1 correspondence with machine code

NOTE: pseudo instruction is supported by RISC-V so any simulation should also support this .pseudo means in machine code it will not directly get implemented first it get converted into general instruction then get executed

Means processor will not directly execute this

W hat this instruction will do in mips beq x28 x1 -96

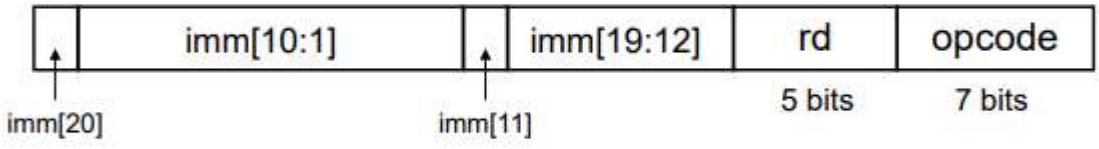
In this case, the instruction is saying, "Branch to a destination that is -96 instructions away if the ff`

So, if the values in registers `x28` and `x1` are equal, the program will jump to an instruction 96 instructions before this `beq` instruction. If they are not equal, it will continue executing the next instruction in sequence.

The 12-bit immediate field in branch instructions represents a signed offset in the range of -2048 to +2047, meaning it can address a range of approximately ±2047 instructions from the current instruction. If you specify a branch with a label like "beq x28, x1, end," the assembler will calculate the offset from the current instruction to the label "end" and ensure that it fits within the 12-bit immediate field. If the label is within this range, it will work correctly without any issue.

Jump Addressing

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



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

RISC V and MIPS are two different assembly language .In the book the field is given of MIPS not RISC V.

Ripes simulator works on the RISC-V assembly language. The point should be noted that RISC V is open source ISA that uses RISC V assembly language. Name of ISA and assembly language is same here .

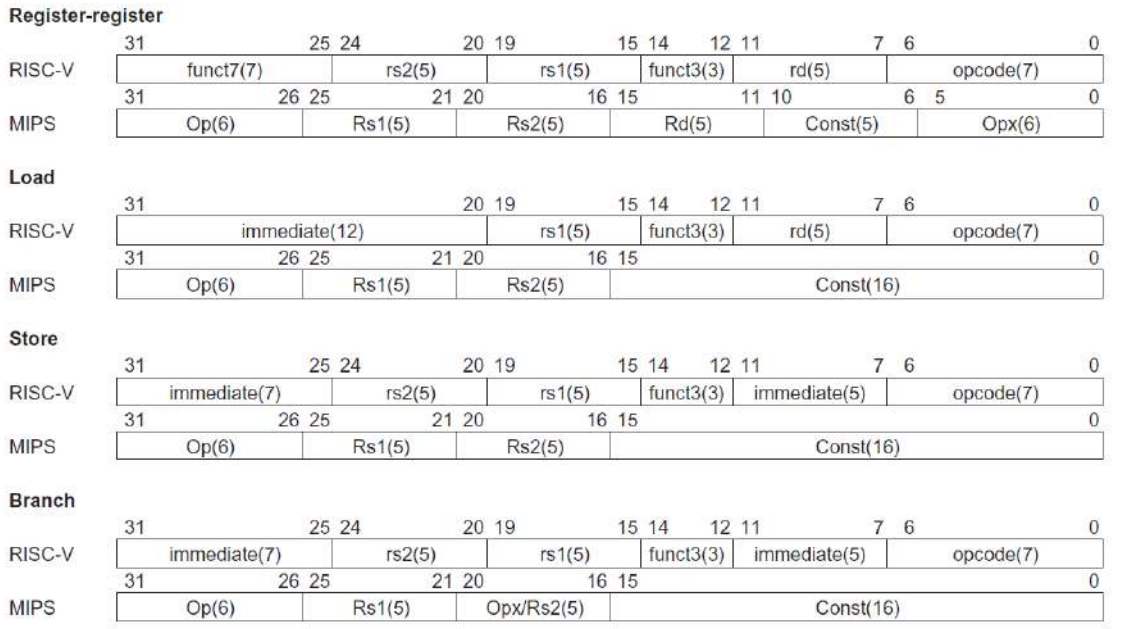
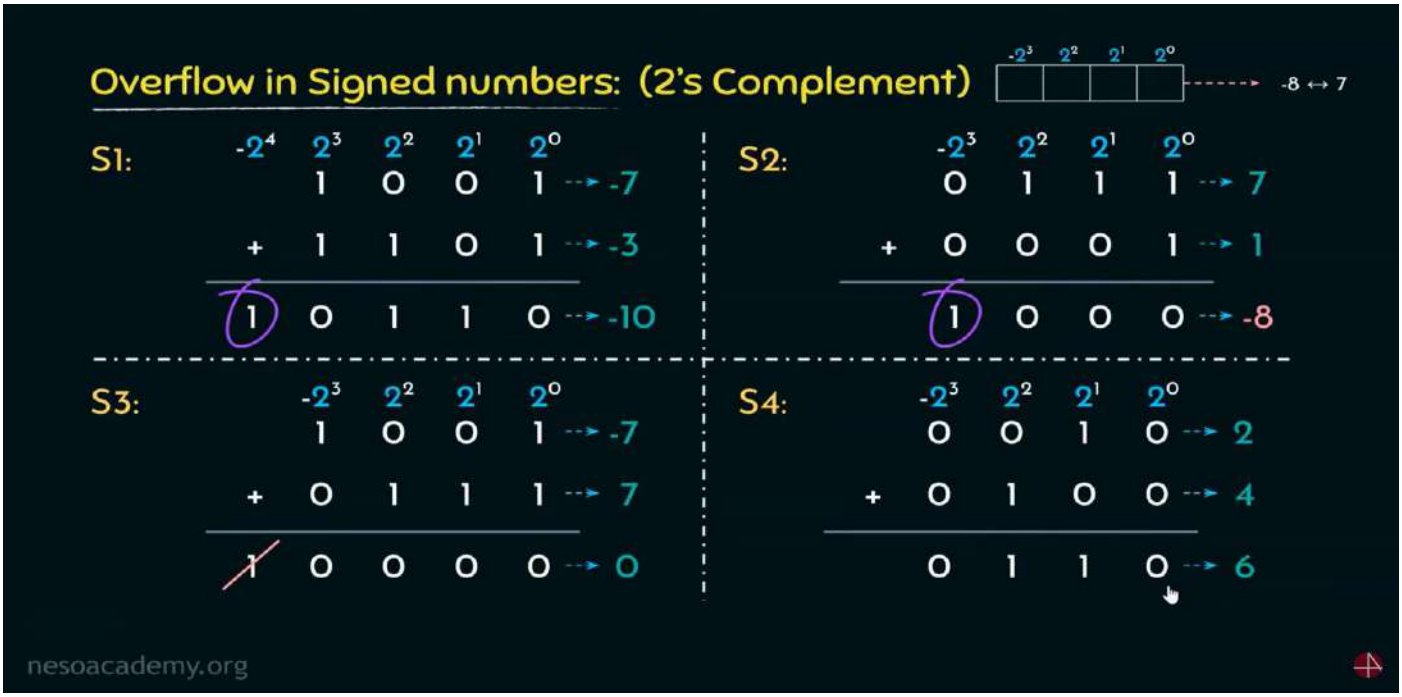


FIGURE 2.29 Instruction formats of RISC-V and MIPS. The similarities result in part from both instruction sets having 32 registers.

Overflow in signed integer



In s1 and s2, overflow is happening while s3 and s4 no overflow is happening.

There are three cases we can think upon

1.add 2.addi 3.sub

Assumption is source operand does not overflow itself. It means input should be provided in range only either it is register or immediate.

1.add: - overflow can only occur if both the source operand will be of same sign. If It is different then no worry.

2.addi: - similar cases as above if imm and source operand have same sign then only overflow will takes place

3.sub: - if two source operand have different sign then only overflow may takes place.

How to get know overflow?

2.4 Integer Computational Instructions

Most integer computational instructions operate on XLEN bits of values held in the integer register file. Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register *rd* for both register-immediate and register-register instructions. No integer computational instructions cause arithmetic exceptions.

arithmetic exception means arithmetically giving wrong answer like overflow

We did not include special instruction-set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using

---

RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: `add t0, t1, t2; bltu t0, t1, overflow`.

For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: `addi t0, t1, +imm; blt t0, t1, overflow`. *This covers the common case of addition with an immediate operand.* similarly, if imm is negative we can use bge

For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

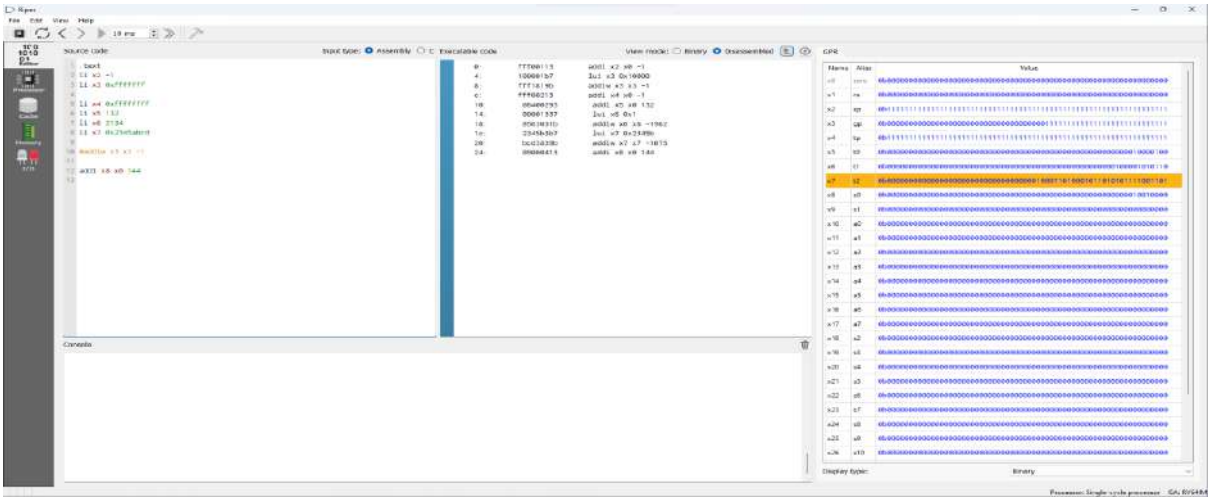
In RV64I, checks of 32-bit signed additions can be optimized further by comparing the results of ADD and ADDW on the operands.

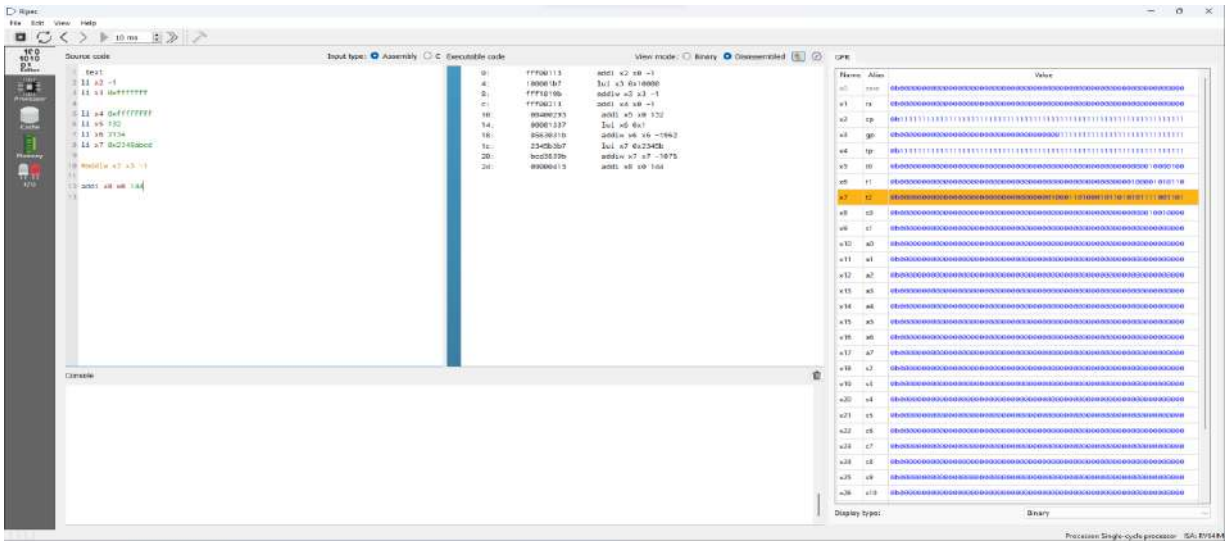
How slt and slti works?

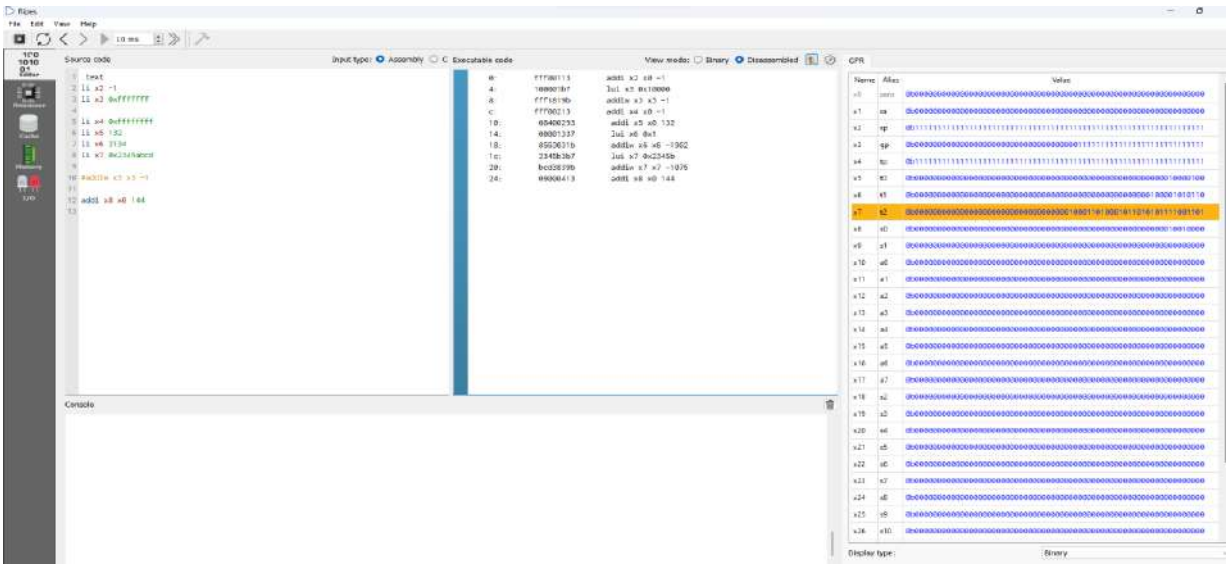
he SLT instruction performs a signed comparison between the values in rs1 and rs2. It sets the value of rd to 1 if rs1 is less than rs2, and it sets rd to 0 if rs1 is greater than or equal to rs2.

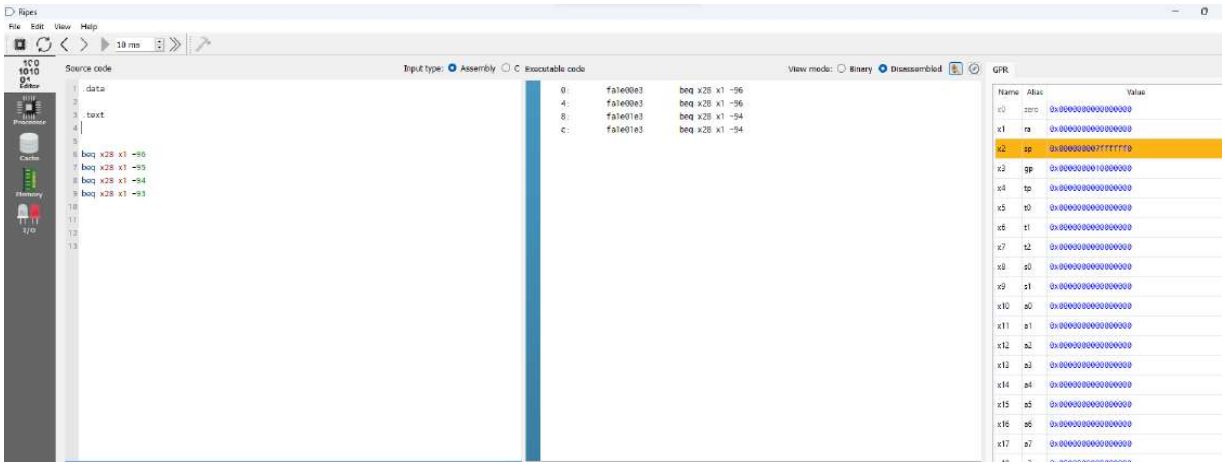
If the value in rs1 is less than the value in rs2, then rd is set to 1.

If the value in rs1 is greater than or equal to the value in rs2, then rd is set to 0.









Procedural calling

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

Execution of a procedure, the program must follow these six step:--

1.put parameter in a place where the procedure can access them.

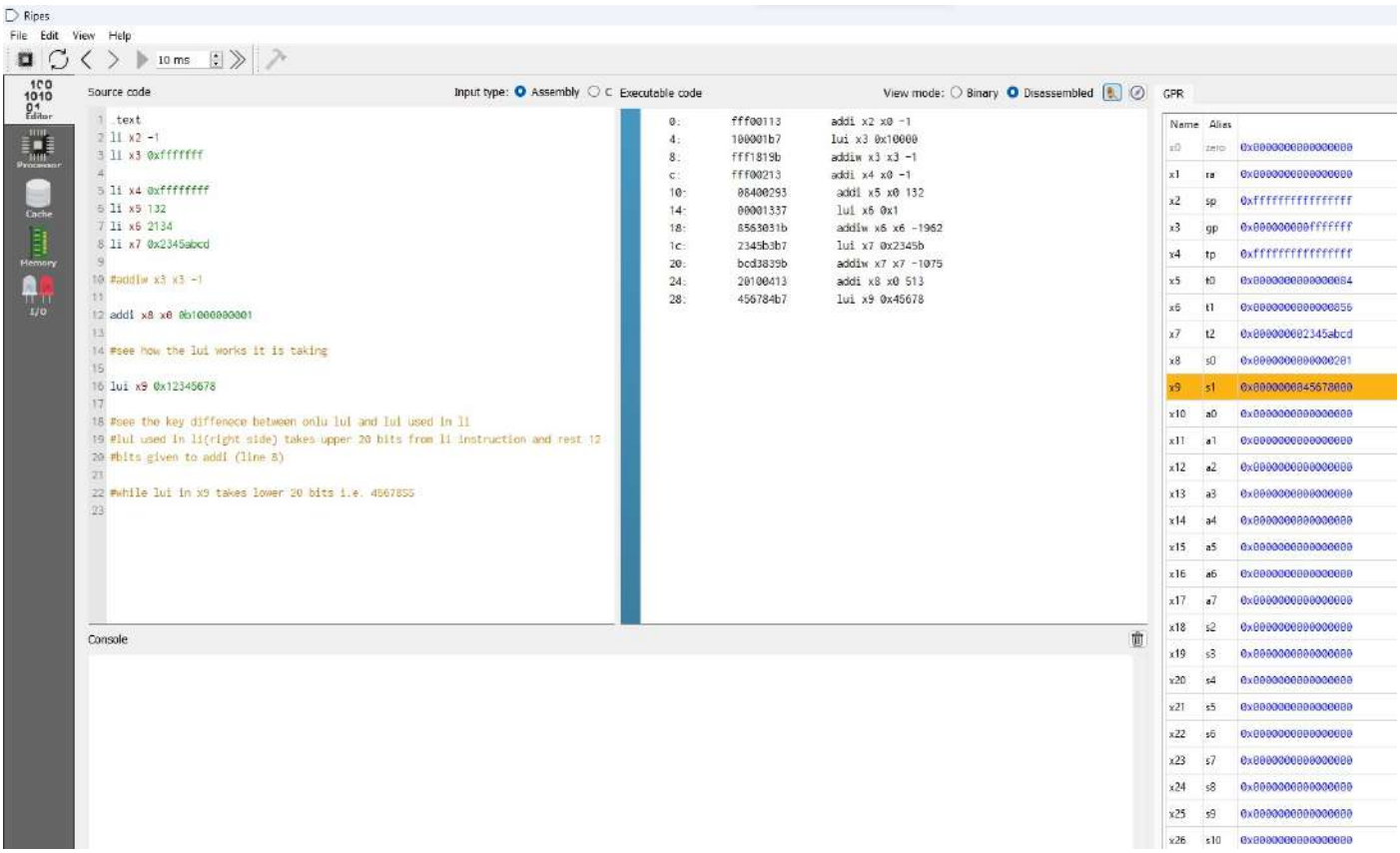
2.Transfer control to the procedure.

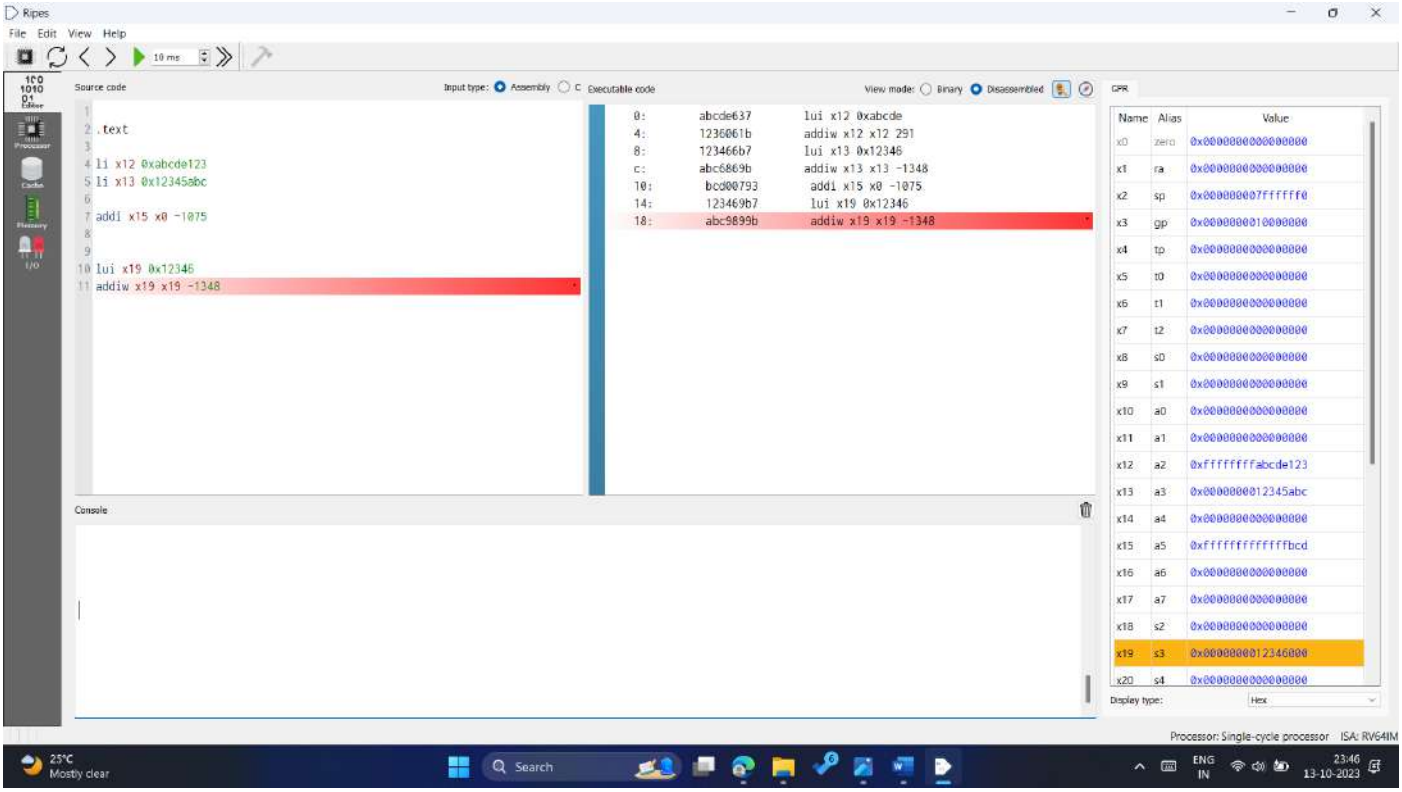
3.Acquire the storage resource needed for the procedure.

4.Perform the desired task.

5.put the result (if any) in a place where the calling program can access it.

6.Return control to the point of origin.





See carefully in the case of 0x12345abc it is lui x13 0x12346 not 0x12345 because we are adding 0xabc and 0xabc in 32 bit 0xfffffabc thus fffff at the beginning is nothing but -1 so 6-1 get converted to 5.,kjmh-

x10–x17: eight parameter registers in which to pass parameters or return values.

. x1: one return address register to return to the point of origin

jump-and-link instruction: - An instruction that branches to an address and simultaneously saves the address of the following instruction in a register (usually x1 in RISC-V).

how the label will work, or function will work. As it is recognized by the instruction?

As this is label so identified by Assembler and fetched to the address next to label

it must be saved somewhere in register as we are not loading so it can't be saved in memory. Jal will jump to the address of procedure and saves the next instruction where this procedure is called. x1 has no relation with the procedure material

return address:- A link to the calling site that allows a procedure to return to the proper address; in RISC-V it is stored in register

caller:- The program that instigates a procedure and provides the necessary parameter values.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

```
int main()          void func (int x)
{                   {
  func(x);            do sth;
  return 0;         }
}
```

In this example main() function is caller and func is callee.

- Caller can become caller in nested procedure.

- A register is allocated that will hold the address of current instruction. The register is known as program counter (PC).

- Remember, PC is a special register not general purpose register.

Jal x_1, Procedure ← label of the func. / procedure

Jalr. x_0, $0(x_1)$. Remember x_1 contain PC+4 to fetch next instruction after completing of procedure. $0(x_1)$ is nothing but address at x_1.

- more sensible name of PC would be instruction address register.

when we have completed the procedural call then we don't want to save following program counter of the procedural call. we have actually saved the address in register x1 we only want to execute the instruction set stored in x1 so we will use jalr. that works on the instruction set itself instead of the label but it will also try save the following pc of procedural so we use x0 as cant be overwritten as we don't want that .we only concern about pc of main or caller not the callee when it completed its work. we can use any other register instead of x0 also .it will not make any changes in execution but will occupy an extra unnecessary register .so jump and link can also be sued to perform an unconditional branch withon a procedure by using x0

- x_5 is used as alternate link address.

```
{   Jal x0, Label // unconditionally branch to Label
```

S'.

Using more register

we have seen that x10 to x17 is used to pass parameter but what we have more than 8 parameter? and what if any register needed by caller contains some data, in that case we have to restore -a the value to memory, we have to spill register to memory.

- the ideal data structure for spilling register is stack that works on LIFO (last in first out principle) so stack pointer that is register x_2, also known by the name sp is used to store the address on stack.

- Stack in RISC-V for spilling used as contiguous memory. we know stack can be implemented with array as well as linked list so, it can both contiguous & Non-contiguous generally.

stacks “grow” from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer.

Compiling a C Procedure That Doesn't Call Another Procedure

Example

Let's turn the example on page 66 from Section 2.2 into a C procedure:

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

What is the compiled RISC-V assembly code?

Answer

The parameter variables `g`, `h`, `i`, and `j` correspond to the argument registers `x10`, `x11`, `x12`, and `x13`, and `f` corresponds to `x20`. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example on page 67, which uses two temporary registers (`x5` and `x6`). Thus, we need to save three registers: `x5`, `x6`, and `x20`. We

“push” the old values onto the stack by creating space for three doublewords (24 bytes) on the stack and then store them:

```
addi sp,  sp, -24  // adjust stack to make room for 3 items
sdx5, 16(sp)  // save register x5 for use afterwards
sdx6, 8(sp)   // save register x6 for use afterwards
sdx20, 0(sp)  // save register x20 for use afterwards
```

Figure 2.10 shows the stack before, during, and after the procedure call.

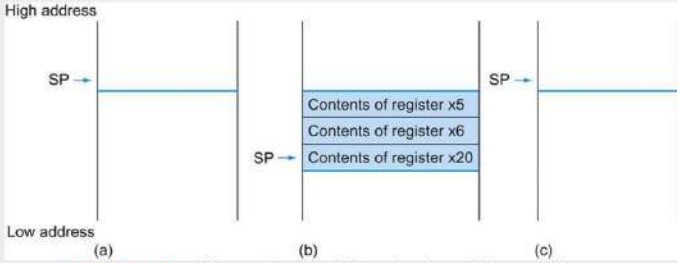


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call.
The stack pointer always points to the “top” of the stack, or the last doubleword in the stack in this drawing.

The next three statements correspond to the body of the procedure, which follows the example on page 67:

```
add x5, x10, x11// register x5 contains g + h
add x6, x12, x13// register x6 contains i + j
sub x20, x5, x6// f = x5 – x6, which is (g + h)−(i + j)
```

To return the value of `f`, we copy it into a parameter register:

```
addi x10, x20, 0 // returns f (x10 = x20 + 0)
```

Before returning, we restore the three old values of the registers we saved by “popping” them from the stack:

```
ld x20, 0(sp) // restore register x20 for caller
ld x6, 8(sp) // restore register x6 for caller
ld x5, 16(sp) // restore register x5 for caller
addi sp, sp, 24  // adjust stack to delete 3 items
```

The procedure ends with a branch register using the return

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

since the caller does not expect registers x5 and x6 to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore x20, since the callee m

ust assume that the caller ne

Nested procedure:

procedure that call other procedure is said to be nested procedure

Recursive procedure is the tybe of nested procedure that called its "colon" each time.
↳ means same procedure.

Compiling a Recursive C Procedure, Showing Nested Procedure Linking

Example very good problem

Let's tackle a recursive procedure that calculates factorial:

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

What is the RISC-V assembly code?

Answer

The parameter variable `n` corresponds to the argument register `x10`. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and `x10`:

```
fact:
  addi  sp, sp, -16// adjust stack for 2 items
   sd x1, 8(sp)// save the return address
   sd x10, 0(sp)// save the argument n
```

we are not deleting from memory we are only moving the stack pointer.this is one of the reason why we get garbage value

this shows that we are using contiguous stack here.

The first time `fact` is called, `sd` saves an address in the program that called `fact`. The next two instructions test whether `n` is less than 1, going to `L1` if `n ≥ 1`.

```
addi  x5, x10, -1  // x5 = n - 1
bge  x5, x0, L1  // if (n - 1) >= 0, go to L1
```

If `n` is less than 1, `fact` returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in `x10`. It then pops the two saved values off the stack and branches to the return address:

```
addi  x10, x0, 1  // return 1
addi  sp, sp, 16  // pop 2 items off stack
jalr   // return to caller
```


Before popping two items off the stack, we could have loaded `x1` and `x10`. Since `x1` and `x10` don't change when `n` is less than 1, we skip those instructions.

If `n` is not less than 1, the argument `n` is decremented and then `fact` is called again with the decremented value:

```
L1: addi x10, x10, -1  // n >= 1: argument gets (n-1)
  jalx1, fact  // call fact with (n-1)
```

The next instruction is where `fact` returns; its result is in `x10`. Now the old return address and old argument are restored, along with the stack pointer:

```
addi  x6, x10, 0  // return from jal: move result of fact(n -
1) to x6:
ld  x10, 0(sp)  // restore argument n
ld  x1, 8(sp)  // restore the return address
addi  sp, sp, 16  // adjust stack pointer to pop 2 items
```

Next, argument register `x10` gets the product of the old argument and the result of fact(n - 1), now in `x6`. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
mul  x10, x10, x6  // return n * fact (n-1)
```

Finally, `fact` branches again to the return address:

```
jalr  x0, 0(x1)  // return to the caller
```

eds its value.

Hardware / Software Interface

- A C variable is generally a location in storage its interpretation depends upon
  1. type → eg integer, character.
  2. storage class
     - Automatic
     - Static
- Static variable exist even the procedure ends. It remain till the program execution and retains its value at different calls of function.

Introduction

CPU performance factor

- Instruction count → this is decided by compiler as well as ISA or processor.
- CPI & cycle time → determined by CPU Hardware. Good hardware tries to minimise cycle time and CPI.

Two RISC-V implementation

- A simplified version
- A more realistic pipelined version.

memory reference → ld, sd

Arithmetic / logic → add, sub, or

control or transfer → beq, bit

Logic basic design

positive logic system:- low voltage = 0

high voltage = 1

negative logic system:- low voltage = 0

high voltage = 1

- one wire is attached to each bit and multi-bits are determined by multi wire connected system known as Buses.

Combinational element

- Operate on data.
- output is function of input.
- output only depends upon present input

eg: adder

$$\begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array}$$

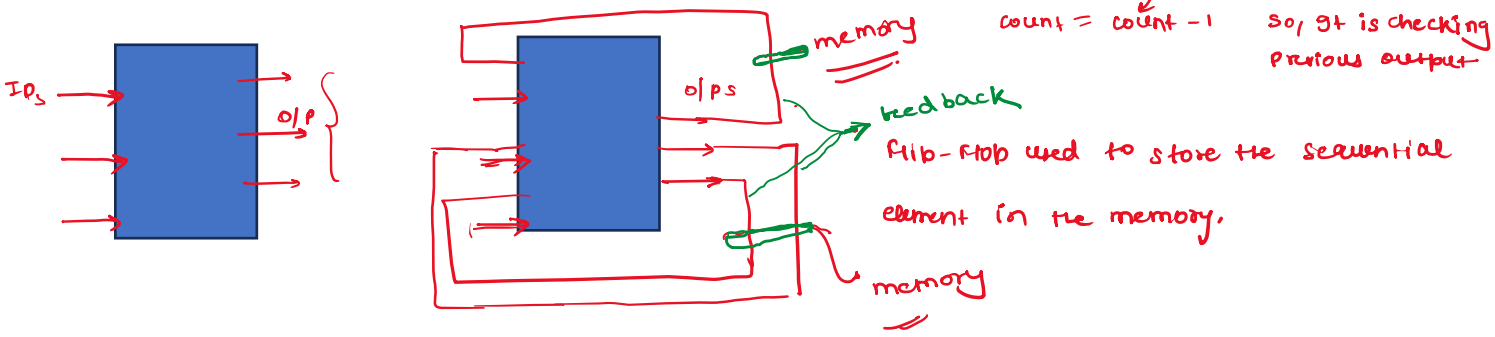
State (sequential element)

- store information
- output depends on the present o/p as well as previous o/p's.

ex:- counter it will check the previous value and will then do.

count = 5

count = count - 1 So, it is checking previous output



flip-flop used to store the sequential element in the memory.

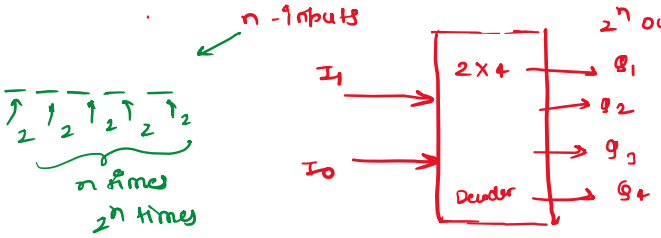
- AND, OR, NOT, XOR, NAND, NOR
- Decoders ; Multiplexers
- Adders , comparators

} Combinational element

Decoders

- Pattern matching

As everything connected inside by wire and thus decoder will help that which wire get operated at given instruction.



$n = 2$ (2 inputs)

where $I_1 I_0 = 00$

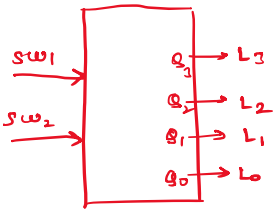
$= 01$

$= 10$

$= 11$

Suppose we are giving some instruction input then decoder will check your opcode and align following function.

Ex :- blow led as per switch values



- we may not need always all pattern output. so we could have simpler decoders like opcode decoding.

As each wire carry one bit and we know that first 7 bit contain opcode. so we will chose corresponding wire of those 7 bits and decode the opcode. Remember sequence as decoder matters as it should be align

- if we have to know two input line so two decoders will be required and thus cost will also be less.

eg: when opcode = 0110011 → R format
0010011 → I format etc

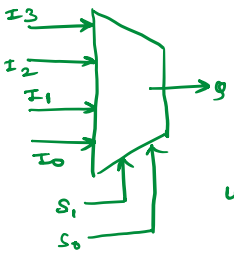
eg: funct 3 decoding is also done by decoder.

- There is the difference between decoder and multiplexer Decoder decodes all valid input and then somewhere while selector chooses the following instruction based on what decoder has given.

Multiplexer (MUX)

n lines should be there for n bits in the input combination of input that can be made

n select lines, 2^n inputs, one output

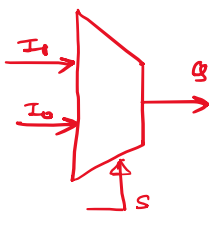


when $S_1 S_0 = 00$ $g = I_0$

$= 01$ $g = I_1$

$= 10$ $g = I_2$

$= 11$ $g = I_3$

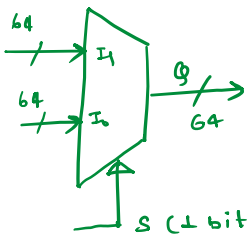


when $S = 1$, $g = I_1$ else $g = I_0$

S is used to select among two inputs I_1 / I_0

ex:- selector will chose you have to do add or addi instruction after the decoders will give opcode.

Input I need not be 1 bit, but a multibit known as "bus"



⟋ This the notation of bus

when $S=0 \Rightarrow g[63:0] = I_0[63:0]$

$S=1 \Rightarrow g[63:0] = I_1[63:0]$

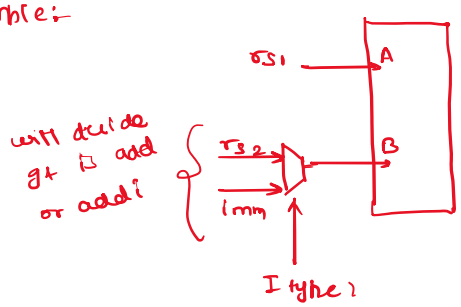
S can be used to select or route an entire entire group of input to output

g where are multiplexers used?

Ans- As only one bit there to chose so, each bus will come as one set Say I_1 and multiplexers are applied after I_1 or I_2.

My question is there must be function or sth to convert those buses into unique code. what is that.

other example:-



A is always rs_1

B can be rs_2 or immediate based on R/I type.

(it will check either it is I type or not

kihjea

Book reading

Simplicity favours regularity

In this we will talk about two type of instruction

1. memory reference Instruction : load & store
2. Arithmetic logic instruction : add, sub, and, or
3. Branch instruction : beq

other instruction has same functioning like these instruction.

- All instruction does these two steps
  1. Read the memory and fetch the instruction
  2. load one or two register according to the need.

- general purpose microprocessors
  processor used to do variety of work
  eg: used in pc.
- Embeded microprocessor
  processor used to do dedicated work
  eg: used in automobiles, washing machine

- All instruction uses ALU
  1. memory reference → to calculate the memory addres by checking offset.
  2. arithmetic - logic → to do for operation execution
  3. conditional branch → for equality test. or determine the next instruction address.
- If instruction type is ld/sd then result of ALU (address) is used to load the value from register or store the value in register.
- Branch require the use of the alu output will go to adder of ALU to add offset or adder of pc that simply increses by 4 is condition is not satisfied.
- In Arithmetic logic instruction result of ALU must be written in register.

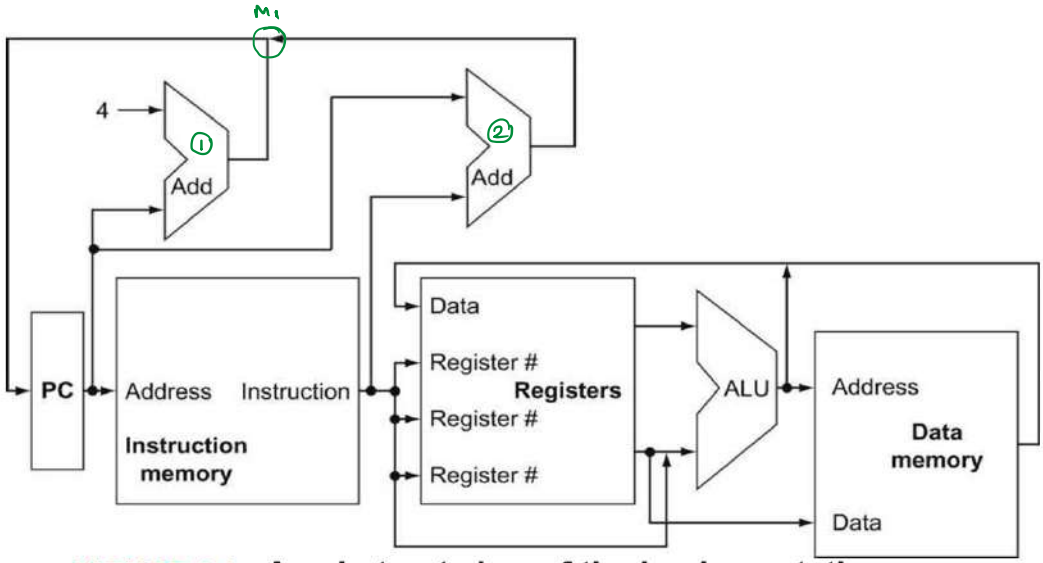


FIGURE 4.1 An abstract view of the implementation of the RISC-V subset showing the major functional units and the major connections between them.

First of all our current pc will fetch the data from instruction memory that contains the 32 bit instruction. It will decodes its instruction and execution will take place after There are two adder to change pc. 1st one is obvious and after every execution of instruction it will increse by +4. 2nd adder will take two values. first one is pc and second one is offset of branch instruction label that already gets decoded in Instruction memory. Both value will then added by adder. Now one control line will come and get added to two adders output that is actually connected with multiplexers. The control line contain ANDs gate where two wire (c_1 & c_2) come. one line come from (c_2) ALU and another line come from control which further come from instruction memory output (c_3). c_3 will decide the instruction is branch or not and same conveyed by c_2

At the AND gate also, 2nd input of AND will come from ALU that tells the condition is true or false because this will decide you have to jump to the different address or not.
↳ (known as 0 output of ALU)

In register section three register and data is there. this data is either the output of ALU (in case of arithmetic-logic) and output of memory in case of load instruction. For selecting between these 2, a multiplexure (M_2) is there. that is controlled by control line 4 that is connected to control. and will decide type of instruction

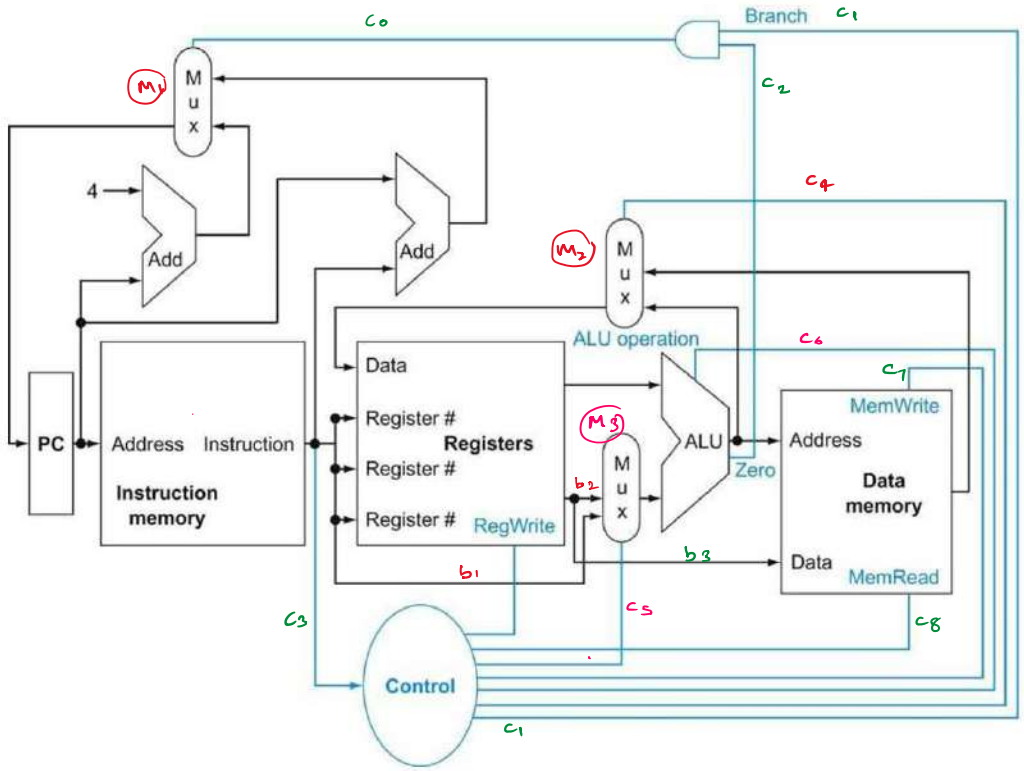


FIGURE 4.2 The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines.

One register from Registers section is always going to ALU but another register is getting decided by MUX (M_3). If instruction does not have immediate then value is in register and it will go to ALU but when it contains immediate then it will directly go from Instruction memory to MUX by b_1. This mux is connected to control line c_5 that decides 'immediate is there or not.

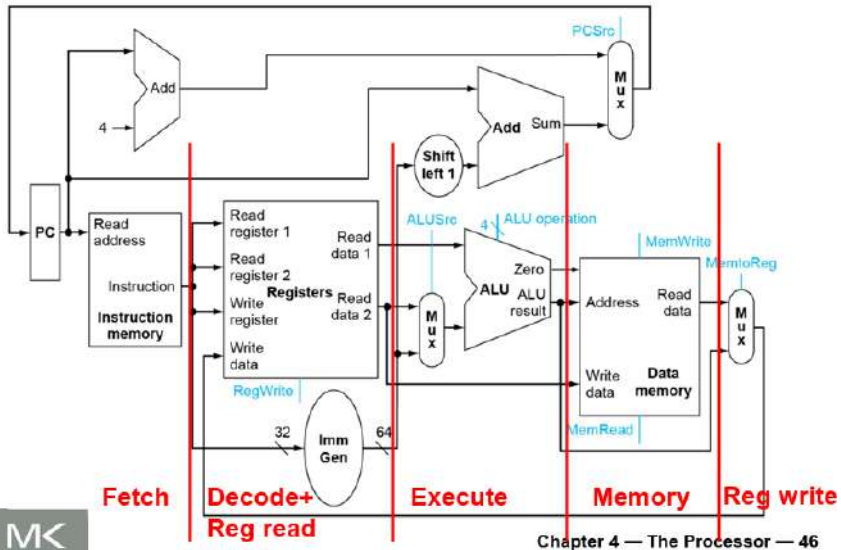
ALU is itself connected to control by c_6 that decides the type of operation to be done on operands coming. Result of ALU either go to Registers in case of arithmetic logic instruction, or it will go to memory in memory reference instruction for load or store or do nothing only send control signal c_2 in case of branch instruction.

Now, in data memory two control wires c_7 and c_8 is coming that will decide we have to read or write in Data memory. There is wire bus b_3 that is connected from Registers to data memory. This will be used in sd instruction when R_1 register value have been stored in memory. This is control by MemWrite control line c_1.

- The selection between wire bus is determined by device called multiplexor, although this device might better be called a data selector.

- In this first design, every instruction begin execution on one clock edge and completes execution on the next clock edge. but this approach is not practical since clock cycle must be severly streched to accomodate the longest instruction.

Multi-cycle Datapath



Five processing steps

IF: Instruction fetches from memory

ID: Instruction decode & register read

EX: Execute operation or calculate address

MEM: Access memory operand

WB: Write result back to register.

Logic design convention

Data path consist of two logical element

① combinational element → element operates on data value → given the same input combinational element gives the same output as it has no internal storage.
ALU is example of combinational element given the input will give same output As it have no internal storage.

② State element → element that operates on state

These element have storage because if we pull the power plug on the computer, we could restart it accurately
ex:- Instruction, data memory, as well as register.

- A state element has atleast two inputs and one outputs.
  - Input
    - Data value to be written.
    - clock → determine when the data value is written
- A state element can be read any time

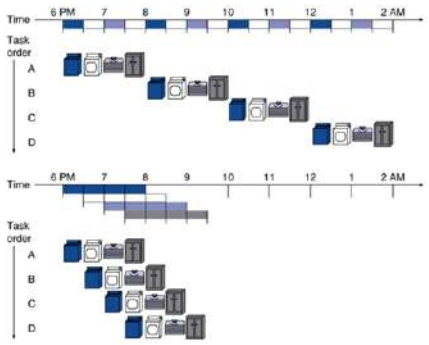
Output of the state element provide the value that written in earlier clock cycle.

- Logical component that contain state are also called Sequential as their output depends on both their inputs and the contents of internal state.

- RISC-V uses flip flops, memories and register as state element.
  D type flip flop is used by RISC-V which contains exactly two inputs (a value and clock) and one output.

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

Now see we have partitioned but we have not changed the time between rising edge of clock so it will be latency time and it will not improve time because 2nd instruction will not come until and unless the time does not change so we will use pipelining that will take the idea to divide 5 cycles and also change the time

So time required to execute next cycle will depend on the slowest cycle of the 5 cycles as you cant set the data until and unless each cycle has completed till then

See this as airport that have different counter .first people will go to counter 1 then it will go to counter 2 and counter 1 is vacant and another person can occupy that place similarly the line will be going on and after 5 counter finally the first person is done with everything and thus 1 instruction get completed.so until and unless next part is counter is clear no one previous to that is allowed. For simplicity we will take this max time as clock rising edge to favour simplicity.

A each cycle is of 0.5 and so total time is

=latency time + (no of stages -1)*time of one cycle =2 + 3*0.5= 3.5

Total number of cycle = p + (N-1) ← Number of stages
↓ Number of instruction

For larger p
total number of cycle ≅ p
as no of stage will be small only.

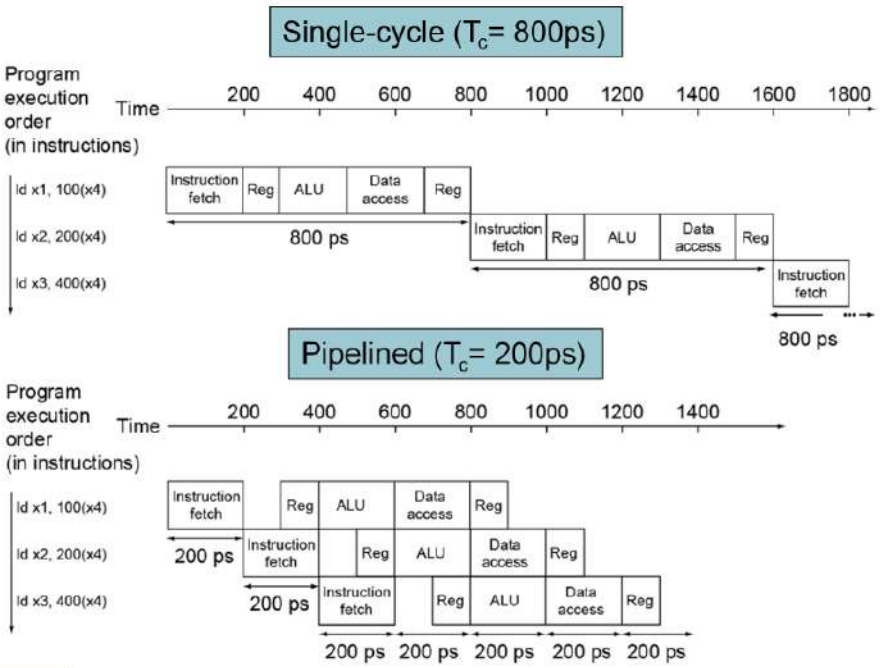
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

If you have few instruction say 1
① Time without pipelining →
Here, max clock cycle = 200ps and thus it will decide Latency time.

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease even it will increase as we are chosing maximum cycle one and multiplying with no of stages