**A logo for a university

Description automatically generated**

**PES UNIVERSITY**

**(Established under Karnataka Act No. 16 of 2013)**

**100 Feet Ring Road, BSK III Stage, Bengaluru-560085**

**Department of Electronics and Communication Engineering**

**Course Title: RISC-V ARCHITECTURE**

**Course Code: UE21EC352A**

**Teacher: PROF. VANAMALA H R**

**Project Title:**

**Understanding risc-v Assembly:**

**LONGEST COMMON SUBSEQUENCE**

**Done By:**

**AKASH RAVI BHAT**

**PES1UG21EC025**

**The Longest Common Subsequence (LCS) problem has various real-life applications, and one notable example is in bioinformatics, particularly in the field of DNA sequence analysis.**

1. **DNA Sequence Alignment:**
   * **In bioinformatics, scientists often deal with the comparison of DNA sequences from different organisms or individuals. The LCS algorithm can be applied to find the longest common subsequence between two DNA sequences. This information is valuable for understanding genetic similarities and differences between species, studying evolutionary relationships, and identifying common genetic patterns.**
   * **For example, when comparing the DNA of two individuals or species, identifying the longest common subsequence helps highlight regions of similarity in their genetic makeup. This can be crucial for identifying shared genes, understanding evolutionary relationships, and studying the genetic basis of diseases.**
   * **LCS algorithms are employed in bioinformatics tools for sequence alignment, where the goal is to find the optimal alignment between two biological sequences, such as DNA or protein sequences. Identifying the longest common subsequence aids in understanding genetic variations, mutations, and functional similarities.**
2. **Version Control Systems:**
   * **In software engineering, version control systems (VCS) like Git use algorithms related to LCS to determine the differences between different versions of source code files. The LCS algorithm helps identify the longest common subsequence of lines or characters between two versions, and the differing elements are used to represent changes (additions, deletions, or modifications).**
   * **When you commit changes to a code repository, the version control system needs to efficiently identify the changes made compared to the previous version. LCS-based algorithms help in this process by identifying the common and differing parts of the code.**
   * **Efficiently computing the LCS is essential for minimizing storage space and optimizing the speed of operations in version control systems.**

**These applications highlight the versatility and significance of the Longest Common Subsequence problem in diverse fields, ranging from bioinformatics to software engineering. The problem-solving approach used in LCS algorithms has practical implications for solving complex sequence matching problems in various domains.**

1. **Plagiarism Detection:**
   * **In the field of academic and content-related writing, plagiarism detection systems use LCS algorithms to identify similarities between documents. By finding the longest common subsequence between two pieces of text, these systems can highlight potential instances of plagiarism or content reuse.**
2. **Speech Recognition:**
   * **In speech recognition systems, the LCS algorithm can be employed to compare spoken words or phrases. By finding the longest common subsequence, the system can identify similarities between the recognized speech and a reference vocabulary, aiding in accurate transcription.**
3. **Image Comparison:**
   * **Image processing and computer vision applications use LCS techniques to compare images. The problem can be applied to find the longest common subsequence of pixel values between two images, helping to identify similarities and differences. This is useful in tasks like image recognition and content-based image retrieval.**
4. **Network Traffic Analysis:**
   * **In cybersecurity, analyzing network traffic patterns is crucial for identifying potential security threats. The LCS algorithm can be used to compare sequences of network events or patterns, helping to detect anomalies, intrusions, or malicious activities.**
5. **Recommender Systems:**
   * **In recommender systems for e-commerce or content platforms, LCS algorithms can be used to analyze user behavior sequences. By finding common subsequences of user interactions, these systems can recommend products or content based on users' historical preferences.**
6. **Automatic Summarization:**
   * **In natural language processing and text summarization, the LCS problem can be utilized to identify common phrases or sentences in a set of documents. This can help in generating concise and meaningful summaries by selecting the most relevant content.**
7. **Comparing Biological Sequences (Proteins):**
   * **In addition to DNA sequences, LCS algorithms are applied to compare protein sequences. Analyzing the longest common subsequence of amino acids in proteins is crucial for understanding protein structures, functions, and evolutionary relationships.**
8. **Code Clone Detection:**
   * **In software engineering, code clone detection involves identifying duplicated or similar code segments in a software project. LCS algorithms can be used to compare code sequences and identify common structures, aiding in the detection of code clones.**

**These applications showcase the versatility of the LCS problem-solving approach in diverse domains, emphasizing its usefulness in analyzing and comparing sequences of various types. The underlying principles of LCS algorithms can be adapted to address challenges in different fields where sequence matching is a key task.**

**SOURCE CODE:**

**.data**

**.align 4**

**# test pattern**

**SequenceA: .string "AKASHRAVIBHAT"**

**SequenceB: .string "AKASHRGUT"**

**SASize: .word 13**

**SBSize: .word 12**

**str: .string "Found LCS length "**

**newline: .string "\n"**

**i: .word 0**

**j: .word 0**

**L: .word 1024**

**.text**

**.global \_start**

**\_start:**

**la a0 SequenceA**

**la a1 SequenceB**

**lw a2 i**

**lw a3 j**

**la a4 L**

**lw a5 SASize**

**lw a6 SBSize**

**jal lcs**

**#print str**

**la a0, str**

**li a7, 4**

**ecall**

**#print a0**

**mv a0 t0**

**li a7 1**

**ecall**

**#print \n**

**la a0, newline**

**li a7, 4**

**ecall**

**#The values in a7 are set to 1 for the "print integer"**

**#system call and 10 for the "exit" system call.**

**j end**

**lcs:**

**addi sp, sp, -4**

**sw ra, 0(sp) # return address**

**addi a2 a2 -1 #set i=-1 first**

**First\_for:**

**addi a2 a2 1 #i=i+1**

**bgt a2 a5 exit**

**addi a3 x0 0 #set j=0**

**Second\_for:**

**#jump to First\_for if j > SBSize**

**bgt a3 a6 First\_for**

**#jump to condition one if i=0 || j=0**

**beq a2 zero condition1**

**bne a3 zero condition2**

**condition1:**

**addi t0 a6 1**

**mul t0 t0 a2 #t0=(SBSize+1)\*i**

**add t1 t0 a3 #t1=(SBSize+1)\*i+j**

**slli t1 t1 2 #t1=((SBSize+1)\*i+j)\*4 RAW Hazard**

**add t1 a4 t1 #t1=((SBSize+1)\*i+j)\*4+a4**

**sw x0 0(t1) #L[i][j]=0**

**addi a3 a3 1 #j=j+1**

**beq x0 x0 Second\_for**

**condition2:**

**addi t2 a2 -1 #t2=i-1**

**addi t3 a3 -1 #t3=j-1**

**add t0 a0 t2 #t0=SequenceA+i-1**

**add t1 a1 t3 #t1=SequenceB+j-1**

**lb t0 0(t0) #t0=SequenceA[i-1]**

**lb t1 0(t1) #t1=SequenceB[j-1]**

**bne t0 t1 condition3**

**addi t0 a6 1**

**mul t0 t0 a2 #t0=(SBSize+1)\*i**

**add t1 t0 a3 #t1=(SBSize+1)\*i+j**

**slli t1 t1 2 #t1=((SBSize+1)\*i+j)\*4 RAW Hazard**

**add t1 a4 t1 #t1=[(SBSize+1)\*i+j]\*4+a4**

**addi t0 a6 1**

**mul t2 t0 t2 #t2=(SBSize+1)\*(i-1)**

**add t3 t2 t3 #t3-[(SBSize+1)\*(i-1)+(j-1)]**

**slli t3 t3 2 #t3=[(SBSize+1)\*(i-1)+(j-1)]\*4 RAW Hazard**

**add t3 a4 t3 #t3=[(SBSize+1)\*(i-1)+(j-1)]\*4+a4**

**lw t3 0(t3) #t3=L[i-1][j-1]**

**addi t3 t3 1 #t3=t3+1**

**sw t3 0(t1) #L[i][j]=L[i-1][j-1]+1**

**addi a3 a3 1 #j=j+1**

**beq x0 x0 Second\_for**

**condition3:**

**addi t0 a6 1 #t0=(SBSize+1)**

**mul t0 t0 a2 #t0=(SBSize+1)\*i**

**add t1 t0 a3 #t1=(SBsize+1)\*i+j**

**slli t1 t1 2 #t1=((SBSize+1)\*i+j)\*4 RAW Hazard**

**mv t0 t1 #t0=((SBSize+1)\*i+j)\*4**

**add t1 a4 t1 #t1=((SBSize+1)\*i+j)\*4+a4**

**addi t2 a6 1 #t2=(SBSize+1)**

**slli t2 t2 2 #t2=(SBSize+1)\*4**

**sub t2 t0 t2 #t2=[(SBSize+1)\*(i-1)+j]\*4 RAW Hazard**

**add t2 a4 t2 #t2=[(SBSize+1)\*(i-1)+j]\*4+a4 WAW Hazard**

**lw t2 0(t2) #t2=L[i-1][j] LOAD Hazard**

**addi t4 t0 -4 #t4=[(SBSize+1)\*i+(j-1)]\*4**

**add t4 a4 t4 #t4=[(SBSize+1)\*i+(j-1)]\*4+a4 WAW Hazard**

**lw t4 0(t4) #t4=L[i][j-1] LOAD Hazard**

**sub t5 t2 t4**

**bge t5 zero index**

**sw t4 0(t1) #L[i][j]=t2=L[i][j-1]**

**beq x0 x0 exit\_condition3**

**index:**

**sw t2 0(t1) #L[i][j]=t2=L[i-1][j]**

**exit\_condition3:**

**addi a3 a3 1 #j=j+1**

**beq x0 x0 Second\_forexit:**

**addi a5 a5 1**

**addi a6 a6 1**

**mul t0 a5 a6 #t0=(SASize+1)\*(SBSize+1)**

**slli t0 t0 2 #t0=(SASize+1)\*(SBSize+1)\*4**

**addi t0 t0 -4 #t0=(SASize+1)\*(SBSize+1)\*4-4 the last word**

**add a4 a4 t0 #a4=a4+t0**

**lw a0 0(a4) #a0=L[SASize][SBSize]**

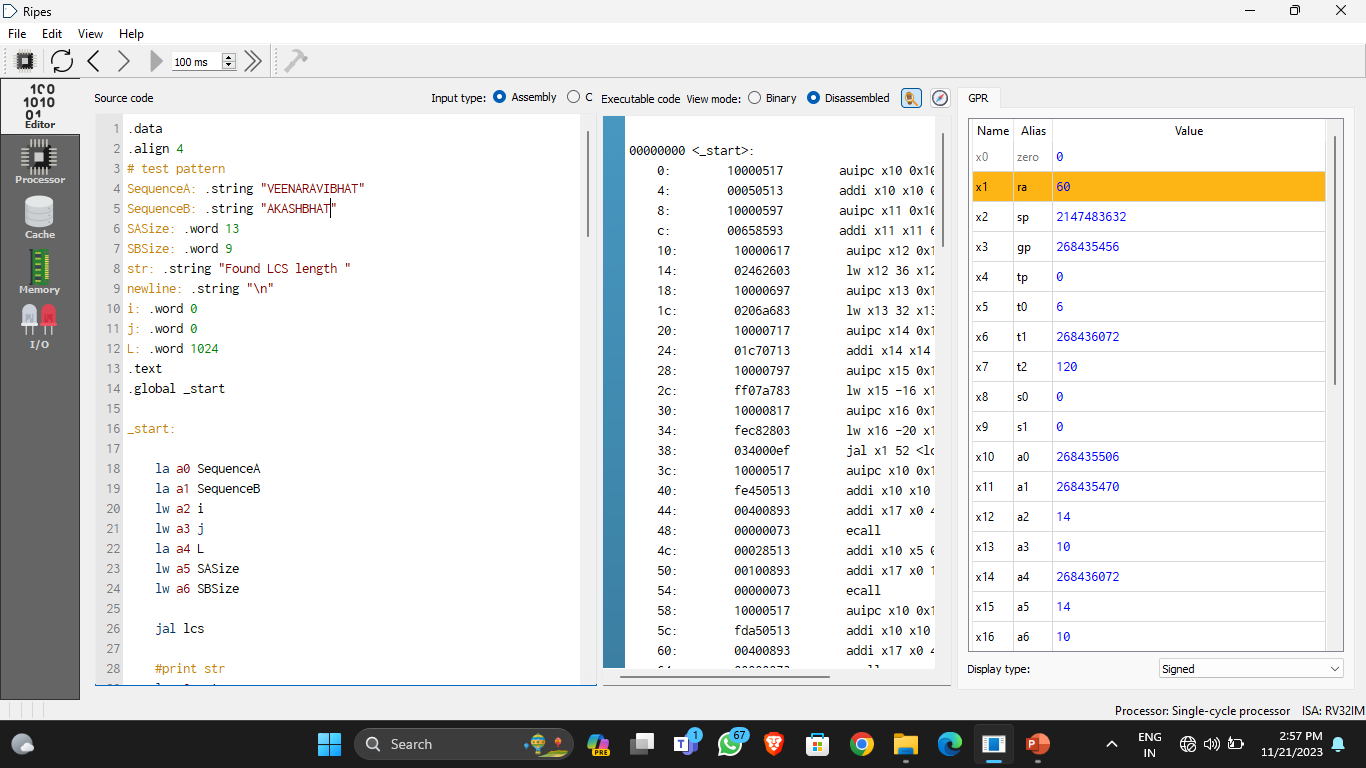
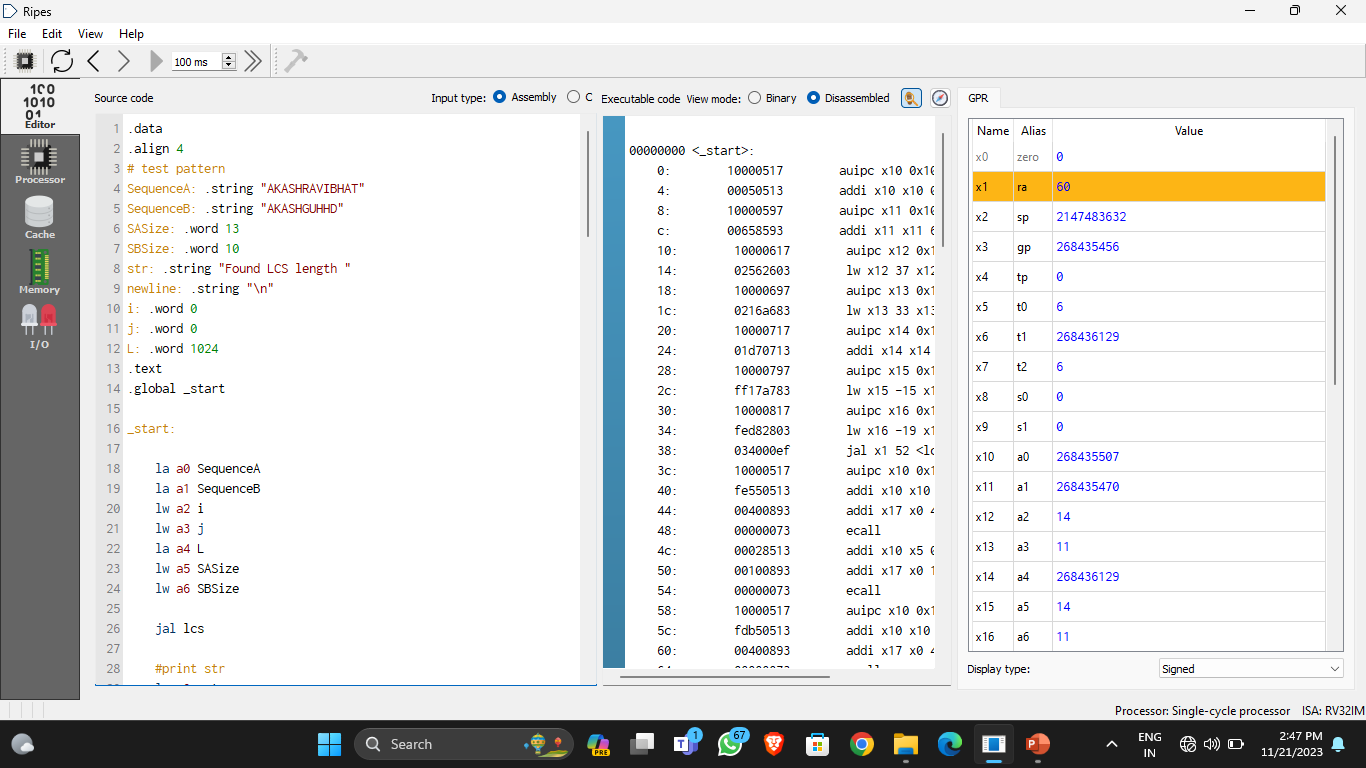
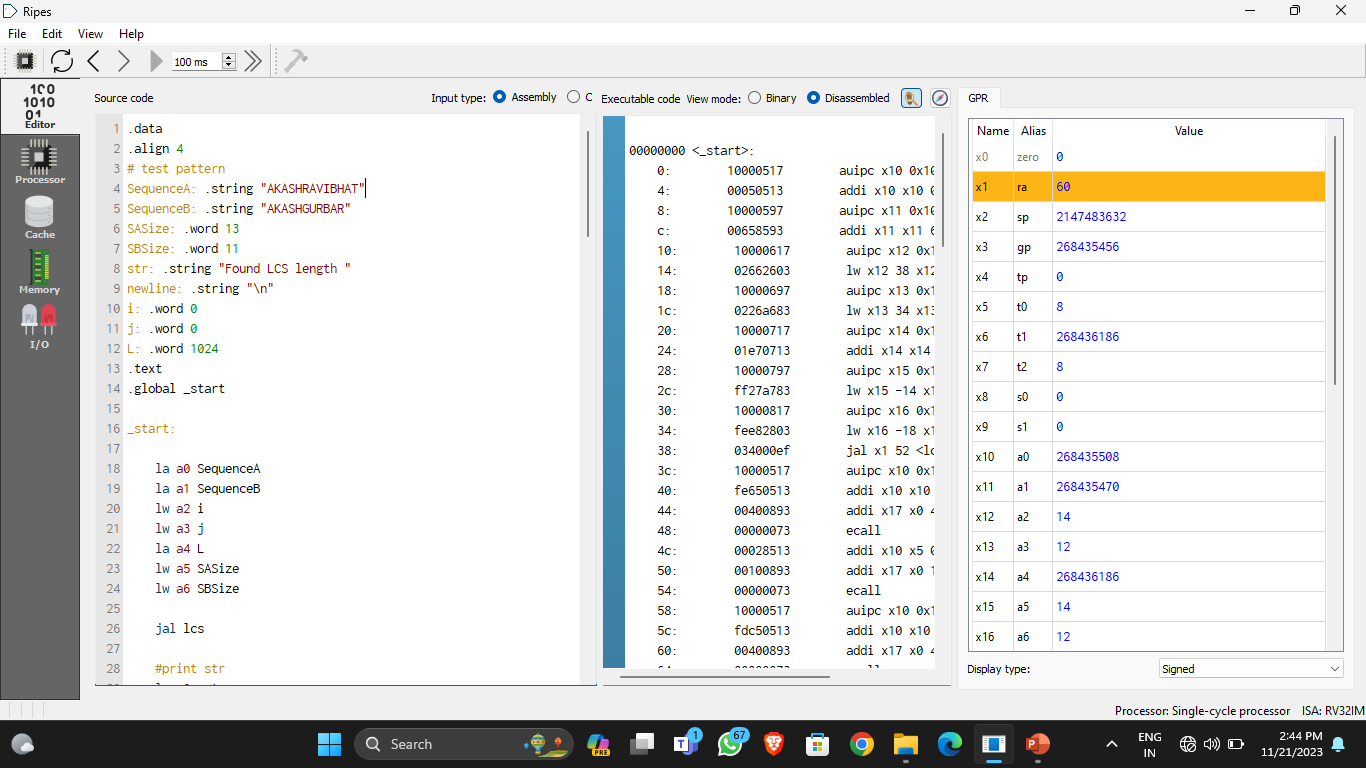
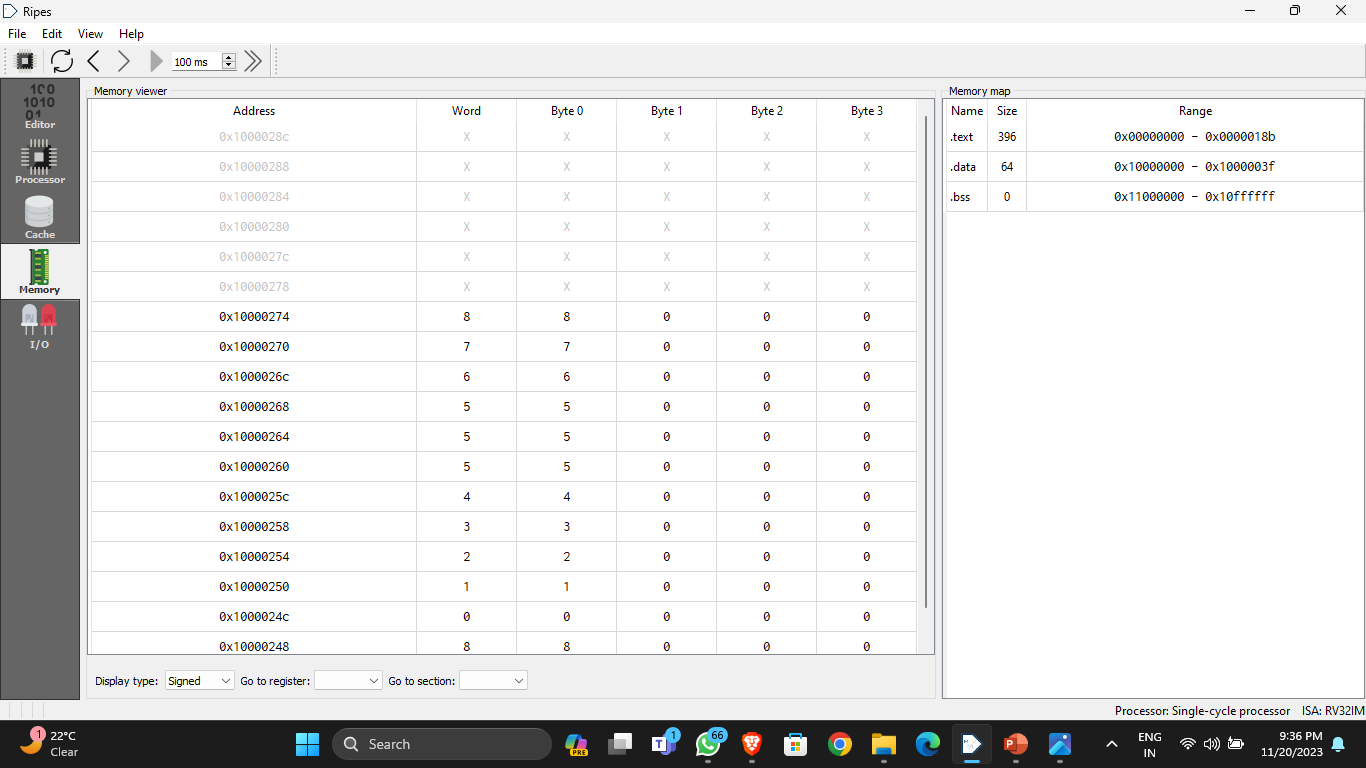
**lw ra, 0(sp) # Reload return address from stack**

**add t0 x0 a0**

**addi sp, sp, 4 # Restore stack pointer**

**jr x1**

**end:nop**

****

**REFERENCE:**

1. **Books:**
   * **"Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.**
   * **"Assembly Language for x86 Processors" by Kip R. Irvine.**
2. **Online Documentation:**
   * **RISC-V ISA specifications:** [**https://riscv.org/specifications/**](https://riscv.org/specifications/)
3. **Educational Platforms:**
   * **Online programming courses or platforms where assembly programming is taught, such as Coursera, edX, or others.**

THANK YOU!