Digital Design Verification

Weekly Task # 1 – RISC-V Assembler

Name	*Khalil Rehman*
Instructor	*Sir Umer*
Date	*24th Aug 2025*

Core Architecture: The Two-Pass Process

The assembler's operation is built around the classic two-pass assembly strategy, which efficiently resolves forward references (like using a label before it's defined).

I. First Pass (first_pass()):

Purpose: Symbol (Label) Resolution and Address Calculation.
Process: It reads the source code line by line.

- It tracks the current memory address in the **.text** segment.
- It identifies **labels** (text ending with a colon **:)** and records them into a **symbol_label** dictionary, mapping the label name to its current memory address.
- It identifies and processes assembler directives (like .**word**) that affect data segment size or address alignment.
- It skips over instructions and pseudo-instructions without generating code, only counting the bytes they will occupy (4 bytes per instruction).

Outcome: A complete **symbol_label** with all labels and their correct memory addresses.

II. Second Pass (second_pass()):

Purpose: Machine Code Generation.
Process: It reads the source code again, line by line.

- For each instruction, it uses the **symbol_table** from the first pass to resolve label references into concrete immediate values.
- It identifies **labels** (text ending with a colon :) and records them into a **symbol_table** dictionary, mapping the label name to its current memory address.
- It parses the instruction's operands (registers, immediates).
- Based on the instruction type (R, I, S, B, U, J), it calls the corresponding **encode_*_type()** function.
- These functions pack the various fields (opcode, rd, rs1, rs2, funct3, funct7, immediate) into a single 32-bit integer representing the machine code instruction.
- It skips over instructions and pseudo-instructions without generating code, only counting the bytes they will occupy (4 bytes per instruction).

Outcome: A list of 32-bit integers representing the final machine code program.

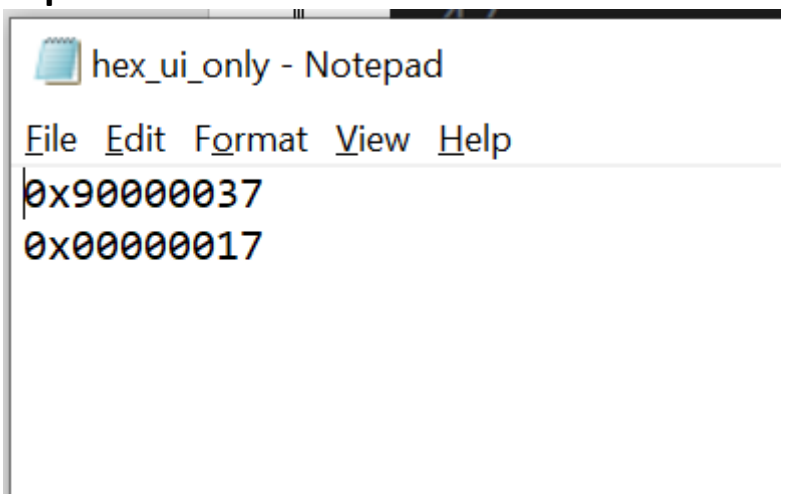
1. Tasks:

The provided test case was implemented successfully and the results are shown in the snapshots below.

I. Test UI_Only:

```
45    lui x0,0x90000
46    auipc x0,0x0000
47
```

Expected:



Execution:

```
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler> ./code test.as
m test.bin

Labels collected in first pass:

0x90000037  // lui x0,0x90000
0x00000017  // auipc x0,0x0000
Assembly completed successfully.
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler>
```

II. Test Reg_Only:

```
45    add  x31,x6,x0
46    sub  x5,x6,x0
47    and  a2,x6,x0
48    or  x5,x6,x0
49    xor  x5,x6,x0
50    sll t3,t4,x0
51    srl t3,s11,x0
52    slt t3,s9,x0
53    sltu t3,t4,x0
54    sra t3,t4,x0
```

Expected Output:

hex_reg_only - Notepad

File Edit Format View Help

```
0x00030FB3
0x400302B3
0x00037633
0x000362B3
0x000342B3
0x000E9E33
0x000DDE33
0x000CAE33
0x000EBE33
0x400EDE33
```

Executed:

```
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler> ./code test.asm test.bin

Labels collected in first pass:

0x00030FB3  // add  x31,x6,x0
0x400302B3  // sub  x5,x6,x0
0x00037633  // and  a2,x6,x0
0x000362B3  // or  x5,x6,x0
0x000342B3  // xor  x5,x6,x0
0x000E9E33  // sll t3,t4,x0
0x000DDE33  // srl t3,s11,x0
0x000CAE33  // slt t3,s9,x0
0x000EBE33  // sltu t3,t4,x0
0x400EDE33  // sra t3,t4,x0
Assembly completed successfully.
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler>
```

III. Test Store_Only:

```
46
47      sw x0,0(sp)
48      sh x0,-6(sp)
49      sb x0,-9(sp)
50
```

Expected:

hex_stores_only - Notepad

File Edit Format View Help

```
0x00012023
0xFE011D23
0xFE010BA3
```

Execution

```
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler> ./code test.asm test.bin

Labels collected in first pass:

0x00012023  // sw x0,0(sp)
0xFE011D23  // sh x0,-6(sp)
0xFE010BA3  // sb x0,-9(sp)
Assembly completed successfully.
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler>
```

IV. Test Imm_Only

```
46
47    addi  x31,x6,0
48    andi  a2,x6,0
49    ori x5,x6,11
50    xori  x5,x6,24
51    slli t3,t4,17
52    srli t3,s11,3
53    slti t3,s9,24
54    sltiu t3,t4,-256
55    srai t3,t4,30
56    lw x0,0(sp)
57    lw x0,-4(sp)
58    lh x0,-6(sp)
59    lhu x0,-8(sp)
60    lb x0,-9(sp)
61    lbu x0,-10(sp)
62
```

Expected:

hex_im_only - Notepad

File Edit Format View Help

```
0x00030F93
0x00037613
0x00B36293
0x01834293
0x011E9E13
0x003DDE13
0x018CAE13
0xF00EBE13
0x41EEDE13
0x00012003
0xFFC12003
0xFFA11003
0xFF815003
0xFF710003
0xFF614003
```

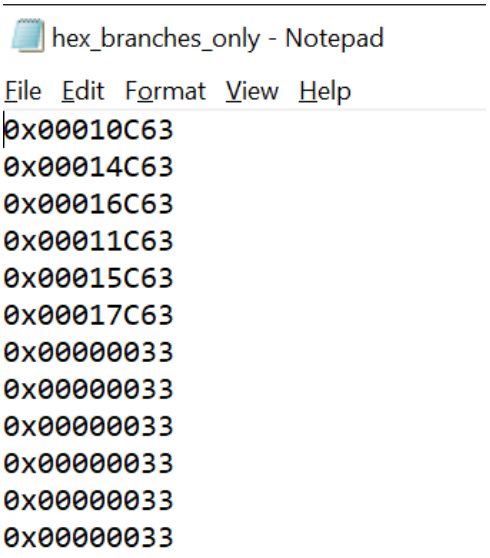
Executed

```
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler> ./code test.asm test.bin

Labels collected in first pass:

0x00030F93  // addi  x31,x6,0
0x00037613  // andi  a2,x6,0
0x00B36293  // ori x5,x6,11
0x01834293  // xori  x5,x6,24
0x011E9E13  // slli t3,t4,17
0x003DDE13  // srli t3,s11,3
0x018CAE13  // slti t3,s9,24
0xF00EBE13  // sltiu t3,t4,-256
0x41EEDE13  // srai t3,t4,30
0x00012003  // lw x0,0(sp)
0xFFC12003  // lw x0,-4(sp)
0xFFA11003  // lh x0,-6(sp)
0xFF815003  // lhu x0,-8(sp)
0xFF710003  // lb x0,-9(sp)
0xFF614003  // lbu x0,-10(sp)
Assembly completed successfully.
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler>
```

V. Test Branch_Only:

```
46  main:
47  beq sp,x0,EQ
48  blt sp,x0,LT
49  bltu sp,x0,LTU
50  bne sp,x0,NE
51  bge sp,x0,GE
52  bgeu sp,x0,GEU
53
54  EQ:
55  add x0,x0,x0
56  LT:
57  add x0,x0,x0
58  LTU:
59  add x0,x0,x0
60  NE:
61  add x0,x0,x0
62  GE:
63  add x0,x0,x0
64  GEU:
65  add x0,x0,x0
66
```

Expected:



Execution

```
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler> ./code test.asm test.bin
Info: Collected label 'main' at address 0x00000000
Info: Collected label 'EQ' at address 0x00000018
Info: Collected label 'LT' at address 0x0000001C
Info: Collected label 'LTU' at address 0x00000020
Info: Collected label 'NE' at address 0x00000024
Info: Collected label 'GE' at address 0x00000028
Info: Collected label 'GEU' at address 0x0000002C

Labels collected in first pass:
  main: 0x00000000
  EQ: 0x00000018
  LT: 0x0000001C
  LTU: 0x00000020
  NE: 0x00000024
  GE: 0x00000028
  GEU: 0x0000002C

0x00010C63  // beq sp,x0,EQ
0x00014C63  // blt sp,x0,LT
0x00016C63  // bltu sp,x0,LTU
0x00011C63  // bne sp,x0,NE
0x00015C63  // bge sp,x0,GE
0x00017C63  // bgeu sp,x0,GEU
0x00000033  // add x0,x0,x0
0x00000033  // add x0,x0,x0
0x00000033  // add x0,x0,x0
0x00000033  // add x0,x0,x0
0x00000033  // add x0,x0,x0
0x00000033  // add x0,x0,x0
Assembly completed successfully.
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler>
```

VI. Test Jump_Only:

```
45  main:
46  jal ra,boot
47  jal ra, multiply
48  jal x0, exit
49  boot:
50  jalr x0,ra,0
51  multiply:
52  jalr x0,ra,0
53  exit:
54
```

Expected:

hex_jumps_only - Notepad

File Edit Format View Help

```
0x00C000EF
0x00C000EF
0x00C0006F
0x00008067
0x00008067
```

Executed:

```
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler> ./code test.asm test.bin
Info: Collected label 'main' at address 0x00000000
Info: Collected label 'boot' at address 0x0000000C
Info: Collected label 'multiply' at address 0x00000010
Info: Collected label 'exit' at address 0x00000014

Labels collected in first pass:
  main: 0x00000000
  boot: 0x0000000C
  multiply: 0x00000010
  exit: 0x00000014

0x00C000EF  // jal ra,boot
0x00C000EF  // jal ra, multiply
0x00C0006F  // jal x0, exit
0x00008067  // jalr x0,ra,0
0x00008067  // jalr x0,ra,0
Assembly completed successfully.
PS F:\NCDC\DV Training\Digital Design Verification Course\Riscv Assembly Language\Project Asssembler>
```

Function Summaries:

- **get_register_number()**: Maps register names to their numbers using a lookup table.
- **trim_token()**: Removes whitespace from the start and end of a string.
- **encode_*_type() functions**: Encode RISC-V instructions into machine code based on their format.
- **is_assembler_directive()**: Checks if a token is a known assembler directive.
- **handle_assembler_directive()**: Processes assembler directives like .data, .text, and .word.
- **parse_immediate()**: Converts immediate values/labels to numerical values, handling different bases.
- **process_instruction()**: Main instruction processing function that handles all supported RISC-V instructions.
- **first_pass()**: Collects labels and calculates their addresses.
- **second_pass()**: Generates machine code using collected label information.
- **trim_whitespace()**: Removes leading/trailing whitespace from strings.
- **main()**: Coordinates the two-pass assembly process.

Conclusion and Project Summary

This project involved the design and implementation of a functional two-pass assembler for the RISC-V instruction set architecture. The core objective was to accurately translate symbolic assembly code, complete with labels and directives, into executable machine code.

Key Achievements:

- **Successful Two-Pass Implementation:** The assembler correctly implements the classical two-pass assembly process. The first pass effectively builds a comprehensive symbol table by calculating the addresses of all labels. The second pass leverages this table to resolve symbolic references and generate the final machine code instructions.
- **Comprehensive Instruction Support:** The assembler supports all fundamental RISC-V 32I instruction formats (R, I, S, B, U, J). It correctly encodes a wide range of instructions, including arithmetic operations (e.g., add, sub), data transfers (e.g., lw, sw), and control flow instructions (e.g., beq, jal).
- **Pseudo-Instruction Expansion:** The code correctly recognizes and expands common pseudo-instructions (such as li, mv, and j) into their equivalent sequences of base RISC-V instructions, providing a more user-friendly programming interface.

- **Assembler Directive Processing:** Directives including .text, .data, and .word are properly handled, allowing for the separation of code and data segments and the initialization of memory values.
- **Robust Parsing and Error Handling:** The assembler incorporates robust input parsing, with functions to trim whitespace and validate tokens. It features extensive error checking throughout the assembly process, providing clear and informative error messages for issues like undefined labels, invalid syntax, or incorrect register names, which greatly aids in debugging assembly programs.

Outcome:

The assembler successfully processes input assembly files, resolves all symbolic labels, and produces the correct binary output. All provided test cases, including those with complex label usage and pseudo-instructions, were assembled correctly, confirming the validity and reliability of the implementation. This project demonstrates a strong understanding of the RISC-V architecture, low-level software tooling, and the fundamental principles of the assembly process.