

CSED311 Lab2: Single-Cycle CPU

Minsu Gong

gongms@postech.ac.kr

Contact the TAs at csed311-ta@postech.ac.kr

Contents

- Assignments
 - Single-Cycle CPU (RV32I)
- Ripes simulator
- How to run C program on Ripes & Verilog RTL (Non-mandatory)

Assignments

Assignments

- Use Verilator
- Implement a **single-cycle RISC-V CPU** (RV32I)
 - Single-cycle CPU
 - Datapath
 - ALU
 - Register file
 - Control unit
 - Generate the control signals used in the datapath
- Your implementation of the **CPU should process one instruction in a cycle**

Assignments

- Skeleton code
 - top.v – Top module (Do not touch)
 - opcodes.v – Opcodes of instructions you must implement
 - Other modules (add more if you need)
 - cpu.v, instruction_memory.v, data_memory.v, register_file.v
- Testbench
 - **Simulation code**
 - tb_top.cpp
 - **Instruction codes** for Verilog RTL (.txt)
 - basic_ripes.txt, non-controlflow_mem.txt, loop_mem.txt
 - **Assembly codes** for Ripes (.asm) (will explain later)
 - basic_ripes.asm, non-controlflow_mem.asm, loop_mem.asm
- Makefile

Assignments

- RV32I instructions you should implement

imm[20 10:1 11 19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000000000000		00000	000	00000	1110011	ECALL

Assignments

- We are going to use the **ECALL** (environment call) instruction to halt the machine at the end of a program
 - Simulation ending condition
 - If instruction == **ECALL**
 - If GPR[x17] == 10
 - Set **is_halted = 1** (Testbench will stop simulation when is_halted is 1)
 - Else
 - Consider **ECALL** as **NOP**
- Example of instructions exiting the program

58:	00f766b3	or x13 x14 x15
5c:	01c12083	lw x1 28 x2
60:	01812403	lw x8 24 x2
64:	02010113	addi x2 x2 32
68:	00a00893	addi x17 x0 10
6c:	00000073	ecall

Assignments

- For other instructions, refer to the RV32I manual
 - References:
 - See RISC-V specification (Vol. 1) provided for the lab
 - <https://github.com/riscv/riscv-isa-manual/releases/tag/Ratified-IMAFDQC>
 - <https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html>

Modularization

- **Modularize the main CPU structure** (strongly recommended)
 - Datapath
 - ALU
 - Register file
 - Control Unit
 - Etc.
 - MUX, adder, ..
- Keep one module in one Verilog file (Verilator issue)
- Match file name with module name (Verilator issue)
- You may modify the interfaces of some of the modules (except top.v, cpu.v)

Magic Memory

- In `instruction_memory.v`, `data_memory.v`
- It is **NOT a realistic model** of the main memory
 - In our lab, memory works like a register file, except that the memory is byte-addressable and accessed with memory address instead of register ID
 - Real memory devices are much slower than CPUs
 - However, **we assume there is a magic memory with very low latency** for simplicity

Initialize instruction memory

The number of instructions < MEM_DEPTH

fe010113
00812e23
02010413
00300793
fef42623
00200793
fef42423
fec42703
fe842783
02f707b3
fef42223
00000793
00078513
01c12403
02010113
00000073

```
always @(posedge clk) begin
    // Initialize instruction memory
    if (reset) begin
        for (i = 0; i < MEM_DEPTH; i = i + 1)
            mem[i] = 32'b0;
        // Provide path of the file including instructions with binary format
        $readmemh(" ", mem);
        //for (i = 0; i < 50; i = i + 1)
        //$display("mem[%d] = %h \n", i, mem[i]);
    end
end
```

Module 'instruction_memory'

Change the path to your .txt file

Evaluation Criteria (Source code)

- **Single-Cycle CPU**
 - The score will be calculated based on the **final register values (x1-x31)** of the Verilog RTL after testbenches for evaluation are executed (i.e., how many registers have the correct values)
 - Testbench will print final register values
 - You can check correct register values by **running .asm file with Ripes**
 - You are encouraged to run your own program on your Verilog RTL model

Evaluation Criteria (Report)

- You can write report in **Korean** or **English**
- Report should include (1) introduction, (2) design, (3) implementation, (4) discussion, and (5) conclusion
- **Key points:**
 - Single-cycle CPU design and implementation
 - Description of whether each module(RF, memory, PC, control unit, ..) is clock synchronous or asynchronous
 - Description of each stage in single-cycle CPU

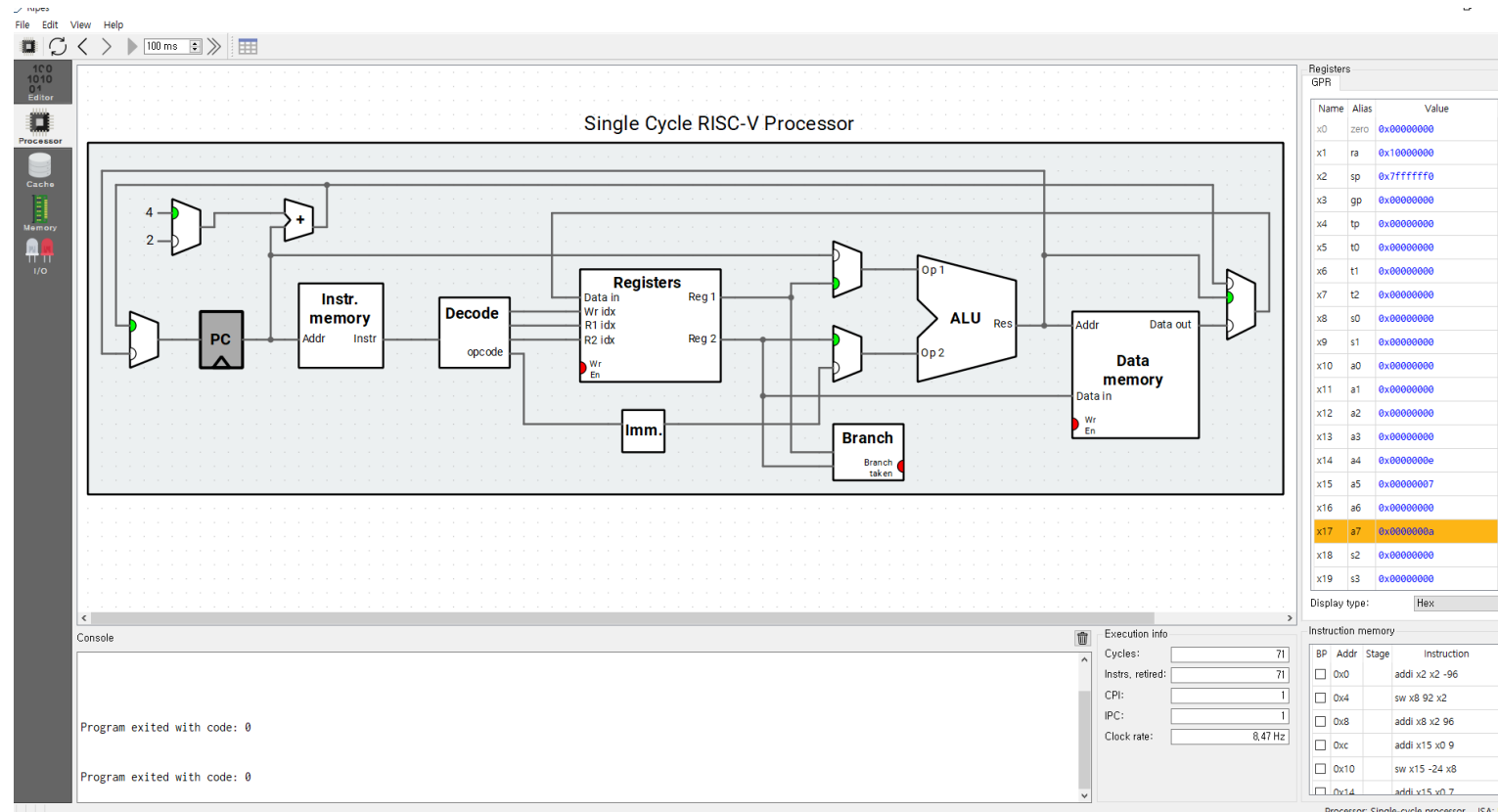
Assignment Submission

- **Submit your assignment to PLMS with filename:**
 - Lab2_{TeamID}_{StudentID1}_{StudentID2}.pdf
 - PDF file of your report
 - Lab2_{TeamID}_{StudentID1}_{StudentID2}.zip
 - Zip file of your source code (**without testbench**)
 - One directory including all codes when unzipped
- **Due date**
 - Code: 2023. 3. 26 / 09:00
 - Report: 2023. 3. 26 / 18:00

Ripes simulator

Ripes simulator

- **Ripes** is a visual computer architecture simulator and assembly code editor built for the RISC-V instruction set architecture
- Ripes can help you debug code



Ripes simulator

- How to install?
 - <https://github.com/mortbopet/Ripes/releases>
 - Install the latest version

▼ Assets

5

 Ripes-v2.2.6-linux-x86_64.AppImage	29.6 MB	Jan 8, 2023
 Ripes-v2.2.6-mac-x86_64.zip	30.4 MB	Jan 8, 2023
 Ripes-v2.2.6-win-x86_64.zip	15 MB	Jan 8, 2023

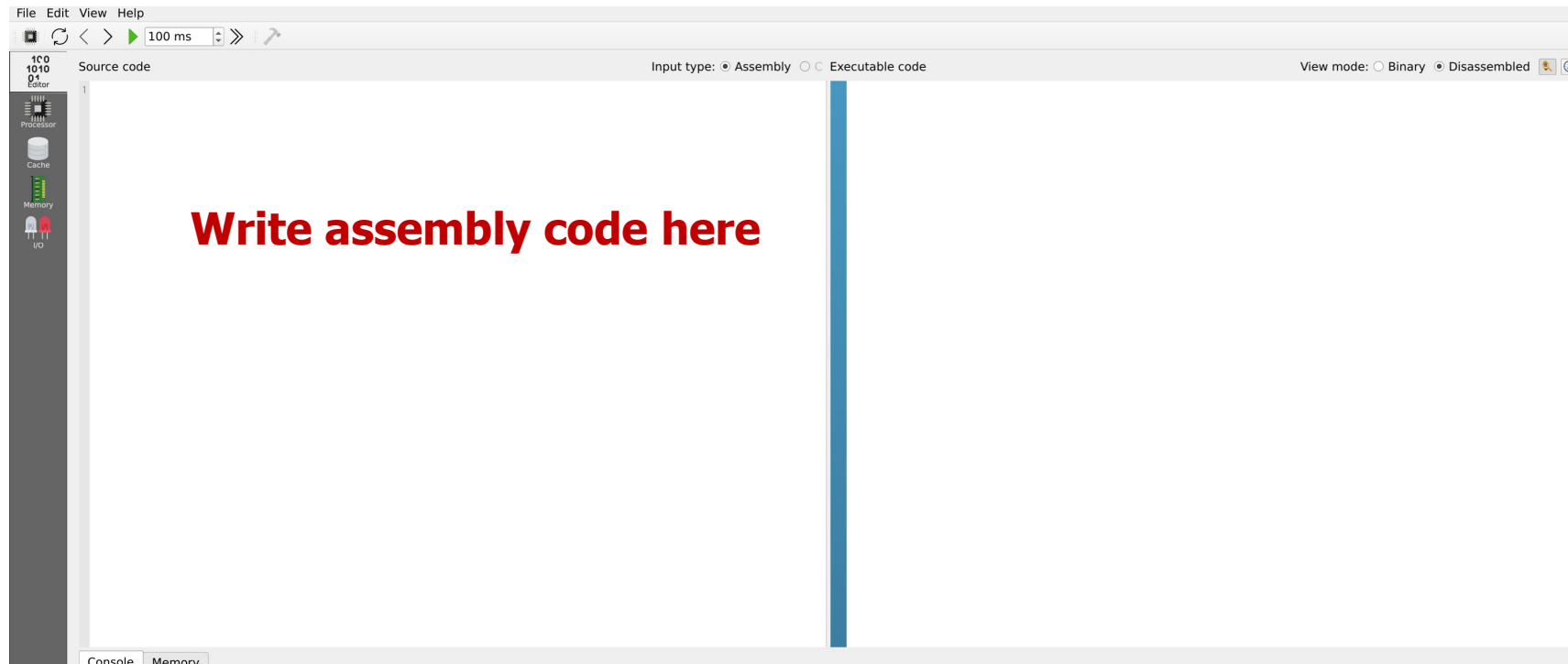
- An error for VCRUNTIME140_1.dll may be displayed
 - Follow the instruction in the link above

Windows

For Windows, the C++ runtime library must be available (if not, a msvcp140.dll error will be produced). You most likely already have this installed, but if this is not the case, you download it [here](#).

Ripes simulator

- Web version is also available
 - <https://ripes.me/>
 - But you can't load a program with web version.
 - Still, you can copy and paste your code to editor



Ripes – Processor configuration

iconengines
imageformats
platforms
styles
d3dcompiler_47.dll
libEGL.dll
libGLESv2.dll
Qt5Charts.dll
Qt5Core.dll
Qt5Gui.dll
Qt5Svg.dll
Qt5Widgets.dll
Ripes

Ripes

Single Cycle RISC-V Processor

Please configure it as below

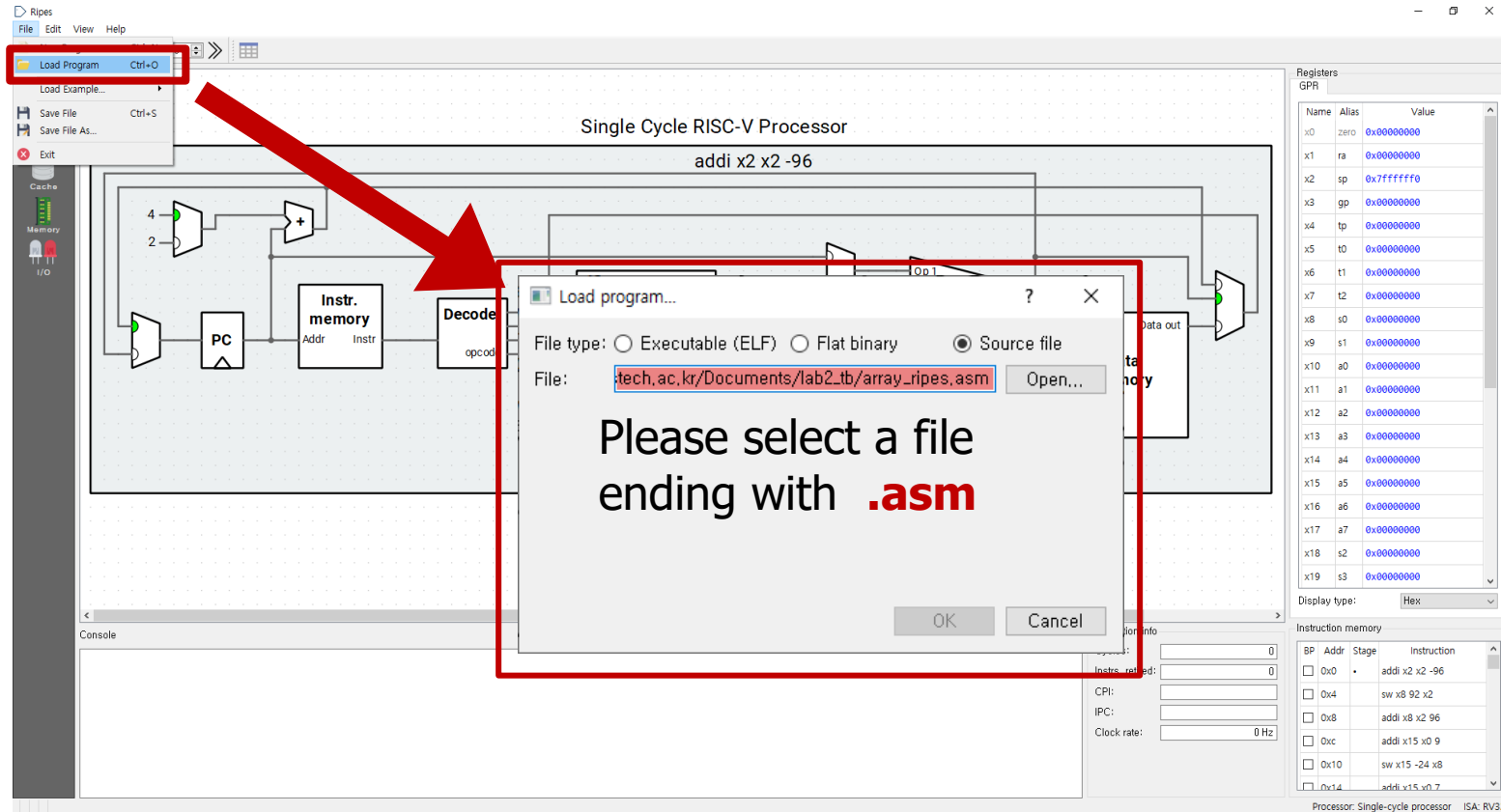
Select Processor

RISC-V
32-bit
Single-cycle processor
5-stage processor w/o forwarding or hazard detection
5-stage processor w/o hazard detection
5-stage processor w/o forwarding unit
5-stage processor
6-stage dual-issue processor
64-bit

Name: Single-cycle processor
ISA: RV32I
ISA Exts. ☒ M ☐ C
Layout: Standard
Description: A single cycle processor
Register initialization: x2 (sp) 0x2ffc

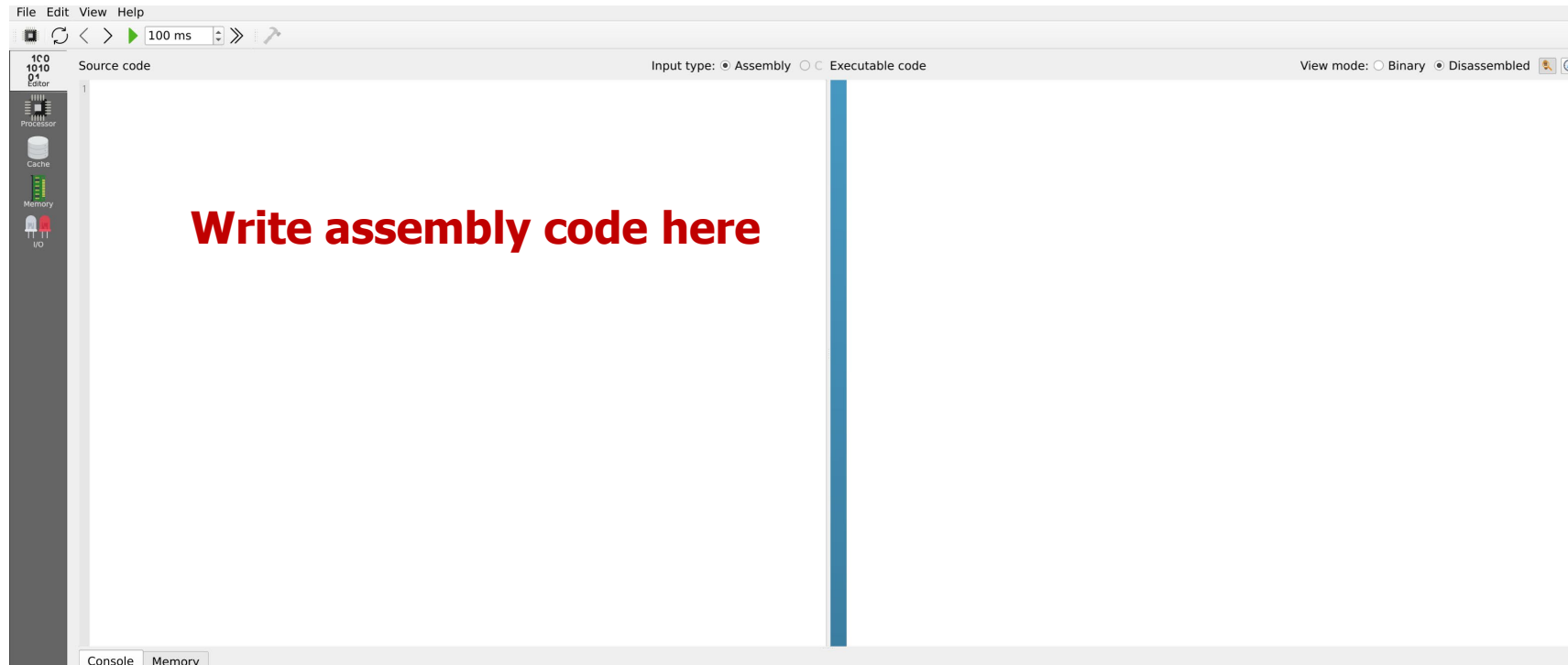
OK Cancel

Ripes – Loading a program



Ripes – Loading a program

- For web version,
 - You can't load a program with web version.
 - Still, you can copy and paste your code to editor



Ripes – Running the program

Run

The screenshot displays the Ripes IDE interface. On the left, a sidebar contains icons for Processor, Cache, Memory, and I/O. The main window is divided into three panes. The left pane shows the source code in assembly, with the first line highlighted. The middle pane shows the disassembled code, with the first instruction highlighted. The right pane shows the General Purpose Register (GPR) table, with the x10 register highlighted.

Source code:

```
1 <main>:  
2 addi sp,sp,-32  
3 sw ra,28(sp)  
4 sw s0,24(sp)  
5 addi s0,sp,32  
6 addi a5,zero,19  
7 sw a5,-20(s0)  
8 addi a5,zero,14  
9 sw a5,-24(s0)  
10 lw a1,-24(s0)  
11 lw a0,-20(s0)  
12 jal ra,<add_func>  
13 sw a0,-28(s0)  
14 lw a4,-28(s0)  
15 lw a5,-20(s0)  
16 add a5,a4,a5  
17 sw a5,-32(s0)  
18 lw a2,-32(s0)  
19 lw a1,-28(s0)  
20 lw ra,28(sp)  
21 lw s0,24(sp)  
22 addi sp,sp,32  
23 li a7,10  
24 ecall  
25  
26 <add_func>:  
27 addi sp,sp,-32  
28 sw s0,28(sp)  
29 addi s0,sp,32
```

Disassembly:

```
0: fe010113 addi x2,x2,-32  
4: 00112e23 sw x1,28(x2)  
8: 00812c23 sw x8,24(x2)  
c: 02010413 addi x8,x2,32  
10: 01300793 addi x15,x0,19  
14: fef42623 sw x15,-20(x8)  
18: 00e00793 addi x15,x0,14  
1c: fef42423 sw x15,-24(x8)  
20: fe842583 lw x11,-24(x8)  
24: fec42503 lw x10,-20(x8)  
28: 034000ef jal x1,52<<add_func>>  
2c: fea42223 sw x10,-28(x8)  
30: fe442703 lw x14,-28(x8)  
34: fec42783 lw x15,-20(x8)  
38: 00f707b3 add x15,x14,x15  
3c: fef42023 sw x15,-32(x8)  
40: fe042603 lw x12,-32(x8)  
44: fe442583 lw x11,-28(x8)  
48: 01c12083 lw x1,28(x2)  
4c: 01812403 lw x8,24(x2)  
50: 02010113 addi x2,x2,32  
54: 00a00893 addi x17,x0,10  
58: 00000073 ecall  
  
0000005c <<add_func>>:  
5c: fe010113 addi x2,x2,-32  
60: 00812e23 sw x8,28(x2)  
64: 02010413 addi x8,x2,32  
68: fea42623 sw x10,-20(x8)
```

GPR:

Name	Alias	Value
x0	zero	0
x1	ra	0
x2	sp	12284
x3	gp	0
x4	tp	0
x5	t0	0
x6	t1	0
x7	t2	0
x8	s0	0
x9	s1	0
x10	a0	0
x11	a1	0
x12	a2	0
x13	a3	0
x14	a4	0
x15	a5	0
x16	a6	0
x17	a7	0
x18	s2	0

As you can see, assembly code uses pseudo-instructions and register aliases.
See “RISC-V Assembly Programmer’s Handbook” Chapter of RISC-V manual.

Ripes – x10 issue

Ripes

File Edit View Help

100 ms

Source code

```
25      addi    a7,zero,4
26      sll     a7,a6,a7
27      xor     a7,a7,a0
28      sub     a6,a6,a0
29      srl     a7,a7,a6
30      addi    a1,zero,63
31      and     a3,a7,a1
32      not     a2,a7
33      sub     a6,a7,a6
34      srli    a7,a7,3
35      or      a3,a4,a5
36      lw      ra,28(sp)
37      lw      s0,24(sp)
38      addi    sp,sp,32
39      li      a7, 10
40      ecall
41
```

Input type: ☒ Assembly ☐ C Executable code

View mode: ☐ Binary ☒ Disassembled

Disassembled code:

```
20:      fe842703      lw x14 -24 x8
24:      fec42783      lw x15 -20 x8
28:      00f707b3      add x15 x14 x15
2c:      00000513      addi x10 x0 0
30:      fff54513      xori x10 x10 -1
34:      00a06533      or x10 x0 x10
38:      00c57513      andi x10 x10 12
3c:      01c50513      addi x10 x10 28
40:      00251513      slli x10 x10 2
44:      00455513      srli x10 x10 4
48:      00050633      add x12 x10 x0
4c:      01f00893      addi x17 x0 31
50:      00000073      ecall
54:      00060533      add x10 x12 x0
58:      00e06813      ori x16 x0 14
5c:      00400893      addi x17 x0 4
60:      011818b3      sll x17 x16 x17
```

GPR

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x00002fdc
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000000
x6	t1	0x00000000
x7	t2	0x00000000
x8	s0	0x00002ffc
x9	s1	0x00000000
x10	a0	0x0000000a
x11	a1	0x00000000
x12	a2	0x0000000a
x13	a3	0x00000000
x14	a4	0x0000000e

Console

Ripes – x10 issue

The screenshot displays the Ripes Simulator interface. On the left is a vertical toolbar with icons for Processor, Cache, Memory, and I/O. The main area is divided into three panes: Source code, Disassembled code, and GPR.

Source code pane: Shows assembly code from line 25 to 41. Line 40, `ecall`, is highlighted in blue.

Disassembled code pane: Shows the corresponding machine code instructions. Line 50, `ecall`, is highlighted in red.

GPR pane: A table of General Purpose Registers. The register `x10` is highlighted with a red border.

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x00002fdc
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000000
x6	t1	0x00000000
x7	t2	0x00000000
x8	s0	0x00002ffc
x9	s1	0x00000000
x10	a0	0x00000014
x11	a1	0x00000000
x12	a2	0x0000000a
x13	a3	0x00000000
x14	a4	0x0000000e

In Ripes Simulator, x10 holds the system call instruction's index when it is called.
You don't need to implement this on your CPU.

How to compile and run your own C program on Ripes and Verilog RTL

(Non-mandatory)

Cross-compiler

- How to install?
 - Use Docker (MacOS/Windows/Linux Support)
 - For Windows users,
 - Use **CSE Education Slurm Cluster** to use Docker
 - You can request the account of CSE Education Slurm Cluster at the below link
 - <https://postechackr.sharepoint.com/sites/cse/SitePages/CSE-Cluster-Howto.aspx?csf=1&e=qwAkG9&cid=dfb4c189-2455-4381-a8c1-2489054f57bb>
 - Or, use **WSL** to run docker on your computer
 - Get docker image
 - `$ docker pull acplpostech/acpl_ubuntu_18.04_riscv:latest`
 - Start docker
 - `$ docker run -v ~/.mnt -it --rm acplpostech/acpl_ubuntu_18.04_riscv:latest /bin/bash`

Cross-compiler

- Risc-V Assembly Code Generate (Use /RISCV_Crosscompile/script.sh)
 - \$ cd /RISCV_Crosscompile
 - \$./script.sh file_name

C Code (example.c)

```
int main()
{
    long long a, b, next;
    long long i;
    a = 0;
    b = 1;
    next = a + b;
    for(i = 0; i<10; i++)
    {
        a = b;
        b = next;
        next = a + b;
    }
    return 0;
}
```

./script.sh example



Assembly Code

```
add.o:      file format elf64-littleriscv

Disassembly of section .text:

0000000000000000 <main>:
0:   fd010113      addi    sp,sp,-48
4:   02813423      sd      s0,40(sp)
8:   03010413      addi    s0,sp,48
c:   fc043823      sd      zero,-48(s0)
10:  00100793      addi    a5,zero,1
14:  fef43423      sd      a5,-24(s0)
18:  fd043703      ld      a4,-48(s0)
1c:  fe843783      ld      a5,-24(s0)
20:  00f707b3      add     a5,a4,a5
24:  fef43023      sd      a5,-32(s0)
28:  fc043c23      sd      zero,-40(s0)
2c:  0300006f      jal     zero,5c <.L2>

0000000000000030 <.L3>:
30:  fe843783      ld      a5,-24(s0)
34:  fcf43823      sd      a5,-48(s0)
38:  fe043783      ld      a5,-32(s0)
3c:  fef43423      sd      a5,-24(s0)
40:  fd043703      ld      a4,-48(s0)
44:  fe843783      ld      a5,-24(s0)
48:  00f707b3      add     a5,a4,a5
4c:  fef43023      sd      a5,-32(s0)
50:  fd843783      ld      a5,-40(s0)
54:  00178793      addi    a5,a5,1
58:  fcf43c23      sd      a5,-40(s0)
```

Cross-compiler

- Risc-V Assembly Code Generate (Use /RISCV_Crosscompile/script.sh)
 - Output file
 - /RISCV_Crosscompile/{file_name}_ripes.asm -> for Ripes input
 - `$ mv /RISCV_Crosscompile/{file_name}_ripes.asm /mnt`
 - `$ exit #exit docker`
 - Now you can find {file_name}_ripes.asm in home directory

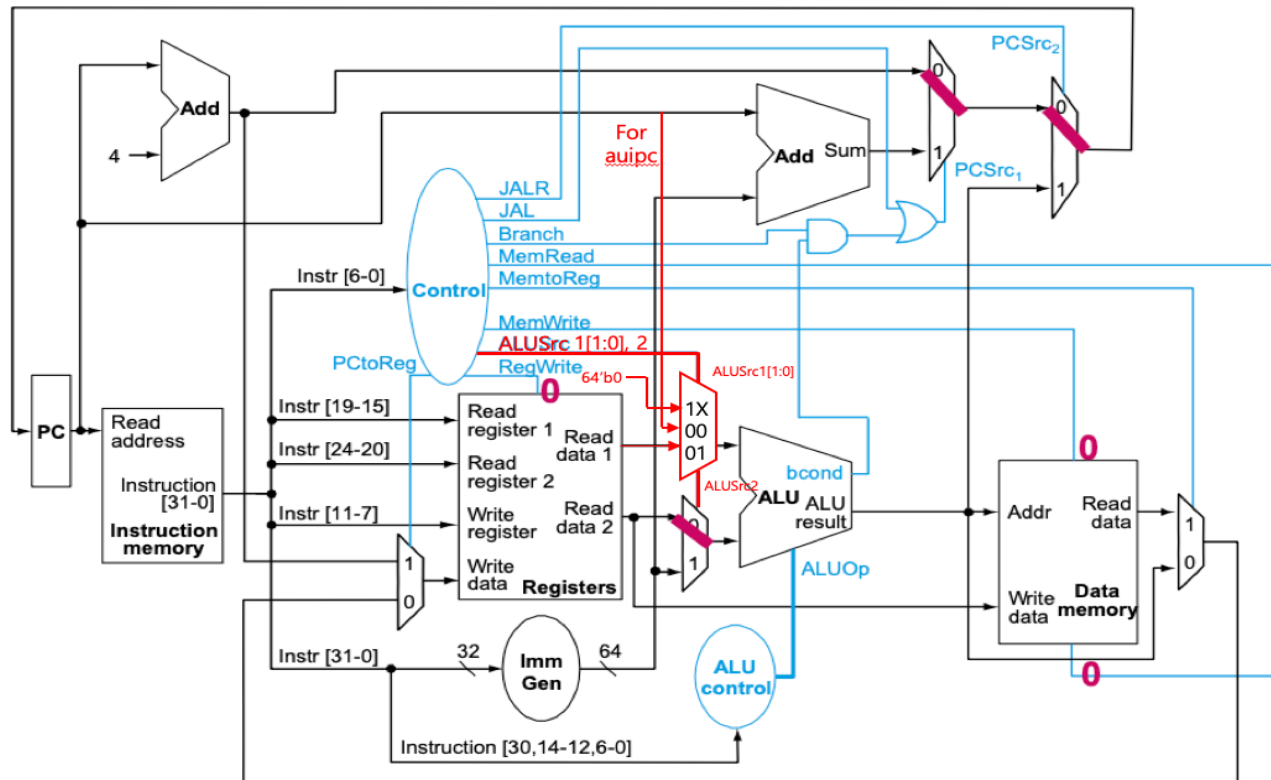
Caution

- Since we model RV32I without the “M” extension for multiply/divide, you can't use multiplication on the c code
- You can still multiply or divide by a power of 2 using shift left or right operations
- You will need to implement some additional instructions that are commonly emitted by the compiler

Assignments (Optional)

- Implement **LUI** and **AUIPC** if you want

Additional datapath component



ALUSrc1[1:0] For LUI Instruction

lui

load upper immediate.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[31:12]					rd	01101	11

Format: lui rd,imm

Description: Build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.

Implementation: $x[rd] = \text{sext}(\text{immediate}[31:12] \ll 12)$

LUI Instruction

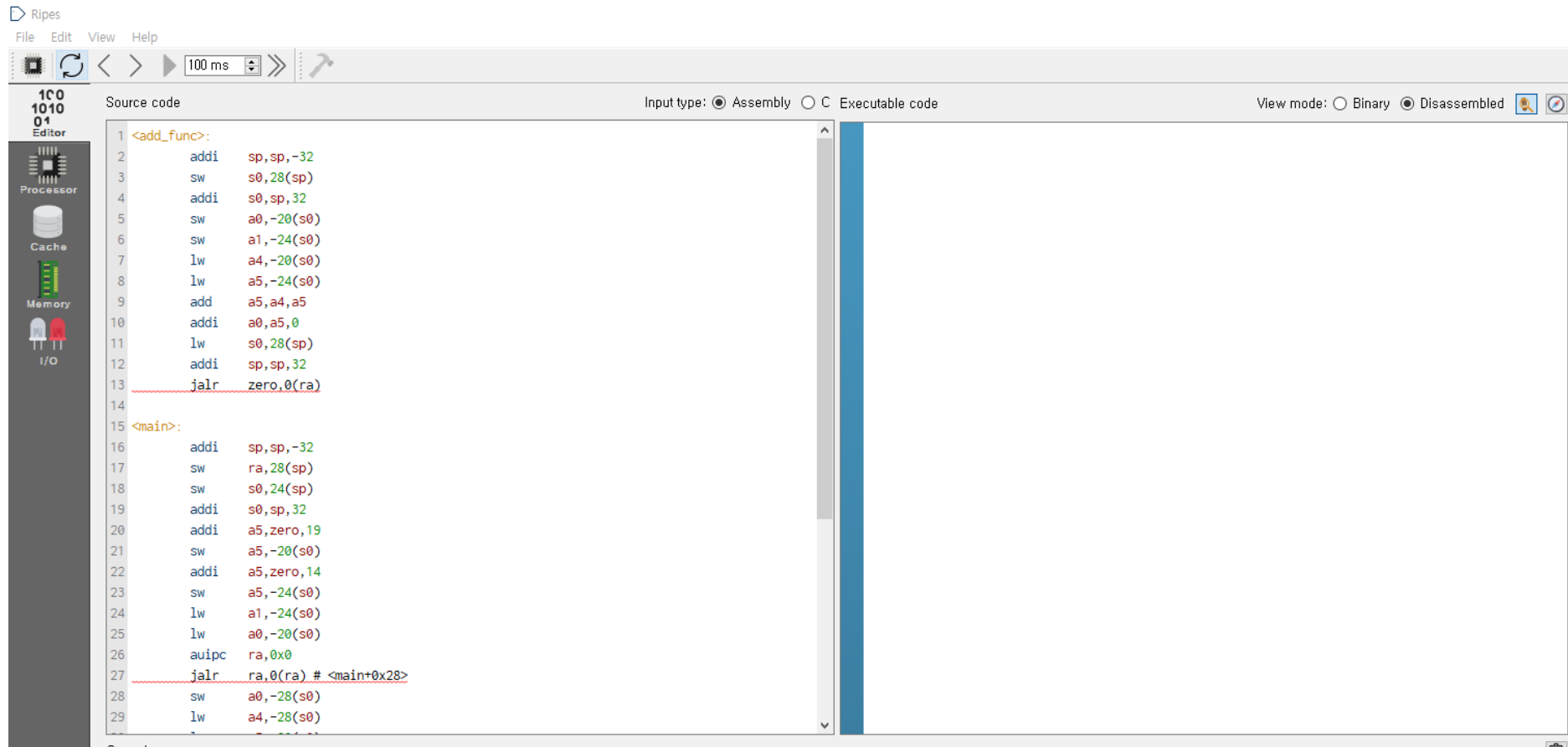
$$x[rd] = \text{Immediate}(\text{shifted}) + 64'b0$$

AUIPC Instruction

$$x[rd] = \text{PC} + \text{Immediate}(\text{shifted})$$

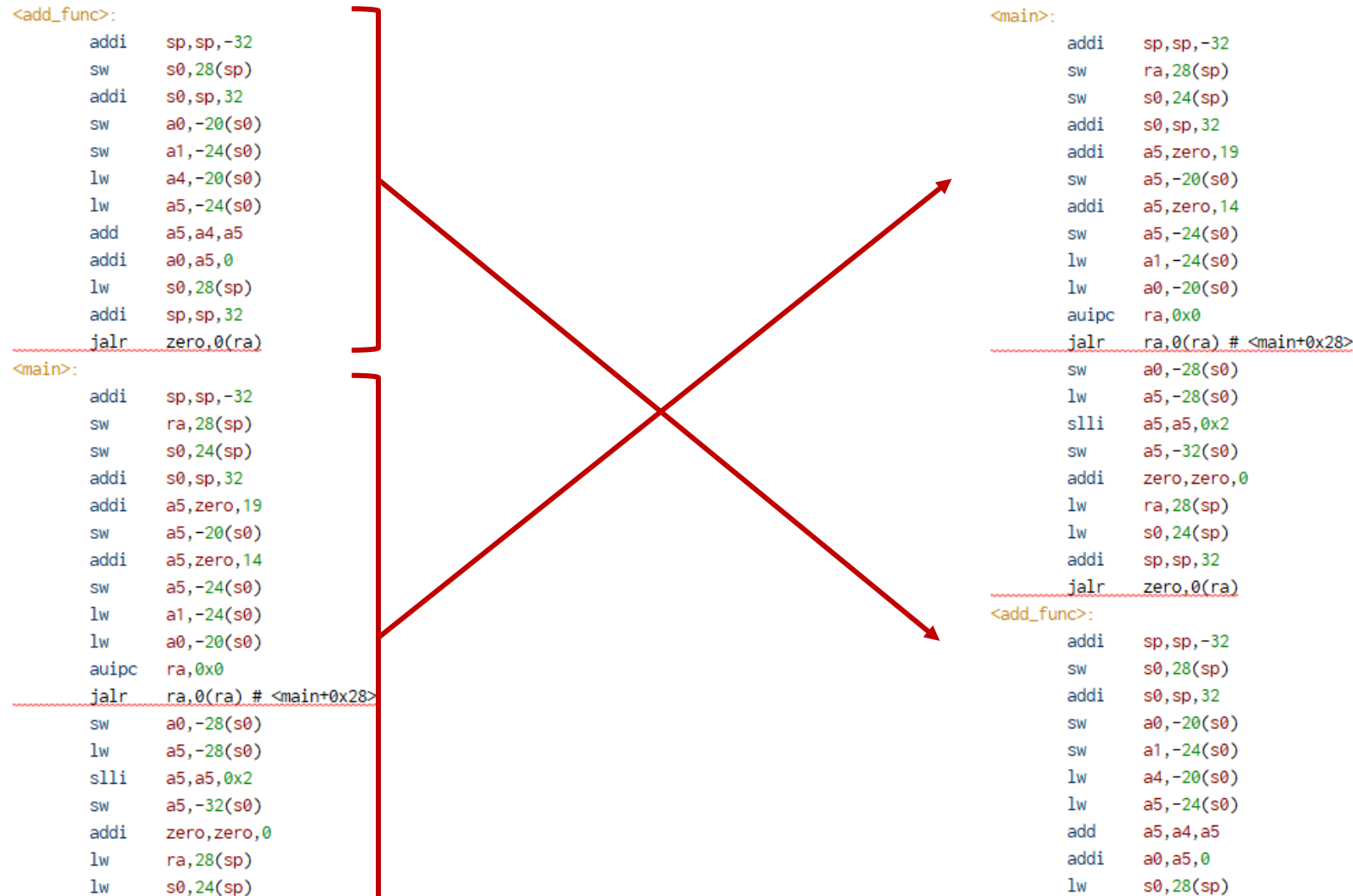
Assembly post-processing in Ripes

1. Make new program with cross-compiler output file ({file_name}_ripes.asm)



Assembly post-processing in Ripes

2. Move the <main> function to the top (because default PC is 0x0 in Ripes)



Assembly post-processing in Ripes

3. Replace 'jalr zero, 0(ra)' at the end of <main> with 'li a7, 10' & 'ecall' (exit inst.)

```
<main>:
    addi    sp,sp,-32
    sw      ra,28(sp)
    sw      s0,24(sp)
    addi    s0,sp,32
    addi    a5,zero,19
    sw      a5,-20(s0)
    addi    a5,zero,14
    sw      a5,-24(s0)
    lw      a1,-24(s0)
    lw      a0,-20(s0)
    auipc   ra,0x0
    jalr    ra,0(ra) # <main+0x28>
    sw      a0,-28(s0)
    lw      a4,-28(s0)
    lw      a5,-20(s0)
    add     a5,a4,a5
    sw      a5,-32(s0)
    addi    a5,zero,0
    addi    a0,a5,0
    lw      ra,28(sp)
    lw      s0,24(sp)
    addi    sp,sp,32
    jalr    zero,0(ra)
```


li a7, 10
ecall

```
<main>:
    addi    sp,sp,-32
    sw      ra,28(sp)
    sw      s0,24(sp)
    addi    s0,sp,32
    addi    a5,zero,19
    sw      a5,-20(s0)
    addi    a5,zero,14
    sw      a5,-24(s0)
    lw      a1,-24(s0)
    lw      a0,-20(s0)
    auipc   ra,0x0
    jalr    ra,0(ra) # <main+0x28>
    sw      a0,-28(s0)
    lw      a4,-28(s0)
    lw      a5,-20(s0)
    add     a5,a4,a5
    sw      a5,-32(s0)
    addi    a5,zero,0
    addi    a0,a5,0
    lw      ra,28(sp)
    lw      s0,24(sp)
    addi    sp,sp,32
    # Exit program
    li      a7, 10
    ecall
```

Assembly post-processing in Ripes

4. For every function call, replace 'auipc ra, 0x0' & 'jalr ra, 0(ra)' with 'jal ra, <add_func>' (target function)

```
<main>:
    addi    sp,sp,-32
    sw      ra,28(sp)
    sw      s0,24(sp)
    addi    s0,sp,32
    addi    a5,zero,19
    sw      a5,-20(s0)
    addi    a5,zero,14
    sw      a5,-24(s0)
    lw      a1,-24(s0)
    lw      a0,-20(s0)
    auipc   ra,0x0
    jalr    ra,0(ra) # <main+0x28>
    sw      a0,-28(s0)
    lw      a4,-28(s0)
    lw      a5,-20(s0)
    add     a5,a4,a5
    sw      a5,-32(s0)
    addi    a5,zero,0
    addi    a0,a5,0
    lw      ra,28(sp)
    lw      s0,24(sp)
    addi    sp,sp,32
    # Exit program
    li     a7, 10
    ecall
```




jal ra,
<add_func>

Assembly post-processing in Ripes

5. For every function return, replace 'jalr zero, 0(ra)' with 'jr ra' (return)

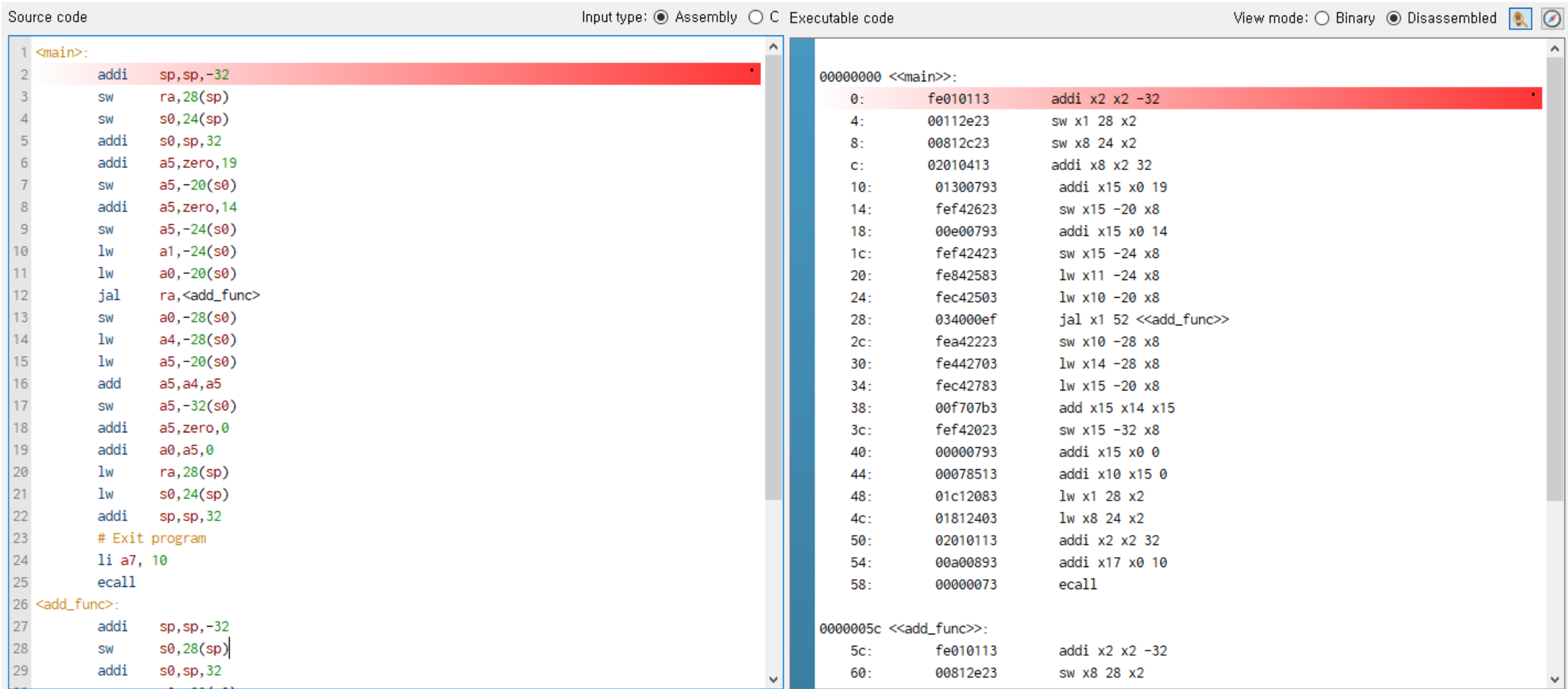
```
<add_func>:  
    addi    sp, sp, -32  
    sw      s0, 28(sp)  
    addi    s0, sp, 32  
    sw      a0, -20(s0)  
    sw      a1, -24(s0)  
    lw      a4, -20(s0)  
    lw      a5, -24(s0)  
    add     a5, a4, a5  
    addi    a0, a5, 0  
    lw      s0, 28(sp)  
    addi    sp, sp, 32  
    jalr    zero, 0(ra)
```



jr ra

Assembly post-processing in Ripes

6. Now you can simulate the source code in Ripes.



The screenshot displays the Ripes tool interface with two main panels. The left panel, titled 'Source code', shows assembly code for a program named <main> and a sub-routine <add_func>. The right panel, titled 'Disassembled', shows the corresponding machine code in hexadecimal and assembly format, with memory addresses on the left.

Input type: ☒ Assembly ☐ C Executable code

View mode: ☐ Binary ☒ Disassembled

Source code:

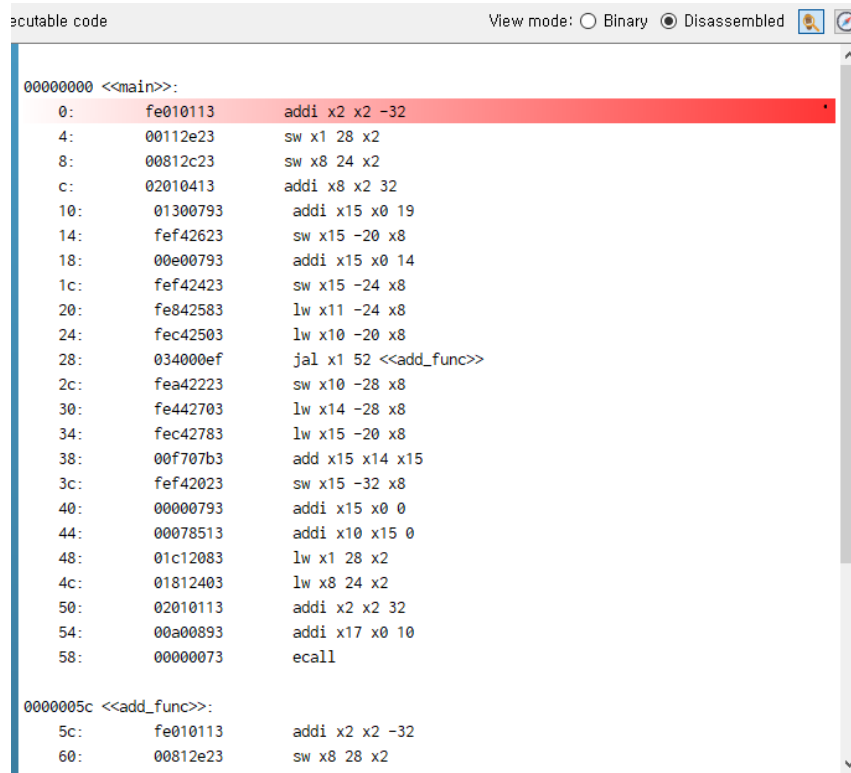
```
1 <main>:  
2     addi    sp,sp,-32  
3     sw      ra,28(sp)  
4     sw      s0,24(sp)  
5     addi    s0,sp,32  
6     addi    a5,zero,19  
7     sw      a5,-20(s0)  
8     addi    a5,zero,14  
9     sw      a5,-24(s0)  
10    lw      a1,-24(s0)  
11    lw      a0,-20(s0)  
12    jal     ra,<add_func>  
13    sw      a0,-28(s0)  
14    lw      a4,-28(s0)  
15    lw      a5,-20(s0)  
16    add     a5,a4,a5  
17    sw      a5,-32(s0)  
18    addi    a5,zero,0  
19    addi    a0,a5,0  
20    lw      ra,28(sp)  
21    lw      s0,24(sp)  
22    addi    sp,sp,32  
23    # Exit program  
24    li      a7,10  
25    ecall  
26 <add_func>:  
27     addi    sp,sp,-32  
28     sw      s0,28(sp)  
29     addi    s0,sp,32
```

Disassembled:

```
00000000 <<main>>:  
0:      fe010113      addi x2 x2 -32  
4:      00112e23      sw x1 28 x2  
8:      00812c23      sw x8 24 x2  
c:      02010413      addi x8 x2 32  
10:     01300793      addi x15 x0 19  
14:     fef42623      sw x15 -20 x8  
18:     00e00793      addi x15 x0 14  
1c:     fef42423      sw x15 -24 x8  
20:     fe842583      lw x11 -24 x8  
24:     fec42503      lw x10 -20 x8  
28:     034000ef      jal x1 52 <<add_func>>  
2c:     fea42223      sw x10 -28 x8  
30:     fe442703      lw x14 -28 x8  
34:     fec42783      lw x15 -20 x8  
38:     00f707b3      add x15 x14 x15  
3c:     fef42023      sw x15 -32 x8  
40:     00000793      addi x15 x0 0  
44:     00078513      addi x10 x15 0  
48:     01c12083      lw x1 28 x2  
4c:     01812403      lw x8 24 x2  
50:     02010113      addi x2 x2 32  
54:     00a00893      addi x17 x0 10  
58:     00000073      ecall  
  
0000005c <<add_func>>:  
5c:     fe010113      addi x2 x2 -32  
60:     00812e23      sw x8 28 x2
```

Assembly post-processing in Ripes

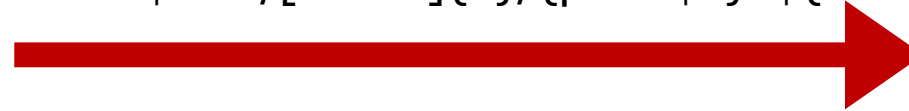
7. Use parser.sh in Docker to extract HEX instruction code from Ripes disassembled instruction code



```
scutable code View mode: Binary Disassembled
00000000 <<main>>:
0: fe010113 addi x2 x2 -32
4: 00112e23 sw x1 28 x2
8: 00812c23 sw x8 24 x2
c: 02010413 addi x8 x2 32
10: 01300793 addi x15 x0 19
14: fef42623 sw x15 -20 x8
18: 00e00793 addi x15 x0 14
1c: fef42423 sw x15 -24 x8
20: fe842583 lw x11 -24 x8
24: fec42503 lw x10 -20 x8
28: 034000ef jal x1 52 <<add_func>>
2c: fea42223 sw x10 -28 x8
30: fe442703 lw x14 -28 x8
34: fec42783 lw x15 -20 x8
38: 00f707b3 add x15 x14 x15
3c: fef42023 sw x15 -32 x8
40: 00000793 addi x15 x0 0
44: 00078513 addi x10 x15 0
48: 01c12083 lw x1 28 x2
4c: 01812403 lw x8 24 x2
50: 02010113 addi x2 x2 32
54: 00a00893 addi x17 x0 10
58: 00000073 ecall

0000005c <<add_func>>:
5c: fe010113 addi x2 x2 -32
60: 00812e23 sw x8 28 x2
```

awk '\$2 ~ /[0-9a-f]{8}/{print \$2}' \${FILE}



```
fe010113
00112e23
00812c23
02010413
01300793
fef42623
00e00793
fef42423
fe842583
fec42503
034000ef
fea42223
fe442703
fec42783
...
```