Lab 3
Assembly Lab II

Video Link : https://youtu.be/p0N8F9C7Fb4

Reference

- **RISC-V Specification**

https://riscv.org/technical/specifications/

- **The RISC-V Reader: An Open Architecture Atlas (Chinese) :**

http://riscvbook.com/chinese/RISC-V-Reader-Chinese-v2p1.pdf

- **Jim's Dev Blog : RISC-V 指令集架構介紹 - RV32I**

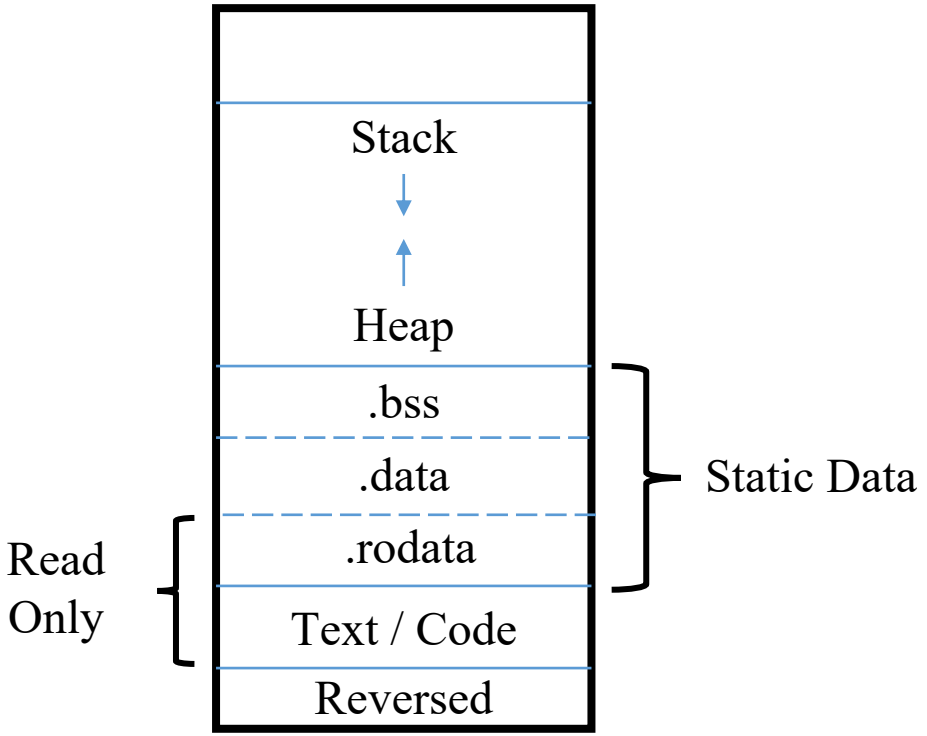
https://tclin914.github.io/16df19b4/

Outline

Assemble Directives – sections

- **.text** :
  code section
- **.data** :
  the section that saves the global variables that were assigned the initialized value (not 0)
- **.bss** (block starting symbol) :
  the section that saves the global variables that were not assigned the initialized value or assigned 0

Directive	Description
`.text`	Subsequent items are stored in the text section (machine code).
`.data`	Subsequent items are stored in the data section (global variables).
`.bss`	Subsequent items are stored in the bss section (global variables initialized to 0).
`.section .foo`	Subsequent items are stored in the section named `.foo`.
`.align n`	Align the next datum on a 2^n-byte boundary. For example, `.align 2` aligns the next value on a word boundary.
`.balign n`	Align the next datum on a n-byte boundary. For example, `.balign 4` aligns the next value on a word boundary.
`.globl sym`	Declare that label sym is global and may be referenced from other files.
`.string "str"`	Store the string str in memory and null-terminate it.
`.byte b1,..., bn`	Store the n 8-bit quantities in successive bytes of memory.
`.half w1,...,wn`	Store the n 16-bit quantities in successive memory halfwords.
`.word w1,...,wn`	Store the n 32-bit quantities in successive memory words.
`.dword w1,...,wn`	Store the n 64-bit quantities in successive memory doublewords.
`.float f1,..., fn`	Store the n single-precision floating-point numbers in successive memory words.
`.double d1,..., dn`	Store the n double-precision floating-point numbers in successive memory doublewords.



Assemble Directives – Preset the data in .data section

Preset the data in .data section

How to preset data ?

- .byte $b_1, b_2, \ldots, b_n$
- .half $h_1, h_2, \ldots, h_n$
- .word $w_1, w_2, \ldots, w_n$
- .dword $d_1, d_2, \ldots, d_n$

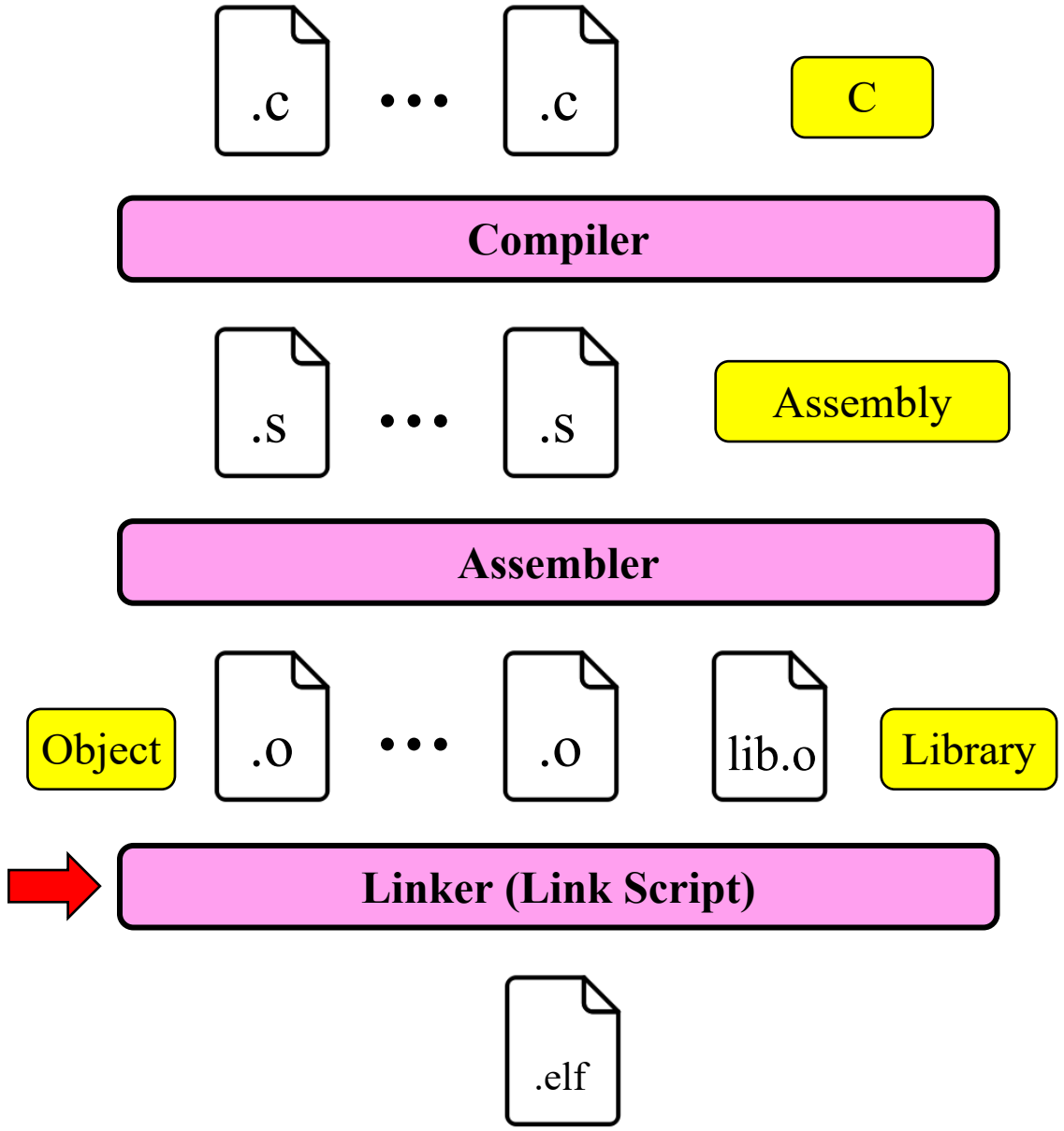
- .align n => "next value" aligns on 2^n bytes
- .balign n => "next value" aligns on n bytes

Where to preset data ?

Edit the **Link Script** to arrange the location of preset data in memory

Link Script : link.ld

```
OUTPUT_ARCH( "riscv" )

SECTIONS
{
    . = 0x00000000;
    .text : { *(.text) }

    . = 0x10000000;
    .data : {*(.data)}

}
```



Assemble Directives – Example of .data sections

Example

```
$ riscv32-unknown-elf-objdump -xsd  example.elf  > example.dump
```

```
.data
    .byte   2, 3, -260          # 0x02, 0x03, 0xfc
    .byte   3, 0xf4, 0xf5982f   # 0x03, 0xf4, 0x2f
    .byte   0b101               # 0x05
    .half   0xef0               # 0x0ef0
    .word   0x1234abcd          # 0x1234abcd
    .align  2                   # next value aligns to 2^2 (4n)
    .byte   0xcd                # 0x01
    .balign 2                   # next value aligns to 2 (2n)
    .byte   0xff                # 0xff
    .align  3                   # next value aligns to 2^3 (8n)
    .byte   0xee                # 0xee
    li      t0, 3               # 0x00300293

.text
main:
    li      t0, 3               # t0 = 3
    li      t1, 4               # t1 = 4
    add     t2, t0, t1          # t2 = t0 + t1 = 7
    ret                         # return
```

.s

```
./CO2021_Lab3/ex1/ex1.s: Assembler messages:
./CO2021_Lab3/ex1/ex1.s:2: Warning: value 0xfffffffffffffefc truncated to 0xfc
./CO2021_Lab3/ex1/ex1.s:3: Warning: value 0xf5982f truncated to 0x2f
```

```
Contents of section .text:
 0000 93023000 13034000 b3836200 67800000  ..0...@...b.g...
Contents of section .data:
 10000000 0203fc03 f42f05f0 0ecdab34 12000000  ...../.....4....
 10000010 cd00ff00 00000000 ee930230 00        ...........0.
Contents of section .riscv.attributes:
 0000 41190000 00726973 63760001 0f000000  A....riscv......
 0010 05727633 32693270 3000               .rv32i2p0.

Disassembly of section .text:

00000000 <main>:
   0:   00300293            li   t0,3
   4:   00400313            li   t1,4
   8:   006283b3            add  t2,t0,t1
   c:   00008067            ret
```

Little Endian

.dump

Assemble Directives – Example of .data sections

Example

```
$ riscv32-unknown-elf-objdump -xsd  example.elf  > example.dump
```

```
.data
    .byte   2, 3, -260          # 0x02, 0x03, 0xfc
    .byte   3, 0xf4, 0xf5982f   # 0x03, 0xf4, 0x2f
    .byte   0b101               # 0x05
    .half   0xef0               # 0x0ef0
    .word   0x1234abcd          # 0x1234abcd
    .align  2                   # next value aligns to 2^2 (4n)
    .byte   0xcd                # 0xcd
    .balign 2                   # next value aligns to 2 (2n)
    .byte   0xff                # 0xff
    .align  3                   # next value aligns to 2^3 (8n)
    .byte   0xee                # 0xee
    li      t0, 3               # 0x00300293

.text
main:
    li      t0, 3               # t0 = 3
    li      t1, 4               # t1 = 4
    add     t2, t0, t1          # t2 = t0 + t1 = 7
    ret                         # return
```

.s

```
./CO2021_Lab3/ex1/ex1.s: Assembler messages:
./CO2021_Lab3/ex1/ex1.s:2: Warning: value 0xfffffffffffffefc truncated to 0xfc
./CO2021_Lab3/ex1/ex1.s:3: Warning: value 0xf5982f truncated to 0x2f
```

```
Contents of section .text:
 0000 93023000 13034000 b3836200 67800000  ..0...@...b.g...
Contents of section .data:
 10000000 0203fc03 f42f05f0 0ecdab34 12000000  ...../.....4....
 10000010 cd00ff00 00000000 ee930230 00        ...........0.
Contents of section .riscv.attributes:
 0000 41190000 00726973 63760001 0f000000  A....riscv......
 0010 05727633 32693270 3000               .rv32i2p0.

Disassembly of section .text:

00000000 <main>:
   0:   00300293                li    t0,3
   4:   00400313                li    t1,4
   8:   006283b3                add   t2,t0,t1
   c:   00008067                ret
```

Little Endian

.dump

Label – How to get the address of preset data

Label

- A symbol represents the **next address**
- It's not an instruction
- **Use la(load address, pseudo instruction) to get the address of LABEL**
  => la rd, LABEL

```
.data
a:  .byte   2, 3, -260          # 0x02, 0x03, 0xfc
b:  .byte   3, 0xf4, 0xf5982f   # 0x03, 0xf4, 0x2f
    .byte   0b101               # 0x05
c:  .half   0xef0               # 0x0ef0
d:  .word   0x1234abcd          # 0x1234abcd
e:  .align  2                   # next value aligns to 2^2 (4n)
f:  .byte   0xcd                # 0x01
    .balign 2                   # next value aligns to 2 (2n)
    .byte   0xff                # 0xff

.text
main:
    la      t0, a           # t0 = Addr(a)
    la      t1, b           # t1 = Addr(b)
    la      t2, c           # t2 = Addr(c)
    la      t3, d           # t3 = Addr(d)
    la      t4, e           # t4 = Addr(e)
    la      t5, f           # t5 = Addr(f)
    la      s0, main        # s0 = Addr(main)
    la      s1, loadData    # s1 = Addr(loadData)
loadData:
    lb      s0, 1(t0)       # s0 = 0x00000003
    lh      s1, 1(t0)       # s1 = 0xfffffc03
    lw      s2, 1(t0)       # s2 = 0xf403fc03
    ret                     # return
```

```
Contents of section .text:
 0000 97020010 93820200 17030010 1303b3ff  ................
 0010 97030010 938373ff 170e0010 130e1eff  ......s.........
 0020 970e0010 938edefe 170f0010 130f8ffe  ................
 0030 17040000 130404fd 97040000 93848400  ................
 0040 03841200 83941200 03a91200 67800000  ............g...
Contents of section .data:
 10000000 0203fc03 f42f05f0 0ecdab34 12000000  ...../.....4....
 10000010 cd00ff                               ...
Contents of section .riscv.attributes:
 0000 41190000 00726973 63760001 0f000000  A....riscv......
 0010 05727633 32693270 3000               .rv32i2p0.

Disassembly of section .text:

00000000 <main>:
   0: 10000297            auipc t0,0x10000
   4: 00028293            mv  t0,t0
   8: 10000317            auipc t1,0x10000
   c: ffb30313            addi  t1,t1,-5 # 10000003 <b>
  10: 10000397            auipc t2,0x10000
  14: ff738393            addi  t2,t2,-9 # 10000007 <c>
  18: 10000e17            auipc t3,0x10000
  1c: ff1e0e13            addi  t3,t3,-15 # 10000009 <d>
  20: 10000e97            auipc t4,0x10000
  24: fede8e93            addi  t4,t4,-19 # 1000000d <e>
  28: 10000f17            auipc t5,0x10000
  2c: fe8f0f13            addi  t5,t5,-24 # 10000010 <f>
  30: 00000417            auipc s0,0x0
  34: fd040413            addi  s0,s0,-48 # 0 <main>
  38: 00000497            auipc s1,0x0
  3c: 00848493            addi  s1,s1,8 # 40 <loadData>

00000040 <loadData>:
  40: 00128403            lb  s0,1(t0) # 10000001 <a+0x1>
  44: 00129483            lh  s1,1(t0)
  48: 0012a903            lw  s2,1(t0)
  4c: 00008067            ret
```

Control Transfer Introduction

Instruction classification of RV32I depends on functionality

- Computation Instruction
  - Register – Register
  - Register – Immediate
  - Long Immediate

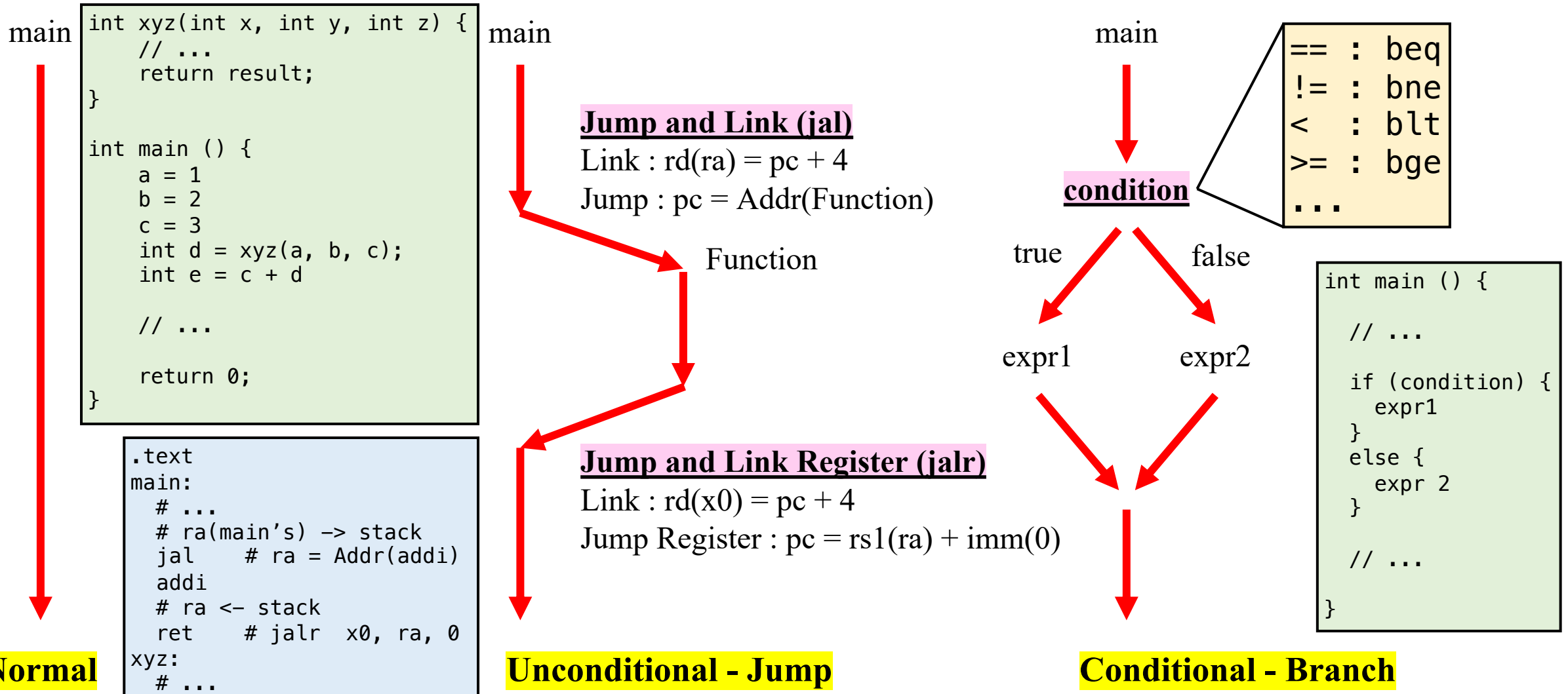
Arithmetic **Set** **Shift** **Logical**

(add, **sub**, slt, sltu, sll, srl, sra, xor, or, and)
(addi, slti, sltiu, slli, srli, srai, xori, ori, andi)
(lui, auipc)

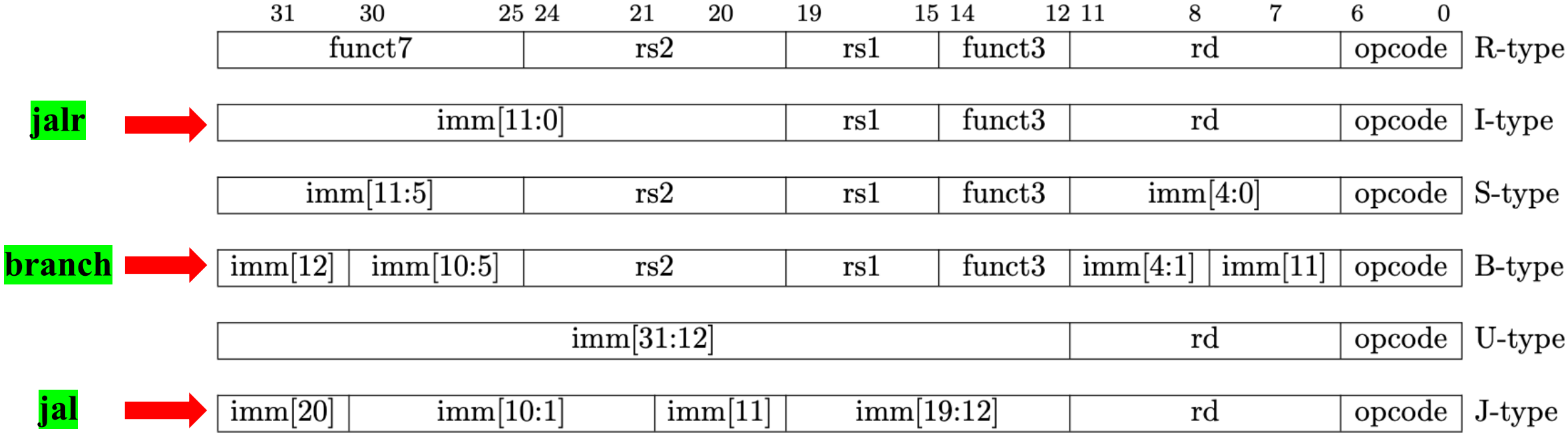
- Load & Store Instruction
  - Load (lb, lh, lw, lbu, lhu)
  - Store (sb, sh, sw)

- Control Transfer Instruction
  - Unconditional Jump (jal, jalr)
  - Conditional Branch (beq, bne, blt, bge, bltu, bgeu)

Functionality of Control Transfer Instructions



Instruction classification of RV32I depends on functionality



jal (Jump and link)

Link : rd = Addr(next instruction) = pc + 4

Jump : pc = Addr(Jump Target) = pc + imm (offset)

jalr (Jump and link register)

Link : rd = Addr(next instruction) = pc + 4

Jump : pc = Addr(Jump Target) = rs1 + imm (offset)

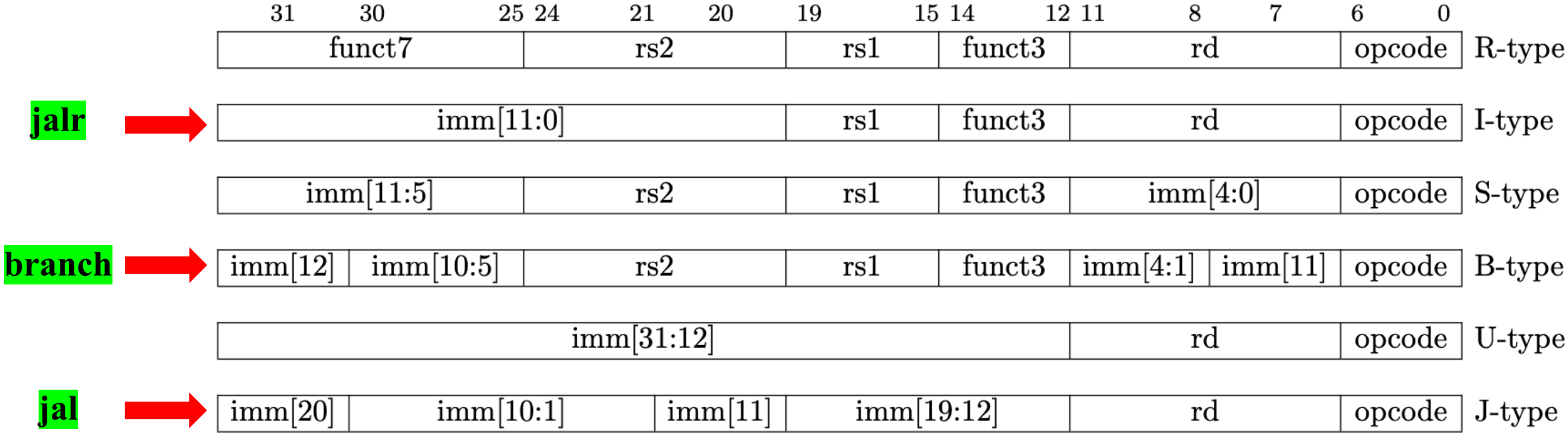
Branch

pc = (**condition**) ? Addr(Jump Target) : Addr(next instruciton)

= (compare rs1 & rs2) ? pc + imm (offset) : pc + 4

PC-relative Addressing mode

Instruction classification of RV32I depends on functionality

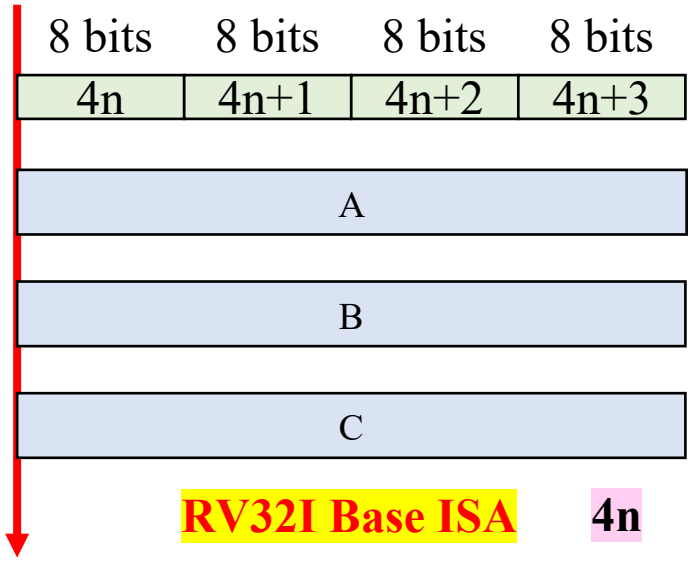


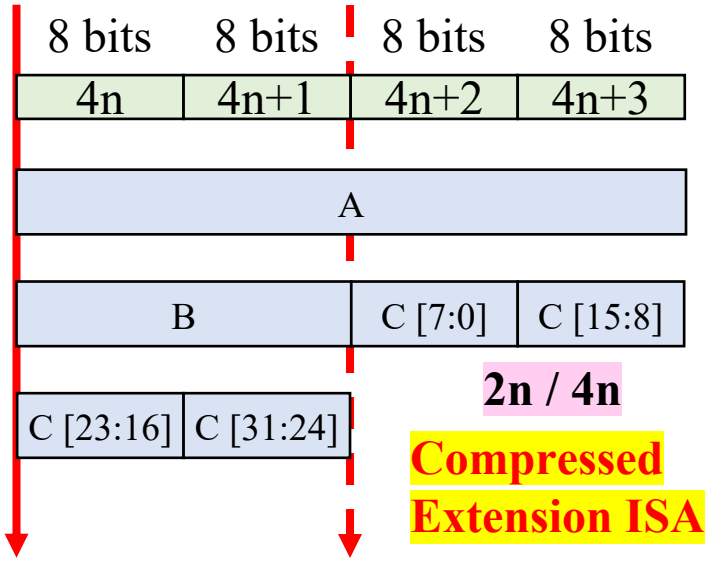
(Kibibyte / Mebibyte) **10 bits = 1KiB**

Immediate ±1MiB

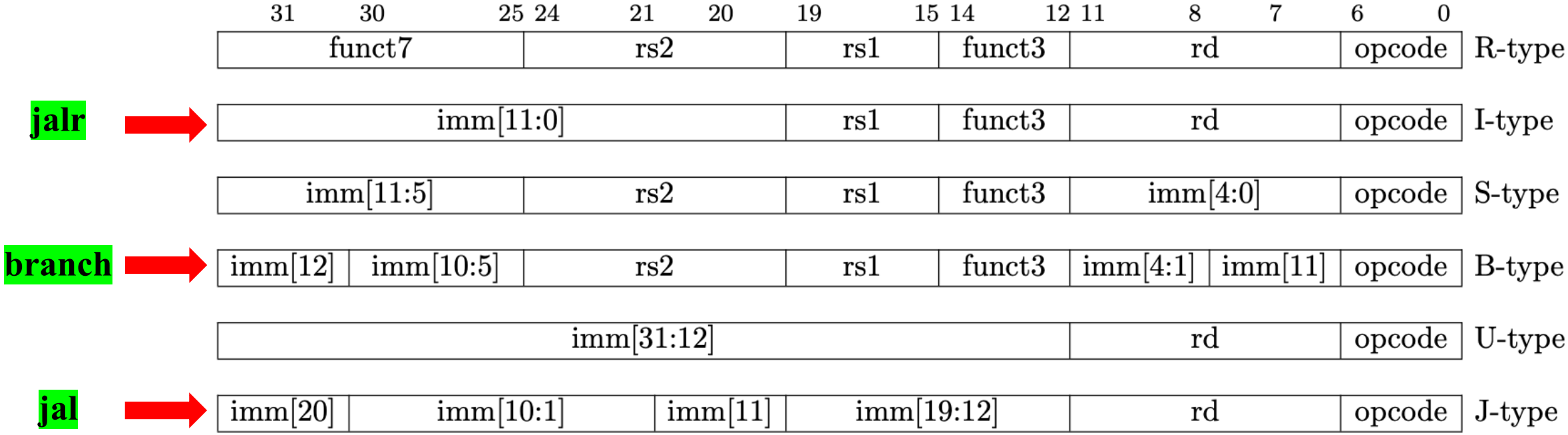
- jal (J type) : imm[20:1]
- branch (B type) : imm[12:1] ±4KiB

jal / branch : imm[0] is always zero





Instruction classification of RV32I depends on functionality



(Kibibyte / Mebibyte) **10 bits = 1KiB**

Immediate

- jal (J type) : imm[20:1] ±1MiB
- jalr (I type) : imm[11:0]
- branch (B type) : imm[12:1] ±4KiB

jal / branch : imm[0] is always zero

Q1 :
Why the immediate of jal & branch doesn't include imm[0] ?
What's the advantage of this design?
Why we need jalr ? (Hint : What's the limitation of jal ?)

HINT : IALIGN can be only 16 or 32

Control Transfer Instruction – (Unconditional)

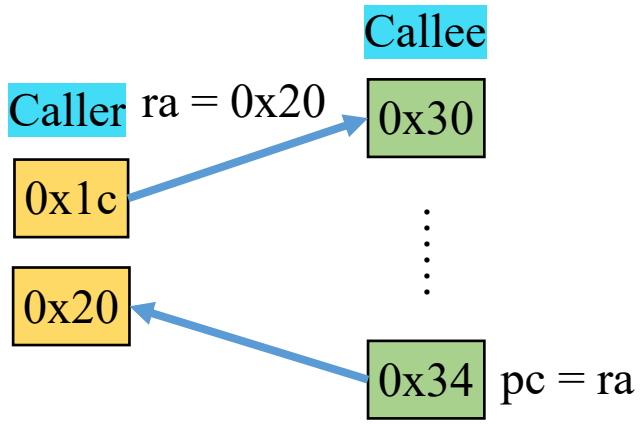
Unconditional operation (J / I)

- jal (Jump And Link, **J type**)
  - rd = pc + 4
  - pc = pc + sext({imm20, 1'b0})
    (or pc = LABEL)
- jalr (Jump And Link Register, **I type**)
  - rd = pc + 4
  - pc = (rs1 + sext(imm12)) & (~32'd1)

Jump format

jal rd, imm20
✓ **jal rd, LABEL**
✓ **jalr rd, rs1, imm12**
jalr rd, imm12(rs1)

```
5  main:
6      ####  Call Function Procedure
7      ##  Caller Saved
8      addi  sp, sp, -4     # Allocate stack space
9      sw    ra, 0(sp)      # Store ra into stack
10     ##  Pass Arguments
11     li    a0, 5          # a0 = 5
12     li    a1, 6          # a1 = 6
13     ##  Jump to Callee
14     jal   ra, func_ADD   # ra = Addr(mv  t0, a0)
15     #### ####
16     mv    t0, a0         # Get return value
17     lw    ra, 0(sp)      # Retrieve ra from stack
18     addi  sp, sp, 4      # Release stack space
19     ret                  # jalr  x0, ra, 0
20 func_ADD:
21     #### a0 : return value
22     #### a0 = a0 + a1
23     add   a0, a0, a1     # a0 = a0 + a1
24     ret                  # jalr  x0, ra, 0
```




```
0000000c <main>:
    c:          ffc10113          addi x2 x2 -4
    10:          00112023           sw x1 0 x2
    14:          00500513           addi x10 x0 5
    18:          00600593           addi x11 x0 6
    1c:          014000ef           jal x1 20 <func_ADD>
    20:          00050293           addi x5 x10 0
    24:          00012083           lw x1 0 x2
    28:          00410113           addi x2 x2 4
    2c:          00008067           jalr x0 x1 0

00000030 <func_ADD>:
    30:          00b50533           add x10 x10 x11
    34:          00008067           jalr x0 x1 0
```

Control Transfer Instruction – (Unconditional)

- "rd = pc + 4" can make us save the return point, usually rd = x1(ra)
- If we don't want to save return point, we can just set rd = x0
- **(Recommend !!!) Always use LABEL**

```
main:
    jal     x0, JumpToHere  # Jump to JumpToHere
    jal     x0, 0x8         # May be jump to unexpected address
    li      t1, 2048        # t1 = 2048
JumpToHere:
    ret                     # jalr  x0, ra, 0
```

```
0000000c <main>:
    c:        0100006f        jal x0 16 <JumpToHere>
    10:        0080006f        jal x0 8
    14:        00001337        lui x6 0x1
    18:        80030313        addi x6 x6 -2048

0000001c <JumpToHere>:
    1c:        00008067        jalr x0 x1 0
```

- **Function call**
  - **jal** is usually used in function call with the jump address is ±1MiB of pc

10 bits = 1KiB

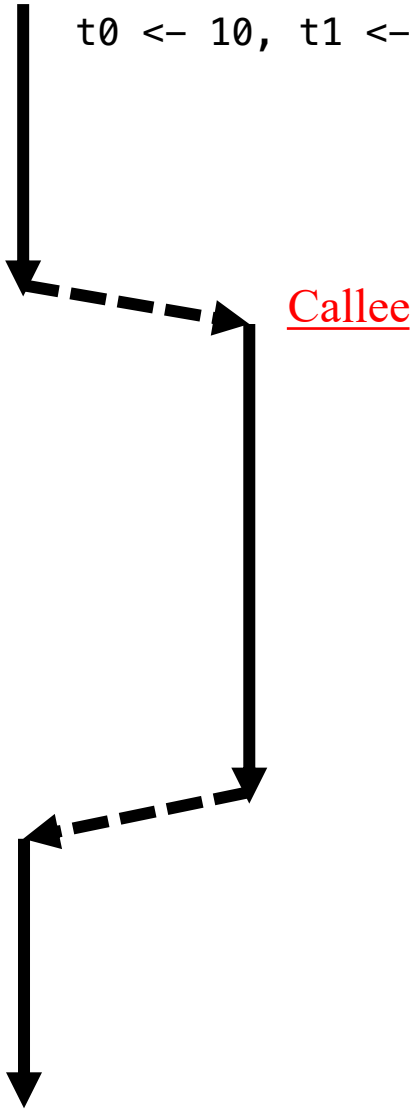
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

  - When the jump address > ±1MiB of pc, we can use two instructions to combine the jump address
    - lui(High 20 bits) + jalr(Low 12 bits) => Absolute address
    - auipc(High 20 bits) + jalr(Low 12 bits) => PC-relative address
- **Function return**
  - **jalr** is usually used in function return because the return address is usually stored in x1(ra)

Function Call – Caller & Callee

Caller

```
t0 <- 10, t1 <- 5, s0 <- 7
```



Callee

parameter

```
int xyz(int x, int y, int z) {
    int result = x + y*2 + z*3;
    return result;
}

int main () {
    int a = 10;
    int b = 5;
    int c = 7;
    int d = xyz(a, b, c);
    int e = a + c + d;
    return 0;
}
```

argument

```
t0(a)  t1(b)  s0(c)

a0(x)  a3(y)  a4(z)

Some instructions
to calculate
result

a0 (result)
```

The programmer of Callee

- Determine which argument register(a0~a7) represents which parameter
  => e.g. a0 is x, a3 is y, a4 is z
- Determine which argument register(s)(a0~a1) to put the return value into
  => e.g. a0 is result

Function Call – Caller & Callee

Caller

```
t0 <- 10, t1 <- 5, s0 <- 7
a0 <- t0, a3 <- t1, a4 <- s0
```

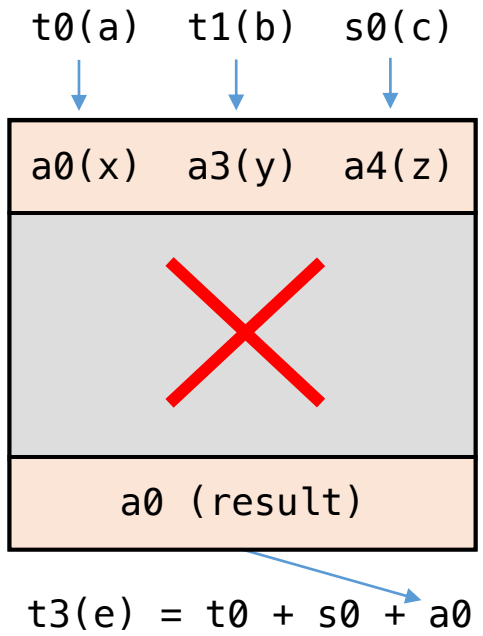
Callee

```
t3 <- t0 + s0 + a0
```

parameter

```
int xyz(int x, int y, int z) {
    int result = x + y*2 + z*3;
    return result;
}

int main () {
    int a = 10;
    int b = 5;
    int c = 7;
    int d = xyz(a, b, c);
    int e = a + c + d;
    return 0;
}
```

argument



The programmer of Callee

- Determine which argument register(a0~a7) represents which parameter
  => e.g. a0 is x, a3 is y, a4 is z
- Determine which argument register(s)(a0~a1) to put the return value into
  => e.g. a0 is result

The programmer of Caller

- Know which argument registers(a0~a7) the arguments need to be put into
- Know which argument register(s) represent the return value of the function
- But don't know the implement detail of the callee

Function Call – Caller & Callee

Caller

```
t0 <- 10, t1 <- 5, s0 <- 7
a0 <- t0, a3 <- t1, a4 <- s0
```

Caller knows the values of all registers it uses

Callee

Callee may change some registers here

```
t0 <- a3 * 2    # t0 = y*2
t1 <- a4 * 3    # t1 = z*3
s0 <- t0 + t1   # s0 = y*2 + z*3
s1 <- a0 + s0   # s1 = x + s0
a0 <- s1        # a0 = s1
```

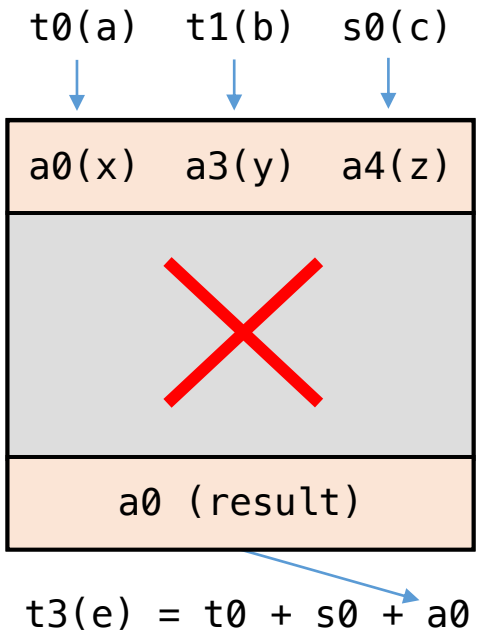
Some registers were changed
However, Caller doesn't know

```
t3 <- t0 + s0 + a0
```

parameter

```
int xyz(int x, int y, int z) {
    int result = x + y*2 + z*3;
    return result;
}

int main () {
    int a = 10;
    int b = 5;
    int c = 7;
    int d = xyz(a, b, c);
    int e = a + c + d;
    return 0;
}
```

argument



The programmer of Callee

- Determine which argument register(a0~a7) represents which parameter
  => e.g. a0 is x, a3 is y, a4 is z
- Determine which argument register(s)(a0~a1) to put the return value into
  => e.g. a0 is result

The programmer of Caller

- Know which argument registers(a0~a7) the arguments need to be put into
- Know which argument register(s) represent the return value of the function
- But don't know the implement detail of the callee

Function Call – Calling Convention

Caller

```
t0 <- 10, t1 <- 5, s0 <- 7
a0 <- t0, a3 <- t1, a4 <- s0
```

Caller knows the values of all registers it uses

Callee

Callee may change some registers here

```
t0 <- a3 * 2    # t0 = y*2
t1 <- a4 * 3    # t1 = z*3
s0 <- t0 + t1   # s0 = y*2 + z*3
s1 <- a0 + s0   # s1 = x + s0
a0 <- s1        # a0 = s1
```

Some registers were changed
However, Caller doesn't know

```
t3 <- t0 + s0 + a0
```

Problem

- There will be possibly produce unexpected result after a function return if the function change the value of the registers used by the Caller

Idea

- We want the value of the registers used by the Caller can be unchanged before jumping to the Callee and after returning to the Caller

Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

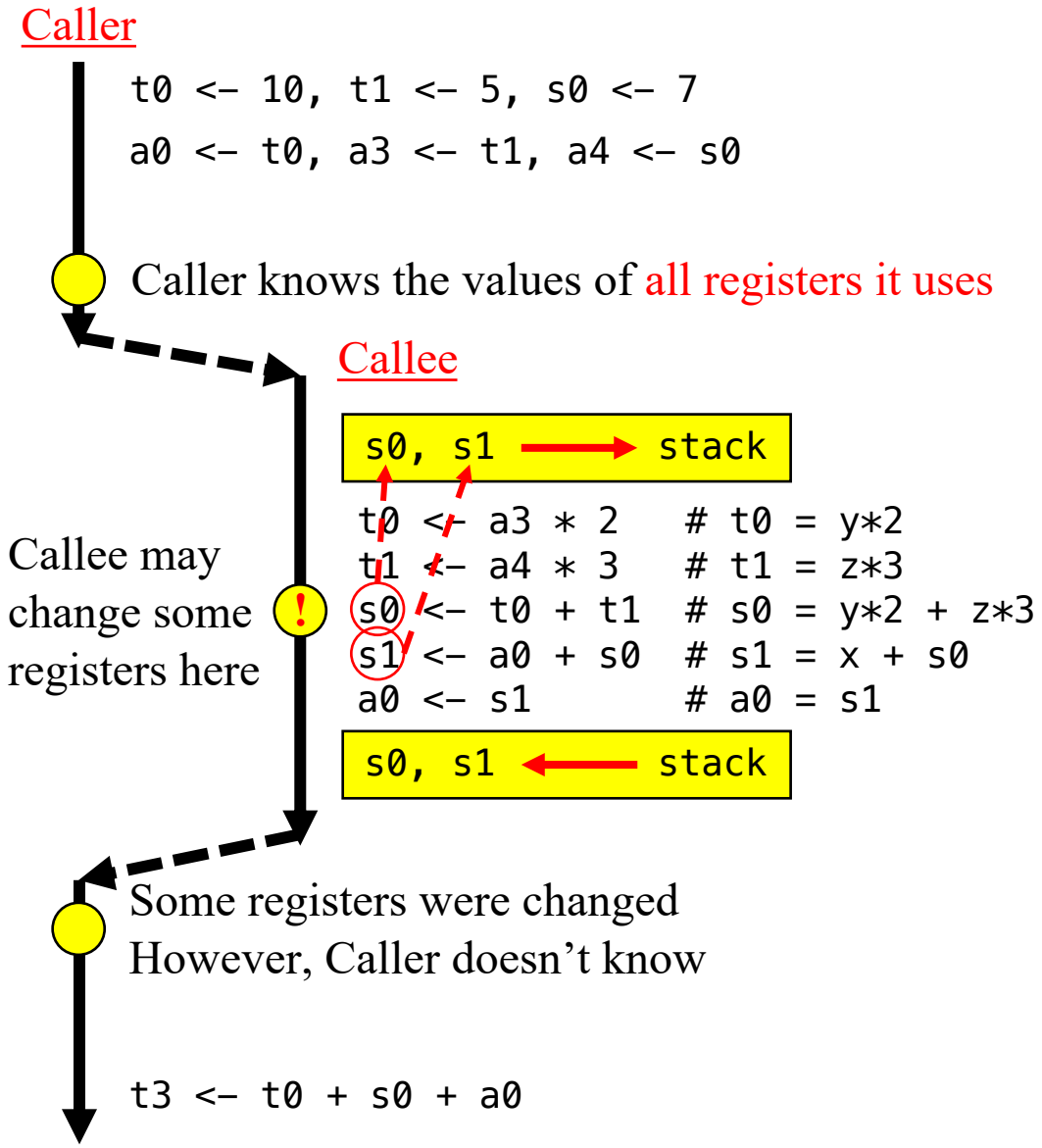
What we need to do :

Obey the Calling Convention defined by RISC-V

By RISC-V Calling Convention, the registers are divided into 2 types

1. Callee-Saved Registers
2. Caller-Saved Registers

Function Call – Calling Convention (Callee-Saved)



Callee-Saved Registers:

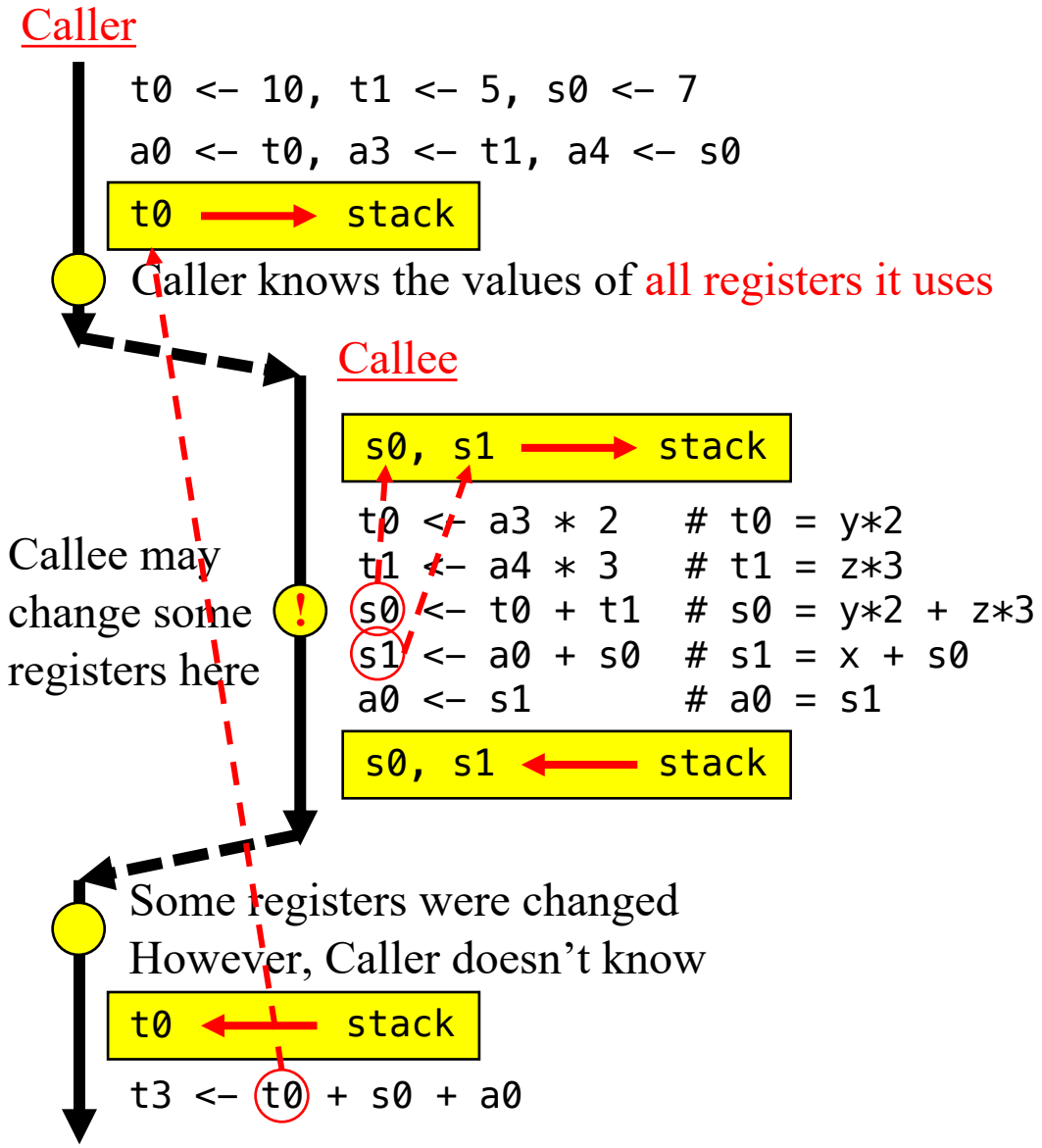
- For the Caller,
  if the programmer of the Caller wants the used callee-saved registers to have the same value before jumping to the Callee and after returning from the Callee
  => The programmer of the Caller doesn't need to do anything
  => The programmer of the Callee is responsible for ensuring this

- The programmer of the Callee doesn't know which callee-saved registers are used by Caller
  => Assume all callee-saved registers are used by the Caller

- The programmer of the Callee ensures that all callee-saved register have the same values at the beginning and the end of the Callee

- If the programmer of the Callee wants to change the value of the callee-saved registers, the programmer need to
  1. Save the original value of the callee-saved registers which he/she wants to use to the stack before changing them
  2. Can use the callee-saved register arbitrarily after saving them
  3. Retrieve the original value of the callee-saved registers saved before from the stack before returning to the Caller

Function Call – Calling Convention (Caller-Saved)



Caller-Saved Registers:

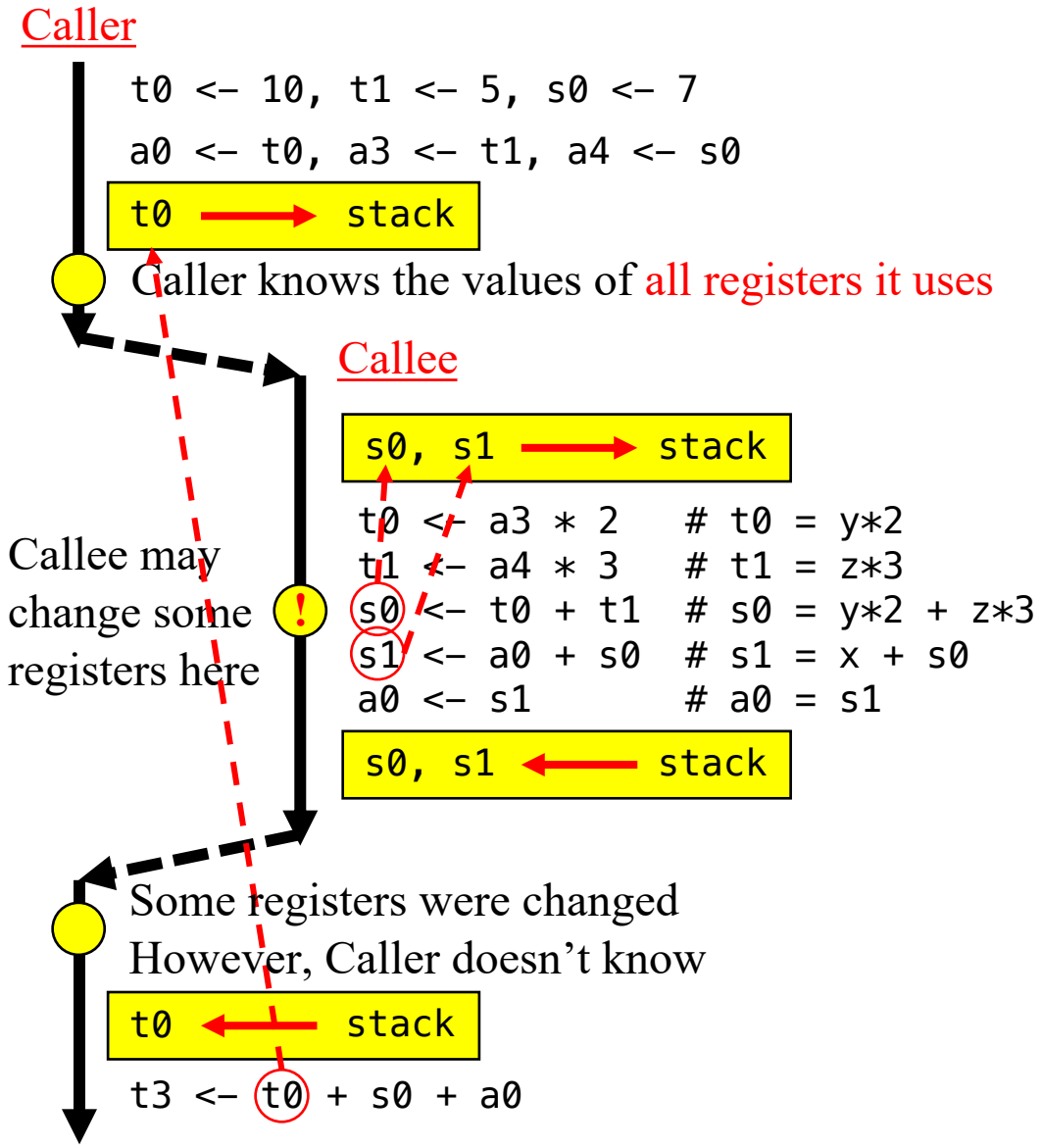
- Besides the Callee-Saved Registers, all other registers(caller-saved registers) can be arbitrarily changed by the Callee without saving and retrieving

- For the Caller,
  if the programmer of the Caller wants the caller-saved registers to have the same value before jumping to the Callee and after returning to the Caller
  => The programmer of the Callee are not responsible for ensuring this
  => The programmer of the Caller need to save them by himself/herself

- The programmer of the Caller doesn't know which caller-saved registers would be changed by the Callee
  => Assume all caller-saved registers have the potential to be changed by the Callee

- Steps
  1. Save the value of the caller-saved registers which the programmer of the Caller wants to keep before jumping to the Callee and after returning from the Callee to the stack
  2. Retrieve the caller-saved registers saved before from the stack after returning from the Callee

Function Call – Calling Convention (Callee-Saved & Caller-Saved)



Caller – Before function call

1. Caller saved (stack):
   Save the caller-saved register we want to keep before jumping to the Callee and after returning from the Callee to the stack
2. Pass arguments through a0 ~ a7
3. Jump to function meanwhile save the return address (pc+4)
   jal ra, LABEL # ra = pc + 4, pc = LABEL

Callee

Every function is a Callee (include main)

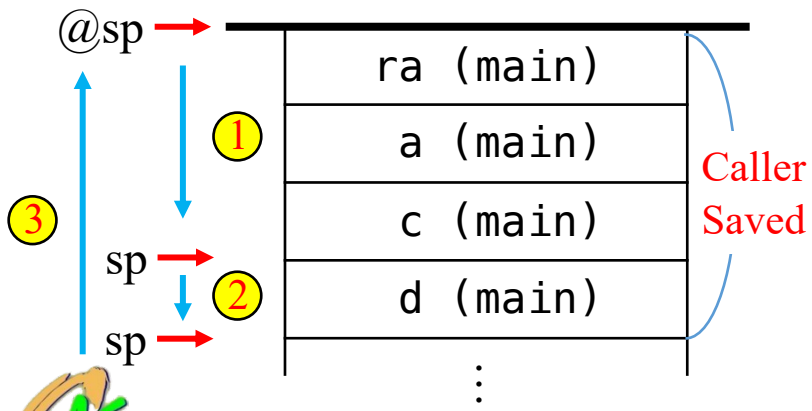
1. Callee saved (stack)
2. /* Computation / Call another function */
3. Pass return value through a0 ~ a1
4. Retrieve return address(ra) & callee saved from stack
5. Return
   jalr x0, ra, 0 # ~~x0 = pc + 4~~, pc = ra + 0

Caller – After function call

1. Get return value from a0 ~ a1
2. Retrieve the caller saved from stack

Function Call – Example (1/3)

```c
int xyz(int x, int y, int z) {
    int result = x + y*2 + z*3;
    return result;
}

int main () {
    int a = 10;
    int b = 5;
    int c = 7;
    int d = xyz(a, b, c);
    int e = xyz(a, c, d);
    int f = a + d + e;
    return 0;
}
```

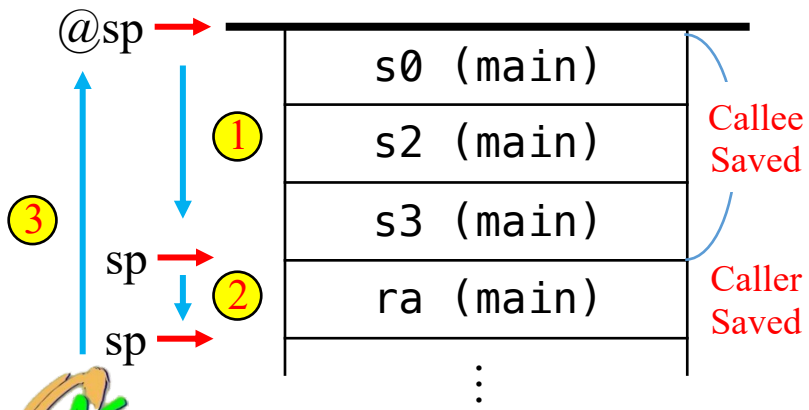



```
6  main:
7      ################################################
8      # a : t0
9      # b : t1
10     # c : t2
11     ################################################
12
13     li    t0, 10          # t0(a) = 10
14     li    t1, 5           # t1(b) = 5
15     li    t2, 7           # t2(c) = 7
16
17     ############ Call Function Procedure ###########
18     # Caller Saved
19     addi  sp, sp, -12     # Allocate stack space
20                           # sp = @sp - 12
21     sw    ra, 8(sp)       # @ra -> MEM[@sp-4]
22     sw    t0, 4(sp)       # t0(a) -> MEM[@sp-8]
23     sw    t2, 0(sp)       # t2(c) -> MEM[@sp-12]
24
25     # Pass Arguments
26     mv    a0, t0          # a0(x) = t0(a)
27     mv    a1, t1          # a1(y) = t1(b)
28     mv    a2, t2          # a2(z) = t2(c)
29
30     # Jump to Callee
31     jal   ra, FUNC_XYZ    # ra = Addr(lw  t0, 4(sp))
32     ################################################
33
34     ## Retrieve Caller Saved
35     lw    t0, 4(sp)       # t0(a) = 10
36     lw    t2, 0(sp)       # t2(c) = 7
37
38     ############ Call Function Procedure ###########
39     # Caller Saved
40     addi  sp, sp, -4      # Allocate stack space
41                           # sp = @sp - 16
42     sw    a0, 0(sp)       # a0(d) -> MEM[@sp-16]
43
44     # Pass Arguments
45     mv    a2, a0          # a2(z) = a0(d)
```

```
46     mv    a0, t0          # a0(x) = t0(a)
47     mv    a1, t2          # a1(y) = t2(c)
48
49     # Jump to Callee
50     jal   ra, FUNC_XYZ    # ra = Addr(lw  t0, 8(sp))
51     ################################################
52
53     ## Retrieve Caller Saved
54     lw    t0, 8(sp)       # t0(a) = 10
55     lw    t3, 0(sp)       # t3(d)
56
57     add   t4, t0, t3      # t4(f) = t0(a) + t3(d)
58     add   t4, t4, a0      # t4(f) = t4 + a0(e)
59
60     ## Retrieve ra & Retrieve Callee Saved
61     lw    ra, 12(sp)      # ra = @ra
62     addi  sp, sp, 16      # Release stack space
63                           # sp = @sp
64     ## return
65     ret                   # jalr  x0, ra, 0
66
67 FUNC_XYZ:
68     ################################################
69     # x : a0
70     # y : a1
71     # z : a2
72     # return value : a0
73     ################################################
74
75     slli  t1, a1, 1       # t1 = y * 2
76     slli  t2, a2, 1       # t2 = z * 2
77     add   t2, t2, a2      # t2 = z * 3
78
79     ## Pass Return Value
80     add   a0, a0, t1      # a0 = x + y*2
81     add   a0, a0, t2      # a0 = x + y*2 + z*3
82
83     ## return
84     ret                   # jalr  x0, ra, 0
```

Function Call – Example (2/3)

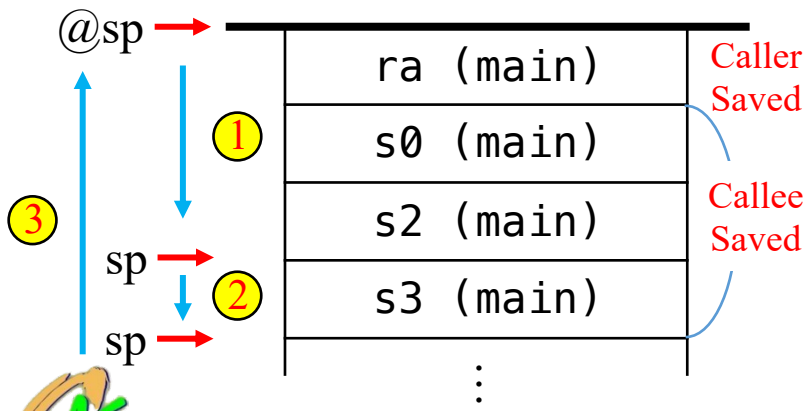
```c
int xyz(int x, int y, int z) {
    int result = x + y*2 + z*3;
    return result;
}

int main () {
    int a = 10;
    int b = 5;
    int c = 7;
    int d = xyz(a, b, c);
    int e = xyz(a, c, d);
    int f = a + d + e;
    return 0;
}
```




```asm
main:
    ################################################
    # a : s0
    # b : t1
    # c : s2
    # d : s3
    ################################################

    ## Callee Saved
    addi  sp, sp, -12     # Allocate stack space
                          # sp = @sp - 12
    sw    s0, 8(sp)       # @s0 -> MEM[@sp-4]
    sw    s2, 4(sp)       # @s2 -> MEM[@sp-8]
    sw    s3, 0(sp)       # @s3 -> MEM[@sp-12]

    li    s0, 10          # s0(a) = 10
    li    t1, 5           # t1(b) = 5
    li    s2, 7           # s2(c) = 7

    ############ Call Function Procedure ###########
    # Caller Saved
    addi  sp, sp, -4      # Allocate stack space
                          # sp = @sp - 16
    sw    ra, 0(sp)       # @ra -> MEM[@sp-16]

    # Pass Arguments
    mv    a0, s0          # a0(x) = s0(a)
    mv    a1, t1          # a1(y) = t1(b)
    mv    a2, s2          # a2(z) = s2(c)

    # Jump to Callee
    jal   ra, FUNC_XYZ    # ra = Addr(mv  s3, a0)
    ################################################

    mv    s3, a0          # s3(d) = a0(return value)

    ############ Call Function Procedure ###########
    # Caller Saved

    # Pass Arguments
    mv    a0, s0          # a0(x) = s0(a)
    mv    a1, s2          # a1(y) = s2(c)
    mv    a2, s3          # a2(z) = s3(d)

    # Jump to Callee
    jal   ra, FUNC_XYZ    # ra = Addr(add  t4, s0, s3)
    ################################################

    add   t4, s0, s3      # t4(f) = s0(a) + s3(d)
    add   t4, t4, a0      # t4(f) = t4 + a0(e)

    ## Retrieve ra & Retrieve Callee Saved
    lw    s0, 12(sp)      # s0 = @s0
    lw    s2, 8(sp)       # s2 = @s2
    lw    s3, 4(sp)       # s3 = @s3
    lw    ra, 0(sp)       # ra = @ra
    addi  sp, sp, 16      # Release stack space
                          # sp = @sp
    ## return
    ret                   # jalr  x0, ra, 0

FUNC_XYZ:
    ################################################
    # x : a0
    # y : a1
    # z : a2
    # return value : a0
    ################################################

    slli  t1, a1, 1       # t1 = y * 2
    slli  t2, a2, 1       # t2 = z * 2
    add   t2, t2, a2      # t2 = z * 3

    ## Pass Return Value
    add   a0, a0, t1      # a0 = x + y*2
    add   a0, a0, t2      # a0 = x + y*2 + z*3

    ## return
    ret                   # jalr  x0, ra, 0
```

Function Call – Example (3/3)

```c
int xyz(int x, int y, int z) {
    int result = x + y*2 + z*3;
    return result;
}

int main () {
    int a = 10;
    int b = 5;
    int c = 7;
    int d = xyz(a, b, c);
    int e = xyz(a, c, d);
    int f = a + d + e;
    return 0;
}
```




```asm
main:
    ################################################
    # a : s0
    # b : t1
    # c : s2
    # d : s3
    ################################################

    ## Save ra & Callee Saved
    addi  sp, sp, -16     # Allocate stack space
                          # sp = @sp - 16
    sw    ra, 12(sp)      # @ra -> MEM[@sp-4]
    sw    s0, 8(sp)       # @s0 -> MEM[@sp-8]
    sw    s2, 4(sp)       # @s2 -> MEM[@sp-12]
    sw    s3, 0(sp)       # @s3 -> MEM[@sp-16]

    li    s0, 10          # s0(a) = 10
    li    t1, 5           # t1(b) = 5
    li    s2, 7           # s2(c) = 7

    ############ Call Function Procedure ###########
    # Caller Saved

    # Pass Arguments
    mv    a0, s0          # a0(x) = s0(a)
    mv    a1, t1          # a1(y) = t1(b)
    mv    a2, s2          # a2(z) = s2(c)

    # Jump to Callee
    jal   ra, FUNC_XYZ    # ra = Addr(mv  s3, a0)
    ################################################

    mv    s3, a0          # s3(d) = a0(return value)

    ############ Call Function Procedure ###########
    # Caller Saved

    # Pass Arguments
    mv    a0, s0          # a0(x) = s0(a)
    mv    a1, s2          # a1(y) = s2(c)
    mv    a2, s3          # a2(z) = s3(d)

    # Jump to Callee
    jal   ra, FUNC_XYZ    # ra = Addr(add  t4, s0, s3)
    ################################################

    add   t4, s0, s3      # t4(f) = s0(a) + s3(d)
    add   t4, t4, a0      # t4(f) = t4 + a0(e)

    ## Retrieve ra & Retrieve Callee Saved
    lw    s0, 8(sp)       # s0 = @s0
    lw    s2, 4(sp)       # s2 = @s2
    lw    s3, 0(sp)       # s3 = @s3
    lw    ra, 12(sp)      # ra = @ra
    addi  sp, sp, 16      # Release stack space
                          # sp = @sp
    ## return
    ret                   # jalr  x0, ra, 0

FUNC_XYZ:
    ################################################
    # x : a0
    # y : a1
    # z : a2
    # return value : a0
    ################################################

    slli  t1, a1, 1       # t1 = y * 2
    slli  t2, a2, 1       # t2 = z * 2
    add   t2, t2, a2      # t2 = z * 3

    ## Pass Return Value
    add   a0, a0, t1      # a0 = x + y*2
    add   a0, a0, t2      # a0 = x + y*2 + z*3

    ## return
    ret                   # jalr  x0, ra, 0
```

Function Call – Question !

Q2 :
Please explain RISC-V calling convention in your word

Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

Control Transfer Instruction – (Conditional) Questions !

Conditional operation (B)

- beq / bne (Branch Equal / Not Equal)
  - beq : pc = (rs1 == rs2) ? pc + sext({imm12, 1'b0}) : pc + 4
  - bne : pc = (rs1 != rs2) ? pc + sext({imm12, 1'b0}) : pc + 4
- blt / bltu (Branch Less Than (Unsigned))
  - blt : pc = ($rs1_{signed}$ < $rs2_{signed}$) ? pc + sext({imm12, 1'b0}) : pc + 4
  - bltu : pc = ($rs1_{unsigned}$ < $rs2_{unsigned}$) ? pc + sext({imm12, 1'b0}) : pc + 4
- bge / bgeu (Branch Greater Equal Than (Unsigned))
  - bge : pc = ($rs1_{signed}$ >= $rs2_{signed}$) ? pc + sext({imm12, 1'b0}) : pc + 4
  - bgeu : pc = ($rs1_{unsigned}$ >= $rs2_{unsigned}$) ? pc + sext({imm12, 1'b0}) : pc + 4

Branch format

Bop rs1, rs2, imm12
Bop rs1, rs2, LABEL

Q3 : Why there is no bgt / ble in standard RV32I ?

- **(Recommend !!!) Always use LABEL**
- Branch are often used in **if…else**, **for**, **while**, **do…while**

Branch example – if … else

```c
int main() {
    int a = 0;
    int b = 10;

    if (b >= 10) {
        a = 5;
        b = 3;
    }
    else {
        a = 3;
        b = 3;
    }

    if (a == b) {
        a = 6;
        b = 11;
    }
    else {
        a = 4;
        b = 2;
    }

    if (b < 10) {
        a = 10;
        b = 5;
    }
    else {
        a = 9;
        b = 7;
    }

    return 0;
}
```

```asm
main:
    ############################################################
    # a : t0
    # b : t1
    # 10 : t2
    ############################################################

    ## Save ra & Callee Saved
    # No Function Call -> No need to save ra
    # No use saved registers -> No need to do Callee Saved

    li    t0, 0               # t0(a) = 0
    li    t1, 10              # t1(b) = 10
    li    t2, 10              # comparison immediate

    blt   t1, t2, main_else_1 # if (b < 10), go to main_else_1
main_if_1:
    li    t0, 5               # a = 5
    li    t1, 3               # b = 3
    jal   x0, main_endIf_1    # x0 = 0, pc = main_endIf_1
main_else_1:
    li    t0, 3               # a = 3
    li    t1, 3               # b = 3
main_endIf_1:

    beq   t0, t1, main_if_2   # if (a == b), go to main_if_2
main_else_2:
    li    t0, 4               # a = 4
    li    t1, 2               # b = 2
    jal   x0, main_endIf_2    # x0 = 0, pc = main_endIf_2
main_if_2:
    li    t0, 6               # a = 6
    li    t1, 11              # b = 11
main_endIf_2:

    bge   t1, t2, main_else_3 # if (b >= c), go to main_else_3
main_if_3:
    li    t0, 10              # a = 10
    li    t1, 5               # b = 5
    jal   x0, main_endIf_3    # x0 = 0, go to main_endIf_3
main_else_3:
    li    t0, 9               # a = 9
    li    t1, 7               # b = 7
main_endIf_3:

    ret                       # jalr  x0, ra, 0
```


Branch example – for, while

```c
int main () {
    int result = 0;
    for (int i = 0 ; i < 10 ; i++) {
        result += i;
    }
    return 0;
}
```

```c
int main () {
    int result = 0;
    int i = 0;
    while (i < 10) {
        result += i;
        i++;
    }
    return 0;
}
```

```asm
main:
    ############################################################
    # t0 : result
    # t1 : i
    # t2 : loop boundary (max)
    ############################################################

    ## Save ra & Callee Saved
    # No Function Call -> No need to save ra
    # No use saved registers -> No need to do Callee Saved

    li    t0, 0                 # t0(result) = 0
    li    t1, 0                 # t1(i) = 10
    li    t2, 10                # max = 10

main_loop_1:
    bge   t1, t2, main_endLoop_1 # if (i >= max), go to main_endLoop_1
    add   t0, t0, t1            # result += i
    addi  t1, t1, 1             # i++
    jal   x0, main_loop_1       # x0 = 0, pc = main_loop_1
main_endLoop_1:

    ret                         # jalr  x0, ra, 0
```

```asm
main:
    ############################################################
    # t0 : result
    # t1 : i
    # t2 : loop boundary (max)
    ############################################################

    ## Save ra & Callee Saved
    # No Function Call -> No need to save ra
    # No use saved registers -> No need to do Callee Saved

    li    t0, 0                 # t0(result) = 0
    li    t1, 0                 # t1(i) = 10
    li    t2, 10                # max = 10

    bge   t1, t2, main_endLoop_1 # if (i >= max), go to main_endLoop_1
main_loop_1:
    add   t0, t0, t1            # result += i
    addi  t1, t1, 1             # i++
    blt   t1, t2, main_loop_1   # if (i < max), go to main_loop_1
main_endLoop_1:

    ret                         # jalr  x0, ra, 0
```

Branch example – do … while

```c
int main () {
    int result = 0;
    int i = 0;
    do {
        result += i;
        i++;
    } while (i < 10);
    return 0;
}
```

```asm
main:
    ############################################################
    # t0 : result
    # t1 : i
    # t2 : loop boundary (max)
    ############################################################

    ## Save ra & Callee Saved
    # No Function Call -> No need to save ra
    # No use saved registers -> No need to do Callee Saved

    li    t0, 0                 # t0(result) = 0
    li    t1, 0                 # t1(i) = 10
    li    t2, 10                # max = 10

main_loop_1:
    add   t0, t0, t1            # result += i
    addi  t1, t1, 1             # i++
    blt   t1, t2, main_loop_1   # if (i < max), go to main_loop_1
main_endLoop_1:

    ret                         # jalr  x0, ra, 0
```


Pseudo Instructions

Pseudo Instruction	Meaning	Base Instruction
la rd, LABEL	rd = Addr(LABEL)	auipc / addi
l{w|h|b} rd, LABEL	rd = sext(*LABEL[{31|15|7}:0])	auipc / l{w|h|b}
bgt(u) rs, rt, {offset|LABEL}	rd = (rs < rs) ? pc + offset : pc + 4	blt rt, rs, offset
ble(u) rs, rt, {offset|LABEL}	rd = (rt >= rs) ? 1 : 0	bge rt, rs, offset
b{eq | ne | lt | ge| gt | le}z rs, {offset|LABEL}	rd = (rs { == |!= | < | >= | > | <= } 0) ? pc + offset : pc + 4	b{eq | ne | lt | ge | gt | le} rs, x0, offset
j {offset|LABEL}	pc += offset, no save ra	jal x0, {offset|LABEL}
jr rs	pc = rs, no save ra	jalr x0, rs, 0
ret	pc = ra, no save ra	jalr x0, ra, 0
jal {offset|LABEL}	pc += offset, ra = pc + 4	jal ra, {offset|LABEL}
jalr rs	pc = rs, ra = pc + 4	jalr ra, rs, 0
call {offset|LABEL}	pc += big_offset, ra = pc + 4 (big_offset > 21 bits)	auipc ra, offset[31:12] jalr ra, ra, {offset[11:0]|LABEL}

Questions !

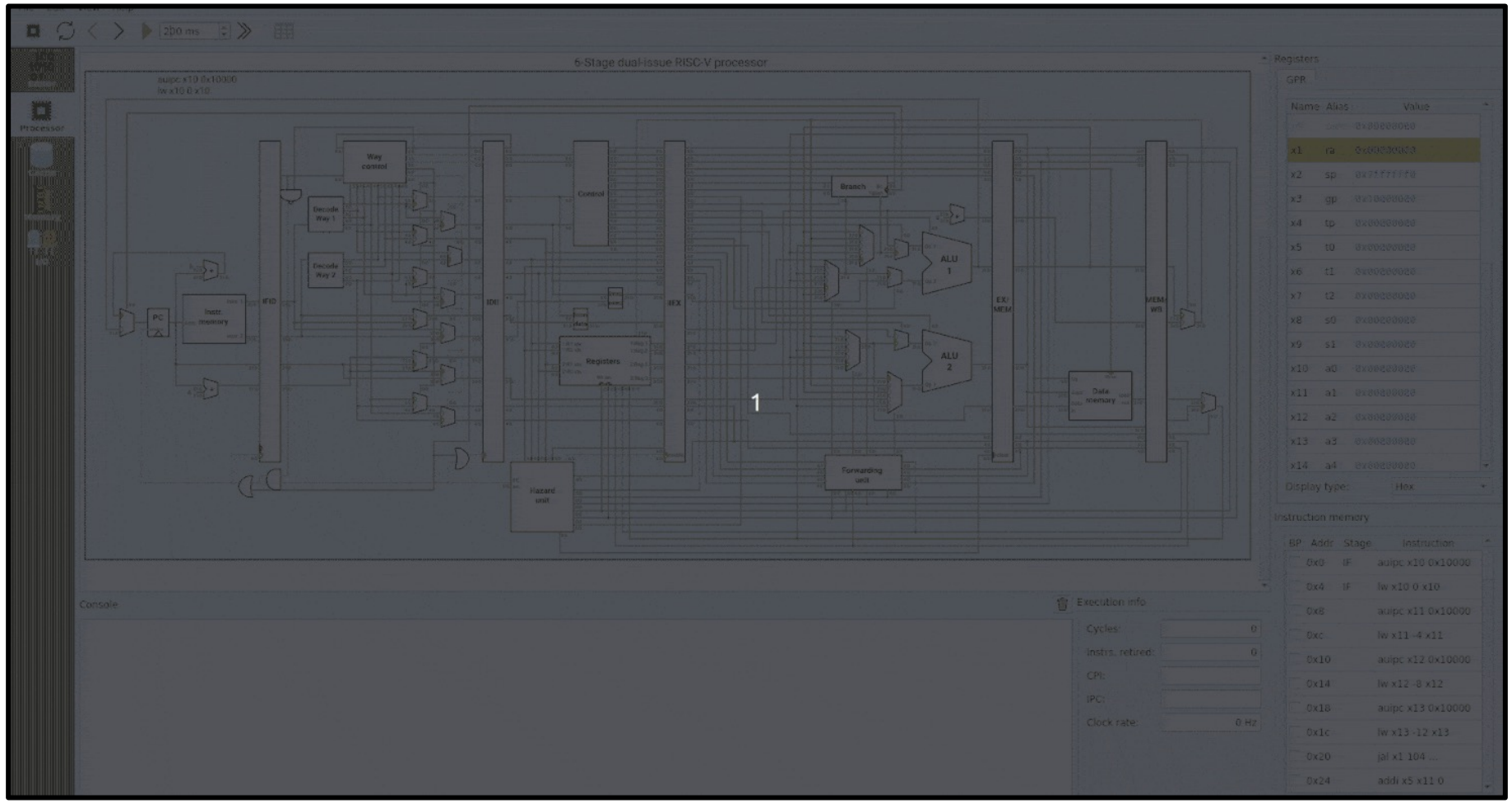
Q4 :
We can find that auipc, jal, branch are all PC-relative instructions.
What's the advantage of PC-relative ?

- auipc (Add Upper Immediate to PC)
  - rd = pc + {uimm20, 12'b0}
- jal (Jump And Link, **J type**)
  - rd = pc + 4
  - pc = pc + sext({imm20, 1'b0})
    (or pc = LABEL)
- branch
  - pc = (condition) ? pc + sext({imm20, 1'b0}) : pc + 4

RIPES Introduction

RIPES Installation

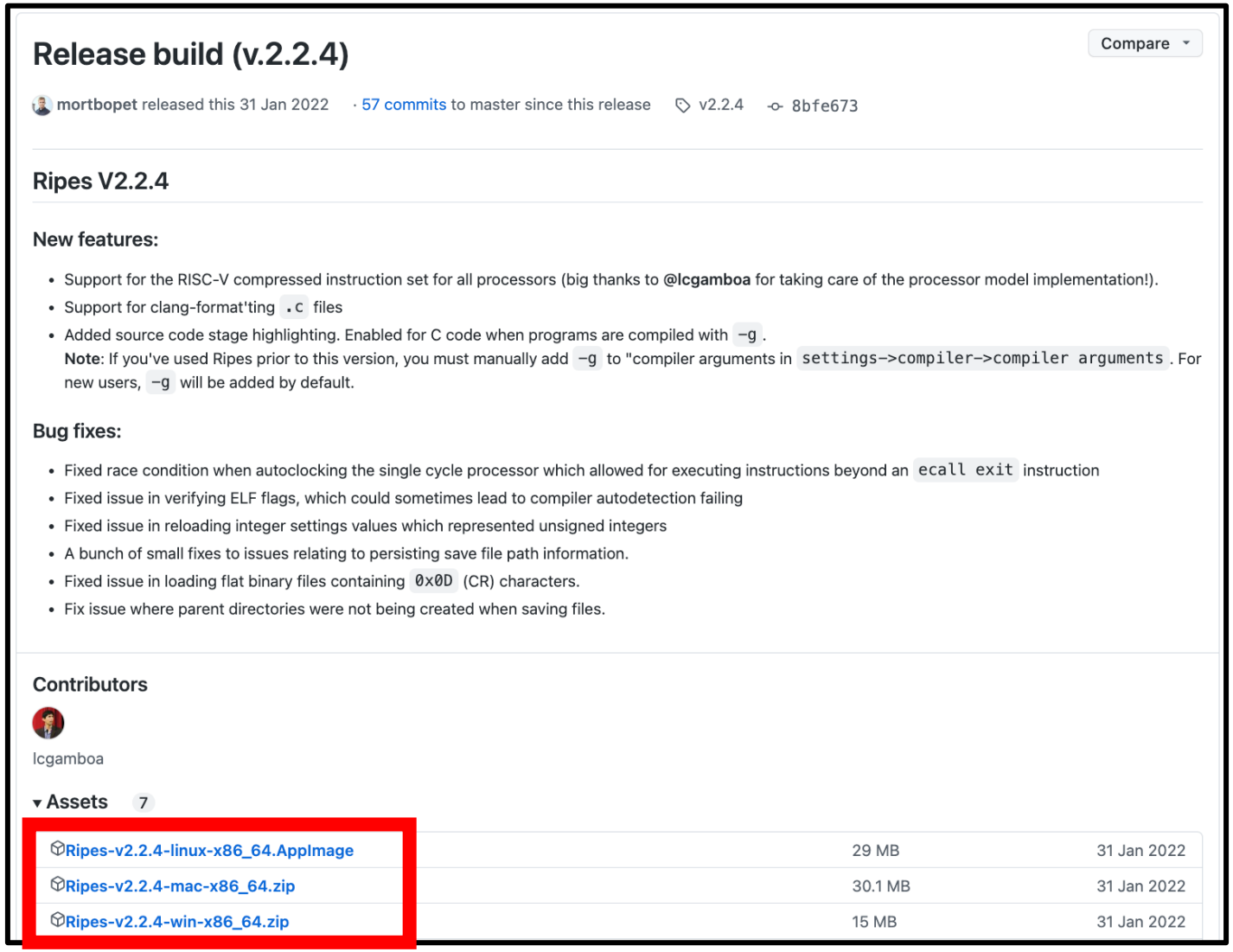
- Ripes is a visual computer architecture simulator and assembly code editor built for the RISC-V instruction set architecture.



RIPES Installation

Link : https://github.com/mortbopet/Ripes/releases/tag/v2.2.4

- According your OS to choose corresponding download file



Release build (v.2.2.4)

Compare

mortbopet released this 31 Jan 2022 · 57 commits to master since this release · v2.2.4 · 8bfe673

Ripes V2.2.4

New features:

- Support for the RISC-V compressed instruction set for all processors (big thanks to **@lcgamboa** for taking care of the processor model implementation!).
- Support for clang-format'ting `.c` files
- Added source code stage highlighting. Enabled for C code when programs are compiled with `-g`.
  Note: If you've used Ripes prior to this version, you must manually add `-g` to "compiler arguments in `settings->compiler->compiler arguments`. For new users, `-g` will be added by default.

Bug fixes:

- Fixed race condition when autoclocking the single cycle processor which allowed for executing instructions beyond an `ecall exit` instruction
- Fixed issue in verifying ELF flags, which could sometimes lead to compiler autodetection failing
- Fixed issue in reloading integer settings values which represented unsigned integers
- A bunch of small fixes to issues relating to persisting save file path information.
- Fixed issue in loading flat binary files containing `0x0D` (CR) characters.
- Fix issue where parent directories were not being created when saving files.

Contributors

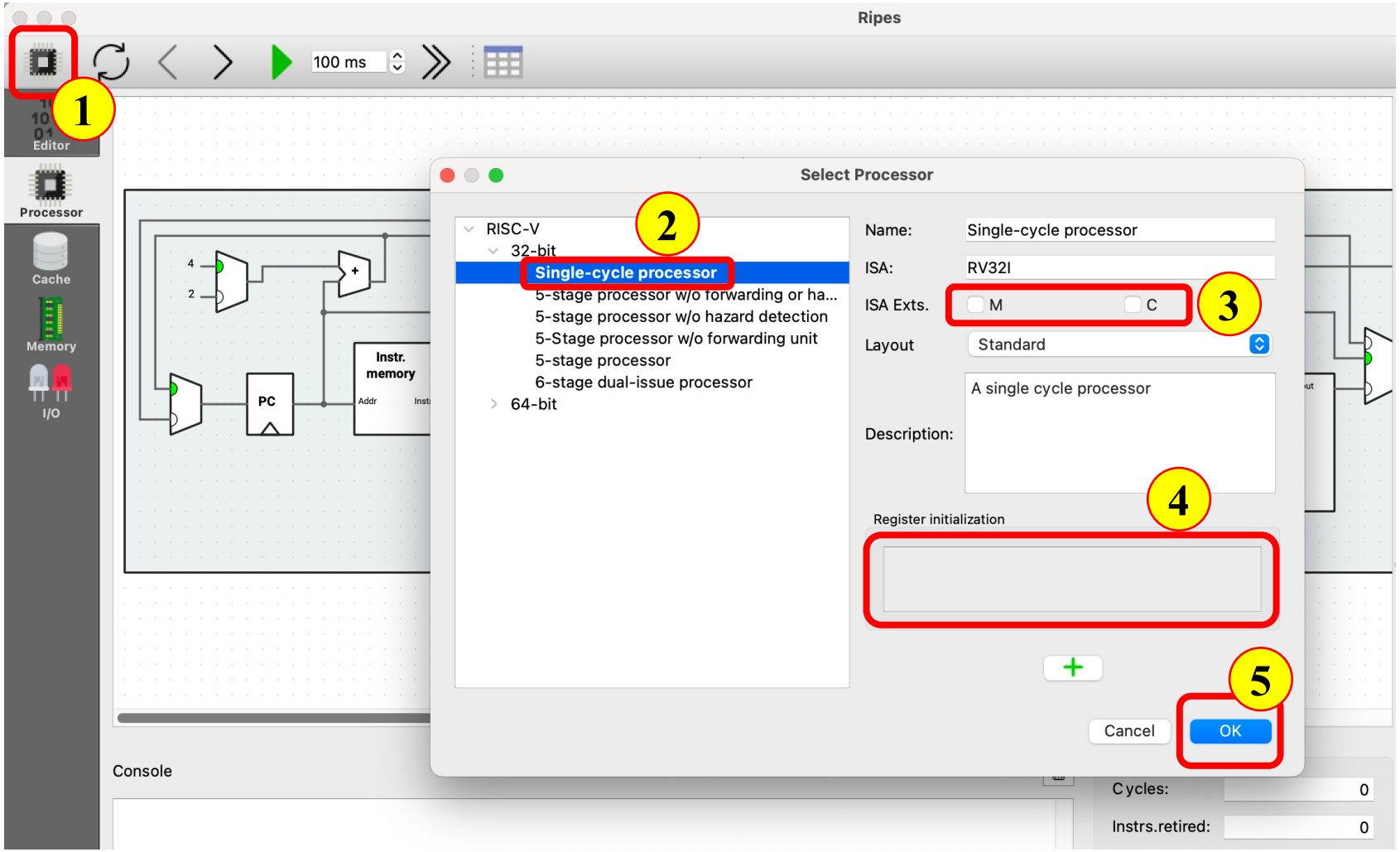
lcgamboa

▾ Assets 7

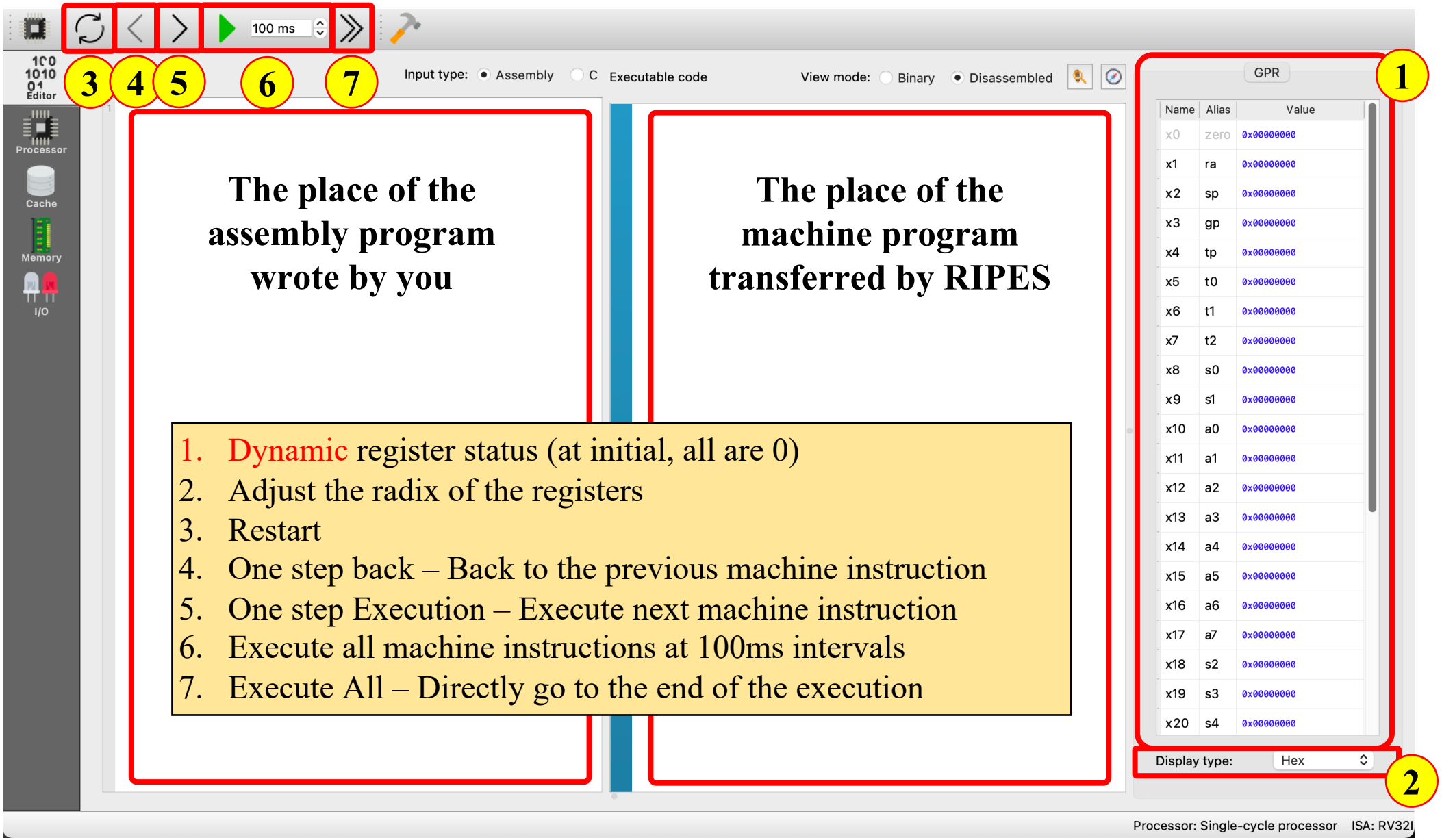
Ripes-v2.2.4-linux-x86_64.AppImage	29 MB	31 Jan 2022
Ripes-v2.2.4-mac-x86_64.zip	30.1 MB	31 Jan 2022
Ripes-v2.2.4-win-x86_64.zip	15 MB	31 Jan 2022

RIPES Setting

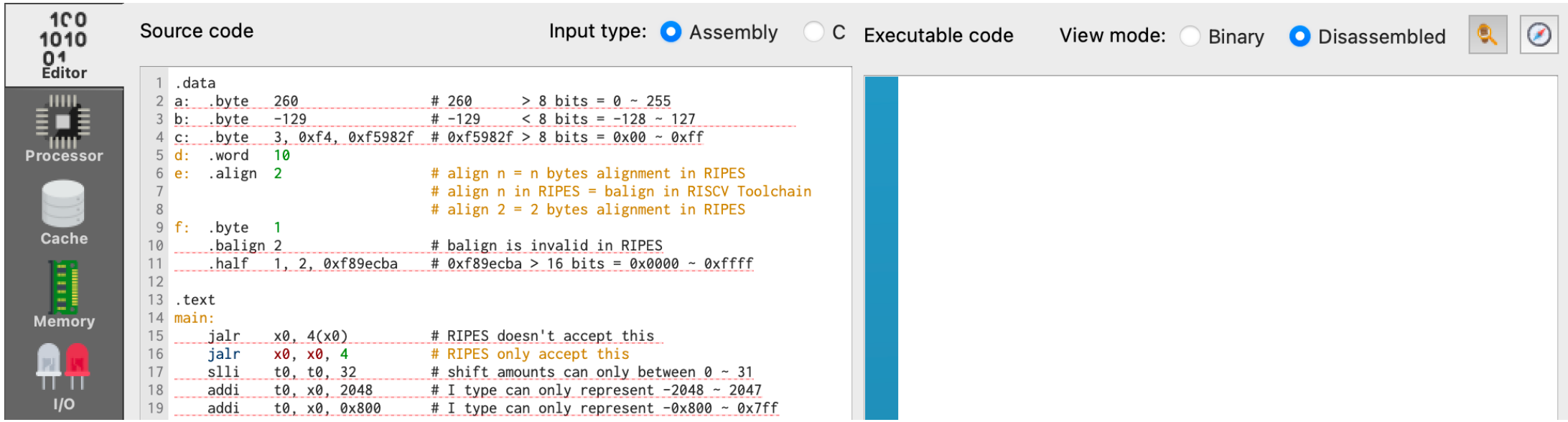
1. Click the IC icon
2. Select Single-cycle processor
3. Deselect M / C Extension
4. Delete all initialization
   (At initial, all are 0)
5. Click “OK”

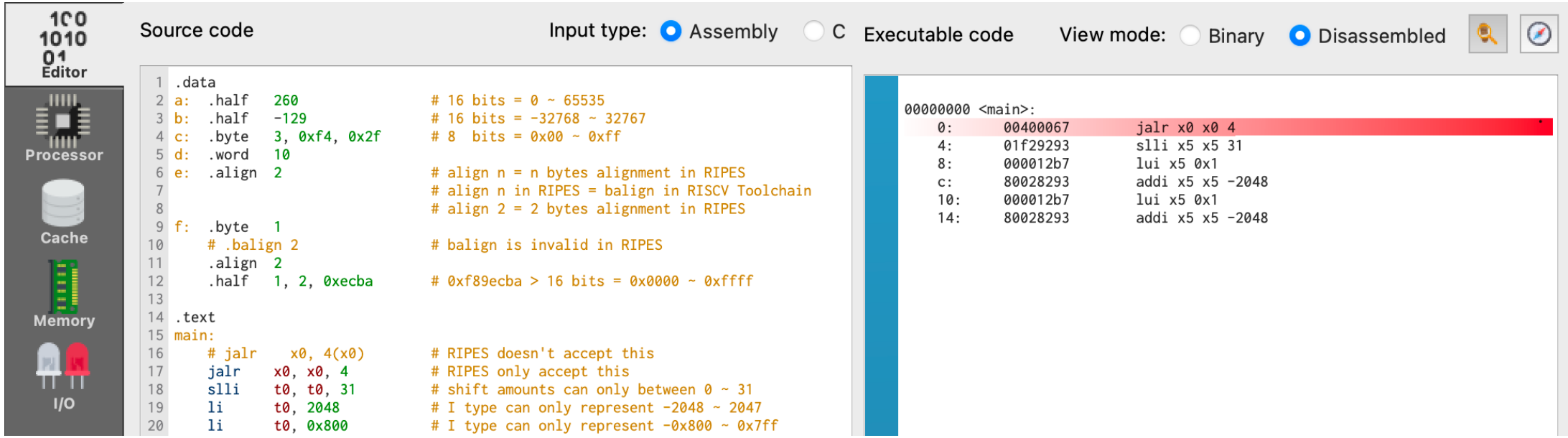


RIPES Editor

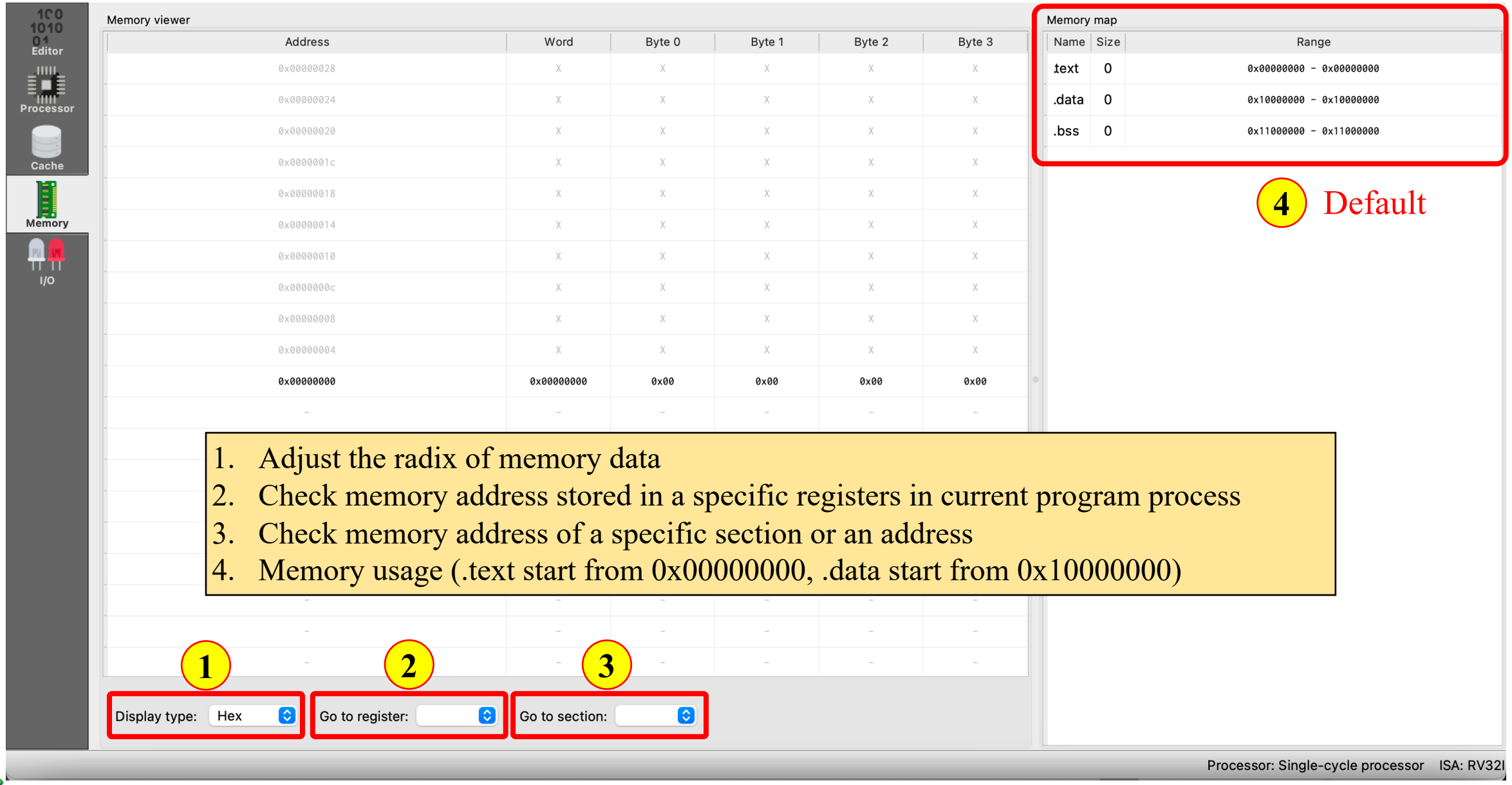


RIPES Editor

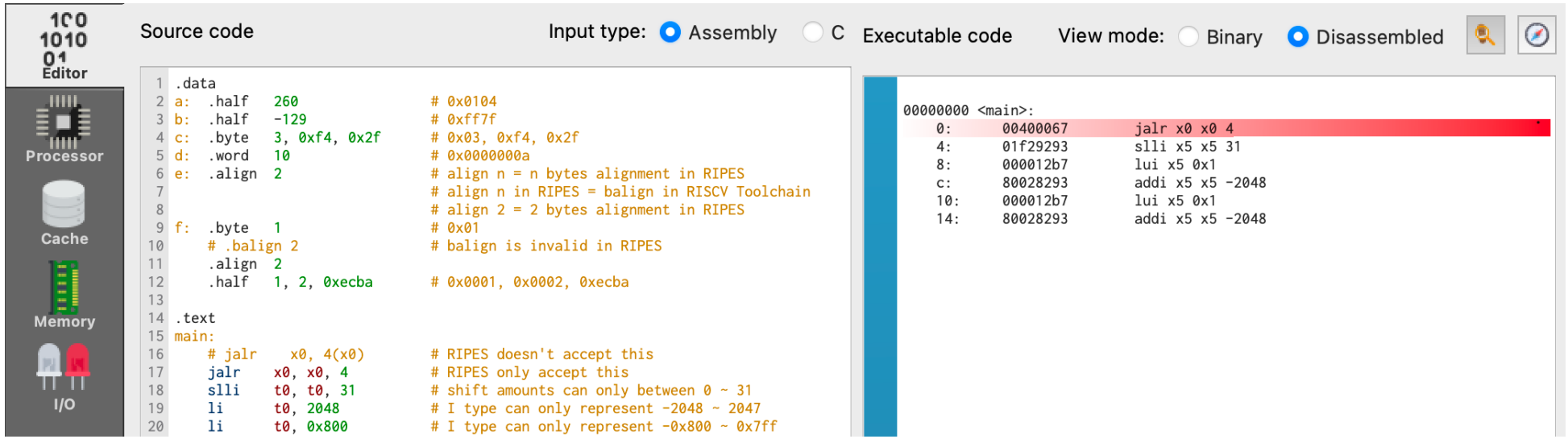


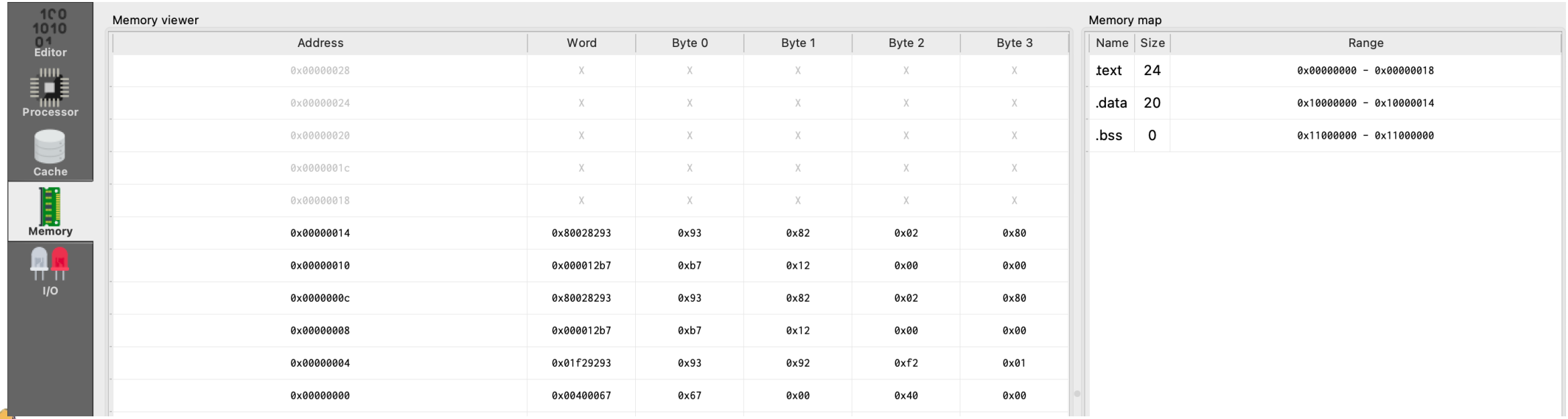


RIPES Memory

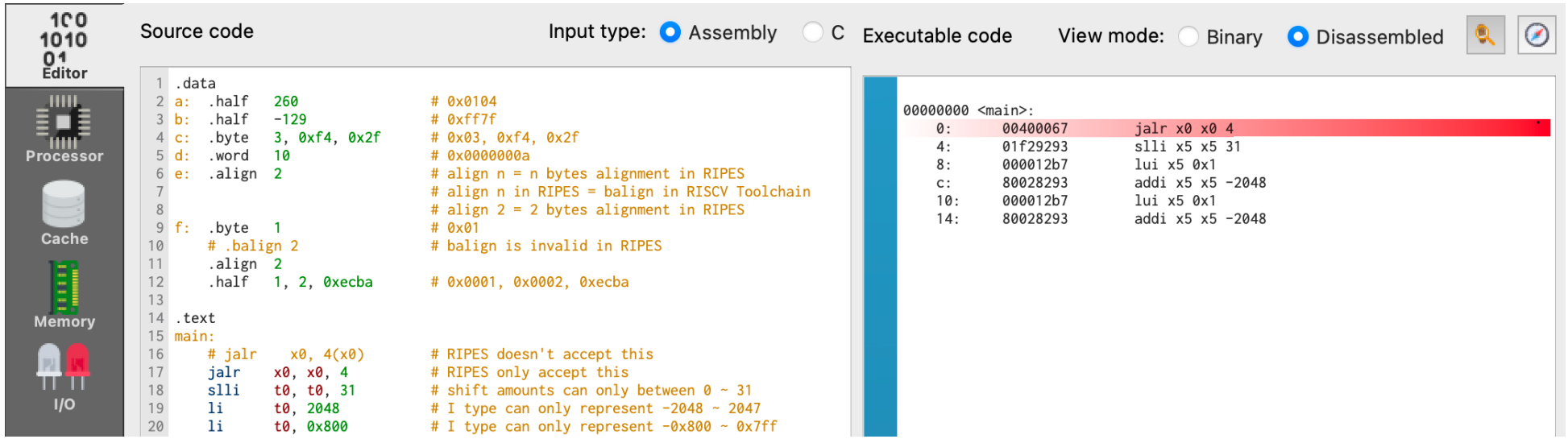


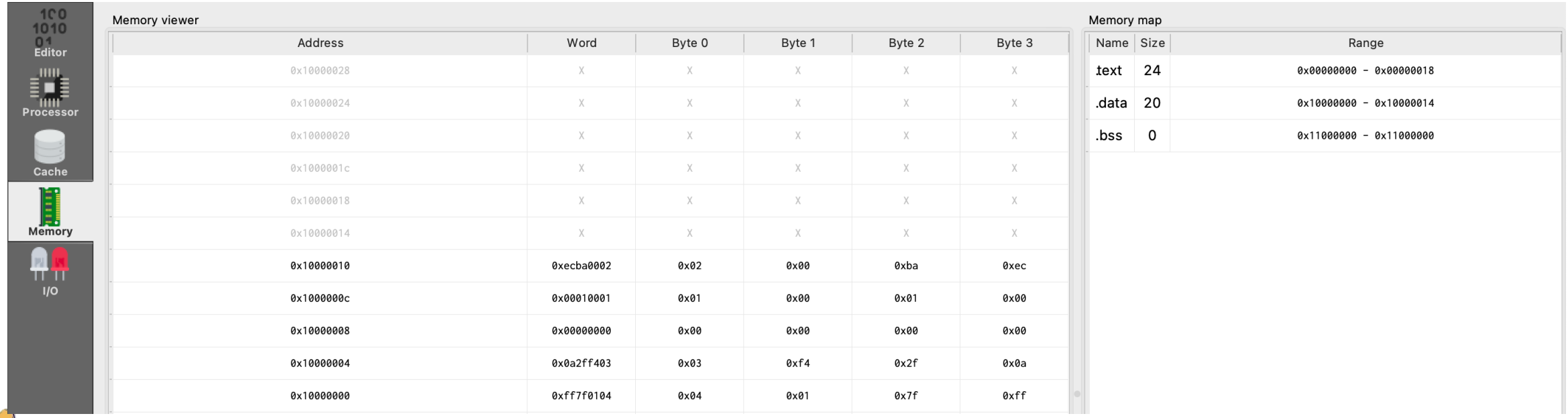
RIPES Memory – .text section



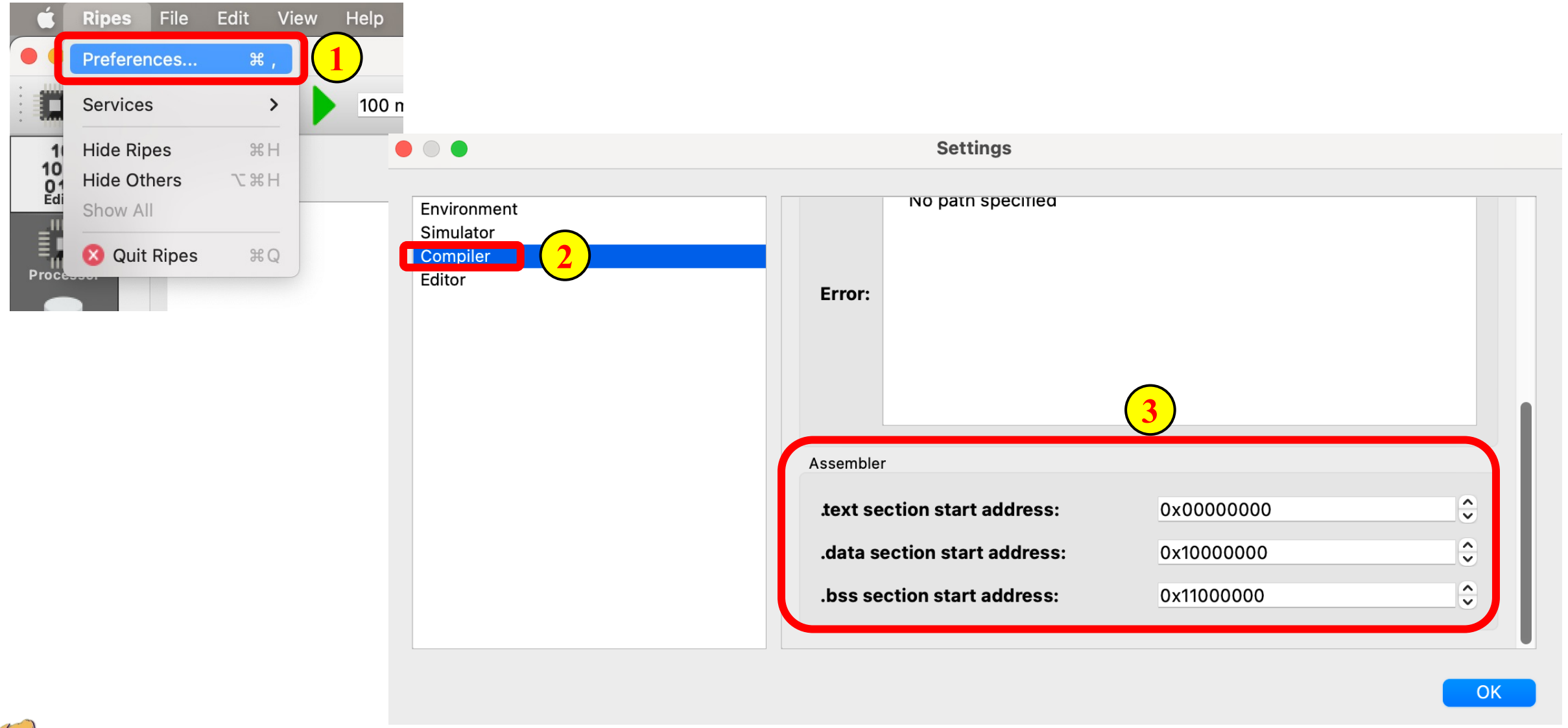


RIPES Memory – .data section





RIPES Memory Address Setting

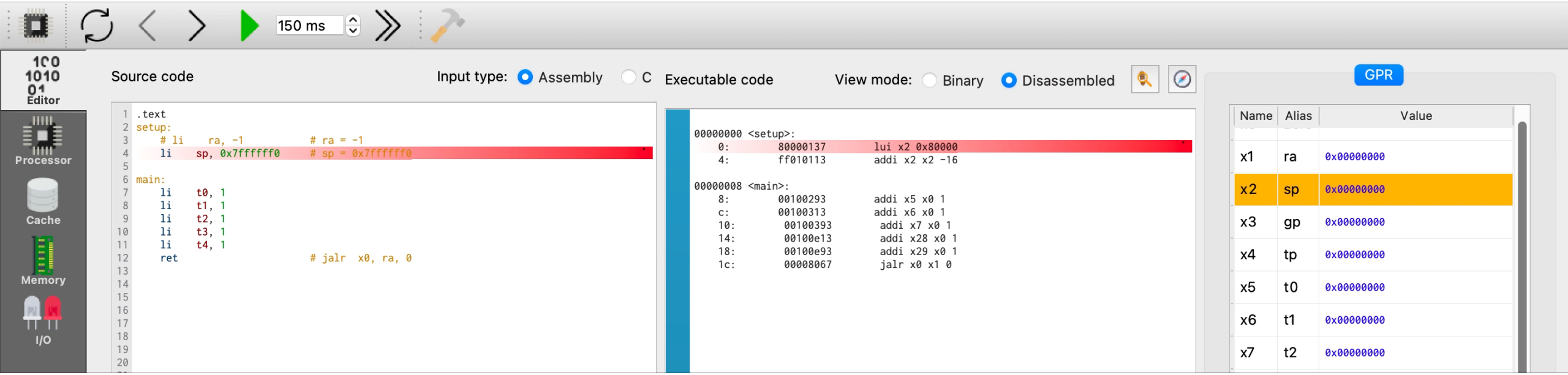


Setup Code

```
1  .text
2  setup:
3      li     ra, -1              # ra = -1
4      li     sp, 0x7ffffff0      # sp = 0x7ffffff0
5
6  main:
7      # ...
8
9      ret                        # jalr  x0, ra, 0
10
```

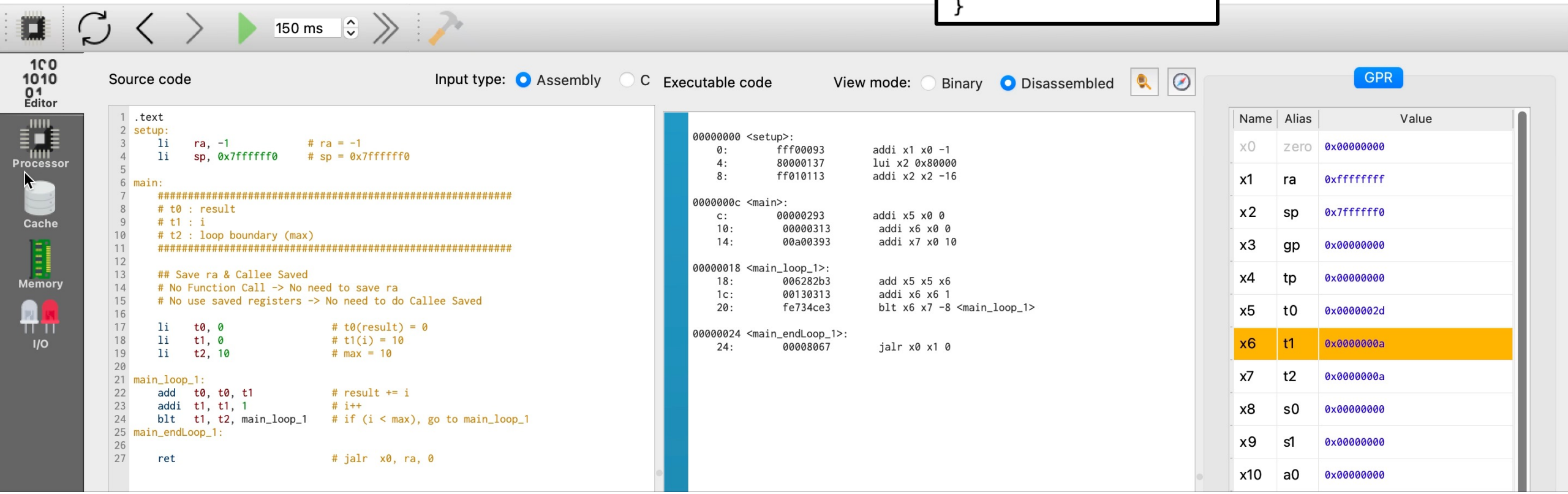
We need to add this code before main code

If not, it will drop into infinite loop



Example : do…while Loop

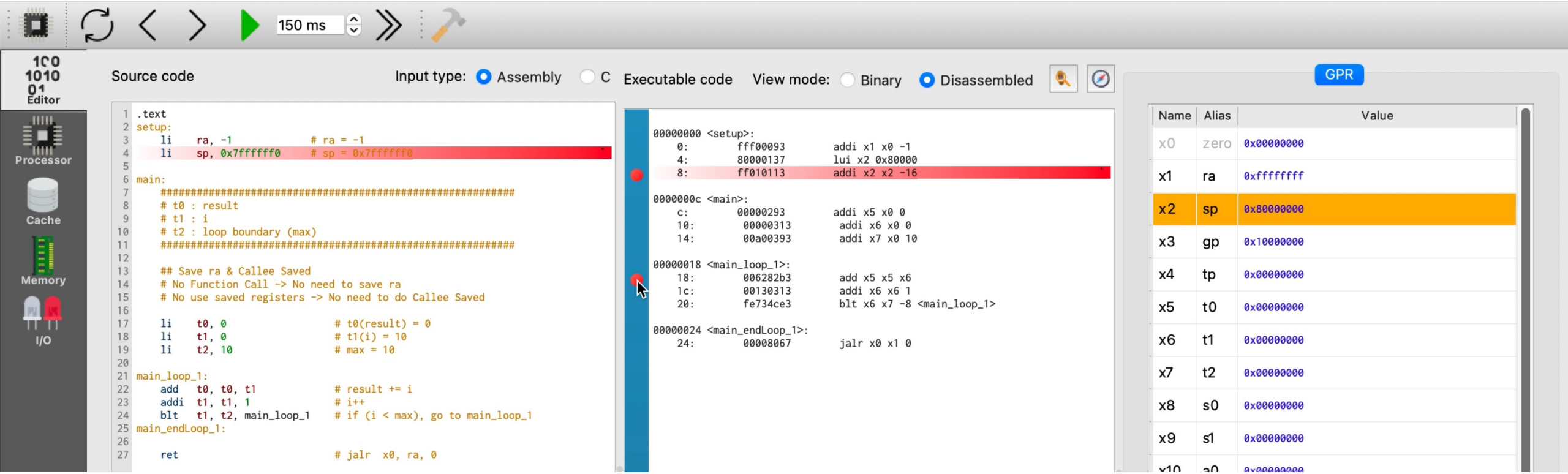
```c
int main () {
    int result = 0;
    int i = 0;
    do {
        result += i;
        i++;
    } while (i < 10);
    return 0;
}
```



Tips : Breakpoint (中斷點)

You can set some breakpoints to help debug

Execution will stop just before the breakpoint



Applications

Applications : Multiplication / Division

- **Multiplication**
  - $x5 = x5 \times 2$

```
slli  x5, x5, 1
```

  - $x5 = x5 \times 3$

```
slli  x6, x5, 1
add  x5, x6, x5
```

Can only multiply with an immediate

  - We can implement variable multiplication by successive addition
  - A * B = A + A + … + A (for/while loop B times addition)

- **Division**
  - $x5 = x5 / 8$

```
srai  x5, x5, 3
```

→ **Can only divide with an immediate that power of 2**

  - We can implement variable division by successive subtraction
  - A / B = A - B - B - B …
    (while loop stop when A < B, the quotient is the times of subtraction)

Applications : Variable Multiplication

```c
int mul(int multiplicand, int multiplier) {

/*  Do abs multiplication */

    // Get abs(multiplicand) & abs(multiplier)
    int a = abs(multiplicand)
    int b = abs(mul tiplier)

    // Do sorting to accelerate multiplication
    int array[2] = {a, b};
    two_sort(&array);

    // Do multiplication through successive addition
    int i = 0;
    int result = 0;
    while (i < array[0]) {
        result += array[1];
        i++;
    }

/* Append sign on the abs_result */

    // Calculate the sign of the result
    int mask_a = multiplicand >>> 31;
    int mask_b = multiplier >>> 31;
    int mask_result = mask_a ^ mask_b;

    // Accroding mask_result
    // 1. keep result positive
    // 2. Convert result to negative
    result = result ^ mask_result;
    result = result - mask_result;

    return result;
}
```

```c
void two_sort (int *array) {
    int a = *(array);
    int b = *(array+1);

    if (a > b) {
        *(array) = b;
        *(array+1) = a;
    }
}
```

t0 = +3 = 0000_0000_..._0003

```
srai    t1, t0, 31   #  t1 = 0000_0000_..._0000 (0, mask)
xor     t0, t0, t1   #  t0 = 0000_0000_..._0003 (+3)
sub     t0, t0, t1   #  t0 = 0000_0000_..._0003 (+3)
```

t0 = -3 = 1111_1111_..._1101

```
srai    t1, t0, 31   #  t1 = 1111_1111_..._1111 (-1, mask)
xor     t0, t0, t1   #  t0 = 0000_0000_..._0010 (+2)
sub     t0, t0, t1   #  t0 = 0000_0000_..._0011 (+3)
```

t1 = mask_result = mask_a ^ mask_b

- ++ : 0000_0000_..._0000
- - - : 0000_0000_..._0000
- + - : 1111_1111_..._1111
- - + : 1111_1111_..._1111

t0 = abs(result)

```
xor     t0, t0, t1   # + → keep,  − → complement
sub     t0, t0, t1   # + → - 0 ,  − → - (-1)
```

Applications : Variable Multiplication

```
23 FUNC_MUL:
24     ########################################################
25     # < Function >
26     #    It can handle multiplication of 2 integer variables
27     #
28     # < Parameters >
29     #    a0 : multiplicand
30     #    a1 : multiplier
31     #
32     # < Return Value >
33     #    a0 : result
34     ########################################################
35     # < Local Variable >
36     #    t0 : mask_a
37     #    t1 : mask_b
38     #    t2 : abs(multiplicand) / smaller one
39     #    t3 : abs(multiplier) / bigger one
40     #    t4 : mask_result
41     #    t5 : i
42     #    t6 : result
43     ########################################################
44
45     ## Save ra & Callee Saved
46     # No use saved registers -> No need to do Callee Saved
47     addi    sp, sp, -4              # Allocate stack space
48                                     # sp = @sp - 4
49     sw      ra, 0(sp)               # @ra -> MEM[@sp - 4]
50
51     ## abs(multiplicand)
52     srai    t0, a0, 31              # t0 = 0xffffffff(-) / 0(+)
53     xor     t2, a0, t0              # Inverse(-) / Keep(+)
54     sub     t2, t2, t0              # -(-1) / -0
55                                     # t2 = abs(multiplicand)
56
57     ## abs(multiplier)
58     srai    t1, a1, 31              # t1 = 0xffffffff(-) / 0(+)
59     xor     t3, a1, t1              # Inverse(-) / Keep(+)
60     sub     t3, t3, t1              # t3 = abs(multiplier)
61
62     ## mask_result
63     xor     t4, t0, t1              # t4 = 0xffffffff(-) / 0(+)
64
65     ## array[2] = {a, b}
66     addi    sp, sp, -8              # Allocate stack space
67                                     # sp = @sp - 12
68     sw      t3, 4(sp)               # t3 -> MEM[@sp - 8]
69     sw      t2, 0(sp)               # t2 -> MEM[@sp - 12]
70
71     ############### Call Function Procedure ###############
72     # Caller Saved
73     addi    sp, sp, -4              # Allocate stack space
74                                     # sp = @sp - 16
75     sw      t4, 0(sp)               # t4 -> MEM[@sp - 16]
76
77     # Pass Arguments
78     addi    a0, sp, 4               # a0 = &array
79
80     # Jump to Callee
81     jal     ra, FUNC_TWO_SORT       # ra = Addr(lw  t4, 0(sp))
82     ########################################################
83
84     # Retrieve Caller Saved
85     lw      t4, 0(sp)               # t4(mask_result)
86     lw      t2, 4(sp)               # t2 = smaller one
87     lw      t3, 8(sp)               # t3 = bigger one
88
89     ## Do multiplication through successive addition
90     li      t5, 0                   # t5(i) = 0
91     li      t6, 0                   # t6(result) = 0
92     bge     t5, t2, FMUL_endWhile_1 # if (i >= smaller), go to endWhile
93 FMUL_while_1:
94     add     t6, t6, t3              # result += bigger
95     addi    t5, t5, 1               # i++
96     blt     t5, t2, FMUL_while_1    # if (i < smaller), go to while
97 FMUL_endWhile_1:
98
99     ## Now, t6 = abs(result)
100    ## Append sign on the t6
101    xor     t6, t6, t4              # Inverse(-) / Keep(+)
102    sub     t6, t6, t4              # -(-1) / -0
103                                    # t6 = result
104
105    ## Pass return value
106    mv      a0, t6                  # a0(return value) = t6
107
108    ## Retrieve ra & Callee Saved
109    lw      ra, 12(sp)              # ra = @ra
110    addi    sp, sp, 16              # sp = @sp
111
112    ## return
113    ret                             # jalr  x0, ra, 0
114
115
116 FUNC_TWO_SORT:
117    ########################################################
118    # < Function >
119    #    Do sorting
120    #    * Put the smaller one in *array
121    #    * Put the bigger one in *(array+1)
122    #
123    # < Parameters >
124    #    a0 : int *array
125    #
126    # < Return Value >
127    #    NULL
128    ########################################################
129    # < Local Variable >
130    #    t0 : a
131    #    t1 : b
132    ########################################################
133
134    ## Save ra & Callee Saved
135    # No Function Call -> No need to save ra
136    # No use saved registers -> No need to do Callee Saved
137
138    lw      t0, 0(a0)               # a = *array
139    lw      t1, 4(a0)               # b = *(array+1)
140
141    bleu    t0, t1, FUNC_TWO_EXIT   # if a <= b, no need to swap
142
143    sw      t1, 0(a0)               # smaller -> *array
144    sw      t0, 4(a0)               # bigger -> *(array+1)
145 FUNC_TWO_EXIT:
146    ret                             # return
```

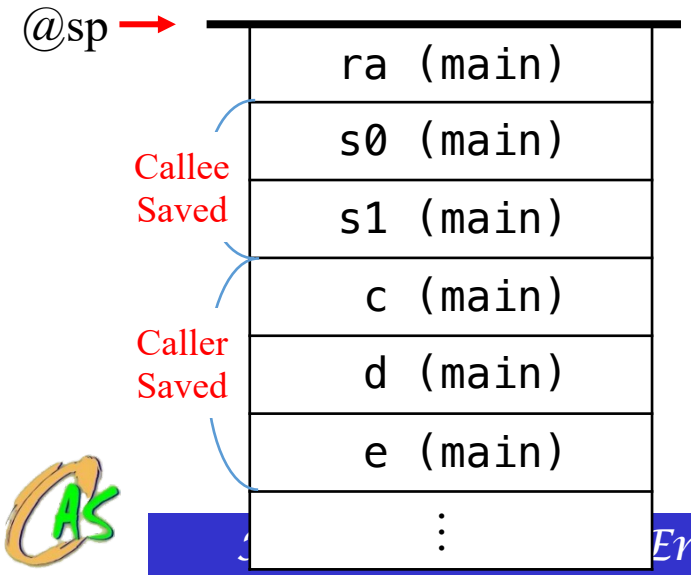

Applications : Variable Division

Do yourself !

```
222 FUNC_DIV:
223     ############################################################
224     # < Function >
225     #    It can handle division of 2 integer variables
226     #
227     # < Parameters >
228     #    a0 : dividend
229     #    a1 : divisor
230     #
231     # < Return Value >
232     #    a0 : result
233     ############################################################
234     # < Local Variable >
235     #    Define by yourself
236     ############################################################
237
238     ## Do yourself
239     ## ...
240
241     ret                                 # jalr  x0, ra, 0
```

Applications : Function Call

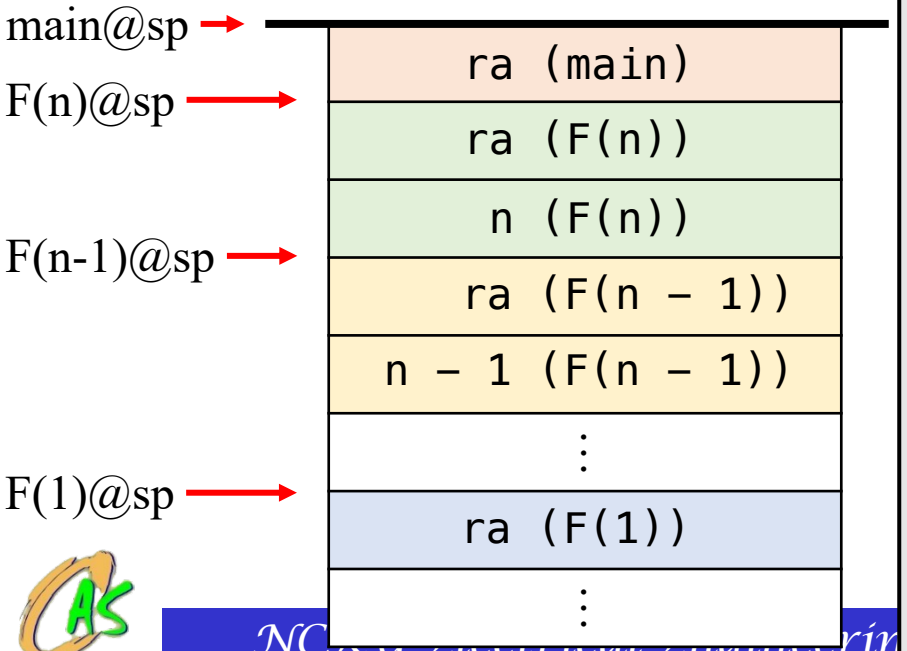
```c
int mul(int multiplicand, int multiplier) {
    // ...
}

int div(int dividend, int divisor) {
    // ...
}

int main() {
    int a = from memory
    int b = from memory
    int c = a + b;
    int d = a - b;
    int e = mul(c, d);
    int f = div(c, d);
    int result = a + b + c + d + e + f;
    return 0;
}
```




```assembly
.data
n1: .word 3
n2: .word 7

.text
setup:
    li      ra, -1
    li      sp, 0x7ffffff0

main:
    ###########################################################
    # < Function >
    #    main procedure
    #
    # < Parameters >
    #    NULL
    #
    # < Return Value >
    #    NULL
    ###########################################################
    # < Local Variable >
    #    s0 : a
    #    s1 : b
    #    t0 : c
    #    t1 : d
    #    t2 : e
    #    t3 : result
    ###########################################################

    ## Save ra & Callee Saved
    addi    sp, sp, -12             # Allocate stack space
                                    # sp = @sp - 12
    sw      ra, 8(sp)               # @ra -> MEM[@sp - 4]
    sw      s0, 4(sp)               # @s0 -> MEM[@sp - 8]
    sw      s1, 0(sp)               # @s1 -> MEM[@sp - 12]

    ## Get a, b
    la      s0, n1                  # s0 = Addr(a)
    lw      s0, 0(s0)               # s0(a) = MEM[s0]
    lw      s1, n2                  # s1(b) = MEM[Addr(b)]

    add     t0, s0, s1              # t0(c) = s0(a) + s1(b)
    sub     t1, s0, s1              # t1(d) = s0(a) - s1(b)

    ############### Call Function Procedure ###############
    # Caller Saved
    addi    sp, sp, -8              # Allocate stack space
                                    # sp = @sp - 20
    sw      t0, 4(sp)               # t0(c) -> MEM[@sp - 16]
    sw      t1, 0(sp)               # t1(d) -> MEM[@sp - 20]

    # Pass Arguments
    mv      a0, t0                  # a0(multiplicand) = t0(c)
    mv      a1, t1                  # a1(multiplier) = t1(d)

    # Jump to Callee
    jal     ra, FUNC_MUL            # ra = Addr(addi  sp, sp, -4)
    ###########################################################


    ############### Call Function Procedure ###############
    # Caller Saved
    addi    sp, sp, -4              # Allocate stack space
                                    # sp = @sp - 24
    sw      a0, 0(sp)               # a0(e) -> MEM[@sp - 24]

    # Pass Arguments
    lw      a0, 8(sp)               # a0(dividend) = c
    lw      a1, 4(sp)               # a1(divisor) = d

    # Jump to Callee
    jal     ra, FUNC_DIV            # ra = Addr(lw  t0, 8(sp))
    ###########################################################

    ## Retrieve Caller Saved
    lw      t0, 8(sp)               # t0 = c
    lw      t1, 4(sp)               # t1 = d
    lw      t2, 0(sp)               # t2 = e

    add     t3, s0, s1              # t3 = s0(a) + s1(b)
    add     t4, t0, t1              # t4 = t0(c) + t1(d)
    add     t3, t3, t4              # t3 = a + b + c + d
    add     t4, t2, a0              # t4 = t2(e) + a0(f)
    add     t3, t3, t4              # t3 = result

    ## Retrieve ra & Retrieve Callee Saved
    lw      s1, 12(sp)               # s1 = @s1
    lw      s0, 16(sp)               # s0 = @s0
    lw      ra, 20(sp)               # ra = @ra

    addi    sp, sp, 24               # Release stack space
                                     # sp = @sp
    ## retrun
    ret                              # jalr  x0, ra, 0
```

Applications : Recursion

```c
int termial(int n) {
    if (n == 1) return 1;
    return n + termial(n-1);
}

int main() {
    int n = 10;
    int result = termial(n);
    return 0;
}
```




```asm
.data
n:  .word 10

.text
setup:
    li      ra, -1
    li      sp, 0x7ffffff0

main:
    ########################################################
    # < Function >
    #    main procedure
    #
    # < Parameters >
    #    NULL
    #
    # < Return Value >
    #    NULL
    ########################################################
    # < Local Variable >
    #    t0 : n
    #    t1 : result
    ########################################################

    ## Save ra & Callee Saved
    # No use saved registers -> No need to do Callee Saved
    addi    sp, sp, -4              # Allocate stack space
                                    # sp = @sp - 4
    sw      ra, 0(sp)               # @ra -> MEM[@sp - 4]

    ## Get n
    la      t0, n                   # t0 = Addr(n)
    lw      t0, 0(t0)               # t0(n) = MEM[t0]

    ############### Call Function Procedure ###############
    # Caller Saved

    # Pass Arguments
    mv      a0, t0                  # a0(n) = t0(n)

    # Jump to Callee
    jal     ra, FUNC_TERMIAL        # ra = Addr(mv  t1, a0)
    ########################################################

    ## Get return value
    mv      t1, a0                  # t1 = termial(n)

    ## Retrieve ra & Retrieve Callee Saved
    lw      ra, 0(sp)               # ra = @ra
    addi    sp, sp, 4               # Release stack space
                                    # sp = @sp
    ## retrun
    ret                             # jalr  x0, ra, 0
```

```asm
FUNC_TERMIAL:
    ########################################################
    # < Function >
    #    n + (n-1) + (n-2) + ... + 1
    #
    # < Parameters >
    #    a0 : n
    #
    # < Return Value >
    #    a0 : n + termial(n-1)
    ########################################################
    # < Local Variable >
    #    t0 : n
    #    t1 : termial(n-1)
    ########################################################

    ## Save ra & Callee Saved
    # No use saved registers -> No need to do Callee Saved
    addi    sp, sp, -4              # Allocate stack space
                                    # sp = @sp - 4
    sw      ra, 0(sp)               # @ra -> MEM[@sp - 4]

    mv      t0, a0                  # t0 = n
    addi    t1, a0, -1              # t1 = n - 1
    beq     t1, x0, endAdd          # if t1 == 0 (n == 1) => return 1
                                    # else => recursion
termialRecur:
    ############### Call Function Procedure ###############
    # Caller Saved
    addi    sp, sp, -4              # Allocate stack space
                                    # sp = @sp - 8
    sw      t0, 0(sp)               # t0(n) -> MEM[@sp - 8]

    # Pass Arguments
    mv      a0, t1                  # a0(n - 1) = t1(n - 1)

    # Jump to Callee
    jal     ra, FUNC_TERMIAL        # ra = Addr(mv  t1, a0)
    ########################################################

    ## Get return value
    mv      t1, a0                  # t1 = termial(n-1)

    ## Retrieve Caller Saved
    lw      t0, 0(sp)               # t0 = n
    addi    sp, sp, 4               # Release stack space
                                    # sp = @sp - 4

endAdd:
    add     a0, t0, t1              # if t0 == 1 => t0 = 1 + 0
                                    # if t0  > 1 => t0 = n + termial(n-1)
                                    # a0 = termial(n)

    ## Retrieve ra & Retrieve Callee Saved
    lw      ra, 0(sp)               # ra = @ra
    addi    sp, sp, 4               # Release stack space
                                    # sp = @sp
    ## return
    ret                             # jalr  x0, ra, 0
```

Question List

Q1 :
Why the immediate of jal & branch doesn't include imm[0] ?
What's the advantage of this design?
Why we need jalr ?

Q2 : Please explain RISC-V calling convention in your word

Q3 : Why there is no bgt(u) / ble(u) in standard RV32I ?

Q4 :
We can find that auipc, jal, branch are all PC-relative instructions.
What's the advantage of PC-relative ?