Computer Architecture Homework 4

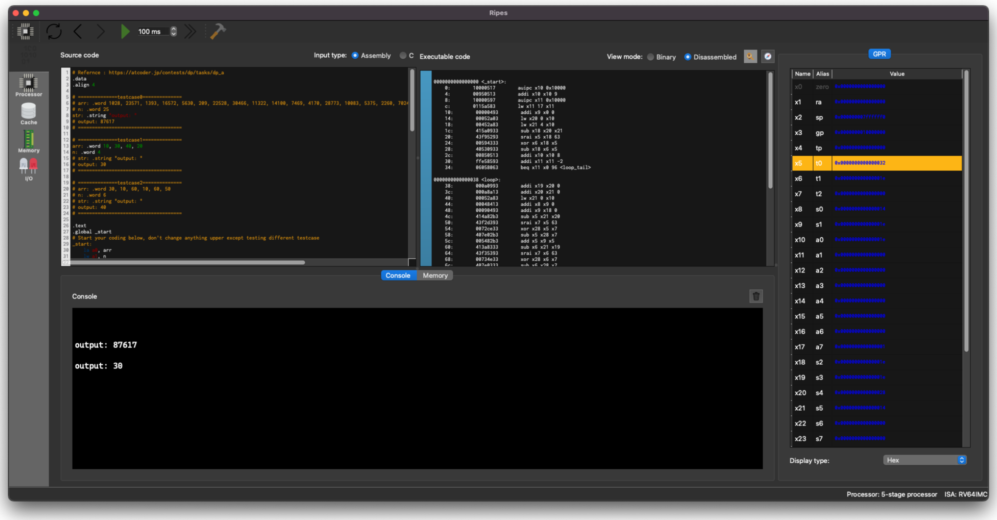
110062219

June 4, 2023

Contents

1 Assembly Coding

1.1 Evidence of Correctness



Source code — Input type: Assembly [✓] / C

```
# Refernce : https://atcoder.jp/contests/dp/tasks/dp_a
.data
.align 4

# ==============testcase0==============
# arr: .word 1028, 23571, 1393, 16572, 5630, 209, 22528, 30466, 11322, 14100, 7469, 4170, 28773, 10083, 5375, 2260, 7024
# n: .word 25
str: .string "output: "
# output: 87617
# ====================================

# ==============testcase1==============
arr: .word 10, 30, 40, 20
n: .word 4
# str: .string "output: "
# output: 30
# ====================================

# ==============testcase2==============
# arr: .word 30, 10, 60, 10, 60, 50
# n: .word 6
# str: .string "output: "
# output: 40
# ====================================

.text
.global _start
# Start your coding below, don't change anything upper except testing different testcase
_start:
    la a0, arr
    lw a1, n
```

Executable code — View mode: Binary / Disassembled [✓]

```
0000000000000000 <_start>:
    0:        10000517        auipc x10 0x10000
    4:        00950513        addi x10 x10 9
    8:        10000597        auipc x11 0x10000
    c:        0115a583        lw x11 17 x11
    10:        00000493        addi x9 x0 0
    14:        00052a03        lw x20 0 x10
    18:        00452a83        lw x21 4 x10
    1c:        415a0933        sub x18 x20 x21
    20:        43f95293        srai x5 x18 63
    24:        00594333        xor x6 x18 x5
    28:        40530933        sub x18 x6 x5
    2c:        00850513        addi x10 x10 8
    30:        ffe58593        addi x11 x11 -2
    34:        06058063        beq x11 x0 96 <loop_tail>

0000000000000038 <loop>:
    38:        000a0993        addi x19 x20 0
    3c:        000a8a13        addi x20 x21 0
    40:        00052a83        lw x21 0 x10
    44:        00048413        addi x8 x9 0
    48:        00090493        addi x9 x18 0
    4c:        414a82b3        sub x5 x21 x20
    50:        43f2d393        srai x7 x5 63
    54:        0072ce33        xor x28 x5 x7
    58:        407e02b3        sub x5 x28 x7
    5c:        005482b3        add x5 x9 x5
    60:        413a8333        sub x6 x21 x19
    64:        43f35393        srai x7 x6 63
    68:        00734e33        xor x28 x6 x7
```

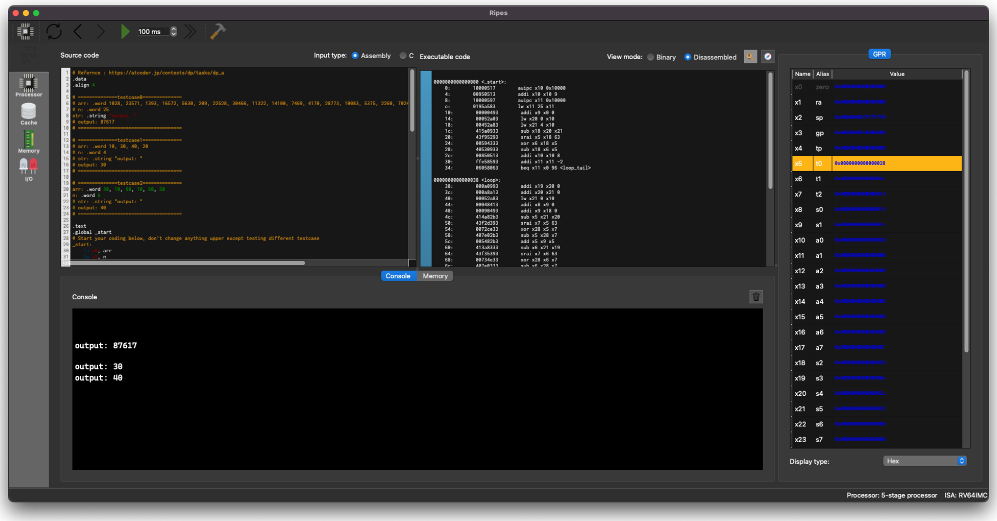
Console

```
output: 87617

output: 30
```

Processor: 5-stage processor ISA: RV64IMC

Figure 1: Testcase 1



Source code — Input type: Assembly [✓] / C

```
# Refernce : https://atcoder.jp/contests/dp/tasks/dp_a
.data
.align 4

# ==============testcase0==============
# arr: .word 1028, 23571, 1393, 16572, 5630, 209, 22528, 30466, 11322, 14100, 7469, 4170, 28773, 10083, 5375, 2260, 7024
# n: .word 25
str: .string "output: "
# output: 87617
# ====================================

# ==============testcase1==============
# arr: .word 10, 30, 40, 20
# n: .word 4
# str: .string "output: "
# output: 30
# ====================================

# ==============testcase2==============
arr: .word 30, 10, 60, 10, 60, 50
n: .word 6
# str: .string "output: "
# output: 40
# ====================================

.text
.global _start
# Start your coding below, don't change anything upper except testing different testcase
_start:
    la a0, arr
    lw a1, n
```

Executable code — View mode: Binary / Disassembled [✓]

```
0000000000000000 <_start>:
    0:        10000517        auipc x10 0x10000
    4:        00950513        addi x10 x10 9
    8:        10000597        auipc x11 0x10000
    c:        0195a583        lw x11 25 x11
    10:        00000493        addi x9 x0 0
    14:        00052a03        lw x20 0 x10
    18:        00452a83        lw x21 4 x10
    1c:        415a0933        sub x18 x20 x21
    20:        43f95293        srai x5 x18 63
    24:        00594333        xor x6 x18 x5
    28:        40530933        sub x18 x6 x5
    2c:        00850513        addi x10 x10 8
    30:        ffe58593        addi x11 x11 -2
    34:        06058063        beq x11 x0 96 <loop_tail>

0000000000000038 <loop>:
    38:        000a0993        addi x19 x20 0
    3c:        000a8a13        addi x20 x21 0
    40:        00052a83        lw x21 0 x10
    44:        00048413        addi x8 x9 0
    48:        00090493        addi x9 x18 0
    4c:        414a82b3        sub x5 x21 x20
    50:        43f2d393        srai x7 x5 63
    54:        0072ce33        xor x28 x5 x7
    58:        407e02b3        sub x5 x28 x7
    5c:        005482b3        add x5 x9 x5
    60:        413a8333        sub x6 x21 x19
    64:        43f35393        srai x7 x6 63
    68:        00734e33        xor x28 x6 x7
```

Console

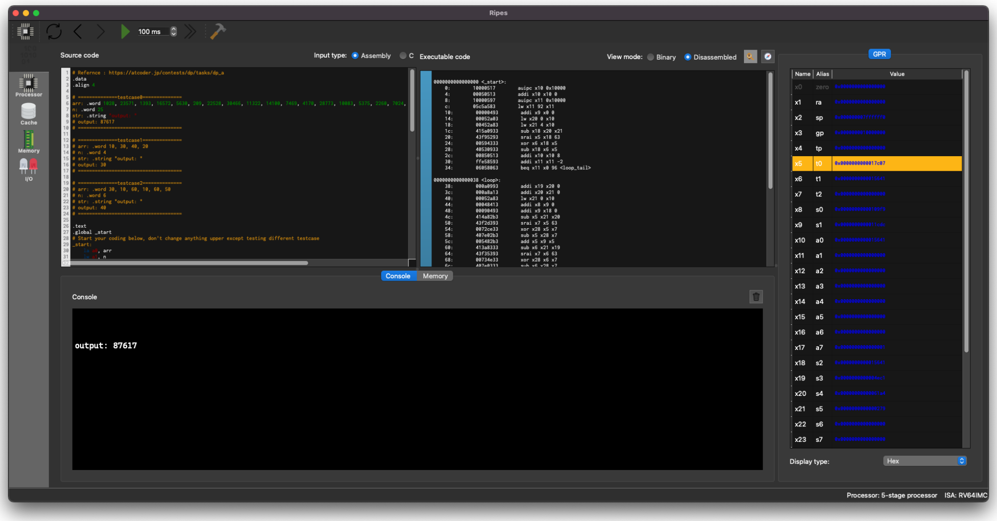
```
output: 87617

output: 30
output: 40
```

Processor: 5-stage processor ISA: RV64IMC

Figure 2: Testcase 2

In addition, I generated a testcase of length 25 and also pass it.



```
# Refernce : https://atcoder.jp/contests/dp/tasks/dp_a
.data
.align 4

# ==============testcase0==============
arr: .word 1028, 23571, 1393, 16572, 5630, 209, 22528, 30466, 11322, 14100, 7469, 4170, 28773, 10083, 5375, 2260, 7024,
n: .word 25
str: .string "output: "
# output: 87617
# ====================================

# ==============testcase1==============
# arr: .word 10, 30, 40, 20
# n: .word 4
# str: .string "output: "
# output: 30
# ====================================

# ==============testcase2==============
# arr: .word 30, 10, 60, 10, 60, 50
# n: .word 6
# str: .string "output: "
# output: 40
# ====================================

.text
.global _start
# Start your coding below, don't change anything upper except testing different testcase
_start:
    la a0, arr
    lw a1, n
```

```
0000000000000000 <_start>:
    0:        10000517        auipc x10 0x10000
    4:        00050513        addi x10 x10 0
    8:        10000597        auipc x11 0x10000
    c:        05c5a583        lw x11 92 x11
    10:        00000493        addi x9 x0 0
    14:        00052a03        lw x20 0 x10
    18:        00452a83        lw x21 4 x10
    1c:        415a0933        sub x18 x20 x21
    20:        43f95293        srai x5 x18 63
    24:        00594333        xor x6 x18 x5
    28:        40530933        sub x18 x6 x5
    2c:        00850513        addi x10 x10 8
    30:        ffe58593        addi x11 x11 -2
    34:        06058063        beq x11 x0 96 <loop_tail>

0000000000000038 <loop>:
    38:        000a0993        addi x19 x20 0
    3c:        000a8a13        addi x20 x21 0
    40:        00052a83        lw x21 0 x10
    44:        00048413        addi x8 x9 0
    48:        00090493        addi x9 x18 0
    4c:        414a82b3        sub x5 x21 x20
    50:        43f2d393        srai x7 x5 63
    54:        0072ce33        xor x28 x5 x7
    58:        407e02b3        sub x5 x28 x7
    5c:        005482b3        add x5 x9 x5
    60:        413a8333        sub x6 x21 x19
    64:        43f35393        srai x7 x6 63
    68:        00734e33        xor x28 x6 x7
```

```
output: 87617
```

Figure 3: Testcase 0

In fact, I came up with the assembly codes from C codes degree by degree. So I submitted my vary low-level, assembly-like, i.e., to name variables after registers and use `goto` and labels instead of loops, C codes to AtCoder and got accepted to ensure the correctness.

1.2 Flowchart

First, I store the address of arr in `a0` and n in `a1`. By means of technique similar to **rolling array**, we optimize the space complexity to constant $O(1)$, i.e., we needn't allocate and maintain the whole dp. Instead, we only need keep track of the 3 recent elements of dp, which are stored in `s0`, `s1` and `s2`, respectively. Likewise, I only record the 3 recent elements of arr in `s3`, `s4` and `s5`, respectively, and advance `a0`, decrement `a1` each time in loop.

The most notable part of my codes is that, as a competitive programmer, I tend to calculate the minimum and abstract values by fancy and elegant **bitwise operations** rather than **branches**. Due to the properties of 2's complement, we have

$$\min\{l, r\} = r \oplus ((l \oplus r) \& -(l < r))$$

and

$$|x| = (x \oplus (x \gg (\#\text{BITS}(x) - 1))) - (x \gg (\#\text{BITS}(x) - 1))$$

, where $\gg$ means **arithmetic right shift** while #BITS(x) represents the number of bits of x.

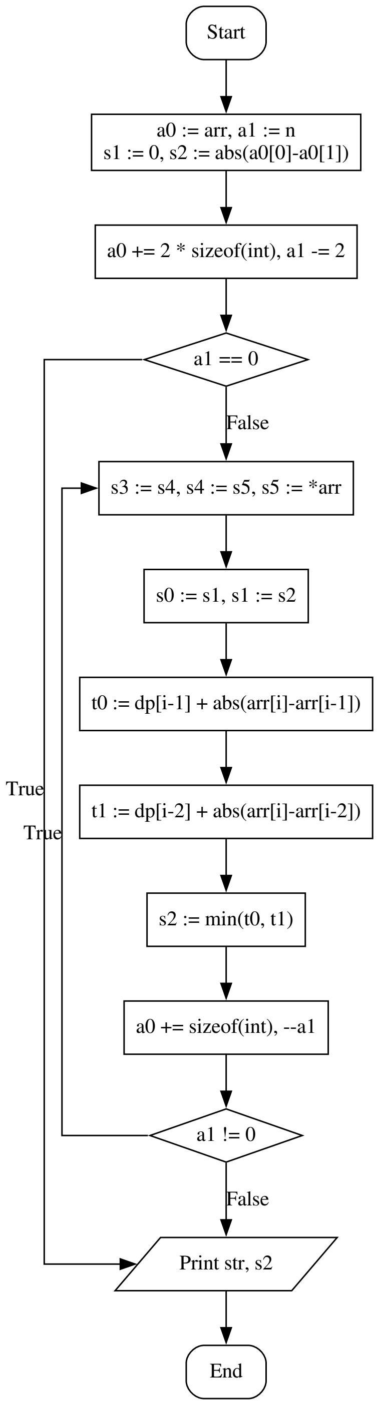

Start
a0 := arr, a1 := n
s1 := 0, s2 := abs(a0[0]-a0[1])
a0 += 2 * sizeof(int), a1 -= 2
a1 == 0
False
True
s3 := s4, s4 := s5, s5 := *arr
s0 := s1, s1 := s2
t0 := dp[i-1] + abs(arr[i]-arr[i-1])
t1 := dp[i-2] + abs(arr[i]-arr[i-2])
True
s2 := min(t0, t1)
a0 += sizeof(int), --a1
a1 != 0
False
Print str, s2
End

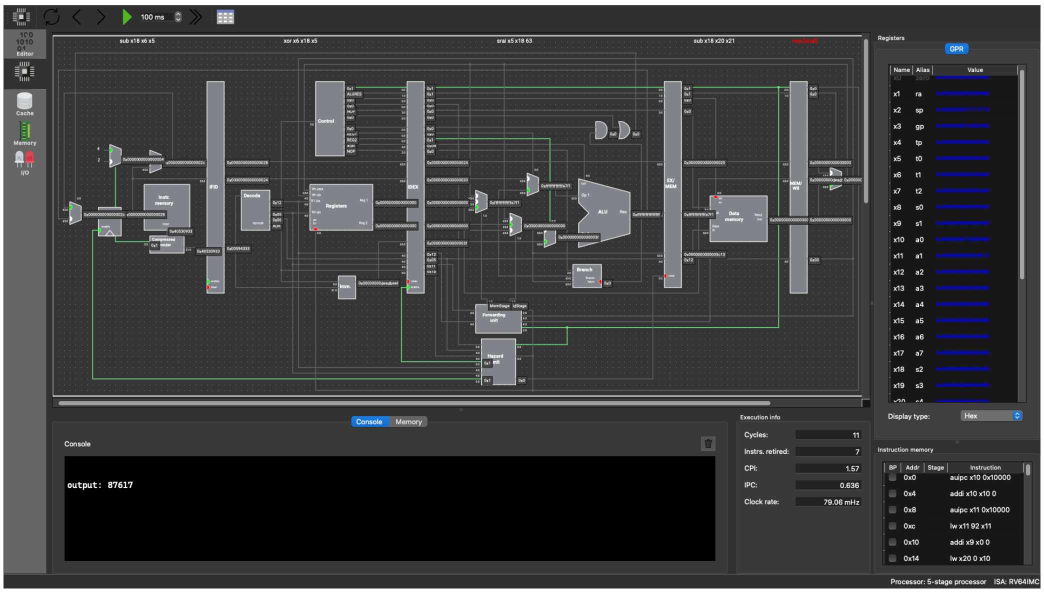
Figure 4: Flowchart

2 Hazards in Codes

2.1 R-type RAW (EX)

The `srai` instruction at line 38 reads the register `s2` as its first operand, which is written by the `sub` instruction at line 36. As a consequence, an EX hazard would occur.

In this case, the forwarding unit would set the signal `alu_reg1_forwarding_ctrl` to be `MemStage`.



Console

```
output: 87617
```

Execution info

Cycles:	11
Instrs. retired:	7
CPI:	1.57
IPC:	0.636
Clock rate:	79.06 mHz

Display type: Hex

Instruction memory

BP	Addr	Stage	Instruction
	0x0		auipc x10 0x10000
	0x4		addi x10 x10 0
	0x8		auipc x11 0x10000
	0xc		lw x11 92 x11
	0x10		addi x9 x0 0
	0x14		lw x20 0 x10

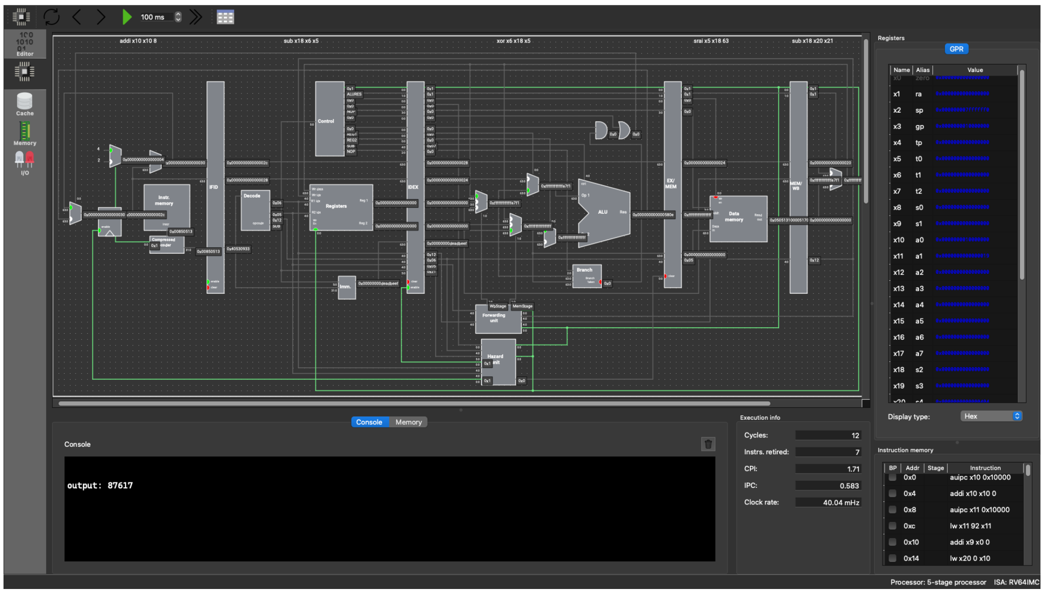
Processor: 5-stage processor ISA: RV64IMC

Figure 5: Type 1

2.2 R-type RAW (MEM)

The `xor` instruction at line 39 reads the register `s2` as its first operand, which is written by the `sub` instruction at line 36. Therefore, a MEM hazard would occur.

In this case, the forwarding unit would set the signal `alu_reg1_forwarding_ctrl` to be `WbStage`.



Console

```
output: 87617
```

Execution info

Cycles:	12
Instrs. retired:	7
CPI:	1.71
IPC:	0.583
Clock rate:	40.04 mHz

Instruction memory

BP	Addr	Stage	Instruction
	0x0		auipc x10 0x10000
	0x4		addi x10 x10 0
	0x8		auipc x11 0x10000
	0xc		lw x11 92 x11
	0x10		addi x9 x0 0
	0x14		lw x20 0 x10

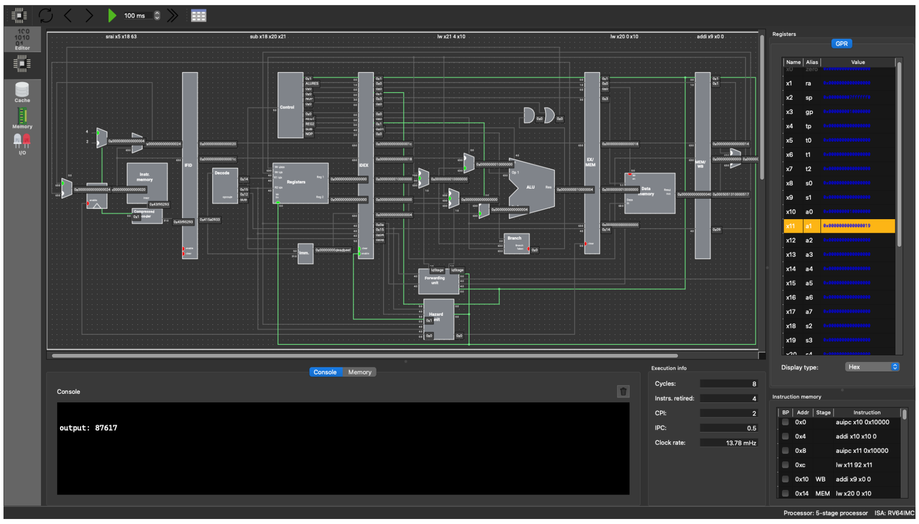
Processor: 5-stage processor ISA: RV64IMC

Figure 6: Type 2

2.3 Load RAW Immediately

The `sub` instruction at line 36 reads the register `s5` as its second operand, which is written by the `lw` instruction at line 35. Accordingly, a load-use hazard would occur.

In this case, the hazard unit have to reset the signal `hazardFEEnable` to stall the pipeline, inserting a `nop` bubble between the two instructions. And then, in the next cycle, the forwarding unit would set the signal `alu_reg2_forwarding_ctrl` to be `WbStage`.



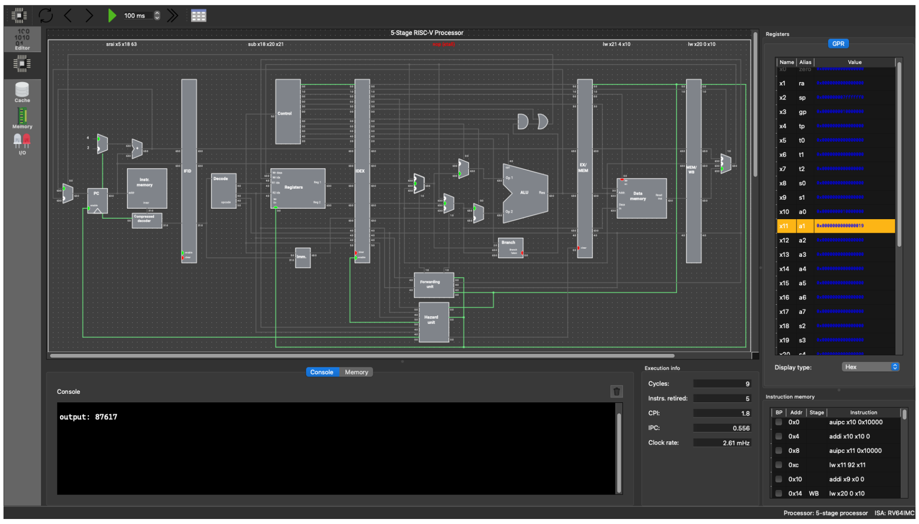
```
output: 87617
```

Cycles: 8
Instrs. retired: 4
CPI: 2
IPC: 0.5
Clock rate: 13.78 mHz

BP	Addr	Stage	Instruction
	0x0		auipc x10 0x10000
	0x4		addi x10 x10 0
	0x8		auipc x11 0x10000
	0xc		lw x11 92 x11
	0x10	WB	addi x9 x0 0
	0x14	MEM	lw x20 0 x10

Processor: 5-stage processor ISA: RV64IMC

(a)



```
output: 87617
```

Cycles: 9
Instrs. retired: 5
CPI: 1.8
IPC: 0.556
Clock rate: 2.61 mHz

BP	Addr	Stage	Instruction
	0x0		auipc x10 0x10000
	0x4		addi x10 x10 0
	0x8		auipc x11 0x10000
	0xc		lw x11 92 x11
	0x10		addi x9 x0 0
	0x14	WB	lw x20 0 x10

Processor: 5-stage processor ISA: RV64IMC

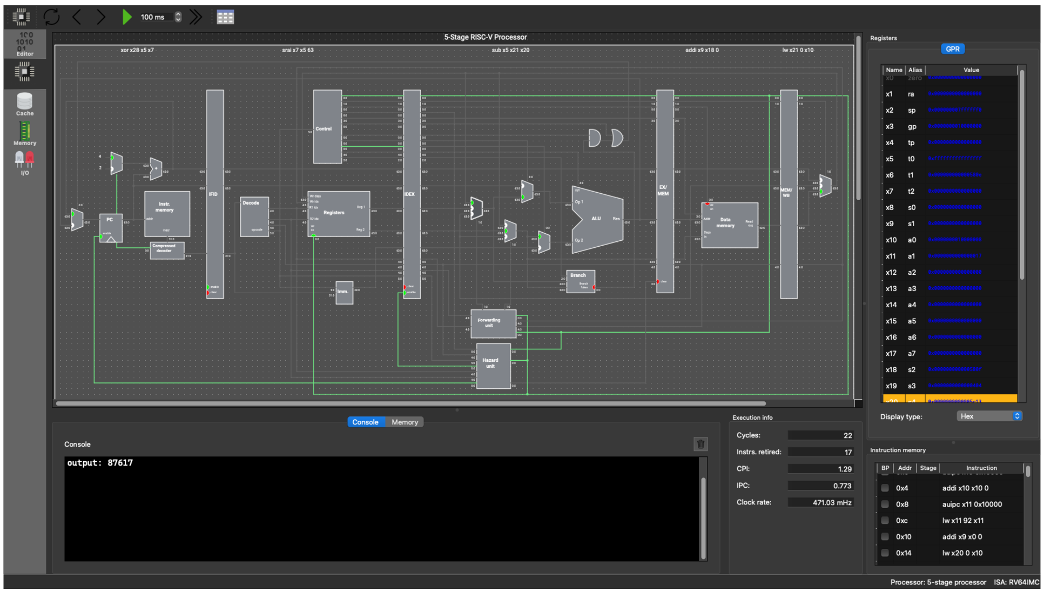
(b)

Figure 7: Type 3

2.4 Load RAW Between One Instruction

The `sub` instruction at line 57 reads the register `s5` as its first operand, which is written by the `lw` instruction at line 52. (There are 2 blank lines and a comment line in between.) Whence a hazard would occur.

In this case, the forwarding unit would set the signal `alu_reg1_forwarding_ctrl` to be `WbStage`.



Registers

Name	Alias	Value
x0	zero	0x0000000000000000
x1	ra	0x0000000000000000
x2	sp	0x000000007ffffff0
x3	gp	0x0000000010000000
x4	tp	0x0000000000000000
x5	t0	0xffffffffffffffff
x6	t1	0x000000000000588e
x7	t2	0x0000000000000000
x8	s0	0x0000000000000000
x9	s1	0x0000000000000000
x10	a0	0x0000000010000008
x11	a1	0x0000000000000017
x12	a2	0x0000000000000000
x13	a3	0x0000000000000000
x14	a4	0x0000000000000000
x15	a5	0x0000000000000000
x16	a6	0x0000000000000000
x17	a7	0x0000000000000000
x18	s2	0x000000000000588f
x19	s3	0x0000000000000404
x20	s4	0x0000000000005c13

Display type: Hex

Console

```
output: 87617
```

Execution info

Cycles:	22
Instrs. retired:	17
CPI:	1.29
IPC:	0.773
Clock rate:	471.03 mHz

Instruction memory

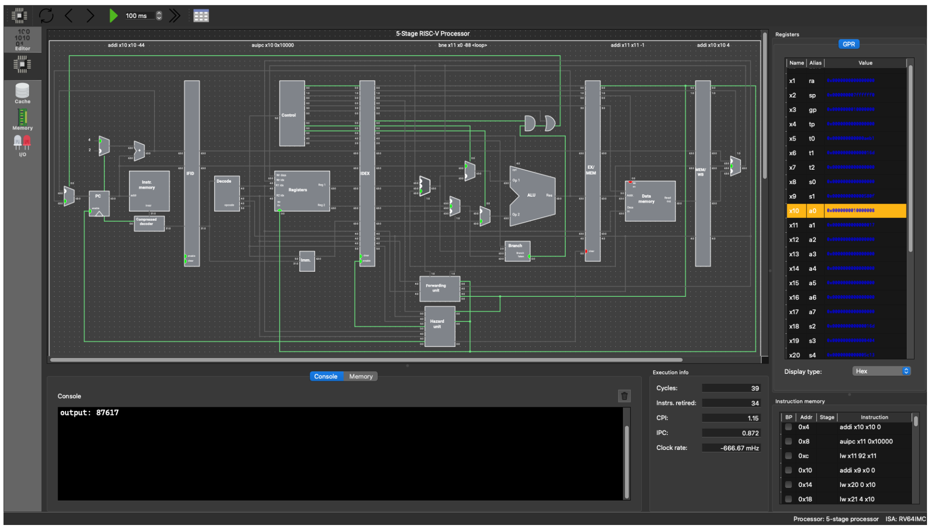
BP	Addr	Stage	Instruction
	0x4		addi x10 x10 0
	0x8		auipc x11 0x10000
	0xc		lw x11 92 x11
	0x10		addi x9 x0 0
	0x14		lw x20 0 x10

Processor: 5-stage processor ISA: RV64IMC

Figure 8: Type 4

2.5 Branches

Ripes happens to be of the **always-not-taken** policy. So when branches taken place actually, for instance at line 80, then the `clear` of `ID/EX` and `IF/ID` must be triggered and the pipeline is flushed thereby.



Console
```
output: 87617
```

Execution info

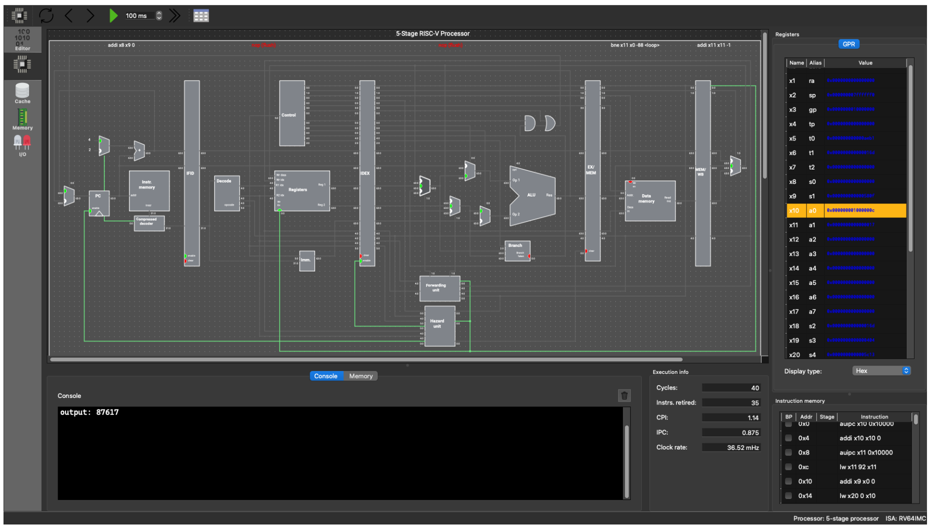
Cycles:	39
Instrs. retired:	34
CPI:	1.15
IPC:	0.872
Clock rate:	-666.67 mHz

Instruction memory

BP	Addr	Stage	Instruction
	0x4		addi x10 x10 0
	0x8		auipc x11 0x10000
	0xc		lw x11 92 x11
	0x10		addi x9 x0 0
	0x14		lw x20 0 x10
	0x18		lw x21 4 x10

Processor: 5-stage processor ISA: RV64IMC

(a)



Console
```
output: 87617
```

Execution info

Cycles:	40
Instrs. retired:	35
CPI:	1.14
IPC:	0.875
Clock rate:	36.52 mHz

Instruction memory

BP	Addr	Stage	Instruction
	0x0		auipc x10 0x10000
	0x4		addi x10 x10 0
	0x8		auipc x11 0x10000
	0xc		lw x11 92 x11
	0x10		addi x9 x0 0
	0x14		lw x20 0 x10

Processor: 5-stage processor ISA: RV64IMC

(b)

Figure 9: Type 5