Sergio Garcia Tapia
Digital Design and Computer Architecture: RISC-V, by Sarah and David Harris
Chapter 6: Architecture
April 8, 2024

Exercise 6.1. Give three examples from the RISC-V architecture of each of the following design principles: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises. Explain how each of your examples exhibits the design principle.

Solution:

(1) In Section 6.2.1, the `add` and `sub` instructions were introduced, and they both have the same format: `mnemonic source1 source1 destination`. The format is consistent with nearly all RISC-V binary operations, making them predictable and their binary encoding consistent and simple. For example, the **func7** bits are all 0 for all binary operations with an `op` code of 51; they differ only in their `funct3` control bits. Another example is that operands all 32 bits in RISC-V 32, in contrast with x86-32 where operands can be 8-bit, 16-bit, or 32-bit. Also instructions in x86-64 vary in size, whereas in RISC-V they are uniformly 16 bits, as mentioned in Section 6.8.5.

(2) As discussed in 6.2.1, RISC-V has a relatively small set of simple instructions that are commonly used, in contrast with CISC architectures that provide complex operations and therefore add overhead to simple instructions. In Section 6.8.6, the author mentions that x86 contains string operations that are usually slower than performing the equivalent operation with a series of simple instructions. By supporting simple byte operations that are fast, RISC-V can reap these benefits.

(3) RISC-V32 has 32 registers, which are sufficient to perform several operations without having to add local variables to the stack in the case of small functions. Since the stack relies on larger memory, accessing is slower, so avoiding this is best.

(4) In Section 6.4, the author mentions compromise regarding the length of the instruction encoding; some instructions do not require 32 bits, but they are encoded as such anyway to support simplicity. Moreover, instead of a single instruction format, there are four to provide enough flexibility. Another example is the bit-swizzling of the immediate encodings discussed in 6.4.5. Though complicated, it ensures that other instruction fields are consistent and simple.

Exercise 6.2. The RISC-V architecture has a register set that consists of 32 32-bit registers. Is it possible to design a computer architecture without a register set? If so, briefly describe the architecture, including the instruction set. What are the advantages and disadvantages of this architecture over the RISC-V architecture?

Solution: According to Section 6.2.2, registers exist to support access to operands quickly so that they can run fast; they do this by providing a small address space. Therefore, it is possible to not use a register set and use memory. One way would be to reserve the first 32 address locations of memory for the same functions as the corresponding registers. According to Section 5.5.4, memory is commonly implemented with DRAM, whereas a register set is implemented with SRAM. DRAM latency is longer than SRAM because its bitline is not actively driven by a transistor; it fundamentally has lower throughput because data must be refreshed periodically and after a read. Moreover, latency and throughput depend on memory size, an therefore since DRAM is typically very large in comparison to a small register set, this means relying on just memory would be too slow. One advantage is that it generally takes less transistors to build DRAM in comparison to SRAM, making them cheaper and less power hungry.

Exercise 6.3. Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

(a) hello there

(b) bag o'chips

(c) To the rescue!

Solution:

(a) Referring to an ASCII take encoding, we see that the encoding is:

HEX	68	65	6c	6c	6f	20	74	68	65	72	65
String	h	e	l	l	o		t	h	e	r	e

(b)

HEX	62	61	67	20	27	6f	63	68	69	70	73
String	b	a	g		o	'	c	h	i	p	s

(c)

HEX	54	6f	20	74	68	65	20	72	65	73	63	75	65	21
String	T	o		t	h	e		r	e	s	c	u	e	!

Exercise 6.4. Repeat Exercise 6.3 for the following strings.

1. Cool
2. RISC-V
3. boo!

Solution:

(a)

HEX	43	6f	6f	6c
String	C	o	o	l

(b)

HEX	52	49	53	43	2d	56
String	R	I	S	C	-	V

(c)

HEX	62	6f	6f	21
String	b	o	o	!

Exercise 6.5. Show how the strings in Exercise 6.3 are stored in byte-addressable memory starting at memory address `0x004F05BC`. The first character of the string is stored at the lowest byte address (in this case, `0x004F05BC`). Clearly indicate the memory address of each byte.

Solution: I am using word addresses that increase upwards. The last character is therefore at the top.

(a)

hello there

Byte	Word Address
65	0x004F05C6
72	0x004F05C5
65	0x004F05C4
68	0x004F05C3
74	0x004F05C2
20	0x004F05C1
6f	0x004F05C0
6c	0x004F05BF
6c	0x004F05BE
65	0x004F05BD
68	0x004F05BC

(b)

bag o'chips	
Byte	Word Address
73	0x004F05C6
70	0x004F05C5
69	0x004F05C4
68	0x004F05C3
63	0x004F05C2
6f	0x004F05C1
27	0x004F05C0
20	0x004F05BF
67	0x004F05BE
61	0x004F05BD
62	0x004F05BC

(c)

To the rescue!	
Byte	Word Address
21	0x004F05C9
65	0x004F05C8
75	0x004F05C7
63	0x004F05CD
73	0x004F05C5
65	0x004F05C4
72	0x004F05C3
20	0x004F05C2
65	0x004F05C1
68	0x004F05C0
74	0x004F05BF
20	0x004F05BE
6f	0x004F05BD
54	0x004F05BC

Exercise 6.5. Repeat Exercise 6.5 for the strings in Exercise 6.4.

Solution: As in Exercise 6.5, I am using word addresses that increase upwards. The last character is therefore at the top.

(a)

Cool	
Byte	Word Address
6c	0x004F05BF
6f	0x004F05BE
6f	0x004F05BD
43	0x004F05BC

(b)

RISC-V	
Byte	Word Address
56	0x004F05C1
2d	0x004F05C0
43	0x004F05BF
53	0x004F05BE
49	0x004F05BD
52	0x004F05BC

(c)

boo!	
Byte	Word Address
21	0x004F05C3
6f	0x004F05C2
6f	0x004F05C1
62	0x004F05C0

Exercise 6.7. The `nor` instruction is not a part of the RISC-V instruction set because the same fucntionality can be implemented using existing instructions. Write a short assembly code snippet that has the same functionality: `s3 = s4 NOR s5`. Use as few instructions as possible.

Solution: Recall that NOR is short for NOT OR, so A NOR B means $\overline{A+B}$, where $+$ is the OR operator and the overline is the NOT operator. The truth table is below:

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

Recalling that NOT can be implemented with `xori` with `-1` as the immediate, do precisely an OR followed by a NOT:

```
or s6, s4, s5
xori s3, s6, -1
```

Exercise 6.8. The `nand` instruction is not a part of the RISC-V instruction set because the same fucntionality can be implemented using existing instructions. Write a short assembly code snippet that has the same functionality: `s3 = s4 NAND s5`. Use as few instructions as possible.

Solution: Recall that NAND is short for NOT AND, so A NAND B means $\overline{A \cdot B}$, where $\cdot$ is the AND operator and the overline is the NOT operator. The truth table is below:

A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

Recalling that NOT can be implemented with `xori` with `-1` as the immediate, do precisely an AND followed by a NOT:

```
and s6, s4, s5
xori s3, s6, -1
```

Exercise 6.9. Convert the following high-level code snippets into RISC-V assembly language. Assume that the (signed) integer variables `g` and `h` are in register `a0` and `a1`, respectively. Clearly comment your code:

(a)

```
if (g > h)
    g = g + 1;
else
    h = h - 1;
```

(b)

```
if (g <= h)
    g = 0;
else
    h = 0;
```

Solution:

(a) We can use a conditional jump, `bge`, to go to the `else` branch when `h >= g`. To ensure we do not execute the `else` branch if we take the `if` branch, we use an unconditional jump with the `j` instruction to the label after the last instruction in the `else` branch. To add the constant, we can use the `addi` command. The assembly follows:

```
    bge     a1, a0     # if h >= g goto else at .L1
    addi    a0, a0, 1  # g = g + 1
    j .L2              # Skip else branch
.L1:
    addi    a1, a1, -1, # h = h - 1
.L2:
```

(b) We can use `blt` to go to the `else` branch if `h > g` (equivalently, `g > h`).

```
    blt     a1, a0      # if h < h goto else at .L1
    addi    a0, zero, 0 # g = 0
    j .L2               # Skip else branch
.L1:
    addi    a1, zero, 0 # h = 0
.L2:
```

Exercise 6.10. Repeat Exercise 6.9 for the following code snippets:

(a)

```
if (g >= h)
    g = g + h;
else
    g = g - h;
```

(b)

```
if (g < h)
    h = h + 1;
else
    h = h * 2;
```

Solution:

(a)

```
        blt     a1, a0      # if h < g goto else at .L1
        add     a0, a0, a1  # g = g + h
        j .L2               # Skip else branch
.L1:
        sub     a0, a0, a1, # g = g - h
.L2:
```

(b) Instead of multiplying by 2 with the `mul` instruction, I used `sll` to shift left by 1 bit, which is equivalent.

```
        bge     a0, a1      # if g >= h goto else at .L1
        addi    a1, a1, 1   # h = h + 1
        j .L2               # Skip else branch
.L1:
        sll     a1, a1, 1   # h = h * 2
.L2:
```

Exercise 6.11. Convert the following high-level code snippet into RISC-V assembly. Assume that the base address of `array1` and `array2` are held in `t1` and `t2` and that `array2` array is initialized before it is used. Use as few instructions as possible. Clearly comment your code.

```
int i;
int array1[100];
int array2[100];
# ... more code
for (i = 0; i < 100; i = i + 1)
    array1[i] = array2[i];
```

Solution:

```
    addi   sp,sp,-16  # Allocate space on stack
    sw     s0,12(sp)  # Store saved register on stack
    sw     s1,8(sp)   # Store saved register on stack
    addi   s0,t1,0    # Move array1 to saved register.
    addi   s1,t2,0    # Move array 2 to saved register.
    addi   t0,zero,0  # i = 0
    addi   t1,zero,100 # t1 = 100
.LOOP:
    bge    s0,t0,.AFTER_LOOP
    slli   t2,t0,2    # t2 = i * 4
    add    t3,s1,t2   # Address of array2[i]
    lw     t4, 0(t3)  # value stored at array2[i]
    addi   t5,s0,t2   # Address of array1[i]
    sw     t4, 0(t5)  # Store value of array2[i] into array1[i]
    addi   t0,t0,1    # i = i + 1
    j      .LOOP      # Repeat
.AFTER_LOOP:
    lw     s0,12(sp)  #  Restore saved registers
    lw     s1,8(sp)
    addi   sp,sp,16   # Deallocate space on stack
```

Exercise 6.12. Repeat Exercise 6.11 for the following high-level code snippet. Assume that the `temp` array is initialized before it is used and that `t3` holds the base address of `temp`.

```
int i;
int temp[100];
...
for (i = 0; i < 100; i = i + 1)
    temp[i] = temp[i] * 128;
```

Solution:

```
    addi   sp,sp,-16  # Allocate space on stack
    sw     s0,12(sp)  # Store saved register on stack
    addi   s0,t3,0    # Move temp to saved register.
    addi   t0,zero,0  # i = 0
    addi   t1,zero,100 # t1 = 100
.LOOP:
    bge    s0,t0,.AFTER_LOOP
    slli   t2,t0,2    # t2 = i * 4
    add    t2,s0,t2   # Address of temp[i]
    lw     t3, 0(t2)  # Value stored at temp[i]
    slli   t3,t3,7    # temp[i] * 128
    sw     t3, 0(t2)  # temp[i] * 128 at temp[i]
    addi   t0,t0,1    # i = i + 1
    j      .LOOP      # Repeat
.AFTER_LOOP:
    lw     s0,12(sp)  #  Restore saved registers
    addi   sp,sp,16   # Deallocate space on stack
```

Exercise 6.13. Write RISC-V assembly code for placing the following immediate (constants) in `s7`. Use a minimum number of instructions.

(a) 29

(b) -214

(c) -2999

(d) `0xABCDE000`

(e) `0xEDCBA123`

(f) `0xEEEEEFAB`

Solution:

(a) `addi s7, zero, 29`

(b) `addi s7, zero, -214`

(c) -2999 is `1111010001001001` as a 16-bit two's complement number. The upper 4 bits are `1111`, and the lower 4 bits are `010001001001`. Since the sign extension of the lower 12 bits would not add 1s, we need to use `lui` to add them:

```
lui s7,-1            # adds all 1s to upper 20 bits
addi s7,0b010001001001 # Append lower 12 bits
```

(d) `lui s7, 0xABCDE`

(e) The lower 12 bits represented by `0x123` are not signed, we can use `lui` without worrying about the sign extension due to `addi`

```
lui s7, 0xEDCBA   # Upper 12 bits
addi s7,s7,0x123  # Sign-extension will subtract 1 from upper bits
```

(f) Since the lower 12 bits `0xFAB` is negative, the sign extension will add 1s to the higher 20 bits, so we must add an extra 1 to the upper immediate

```
lui s7,0xEEEEF000 # Store 1 bit higher than 0xEEEEE
addi s7,s7,0xFAB  # Sign-extension will subtract 1 from upper bits.
```

Exercise 6.14. Repeat Exercise 6.13 for the following immediates.

(a) 47

(b) -349

(c) 5328

(d) `0xBBCCD000`

(e) `0xFEEBC789`

(f) `0xCCAAB9AB`

Solution:

(a) `addi s7,zero,47`

(b) `addi s7,zero,-349`

(c) Since 5328 is `01010011010000` in 14-bit two's complement binary representation, we need to load the upper bits `01` first, adding 1 to it in anticipation of the sign extension from `addi`:

```
lui s7,0b10              # 1 higher than necessary
addi s7,s7,0b010011010000 # Lower 12 bits
```

(d) `lui s7,0xBBCCD`

(e) Since `0x789` is positive, the sign extension will not add 1s to the higher 20 bits, so we do not add 1 to the upper immediate.

```
lui s7,0xFEEBC    # 1 higher than necessary
addi s7,s7,0x789  # sign extension subtacts 1
```

(f) This time we do need to add 1 to the upper immediate since the leading bit of the lower immediate 0x9AB is 1.

```
lui s7,0xCCAAC    # 1 higher than necessary
addi s7,s7,9AB    # Sign extension subtracts 1e
```

Exercise 6.15. Write a function in a high-level language for

```
int find42(int array[], int size)
```

`size` specifies the number of elements in `array`, and `array[]` specifies the base address of the array. The function should return the index number of the first entry that holds the value 42. If no array entry is 42, it should return -1. Clearly comment your code.

Solution:

```
int find42(int array[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        if (array[i] == 42)
            return i;         // Return index matching 42
    return -1;                // Not found
}
```

Exercise 6.16. The high-level `strcpy` (string copy) copies the character string `src` to the character string `dst`

```
// C code
// WARNING: This is vulnerable to a buffer overflow attack.
void strcpy(char dst[], char src[]) {
    int i = 0;
    do {
        dst[i] = src[i];
    } while (src[i++]);
}
```

(a) Implement the `strcpy` function in RISC-V assembly. Use `t0` for `i`.

(b) Draw a picture of the stack before, during, and after the `strcpy` function call. Assume `sp` = `0xFFC000` just before `strcpy` was called.

Solution:

(a)

```
   # void strcpy(char dst[], char src[]);
strcpy:
   addi   t0,zero,0     # i = 0
.loop:
   slli   t1,t0,2       # t1 = i * 4
   addi   t2,a0,t1      # address of dst[i]
   addi   t3,a1,t1      # address of src[i]
   lw     t3, 0(t3)     # value of src[i]
   sw     t3, 0(t2)     # store src[i] into dst[i]
   addi   t0,t0,1       # i = i + 1
   bne    t3, .loop     # if src[i] != 0 goto .loop
.return
   jr     ra            # return
```

(b) My implementation used only temporary registers so the stack frame for `strcpy` is empty. The stack frame for the calling function contains the elements of `src` and `dst`, since local arrays are stored on the stack.

Exercise 6.17. Convert the high-level function from Exercise 6.15 into RISC-V assembly code. Clearly comment your code.

Solution:

```
   # int find42(int array[], int size)
find42:
   addi   t0, zero, 0      # i = 0
   addi   t1, zero, 42     # t1 = 42
.loop:
   bge    t0,a1,.not_found # if i >= size goto .not_found
   slli   t2,t0,2          # t2 = i * 4
   add    t2,a0,t2         # add base address of array to t2
   lw     t3, 0(t2)        # get value at array[i]
   be     t3,t1            # if array[i] == 42 goto .found
   addi   t0,t0, 1         # i = i + 1
   j      .loop            # Repeat
.found:
   add    a0,zero,t0       # found, return i
   jr     ra               # return
.not_found:
   addi a0, zero,-1        # not found, returning -1
   jr ra                   # return
```

Exercise 6.18. Consider the RISC-V assembly code below. `func1`, `func2`, and `func3` are nonleaf functions. `func4` is a leaf function. The code is not shown for each function, but the comments indicate which registers are used within each function. You may assume that the functions do not need to save any nonpreserved registers on their stacks.

```
0x00091000 func1: ... # func1 uses t2-t3, s4-s10
...
0x00091020    jal func2
...
0x00091100 func2: ... # func2 uses a0-a2, s0-s5
...
0x0009117C    jal func3
...
0x00091400 func3: ... # func3 uses t3, s7-s9
...
0x00091704    jal func4
...
0x00093008 func4: ... # func4 uses s10-s12
...
0x00093118    jr ra
...
```

(a) How many words are the stack frames of each function?

(b) Sketch the stack after `func4` is called. Clearly indicate which registers are stored where on the stack and mark each of the stack frames. Give values where possible. Assume that `sp` = `0xABC124` just before `func1` is called.

Solution:

(a) By the callee-saved rule, `func1` pushes `s4` through `s10` and `ra` on the stack so it can use those registers. By the instructions, functions do not need to save non-preserved registers, so `t2-t3` are not saved. Therefore, `func1` has a stack frame that is 8 words deep.

By the callee-saved rule, `func2` pushes `s4`, `s5`, and `ra` on the stack, since they are callee-preserved registers. Its stack frame is three words deep.

`func3` pushes `s7-s9` and `ra` on the stack, making its stack frame 4 words deep.

`func4` pushes `s10` on the stack, making its stack frame 1 word deep. Note that the return address register need not be pushed because it is a leaf function.

(b) See the Figure 1:

Exercise 6.19. Each number in the Fibonacci series is the sum of the two previous two numbers. The following table lists the first few numbers in the sequence, `fib(n)`.

n	1	2	3	4	5	6	7	8	9	10	11
`fib(n)`	1	1	2	3	5	8	13	21	34	55	89

(a) What is `fib(n)` for $n = 0$ and $n = -1$?

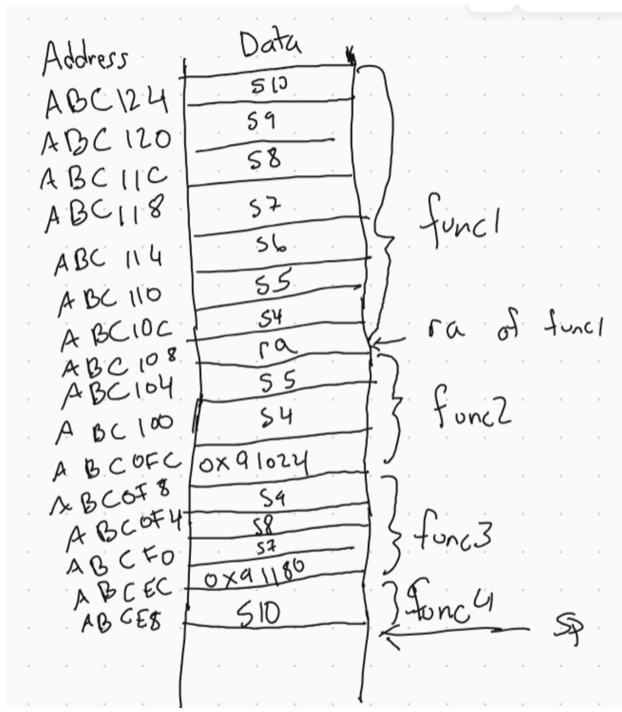

Figure 1: Exercise 6.18: Stack frame after calling `func4`

(b) Write a function called `fib` in a high-level language that returns the Fibonacci number for any non-negative value of `n`. Hint: You probably will want to use a loop. Clearly comment your code.

(c) Convert the high-level function of part (b) into RISC-V assembly code. Add comments after every line that explain clearly what it does. Use a simulator to test your code on `fib(9)`. (See the preface for links to a RISC-V simulator).

Solution:

(a) `fib(n)` is not defined for $n = 0$ or $n = -1$; it is only defined for non-negative integers.

(b) The following is a rudimentary implementation of `fib(n)` in C:

```
/* fib: Returns the n-th Fibonacci term for non-negative n; returns -1 on
   bad input or overflow */
long fib(long n) {
    if (n <= 0)    # Bad input
        return -1;
    long f0 = 0;
    long f1 = 1;
    for (long m = 1; m < n && f1 >= f0; m++) {
        long temp = f1;
        f1 += f0;
        f0 = temp;
    }
```

```
    if (f1 <= f0)      # Overflow
        return -1;
    return f1;
}
```

(c)

```
.globl main
main:
    addi    a0,zero,9
    jal     fib
    jr      ra
    # long fib(long n);
fib:
    bge     zero,a0,.failed # if n <= 0 goto .failed
    addi    t0,zero,0       # f0 = 0
    addi    t1,zero,1       # f1 = 1
    addi    t2,zero,1       # m = 1
.loop:
    bge     t2,a0,.end      # if m >= n
    blt     t1,t0,.failed # if f0 >= f1 goto .end
    addi    t3,t1,0         # temp = f1;
    add     t1,t1,t0        # t1 = t1 + t0
    addi    t0,t3,0         # t0 = temp
    addi    t2,t2,1         # m++
    j       .loop           # repeat
.end:
    bge     t0,t1,.failed # if f0 >= f1 goto .failed
    addi    a0,t1,0         # set f1 as return value
    jr      ra              # return
.failed:
    addi    a0,zero,-1 # return -1;
    jr      ra
```

When run at `https://venus.kvakil.me/`, it placed `0x22` in register `a0`, which gives 34.

Exercise 6.20. Consider Code Example 6.28. For this exercise, assume *factorial(n)* is called with input argument `n = 5`.

(a) What value is in `a0` when `factorial` returns to the calling function?

(b) Suppose you replace the instructions at address `0x8508` and `0x852C` with nops. Will the program:

(1) Enter an infinite loop but not crash;

(2) crash (cause the stack to grow or shrink beyond the dynamic data segment or the PC to jump to a location outside the program);

(3) produce an incorrect value in a0 when the program returns to loop (if so, what value?); or

(4) run correctly despite the deleted lines?

(c) Repeat part (b) with the following instruction modifications:

(1) Replace the instructions at addresses 0x8504 and 0x8528 with nops.

(2) Replace the instruction at address 0x8518 with a nop.

(3) Replace the instruction at address 0x8530 with a nop.

Solution: The code for factorial is below:

```
factorial:
0x8500: addi  sp, sp,    -8     # make room for a0, ra
0x8504: sw    a0  4(sp)
0x8508: sw    ra, 0(sp)
0x850C: addi  t0, zero,  1      # temporary = 1
0x8510: bgt   a0, t0,    else   # if n>1 goto else
0x8514: addi  a0, zero,  1      # otherwise return 1
0x8518: addi  sp, sp,    8      # restore sp
0x851C: jr    ra                # return
else:
0x8520: addi  a0, a0,    -1     # n = n - 1
0x8524: jal   factorial         # recursive call
0x8528: lw    t1, 4(sp)         # restore n into t1
0x852C: lw    ra, 0(sp)         # restore ra
0x8530: addi  sp, sp,    8      # restore sp
0x8534: mul   a0, t1,    a0     # a0 = n * factorial(n-1)
0x8538: jr    ra                # return
```

(a) a0 will have the value $5! = 120$.

(b) Instruction 0x8508 pushes the ra register onto the stack in case the function recursively calls itself. Instruction 0x8528 restores loads n from the stack onto register t1.

When the function is called with n set to 5, it calls itself recursively without storing ra on the stack. However, the jal factorial instruction at each recursion level overwrites ra. This means that when it reaches the base case, calling itself with n set to 1, it returns to the level where n is 2, and then the instruction at address 0x852C, namely lw ra, 0(sp), will fail to find an address at that location. Since that stack address was not initialized (as a consequence of making 0x8058 a nop), it may have an existing value that is invalid, causing the program to crash because the PC will jump to a location outside the program.

(c) If 0x8504 and 0x8528 are replaced with nops, then when we get to the instruction at address 0x8534 to compute n * factorial(n-1), register t1 will not have the expect value n. Suppose that t1 had value x before factorial(5) was called. An incorrect value will be produced in `a0` when all `factorial(5)` returns, namely x^4, assuming no overflow.

If the instruction at `0x8518` is replaced with a nop, then the base case will not restore the stack pointer. When the instruction at address `0x852C` is executed, `factorial(2)` will return to address `0x8528` again as it should. However, `factorial(2)` will return 1 instead of 2, and similarly, `factorial(3)` will return 2, `factorial(4)` will return 6, and instead of ending, `factorial(5)` will jump again to `0x8528`, computing one more iteration. This time it will find `5` at `4(sp)`. Surprisingly, it will compute the correct return value.

The instruction at `0x8530` restores the stack pointer when a non-leaf instance of `factorial` is about to return. This means that `factorial(2)` will return to `factorial(3)` but `factorial(3)` will see the same stack values that `factorial(2)` saw. The code results in an infinite loop.