ECE 2300 Digital Logic and Computer Organization

Topic 9: Instruction Set Architecture

http://www.csl.cornell.edu/courses/ece2300
School of Electrical and Computer Engineering
Cornell University

revision: 2025-12-09-12-43

List of Problems

Problem 1. Pico-Processor Instruction Set Architecture

We will start off by working with the ISA of the pico-processor introduced in lecture. We will practice both *assembling* instructions (converting assembly to machine code) and *disassembling* machine code (converting machine code back to assembly instructions).

Part 1.A Assemble

Convert the following pico-processor assembly instruction into machine code:

```
add rB
```

Part 1.B Disassemble

Convert the following pico-processor machine code to assembly language:

```
01000011
```

Part 1.C Reverse Engineering Your Friend's Code (Again)

While hacking your friend's pico-processor (as usual), you successfully extracted the following machine code. You are wondering what your friend is computing. **Convert the machine code into assembly language to find out.**

```
1  01000000
2  00100001
3  01011000
4  00111000
5  01011000
6  00111000
7  01011000
8  00111000
9  01011000
10 00111000
```

What sequence is being computed?

What issue would occur when executing the instruction at address 9 (line 10)?

Problem 2. TinyRV1 Instruction Set Architecture

In the following assignments, we will *execute* TinyRV1 assembly code using the provided sheets. Use the register and memory fields on the sheets to track the state as you step through each instruction (similar to the lectures and discussion sections).

Part 2.A Program 1: Subtraction

We will start with a *simple* subtraction computation. Unfortunately, since we do not have a subtraction assembly instruction in TinyRV1, it becomes a bit more complicated. Check it out:

```
□□□ □□□ 00 addi x5, x0, -1
□□□ □□□ 04 lw   x6, 256(x0)  # x6 = 71
□□□ □□□ 08 lw   x7, 260(x0)  # x7 = 87
□□□ □□□ 12 mul  x7, x7, x5   # x7 = -87
□□□ □□□ 16 add  x5, x6, x7   # x5 = 71 + (-87)
□□□ □□□ 20 sw   x5, 256(x0)
```

Program Counter

Registers

x31
x30
x29
x28
...
x11
x10
...
x7
x6
x5
...
x0

Memory

508	
...	
280	
276	
272	
268	
264	
260	87
256	71
...	
56	
52	
48	
44	
40	
36	
32	
28	
24	
20	
16	
12	
8	
4	
0	

Part 2.B Program 2: Indirect Pointer Operation

In this problem, we will execute a program that uses an indirect pointer operation. Indirect pointers are pointers that point to another pointer, which then points to the actual value. For instance, in this example, the outer pointer points to memory address 256. At address 256, another pointer is stored, which points to memory address 504. At memory address 504, the actual data (48) is stored.

```
□□□ □□□ 00 addi x5, x0, 256
□□□ □□□ 04 lw   x6, 0(x5)
□□□ □□□ 08 lw   x7, 0(x6)
□□□ □□□ 12 addi x7, x7, -6
□□□ □□□ 16 addi x5, x5, 4
□□□ □□□ 20 lw   x6, 0(x5)
□□□ □□□ 24 sw   x7, 0(x6)
```

Program Counter

Registers

x31
x30
x29
x28
...
x11
x10
...
x7
x6
x5
...
x0

Memory

508	
504	48
...	
276	
272	
268	
264	
260	276
256	504
...	
56	
52	
48	
44	
40	
36	
32	
28	
24	
20	
16	
12	
8	
4	
0	

Part 2.C Program 3: Infinite Loop

Unfortunately, we messed up the continuation condition of our *for* loop. Execute the code for 5 iterations. **What major issue could arise if you did not stop the loop? What would we need to change to fix this issue?**

```
□□□ □□□ 00 addi x5, x0, -1  # x5 = -1
□□□ □□□ 04 addi x6, x0, 56  # addr
□□□ □□□ 08 addi x7, x0, 7
□□□ □□□ 12 sw   x5, 0(x6)   # <--------------|
□□□ □□□ 16 addi x5, x5, -1  # x5 = x5-1      |
□□□ □□□ 20 addi x6, x6, -4  #                 |
□□□ □□□ 24 bne  x5, x7, 12  # if x5 != 7 ----|
□□□ □□□ 28 addi x0, x0, 0
```

Program Counter

Registers

x31
x30
x29
x28
...
x11
x10
...
x7
x6
x5
...
x0

Memory

508
504
...
276
272
268
264
260
256
...
56
52
48
44
40
36
32
28
24
20
16
12
8
4
0

Part 2.D Program 4: Complex Number Multiplication

Next, we will compute the product of two complex numbers using the following formula:

$$(a+bi)(c+di) = ac + adi + bci + bdi^2 = ac - bd + (ad+bc)i$$

```
□□□ □□□ 00 addi x5,  x0,  256
□□□ □□□ 04 lw   x28, 0(x5)      # a
□□□ □□□ 08 lw   x29, 4(x5)      # b
□□□ □□□ 12 lw   x30, 8(x5)      # c
□□□ □□□ 16 lw   x31, 12(x5)     # d
□□□ □□□ 20 mul  x10, x28, x30   # ac
□□□ □□□ 24 mul  x6,  x29, x31   # bd
□□□ □□□ 28 addi x7,  x0,  -1
□□□ □□□ 32 mul  x6,  x6,  x7    # -bd
□□□ □□□ 36 add  x10, x10, x6    # ac-bd
□□□ □□□ 40 mul  x11, x29, x30   # bc
□□□ □□□ 44 mul  x6,  x28, x31   # ad
□□□ □□□ 48 add  x11, x11, x6    # ad+bc
□□□ □□□ 52 sw   x10, 16(x5)
□□□ □□□ 56 sw   x11, 20(x5)
```

Program Counter

Registers

x31
x30
x29
x28
...
x11
x10
...
x7
x6
x5
...
x0

Memory

508	
...	
280	
276	
272	
268	2
264	6
260	9
256	4
...	
56	
52	
48	
44	
40	
36	
32	
28	
24	
20	
16	
12	
8	
4	
0	

Part 2.E Program 5: Vector-Vector-Mult

In this assignment, we are executing two different implementations of element-wise vector multiplication. In this operation, each element of vector **A** is multiplied with the corresponding element of vector **B** and stored in the corresponding element of vector **C**. First, we will execute an implementation with a loop. Then, we will execute an implementation in which the loop is unrolled. We will compare both implementations.

$$C[i] = A[i] \cdot B[i]$$

2.E.1 Loop Implementation

```
00 addi x5,  x0,  3    # array len
04 addi x6,  x0,  256  # ptr A
08 addi x7,  x0,  268  # ptr B
12 addi x28, x0,  280  # ptr C
16 lw   x29, 0(x6)     # get elmnt a <--|
20 lw   x30, 0(x7)     #       and b    |
24 mul  x29, x29, x30  # c=a*b          |
28 sw   x29, 0(x28)    # store elmnt c  |
32 addi x5,  x5,  -1   #                |
36 addi x6,  x6,  4    # incr ptrs      |
40 addi x7,  x7,  4    #                |
44 addi x28, x28, 4    #                |
48 bne  x5,  x0,  16   # loop ----------|
52 addi x0,  x0,  0
```

Program Counter

Registers

x31
x30
x29
x28
...
x7
x6
x5
...
x1
x0

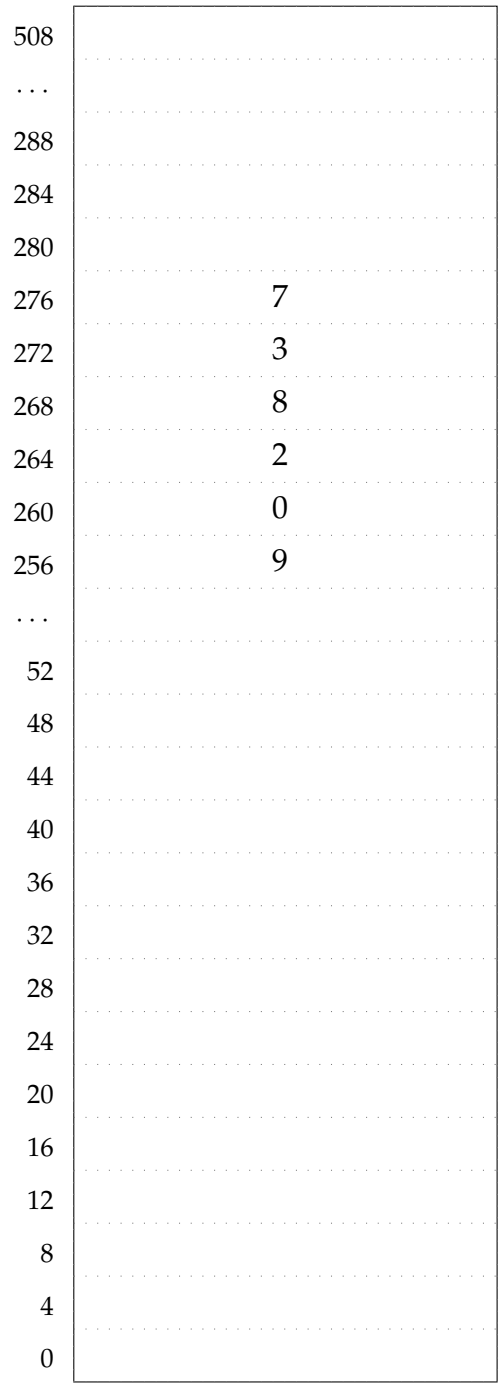
Memory

508	
...	
288	
284	
280	
276	7
272	3
268	8
264	2
260	0
256	9
...	
52	
48	
44	
40	
36	
32	
28	
24	
20	
16	
12	
8	
4	
0	

2.E.2 Unrolled

```
□□□ □□□ 00 addi x5,  x0,  256  # ptr A
□□□ □□□ 04 addi x6,  x0,  268  # ptr B
□□□ □□□ 08 addi x7,  x0,  280  # ptr C
□□□ □□□ 12 lw   x28, 0(x5)
□□□ □□□ 16 lw   x29, 0(x6)
□□□ □□□ 20 mul  x28, x28, x29  # C[0]=A[0]*B[0]
□□□ □□□ 24 sw   x28, 0(x7)
□□□ □□□ 28 lw   x28, 4(x5)
□□□ □□□ 32 lw   x29, 4(x6)
□□□ □□□ 36 mul  x28, x28, x29  # C[1]=A[1]*B[1]
□□□ □□□ 40 sw   x28, 4(x7)
□□□ □□□ 44 lw   x28, 8(x5)
□□□ □□□ 48 lw   x29, 8(x6)
□□□ □□□ 52 mul  x28, x28, x29  # C[2]=A[2]*B[2]
□□□ □□□ 56 sw   x28, 8(x7)
□□□ □□□ 60 addi x0,  x0,  0
```

Memory



Program Counter

Registers

2.E.3 Comparative Analysis

Compare both vector-vector-add implementations in terms of dynamic instruction count, instruction memory requirements, and register requirements.

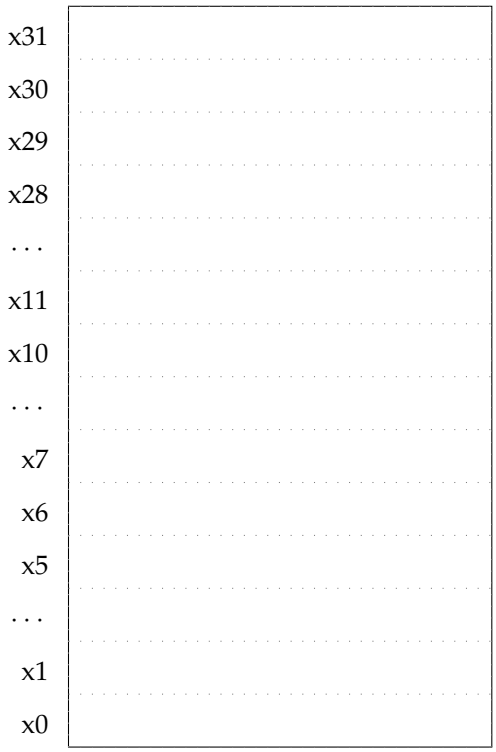
Part 2.F Program 6: Fibonacci Sequence

In this assignment, we will evaluate an implementation for computing the *Fibonacci Sequence* (utilizing the formula below). The implementation contains a loop. In each iteration, it computes the next element by jumping to the *fibcomp* function, performing the computation, and then jumping back into the loop.

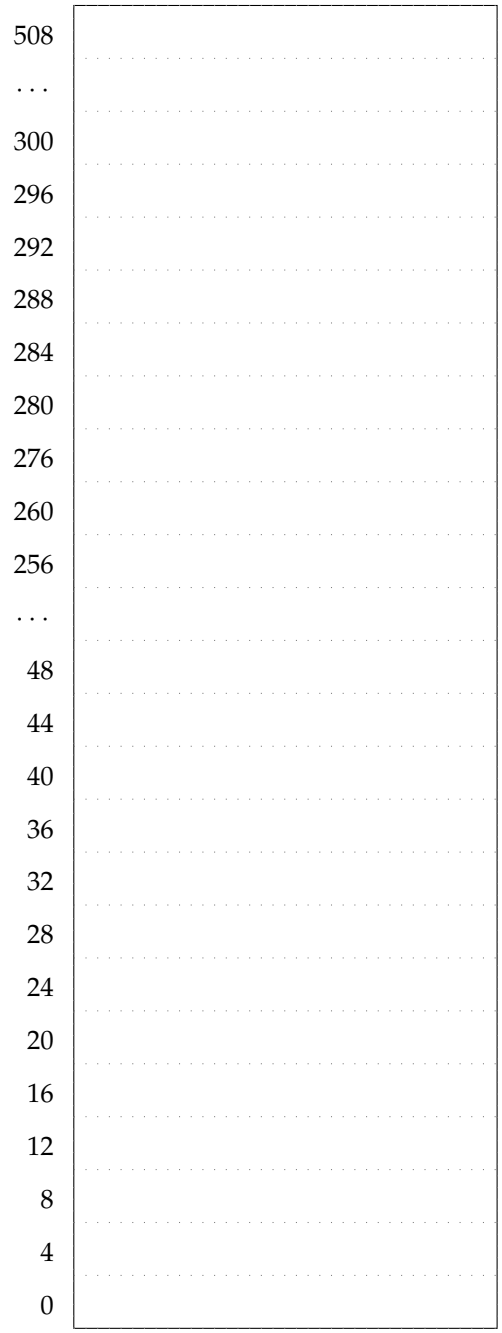
$$F_n = F_{n-1} + F_{n-2}$$

```
  jal  x0,  start
fibcomp:
  add  x7,  x5,  x6    # Fn=(Fn-1)+(Fn-2)
  addi x6,  x5,  0     # store Fn-1 in x6
  addi x5,  x7,  0     # store Fn in x5
  jr   x1
start:
  addi x10, x0,  4     # len
  addi x5,  x0,  1     # set initial Fn-1
  addi x6,  x0,  0     #         and Fn-2
loop:                  # <-------|
  jal  x1,  fibcomp    #         |
  addi x10, x10, -1    #         |
  bne  x10, x0,  loop  # loop ---|
  sw   x5,  256(x0)
  addi x0,  x0,  0
```

Registers



Memory



Rachel's Solution

`https://vod.video.cornell.edu/id/1_0vsflxls`

Part 2.G Program 7: Code Injection Attack

Program 7 is the firmware of an embedded device. The main function calls the `get_secret` function, which contains *secure* code. This function returns highly sensitive data (42); however, it only does so when the provided key (passed in register `x10`) equals 7. If the key is incorrect, a conditional branch skips the sensitive code and returns zero.

We are security analysts hired by the device developer. The engineering manager confidently claims they have developed an unbreakable device. But have they? Upon closer inspection, we notice legacy code early in the main function that reads an address and a value from memory-mapped I/O, then writes the value to that address. **Can we exploit this to access the sensitive information?**

Hint: The decimal value 19 (hex `0x13`) is the TinyRV1 encoding of the `nop` instruction (`addi x0, x0, 0`). How would we need to configure the external I/O `in0` at address 512 to overwrite the `bne` instruction in `get_secret`?

Configure `in0` for the code injection attack and execute the program on this worksheet.

```
 jal  x0, main
get_secret:
 addi x5,  x0,  7         # correct key
 addi x6,  x0,  0
 bne  x5,  x10, jump_ret  # <- key check
 addi x6,  x0,  42        # SENSITIVE DATA
jump_ret:
 addi x10, x6,  0
 jr   x1
main:
# legacy code
 lw   x5,  512(x0)        # addr
 lw   x6,  516(x0)        # value
 sw   x6,  0(x5)
# ...
 addi x10, x0,  0         # false key arg
 jal  x1,  get_secret     # get secret?
 addi x5,  x10, 0         # use secret?
```

External I/O

516	19: `addi x0, x0, 0`
512	?

Memory

508	
...	
48	
44	
40	
36	
32	
28	
24	
20	
16	
12	
8	
4	
0	

Registers

...	
x10	
...	
x6	
x5	
...	
x1	
x0	

Never Trust User Input

What we just performed was a code injection attack. We were able to inject code (in this case, a single instruction) to modify the application's behavior. This exercise also illustrates an important security principle: never trust user input. A user might not always be a benign actor who respects your system's boundaries.
This principle applies broadly across software development. For example, a common security vulnerability in web applications is SQL injection, where applications pass user input directly to a database without validation. An attacker could craft a malicious request to extract, modify, or even delete entire databases. The lesson: always validate user input and verify that requests are legitimate. In our case, the legacy code should have verified that the store address lies within a valid data region, not the code region.

Niklas's Solution

`https://vod.video.cornell.edu/id/1_fbefw5hb`