CS211 Computer Architecture II

ISAs, Microprogramming and ROP
Homework #1

Assigned 14/09/2020 *Due 23:59:59 28/09/2020*

https://toast-lab.gitee.io/courses/CS211@ShanghaiTech/Fall-2020/

The homework is intended to help you learn the material, and we encourage you to collaborate with other students and to ask questions in discussion sections and office hours to understand the problems. However, each student must turn in their own solution to the problems.

The homework also provides essential background material for the mid-term and final exams. It will be graded primarily on an effort basis, but if you do not work through it, you are unlikely to succeed on the mid-term and final exams! We will distribute solutions to homework assignment. Note that each homework assignment is due at its respective due time, and all assignments are to be submitted **in English**. However, late submissions will **NOT** be accepted, except for extreme circumstances and/or with prior arrangement.

Name: _____

ID: _____

Problem 1: CISC, RISC, and Stack: Comparing ISAs (2.5 points)

In this problem, your task is to compare three different ISAs: x86 (a CISC architecture with variable-length instructions), RISC-V (a load-store, RISC architecture with 32-bit instructions in its base form), and a stack-based ISA.

Problem 1.A CISC (0.5 point)

Let us begin by considering the following C code:

```
int modulo(int x, int y) {
        for (; x >= y; x -= y);
        return x;
}
```

Using gcc and objdump on an x86 machine, we see that the above loop compiles to the following x86 instruction sequence. On entry to this code, register `%edx` contains `x`, and register `%eax` contains `y`. Throughout parts (**1.A--1.C**), we will ignore what happens in the `done` label and return statement.

```
start:
        cmp     %eax, %edx
        jl       done
loop:
        sub     %eax, %edx
        cmp     %edx, %eax
        jle     loop
done:
        ret
```

The meanings and instruction lengths of the instructions used above are given in the following table. Registers are denoted with $R_{SUBSCRIPT}$, register contents with $<R_{SUBSCRIPT}>$.

Instruction	Operation	Length
sub R_{SRC}, R_{DEST}	$<R_{DEST}>$ - $<R_{SRC}>$ ➔ R_{DEST}	2 bytes
cmp R_{SRC1}, R_{SRC2}	$<R_{SRC2}>$ - $<R_{SRC1}>$ ➔ Temp	6 bytes
jl label	if (SF != OF) jump to the address specified by label	2 bytes
jle label	if ((SF != OF) **or** (ZF == 1)) jump to the address specified by label	2 bytes
ret	return to the caller function	1 byte

Notice that the jump instructions `jl` and `jle` (jump if less, and jump if less or equal) depend on status flags, i.e., `SF`, `ZF` and `OF` in the table. Status flags are set by the instruction preceding a jump, based on the result of the computation. The meanings of the status flags are given in the following table:

Name	Purpose	Condition Reported
ZF	Zero flag	Result is zero (cmp performs an actual subtraction)
SF	Sign flag	Set equal to high-order bit of result (0 if positive 1 if negative)
OF	Overflow flag	Set if result is too large a positive number or too small a negative number (excluding sign bit) to fit in destination operand; cleared otherwise.

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if x = 13 and y = 3? Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?

Problem 1.B RISC (1 points)

Translate each of the x86 instructions in the following table into one or more RISC-V instructions. Place the loop label where appropriate. You should use the minimum number of instructions needed to translate each x86 instruction. Assume that upon entry, x1 contains x and x2 contains y. If needed, use x3 as a condition register, and x4, x5, etc., for temporaries. You should not need to use any floating-point registers or instructions in your code. A description of the RISC-V instruction set architecture can be found in the class website alongside L03.

x86 instruction	label	RISC-V instruction sequence
	start:	
cmp %eax, %edx		
jl done		
sub %eax, %edx		
cmp %edx, %eax		
jle loop		
	done:	
ret		

How many bytes is the RISC-V program using your direct translation? How many bytes of RISC-V instructions need to be fetched for x = 13 and y = 3 using your direct translation? Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored?

Problem 1.C **Stack (1 points)**

In a stack architecture, all operations occur on top of the stack. Only push and pop access memory, and all other instructions remove their operands from the stack and replace them with the result. The hardware implementation we will assume for this problem uses stack registers for the top two entries; accesses that involve other stack positions (e.g., pushing or popping something when the stack has more than two entries) use an extra memory reference. Assume each instruction occupies three bytes if it takes an address or label; other instructions occupy one byte.

Instruction	**Definition**
PUSH A	load value at M[A]; push value onto stack
POP A	pop stack; store value to M[A]
ADD	pop two values from the stack; ADD them; push result onto stack
SUB	pop two values from the stack; SUBtract top value from the 2nd; push result onto stack
INC	pop value from top of stack; increment value by one; push result onto stack
ZERO	zero out value at top of stack
DEC	pop value from top of stack; decrement value by one; push result onto stack
BEQZ *label*	pop value from stack; if it's zero, continue at *label*; else, continue with next instruction
BNEZ *label*	pop value from stack; if it's not zero, continue at *label*; else, continue with next instruction
BNEG *label*	pop value from stack; if it's negative (less than zero), continue at *label*; else, continue with next instruction
JUMP *label*	continue execution at location *label*

Translate the `modulo` loop to the stack ISA. For uniformity, please use the same control flow as in parts (**1.A**) and (**1.B**). Assume that when we reach the loop, `x` and `y` are at the top of the stack. At the end of the loop, `x` should be at the top of the stack.

How many bytes is the stack program using your translation? How many bytes of instructions need to be fetched for `x = 13` and `y` = 3 using your translation? Assuming 32-bit data values, how many bytes of data memory need to be loaded? Stored? Would the number of bytes loaded and stored change if the stack could fit 8 entries in registers?

Problem 2: ROP (0.5 point)

In this problem, we explore how to explore and leverage code snippets to lanuch a return-oritend programming (ROP) attack. In L03, we have studied that code snippets ended with `ret`, also known as gadgets, can be chained to achieve some purpose.

Problem 2.A Modular multiplication (0.5 point)

Many cryptographic algorithms, such as RSA and Diffie-Hellman key exchange, are based on modular multiplication, e.g., $A * B \bmod N$. In this problem, we would consider using ROP to make a modular multiplication. Some code snippets that might be useful for you are listed below.

```
/* For instructions with two operands, the left operand is RSRC
(source register) or immediate value, and the right one is RDEST (destination
register). */

start:                              /* address: 0x8000 */
      cmp    %eax, %edx
      jl      done
loop:                              /* address: 0x8008 */
      sub    %eax, %edx
      cmp    %edx, %eax
      jle    loop
done:                              /* address: 0x8011 */
      ret
…

G0:                                 /* address: 0x8080 */
      xor     %ebx, %ebx           /* <RSRC> xor <RDEST> ➔ RDEST */
      mov     $0x08, %eax          /* imm ➔ RDEST */
      ret
…

G1:                                /* address: 0x9090 */
      add    %ecx, %ebx            /* <RSRC> + <RDEST> ➔ RDEST */
      ret
…

G2:                                /* address: 0xA0A0 */
      mov    %ebx, %edx            /* <RSRC> ➔ <RDEST> */
      mov    %ecx, %eax
      sub    %eax, %edx            /* <RDEST> - <RSRC> ➔ RDEST */
      ret
…

G3:                                 /* address: 0xB0B0 */
      dec    %ebx                  /* <RDEST> - 1 ➔ <RDEST> */
      ret
…
```

```
G4:                                  /* address: 0xC0C0 */
      pop   %ecx                     /* pop stack; store the value to R_DEST */
      ret
…

G5:                              /* address: 0xD0D0 */
      shl    $0x08, %ebx         /* shift bits of <R_DEST> to the left for 0x08 times ➔ <R_DEST> */
      ret
…

G6:                              /* address: 0xE0E0 */
      and     $0x07, %ecx        /* <R_DEST> bitwise_and 0x07 ➔ <R_DEST> */
      and     $0x07, %edx
      ret
…

G7:                              /* address: 0xF0F0 */
      mul    %ebx, %ecx          /* <R_SRC> * <R_DEST> ➔ R_DEST */
      mov    %edx, %ebx
      ret
…
```

Using all or some of these code snippets, could you build a ROP case to calculate

(1689 * 255 % 8)

and store the result into `ebx` register?

If yes, please illustrate one case. Otherwise, give your reason. You can draw a diagram for illustration with descriptions. When you draw the memory space, you can use a simplified layout that is similar to ones shown in the lecture slides.

Hint: You may find none, one, or multiple cases. Bonus points may be accredited to students that submit reasonable and interesting answer.