Spring 2024
Exam II
2024-04-05

Instructions

This exam contains 12 pages (including this cover page) and 22 questions. It is out of 76 points.

You have **50 minutes** to complete the examination. As a courtesy to your classmates, we ask that you not leave during the last 15 minutes.

For this exam, you have been given a separate answer sheet to fill in your responses.

DO shade in the bubbles on your answer sheet without going outside the lines

DO feel free to write on this exam packet

DO assume all multiple-choice questions on this test are single-select **unless otherwise indicated**

DO NOT write anything on your answer sheet except for your name, computing ID, signature, and answers in the designated areas

DO NOT use a calculator, consult notes, or collaborate with classmates

We will use the following data type sizes:

x86-64 Suffix	C Types	size in bits
`b`	`char`	8
`w`	`short`	16
`l`	`int` and `float`	32
`q`	`long` and `double`	64

Function arguments are in (in order) `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`; return values are in `%rax`.

The next page contains reference material which you are welcome to refer to during the test if you would like.

Our Example ISA

This is the same ISA used in HW03 and HW04, but presented to fit onto one printed page.

Each instruction is one or two bytes, with the meaning of those bytes being:

7	6	5	4	3	2	1	0
`0`	`icode`			`a`		`b`	

byte at `pc`

7	6	5	4	3	2	1	0
immediate							

byte at `pc + 1`

Not all instructions have the second byte; those that do describe it below as the byte "at `pc + 1`".

In the table below `rA` means "the value stored in register number `a`" and `rB` means "the value stored in register number `b`."

`icode`	`b`	Behavior	add to `pc`
0		`rA = rB`	1
1		`rA += rB`	1
2		`rA &= rB`	1
3		`rA` = read from memory at address `rB`	1
4		write `rA` to memory at address `rB`	1
5	0	`rA = ~rA`	1
5	1	`rA = -rA`	1
5	2	`rA = !rA`	1
5	3	`rA = pc`	1
6	0	`rA` = read from memory at `pc + 1`	2
6	1	`rA` += read from memory at `pc + 1`	2
6	2	`rA` &= read from memory at `pc + 1`	2
6	3	`rA` = read from memory at the address stored at `pc + 1`	2
7		if `rA <= 0`, set `pc = rB`	N/A
7		if `rA > 0`, do nothing	1
0	0	Decrement `rsp` and push the contents of `rA` on the stack	1
0	1	Pop the top value from the stack into `rA` and increment `rsp`	1
0	2	Push `pc + 2` onto the stack, set `pc = M[pc+1]`	2
0	3	`pc` = pop the top value from the stack	

Note the stack operations have the reserve bit set to 1. They are just not depicted here.

1 Toy ISA Stack Operations

1. (4 points) **You may assume that only the steps listed below affect register values.**

   1. pushes values `r0`, `r1`, and `r2` (in order) on the stack
   2. pops into `r1`
   3. pushes `r3` on the stack
   4. calls the function foobar, whose code is located in memory from `0x90` to `0xA2`
   5. returns from the function
   6. pops into `r3`

   If the code starts with `rsp` = `0xE3` and `pc` = `0x10`, what are the values of `rsp` and `pc` when the code reaches the first instruction in the foobar function? **Write you answer in the blanks space corresponding to question 1 on the scantron.**

   Solution:

   `rsp` = `0xDF`

   `pc` = `0x90`

2. (2 points) If `pc` = `0x1D` after the program returns from the foobar function, what was the value of `pc` when the function was called? **Write you answer in the blanks space corresponding to question 2 on the scantron.**

   Solution: `pc` = `0x1B`

3. (4 points) If the code starts with `pc` = `0x10` and halts when `pc = 0x1E`, which of the following starting values of `rsp` would likely cause the program to behave incorrectly? **Select all that apply.**

   (A) `0x00`
   (B) `0x11`
   (C) `0x21`
   (D) `0x90`
   (E) `0x91`
   (F) `0xA3`
   (G) `0xA5`
   (H) `0xFF`

   Solution: (C) `0x21`, (E) `0x91`, (F) `0xA3` and (G) `0xA5`

   Whether (B) `0x11` was selected does not affect the credit awarded.

2 The X86 Stack (and some AT&T Syntax Assembly)

4. (4 points) Imagine that a function pushes 50 `char`s to the stack. Which of the following could be done to reset the stack? **Select all that apply.**

(A) `addq $49, %rsp`

(B) `addq $50, %rsp`

(C) `subq $50, %rsp`

(D) Execute `popq %rax` 50 times

(E) Execute `popw %ax` 25 times

(F) `xorq %rsp, %rsp`

Solution:

(B) `addq $50, %rsp`

(E) Execute `popw %rax` 25 tines

5. (2 points) If a function takes in 8 arguments, how many arguments will be passed on the stack?

(A) 0 (B) 1 (C) 2 (D) 4 (E) 6 (F) 8

Solution: (C) 2

The questions on this page all consider the following x86 Assembly function:

```
 1   stack_function:
 2       pushq %rbp
 3       movq %rsp, %rbp
 4       xorq %rcx, %rcx
 5   loop:
 6       cmpq $50, %rcx
 7       jge end
 8       pushq %rcx
 9       addq $1, %rcx
10       jmp loop
11   end:
12       movq (%rsp), %rax
13       # INSERT CODE HERE #
14       pop %rbp
15       retq
```

6. (2 points) How many values have been pushed onto the stack once the code reaches the `end:` label on line #11?

(A) 0	(C) 49	(E) 51
(B) 1	(D) 50	(F) 52

Solution: (E) 51

7. (4 points) As written, `stack_function` will crash with a segmentation fault instead of returning to the function which called it. Either of the 2 incomplete lines of Assembly below could be inserted on line #13 to avoid a crash.

   Define **both** possible solutions to avoiding a crash by filling in the blanks below with register names or immediates.

   Solution Blank 1 (B1) (*complete both of these*): `movq %`**rbp**`, %rsp`

   Solution Blank 2 (B2) (*complete both of these*): `addq $`**400/0x190**`, %rsp`

8. (6 points) Imagine we reworked `stack_function` to take in an input instead of hard-coding the `$50` compare value on line #6. Order the instructions below to

   - call `stack_function` with an argument of 16 and
   - set `rdx` equal to twice the value returned by `stack_function`.

   Clearly write the line numbers in the boxes on your answer sheet. Leave any unnecessary boxes blank. Note: You only need to use a subset of the instructions below.

1. `addq %rax, %rdx`
2. `addq %rdx, %rax`
3. `callq stack_function`
4. `movq $16, %rax`
5. `movq $16, %rdi`
6. `movq %rax, $16`
7. `movq %rax, %rdx`
8. `movq %rdx, $16`
9. `movq %rdx, %rax`

Solution:

5	3	7	1				

3 Assembly

9. (4 points) Which of the instructions below could change the flags? Select all that apply

   (A) `leaq (%rax), %rax`

   (B) `je loop`

   (C) `movq $5, %rax`

   (D) `xor $rsp, $rsp`

   (E) `addq %rcx, %rdx`

   (F) `pushq $2130`

   (G) `test $1, %rsi`

   (H) `ret`

Solution:

(D) `xor $rsp, $rsp`

(E) `addq %rcx, %rdx`

(G) `test $1, %rsi`

10. (4 points) If we are working on a 32 bit machine, which flag(s) would be set to 1 after executing `add $0xFF000000, $0x11000000`? Select all that apply. Leave the answer blank if no flags are set.

    (A) Carry flag (B) Zero flag (C) Sign flag (D) Overflow flag

Solution: (A) Carry flag

11. (5 points) Consider the following C program, as well as an incomplete Assembly version of the same program:

```
0   int main() {
1       int x = 1;
2       while (x < 7) {
3           x = 2x + 1;
4       }
5       return x;
6   }
```

```
0   main:
1        movq $1, %rax
2   loop:
3       # INSERT CODE HERE #
4       # INSERT CODE HERE #
5       addq %rax, %rax
6       addq $1, %rax
7       jmp loop
8    end:
9       retq
```

Complete the following Assembly lines to make the two programs match.

Line 3: `cmpq (B1)` **\$6** , `(B2)` **%rax**

Line 4: `jg` (B3) **end**

Assume the first 8 registers and the given segment of (little-endian) memory have the following initial values:

Register	Value (hex)
rax	0x70000A
rcx	0x3
rdx	0x86
rbx	0x100
rsp	0x700008
rbp	0x14
rsi	0xA
rdi	0x70000B

Memory Address	Value (hex)
0x700000	0x00
0x700001	0x04
0x700002	0x08
0x700003	0x0C
0x700004	0x10
0x700005	0x14
0x700006	0x18
0x700007	0x1C

Memory Address	Value (hex)
0x700008	0x00
0x700009	0x31
0x70000A	0x41
0x70000B	0x59
0x70000C	0x26
0x70000D	0x12
0x70000E	0x02
0x70000F	0x05

Determine whether each instruction changes a register and, if so, fill in the name of the changed register and its new value.

Each instruction below is independent; do not use the result of one as the input for the next.

12. (2 points) `movl (%rdi), %rax`

    (A) No change

    (B) Register: %**rax** (B1)
    New value: **0x02122659** (B2)

13. (2 points) `movq %rdi, (%rax)`

    (A) No change ✓

    (B) Register: %__________ (B1)
    New value: __________ (B2)

14. (2 points) `leaq -0xA(%rdi), %rsi`

    (A) No change

    (B) Register: %**rsi** (B1)
    New value: **0x700001** (B2)

15. (2 points) `movw 0x3(%rax), %dx`

    (A) No change

    (B) Register: %**(r)dx** (B1)
    New value: **0x212** (B2)

16. (2 points) `movb (%rsp), %bl`

    (A) No change ✓

    (B) Register: %__________ (B1)
    New value: __________ (B2)

17. (2 points) `retq`

    (A) No change

    (B) Register: %**rsp** (B1)
    New value: **0x700010** (B2)

4 Pointers

The questions below consider the following 2d array, where `&row_powers[0][0]` is `0xE0` and the array is stored in row major order.

```
int row_powers[3][4] = { {1, 2, 4, 8}, {1, 3, 9, 27}, {1, 4, 16, 64} };
```

18. (2 points) What is the output of `row_powers[1][2]`

(A) 1 (B) 2 (C) 3 (D) 4 (E) 8 (F) 9 (G) Segmentation fault (H) Random value

Solution: (F) 9

19. (2 points) What is the output of `*(row_powers + 6)`?

(A) 1 (B) 2 (C) 3 (D) 4 (E) 8 (F) 9 (G) Segmentation fault (H) Random value

Test program would look like this. This could have been clearer in the question. So we decided to drop the question.

```
#include <stdio.h>

int main(){
   int row_powers[3][4] = { {1, 2, 4, 8}, {1, 3, 9, 27}, {1, 4, 16, 64} };
    int * row_powers_p = row_power;
    printf("%d \n",*(row_powers_p + 6));
}
```

Solution: //Question dropped.

(F) 9

Questions assuming that we treating `row_powers` though the array has been decade to a pointer

20. (4 points) For the following descriptions, select the letter of the corresponding code snippet.

	`int *p;`	`*p =`	`= *p`	`= &p`
Go to the address stored in the pointer and retrieve the value	(A)	(B)	(C)	(D)
Declare a pointer	(A)	(B)	(C)	(D)
Get the address of a variable	(A)	(B)	(C)	(D)
Go to the address stored in the pointer and update the value	(A)	(B)	(C)	(D)

Solution: (C), (A), (D), (B)

21. (5 points) What are the following values of i, j, *k, l, and &l after the following code is executed?

```
long i = 3;  //i is at memory address 0x2
long j = 1;  //j is at memory address 0xC
long *k = &i; //*k is at memory address 0x10
long **l = &k; //**l is at memory address 0x18
*k = 101;
**l = 7;
long *m = &j; //*m is at memory address 0xF0
*m = (long) (*l + 8);
```

i: B1 **j:** B2 ***k:** B3 **l:** B4 **&l:** 0x18

Solution:

i: 7 **j:** 66 or 0x42 ***k:** 7 **l:** 0x10 **&l:** 0x18

5 C (and more Assembly)

22. (10 points) The following C and x86 Assembly code implement `next_state_function`, which accepts some `current_state` and returns a `next_state`. Assuming standard x86 calling conventions, fill in all the blanks to complete both implementations. Exclude `%` and `$` prefixes.

C function

```
int next_state_function(int current_state) {
    int next_state = /* BLANK 1 (B1)*/;
    switch(current_state) {
        case 0:
            next_state = 1;
            break;
        case 1:
        case 2:
            next_state = current_state + 1;
            break;
        case 3:
            next_state = 0;
            break;
        default:
            next_state = /* BLANK 2 (B2)*/;
            break;
    }
    return next_state;
}
```

x86 Assembly function

```
    next_state_function:
        movq $-1, %rax
        cmpq $# BLANK 3 (B3) #, %rdi
        ja .L4
        jmp *.L1(,%rdi,8)
    .L2:
        movq  1, %rax
        retq
    .L9:
        addq $# BLANK 4 (B4) #, %# BLANK 5 (B5)#
        movq %# BLANK 6 (B6) #, %# BLANK 7 (B7)#
        retq
    .L5:
        movq $0, %# BLANK 8 (B8)#
        retq
    .L4:
        movq %rdi, %rax
        retq
```

Elsewhere in the same Assembly file

```
.section    .rodata
    .align 8
.L1:
    .quad   .L2
    .quad   .L9
    .quad   .L9
    .quad   .L5
```

Solution:

B1: `-1`

B2: `current_state`

B3: `3`

B4: `1`

B5: `rdi`

B6: `rdi`

B7: `rax`

B8: `rax`