CMPT 295

Name: Junchen Li
Student ID: 301385486

Date: 2020/3/18

Assignment 7

Objectives:

- Another look at recursion in x86-64 assembly code
- Designing and evaluating instruction sets (ISA)

Submission:

- Submit your document called **Assignment_7.pdf**, which must include your answers to all of the questions in Assignment 7.
    - Add your full name and student number at the top of the first page of your document **Assignment_7.pdf**.
- When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
- **If you hand-write your answers (as opposed to using a computer application to write them):** When putting your assignment together, do not take photos (no .jpg) of your assignment sheets! Scan them instead! Better quality -> easier to read -> easier to mark!
- Submit your assignment **Assignment_7.pdf** electronically on CourSys.

Due:

- Thursday, March 19 at 3pm.
- Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.

Marking scheme:

This assignment will be marked as follows:

- All questions will be marked for correctness.

The amount of marks for each question is indicated as part of the question.

A solution will be posted after the due date.

---

1. [7 marks] Another look at recursion in x86-64 assembly code

Consider the following recursive implementation of the function factorial `fact` written in x86-64 implementation:

```
1   # fact(n): Buggy version
2     .globl fact
3   # n in edi
4   fact:
5     cmpl $1, %edi
6     jg endif
7     movq $1, %rax
8     ret
9   endif:
10    decl %edi
11    pushq %rdi
12    call fact
13    imulq (%rsp), %rax
14    leaq 8(%rsp), %rsp
15    popq %rdi
16    ret
```

Refresher:

Mathematically, the formula for a factorial is as follows.

If n is an integer greater than or equal to 1, then

$$n! = n(n-1)(n-2)(n-3)\ldots(3)(2)(1)$$

If $n = 0$, then $n! = 1$.

Download the files `main.c`, `fact.s` and `makefile`. When we compile and execute the above function `fact`, we get a `segmentation fault` error. Verify that this is indeed the case.

Your task is to debug the function so that it produces the expected results.

You are to do so by hand tracing it with the test case n= 4. As you are hand tracing the function, you may want to draw its stack diagram and its register table as this may help you visualize what is happening with the code. You do not have to hand in your stack diagram/register table.

带格式的: 字体:12 pt
带格式的: 列出段落, 两端对齐, 定义网格后自动调整右缩进, 已编号 + 级别: 1 + 编号样式: 1, 2, 3, … + 起始于: 1 + 对齐方式: 左 + 对齐位置: 0 cm + 缩进位置: 0.63 cm, 对齐网格
删除的内容:
带格式的: 字体:12 pt
带格式的: 正文, 两端对齐, 定义网格后自动调整右缩进, 对齐网格

CMPT 295 – Spring 2020

Once you have figured out where the problem(s) is/are located, fix the above code by doing the least number of fixes as possible. This means that you cannot rewrite the entire function.

Writing your working version below:

```
1   # fact(n): Working version
2     .globl fact
3   # n in edi
4   fact:
        cmpl    $1, %edi        // if n<=1
        jg      endif           // if it is wrong then go to the ret
        movq    $1, %rax        // move one into the return value
        ret

ABOVE part is talking about base case.

endif:
        pushq   %rdi            // push a callee saved into stack to store value for n.
        decl    %edi            // let n++ for the next call parameter
        call    fact            // for the next recursive loop
        imulq   (%rsp), %rax    // n *fact (n-1)
        popq    %rdi            // pop the stack

ABOVE part is talking about recursive case.

        ret
```

Finally, comment your working version above. In your comments, make sure you indicate the **base case** and the **recursive case.**

[13 marks] Designing and evaluating instruction sets (ISA)

Note: What you need to do in this question is highlighted in this blue colour.

Instruction Set 1 – x295

Description of ISA

During our lectures 22, 23 and 24, we specified an instruction set architecture (ISA) called **x295**, with the following components:

- Memory model of the computer
  - Size of external memory (RAM): $2^{12} \times 16$
  - Memory address: 12 bits
  - Word size: 16 bits
  - Number of registers: 0
- Instruction set
  - Maximum number of instructions: 16
  - Opcode size: 4 bits ($2^4 = 16$)
  - Operand Model:
    - Memory (only) – only memory locations are operands, holding values (no registers are used as operands)
    - 3 operands
    - Operand order: Dest, Src1, Src2
  - Memory addressing mode: Direct
  - Instructions (so far):
    - `ADD a,b,c` Meaning: `M[c] <- M[a] + M[b]`
    - `SUB a,b,c` Meaning: `M[c] <- M[a] - M[b]`
    - `MUL a,b,c` Meaning: `M[c] <- M[a] * M[b]`
  - Template:

opcode	Dest (12 bits)	XXXX	Src1 (12 bits)	XXXX	Src2 (12 bits)

This template can be used to form all three instructions.

- Data size : 16 bit

带格式的: 字体:12 pt
删除的内容: Manipulating floating point values
删除的内容:
带格式的: 字体:12 pt, 字体颜色: 蓝色
带格式的: 列出段落, 两端对齐, 缩进: 左: 0.63 cm, 定义网格后自动调整右缩进, 对齐网格

Evauation of ISA

We wrote the following C program using the instructions in this **x295** instruction set:

C program	x295 program

`z = (x + y) * (x - y)`	ADD x, y, tmp1
	SUB x, y, tmp2
	MUL tmp1, tmp2, z
	(where tmp1 and tmp2 are memory addresses holding temporary results)

1. In order to evaluate our instruction set, we used the metric called memory traffic, i.e., we counted the number of memory accesses our program (written in x295) made during its execution. In other words, we counted how many time the execution of our program required a word (16 bits) to be read from or written to memory.

   Note that we first did ascertain the fact that we were able to completely write the above C program using the instructions found in our **x295** instruction set and hand tracing it allowed to ascertain that it would produce the expected result (e.g., test case: x = 3, y = 2, expected result: 5).

The table below show the results of our evaluation:

Instructions of x295 program	**Fetch**	**Execute**	
ADD x, y, tmp1	3 since the binary encoding of the ADD instruction is 3-word wide	2 since the ADD instruction requires the value of two operands read from memory	1 since the ADD instruction writes its result to memory
SUB x, y, tmp2	3 since the binary encoding of the SUB instruction is 3-word wide	2 since the SUB instruction requires the value of two operands read from memory	1 since the SUB instruction writes its result to memory
MUL tmp1, tmp2, z	3 since the binary encoding of the MUL instruction is 3-word wide	2 since the MUL instruction requires the value of two operands read from memory	1 since the MUL instruction writes its result to memory
Grand Total: 18	Total: 9	Total: 6	+ 3

2. We also evaluated our instruction set using the metric called static code size:
   - The code size of our x295 program is 3 instructions
   - Since each instruction is 3-word long, the code size of our x295 program is also 9 words.

CMPT 295 – Spring 2020

Can we do better? Can we design an ISA that reduces the number of memory accesses.

Strategy 1 is to introduce registers.

Instruction Set 2 – x295+

Description of ISA

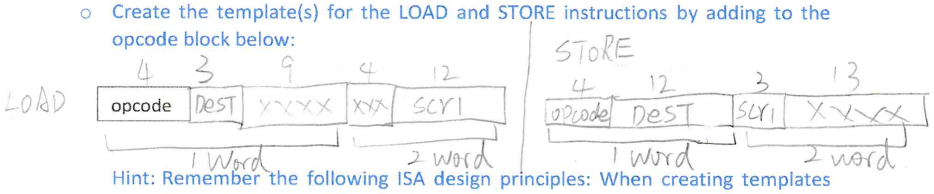
With the idea of reducing the number of memory accesses, we specified a second instruction set architecture (ISA) with the same components as x295, but with the addition of:

- Memory model of the computer
  - Number of registers: 8 x 16-bit registers
- Instruction set
  - Operand Model:
    - Registers
  - Instructions:
    - `ADD rA,rB,rC` Meaning: `rC <- rA + rB`
    - `SUB rA,rB,rC` Meaning: `rC <- rA - rB`
    - `MUL rA,rB,rC` Meaning: `rC <- rA * rB`
    - `COPY rA,rC` Meaning: `rC <- rA`
    - `LOAD a,rC` Meaning: `rC <- M[a]`
    - `STORE rA,c` Meaning: `M[c] <- rA`
  - Template 1:

opcode	Dest	Src1	Src2	XXX

This template can be used to form the instructions ADD, SUB, MUL and COPY.

- Create the template(s) for the LOAD and STORE instructions by adding to the opcode block below:



Hint: Remember the following ISA design principles: When creating templates (formats) to encode instruction set ...

- Create as few of them as possible. This will help simplify the design of the CPU.

CMPT 295 – Spring 2020

- Place the fields that have the same purpose (such as Opcode, Dest and Src) in the same location in as many of your templates as possible.

Evauation of ISA

Now, write the C program using the instructions in this new **x295+** instruction set:

C program	x295+ program

`z = (x + y) * (x - y)`	LOAD X, r0
	LOAD Y, r1
	ADD r0, r1, r2
	SUB r0, r1, r3
	MUL r2, r3, r6
	STORE r5, Z
	Put all value of parameters into registers and use them without going into memory everytime.

CMPT 295 – Spring 2020

1. Show the results of your evaluation (using the metric memory traffic, i.e., counting the number of memory accesses):

Write each line of your x295+ program	Fetch	Execute
LOAD x, r0 LOAD y, r1	4 3 9 4 12 2 4 3 4 4 12 2	1 1
ADD r0, r1, r2 SUB r0, r1, r3	4 3 3 3 xx 1 4 3 3 3 xxx 1	0 0
MVL r2, r3, r6 STORE r1, z	4 3 3 3 xxx 1 4 12 3 13 2	0 1
Grand Total:	Total: 9	Total: 3 = 12

2. How many instructions and words does the x295+ programs contain (static code size metric):

Answer. we uses six instructions and the static code size metric is 9 totally. (2*3+3*1)

9

CMPT 295 – Spring 2020

Can we do better? Can we design an ISA that reduces even more the number of memory accesses.

Strategy 2 is to reduce the number of operands used in an instruction.

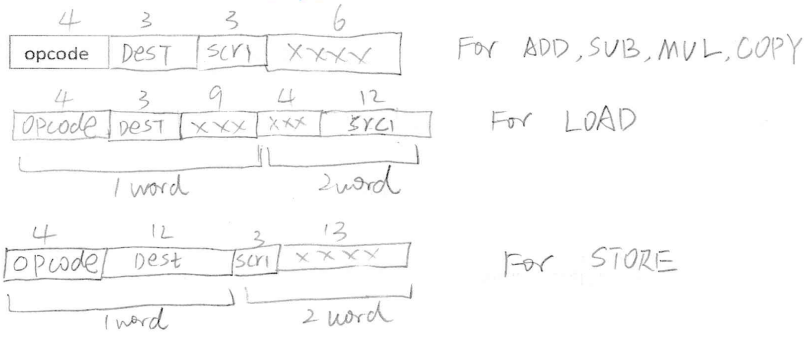
Instruction Set 2 – x295++

Description of ISA

What happens when we reduce the number of operands?

Let's specified a third instruction set architecture (ISA) with the same components as x295+ and x295, but with the following modifications:

- Instruction set
  - Operand Model:
    - 2 operands
  - Instructions:
    - `ADD rA,rC` Meaning: `rC <- rA + rC`
    - `SUB rA,rC` Meaning: `rC <- rA - rC`
    - `MUL rA,rC` Meaning: `rC <- rA * rC`
    - `COPY rA,rC` Meaning: `rC <- rA`
    - `LOAD a,rC` Meaning: `rC <- M[a]`
    - `STORE rA,c` Meaning: `M[c] <- rA`
  - Create the template(s) necessary to form all the instructions of x295++ ISA and indicate which template would be used to form which instruction(s). To do so, copy the opcode block below as many times as needed and add to it the rest of the fileds to create each template:



CMPT 295 – Spring 2020

Evaluation of ISA

Now, write the C program using the instructions in this new **x295++** instruction set:

C program	x295++ program

`z = (x + y) * (x - y)`	LOAD x, r0
	COPY r1, r0
	LOAD y, r2
	ADD r2, r0
	SUB r1, r2
	MUL r0, r2
	STORE r2, z
	
	// r0 = x+y
	// r1 = r0 = x
	// r2 = y x-y
	// (x+y)*(x-y)
	↓
	z

1. Show the results of your evaluation (using the metric memory traffic, i.e., counting the number of memory accesses):

Write each line of your x295++ program	Fetch	Execute
LOAD x, r0	2	1
COPY r1, r0	1	0
LOAD y, r2	2	1
ADD r2, r0	1	0
SUB r1, r2	1	0
MUL r0, r2	1	0
STORE r2, z	2	1
Grand Total:	Total: 10	Total: 3

=13

CMPT 295 – Spring 2020

2. How many instructions and words does the x295++ programs contain (static code size metric): The total instructions is seven and there are ten words.

Make sure this is the smallest program you can write using **x295++**!

Conclusion

Considering the memory traffic metric (number of memory accesses required by our test program), which instruction set (x295, x295+ or x295++) produces the most time efficient program?

The x295+ and x295++ both are the efficient time program. Both are three.

Considering the static code size metric (number of instructions/words required to implement our test program), which instruction set (x295, x295+ or x295++) produces the smallest program?

The x295+ is smallest program for static code size metric