CMPT 295

Name: Junchen Li

Student ID: 301385486

Date: 2020/4/1

Assignment 8 – out of 30 marks

Objectives:

- Manipulating multidimensional arrays in x86-64 assembly code
- Instruction Set Architecture – Memory address resolution versus word size – Memory addressing modes
- Timing Diagram
- Sequential and Staged Execution Analysis
- Staged Execution

Submission:

- Submit your document called **Assignment_8.pdf**, which must include your answers to all of the questions in Assignment 8.
    - Add your full name and student number at the top of the first page of your document **Assignment_8.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_8.pdf** electronically on CourSys.

Due:

- Thursday, April 2 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. [5 marks] Manipulating multidimensional arrays in x86-64 assembly code

   Do Problem 3.64 on pages 316 and 317 in our textbook. In addition, replicate the x86-64 assembly code (given in the problem) below and add a descriptive comment to each of its lines.

M[X][Y] A[S]=M+L(S)

L= size of data A[S][W]=M+S*X*L+L*W
=M+L(S*X+W)

A[S][W][T] A[S][W][K]=A+L*W*T*i+T*j*L+k*L
=A+L(W*T*i+T*j+K)

W*T=65
T=13
W=65/13=5

Total size is 3640 from assembly code. And for long the data size is 8
S*W*T*8=3640
S=3640/(8*65)=7

S=7, W=5, T=13

```
long A[R][S][T];

long store_ele(long i, long j, long k, long *dest)
{
        *dest = A[i][j][k];
        return sizeof(A);
}
```

In compiling this program, gcc generates the following assembly code:

```
long store_ele(long i, long j, long k, long *dest)
i in %rdi, j in %rsi, k in %rdx, dest in %rcx
store_ele:
leaq    (%rsi,%rsi,2), %rax      // %rax = 2j+j = 3*j
leaq    (%rsi,%rax,4), %rax      // %rax = 4*(3*j)+j = 13*j
movq    %rdi, %rsi               // %rsi = i
salq    $6, %rsi                 // %rsi = i*2^6 = i*64
addq    %rsi, %rdi               // %rdi = i+i*64 = 65*i
addq    %rax, %rdi               // %rdi = 65*i+13*j
addq    %rdi, %rdx               // %rdx = K+65i+13j
movq    A(,%rdx,8), %rax         // %rax = A+8(K+65*i+13j)
movq    %rax, (%rcx)             // *dest = A[65i+13j+K]
movl    $3640, %eax              // sizeof(A)=3640
ret                              // %rax=3640
                                 // return rax
```

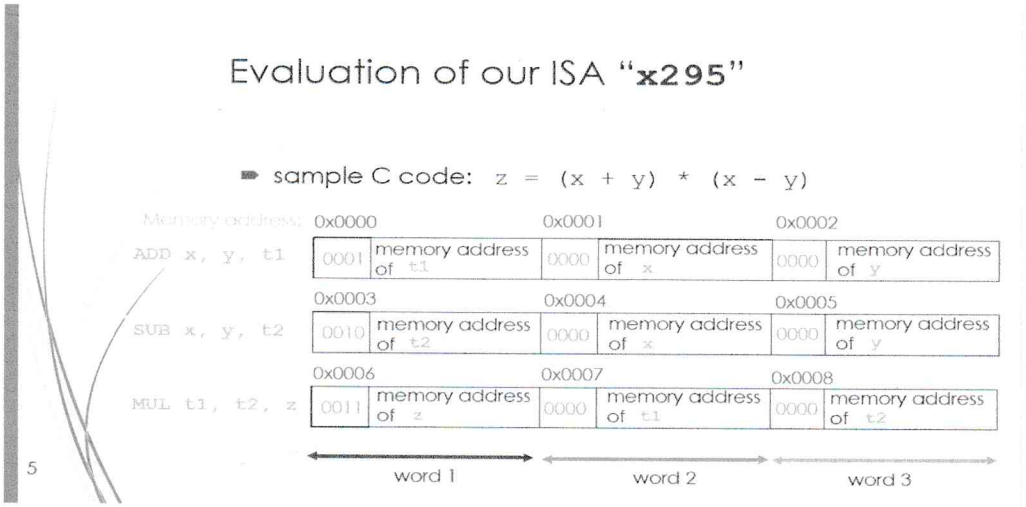
2. [8 marks] Instruction set architecture – Memory address resolution versus word size – Memory addressing modes

Part 1

In our lectures (Lectures 22 and 23) and our Assignment 8, we created three instruction sets, namely x295, x295+ and x295++ as part of our exploration of instruction set architecture (ISA) design. For all three of them, we used the following Memory Model:

- Size of memory: $2^{12} \times 16$ -> this is the size of the external memory, i.e., external to the CPU (processor) – this memory is often referred to as RAM
- Memory address size: 12 bits
- Word size: 16 bits

On Slide 5 of Lecture 23, in our effort to evaluate our ISA, we used three instructions from our x295 instruction set to translate the sample C code: `z = (x + y) * (x - y)`. Here is the complete Slide 5:



As you can see, this is a scenario in which the ADD instruction is stored at memory addresses `0x0000` to `0x0002`, the SUB instruction stored at memory addresses `0x0003` to `0x0005` and the MUL instruction at memory addresses `0x0006` to `0x0008`.

As we know from our experience of using the computers in CSIL (64-bit computers) and writing x86-64 assembly code, it is not always the case that the smallest addressable memory location is a word of 16 bits. Actually, on most 64-bit computers nowadays, word size is 64 bits and the smallest addressable memory location (often called the **memory address resolution** or **memore resolution**) is 8 bits, i.e., 1 byte.

Let's now make one (1) change to our x295 ISA:

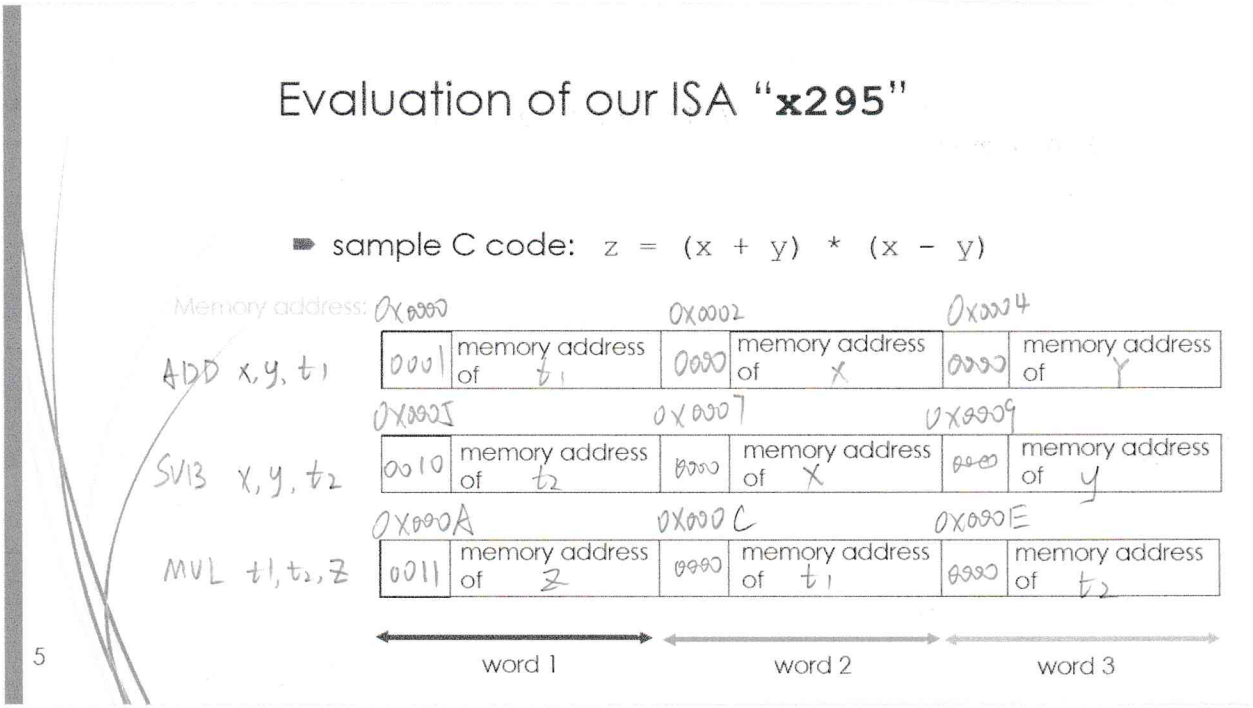
Memory model:

- Size of external memory (RAM): $2^{12} \times 8$

  Let's assume that the decrease in memory locations is not impacting our instruction sets.
- Memory address size: 12 bits
- Word size: 16 bits

Everything else about our x295 ISA (its assembly instructions, its templates, its opcodes, etc...) remain as previously defined.

Considering this change to oru x295 ISA, redo Slide 5 of Lecture 23:



Part 2

In our Lecture 25, in order to further decrease the effect of the von Newmann bottleneck, we used a third strategy to reduce the number of memory accesses the processor made when it fetched and executed our instructions. We did this by introducing other types (modes) of operands into our x295++ ISA, namely:

- immediate mode, in which the operand is a constant value

Note that our x295++ ISA already had the operand types (modes) **register**, in which an operand can be a register, and **memory**, in which an operand can represent a memory address.

Let's now investigate the effect of these various operand types on the execution of our instructions.

But first, let's modify the Memory Model of our x295++ ISA by making the one (1) change discussed in Part 1 of this question:

Memory model:

- Size of external memory (RAM): $2^{12} \times 8$
  i.e., the memory address resolution (smallest addressable memory location) is now 1 byte.

Now, let's imagine that some assembly instructions (from our x295++ instruction set) are loaded into contiguous memory locations starting at the memory address `0x240`. A partial result of disassembling thes instructions produces the following:

```
0x240   COPY $76₁₀, r2
?????   COPY r0, r2
?????   LOAD 0x510, r2
?????   LOAD (r1), r2
?????   LOAD -16₁₀(r3), r2
?????   LOAD 0(r0,r1), r2
?????   LOAD -8(r1,r0,2), r2
```

In addition, consider the following stack:

0x528	-20_{10}
0x520	0_{10}
0x518	20_{10}
0x510	40_{10}
0x508	60_{10}
0x500	80_{10}

and the following register file:

r0	24_{10}
r1	0x500
r2	40_{10}
r3	0x520
rip	0x240

As you hand trace the assembly instructions, starting with the instruction indicated by PC (%rip), complete the table below:

Content of PC	Assembly instruction Word size = 16	Meaning	Effective Address	Content of rC
0x240	COPY $\$76_{10}$, r2 4 9 3 (2)	rC ← value	n/a	76_{10}
0X242	COPY r0, r2 4 3 3 (2)	rC ← ra	n/a	24_{10}
0X244	LOAD 0x510, r2 4 12 2 (4)	rC ← M[a]	0x510	0XJ10
0X246	LOAD (r1), r2 4 2 3 (2)	rC ← M[ra]	0x500	80_{10}
0X249	LOAD -16_{10}(r3), r2 4 J 3 3 (2)	r2 ← M[ra+value]	0XJ10	40_{10}
0x24C 16 12 13	LOAD 0(r0,r1), r2 4 0 2 2 2 11 8 12 3	rC ← M[rA + rB + value]	0XJ18	20_{10}
0x250	LOAD -8(r1,r0,2), r2	rC ← M[rB + (rA*s) +value]	0X528	-20_{10}

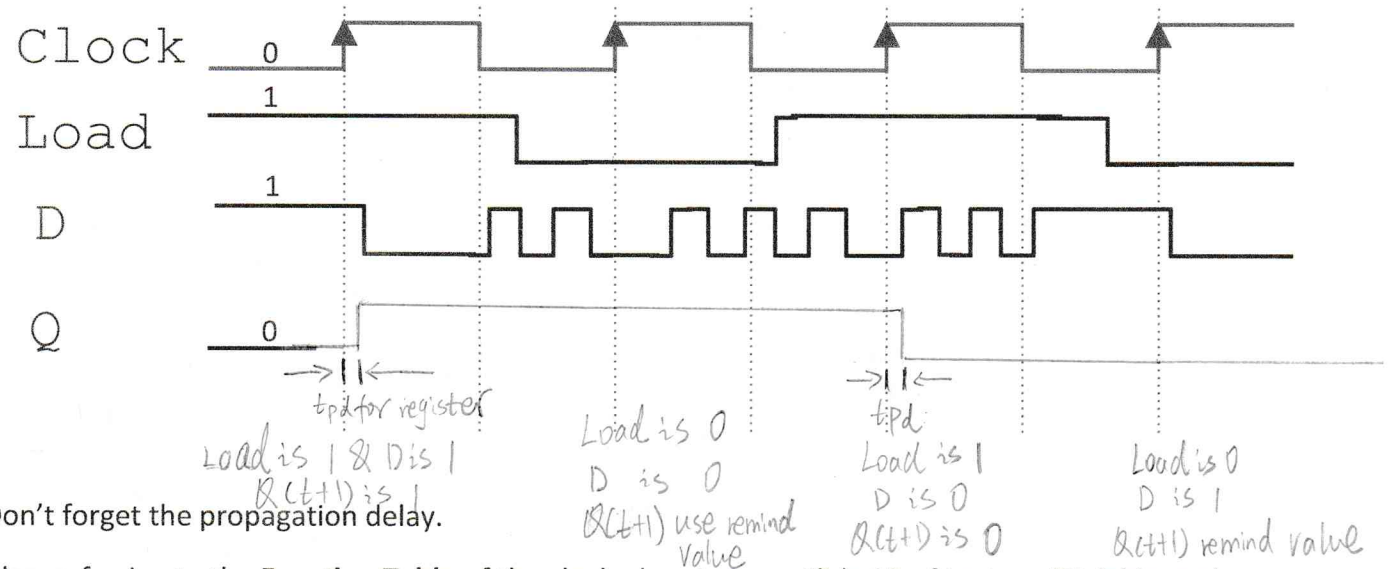
Note: The above assembly instructions do not form a sensical program.

Function table

Load	D	Q(t+1)
0	X	Q(t)
1	0	0
1	1	1

3. [5 marks] Timing Diagram

Complete the Q line on the following timing diagram:



Don't forget the propagation delay.

Also, referring to the **Function Table** of the clocked register on Slide 15 of Lecture 27, lable each segment of the Q line with the appropriate row of the Function Table. The 4 segments of the Q line to be labelled are as follows:

- Segment 1 of the Q line goes from the first rising edge of the clock to the second rising edge of the clock,
- Segment 2 of the Q line goes from the second rising edge of the clock to the third,
- Segment 3 of the Q line goes from the third rising edge of the clock to the fourth, and
- Segment 4 of the Q line goes from the fourth rising edge of the clock onwards.

---

4. [4 marks] Sequential and Staged Execution Analysis

Imagine we built a processor. The propagation delay of its entire combinational logic circuit is 600 ps and the propagation delay to load the clocked register we used is 30 ps.

a. What is the minimum clock cycle time (in picoseconds), the latency (in picoseconds) and throughput (in GIPS) of our processor if it executes instructions sequentially (one after the other)?

b. What is the minimum clock cycle time (in picoseconds), the latency (in picoseconds) and throughput (in GIPS) of our processor if it executes instruction using staged execution with **8 equal stages**?

a) total combinational logic circuit is 600ps, there are total five stages, so each stage is $600/5 = 120$ ps. Min clock cycle time is $120ps + 30ps = 150ps$. The latency is $600ps + 5 \cdot 30 = 750ps$. The throughput is $\frac{1 \text{ instruction}}{750 ps} = 1.3$ GIPS

b) each stage is $600/8 = 75$ ps, min clock cycle time is $75ps + 30ps = 105ps$. The latency is $600ps + 8 \cdot 30 = 840ps$ and the throughout is $\frac{1 \text{ instruction}}{840} = 1.2$ GIPS.

---

5. [8 marks] Staged Execution

In our Lecture 28, we described the detailed execution of three assembly instructions of our x295++ instruction set, namely:

1. `COPY r1, r2`
2. `LOAD 8(r6), r4`
3. `ADD r0, r5`

We did so by listing the micro-instructions into which each of the above instructions are translated during their staged execution.

In this question, you are asked to describe the detailed execution of other assembly instructions of our x295++ instruction set:

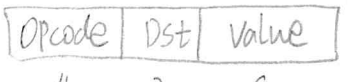
1. `ADD $0xA, r4 (opcode: 0100)`
2. `LOAD 0xFF0, r2 (opcode: 1010, 0xFF0 is a memory address)`
3. `STORE r0, 0(r5) (opcode: 1111)`
4. `SUB r3, r7 (opcode: 0010)`

When answering this question:

- Follow the manner in which this detailed description was done on Slides 7, 8 and 9 of Lecture 28, and
- Assume that the memory model of our x295++ ISA has now been modified such that its Memory address resolution (smallest addressable memory location) is now a 1 byte (its word size remains set to 16 bits - no change).

Suggestion: When answering this Question 5, you may find Section 4.3.1 of our textbook very useful.

① ADD $0xA, r4



0100 100 1010 ← M[PC]
ADD (4) (A)
PC ← PC+2 (1word)
ValA ← value
} Fetch ①

ValB ← r[4] Decode ②

ValC ← valA+valB Execute ③

r[4] ← valC write back ⑤

② LOAD 0xFFF0, r2

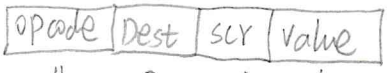
1111 111 0000

1010 010 111111 110000 ← M[PC]
LOAD (2) 0xFFF0
PC ← PC+2 (1word)
} Fetch ①

ValB ← M[value] memory ④

r[2] ← ValB write back ⑤

③ STORE r0, 0(r5)



1111 101 000 000000 ← M[PC]
ValC ← value
PC ← PC+2 (1word)
} Fetch ①

ValA ← r[5]
ValB ← r[0]
} Decode ②

ValE ← ValA+value Execute ③

M[value] ← ValB write back ⑤

④ SUB r3, r7

0010 111 011 xxx ← M[PC]
PC ← PC+2 (1word)
} Fetch ①

ValA ← r[7]
ValB ← r[3]
} Decode ②

ValE ← ValA - ValB // r7 ← r7 - r3
Execute ③

r[7] ← ValE write back ⑤

Bonus 2 marks:

Even though we have not included the following assembly instruction in our x295++ instruction set, describe the detailed execution of:

5. `POP r7 (opcode: 0110)`

Note: We can assume that the meaning of the `POP` instruction in x295++ is equivalent to the `POPQ` instruction in x86-64. rsp stack pointer

---

POP r7

0110 0111 } Fetch.

PC ← PC + 1

valA ← r[7] Decode

[rsp] ← [rsp] + 8 memory