Assignment 7

Objectives:

- 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**.
- Submit your assignment **Assignment_7.pdf** on CourSys.

Due:

- Friday Nov. 12 at 23:59:59.
- Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted on Monday) in order to provide feedback to the student.

Marking scheme:

- The marks assigned to each question is indicated in [ ].
- This assignment will be marked for correctness.
- A solution will be posted on Monday after the due date.

Note: In our lectures, we designed and evaluated the instruction set called **`x295M`**. In this assignment, we shall design and evaluate three (3) other instruction sets: **`x295`**, **`x295+`** and **`x295++`**.

Be aware: All of these 4 instruction sets are different!

Instruction Set 1 – x295

A. Description of x295 instruction set architecture (ISA)

- Data type size : 16 bits (for example, an integer has 16 bits)
- Memory model
  - Size of memory: $\mathbf{2^{12} \times 16}$
    - m = 12 -> this means that each memory address has 12 bits.
    - n = 16 -> this means that the address resolution, i.e., the smallest addressable memory "chunk", is a memory "chunk" of 16 bits.
      - Each memory "chunk" of 16 bits has a distinct memory address.
      - This is not a byte-addressable computer
  - Word size: 16 bits -> this means that when the microprocessor reads from/writes to memory, it reads/writes 16 bits at a time.
    - In this ISA, the address resolution == the word size. It is not always the case.
  - Number of registers: 0
- Instruction set (assembly and machine instructions)
  - Maximum number of instructions: 16
    - This means that we need a maximum of 4 bits to distinctly represent each of these 16 instructions.
    - Therefore, the size of the opcode field, in the machine instructions, will be 4 bits ($2^4 = 16$)
  - Operand Model:
    - Memory (only) – only memory locations are operands, no registers are used as operands except the register representing the stack pointer
    - *3-operand* model
    - In the machine instructions, the order of these operands is: Dest, Src1, Src2
  - Memory addressing mode: Direct, Base and Displacement and Indirect
    - These may not all be used in the instructions found in this assignment.
  - Assembly instructions (in this assignment we shall only define a subset of these instructions) and their format and meaning:
    - `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]`

  - In these assembly instructions, the order of the operands is: Src1, Src2, Dest.

- Machine code format:

opcode	Dest (12 bits)	padding	Src1 (12 bits)	padding	Src2 (12 bits)
4 bits		4 bits		4 bits	

This format is made of 3 words, each word is 16 bits in length (word size). This format must be used to form all three machine instructions corresponding to the three assembly instructions listed above.

The bit patterns for the opcode are:

Opcode (instruction)	**Bit pattern (4 bits)**
padding	`0000`
`ADD`	`0001`
`SUB`	`0010`
`MUL`	`0011`
…	…

B. Evaluation of x295 instruction set

C program	**x295 assembly program**	**x295 machine code**

`z = (x + y) * (x - y);`	`ADD x, y, tmp1`	`0001 <Dest 12 bits>`
		`0000 <Src1 12 bits>`
		`0000 <Src2 12 bits>`

	`SUB x, y, tmp2`	`0010 <Dest 12 bits>`
		`0000 <Src1 12 bits>`
		`0000 <Src2 12 bits>`

	`MUL tmp1, tmp2, z`
	(where `tmp1` and `tmp2` are memory addresses holding temporary results)
		`0011 <Dest 12 bits>`
		`0000 <Src1 12 bits>`
		`0000 <Src2 12 bits>`
		Note: In the above machine code, we abstractly express the memory addresses as `<Dest 12 bits>`, `<Src1 12 bits>` and `<Src2 12 bits>` as opposed to using actual memory addresses in

		the memory address fields (fields in purple).

1. The first step in evaluating our **x295** instruction set is to write an assembly program using the assembly instructions defined by our **x295** instruction set. This step has been done for us and is displayed in the table above in column "**x295 assembly program**".

   Note that as we are performing this step, we are verifying that our **x295** instruction set contains sufficient instructions to allow us to write such program in **x295** assembly code.

   (There is nothing for us to do in this part of our **x295** instruction set evaluation. We can use this as a model to follow when answering the rest of this assignment.)

   Once we have our **x295** assembly program, we need to transform it into its **x295** machine code equivalent. This step has been done for us and is displayed in the table above in column "**x295 machine code**".

   Note that as we are performing this step, we are verifying that our **x295** instruction set contains sufficient instructions to allow us to write such program in **x295** machine code.

   (Again, there is nothing for us to do in this part of our **x295** instruction set evaluation. We can use this as a model to follow when answering the rest of this assignment.)

2. The next step is evaluating our **x295** instruction set is to execute (hand trace) our assembly program or its corresponding machine code and using the metric (criteria) called ***memory traffic***, we count the number of memory accesses our program makes during its execution. In other words, we count how many time the execution of our program required a word (16 bits) to be read from or written to memory.

   Note that as we are performing this step, we are verifying that the meaning of the instructions contained in our **x295** instruction set is such that hand tracing these instructions does indeed produce the result the software developer that wrote the above C program expected. This is to say that if we use the test case: x = 3, y = 2 when hand tracing our **x295** assembly code (or its machine code equivalent), we would obtain the same (expected) result as if we were to hand trace the C program itself.

   For us to do: As part of this step in the evaluation of our **x295** instruction set, we are asked to evaluate the number of **word size memory accesses** made by the microprocessor when it is fetching and decoding/executing each assembly code statement (or machine code statement) listed in the left column in the table below. Also we need to justify our count. Finally, let's total our counts. **[2 marks]**

x295 program **(in assembly and machine code)**	**Fetch** **(number of word size memory accesses)** **+** **Provide an explanation explaning the count**	**Decode/Execute** **(number of word size memory accesses)** **+** **Provide an explanation explaning the count**

`ADD x, y, tmp1`
 	
`0001 <Dest 12 bits>`
`0000 <Src1 12 bits>`
`0000 <Src2 12 bits>`
	Count:
	
	Explanation:
		Count:
		
		Explanation:

`SUB x, y, tmp2`
 	
`0010 <Dest 12 bits>`
`0000 <Src1 12 bits>`
`0000 <Src2 12 bits>`
	Count:
	
	Explanation:
		Count:
		
		Explanation:

`MUL tmp1, tmp2, z`
 	
`0011 <Dest 12 bits>`
`0000 <Src1 12 bits>`
`0000 <Src2 12 bits>`
	Count:
	
	Explanation:
		Count:
		
		Explanation:

Grand Total:	**Total:**	**Total:**

3. We also evaluate our instruction set using the metric (criteria) called ***static code size***:
   - The code size of our **`x295`** program is **3 instructions**.
   - And since each instruction is 3 word long, the code size of our **`x295`** program is **9 words**.

(There is nothing for us to do in this part of our **x295** instruction set evaluation. We can use this as a model to follow when answering the rest of this assignment.)

Instruction Set 2 – x295+

A. Description of x295+ instruction set architecture (ISA)

Considering the results of our **x295** instruction set evaluation, one possible improvement in order to reduce the number of memory accesses could be to reduce the size of the machine instructions. Since these are made of an opcode and a memory address, we could introduce registers which could hold these memory addresses and take less space in the machine code than memory addresses (i.e., registers will take fewer bits to identify than memory addresses).

With the idea of introducing registers, we specified a second instruction set architecture (ISA) with the same components as **x295**, but with the following modifications:

- Memory model
  - Number of registers: 8 x 16-bit registers
    - This means that we need a maximum of 3 bits to distinctly represent each of these 8 (2^3) registers.
    - Therefore, the size of any register field, in the machine instructions, will be 3 bits in length.
    - Each register contains 16 bits.
  - The bit patterns to uniquely identify each the registers are:

Register	Bit pattern (3 bits)
r0	000
r1	001
r2	010
r3	011
r4	100
r5	101
r6	110
r7	111

- Instruction set (assembly and machine instructions)
  - Operand Model:
    - Registers

- We shall also use memory addresses

- Assembly instructions:

Data manipulation type of 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`

Data transfer type of instructions

- `LOAD a,rC` Meaning: `rC <- M[a]`
- `STORE rA,c` Meaning: `M[c] <- rA`

- Machine code format:

Format 1:

Opcode (4 bits)	Dest (3 bits)	Src1 (3 bits)	Src2 (3 bits)	XXX

This format is made of 16 bits (1 word) and it can be used to form the machine instructions corresponding to the ADD, SUB and MUL assembly instructions.

XXX -> these X's can be either 0's or 1's – they are not used by the microprocessor.

Format 2:

Opcode (4 bits)	Dest (3 bits)	XXX	XXXXXX	padding	Memory address of Src (12 bits)

This format is made of 32 bits (2 words) and it can be used to form the machine instructions corresponding to the LOAD assembly instruction.

XXX, XXXXXX and XXXX -> these X's can be either 0's or 1's. – they are not used by the microprocessor.

Format 3:

Opcode (4 bits)	XXX	Src (3 bits)	XXXXXX	padding	Memory address of Dest (12 bits)

This format is made of 32 bits (2 words) and it can be used to form the machine instructions corresponding to the STORE assembly instruction.

XXX and XXXXXX -> these X's can be either 0's or 1's– they are not used by the microprocessor.

When the microprocessor decodes the opcode for a LOAD instruction, it will know where to find the field **Dest** and where to find the field **Src** in the instruction itself (**Dest** -> the first field after the opcode in the first word of the insruction and **Src** -> the memory address located in the least significant 12 bits of the second word composing this instruction). It will also know to ignore all other bits (X's) in the instruction.

When the microprocessor decodes the opcode for a STORE instruction, it will know where to find the field **Src** and where to find the field **Dest** in the instruction itself (**Src** -> the second field after the opcode in the first word of the insruction and **Dest** -> the memory address located in the least significant 12 bits of the second word composing this instruction). It will also know to ignore all other bits (X's) in the instruction.

Note: About ISA design principles: When creating formats to encode instruction set …

- As few of them as possible were created.
- The fields that have the same purpose (such as Opcode, Dest and Src) are placed in the same location in as many of the formats as possible.

This helps simplify the design of the microprocessor (its datapath).

The bit patterns for the opcode are:

Opcode (instruction)	**Bit pattern (4 bits)**
padding	`0000`
`ADD`	`0001`
`SUB`	`0010`
`MUL`	`0011`
`LOAD`	`1010`
`STORE`	`1011`
…	…

B. Description of x295+ instruction set

C program	**x295+ assembly program**	**x295+ machine code**

`z = (x + y) * (x - y);`		`1010 000 XXX XXXXXX`
		`0000 <Src 12 bits>`
		
		`1010 001 XXX XXXXXX`
		`0000 <Src 12 bits>`
		
		`0001 010 000 001 XXX`
		
		`0010 011 000 001 XXX`

		`0011 100 010 011 XXX`
		
		`1011 XXX 100 XXXXXX`
		`0000 <Src 12 bits>`

1. The first step in evaluating our **`x295+`** instruction set is to translate the C program into
   - an assembly program using the assembly instructions defined by our **`x295+`** instruction set and
   - its corresponding machine code using the machine instructions defined by our **`x295+`** instruction set.

   For us to do: Our task is to complete the middle column (assembly code) in the table above. The machine code has already been given. **[3 marks]**

2. The next step in evaluating our **x295+** instruction set is to execute (hand trace) our assembly program or its corresponding machine code and using the metric (criteria) called ***memory traffic***, we count the number of memory accesses our program makes during its execution. In other words, we count how many time the execution of our program required a word (16 bits) to be read from or written to memory.

   For us to do: As part of this step in the evaluation of our **`x295+`** instruction set, complete the table below. **[4 marks]**

`x295+` program **(in assembly and machine code)**	**Fetch** **(number of word size memory accesses)** **+** **Provide an explanation explaning the count**	**Decode/Execute** **(number of word size memory accesses)** **+** **Provide an explanation explaning the count**
Assembly code: **Machine code:**	**Count:** **Explanation:**	**Count:** **Explanation:**

`1010 000 XXX XXXXXX`
`0000 <Src 12 bits>`		

Assembly code:
 	
Machine code:
`1010 001 XXX XXXXXX`
`0000 <Src 12 bits>`
	Count:
	
	Explanation:
		Count:
		
		Explanation:

Assembly code: **Machine code:** `0001 010 000 001 XXX`	**Count:** **Explanation:**	**Count:** **Explanation:**
Assembly code: **Machine code:** `0010 011 000 001 XXX`	**Count:** **Explanation:**	**Count:** **Explanation:**
Assembly code: **Machine code:** `0011 100 010 011 XXX`	**Count:** **Explanation:**	**Count:** **Explanation:**
Assembly code: **Machine code:**	**Count:** **Explanation:**	**Count:** **Explanation:**

`1011 XXX 100 XXXXXX`
`0000 <Src 12 bits>`		

Grand Total:	**Total:**	**Total:**

3. We also evaluate our instruction set using the metric (criteria) called ***static code size***. **For us to do:** Fill in the blanks in the statement below: **[0.5 marks]**
   - The code size of our **`x295+`** program is __________ **instructions** (__________ **words**).

Instruction Set 3 – x295++

A. Description of x295++ instruction set architecture (ISA)

Would reducing the number of operands to the instructions in our instruction set decrease the number of memory accesses the microprocessor does when fetching, decoding and executing our program?

Would introducing another instruction, namely COPY, in our instruction set also decrease the number of memory accesses the microprocessor does when fetching. decoding and executing our program?

With the above two ideas in mind, we specified a third instruction set architecture (ISA) with the same components as **`x295`** and **`x295+`**, but with the following modifications:

- Instruction set (assembly and machine instructions)
  - Operand Model:
    - 2 operand model
  - Assembly 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`
  - Machine code formats:

Format 1:

Opcode (4 bits)	Dest (3 bits)	Src (3 bits)	XXXXXX

This format is made of 16 bits (1 word) and it can be used to form the machine instructions corresponding to the ADD, SUB, MUL and COPY assembly instructions.

XXXXXX -> these X's can be either 0's or 1's.

The bit patterns for the opcode are:

Opcode (instruction)	**Bit pattern (4 bits)**
padding	`0000`
`ADD`	`0001`
`SUB`	`0010`
`MUL`	`0011`
`COPY`	**`1001`**
`LOAD`	`1010`
`STORE`	`1011`
…	…

B. Evaluation of x295++ instruction set

C program	**x295++ assembly program**	**x295++ machine code**
`z = (x + y) * (x - y);`		

1. The first step in evaluating our **`x295++`** instruction set is to translate the C program into
   - an assembly program using the assembly instructions defined by our **`x295++`** instruction set and
   - its corresponding machine code using the machine instructions defined by our **`x295++`** instruction set.

   For us to do: Our task is to complete the middle column (assembly code) and the right column (machine code) in the table above. **[5 marks]**

   Challenge: Can you express your **x295++** assembly program and machine code with the fewest number of instructions. This may require you to first readjust the above C program.

2. The next step in evaluating our **x295++** instruction set is to execute (hand trace) our assembly program or its corresponding machine code and using the metric (criteria) called ***memory traffic***, we count the number of memory accesses our program makes during its execution. In other words, we count how many time the execution of our program required a word (16 bits) to be read from or written to memory.

For us to do: As part of this step in the evaluation of our **x295++** instruction set, complete the table below. **[4 marks]**

x295++ program **(in assembly and machine code)**	**Fetch** **(number of word size memory accesses)** **+** **Provide an explanation explaning the count**	**Decode/Execute** **(number of word size memory accesses)** **+** **Provide an explanation explaning the count**
... expand the table by adding as many rows as needed using the row format seen in the tables above ...		
Grand Total:	**Total:**	**Total:**

3. We also evaluate our instruction set using the metric (criteria) called ***static code size***. **For us to do:** Fill in the blanks in the statement below: **[0.5 marks]**
   - The code size of our **x295++** program is __________ **instructions** (__________ **words**).

Conclusion

For us to do: 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? **[0.5 marks]**

For us to do: 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? **[0.5 marks]**