In this assignment, you will implement a 32-bit custom RISC-V processor and an LLVM compiler extension for it. You will add the given custom instructions, in addition to a subset of the basic RISC-V instruction set, to the LLVM compiler backend as a RISC-V extension. Then, compile programs that use these instructions and run them on a simple processor you design that supports the custom instruction extension. Table 1 shows the basic instructions you will use from RISC-V ISA, the custom instructions to be added and their encodings, and Table 2 explains the functions of these custom instructions.

TABLE 1: Base and Custom RISC-V Instructions to be Implemented

Instruction Types for Base and Custom Instructions

31 30 29 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type
s1	funct11	rs1	funct3	rd	opcode	Custom-type1
s2	imm[9:5] rs2	rs1	funct3	imm[4:1|10]	opcode	Custom-type2

A Subset of RISC-V Base Instructions

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
0000000	imm[4:0]	rs1	001	rd	0010011	SLLI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	111	rd	0110011	AND

Custom Instructions

0000000	rs2	rs1	000	rd	1110111	SUB.ABS
imm[11:0]		rs1	100	rd	1110111	AVG.FLR
imm[11:0]		rs1	101	rd	1110111	MOVU
0000010	rs2	rs1	001	rd	1110111	SRT.CMP.ST
0000100	rs2	rs1	110	rd	1110111	LD.CMP.MAX
0001000	rs2	rs1	111	rd	1110111	SRCH.BIT.PTRN
s1	10101010101	rs1	010	rd	1110111	SEL.PART
s2	imm[9:5] rs2	rs1	000	imm[4:1|10]	1111111	SEL.CND
s2	imm[9:5] rs2	rs1	111	imm[4:1|10]	1111111	MAC.LD.ST

TABLE 2: Explanations of Custom RISC-V Extension Instructions

Instruction	Function and Format of the Instruction
SUB.ABS	Subtracts the value in register *rs2* from *rs1*. It then calculates the absolute value of the result and writes it to the *rd* register. Assembly format: ***sub.abs rd, rs1, rs2***
AVG.FLR	Calculates the average of the values in register *rs1* and *sign extended immediate* value, then writes the floor integer value of this average to the *rd* register. Assembly format: ***avg.flr rd, rs1, imm***
MOVU	Writes the *unsigned extended immediate* value to the *rd* register. (Here, *rs1* is the *x0* register in RISC-V, which must always be 0.) Assembly format: ***movu rd, rs1, imm***
SRT.CMP.ST	Compares the signed values in registers *rs1* and *rs2*, then stores the smaller of the two values to the memory address specified in rd, and the larger value to the subsequent memory address (*rd* + *4*). *Execute stage must take 2 cycles for this instruction.* Assembly format: ***srt.cmp.st rd, rs1, rs2***
LD.CMP.MAX	Looks -loads- values from three different addresses in the memory, the addresses are the unsigned values in registers *rd*, *rs1* and *rs2*. Then, it loads the maximum value among them to the *rd* register. *Execute stage must take 3 cycles for this instruction.* Assembly format: ***ld.cmp.max rd, rs1, rs2***
SRCH.BIT.PTRN	Searchs for an 8-bit pattern on rs1 register value and if it is found, it writes "1" to the *rd* register, if not, writes "0". The 8-bit pattern is determined by the lower 8-bits of rs2 register value. Assembly format: ***srch.bit.ptrn rd, rs1, rs2***
SEL.PART	Selects 16-bit part of the value of rs1 register. If *s1* select bit is "1" then it selects most significant 16-bit, otherwise it selects least significant 16-bit, then it writes unsigned extended selected value to *rd* register. Assembly format: ***sel.part rd, rs1, s1***
SEL.CND	By checking the s2 selection value, it does signed comparison on the values of the registers rs1 and rs2 and if the relevant condition is met, it branches with the sign extended immediate value. (updates the program counter) (also be aware that imm[0] = 0). The corresponding comparisons that need to be made according to the selection bits are as follows: ***s2 = 00*** –> It will branch if the condition ***rs1_value == rs2_value*** is true. ***s2 = 01*** –> It will branch if the condition ***rs1_value >= rs2_value*** is true. ***s2 = 10*** –> It will branch if the condition ***rs1_value < rs2_value*** is true. ***s2 = 11*** –> It will behave like *nop* instruction, meaning no comparison or branching will be performed, only the *program counter* will be updated for the next instruction. (*pc* + 4) Assembly format: ***sel.cnd rs1, rs2, imm, s2***
MAC.LD.ST	It loads a pair of values from the memory addresses specified by rs1 and rs2, multiplies them, and adds this value to the value in the memory address given by immediate value. (Be aware that imm[0] = 0) (It should load the value from the address, do addition with this value and the product, then should store in the same address) *s2* value shows how many times to do this operation, so the operation should be done *s2 + 1* times and each time *rs1* and *rs2* addresses incremented by 4 for each multiply-accumulate operation. For example, if *s2* is "10", then the operation will be done 3 times, these operations will be performed in order: 1-) Load value from the address specified in rs1 register 2-) Load value from the address specified in rs2 register and perform Product = Mem[rs1] * Mem[rs2] 3-) Load value from the address specified as immediate value and perform Mem[imm] + Product 4-) Store the latest value to the address specified as immediate value 5-6-7-8-) Repeat 1-2-3-4 steps for rs1+4 and rs2+4 addresses 9-10-11-12-) Repeat 1-2-3-4 steps for rs1+8 and rs2+8 addresses *Execute stage must take (s2+1)*4 cycles for this instruction.* Assembly format: ***mac.ld.st rs1, rs2, imm, s2***

You need to check RISC-V ISA documentation for base instruction explanations: https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf

You can also use other helpful resources such as: https://msyksphinz-self.github.io/riscv-isadoc

[40 Points] RISC-V ISA LLVM Custom Instruction Extension

LLVM is a modular compiler library that translates from many front-end languages (C/C++, Swift, Rust, Julia, etc.) to a common intermediate language (LLVM IR) and can generate specific code from this intermediate language on many backends (x86, ARM, MIPS, RISC-V, etc.). (https://llvm.org) In this assignment, you will add an instruction extension, in addition to the standard I, M, C, A, F, D, and Q extensions, to LLVM's RISC-V backend to be able to translate the given custom instructions into machine code by using the flexibility provided by LLVM. You will use *LLVM Target Description Language* for instruction definitions.

Before adding the extension, you need to install the necessary dependencies and build the library correctly.

Note: *If you are a Windows user, you can use a virtual machine on a program like Ubuntu WSL (https://ubuntu.com/wsl) or VirtualBox (https://www.virtualbox.org) and following the steps for Ubuntu.*

If you are using Linux (e.g., Ubuntu), install necessary dependencies (*gcc, g++, git, cmake, ninja*) in a terminal as follows:

```
sudo apt update
sudo snap refresh
sudo apt install gcc g++
sudo apt install git
sudo snap install cmake --classic
sudo apt install ninja-build
```

If you are using macOS, install them as below:

```
brew update
brew install gcc
brew install git
brew install cmake
brew install ninja
```

Note: *The next steps do not differ for Linux and macOS systems, use the same commands.*

Clone the LLVM library from GitHub and checkout the commit version *461274b81d8641eab64d494accddc81d7db8a09e* (LLVM version *18.1.0*) (you can also download it from https://github.com/llvm/llvm-project/archive/refs/tags/llvmorg-18.1.0.zip):

```
git clone https://github.com/llvm/llvm-project
cd llvm-project
git checkout 461274b81d8641eab64d494accddc81d7db8a09e
```

After cloning the library, create a *build* folder and build the necessary tools;

clang –> to compile C code

llc –> to compile LLVM IR or LLVM bytecode intermediate language

llvm-objdump –> to get hex code from object code and see the assembly equivalents

Build it with the configurations provided for the RISC-V backend:

```
cd llvm-project
mkdir build
cd build
export CXXFLAGS="-std=c++11 -include limits"
cmake -G Ninja \
-DCMAKE_BUILD_TYPE=Release \
-DLLVM_ENABLE_PROJECTS=clang \
-DLLVM_TARGETS_TO_BUILD=RISCV \
-DBUILD_SHARED_LIBS=True \
-DLLVM_PARALLEL_LINK_JOBS=1 \
../llvm
ninja -j1 clang llc llvm-objdump
```

By this way, *clang, llc, and llvm-objdump* binaries we need are created in *llvm-project/build/bin* folder path. For faster build times, you can assign more threads to *ninja -j1* command instead of "1". You can find the maximum number of threads that you can assign via *nproc* command in the terminal.

If the library is built without any problems, you can then proceed to add the extension.

The modifications you need to make to the files to add the extension are described below. The extension name should be your surname without any Turkish characters. "Kasirga" is used here as an example.

The file modifications required to add the instruction extension should be made in **llvm-project/llvm/lib/Target/RISCV** folder path. Navigate to **llvm-project/llvm/lib/Target/RISCV** folder path in the LLVM library and create **RISCVInstrInfo<Surname>.td** file where you will define the instructions in the extension:

```
cd llvm-project/llvm/lib/Target/RISCV
touch RISCVInstrInfoKasirga.td
```

At the end of **RISCVInstrInfo.td** file, include **RISCVInstrInfo<Surname>.td** file that you created by adding a line like the following:

```
include "RISCVInstrInfoKasirga.td"
```

In **RISCVFeatures.td** file, define the extension as follows (the parts you need to change with your surname are shown in red):

```
def FeatureExtKasirga
    : SubtargetFeature<"kasirga", "HasExtKasirga", "true",
                       "'K' (Kasirga Instructions)">;
def HasExtKasirga : Predicate<"Subtarget->hasExtKasirga()">,
                            AssemblerPredicate<(all_of FeatureExtKasirga),
                            "'K' (Kasirga Instructions)">;
```

Now, you can add custom instructions to **RISCVInstrInfo<Surname>.td** file. You can add a sample instruction with an encoding *0000000_rs2_rs1_000_rd_0000000* as follows:

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
def SAMPLE_INSTRUCTION : RVInst<(outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
    "sample.instruction", "$rd, $rs1, $rs2", [], InstFormatOther>, Sched<[]>
 {
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;

    let Inst{31-25} = 0b0000000;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b000;
    let Inst{11-7} = rd;
    let Inst{6-0} = 0b0000000;
}
```

Here, for the immediate values of instructions that have immediate part, you should use definitions such as *simm12*, *simm13_lsb0*, and *simm21_lsb0_jal*, instead of *GPR* register definition. You can add other instruction definitions as shown in the example, by looking at how similar instructions are decoded in the **RISCVInstrInfo<Extension_Name>.td** files, or by searching and trying.

After adding the instructions, you need to go to **llvm-project/build** folder and rebuild the library in the same way for the changes to take effect:

```
cd llvm-project/build
ninja -j1 clang llc llvm-objdump
```

Now, you will be able to compile programs containing the instructions you added with *clang* and *llc*, and with *llvm-objdump* you will be able to see the *hex codes* of compiled programs.

After rebuilding the library, save the following C code to a file, written as *inline assembly* (*https://en.cppreference.com/w/c/language/asm*), compile the program with *clang* and *llc*, and look at the hex output with *llvm-objdump*:

```
// example.c
asm ("lui x3, 0x10001");
asm ("movu x5, x0, 250");
asm ("movu x6, x0, 125");
asm ("sw x5, 0(x3)");
asm ("sw x6, 4(x3)");
asm ("sw x5, 8(x3)");
asm ("lw x4, 0(x3)");
asm ("sub.abs x7, x4, x6");
asm ("avg.flr x8, x7, 50");
asm ("srch.bit.ptrn x9, x5, x6");
asm ("add x12, x3, 12");
asm ("srt.cmp.st x12, x5, x6");
asm ("add x13, x3, 0");
asm ("add x14, x3, 4");
asm ("add x15, x3, 8");
asm ("ld.cmp.max x13, x14, x15");
asm ("bge x5, x6, 0x00000148");
asm ("sel.part x10, x7, 1");
asm ("jal x1, 0x00000200");
asm ("0x00000148:\n"
                "sel.cnd x7, x8, 0x00000208, 2");
asm ("0x00000200:\n"
                "add x20, x3, 20\n"
                "mac.ld.st x14, x15, 0x00000600, 0");
asm ("0x00000208:\n"
                "sel.cnd x1, x1, 0x00000208, 0");
```

Note: *The example C program is written as inline assembly, but you also should be able to compile and run basic behavioral C codes that matches your basic RISC-V instruction set. You may need to check assembly output if it is suitable for your instructions in the processor or it is possible to disable other instructions as well.*

While compiling, the flag should be in the form of *-mattr=+<surname>*:

```
## Compiling C code to RISC-V Assembly code - not needed, just for checking asm output
llvm-project/build/bin/clang -S -target riscv32 -O0 example.c -o example.s

## Compiling C code to LLVM IR code
llvm-project/build/bin/clang -S -emit-llvm -target riscv32 -O0 example.c -o example.ll

## Compiling LLVM IR code to object code
llvm-project/build/bin/llc -mtriple=riscv32 -O0 -mattr=+kasirga -filetype=obj \
example.ll -o example.o

## Getting a hex output from object code
llvm-project/build/bin/llvm-objdump -d example.o
```

The example output of *llvm-objdump* is as follows (The red part shows the hex code of the relevant instruction, and the green part shows the assembly equivalents of the corresponding instructions. The numbers without *0x* in the assembly part are in decimal):

```
shc@kasirga:~/projects$ llvm-project/build/bin/llvm-objdump -d example.o

example.o:    file format elf32-littleriscv

Disassembly of section .text:

00000000 <.text>:
       0: b7 11 00 10 lui gp, 0x10001
       4: f7 52 a0 0f movu t0, zero, 0xfa
       8: 77 53 d0 07 movu t1, zero, 0x7d
       c: 23 a0 51 00 sw t0, 0(gp)
      10: 23 a2 61 00 sw t1, 4(gp)
      14: 23 a4 51 00 sw t0, 8(gp)
      18: 03 a2 01 00 lw tp, 0(gp)
      1c: f7 03 62 00 sub.abs t2, tp, t1
      20: 77 c4 23 03 avg.flr s0, t2, 0x32
      24: f7 f4 62 10 srch.bit.ptrn s1, t0, t1
      28: 13 86 c1 00 addi a2, gp, 0xc
      2c: 77 96 62 04 srt.cmp.st a2, t0, t1
      30: 93 86 01 00 mv a3, gp
      34: 13 87 41 00 addi a4, gp, 0x4
      38: 93 87 81 00 addi a5, gp, 0x8
      3c: f7 66 f7 08 ld.cmp.max a3, a4, a5
      40: 63 d4 62 14 bge t0, t1, 0x188 <.text+0x188>
      44: 77 a5 53 d5 sel.part a0, t2, 0x1
      48: ef 00 00 20 jal 0x248 <.text+0x248>
      4c: 7f 84 83 a0 sel.cnd t2, s0, 0x254, 0x2
      50: 13 8a 41 01 addi s4, gp, 0x14
      54: ff 70 f7 20 mac.ld.st a4, a5, 0x654, 0x0
      58: 7f 84 10 20 sel.cnd ra, ra, 0x260, 0x0
```

After adding the instructions, you can check whether the hex code for this example program appears same in your compiler. You should see that the hex code for the instructions comes in this little endian format. For example, the hex code for the first *movu* instruction is ***f7_52_a0_0f***, but you should actually translate it as ***0f_a0_52_f7***. This will show you the correct binary equivalent, ***0000111111010_00000_101_00101_1110111***.

You can test your processor with a testbench by running this program on the processor you will design in the next section.

[60 Points] Verilog Design of Custom RISC-V Processor

Write a multi cycle 32-bit processor that executes the given instructions in Verilog hardware description language using behavioral modeling. Let the file and module name be your surname, in lowercase letters and without any Turkish characters, as <surname>.v.

The processor will have these four stages: Fetch, Decode, Execute (including memory operations), Writeback. Before the processor fetches next instruction, the current instruction must go through all stages. All stages will take one cycle to operation, so each instruction will be retired in exactly 4 cycles, except *SRT.CMP.ST* (2 cycles in Execution stage), *LD.CMP.MAX* (3 cycles in Execution stage), *MAC.LD.ST* (4*(s2+1) cycles in Execution stage) instructions. For these instructions, the Execute stage will take longer and must not go further to the Writeback until Execute stage cycles to be completed. Note that the cycles should be exact. For example, if *s2* value is "1" for *MAC.LD.ST* instruction meaning two MAC operations, then it should take 8 cycles (11 cycles in total for this instruction) exactly in the Execute stage.

Note that the program counter will start at *0x0*.

Note that data memory is combinational for reads and sequential for writes. That means you can read immediately, but the data will be written in one cycle, so the change can be seen in the next cycle. The data memory will be in your testbench, not inside your processor.

The inputs and outputs of the "<surname>" module can be seen in Table 3.

TABLE 3: Input and Output Signals of <*surname*> Module

Signal	Direction	Width	Explanation
clk_i	Input	1 bit	Clock signal for sequential logic.
rst_i	Input	1 bit	The signal that returns the circuit to its initial state. The program counter and registers should be resetted to 0.
inst_i	Input	32 bits	The instruction that the processor will execute at the current clock tick. ***Note:*** *The instructions are coming in little endian format as the compiler's hex output in the previous section.*
pc_o	Output	32 bits	Address of the instruction to be fetched in the next cycle.
regs_o	Output	32 x 32 bits (1024 bits) {x31, ..., x0}	Gives inner register file as an output.
data_mem_we_o	Output	1 bit	Write enable signal to write to data memory.
data_mem_addr_o	Output	32 bits	The address to reach data memory.
data_mem_wdata_o	Output	32 bits	Data to be written to data memory.
data_mem_rdata_i	Input	32 bits	Read data from data memory.
cur_stage_o	Output	2 bits	The signal shows current operating stage in the core. Default 0 is Fetch, 1 is Decode, 2 is Execute and 3 is Writeback.

At each positive edge of the clock, your processor must decode the instruction received via *inst_i* signal and update the register values and program counter, do memory operations according to the calculation to be performed. The first instruction should be fetched when the program counter is at "0" in the first tick after your circuit is reset. As noted, instructions will come in little endian format as it is shown in the previous section, so you need to handle *inst_i* according to this format.

[30 Points] Drawing of Custom RISC-V Processor

In this part, you are required to draw the processor that executes all the given instructions you built in the previous part, by hand. (You can also use drawio-like drawing tools.) Additionally, create the truth table of the control unit. Save it as <Surname_StudentNumber>.pdf.

Submission (Deadline: 16.11.2025 23.59)

⚠**You can use online resources or generative AI tools however we will do a plagiarism check across all codes and any matches will get 0 points from the homework. If you do it yourself -even with some help of AI-, you have nothing to worry about.**

1-) RISCVFeatures.td
2-) RISCVInstrInfo.td
3-) RISCVInstrInfo<Surname>.td
4-) <surname>.v
5-) <Surname_StudentNumber>.pdf
Compress these files and save it as <Surname_StudentNumber>.zip and upload it to https://uzak.etu.edu.tr.