Bu ödevde 32 bit RISC-V buyruk kümesi mimarisi için vereceğimiz özel buyrukları gerçekleyeceksiniz. Bu özel buyrukları LLVM derleyici kütüphanesine RISC-V eklentisi olarak ekleyip daha sonra bu buyrukları içeren programları derleyip tasarlayacağınız özel buyruk eklentili basit bir işlemci üzerinde çalıştıracaksınız. Tablo 1'de eklenmesi gereken buyruklar ve buyruk kodlamaları görülebilir, buyrukların işlevleri ise Tablo 2'de açıklanmıştır.

TABLO 1: RISC-V Buyruk Kümesi Eklentisi Özel Buyruklar

Özel Buyruklar İçin Kullanılan RISC-V Buyruk Tipleri

31 30 29 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-tipi
imm[11:0]		rs1	funct3	rd	opcode	I-tipi
imm[20|10:1|11|19:12]				rd	opcode	J-tipi
s1 | funct11		rs1	funct3	rd	opcode	Özel-tip1
s2 | imm[9:5]	rs2	rs1	funct3	imm[4:1|10]	opcode	Özel-tip2

Özel Buyruklar

0000000	rs2	rs1	000	rd	1110111	KAREAL.TOPLA
1000010	rs2	rs1	001	rd	1110111	CARP.CIKAR
imm[11:0]		rs1	100	rd	1110111	SIFRELE
imm[11:0]		rs1	101	rd	1110111	TASI
imm[20|10:1|11|19:12]				rd	1111111	IKIKAT.ATLA
s1 | 10101010101		rs1	010	rd	1110111	BITSAY
s2 | imm[9:5]	rs2	rs1	111	imm[4:1|10]	1111111	SEC.DALLAN

TABLO 2: RISC-V Buyruk Kümesi Eklentisi Özel Buyrukların İşlevleri

Buyruk	**Buyruğun İşlevi ve Formatı**
KAREAL.TOPLA	*rs1* ve *rs2* yazmaçlarındaki değerlerin karelerinin toplamını *rd* yazmacına yazar. Assembly formatı: ***kareal.topla rd, rs1, rs2***
CARP.CIKAR	*rs1* ve *rs2* yazmaçlarındaki değerleri çarpar, çıkan sonuçtan *rs1* yazmacındaki değeri çıkarıp *rd* yazmacına yazar. Assembly formatı: ***carp.cikar rd, rs1, rs2***
SIFRELE	*rs1* yazmacındaki değer ile *işaretle genişletilmiş anlık değeri* "xor"layıp *rd* yazmacına yazar. Assembly formatı: ***sifrele rd, rs1, imm***
TASI	*Sıfırla genişletilmiş anlık değeri*, *rs1* yazmaç değeriyle toplar ve *rd* yazmacına yazar. (*rs1* RISC-V'de her zaman 0 olması gereken *x0* yazmacıdır.) Assembly formatı: ***tasi rd, rs1, imm***
IKIKAT.ATLA	*İşaretle genişletilmiş anlık değerin* 2 katına atlar (*program sayacını* günceller) ve *program sayacının* bir sonraki buyruk için olan değerini ($ps + 4$) *rd* yazmacına kaydeder. (ayrıca *imm[0] = 0*) Assembly formatı: ***ikikat.atla rd, imm***
BITSAY	*s1 (1 bitlik seçim) bitine* bakarak *rs1* yazmacındaki değerde bulunan "0" ya da "1" sayısını sayıp *rd* yazmacına yazar. *Seçim biti* "1" ise "1", "0" ise "0" sayısını sayar. Assembly formatı: ***bitsay rd, rs1, s1***
SEC.DALLAN	*s2 (2 bitlik seçim) bitine* bakarak *rs1* ve *rs2* yazmaç değerlerini *işaretli* olarak karşılaştırıp ilgili koşulu sağlıyorsa, *işaretle genişletilmiş anlık değerle* dallanır. (*program sayacını* günceller) (ayrıca *imm[0] = 0*) Seçim bitlerine göre denk gelen buyruklar ve yapılması gereken karşılaştırmalar aşağıdaki şekildedir: ***s2 = 00*** ise *nop* buyruğu gibi davranacak, yani karşılaştırma ve dallanma işlemi yapılmayacak, sadece, *program sayacı* bir sonraki buyruk için güncellenecek. ($ps + 4$) ***s2 = 01*** ise *beq* buyruğu gibi davranacak, yani ***rs1_deger == rs2_deger*** koşulu doğruysa dallanacak. ***s2 = 10*** ise *blt* buyruğu gibi davranacak, yani ***rs1_deger <rs2_deger*** koşulu doğruysa dallanacak. ***s2 = 11*** ise *bge* buyruğu gibi davranacak, yani ***rs1_deger >= rs2_deger*** koşulu doğruysa dallanacak. Assembly formatı: ***sec.dallan rs1, rs2, imm, s2***

Özel buyruklar için ihtiyacınız yok fakat standart buyrukları incelemek isterseniz RISC-V dokümanına bakabilirsiniz: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[50 Puan] RISC-V Buyruk Kümesi LLVM Eklentisi

LLVM, birçok frontend dilden (C/C++, Swift, Rust, Julia vb.) ortak bir ara dile (LLVM IR) çevrim yapan ve bu ara dilden de birçok backende (x86, ARM, MIPS, RISC-V vb.) özel kod üretebilen modüler bir derleyici kütüphanesidir. (https://llvm.org) Siz de bu ödevde, LLVM'in sağladığı bu esnekliği kullanarak verilen özel buyrukları makine koduna çevirebilmek için LLVM'in RISC-V backendine, standart I, M, C, A, F, D, Q eklentileri dışında bir buyruk eklentisi ekleyeceksiniz. Buyruk tanımlamaları için *LLVM Target Description Language* kullanacaksınız.

Eklentiyi eklemeden önce gerekli bağlılıkları yüklemeniz ve kütüphaneyi doğru bir şekilde build etmeniz gerek.

Not: *Windows kullanıcısıysanız Ubuntu WSL (https://ubuntu.com/wsl) ya da VirtualBox (https://www.virtualbox.org) gibi bir program üzerinde sanal makine kullanmanızı ve ubuntu için olan adımları takip etmenizi tavsiye ederim.*

cmake ve *ninja* build araçlarını, ayrıca yoksa *gcc*, *g++* derleyicilerini ve *git*i, linux (örn. ubuntu) kullanıyorsanız terminali açın ve aşağıdaki gibi yükleyin:

```
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
```

macOS kullanıyorsanız aşağıdaki gibi yükleyin:

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

Not: *Bundan sonraki adımlar linux ve macOS sistemler için farketmiyor, aynı komutları kullanın.*

LLVM kütüphanesini github'dan klonlayın ve `1fdec59bffc11ae37eb51a1b9869f0696bfd5312` commit versiyonuna getirin (LLVM 11.1.0 versiyonu) (https://github.com/llvm/llvm-project/archive/refs/tags/llvmorg-11.1.0.zip adresinden de indirebilirsiniz):

```
git clone https://github.com/llvm/llvm-project
cd llvm-project
git checkout 1fdec59bffc11ae37eb51a1b9869f0696bfd5312
```

Kütüphaneyi elde ettikten sonra aşağıdaki gibi bir *build* klasörü oluşturun ve bu klasörün içine gerekli araçları;
clang –> C kodunu derlemek için
llc –> LLVM IR veya LLVM bytecode ara dilini derlemek için
llvm-objdump –> object koddan hex kodu çıktısını almak ve assembly karşılıklarını görmek için
RISC-V backendi için verilen konfigürasyonlar ile build edin:

```
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
```

Bu şekilde, ihtiyacımız olan *clang, llc ve llvm-objdump* programları *llvm-project/build/bin* dosya yolunda oluşmuş oldu. Burada daha hızlı build etmek için *ninja -j1* ifadesinde "1" yerine daha fazla iş parçacığı (thread) atayabilirsiniz. Maksimum atayabileceğiniz iş parçacığı sayısını, terminalde *nproc* komutu ile öğrenebilirsiniz.

Eğer kütüphane sorunsuz olarak build edildiyse bundan sonra eklentiyi ekleme işlemine geçebilirsiniz.

Eklentiyi eklemek için dosyalarda yapmanız gereken değişiklikler aşağıda anlatılmıştır. Eklenti ismi Türkçe karakter olmadan soyadınız olsun. Burada örnek olarak "kasirga" kullanılmıştır.

Buyruk eklentisini eklemeniz için gereken dosya değişiklikleri **llvm-project/llvm/lib/Target/RISCV** dosya yolunda yapılmalıdır. LLVM kütüphanesinde **llvm-project/llvm/lib/Target/RISCV** dosya yoluna gidin ve eklentideki buyrukları tanımlayacağınız **RISCVInstrInfo<Soyad>.td** dosyasını oluşturun:

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

RISCVInstrInfo.td dosyasının sonunda, oluşturduğunuz **RISCVInstrInfo<Soyad>.td** dosyasını aşağıdaki gibi bir satır ekleyerek dahil edin:

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

RISCV.td dosyasında, eklentiyi aşağıdaki gibi tanımlayın (soyadınızı eklemek için değiştirmeniz gereken yerler kırmızı renk ile gösterilmiştir):

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

RISCVSubtarget.h dosyasında ise aşağıdaki gibi satırları ekleyin:

```
bool HasExtKasirga = false;

bool hasExtKasirga() const { return HasExtKasirga; }
```

Bundan sonra **RISCVInstrInfo<Soyad>.td** dosyasına özel buyrukları ekleyebilirsiniz. *0000000_rs2_rs1_000_rd_0000000* kodlamasına sahip örnek bir buyruğu aşağıdaki gibi ekleyebilirsiniz:

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
def ORNEK_BUYRUK : RVInst<(outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
    "ornek.buyruk", "$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 Opcode = RISCVOpcode<0b0000000>.Value;
}
```

Burada anlık değeri olan buyrukların anlık değerleri için *GPR* yazmaç tanımlaması yerine, *simm12*, *simm13_lsb0*, *simm21_lsb0_jal* gibi tanımlamalar kullanmalısınız. Diğer buyruk yapılarını örnekte gösterildiği gibi, **RISCVInstrInfo<Eklenti_İsmi>.td** dosyalarında benzer buyrukların nasıl çözüldüğüne bakarak ya da internette araştırarak ekleyebilirsiniz.

Buyrukları ekledikten sonra değişikliklerin olması için **llvm-project/build** klasörüne giderek kütüphaneyi aynı şekilde tekrar build etmeniz gerekiyor:

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

Bundan sonra eklediğiniz buyrukları içeren programları *clang* ve *llc* ile derleyebileceksiniz ve *llvm-objdump* ile, derlenmiş programlardaki *hex (on altılık taban) kodunu* görebileceksiniz.

Kütüphaneyi tekrar build ettikten sonra aşağıdaki *inline assembly (https://en.cppreference.com/w/c/language/asm)* olarak yazılmış C kodunu bir dosyaya kaydedin, *clang* ve *llc* ile programı derleyin ve *llvm-objdump* ile çıktıya bakın:

```
// ornek.c
asm ("tasi x5, x0, 17");
asm ("tasi x6, x0, 243");
asm ("sifrele x7, x6, 112");
asm ("carp.cikar x6, x5, x7");
asm ("kareal.topla x8, x6, x7");
asm ("bitsay x9, x8, 1");
asm ("bitsay x10, x8, 0");
asm ("ikikat.atla x15, 0x000f1e00");

asm ("0x001e3c00:\n"
                "carp.cikar x6, x6, x10\n"
                "sec.dallan x8, x6, 0x00000114, 2\n"
                "kareal.topla x6, x6, x10\n"
                "ikikat.atla x15, 0x000f1e00");

asm ("0x00000114:\n"
                "sec.dallan x1, x1, 0x00000114, 0");
```

Not: *Örnek C programı inline assembly olarak yazıldı fakat örneğin normal bir C kodunda CARP.CIKAR buyruğu için a * b - a gibi işlemlerin örüntülerini LLVM backendine daha fazla şey ekleyerek yakalayabilirdiniz, böylece assembly yazmaktan kurtulmuş olurdunuz.*

Derlerken *-mattr* flagi *+<soyad>* şeklinde olmalı:

```
## Gerekmiyor fakat assembly kodunu ayrıca görmek isterseniz aşağıdaki komutu
kullanabilirsiniz:
## C kodunu RISC-V Assembly koduna derleme
llvm-project/build/bin/clang -S -target riscv32 -O0 ornek.c -o ornek.s

## C kodunu LLVM IR koduna derleme
llvm-project/build/bin/clang -S -emit-llvm -target riscv32 -O0 ornek.c -o ornek.ll

## LLVM IR kodunu object koda derleme
llvm-project/build/bin/llc -mtriple=riscv32 -O0 -mattr=+kasirga -filetype=obj \
ornek.ll -o ornek.o

## Object koddan hex çıktısını alma
llvm-project/build/bin/llvm-objdump -d ornek.o
```

Örnek olarak oluşan *llvm-objdump* çıktısı aşağıdaki gibidir (Kırmızı renkli kısım ilgili buyruğun hex (on altılık taban) kodu, yeşil renkli kısım ise denk gelen buyrukların assembly karşılığını göstermektedir. Assembly kısmındaki sayılar ondalık (decimal) tabandadır.):

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

ornek.o:    file format elf32-littleriscv


Disassembly of section .text:

00000000 <.text>:
       0: f7 52 10 01   tasi    t0, zero, 17
       4: 77 53 30 0f   tasi    t1, zero, 243
       8: f7 43 03 07   sifrele t2, t1, 112
       c: 77 93 72 84   carp.cikar  t1, t0, t2
      10: 77 04 73 00   kareal.topla    s0, t1, t2
      14: f7 24 54 d5   bitsay  s1, s0, 1
      18: 77 25 54 55   bitsay  a0, s0, 0
      1c: ff 17 1f 60   ikikat.atla a5, 990720
      20: 77 13 a3 84   carp.cikar  t1, t1, a0
      24: 7f 7a 64 90   sec.dallan  s0, t1, 276, 2
      28: 77 03 a3 00   kareal.topla    t1, t1, a0
      2c: ff 17 1f 60   ikikat.atla a5, 990720
      30: 7f fa 10 10   sec.dallan  ra, ra, 276, 0
```

Buyrukları ekledikten sonra sizin derleyicinizde bu örnek program için hex kodunun bu şekilde gelip gelmediğini kontrol edebilirsiniz. Buyrukların hex (on altılık taban) kodunun küçüğü başta ve bayt adreslemeye uygun bayt bayt (8 bit 8 bit) geldiğini görüyorsunuz. Örneğin ilk *tasi* buyruğunun hex kodu ***f7_52_10_01*** şeklinde fakat aslında ***01_10_52_f7*** olarak çevirmelisiniz, bu şekilde doğru binary karşılığı olan ***000000010001_00000_101_00101_1110111*** karşılık geldiğini göreceksiniz. Bu programı 2. bölümde tasarlayacağınız işlemci üzerinde çalıştırarak işlemcinizi deneyebilirsiniz.

Not: *Verilog kısmında buyruk belleğiniz olmadığı ve program sayacı sembolik olduğu için program normalde girmesi gereken döngüye girmeyecek (tabii denerken hex kodunu direkt bu şekilde verirseniz öyle yoksa test-bench'te nasıl verdiğinize bağlı aslında) fakat bu ödev kapsamında önemli değil.*

[50 Puan] RISC-V Buyruk Kümesi Eklentili İşlemci Verilog Tasarımı

Verilen buyrukları gerçekleyen tek vuruşlu 32 bit işlemciyi Verilog donanım tanımlama dilinde davranışsal modelleme kullanarak yazın. Dosya ismi küçük harflerle ve Türkçe karakter olmadan <soyad>.v şeklinde soyadınız olsun. "<soyad>" modülünün giriş çıkışları Tablo 3'te görülebilir.

TABLO 3: *<soyad>* Modülünün Giriş ve Çıkış Sinyalleri

Sinyal	**Yön**	**Genişlik**	**Açıklama**
saat	Giriş	1 bit	Sıralı mantık için gerekli saat girişi.
reset	Giriş	1 bit	Devreyi başlangıç durumuna döndüren sinyal. *program_sayaci* ve *yazmaclar* sıfırlanmalı.
buyruk	Giriş	32 bit	İşlemcinin şu anki saat vuruşunda yürüteceği buyruk. ***Not:*** *Buyrukların bayt bayt 32 bitlik pakette geldiğini varsaymalısınız. Aşağıda daha detaylı açıklanmıştır.*
program_sayaci	Çıkış	32 bit	Bir sonraki çevrim getirilecek buyruğun adresi.
yazmaclar	Çıkış	32 adet 32 bit	32 bit genişliğe sahip 32 adet değerlerin saklandığı yazmaçlar.

İşlemciniz her saat vuruşunda buyruk sinyali ile gelen buyruğu çözmeli, *yazmaç değerlerini* ve *program sayacını* yapılacak hesaplamaya göre güncellemelidir. Devreniz, resetlendikten sonraki ilk çevrim "0" *program sayacı* değerindeyken ilk buyruk getirilmelidir. Yukarıda derleyici hex çıktısında da gösterildiği gibi, RISC-V'de bayt adresleme kullanıldığından ve buyruklar küçüğü başta (little endian) olduğundan gelen buyruklar önce uygun formata çevrildikten sonra çözülmelidir. Örneğin buyruk girişinden on altılık tabanda ***34_20_2F_73*** şeklinde bir buyruk geldiğinde, bayt bayt okunacak bir buyruk geldiği varsayılarak ***73_2F_20_34*** şekline dönüştürüldükten sonra çözülmelidir.

[Bonus][20 Puan] RISC-V Buyruk Kümesi Eklentili İşlemci Çizimi

Verilen buyrukları gerçekleyen tek vuruşlu 32 bit işlemciyi elle çizip denetim tablosunu oluşturun. <Soyad_Öğrenci Numarası>.pdf olarak kaydedin.

Ödev Teslimi (Son Teslim Tarihi: 21.02.2023 23.59)

1-) RISCV.td
2-) RISCVSubtarget.h
3-) RISCVInstrInfo.td
4-) RISCVInstrInfo<Soyad>.td
5-) <soyad>.v
Bonus-) <Soyad_Öğrenci Numarası>.pdf
dosyalarını ve varsa eklemek istediğiniz diğer dosyalarınızı sıkıştırın ve <Soyad_Öğrenci Numarası>.zip olarak kaydederek `https://uzak.etu.edu.tr`'ye yükleyin.