# İçindekiler

# 

[**İçindekiler 1**](#_fkco26ibm5qs)

[**Spike ile Cosim 3**](#_1vxfqaeayq2q)

[Giriş 3](#_bgm5hpdeg8xe)

[Cosim’den kastımız nedir, Amacımız 3](#_6ob8ev1v1j08)

[Spike nedir kısaca 3](#_iw0pqefnw001)

[Özetle neler yaptık 3](#_s6b8y2hmr6x)

[Cosim Verilog Tarafı 5](#_xz57kzowuhmy)

[cosim\_constants\_pkg: 6](#_uits8hcxba57)

[csr\_ids\_pkg: 6](#_100533qwom6l)

[cosim\_pkg: 6](#_vmdqeholw6bz)

[Fonksiyonlar 6](#_v69fz47tpzsj)

[Tür Tanımları 8](#_fo8i2js9x3wa)

[Cosim Kullanımı 12](#_ojrn98vodlxv)

[Verilator 13](#_9njkjuvj6ce3)

[Testbench’ten Örnek 16](#_cocnrjqjc4oz)

[args.txt Dosyası 19](#_br3mkw4ig53y)

[RISCV Proxy-Kernel 19](#_lkjltgtjwu7k)

[Baremetal Programı Sonlandırma 20](#_9x931zn0661a)

[Cosim C++ Tarafı 21](#_8ygsf3h9uyzg)

[Spike’ın Genel Akışı 21](#_2qgfqhx0cwr6)

[Kısaca 21](#_1w6ofxon4oz1)

[sim\_t Sınıfı 21](#_n1d689vpqg70)

[İlklendirme İşlemleri 22](#_wfbidgad7qx0)

[Host-OS ile İletişim Olmadan (idle Döngüsü) 23](#_2amu784xf5ds)

[Tohost-Idle-Fromhost Döngüsü 24](#_dksswas4ky9x)

[Spike’ta Yapılan Değişiklikler 25](#_nxq17ppvp9fj)

[Run’ı, ilklendirme ve döngü adımı olarak parçalamak 25](#_pu7ntturow0h)

[bool htif\_t::exit\_code\_not\_zero(); 25](#_igx6xdg69800)

[bool htif\_t::communication\_available(); 25](#_wcrdf7qnv27q)

[void htif\_t::single\_step\_without\_communication(); 25](#_c2mo7v5ga03u)

[void htif\_t::single\_step\_with\_communication(std::queue<reg\_t> \*fromhost\_queue, std::function<void(reg\_t)> fromhost\_callback); 26](#_z50f6igzzyke)

[void sim\_t::prerun(); 26](#_2p91ibp7n5jg)

[Idle’ın 5000 yerine 1 adım ilerleyen versiyonu 26](#_c4ufmxk2zyso)

[İşlemleri string’e çevirmeden yürütme 26](#_fwnqjd9ra6oh)

[Simülasyonun interactive moda girmesine engel olmak 27](#_yp1z9z4dts3d)

[Ctrlc ile interactive moda girme 27](#_z6krut8fgbdc)

[bool disable\_interactive\_mode parametreli sim\_t kurucusu 28](#_gyv20x85j52m)

[Bazı Değişikliklerin Spike’tan Ayrılamamasının Sebebi 28](#_5tm4x6ju8kz0)

[1- virtual function table 28](#_4cjfbmxqdg6l)

[2- volatile bool ctrlc\_pressed 28](#_79101r2voyt)

[Eklenen Fonksiyonlar 28](#_2ziggvhmcv72)

[void create\_sim\_with\_args(int argc, char\*\*argv) 28](#_jnmq1996bal1)

[argv\_argc\_t \*read\_args\_from\_file(const char \*filename) 29](#_vjulhxnmyeq3)

[void init() 29](#_bqo8z55ihbre)

[void step() 32](#_n8tmfrj77wv7)

[svBit simulation\_completed() 32](#_ajuw0ae449vp)

[void get\_pc(svBitVecVal\* pc\_o, int processor\_i) 32](#_hol0vyduk51w)

[void get\_log\_reg\_write(svBitVecVal\* log\_reg\_write\_o, int\* inserted\_elements\_o, const int processor\_i) 33](#_ctga2ac3lkzg)

[void get\_log\_mem\_read(svBitVecVal\* log\_mem\_read\_o, int\* inserted\_elements\_o, const int processor\_i) 35](#_aino9q781yhh)

[void get\_log\_mem\_write(svBitVecVal\* log\_mem\_write\_o, int\* inserted\_elements\_o, const int processor\_i) 36](#_iepti8o558e0)

[DPI ve Verilator 37](#_9j50u37zol5y)

[**Referanslar 38**](#_buxylja6734e)

# 

# Spike ile Cosim

[[1](#kix.905snbnvfjc7)]

## Giriş

### Cosim’den kastımız nedir, Amacımız

Amacımız doğrulama aşamasında zaman kazanabilmek için tasarımımızın ve spike’ın simülasyonlarının ayrı ayrı tamamlanmasını bekleyip sonuçlarını karşılaştırmak yerine tasarımızın simülasyonunu ve spike’ı eşgüdümlü bir şekilde çalıştırarak karşılaştırma yapmak.

### Spike nedir kısaca

Spike bir RISCV ISA simülatörü, işlemcimizin yürüttüğü buyruklara karşı doğrulamasını yaparken kullandığımız referans model.

ISA simülatörü, buyruk kümesinin simüle edilmesini sağlar. Herhangi bir mikromimarinin benzetimi değildir. ISA simülatörü sayesinde bir buyruk işlemciye girdiğinde etkilerinin neler olacağını ve makinenin durumunu nasıl değiştireceğini gözlemleriz ve bunu kendi tasarımımızı doğrulamak ve onaylamak için kullanırız.

**Kullanımı: “**spike” bir terminal uygulamasıdır. “spike” uygulaması; komut satırı argümanı olarak verdiğimiz, riscv üzerinde koşmak için derlenmiş .elf (executable linkable format) dosyamızın içindeki buyrukları sırayla yürütür. Belirttiğimiz seçeneklere göre yürütme sırasında gerçekleşen durum değişikliklerini belirtilen dosyaya yazar.

### Özetle neler yaptık

SystemVerilog testbench’lerimizde kullanılabilecek bir arayüz tasarladık. Bu arayüz “SystemVerilog Direct Programming Interface” (DPI[[1]](#footnote-0)) ile Spike’ın fonksiyonlarından türettiğimiz fonksiyonlara erişim sağlıyor.

Bu arayüz hâlihazırda şunlardan ibaret:

**// cosim\_pkg**

**import "DPI-C" function void init();**

**import "DPI-C" function void step();**

**import "DPI-C" function bit simulation\_completed();**

**import "DPI-C" function void get\_log\_reg\_write(**

**output commit\_log\_reg\_item\_t log\_reg\_write\_o[CommitLogEntries],**

**output int inserted\_elements\_o,**

**input int processor\_id = 0**

**);**

**import "DPI-C" function void get\_log\_mem\_read(**

**output commit\_log\_mem\_item\_t log\_mem\_read\_o[CommitLogEntries],**

**output int inserted\_elements\_o,**

**input int processor\_id = 0**

**);**

**import "DPI-C" function void get\_log\_mem\_write(**

**output commit\_log\_mem\_item\_t log\_mem\_write\_o[CommitLogEntries],**

**output int inserted\_elements\_o,**

**input int processor\_id = 0**

**);**

**import "DPI-C" function void get\_pc(**

**output reg\_t pc\_o,**

**input int processor\_id = 0**

**);**

**init** bir dosyadan[[2]](#footnote-1) Spike’a verilecek komut satırı argümanlarını okur ve simülasyonu oluşturur.

**step** fonksiyonu simülasyonu bir adım[[3]](#footnote-2) ilerletmeye yarar.

**simulation\_completed** fonksiyonu, spike tarafında çalışan kodun çıkış sinyali üretip üretmediğini kontrol etmek için kullanılır. (bkz. [baremetal programı sonlandırma](#_9x931zn0661a))

**get\_log\_reg\_write** fonksiyonu simülasyonun son adımında register’l(ar)a yapılan yazma işlemlerini **log\_reg\_write\_o** dizisi içerisine yazar, eklenen eleman sayısını **inserted\_elements\_o**’ya yazar. **processor\_id** parametresi ile çok çekirdekli bir simülasyon yapıyorsak hangi çekirdeğin yazma/okuma işlemini kontrol etmek istediğimizi seçiyoruz.

**get\_log\_mem\_read** fonksiyonu benzer şekilde son adımda yapılan bellek okuma işlemlerini,

**get\_log\_mem\_write** fonksiyonu da belleğe yazma işlemlerini verilog tarafına çekmeye yarar.

**commit\_log\_mem\_item\_t** ve **commit\_log\_reg\_item\_t,** nereye ne yazılmış, nereden ne okunmuş gibi bilgileri tutan SystemVerilog **struct**’larıdır.

**get\_pc** fonksiyonu, processor\_id ile belirttiğimiz işlemcinin pc’sini okumaya yarar.

## Cosim Verilog Tarafı

Bu bölümde verilog tarafında cosim’in yapısını, fonksiyonlarını ve tanımlanan türleri anlatacağız.



Cosim’in verilog tarafını SystemVerilog package’leri şeklinde oluşturduk. Birkaç pakete ayrılmasının sebebi, **cosim\_pkg[[4]](#footnote-3)**’in **import**’landığında kullanan kişiyi ilgilendirmeyen fonksiyon, tür, sabitleri gösterip **import**’landığı yerde isim kalabalığı yapmasını istememem. **cosim\_pkg**’de sadece son kullanıcıyı ilgilendiren fonksiyon, tür ve sabitler var. Bir çeşit kapsülleme[[5]](#footnote-4).

Dosyaların isimlendirilmesindeki **\_0**, **\_1** gibi önekler dikkatinizi çekmiştir. Çok önemli bir nokta olmamakla birlikte kısaca açıklayalım:

lowRISC’e göre [[2](#z9uo3v1z57u8)] paket A, paket B’yi kullanıyorsa B, A’yı kullanmamalı. Ama normalde SystemVerilog’da böyle bir kısıt yok. Dolayısıyla verilator’de[[6]](#footnote-5) paketler arası bağımlılık ilişkisini[[7]](#footnote-6) otomatik olarak çözmek gibi bir kavram yok (modüller için olmasının aksine). Ben de burada verilator’e yardımcı olmak için paketleri bağımlılık ilişkisine göre isimlendirdim ki makefile’da paket dosyalarını dizinden çekerken doğru sırayla çeksin, derlenirken doğru sırayla derlesin, bir paketten diğerine atıf olduğunda kendisine atıfta bulunulan paketin henüz derlenmemiş olma durumu olmasın.

**Paketleri tek tek açıklamak gerekirse:**

### **cosim\_constants\_pkg**:

(**\_0cosim\_constants\_pkg.sv** dosyasında) Cosim’de kullanılan türlerin bit genişliklerini bulundurur.

### csr\_ids\_pkg:

(**\_1csr\_ids\_pkg.sv** dosyasında) CSR isimlerinin Spike tarafında hangi id’lerle (sayılarla) eşleştirildiği bilgisini taşıyan **csr\_id\_e** isimli bir **enum**’u içeriyor. Normalde bu enum’un **cosim\_pkg**’de bulunması gerekiyordu, ayrı dosyada olmasının tek sebebi çok uzun olması. Bu enum, **cosim\_pkg**’da **typedef csr\_ids\_pkg::csr\_id\_e csr\_id\_e** şeklinde dışarıya (**cosim\_pkg**’ı importlayan tarafa) sunuluyor.

### cosim\_pkg:

(**cosim\_pkg.sv** dosyasında) Son kullanıcının **import**’laması gereken[[8]](#footnote-7) tek paket. Burada Spike ile etkileşimde bulunacak fonksiyonlar ve bu fonksiyonların çıktılarını kullanıma uygun şekilde sunmak için **struct** ve **enum**’ların tanımları var. Bu fonksiyon ve türleri tek tek açıklayalım.

#### Fonksiyonlar

**void init():** simülasyonu yürütmeye hazır bir vaziyette ilklendirmek için kullanılıyor. Spike’ın simülasyon oluşturması için normalde komut satırından verdiğimiz argümanları[[9]](#footnote-8), önceden belirtilmiş[[10]](#footnote-9) bir dosyadan okuyor.

Komut satırı argümanları ile ilgili ayrıca şunlara değinmek isterim, spike’ta seçenek olarak bırakılmış ama cosim’de seçenek olması mantıklı olmayan:

**-d**[[11]](#footnote-10): (interactive mod[[12]](#footnote-11)) devre dışı.

**--log-commits**[[13]](#footnote-12): her türlü aktif.

Bunlar kodun içinde macro[[14]](#footnote-13) tanımlamalarıyla kontrol ediliyor.

**void step():** simülasyonu bir “adım” ilerletir. Bu bir adım, bir buyruğun tamamlanması ve gerekli cihazların[[15]](#footnote-14) ilerletilmesinden oluşur. Buyruk yürütülmesi, çok işlemcili bir simülasyonda “sıradaki” işlemci tarafından yapılır. Spike’ta çok işlemcili bir simülasyonda sıradaki işlemci 5000 buyruk yürüttükten sonra sıra bir sonraki işlemciye devredilir.

**function void get\_log\_reg\_write(**

**output commit\_log\_reg\_item\_t log\_reg\_write\_o[CommitLogEntries],**

**output int inserted\_elements\_o,**

**input int processor\_id = 0**

**);**

**function void get\_log\_mem\_read(**

**output commit\_log\_mem\_item\_t log\_mem\_read\_o[CommitLogEntries],**

**output int inserted\_elements\_o,**

**input int processor\_id = 0**

**);**

**function void get\_log\_mem\_write(**

**output commit\_log\_mem\_item\_t log\_mem\_write\_o[CommitLogEntries],**

**output int inserted\_elements\_o,**

**input int processor\_id = 0**

**);**

Bu fonksiyonlar Spike tarafında en son adımda ne olup bitmiş takip etmek için kullanılıyor.

**get\_log\_reg\_write**, simülasyonun son adımında yürütülen buyruk hangi register(lar)a ne yazmışsa **log\_reg\_write\_o** dizisi içine yazıyor. **get\_log\_mem\_read** bellek okuma, **get\_log\_mem\_write** da bellek yazma işlemlerini sırasıyla **log\_mem\_read\_o** ve **log\_mem\_write\_o** dizilerine yazıyor. Fonksiyonlar, eklenen eleman sayısını **inserted\_elements\_o**’ya yazar. Dizilerin “unpacked” boyutunun uzunluğunu, fonksiyonlardan dinamik diziler[[16]](#footnote-15) döndüremediğim için **CommitLogEntries[[17]](#footnote-16)** şeklinde bir sabit olarak tanımladım.

**function bit simulation\_completed();** Bu fonksiyon,

**htif\_t::exitcode != 0**[[18]](#footnote-17) mı diye kontrol etmeye yarıyor. Bu koşulu kontrol ederek testbench’imizi sonlandırmaya karar verebiliriz.

NOT: **htif\_t::exitcode != 0** olması için Spike’ta koşan kodun (test girdimizin) bellekteki belli allanlara belli değerler yazması gerekiyor. Normalde bu olay proxy-kernel[[19]](#footnote-18) tarafından hallediliyor. Spike üzerinde baremetal kod çalıştırırken bu işlemin nasıl yapılacağına [baremetal programı sonlandırma bölümü](#_9x931zn0661a)nde bahsedeceğiz.

#### Tür Tanımları

C++ tarafından SystemVerilog tarafına çektiğimiz simülasyona dair kayıtları uygun şekilde temsil edebilmek için yapılan tür tanımları.

**typedef struct packed {**

**reg\_t addr;**

**reg\_t wdata;**

**bit [55:0] reserved; // c tarafindaki alignment'a uydurmak icin**

**byte len;**

**} commit\_log\_mem\_item\_t;**

Spike tarafında bir işlemcinin en son adımda yaptığı bellek okuma/yazma işlemleri, her işlemci (**processor\_t** türünden nesne) için **state\_t[[20]](#footnote-19)** türünden **state** isimli alanı içerisindeki **commit\_log\_mem\_t** türünden **log\_mem\_write** ve **log\_mem\_read** alanlarında kayıt altında tutulur. **commit\_log\_mem\_t**[[21]](#footnote-20) türü aslında bir listedir[[22]](#footnote-21). Bu listenin her bir elemanı[[23]](#footnote-22) 1/2/4/8 byte’lık okuma/yazma işlemini tutar. Verilog tarafında da bu listenin elemanlarını saklamak için tanımladığımız tür **commit\_log\_mem\_item\_t**’dır.

Burada bit aralıkları:

**addr**: 191:128 **wdata**:127:64 **reserved**: 63:8 **len**: 7:0

Okuma işleminde **wdata** alanı 0 olarak kalır. Okuma ve yazma işlemleri için **len** alanı; 1 (byte okuma/yazma), 2 (half), 4 (word), 8(double word) değerlerinden birini alır. Reserved alanı, c tarafındaki 64 bit hizalamaya uydurmak için bulunmakta.

**typedef struct packed {**

**reg\_key\_t key;**

**freg\_t value;**

**} commit\_log\_reg\_item\_t;**

Spike tarafında işlemcinin en son adımda yaptığı register yazma işlemleri, her işlemci (**processor\_t** türünden nesne) için **state\_t** türünden **state** alanı içerisindeki **commit\_log\_reg\_t** türünden **log\_reg\_write** alanında kayıt altında tutulur. **commit\_log\_reg\_t** aslında bir sözlüktür[[24]](#footnote-23). Bu sözlüğün elemanları olan anahtar-değer çiftlerinde anahtar hangi register’a yazıldığını, değer de ne yazıldığını ifade eder. Yukarıdaki **commit\_log\_reg\_item\_t** türü de bu anahtar değer çiftlerini verilog tarafında saklamak için tanımlandı.

Burada bit aralıkları: **key**: 191:128 **value**: 127:0

Spike tarafındaki register yazma kayıtlarında, register isimlerinin kodlamasını okunabilir şekilde verilog tarafında aktarabilmek için **reg\_key\_t** tanımlandı.

**typedef struct packed {**

**reg\_id\_t reg\_id;**

**reg\_key\_type\_e reg\_type;**

**} reg\_key\_t;**

**reg\_type**: 3:0 bitler. burada **reg\_key\_type\_e** hangi tür register olduğu bilgisini (integer, float, vector, csr) tutan bir **enum**.

**reg\_id**: 63:4 bitler. Bir türe ait hangi register olduğu bilgisini tutar. integer’ın hangi register’ı, float’ın hangi register’ı gibi. Burada **reg\_id\_t** bir **union**. integer, float ve vector register’lar için düz sayı[[25]](#footnote-24); csr’lar için spike’ta makrolarla kodlandığı[[26]](#footnote-25) şekilde bir **enum**.

Burada kullanılan **reg\_id\_t**, **reg\_key\_type\_e** ve **csr\_id\_e** türleri de şu şekilde tanımlanır:

**typedef enum bit [REG\_KEY\_TYPE\_W-1:0] {**

**XREG = 'b0000,**

**FREG = 'b0001,**

**VREG = 'b0010,**

**VREG\_HINT = 'b0011,**

**CSR = 'b0100**

**} reg\_key\_type\_e**

**REG\_KEY\_TYPE\_W** = 4, **REG\_KEY\_ID\_W** = 60, bu değerler spike’ta register yazma işlemlerini kaydederken[[27]](#footnote-26) anahtar-değer ikililerinde anahtarın oluşturulma şeklinden geliyor.

**// csr'larin ozel id'leri var**

**// diger id'ler duz 0'dan 31'e.**

**typedef union packed {**

**bit [REG\_KEY\_ID\_W-1:0] xr\_fr\_vr\_id;**

**csr\_id\_e csr\_id;**

**} reg\_id\_t;**

**typedef enum bit [REG\_KEY\_ID\_W-1:0] {**

**// riscv/encoding.h**

**CSR\_FFLAGS = REG\_KEY\_ID\_W'('h1),**

**CSR\_FRM = REG\_KEY\_ID\_W'('h2),**

**CSR\_FCSR = REG\_KEY\_ID\_W'('h3),**

**CSR\_VSTART = REG\_KEY\_ID\_W'('h8),**

**CSR\_VXSAT = REG\_KEY\_ID\_W'('h9),**

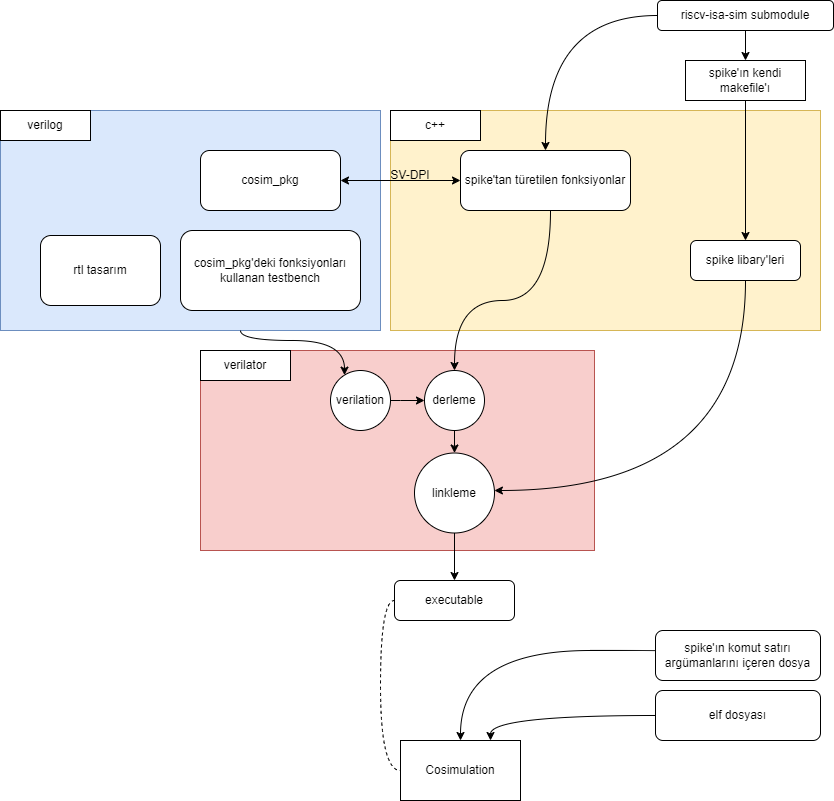
**CSR\_VXRM = REG\_KEY\_ID\_W'('ha),**

**CSR\_VCSR = REG\_KEY\_ID\_W'('hf),**

**CSR\_SEED = REG\_KEY\_ID\_W'('h15),**

**// devam ediyor**

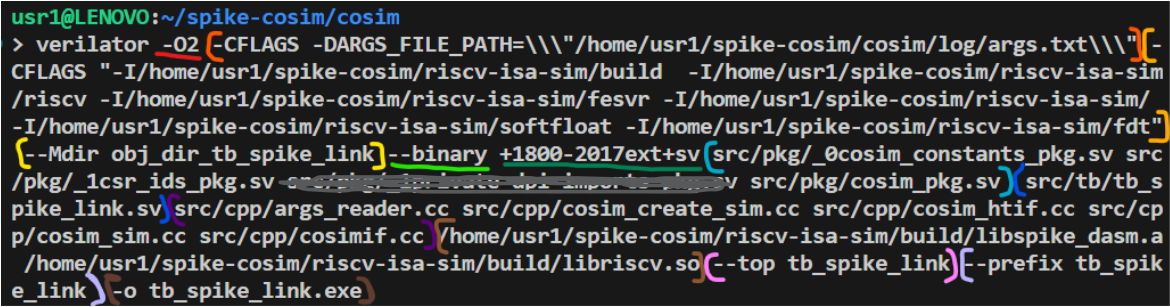
## Cosim Kullanımı

Özet: Riscv-isa-sim submodule’ünü derleyin. verilator ile **ARGS\_FILE\_PATH** makrosunu tanımlayarak ve gcc include path’ine gerekli eklemeleri[[28]](#footnote-27) yaparak; **riscv-isa-sim/build/libriscv.so** ve **riscv-isa-sim/build/libdisasm.a** library’lerini, **cosim/src/cpp** ve **cosim/src/pkg**’yi, rtl tasarımınızı ve **cosim\_pkg** fonksiyonlarını kullandığınız testbench’i derleyin/link’leyin. (spike-cosim reposunda bu işlemi örnek bir testbench için yapan bir makefile[[29]](#footnote-28) mevcut)

**ARGS\_FILE\_PATH** makrosu ile gösterilen dosyanın[[30]](#footnote-29) içindeki komut satırı argümanlarını koşmak istediğiniz .elf dosyasını gösterecek şekilde değiştirin. Verilator’un oluşturduğu executable’ı çalıştırın. Örnek kullanım için detaylı talimatlar README.md[[31]](#footnote-30)de mevcut. Burada verilator’un kullanımıyla ilgili kısaca bahsedeceğiz.

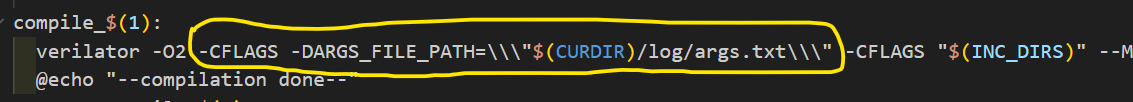
### Verilator

Aşağıda makefile[[32]](#footnote-31) içindeki derleme kuralını çalıştırdığımda çalışan komutu açıklıyorum:



●● optimizasyon seviyesi (verilator’den gcc’ye gönderiliyor)

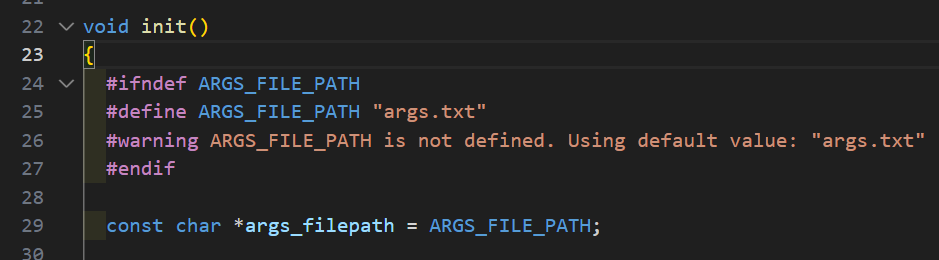
●● **-CFLAG** ile verilator’e gcc’ye göndermesi gereken flag’leri/ayarları/seçenekleri belirtiyoruz. Bu turuncunun içindeki kullanımda **-ASRGS\_FILE\_PATH** makrosunu tanımladığımız tek bir flag gönderiyoruz. Bu makronun, makefile[[33]](#footnote-32) içerisinde **CURDIR[[34]](#footnote-33)** değişkenini kullanarak makefile’ın konumuna göre hesaplanmasının daha evrensel olduğunu düşündüğüm için derleme kuralına bu şekilde ekledim.

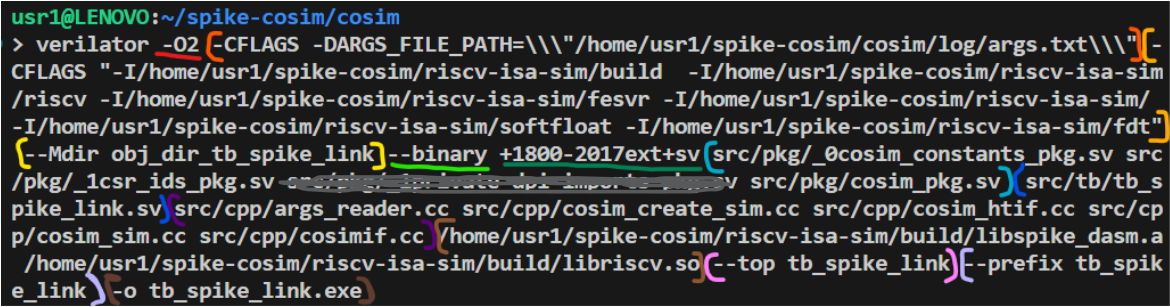


Yazım şeklindeki gariplik (tekrar eden “**\**”lar) şundan kaynaklanıyor:

shell, komutları parse ederken tırnak (**”**) işaretlerini ve ters eğik çizgiyi (**\**) kaldırıyor. Shell bu tırnak işaretlerini çağrılan executable’a tırnak işareti şeklinde aktarsın diye **\”** şeklinde veriyoruz. Burada iki seviye[[35]](#footnote-34) parse’lama olduğu için ters eğik çizgileri korumak için onları da **\\** olarak veriyoruz.

**ARGS\_FILE\_PATH**’i derleme komutunda tanımlamaya alternatif olarak cosimif.cc[[36]](#footnote-35)de kullanıldığı yerde de tanımlanabilir.





●● Burada yine verilator’un **-CFLAGS** flag’iyle gcc’ye include path’lerini gönderiyoruz. **-CFLAGS**’in buradaki kullanımında gcc’ye birden fazla flag gönceriyoruz. Onun için “flag1 flag2” şeklinde tırnak içerisinde sıralıyoruz. Spike’tan türettiğimiz fonksiyonlar, spike’taki header’ları **#include**’layarak spike’taki fonksiyonları kullanıyor.

●● **--Mdir** flag’ini, verilator’e ürettiği çıktıları koyacağı dizini bildirmek için kullanıyoruz. Dizin mevcut değilse oluşturuluyor. Varsayılan olarak bu dizinin ismi **obj\_dir**

●● --binary executable bir çıktı oluştur demek. --binary yerine kullanabileceğimiz seçeneklerden bazıları şunlar:

--cc: verilog kodlarını c++’a dönüştür.

--sc: verilog kodlarını systemC’ye dönüştür.

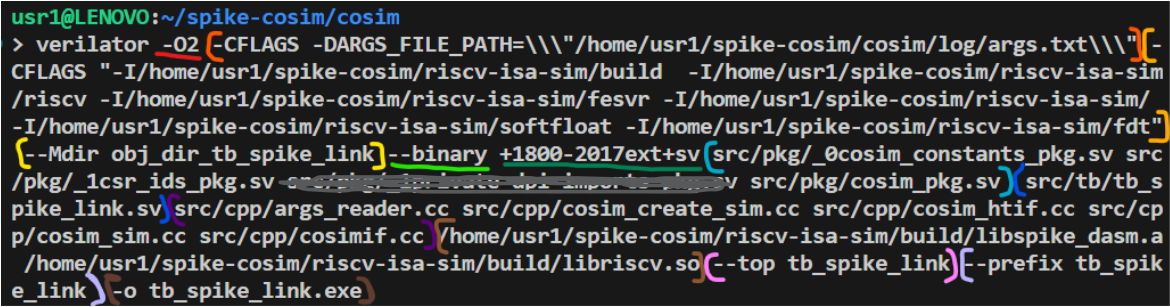
--lint-only: verilog kodlarının içerisinde uyarıya [[4](#kix.8zgyft1dwq4c)] sebep olabilecek herhangi bir nokta var mı kontrol et.

Tamamı için şuraya bakabilirsiniz: [[5](#2vctjswabn8g)]

●● burada sv uzantılı dosyaları 1800-2017 verilog standartına göre okumasını belirtiyoruz.

●● SystemVerilog paketlerini barındıran dosyalar. Verilator’de A paketini kullanan paket/modülü derlerken Paket A’nın, komut satırı argümanı olarak daha önce yazılması gerekiyor. Paket A, paket B’yi kullanıyorsa B’nin tanımlandığı dosyayı A’nınkinden önce yazmamız gerekiyor.

●● **cosim\_pkg**’deki fonksiyonları kullandığım bir testbench.

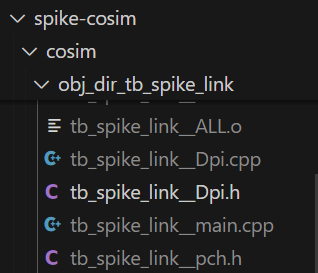


●● **cosim\_pkg**’de SV-DPI ile importlanan fonksiyonların ve bazı yardımcı fonksiyonların tanımlandığı c++ dosyaları.[[37]](#footnote-36)

●● Spike’ın cosim için değiştirilmiş hâlinin derlenmesiyle oluşturulmuş library’lerden ikisi[[38]](#footnote-37)

●● çıktı oluşturulacak top modül olarak **tb\_spike\_link**[[39]](#footnote-38)modülünü seçiyoruz. Aslında benim örneğimde sadece bir tane top modül adayı[[40]](#footnote-39) (**tb\_spike\_link**) mevcut olduğu için bu şekilde belirtmeme gerek yoktu. Fakat birden fazla top modül adayı olduğunda eğer **--top** ile hangi top modül için çıktı oluşturulmasını istediğimizi belirtmezsek verilator bütün top modül adayları için gerekli çıktıları[[41]](#footnote-40) oluşturur.

●● **--prefix** flag’i ile verilator’un oluşturduğu çıktılarda yapacağı isimlendirme için kullanacağı ön eki belirtiyoruz. Aşağıdaki resimde bu çıktılardan bazılarının isimleri mevcut:



●● **-o** flag’i ile oluşturulacak çalıştırılabilir dosyanın ismini belirtiyoruz.

### Testbench’ten Örnek

Bu bölümde testbench içerisinde cosim\_pkg fonksiyon ve türlerini nasıl kullandığımıza dair örnek mevcut.

Modülümüzün içinde **cosim\_pkg**’nin bütün tanımlarını **import**’luyoruz.

**import cosim\_pkg::\*;**

Simülasyon döngüsü boyunca Spike tarafından çektiğimiz verileri tutmak için 3 tane dizi tanımlıyoruz.

**commit\_log\_reg\_item\_t log\_reg\_write\_from\_c [CommitLogEntries];**

**commit\_log\_mem\_item\_t log\_mem\_read\_from\_c [CommitLogEntries];**

**commit\_log\_mem\_item\_t log\_mem\_write\_from\_c [CommitLogEntries];**

**int num\_elements\_inserted\_from\_c\_side; // 3'u icin de kullaniliyor.**

Simülasyon döngüsü içerisinde kullanılacak bazı değişkenler.

**reg\_t temp\_key;**

**freg\_t temp\_value;**

**reg\_t temp\_pc;**

İlk önce **cosim\_pkg**’de tanımlanan **init()** fonksiyonunu çağırmamız gerekiyor.

**initial begin: cosimulation**

**init();**

**// bu arada simulation\_loop olacak**

**$finish;**

**end: cosimulation**

Simülasyon döngüsü genel hatlarıyla.

**for (;;) begin: simulation\_loop**

**if (simulation\_completed()) begin // htif\_t::exitcode != 0**

**$display("simulation completed");**

**break;**

**end**

**get\_pc(temp\_pc);**

**$display("pc before execution: %0h", temp\_pc);**

**step();**

**// kontroller, karsilastirmalar.**

**wait\_key();**

**end: simulation\_loop**

**cosim\_pkg**’de tanımlanan **bit simulation\_completed()** fonksiyonu ile spike tarafında simülsayonun tamamlanıp tamamlanmadığını kontrol ederek simülasyon döngüsünü kırmaya karar verebiliriz.

**void get\_pc(output reg\_t pc\_o, input int processor\_id = 0)** ile **temp\_pc** değişkeninin içine o anki pc’yi yazıyoruz.

**void step()** fonksiyonuyla spike tarafında bir adım ilerlemesini söylüyoruz.

**wait\_key** yine DPI ile c++ tarafından importlanıyor, cosim’in kritik bir parçası değil. Sadece bir adım ilerledikten sonra ekrana basılan çıktıları takip edebilmek için yürütmeyi bekletmeye yarıyor.

Spike tarafından veri çekme. RTL’imizin oluşturduğu çıktılarla karşılaştırma burada yapılabilir.

**get\_log\_reg\_write(log\_reg\_write\_from\_c, num\_elements\_inserted\_from\_c\_side);**

**for (int ii = 0; ii < num\_elements\_inserted\_from\_c\_side; ii = ii + 1) begin: log\_reg\_write\_itr**

**$display("log\_reg\_write\_from\_c[%0d] reg\_type: %0s",**

**ii, log\_reg\_write\_from\_c[ii].key.reg\_type.name);**

**if (log\_reg\_write\_from\_c[ii].key.reg\_type == CSR) begin**

**$display("log\_reg\_write\_from\_c[%0d] csr name: %0s",**

**ii, log\_reg\_write\_from\_c[ii].key.reg\_id.csr\_id.name);**

**end else begin**

**$display("log\_reg\_write\_from\_c[%0d] reg\_id: %0d",**

**ii, log\_reg\_write\_from\_c[ii].key.reg\_id);**

**end**

**$display("log\_reg\_write\_from\_c[%0d].value: %0h", ii, log\_reg\_write\_from\_c[ii].value);**

**end**

**get\_log\_reg\_write** ile simülasyon döngüsü içerisinde register yazma olaylarına göz attığımız bir kod parçası.

**get\_log\_mem\_read(log\_mem\_read\_from\_c, num\_elements\_inserted\_from\_c\_side);**

**for (int ii = 0; ii < num\_elements\_inserted\_from\_c\_side; ii = ii + 1) begin: log\_mem\_read\_itr**

**$display("log\_mem\_read\_from\_c[%0d].addr: %0h", ii, log\_mem\_read\_from\_c[ii].addr);**

**$display("log\_mem\_read\_from\_c[%0d].wdata: %0h", ii, log\_mem\_read\_from\_c[ii].wdata);**

**$display("log\_mem\_read\_from\_c[%0d].len: %0d", ii, log\_mem\_read\_from\_c[ii].len);**

**end**

Bellek okuma olaylarına bakıyoruz.

**get\_log\_mem\_write(log\_mem\_write\_from\_c, num\_elements\_inserted\_from\_c\_side);**

**for (int ii = 0; ii < num\_elements\_inserted\_from\_c\_side; ii = ii + 1) begin: log\_mem\_write\_itr**

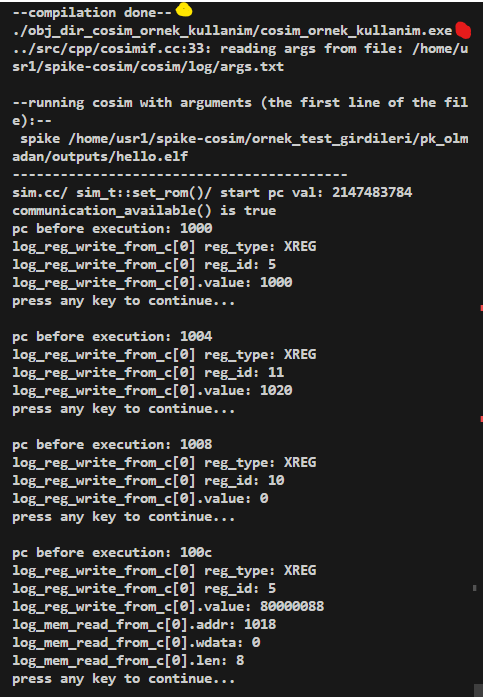
**$display("log\_mem\_write\_from\_c[%0d].addr: %0h", ii, log\_mem\_write\_from\_c[ii].addr);**

**$display("log\_mem\_write\_from\_c[%0d].wdata: %0h", ii, log\_mem\_write\_from\_c[ii].wdata);**

**$display("log\_mem\_write\_from\_c[%0d].len: %0h", ii, log\_mem\_write\_from\_c[ii].len);**

**end**

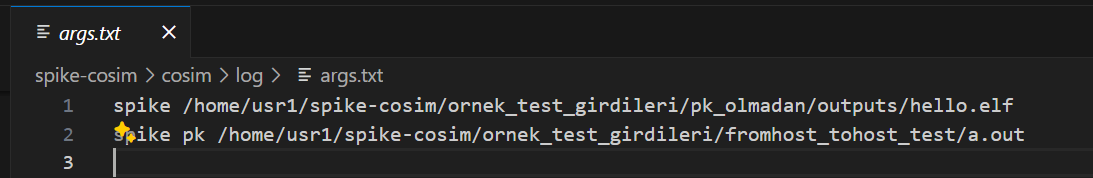
Bellek yazma olaylarına bakıyoruz.

Bu testbench’i (systemVerilog paketleri ve gerekli c++ kaynak dosyaları ve kütüphanelerle birlikte) verilator ile derleyip oluşan çıktıyı çalıştırdığımızda yandaki çıktıyı oluşturur. Sarıyla işaretlenen makefile’ın içinden **@echo** komutuyla yapılan bildirim. Kırmızı işaretlenen, makefile’ın executable dosyayı çalıştırdığı komut. 

### 

### args.txt Dosyası

Daha önceki bölümlerde bahsedildiği gibi, bu dosya spike’ın komut satırı argümanlarını yazdığımız **void cosim\_pkg::init()**[[42]](#footnote-41) fonksiyonu tarafından okunan dosya. Aşağıdaki resimde dosya içeriği iki satırdan oluşuyor. Sadece ilk satırı okunuyor. İkinci satırı örnekler arasında hızlı geçiş yapabilmek için ekledim.



Bu dosyanın yolu cosim’in örnek testbench için hazırlanmış makefile’ındaki[[43]](#footnote-42) derleme kuralında verilator’e **-CFLAGS -DARGS\_FILE\_PATH=\\\"$(CURDIR)[[44]](#footnote-43)/log/args.txt\\\"**

olarak ondan da gcc’ye **-DARGS\_FILE\_PATH=\"<dosya yolu>\"** olarak veriliyor. İstenilirse cosimif.cc[[45]](#footnote-44)de

**#ifndef ARGS\_FILE\_PATH**

**#define ARGS\_FILE\_PATH "args.txt"**

**#warning ARGS\_FILE\_PATH is not defined. Using default value: "args.txt"**

**#endif**

Şu satırlar değiştirilerek girilebilir.

### RISCV Proxy-Kernel

Spike üzerinde her zaman baremetal kod çalıştırmak istemeyebiliriz. Bazen printf, fopen, fgets, fclose, scanf, fprintf, fscanf gibi işletim sistemi kullanan fonksiyonları da kullanmamız gerekebilir. Gelgelelim spike, üzerinde işletim sistemi koşabilecek bir simülatör değil. Sadece processor, ns16550[[46]](#footnote-45), plic, clint, remote\_bitbang gibi bazı temel cihazlar modellenmiş. O yüzden işletim sistemi gerektiren fonksiyon çağrıları yapmak istediğimizde bu çağrılar aslında host-os’e yönlendiriliyor. Bu çağrılar proxy-kernel tarafından host-os’in anlayacağı bir dile çevriliyor ve bellekte belli alanlara yazılıyor. Daha sonra Spike’ın **fesvr** (front end server) kütüphanesi içerisindeki **htif\_t**[[47]](#footnote-46)sınıfı, proxy-kernel’in, host-os’in anlayacağı dile çevirdiği bu çağrıları host-os’e iletiyor ve gerekli işlemler host-os tarafından yapılıyor.

riscv proxy-kernel, riscv simülatörleri üzerinde koşar. riscv için derlenir (riscv-gnu-toolchain ile). Spike üzerinde koşarız. Sistem çağrıları yaptığımız bir kodu derleyip spike üzerinde koşmak için

**spike [diger komut satiri argumanlari] pk dosya.elf** şeklinde çalıştırırız.

Burada aslında spike’a elf dosyası olarak pk’yi vermiş oluyoruz. Pk’ye de girdi olarak **dosya.elf**’i vermiş oluyoruz. Spike pk’yi çalıştırmaya başladığında pk, **dosya.elf**’i belleğe map’liyor, ve kontrolü **dosya.elf**’e devrediyor.

**dosya.elf** içerisinde sistem çağrısı yaptığımız yerlerde kontrol bizim kodumuzdan pk’ye geçiyor. pk, bellekte **magic\_mem**[[48]](#footnote-47) isimli global array’e gerekli mesajı yazıyor ve **magic\_mem**’in adresini bellekte **tohost[[49]](#footnote-48)** isimli global değişkene yazıyor. Daha sonra spike’ın host-target arayüzü, **tohost**’un sıfırdan farklı bir değer aldığını görüyor. Onu okuyup yerine sıfır yazıyor. Sonra **tohost**’tan okuduğu değere göre sistem çağrılarını oluşturuluyor.

#### Baremetal Programı Sonlandırma

Cosim\_pkg>[Fonksiyonlar](#_v69fz47tpzsj) bölümünde simülasyonun Spike tarafında tamamlandığını kontrol etmek için kullandığımız **cosim\_pkg::simulation\_completed** fonksiyonunu anlatırken **htif\_t::exitcode**’un sıfırdan farklı bir değer almasından, bunun için de bellekte belli alanlara belli değerler yazılması gerektiğinden bahsetmiştik. Spike’ı Riscv-pk ile kullanırken[[50]](#footnote-49) test girdimizde[[51]](#footnote-50) programın sonlandığı kısımda belleğin belli alanlarına belli değerler yazarak programın sonlandığını host-target-interface’e bildirme işlemini proxy-kernel hallediyor. Dolayısıyla, spike üzerinde baremetal kod çalıştırırken host-target-interface’e programı sonlandırma mesajını iletecek işlemleri kendimiz yapmıyorsak **cosim\_pkg::simulation\_completed** fonksiyonu hiç bir zaman **true** (= 1) döndürmeyecek.

htif’e bu çıkış mesajını iletmek için baremetal kodumuzda şunları yapmamız gerekiyor:

Şu üç alanı tanımlıyoruz:

**volatile static long long int magic\_mem[8]; // bunun ismi onemli degil.**

**volatile static long long int\* tohost \_\_attribute\_\_((used));// bunlarin**

**volatile static long long int\* fromhost \_\_attribute\_\_((used));// onemli**

Bu alanlar üzerinde aşağıdaki işlemleri yapan fonksiyonu tanımlıyoruz:

**void baremetal\_exit(long long int exit\_code){**

**magic\_mem[1] = exit\_code;**

**magic\_mem[0] = 93; // 93: exit see riscv-isa-sim/fesvr/syscall.cc } syscall\_t::syscall\_t**

**for (int i = 2; i < 8; i++)**

**magic\_mem[i] = 0;**

**tohost = magic\_mem;**

**while (!fromhost);**

**fromhost = 0;**

**while (1);**

**}**

**baremetal\_exit** fonksiyonunu, test girdimizde[[52]](#footnote-51) programı sonlandırmak istediğimiz yerde çağırarak htif’e programı sonlandırma mesajı göndermiş oluyoruz ve **htif\_t::exitcode != 0**[[53]](#footnote-52) koşulu gerçekleşmiş oluyor.

## Cosim C++ Tarafı

### Spike’ın Genel Akışı

Bu bölümde Spike’ın nasıl çalıştığını önce kısaca anlatıp daha sonra spike üzerinde yaptığımız değişikliklere temel oluşturması için ayrıntılı bir şekilde anlatacağız.

#### Kısaca

**main** fonksiyonunda komut satırı argümanları çözümlendikten sonra bu argümanlarda belirttiğimiz şekilde bir simülasyon nesnesi oluşturulur.

Sonra bu simülasyon nesnesinin **run** metodunu çağırır. **run** metodunda ilk olarak test girdimiz olan .elf dosyası okunup spike’ta modellenen belleğe yüklenir. Daha sonra simülasyon döngüsü başlar. Simülasyon döngüsü; .elf dosyamızda **fromhost** ve **tohost**[[54]](#footnote-53) denilen iletişim kanalları tanımlanmışsa target[[55]](#footnote-54) ile host[[56]](#footnote-55) arasında iletişim[[57]](#footnote-56) kurulacak şekilde, bu kanallar tanımlı değilse host ve target arasında iletişim kurulmadan gerçekleşir.

#### sim\_t Sınıfı

**sim\_t**[[58]](#footnote-57) Spike’ta modellenen cihazların bulunduğu kısımdır. Bu cihazlar şunlardan ibarettir: **processor\_t** sınıfından nesne(ler), **ns16550\_t**, **plic\_t** ve **clint\_t** türünden birer tane nesne ve komut satırı argümanlarında belirtilmişse **remote\_bitbang\_t** türünden bir nesne ve bellek(ler). Bu cihazlar arasındaki iletişim **memory-mapped-io** şeklinde gerçekleşir. **sim\_t** sınıfı, **htif\_t**[[59]](#footnote-58) sınıfını kalıtır[[60]](#footnote-59). **htif\_t** temel olarak host ile target arasındaki iletişimi sağlamakla görevlidir.

#### İlklendirme İşlemleri

**main[[61]](#footnote-60)** fonksiyonunda komut satırı argümanları çözümlenip belirtilen konfigürasyona[[62]](#footnote-61) göre bir simülasyon nesnesi oluşturuluyor. Daha sonra bu nesnenin **run**  metodunu çağırıyor.

**NOT:** “**sınıf\_ismi::metod\_ismi**” gösteriminde bahsi geçen metod, aksi belirtilmediği sürece (metod **static** olarak tanımlanmadığı sürece veya metin içerisinde açıkça sınıf metodu olarak belirtilmediği sürece) nesne metodudur. Yani bir nesne üzerinden çağrılan, bir nesneye iliştirilmiş bir metod. “... nesnesinin ... metodu” gibi bir ifadede metod yine nesne metodudur.

**sim\_t::run**  metodu, **sim\_t::set\_procs\_debug** metodu ile processor(ler)e işlem kayıtlarının tutulup tutulmayacağı bilgisini gönderir. Sonra **htif\_t::set\_expected\_xlen** metodu ile **htif\_t** nesnesine .elf dosyasında hedef mimarinin belirtildiği bölümde bulunması gereken xlen’i**[[63]](#footnote-62)** gönderir. Daha sonra da **htif\_t::run[[64]](#footnote-63)** metodunu çağırır.

Burada **htif\_t::run** metodu aslında bir **sim\_t** nesnesi üzerinden çağrıldığı için mevzubahis **htif\_t** nesnesi gerektiğinde polimorfizm[[65]](#footnote-64) ile bir **sim\_t** nesnesi olarak muamele görebilir.

**htif\_t::run** metodu, önce **htif\_t::start** metodunu çağırır. Bu metod komut satırı argümanlarında belirtilen .elf dosyasını simülasyonun bellek alanına map’leyip[[66]](#footnote-65) bu .elf dosyasındaki sembol[[67]](#footnote-66) isimlerini ve program giriş adresini kaydettikten sonra **virtual void htif\_t::reset()** metodunu çağırır.

**reset** metodu, **htif\_t**’de **virtual**[[68]](#footnote-67) olarak tanımlandığı ve **htif\_t**’yi kalıtan **sim\_t**’de **override**[[69]](#footnote-68) edildiği ve mevzubahis **htif\_t** nesnesi, “**sim\_t** ile polimorfik” olduğu için **reset** metoduna yapılan çağrı aslında **sim\_t**’nin **reset** metoduna yapılan bir çağrı oluyor.

**sim\_t::reset**’te pc’nin **DEFAULT\_RSTVEC[[70]](#footnote-69)** değerinden .elf entry değerine zıplamasını sağlayan boot kodlarını bus’ta[[71]](#footnote-70) **boot\_rom** isimlendirilen böylege **rom\_device\_t[[72]](#footnote-71)** olarak ekleyen **sim\_t::set\_rom** metodu çağrılır.

**htif\_t::run**’da, **htif\_t::start**’tın tamamlanmasından[[73]](#footnote-72) sonra simülasyon döngüsü dediğimiz aşama başlayacaktır. Bundan önce simülasyon döngüsünde host-target mesajlaşmasında host’tan gelen mesajları tutan, **fromhost\_queue** denilen bir kuyruk ve bu kuyruğa mesaj eklemek için **fromhost\_calback** denilen bir callback[[74]](#footnote-73) oluşturulur.

**auto enq\_func = [](std::queue<reg\_t>\* q, uint64\_t x) { q->push(x); };**

**std::queue<reg\_t> fromhost\_queue;**

**// !!! fromhost\_calback bir std::function. reg\_t parametre alip void donduruyor.**

**// asagida**

**// enq\_func’in ilk parametresi fromhost\_queue’ya baglaniyor.**

**// 2. parametresi fromhost\_callback’in 1. (\_1’den dolayi) parametresi icin bir placeholder**

**std::function<void(reg\_t)> fromhost\_callback =**

**std::bind(enq\_func, &fromhost\_queue, std::placeholders::\_1);**

**// !!! yani fromhost\_callback, fromhost\_queue'ye**

**// reg\_t turunden bir seyler push'lamaya yariyor.**

**// fromhost\_callback(16) ⇔ (&fromhost\_queue)->push(16)**

Daha sonra simülasyon döngüsü başlar. Bu döngü iki farklı şekilde işleyebilir:

1- host-os ile iletişim olmadan (idle döngüsü)

2- host-os ile iletişim hâlinde (tohost-idle-fromhost döngüsü)

#### Host-OS ile İletişim Olmadan (idle Döngüsü)

Spike’a girdi olarak verdiğimiz .elf dosyasında fromhost ve tohost[[75]](#footnote-74) sembolleri mevcut değilse diğer bir deyişle .elf dosyasını oluşturmak için derlediğimiz kodda fromhost ve tohost alanları tanımlı değilse[[76]](#footnote-75) simülasyon döngüsü; spike’ta koşan kodumuz ve spike processor’lerinin (target), host-os (host) ile iletişimi (sistem çağrıları) olmadan gerçekleşir. Bu döngüde sadece simülasyon nesnesinin içerisinde modellenen cihazlar (başlıca processor’ler ve bellek) çalışmaktadır.

htif\_t::run() metodu içinde,

**if (tohost\_addr == 0) {**

**while (!signal\_exit)**

**idle();**

**}**

Kod parçası, bu döngüden sorumludur. Burada **htif\_t::idle**, **virtual** olarak tanımlanan bir metoddur. Mevzubahis **htif\_t** nesnesi, polimorfik bir **sim\_t** nesnesi olduğu için ve **sim\_t**’de **idle override** edildiği için **htif\_t::idle**’a yapılan çağrı aslında **sim\_t::idle**’a yapılan bir çağrıdır.

**sim\_t::idle**, eğer komut satırı argümanı olarak belirtilmişse veya ctrl+c ile spike’ı kesmişsek simülasyonu interactive modda çalıştırır. Aksi takdirde (hızlı mod) simülasyonu **sim\_t::step** metodu ile 5000 adım ilerletir. Ve her 5000 adımda diğer simülasyon cihazlarından gerekli olanlar birer adım ilerletilir (ns16550, plic, clint)

#### Tohost-Idle-Fromhost Döngüsü

Spike’a girdi olarak verdiğimiz .elf dosayasında fromhost ve tohost sembolleri tanımlıysa simülasyon döngüsü bu şekilde ilerler. Bu döngü üç fazın arka arkaya tekrar etmesinen ibarettir.

1- tohost

2- idle

3- fromhost

Tohost fazında spike’ın host-target arayüzü, belleğin tohost bölgesine (64 bitlik bir alan, aslında bir pointer) sıfırdan farklı bir değer yazılmış mı diye kontrol eder, yazıldıysa o değeri okuyup yerine sıfır yazar. Tohost’tan okuduğu değer için gerekli komutları oluşturur, ve [ilkendirme işlemleri bölümü](#_wfbidgad7qx0)nde bahsettiğimiz fromhost\_callback ile birlikte front end server’daki araçlara[[77]](#footnote-76) gönderir. Fromhost\_callback, host-os’in gerçekleştirdiği işlemlerin spike işlemcisine geri döndürülmesi gereken cevapları host-target arayüzündeki fromhost\_queue’ye ekleyebilmeleri için vardır.

Idle fazı, [idle döngüsü bölümü](#_2amu784xf5ds)ndekiyle aynı işler.

Fromhost bölümünde htif, fromhost\_queue’ya front-end server’daki diğer araçlar tarafından bir şey eklenmiş mi diye bakar. Eklenmişse eklenen şeyi spike işlemcisinde koşan kodun okuyabilmesi için bellekteki fromhost bölgesine yazar.

Pekiyi, idle fazında 5000 simülasyon adımı yürütüldüğünü söylemiştik. Aşağıdaki gibi bir durum oluşursa ne olacak?

Spike’ta koşan kodumuz bir sistem çağrısı yapmış ve proxy-kernel’in ilgili fonksiyonları da bu sistem çağrısını bellekte gerekli yerlere yazmış olsun. Simülasyon tarafında bu olayların olması 500 buyruk sürdüğünü varsayalım. yani 500 simülasyon adımında tamamlanmış olsunlar. Kalan 4500 adımda ne olacak?

Burada pk’in ilgili fonksiyonu fromhost bölgesini dinlemektedir. Fromhost’a bir şey yazılıncaya kadar boş bir döngüde takılı vaziyette bekler.

### Spike’ta Yapılan Değişiklikler

Spike’taki değişiklikler genel olarak

- sim\_t::run ve htif\_t::run metodunu ilklendirme ve döngü adımı olarak parçalamak

- sim\_t::idle metodunun 5000 adım yerine 1 adım ilerleyen eşdeğerini tanımlamak

- işlemcinin yürütme sırasında işlem kayıtlarını output stream’e string’e çevrilerek eklemesini iptal etmek

- simülasyonun interactive moda girmeyecek bir şekilde oluşturulmasını sağlamak

Çevresinde şekillendi.

Başta spike’ı yeniden derlemeye gerek olmasın, tanımlanacak fonksiyonların sadece imzalarını spike’ın header’larına ekleyelim ve fonksiyon tanımları tamamen ayrı dosyalarda bulunsun ve sadece o dosyaları derlemeye gerek olsun şeklinde istemiştik. Fakat daha sonra teknik kısıtlardan [] dolayı bu mümkün olmadı, ve spike’ın bazı fonksiyon tanımları ve imzalarını içererek derlenmesini gerektirecek bir tasarım oldu.

#### Run’ı, ilklendirme ve döngü adımı olarak parçalamak

sim\_t::run, ufak bir ilklendirme işlemi yaparak htif\_t::run’ı çağırıyordu. htif\_t::run, spike’ta simülasyon (bkz [ilklendirme işlemleri](#_wfbidgad7qx0), [tohost-idle-fromhost döngüsü](#_dksswas4ky9x)) sınıfıyla host-os arasında bağlantı kurup simülasyon sınıfıyla idle metodu vasıtasıyla etkileşimde bulunarak simülasyonun yürütülmesinden sorumluydu. Fakat ilklendirme, döngü adımı, döngüyü kırma kontrolü gibi bizim testbench’ten ayrı ayrı çalıştırmak isteyeceğimiz işlemlerin hepsi bu run metodunun içerisinde yapılıyordu. Dolayısıyla run metodunu parçalamak gerekti. Bunun için aşağıdaki fonksiyon/metodlar tanımlandı.

##### bool htif\_t::exit\_code\_not\_zero();

Bu metodun imzası htif\_t sınıfının tanımında[[78]](#footnote-77) public olarak eklendi. Bu metodun tanımı **spike-cosim/cosim/ src/cpp/cosim\_htif.cc** dosyasındadır. Simülasyon döngüsünün tamamlandığı kontrolünü yapmaya yarayan metoddur. simulation\_completed [] tarafından kullanılmaktadır.

##### bool htif\_t::communication\_available();

Bu metodun imzası htif\_t sınıfının tanımında public olarak eklendi. Bu metodun tanımı **spike-cosim/cosim/ src/cpp/cosim\_htif.cc** dosyasındadır. Elf dosyasında fromhost -tohost sembollerinin tanımlı olup olmadığını, yani elf dosyasında koşan kod ile htif vasıtasıyla iletişimin mümkün olup olmadığını kontrol etmeye yarar. init’in [] içerisinde kullanılmaktadır.

##### void htif\_t::single\_step\_without\_communication();

Bu metodun imzası htif\_t sınıfının tanımında public olarak eklendi. Bu metodun tanımı **spike-cosim/cosim/src/cpp/cosim\_htif.cc** dosyasındadır. init, [] [communication\_available](#_wcrdf7qnv27q) metodu ile iletişimin mümkün olmadığını tespit ederse (bkz. [Idle döngüsü](#_2amu784xf5ds)) step\_callback[[79]](#footnote-78) olarak single\_step\_without\_communication’u seçer. single\_step\_without\_communication’un tek yaptığı şey, idle\_single\_step’i (bkz. [Idle’ın 5000 yerine …](#_c4ufmxk2zyso)) çağırmaktır.

##### void htif\_t::single\_step\_with\_communication(std::queue<reg\_t> \*fromhost\_queue, std::function<void(reg\_t)> fromhost\_callback);

Bu metodun imzası htif\_t sınıfının tanımında public olarak eklendi. Bu metodun tanımı **spike-cosim/cosim/src/cpp/cosim\_htif.cc** dosyasındadır. init, [] [communication\_available](#_wcrdf7qnv27q) metodu ile iletişimin mümkün olduğunu tespit ederse (bkz. [Tohost-idle-fromhost döngüsü](#_dksswas4ky9x)) step\_callback [] olarak single\_step\_with\_communication’u seçer. Single\_step\_with\_communication, htif\_t::run’daki tohost - idle - fromhost aşamalarını içeren bir metoddur. Yalnız idle yerine idle\_single\_step’i (bkz. [Idle’ın 5000 yerine …](#_c4ufmxk2zyso)) kullanır.

##### void sim\_t::prerun();

Bu metodun imzası sim\_t sınıfının tanımına public olarak eklendi. Bu metodun tanımı **spike-cosim/cosim/src/cpp/cosim\_sim.cc** dosyasındadır. [İlklendirme işlemlerinde](#_wfbidgad7qx0) bahsedilen set\_procs\_debug ve set\_expected\_xlen işlemlerini yapar. init [] tarafından çağrılır.

#### Idle’ın 5000 yerine 1 adım ilerleyen versiyonu

Spike’ta idle dediğimiz simülasyonun kendi kendine çalıştığı aşamayı temsil eden metod, simülasyonu 5000 adım ilerletmektedir. Yani 5000 buyruk ilerletilir. Fakat biz her bir buyruğun sonucunu görebilmek istiyoruz. Bu yüzden bu idle’ın tek adım ilerleten bir versiyonu olan idle\_single\_step metodu tanımlandı. Bunu tanımlarken spike’taki mekanizmadan sapmamak için aynı idle’ın kendisinde olduğu gibi, htif\_t’de protected virtual, sim\_t’de private override şeklinde tanımlandı.

#### İşlemleri string’e çevirmeden yürütme

Spike’ta işlemci tarafından yapılan işlemler, --log-commits komut satırı argümanıyla çalıştırdıysak her bir buyruk için kaydedilir. Cosim’de adım adım karşılaştırabilmek için bu değerlere ihtiyacımız var. Bu kayıt varsayılan olarak stderr dosyasına bastırılır. Fakat bu çıktı insan tarafından okunabilir string formatındadır. biz zaten cosim arayüzünde bu değerlere doğrudan sayısal olarak erişime sahip olduğumuz için[[80]](#footnote-79) bunların string’e dönüştürülüp stderr dosyasına bastırılması gereksiz bir angaryadır. Ayrıyeten bu çıktılar stderr ve stdout’a kendi bastırmak istediğimiz mesajların arasında gözükmektedir. Dolayısıyla cosim’de kullanılmak üzere şu fonksiyonlar tanımlanmıştır:

void processor\_t::step\_without\_print(size\_t n);[[81]](#footnote-80)

void sim\_t::step\_without\_print(size\_t n);[[82]](#footnote-81)

static inline reg\_t execute\_insn\_logged\_without\_print(processor\_t\* p, reg\_t pc, insn\_fetch\_t fetch);[[83]](#footnote-82)

Bunların spike’taki asıl muadillerinden tek farkı string’e dönüştürme ve log\_file’a bastırma işlemlerinin kaldırılmış olmasıdır.

#### Simülasyonun interactive moda girmesine engel olmak

Daha önceki bölümlerde spike’ın interactive modunu cosim’de kullanmanın gereksiz olacağından bahsetmiştik. Interactive mod’un kullanılabilir veya kullanılamaz olmasını spike-cosim/cosim/src/cpp/cosim\_conf.h dosyasındaki DISABLE\_INTERACTIVE\_MODE makro tanımıyla değiştirebiliriz. Eğer bu makro tanımlıysa, diğer bir deyişle interactive mode devre dışı bırakılmışsa, komut satırı argümanlarının çözümlendiği create\_sim\_with\_args [] fonksiyonunda -d (interactive mod’da çalıştıran komut satırı argümanı) seçeneği etkisizdir, uyarı mesajı verip devam eder. Yine eğer interactive mod devre dışı bırakılmışsa ctrl+c ile de interactive moda giremeyiz. Bu noktada ctrl+c ile interactive moda girme olayına bir parantez açalım.

##### Ctrlc ile interactive moda girme

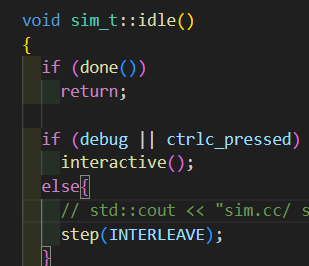
riscv-isa-sim/riscv/sim.cc dosyasında tanımlanan volatile bool ctrlc\_pressed değişkeni, simülasyon sınıfının interactive modda başlatılmamış hâlde de (-d komut satırı argümanı kullanılmadan) interactive mod’a geçebilmesini sağlayan mekanizmanın bir parçası.

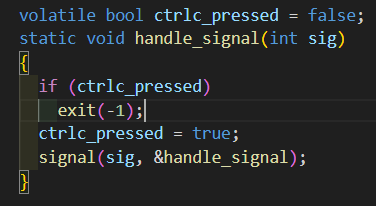
Ctrlc\_pressed’in okunduğu kısım:

sim\_t::idle, eğer ctrlc\_pressed ise interactive modu çalıştırıyor, ctrlc\_pressed değilse sim\_t::step(5000) yürütüyor.

Ctrlc\_pressed’e yazılan kısım:

handle\_signal diye bir fonksiyon’da (metod değil, nesneye veya sınıfa ait değil, global bir fonksiyon) ilk girişte true’ya çevriliyor (ctrl+c’ye bir kere basınca ctrlc\_pressed true oluyor) ikinci girişte (ctrlc\_pressed true iken girişte) process abort ediliyor.





Handle\_signal ise sim\_t constructor’ında[[84]](#footnote-83) spike process’ine gelen interrupt’larda ne yapılacağını bildirmek için signal diye bir sistem fonksiyonuna gönderiliyor.

**signal(SIGINT, &handle\_signal);**

##### 

##### bool disable\_interactive\_mode parametreli sim\_t kurucusu

Bu kurucu, disable\_interactive\_mode parametresine göre handle\_signal fonksiyonunun process’e gelen kesme sinyali ile ilgilenip ilgilenmeyeğinin seçimini yapıyor:

**if (!disable\_interactive)**

**signal(SIGINT, &handle\_signal);**

### Bazı Değişikliklerin Spike’tan Ayrılamamasının Sebebi

#### 1- virtual function table

Spike’ta yapılan bazı değişikliklerin spike’ın kaynak kodlarından ayrılamamasının sebebi, idle\_single\_step metodunu spike’taki mekanizmadan sapmamak için virtual-override şeklinde tanımlamak istememdi. Bu şekilde yapmak isteyince ayrı dosyaya koyamıyoruz, çünkü virtual function table[[85]](#footnote-84) sınıf tanımı yapılırken oluşturuluyor, yani spike derlenirken oluşturuluyor. Idle\_single\_step’in sadece imzasını header’lara (htif\_t ve sim\_t header’ına) ekleyip sim\_t::idle\_single\_step tanımını ayrı bir kaynak dosyasına koyup bu ayrı kaynak dosyasını derleyip spike library’lerine karşı linklesek bile virtual function table güncellenmiyor, sim\_t::idle\_single\_step kendi başına bir metod gibi duruyor. Dolayısıyla idle\_single\_step’in spike kaynak kodlarında bulunup spike’la beraber derlenmesi gerekiyordu. Bunun sonucu olarak da idle\_single\_step’in kullandığı diğer metodların da (execute\_insn\_logged\_without\_print, processor\_t::step\_without\_print, sim\_t::step\_without\_print) spike kaynak kodlarında tanımlanması gerekiyordu.

#### 2- volatile bool ctrlc\_pressed

[Ctrl+c ile interactive moda girme](#_z6krut8fgbdc) bölümünde bahsedilen volatile static bool ctrlc\_pressed ve void handle\_signal sim.cc dosyasında tanımlı oldukları için bool disable\_interactive mod parametreli sim\_t constructor’unun da sim.cc dosyasında bulunması gerekti. Çünkü handle\_signal fonksiyonunu kullanıyordu. Aslında bunun etrafından dolaşmak mümkündü, ama zaten spike’ın cosim fork’unun [virtual function table](#_4cjfbmxqdg6l) bölümünde bahsettiğimiz sebepten dolayı yeniden derlenmesi gerekecekti.

### Eklenen Fonksiyonlar

Bu bölümde bahsedilecek olan fonksiyonlar, spike-cosim/cosim/src/cpp dizini altındaki kaynak dosyalarında tanımlanmıştır.

#### void create\_sim\_with\_args(int argc, char\*\*argv)

Bu fonksiyon, argc ve argv argümanlarında verilen değerleri komut satırı argümanlarıymış gibi çözümler, bir simülasyon nesnesi oluşturup nesnenin pointer’ını döndürür. riscv-isa-sim/spike\_main/spike.cc dosyasındaki main’in sim\_t nesnesi oluşturan kısmına eşdeğer. spike-cosim/cosim/src/cpp/cosim\_create\_sim.cc dosyasında tanımlıdır. spike.cc dosyasında bulunan yardımcı fonksiyonlar da cosim\_create\_sim.cc dosyasına kopyalanmıştır. Bu fonksiyonun spike’ın main’inden farkları şöyle:

- Sim\_t nesnesini oluşturup run metodunu çağırmak yerine sim\_t nesnesini oluşturup pointer’ını döndürüyor. Bu pointer, init’te [] global bir değişkene kaydedilip diğer fonksiyonlar tarafından kullanılıyor.

- sim\_t nesnesinin pointer’ını döndürebilmek için, nesne stack’te değil heap’te oluşturuluyor. (stack’te oluşturulan bir nesnenin pointer’ını döndürürsek, fonksiyon döndükten sonra stack başka fonksiyonlara ayrılacağı için nesnenin üzerine yazarlar)

sim\_t <nesne\_ismi> (<constructor parametreleri>); şeklinde oluşturmak yerine “new”[[86]](#footnote-85) ile oluşturuluyor.

- main’in dump\_dts yaptığı[[87]](#footnote-86) yerde return 0 yerine exit(0) yapıyor.

- --log-commits komut satırı argümanı verilsin verilmesin işlem kaydının tutulması için bool log\_commits true olarak ayarlanıyor.

- sim\_t’nin “[simülasyonun interactive moda girmesine engel olmak](#_yp1z9z4dts3d)” bölümünde bahsettiğimiz kurucusunu kullanıyor.

- cfg\_t[[88]](#footnote-87) türünden nesne, stackte oluşturulmak yerine heap’te oluşturuluyor. Yani cfg\_t <nesne ismi>(<kurucu argümanları>) şeklinde değil cfg\_t\* cfg\_ptr = new cfg\_t(<kurucu argümanları>);

Ve #define cfg (\*cfg\_ptr) şeklinde oluşturuluyor.

#### argv\_argc\_t \*read\_args\_from\_file(const char \*filename)

Filename olarak verilen dosyanın ilk satırını okur, (birden fazla satır varsa görmezden gelir) argv\_argc\_t denilen int argc ve char \*\*argv alanlarını içeren bir struct pointer’ı döndürür. Daha sonra bu döndürülen değerler, init [] fonksiyonu tarafından create\_sim\_with\_args [] fonksiyonuna verilir.

#### void init()

(**NOT:** init’ten itibaren anlatılan fonksiyonların imzaları verilator tarafından verilog kodundaki DPI import ifadelerine göre otomatik oluşturulmuştur. Bu fonksiyonlar spike-cosim/cosim/src/cpp/cosimif.cc’de tanımlanmıştır.)

DPI ile import’lanan void cosim\_pkg::init() fonksiyonunun tanımlandığı kısımdır. Görevi [args.txt](#_br3mkw4ig53y) dosyasından spike’a verilecek komut satırı argümanlarını okuyup simülasyonu oluşturup diğer ilklendirme işlemlerini yaparak adım adım ilerletmeye hazır hâle getirmektir. Sırayla yaptığı işler şu şekildedir:

1- ARGS\_FILE\_PATH makrosuyla belirtilen dosyanın içeriğini [read\_args\_from\_file](#_vjulhxnmyeq3) fonksiyonu ile okur.

**#ifndef ARGS\_FILE\_PATH**

**#define ARGS\_FILE\_PATH "args.txt"**

**#warning ARGS\_FILE\_PATH is not defined. Using default value: "args.txt"**

**#endif**

**const char \*args\_filepath = ARGS\_FILE\_PATH;**

**printf(\_\_FILE\_\_ ":%d: reading args from file: %s\n", \_\_LINE\_\_, args\_filepath);**

**argv\_argc\_t \*argc\_argv = read\_args\_from\_file(args\_filepath);**

2- okunan argümanları [create\_sim\_with\_args](#_jnmq1996bal1) fonksiyonuna vererek create\_sim\_with\_args’ın döndürdüğü pointer’ı simulation\_object isimli global bir değişkene kaydeder. Daha sonra bu değişken, init fonksiyonu döndükten sonra da simülasyon nesnesine erişebilmek için kullanılacaktır.

**/\* global scope \*/**

**sim\_t \*simulation\_object;**

**/\* inside init function \*/**

**simulation\_object = create\_sim\_with\_args(argc\_argv->argc, argc\_argv->argv);**

**NOT:** c/c++ dillerinde[[89]](#footnote-88) global scope’ta değişken/nesne tanımlamanın, değişken/nesne’nin aynı dosyadaki bütün fonksiyonlar tarafından görülebiliyor olmasının yanında ima ettiği diğer bir anlam da şudur:

Değişken/nesne stack’te değil bss’te veya sıfırdan farklı ilk değer alıyorsa data segmentinde bulunur. Bu da fonksiyon çağrıları arasında değerinin korunmasını sağlamış olur. Stack’ta dursaydı, yani bir fonksiyona ait lokal ve static[[90]](#footnote-89) olmayan bir değişken/nesne olsaydı, o fonksiyon döndükten sonra çağrılan fonksiyonlar kendilerine ayırdıkları stack’te daha önceden bırakılmış bir değişken/nesne olduğunun farkında olmadan üzerine yazacaklardı.

**NOT:** burada simülasyon nesnesi create\_sim\_with\_args fonksiyonunun içerisinde “new” ile oluşturulduğu için heap’te bulunur. Bahsi geçen global scope kavramı “simulation\_object” pointer’ı için geçerlidir. Fakat heap’teki bir nesne de “delete” veya “free” kullanılarak işletim sistemine serbest bırakılması söylenene kadar data ve bss’te olanlar gibi fonksiyon çağrıları arasında korunur.

3- bu pointer üzerinden [prerun](#_2p91ibp7n5jg) ve sim\_t’nin kalıttığı htif\_t’nin start (bkz. [Ilklendirme işlemleri](#_wfbidgad7qx0))metodunu çağırır.

**simulation\_object->prerun();**

**((htif\_t\*)simulation\_object)->start();**

Yukarıda gördüğümüz ((htif\_t\*) simulation\_object) yazım şekli, sim\_t\* (sim\_t pointer’ı) türünden simulation\_object’i htif\_t\* (htif\_t pointer’ı) türüne “type-casting”[[91]](#footnote-90) yapmak için kullanılmıştır. Start metodu, bir htif\_t nesnesi üzerinden çağrılabildiğinden (çünkü htif\_t içerisinde tanımlı) bunu yapmak gerekiyor.

**NOT:** bir nesne pointer’ından başka nesne pointer’ına yapılan “type-casting”, sadece aşağıdan yukarıya[[92]](#footnote-91) veya yukarıdan aşağıya olduğu durumda yani htif\_t nesnesine olan bir pointer’ı sim\_t pointer’ına dönüştürüyorsak sadece mevzubahis htif\_t nesnesi gerçekten bir sim\_t nesnesinin “base” nesnesi ise yapılabilir. Yoksa runtime’da hata alırız.

4- htif::run’ın ilk kısımlarında yapılana benzer şekilde (bkz. [Ilklendirme işlemleri](#_wfbidgad7qx0)) global scope’ta tanımlanan fromhost\_queue’ya ekleme yapan bir fonksiyon oluşturulur ve global scope’ta tanımlanan fromhost\_callback’e kaydedilir. Burada fromhost\_queue ve fromhost\_callback’ın htif\_t::run’dakinin aksine lokal değil global scope’ta tanımlanmasının sebebi, init fonksiyonu döndükten sonra da kullanılabilir olmalarını istememizdir.

**/\* global scope \*/**

**std::queue<reg\_t> fromhost\_queue;**

**std::function<void(reg\_t)> fromhost\_callback;**

**/\* inside init function \*/**

**auto enq\_func = [](std::queue<reg\_t> \*q, uint64\_t x)**

**{ q->push(x); };**

**fromhost\_callback = std::bind(enq\_func, &fromhost\_queue, std::placeholders::\_1);**

**NOT:** c++ dilinde diğer nesne yönelimli dillerin aksine,

std::queue<reg\_t> fromhost\_queue;

ifadesi boş bir referans[[93]](#footnote-92) oluşturmak yerine aslında “varsayılan ilklendirilmiş” [] bir nesne oluşturmaktadır.

5- host ile target arasında iletişimin mümkün olup olmamasına (bkz. [communication\_available](#_wcrdf7qnv27q)) göre global’de tanımlanan step\_callback isimli callback belirlenir. Step\_callback, daha sonra step [] fonksiyonu tarafından çağrılacaktır.

Iletişim mümkünse single\_step\_with\_communication metodunun, birinci argümanı (hemen aşağıdaki [NOT](#g242dofqgf7n)’a bakınız) simulation\_object, ikincisi fromhost\_queue’nun adresi, üçüncü argümanı da fromhost\_callback’e bağlanmış hâli step\_callback olarak belirlenir.

Iletişim mümkün değilse single\_step\_without\_communication metodunun birinci argümanı (hemen aşağıdaki [NOT](#g242dofqgf7n)’a bakınız) simulation\_object’e bağlanmış hâli step\_callback olarak belirlenir.

**/\* global scope \*/**

**std::function<void()> step\_callback;**

**/\* inside init function \*/**

**if (((htif\_t\*)simulation\_object)->communication\_available())**

**{**

**printf("communication\_available() is true\n");**

**// htif\_t pointer'ine type-cast yapmaya gerek yoktu muhtemelen ama acik acik gostermek istedim**

**step\_callback = std::bind(&htif\_t::single\_step\_with\_communication, (htif\_t\*)simulation\_object, &fromhost\_queue, fromhost\_callback);**

**}**

**else**

**{**

**printf("communication\_available() is false\n");**

**step\_callback = std::bind(&htif\_t::single\_step\_without\_communication, (htif\_t\*)simulation\_object);**

**}**

**NOT:** nesne metodları aslında hangi nesne üzerine çağrıldıklarını söyleyen gizli bir argümana sahiptirler. Derleyici, “nesne1.metod1()” veya “nesne\_ptr1->metod1()” ifadesinde, metod1 isimli metodun gizli argümanına nesne1’in pointer’ını veya nesne\_ptr1’i geçer.

#### void step()

DPI ile importlanan void cosim\_pkg::step() fonksiyonunun tanımlandığı kısımdır. [Init](#_bqo8z55ihbre) tarafından belirlenen step\_callback’i çağırır.

**void step()**

**{**

**step\_callback();**

**}**

#### svBit simulation\_completed()

DPI ile importlanan bit cosim\_pkg::simulation\_completed() fonksiyonunun tanımlandığı kısımdır. [Init](#_bqo8z55ihbre) tarafından kaydedilen sim\_t pointer’i üzerinden sim\_t’nin kalıttığı htif\_t’nin exit\_code\_not\_zero metodunu çağırır, döndürdüğü değeri döndürür.

**svBit simulation\_completed()**

**{**

**return ((htif\_t\*)simulation\_object)->exitcode\_not\_zero();**

**}**

svBit, DPI’ın c/c++ header’ında SystemVerilog’un “bit” türüne eşdeğer olarak tanımlanmıştır.

#### void get\_pc(svBitVecVal\* pc\_o, int processor\_i)

DPI ile importlanan void cosim\_pkg::get\_pc(output reg\_t pc\_o, input int processor\_id = 0) fonksiyonunun tanımlandığı kısımdır. Simülasyon nesnesine [init](#_bqo8z55ihbre) tarafından kaydedilen global simulation\_object isimli pointer üzerinden erişerek simülasyonun processor\_i argümanı ile verilen processor’ünün pc’sini pc\_o pointer’ı ile gösterilen yere yazar.

**void get\_pc(svBitVecVal\* pc\_o, int processor\_i)**

**{**

**\*((reg\_t\*)pc\_o) = simulation\_object->get\_core(processor\_i)->get\_state()->pc;**

**}**

Burada svBitVecVal, DPI’ın c/c++ header’ında SystemVerilog’daki 32 bitlik 2-state (0-1) veriyi göstermek için tanımlanmış tür. Verilator’de [] (DPI ve Verilator kısmında daha detaylı anlatılacak) bir fonksiyon parametresi, 2-state bir packed/unpacked array ve output yönlü verilmişse c/c++ tarafına svBitVecVal pointer’ı olarak dönüştürülüyor. Ben burada (reg\_t\*) ifadesi ile pc\_o ile gösterilen alana 64 bitlik yazma yapacağımı bildiğimden pc\_o adlı pointer’ı 64 bitlik bir tür olan reg\_t türünü gösteren bir pointer’a dönüştürüyorum. Daha sonra “\*(pointer) =” ifadesiyle o pointer’ın gösterdiği yere bir değer yazıyorum. Bu pointer, SystemVerilog tarafında importlanan fonksiyonun çağrısının yapıldığı yerde output yönlü argümanı göstermekte (bu işi verilator hallediyor). Yani verilog tarafında şu şekilde çağrı yapıldığı zaman:

**reg\_t temp\_pc;**

**initial begin: cosimulation**

**init();**

**for (;;) begin: simulation\_loop**

**get\_pc(temp\_pc);**

C++ tarafında pc\_o isimli pointer, verilog tarafındaki temp\_pc adlı değişkeni gösterir durumda.

**NOT:** reg\_t türü, verilog ve c++ taraflarında ayrı ayrı tanımlanmıştır.

#### void get\_log\_reg\_write(svBitVecVal\* log\_reg\_write\_o, int\* inserted\_elements\_o, const int processor\_i)

DPI ile importlanan cosim\_pkg::get\_log\_reg\_write fonksiyonunun tanımlandığı kısımdır. processor\_i ile belirtilen işlemcinin son buyruk yürütmede yaptığı register yazma işlemlerini [init](#_bqo8z55ihbre) tarafından kaydedilen simulation\_object pointer’ı üzerinden erişerek log\_reg\_write\_o adlı pointer ile gösterilen alana yazar, kaç tane işlem eklenmişse inserted\_elements\_o adlı pointer ile gösterilen int türünden değişkene yazar.

Yaptığı işlemler şu şekilde:

**void get\_log\_reg\_write(svBitVecVal\* log\_reg\_write\_o, int\* inserted\_elements\_o, const int processor\_i)**

**{**

**auto map\_from\_c\_side = simulation\_object->get\_core(processor\_i)->get\_state()->log\_reg\_write;**

**int& num\_entries = \*inserted\_elements\_o;**

**num\_entries = 0;**

**auto item\_ptr = (commit\_log\_reg\_item\_t\*) log\_reg\_write\_o;**

**for (auto x: map\_from\_c\_side){**

**item\_ptr[num\_entries].key = x.first;**

**item\_ptr[num\_entries].value = x.second;**

**num\_entries++;**

**}**

**}**

Yine burada da verilator’un, verilog tarafındaki import ifadesinde output yönlü, 2-state, bu sefer “unpacked array of packed struct” olan log\_reg\_write\_o parametresini c++ tarafına svBitVecVal\* <parametre ismi> şeklinde dönüştürmesi söz konuzu. Yani packed olsun unpacked olsun, temsil edilen tür 32 bit olsun 64 olsun veya log\_reg\_write\_o’nun gösterdiği dizinin elemanları gibi 192 bit olsun, eğer output yönlü, 2-state ve c’de doğrudan karşılığı olmayan (bit[31:0] ve bit[63:0] da dahil) elemanlardan oluşuyorsa c/c++ tarafına svBitVecVal\* olarak dönüştürülüyor. Buna uygun muamele yapmak programcının sorumluluğuna bırakılıyor.

C’de karşılığı olan türler[[94]](#footnote-93) de output yönlüyse , <türün c tarafındaki karşılığı>\* <parametre ismi> olarak dönüştürülüyor.

**commit\_log\_reg\_item\_t log\_reg\_write\_from\_c [CommitLogEntries];**

**int num\_elements\_inserted\_from\_c\_side;**

**initial begin: cosimulation**

**init();**

**for (;;) begin: simulation\_loop**

**get\_log\_reg\_write(log\_reg\_write\_from\_c, num\_elements\_inserted\_from\_c\_side);**

Bu örnekte verilog tarafında dpi ile importlanan get\_log\_reg\_write fonksiyonu çağrıldığında c++ tarafındaki fonksiyonun log\_reg\_write\_o isimli pointer’ı verilog tarafında log\_reg\_write\_from\_c olarak geçilen argümanı gösterir. (bu işi verilator hallediyor.) benzer durum inserted\_elements\_o için de geçerli.

Fonksiyonu açıklamaya tekrar geri dönecek olursak:

“**auto map\_from\_c\_side =**” ifadesi ile “map\_from\_c\_side isimli değişkenin türünü sağ taraftaki ifadeden çıkar” demiş oluyoruz. (type inference []) sağ tarafta ise spike simülasyonunun ilgili processor’un state’inde tutulan son buyruk tarafından yapılan register yazma işlemlerine erişiyoruz.

**int& num\_entries = \*inserted\_elements\_o;** ifadesinde inserted\_elements\_o pointer’ının gösterdiği int türünden değişkene “referans” olacak num\_entries diye bir referans oluşturuyoruz.

C++’da referans, [] yazımı kolaylaştırmak için sürekli pointer “dereference” etmemek için ortaya çıkarılmış özel bir türdür. Tanımlandıkları yerde referans olacakları değişkenle eşleştirilecek şekilde ilklendirilmek zorundalardır.

<referans olunacak tür>& <referans olacak değişken ismi> = <referans olunacak değişken>;

Yazımıyla oluşturulurlar. Kullanımı şu şekildedir, referans’a yapılan her değişiklik referans olunan değişkeni etkiler. (ters yönde de geçerli)

Başka bir deyişle,

int& referans\_olacak\_degisken = referans\_olunacak\_degisken;

ile

#define referans\_olacak\_degisken (\*(&referans\_olunacak\_degisken))

Eşdeğer ifadeler. (ilk ifadede & sembolü, takip eden türün bir referans olduğu anlamına geliyor. Ikincisinde & sembolü özel bir unary operator, takip eden değişkenin adresini döndürüyor.)

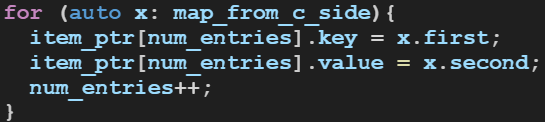
Ikisini de yazdıktan sonra

referans\_olacak\_degisken = <deger>;

Ifadesi, referans\_olunacak\_degisken’i etkileyecek.

**auto item\_ptr = (commit\_log\_reg\_item\_t \*) log\_reg\_write\_o;** ifadesinde log\_reg\_write\_o isimli pointer’ı commit\_log\_reg\_item\_t pointer’ına “type-cast” ediyoruz. Auto, sol taraftaki değişkenin türünü sağ taraftaki ifadeden çıkarmaya (type-inference []) yarıyor.

log\_reg\_write\_o, register yazma işlemlerini verilog tarafına aktarmak için içerisine veri yazacağımız diziyi gösteren bir pointer. Bu dizinin elemanları verilog tarafında commit\_log\_reg\_item\_t isimli bir türdür. (bkz. Cosim\_pkg > [tür tanımları](#_fo8i2js9x3wa)) Dolayısıyla bu diziye yazma yapmak için “dereference” edeceğim pointer’ın türü de commit\_log\_reg\_item\_t’nin c++ tarafındaki muadili olmalı. Çünkü hem yazma işlemini o türün bellekteki yerleşimine göre yapsın, hem pointer aritmetiğini [] doğru yapsın. Yani item\_ptr[num\_entries] ifadesini kullandığımda dizinin başını (yani ilk elemanın yerini) gösteren item\_ptr’ye (her bir elemanın boyutuna göre) doğru kayma miktarını eklesin.

Soldaki döngüde spike tarafından ulaştığımız register yazma işlemlerini içeren map’in tüm elemanlarını dönerek item\_ptr vasıtasıyla yazmamız gereken yere yazıyoruz. 

**NOT:** Burada değinmemiz gereken diğer bir kavram, c tarafından verilog tarafına doğrı bir şekilde yazma yapabilmek için **verilog tarafındaki ve c++ tarafındaki commit\_log\_reg\_item\_t türlerinin bellek yerleşimlerini** birbirine nasıl uydurduğumuzdur.

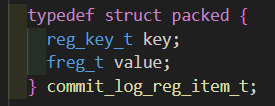
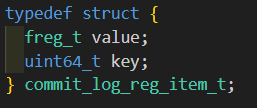
Verilog tarafında commit\_log\_reg\_item\_t bir packed struct. Verilog’da packed struct’ın alanları bellekte yerleşirken önce yazılan alan büyük adreste duruyor. C’de olanın tam tersi. Unpacked struct kullanabilseydim bir ayarlama yapmama gerek olmayacaktı:

*“A packed structure is a mechanism for subdividing a vector into subfields, which can be conveniently accessed as members. Consequently, a packed structure consists of bit fields, which are packed together in memory without gaps. An unpacked structure has an implementation-dependent packing, normally matching the C compiler”* SystemVerilog language standard 7.2.1 []

Fakat verilator, unpacked struct için pointer dereference ile doğrudan bir atama yapmayı desteklemiyor.

Dolayısıyla verilog tarafında packed struct’la devam ettim, c tarafında da verilog tarafındaki yerleşime benzetmek için struct’ın alanlarını sondan başa doğru yazdım:

Verilog tarafında: c++ tarafında:



#### void get\_log\_mem\_read(svBitVecVal\* log\_mem\_read\_o, int\* inserted\_elements\_o, const int processor\_i)

DPI ile importlanan cosim\_pkg::get\_log\_mem\_read fonksiyonunun tanımlandığı kısımdır. Processor\_i ile belirtilen işlemcinin son buyruk yürütmede yaptığı bellek okuma işlemlerini log\_mem\_read\_o ile gösterilen dizi’ye kaydeder. Kaç tane eleman eklenmişse inserted\_elements\_o ile gösterilen int türünden değişkene yazar. Bellek okuma işlemlerine init tarafından kaydedilen simulation\_object pointer’ı üzerinden erişir.

Şu şekilde çalışır:

**void get\_log\_mem\_read(svBitVecVal\* log\_mem\_read\_o, int\* inserted\_elements\_o, const int processor\_i){**

**auto mem\_read\_vector = simulation\_object->get\_core(processor\_i)->get\_state()->log\_mem\_read;**

**int& num\_entries = \*inserted\_elements\_o;**

**num\_entries = 0;**

**auto item\_ptr = (commit\_log\_mem\_item\_t\*) log\_mem\_read\_o;**

**for (auto x: mem\_read\_vector){**

**item\_ptr[num\_entries].addr = std::get<0>(x);**

**item\_ptr[num\_entries].wdata = std::get<1>(x);**

**item\_ptr[num\_entries].len = std::get<2>(x);**

**// item\_ptr[num\_entries].reserved = {0, 0, 0, 0, 0, 0, 0};**

**num\_entries++;**

**}**

**}**

“auto mem\_read\_vector = …” ile spike tarafındaki bellek okuma işlemlerine erişiyoruz. (ayrıntılı anlatımı için [get\_log\_reg\_write](#_ctga2ac3lkzg) bölümüne bakınız.)

“int& num\_entries = ...” ile inserted\_elements\_o pointer’ının gösterdiği int’e bir referans oluşturuyoruz.

**commit\_log\_mem\_item\_t log\_mem\_read\_from\_c [CommitLogEntries];**

**int num\_elements\_inserted\_from\_c\_side;**

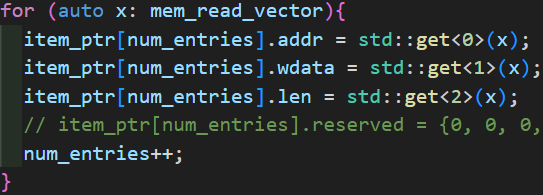
**initial begin: cosimulation**

**init();**

**for (;;) begin: simulation\_loop**

**get\_log\_mem\_read(log\_mem\_read\_from\_c, num\_elements\_inserted\_from\_c\_side);**

C++ tarafında log\_mem\_read\_o pointer’ı ile gösterilen dizi, verilog tarafında fonksiyon çağrısında fonksiyonun output yönlü parametresine verilen (bu örnekte log\_mem\_read\_from\_c) argümandır. Bu diziye uygun yerleşimde/formatta erişmek için log\_mem\_read\_o pointer’ını verilog tarafındaki commit\_log\_mem\_item\_t türüne eşdeğer türün pointer’ına type-casting yapıyoruz:

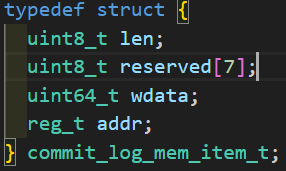
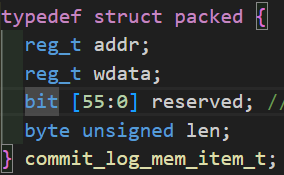
“auto item\_ptr = (commit\_log\_mem\_item\_t\*) log\_mem\_read\_o;”

Yandaki döngü ile spike’tan eriştiğimiz mem\_read\_vector’deki elemanları tek tek gezerek item\_ptr ile hedef alana yazıyoruz. std::get<0>(x) ifadesi, tupe’ın 0. Elemanına erişmek anlamına geliyor.

Reserved alanına 0 yapmak zaten verilog tarafında okunmayacağı için gereksizdi.

**NOT:** [get\_log\_reg\_write](#_ctga2ac3lkzg)’ın son kısmında da bahsettiğimiz gibi c tarafından verilog tarafına doğru bir şekilde yazma yapabilmek için verilog tarafında tanımlanan commit\_log\_mem\_item\_t türü ile bunun c tarafındaki eşdeğerinin bellek yerleşimlerinin birbirine uygun olması gerekiyor. Bunun için:

Verilog tarafında: c++ tarafında:



Şeklinde türler tanımlandı. Struct’ların alanlarının sırasının ters olması get\_log\_reg\_write’ın son kısmında not’ta bahsettiğimiz sebepten dolayı. Verilog tarafındaki struct’ta len ile wdata arasında 56 bitlik boşluk bırakılmasının sebebi, c++ tarafında wdata’nın (uint8\_t reserved[7] alanını koymasam da) 64 bit hizalanmasına uydurmak için. C++ tarafında reserved’in bulunması sadece açık açık göstermek için.

#### void get\_log\_mem\_write(svBitVecVal\* log\_mem\_write\_o, int\* inserted\_elements\_o, const int processor\_i)

DPI ile importlanan cosim\_pkg::get\_log\_mem\_write fonksiyonunun tanımlandığı kısımdır. Log\_mem\_read ile tek farkı, onun okuma işlemlerini spike’tan çekip verilog tarafına aktarmak için yaptıklarını bu fonksiyon yazma işlemlerini spike’tan çekip verilog tarafına aktarmak için yapıyor.

## DPI ve Verilator

# Referanslar

[1]: Spike-cosim reposu, şuradan erişilebilir:

<https://github.com/farukyld/spike-cosim>

[2]: lowRISC package cyclic dependency, şuradan erişilebilir:

<https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md#package-dependencies:~:text=there%20must%20not%20be%20any%20cyclic%20dependencies>

[3]: Wikipedia kapsülleme, şuradan erişilebilir:

<https://tr.wikipedia.org/wiki/Kaps%C3%BClleme>

[4]: Verilator warnings, şuradan erişilebilir:

<https://verilator.org/guide/latest/warnings.html>

[5]: verilator kullanım çeşitleri, şuradan erişilebilir:

<https://verilator.org/guide/latest/verilating.html#:~:text=Verilator%20may%20be%20used%20in%20five%20major%20ways%3A>

<https://github.com/farukyld/spike-cosim/issues/1>

C++ unordered\_map, şuradan erişilebilir:

<https://en.cppreference.com/w/cpp/container/unordered_map>

Wikipedia associative array, şuradan erişilebilir:

<https://en.wikipedia.org/wiki/Associative_array#:~:text=empty%20associative%20array.-,Example,-%5Bedit%5D>

Wikipedia callback, şuradan erişilebilir:

<https://en.wikipedia.org/wiki/Callback_(computer_programming)>

Wikipedia virtual method table, şuradan erişilebilir:

<https://en.wikipedia.org/wiki/Virtual_method_table>

C++ default initialization, şuradan erişilebilir:

<https://en.cppreference.com/w/cpp/language/default_initialization>

Wikipedia type inference, şuradan erişilebilir:

<https://en.wikipedia.org/wiki/Type_inference#:~:text=Type%20inference%20is%20the%20ability,type%20annotations%20having%20been%20given>.

1. SystemVerilog’un bir parçası. SystemVerilog ve c/c++ arasında fonksiyon alışverişinin nasıl olacağını (“**import**” ve “**export**”) belirten bir standart. Sentez araçları tarafından bu standarta göre gerekli bağlantıyı sağlayacak derleme/sentez yapılıyor. Bkz. [DPI ve Verilator](#_9j50u37zol5y) [↑](#footnote-ref-0)
2. Bu dosyanın konumu **spike-cosim/cosim/src/cpp/cosimif.cc** kodunun içerisinde **ARGS\_FILE\_PATH** makrosu ile varsayılan bir değere tanımlanmıştır. Buradan elle değiştirilebilir. **spike-cosim/cosim/makefile** içerisindeki örnek testbench için derleme kurallarında **spike-cosim/log/args.txt**’yi kullanacak şekilde ayarlanmaktadır. Bkz. [args.txt dosyası](#_br3mkw4ig53y) [↑](#footnote-ref-1)
3. Simülasyon adımı. Spike tarafında sıradaki işlemcinin (Spike’ta çekirdekler sırayla ilerletiliyor. Sıradaki çekirdek 5000 adım ilerledikten sonra sıra sonraki çekirdeğe geçiyor.) bir buyruk yürütmesi ve Spike’ta canlandırılan diğer cihazlardan (ns16550, plic, clint, varsa remote\_bitbang) bir adım ilerletilmesi. [↑](#footnote-ref-2)
4. Cosim’i kullanacak testbench’in **import**’layacağı paket [↑](#footnote-ref-3)
5. *“herhangi bir nesnenin metotlarını, verilerini ve değiskenlerini diğer nesnelerden saklayarak ve bunlara erişimini sınırlandırarak yanlış kullanımlardan koruyan bir konsepttir*.” [[3](#ywcveppd4oz3)] [↑](#footnote-ref-4)
6. Simülasyon yapmak ve DPI bağlantısını gerçeklemek için kullandığımız araç. Verilog kodlarını c++’a çevirip gcc ile derleme ve link’leme işlemlerini otomatikleştiriyor. Son çıktı olarak bir executable oluşturuyor, bunu koşarak simülasyonu çalıştırmış oluyoruz. [↑](#footnote-ref-5)
7. Hangi paketin hangi paketi kullandığı ilişkisi. [↑](#footnote-ref-6)
8. Cosim tasarımında önemli bir tasarım hatası yoksa **cosim\_pkg**’ın **import**’lanması yeterli olmalı [↑](#footnote-ref-7)
9. Mesela normalde spike’ı çalıştırırken verdiğimiz: **spike <flag’ler, seçenekler> hello.elf**

   gibi komut satırı argümanları. [↑](#footnote-ref-8)
10. Argümanların okunduğu dosyanın yolu, **spike-cosim/cosim/makefile** dosyasında gcc derleyicisine **-DARGS\_FILE\_PATH=\"<dosya yolu>\"** şeklinde geçiliyor. Istenilirse **spike-cosim/cosim/src/**

    **cpp/cosimif.cc** dosyasında kodun içinde **ARGS\_FILE\_PATH** macro’suna elle de girilebilir. [↑](#footnote-ref-9)
11. **-d** seçeneği (interactive mod) ihtiyaç olmadığı için devre dışı bırakıldı. Ayrıca Spike’ın simülasyon sırasında **ctrl+c** ile interactive mod’a girme özelliği de devre dışı.

    Eğer **-d** seçeneğini komut satırı argümanı olarak **ARGS\_FILE\_PATH**’ta belirtilen dosyanın içine eklerseniz uyarı mesajı verir ve devam eder. [↑](#footnote-ref-10)
12. Spike’ta koşan kodumuzu komut satırından adım adım ilerleterek register’ların ve belleğin durumunu inceleyebileceğimiz etkileşimli mod. [↑](#footnote-ref-11)
13. **--log-commits** seçeneği, simülasyon adımlarında gerçekleşen olayların kayıt altında tutulması için her türlü -girseniz de girmeseniz de- aktif oluyor. [↑](#footnote-ref-12)
14. Bu özellikler, **spike-cosim/cosim/src/cpp/cosim\_conf.h** dosyası içinde tanımlanan macro’larla kontrol ediliyor [↑](#footnote-ref-13)
15. (atladıklarım olabilir) plic, clint, ns16550 ve komut satırı argümanı olarak belirtilmişse remote\_bitbang [↑](#footnote-ref-14)
16. Uzunluğu değişebilen diziler. SystemVerilog Language Standard 7.5 [↑](#footnote-ref-15)
17. Bu değer 16, fazlasıyla yetiyor. Şu ana kadar en fazla eleman eklendiğini gördüğüm durum: illegal bir buyruk koşmaya çalıştığımızda şu 6 tane register’a bazı değerler yazılıyor: **CSR\_MTINST, CSR\_MTVAL2, CSR\_MTVAL, CSR\_MCAUSE, CSR\_MSTATUS, CSR\_MEPC.** [↑](#footnote-ref-16)
18. Spike’ta simülasyonu bitirmek için kontrol edilen koşul. [↑](#footnote-ref-17)
19. riscv-pk, spike’ta koştuğumuz kod ile host-os arasında tercümanlık yapan bir yazılım. Bkz. [RISCV proxy-kernel](#_lkjltgtjwu7k) [↑](#footnote-ref-18)
20. Hart’ların durumlarıyla ilgili her türlü bilgiyi içeren bir **struct**. Register file’lar, csr’lar, pc,... [↑](#footnote-ref-19)
21. “Bellek işlem kaydı” şeklinde çevirebiliriz. [↑](#footnote-ref-20)
22. **typedef std::vector<std::tuple<reg\_t, uint64\_t, uint8\_t>> commit\_log\_mem\_t;**

    yani, commit\_log\_mem\_t türü, elemanları 3’lü tupple’lar olan bir vector (c++’ta bir liste) [↑](#footnote-ref-21)
23. **std::tuple<reg\_t, uint64\_t, uint8\_t>** ,ilk eleman (reg\_t) adres, ikincisi (varsa) yazılan değer, üçüncüsü yazılan/okunan uzunluk (1, 2, 4, 8’den biri). [↑](#footnote-ref-22)
24. **typedef std::unordered\_map<reg\_t, freg\_t> commit\_log\_reg\_t;** anahtar-değer ikililerini saklamaya yarayan yapılardan biri. Saklamak istediğimiz değerlere, anahtarları kullanarak bir takım hash işlemleriyle hızlı erişim sağlar. Değerleri indislemek istediğimiz anahtarların seyrek dağılım göstermesi tercih edilme sebeplerinden biridir. [] [] [↑](#footnote-ref-23)
25. **reg\_id**, x0 için 0, x1 için 1, x2 için 2 gibi. [↑](#footnote-ref-24)
26. **riscv-isa-sim/riscv/encoding.h** içerisinde [↑](#footnote-ref-25)
27. **riscv-isa-sim/riscv/decode\_macros.h** [↑](#footnote-ref-26)
28. **SPIKE:=spike-cosim/riscv-isa-sim # gosterecek sekilde tanimlanmissa**

    **-I$(SPIKE)/build -I$(SPIKE)/riscv -I$(SPIKE)/fesvr -I$(SPIKE)/ -I$(SPIKE)/softfloat -I$(SPIKE)/fdt** [↑](#footnote-ref-27)
29. **spike-cosim/cosim/makefile** [↑](#footnote-ref-28)
30. Cosim reposundaki örnekte **spike-cosim/cosim/log/args.txt** [↑](#footnote-ref-29)
31. **spike-cosim/README.md** [↑](#footnote-ref-30)
32. **spike-cosim/cosim/makefile** [↑](#footnote-ref-31)
33. **spike-cosim/cosim/makefile** [↑](#footnote-ref-32)
34. Makefile dosyasının konumunu taşıyan bir makefile değişkeni [↑](#footnote-ref-33)
35. Birincisi verilator’u çağırırken yapılan parse’lama, ikincisi verilator’un gcc’yi çağırdığı komutta yapılan parse’lama [↑](#footnote-ref-34)
36. **spike-cosim/cosim/src/cpp/cosimif.cc** [↑](#footnote-ref-35)
37. **cosim/src/cpp** içerisinde [↑](#footnote-ref-36)
38. Cosim için sadece bu library’lere ihtiyaç duyuluyor. [↑](#footnote-ref-37)
39. (yeni adı **cosim\_ornek\_kullanim**) [↑](#footnote-ref-38)
40. Verilator, başka modüller tarafından örneklenmeyen modülleri top modül adayı olarak belirler. [↑](#footnote-ref-39)
41. Bahsedilen çıktı: **--binary** flag’iyle kullandıysak çalıştırılabilir dosya, **--cc** flag’i ile çalıştırdıysak c++ dosyaları gibi. [↑](#footnote-ref-40)
42. Fonksiyonun esas tanımı **spike-cosim/coim/src/cpp/cosimif.cc** dosyasında. [↑](#footnote-ref-41)
43. spike-cosim/cosim/makefile [↑](#footnote-ref-42)
44. makefile’ın konumunu tutan built-in bir makefile değişkeni. [↑](#footnote-ref-43)
45. **spike-cosim/cosim/src/cpp/cosimif.cc** [↑](#footnote-ref-44)
46. Bir uart [↑](#footnote-ref-45)
47. Host target interface. Host: host işletim sistemi. Target: spike (spike’taki processor) [↑](#footnote-ref-46)
48. Spike’ı pk ile çalıştırdığımızda bu alan pk tarafından tanımlanıyor. [↑](#footnote-ref-47)
49. Spike’ı pk ile çalıştırdığımızda bu alan pk tarafından tanımlanıyor. [↑](#footnote-ref-48)
50. **spike pk spike-cosim/ornek\_test\_girdileri/fromhost\_tohost\_test/a.out** [↑](#footnote-ref-49)
51. Spike’ta koşan kod. Simülasyonun girdisi. .elf dosyası [↑](#footnote-ref-50)
52. Spike’ta koşan kod. Simülasyonun girdisi. .elf dosyası [↑](#footnote-ref-51)
53. **cosim\_pkg::cosim\_completed** tarafından kontrol edilen koşul. [↑](#footnote-ref-52)
54. Spike’ı proxy-kernel ile çalıştırdığımızda bu kanallar proxy-kernel tarafından tanımlanıp işletiliyor. [↑](#footnote-ref-53)
55. spike’taki processor’ler ve spike üzerinde koştuğumuz test girdisi [↑](#footnote-ref-54)
56. host-os [↑](#footnote-ref-55)
57. Host-os’e sistem çağrıları yapmak ve bu çağrıların dönüşlerini almak [↑](#footnote-ref-56)
58. Tanımı şu dosyalarda: **riscv-isa-sim/riscv/sim.h riscv-isa-sim/riscv/sim.cc** [↑](#footnote-ref-57)
59. Host target interface [↑](#footnote-ref-58)
60. Yani **sim\_t**, **htif\_t**’nin özelliklerinden public ve protected olanları alır. Ayrıyeten **sim\_t** kendisi de başka özellikler ekler. Örnek olarak, **dunya\_t** ve **gezegen\_t** olarak iki farklı sınıf tanımlıyor olsak **dunya\_t** sınıfı **gezegen\_t** sınıfını kalıtıyor olurdu. [↑](#footnote-ref-59)
61. **riscv-isa-sim/spike\_main/spike.cc**’de tanımlı [↑](#footnote-ref-60)
62. pmp granüleritesi, önbellek ayarları, processor sayısı, isa-spec, işlem kaydı yapılacak mı, hangi dosyaya yapılacak gibi ayarlar. [↑](#footnote-ref-61)
63. integer register’ların bit genişliği. [↑](#footnote-ref-62)
64. C++ syntax’ında **htif\_t** türünün **run** metodu [↑](#footnote-ref-63)
65. Polimorfizm: nesne yönelimli programlamada ata sınıfın nesnesinin, hem ata sınıf hem alt sınıf nesnesi olarak muamele görebilmesi [↑](#footnote-ref-64)
66. **mmap** diye bir standard c library fonksiyonu ile bir dosya, içeriği kopyalanmak zorunda kalmadan sanal bir şekilde erişilebilir oluyor. [↑](#footnote-ref-65)
67. Kodumuzun içerisindeki değişken, fonksiyon, sınıf, metot isimleri; kod derlendikten sonra binary dosyasının içinde <adres> <sembol\_ismi> [<tanım>] şeklinde birer sembol olarak yer alır. [↑](#footnote-ref-66)
68. ata sınıf’ın, tanımı kalıtan sınıflar tarafından “override” edilebilecek metodu. [↑](#footnote-ref-67)
69. ata sınıftaki tanımı geçersiz kılıp yeniden tanımlamak [↑](#footnote-ref-68)
70. **riscv-isa-sim/riscv/platform.h** dosyasında tanımlı. Processor’un state’inin içindeki pc değişkeni, **state\_t::reset** fonksiyonunda bu değeri (0x1000) alıyor. [↑](#footnote-ref-69)
71. Simülasyon cihazlarından bir tanesi [↑](#footnote-ref-70)
72. **riscv-isa-sim/riscv/devices.h** dosyasında tanımlı [↑](#footnote-ref-71)
73. .elf yükleme ve boot yükleme işlemleri bittikten sonra [↑](#footnote-ref-72)
74. *In computer programming, a callback or callback function is any reference to executable code that is passed as an argument to another piece of code; that code is expected to call back (execute) the callback function as part of its job.* [] [↑](#footnote-ref-73)
75. Tohost ve fromhost, bellekte target ve host arasında iletişimi sağlamakta kullanılan alanlar. [↑](#footnote-ref-74)
76. Spike’ı riscv-pk ile çalıştırırsak pk bu işi bizim için yapar. [] [↑](#footnote-ref-75)
77. device\_list\_t htif\_t::device\_list. device\_list\_t risv-isa-sim/fesvr/device.h’ta tanımlı. Bu listedeki en önemli araç syscall\_t syscall\_proxy. [↑](#footnote-ref-76)
78. risc v-isa-sim/fesvr/htif.h’de [↑](#footnote-ref-77)
79. Step [] tarafından çağrılan callback [↑](#footnote-ref-78)
80. cosim\_pkg [fonksiyonları](#_v69fz47tpzsj) bölümü, get\_log\_reg\_write, get\_log\_mem\_read, get\_log\_mem\_write [↑](#footnote-ref-79)
81. Imzası riscv-isa-sim/riscv/processor.h, tanımı processor.cc [↑](#footnote-ref-80)
82. Imsası riscv-isa-sim/riscv/sim.h, tanımı sim.cc [↑](#footnote-ref-81)
83. Imzası riscv-isa-sim/riscv/processor.h, tanımı riscv-isa-sim/riscv/execute.cc [↑](#footnote-ref-82)
84. Bir sınıfın nesnesini oluşturmak için var olan metod. kurucu. [↑](#footnote-ref-83)
85. Bir programlama dilinin runtime’da nesnelere metod bağlamasını sağlayan yapı [] [↑](#footnote-ref-84)
86. Işletim sisteminden nesnenin kaplayacağı alan kadar heap’ten alan isteyip verilen kurucu argümanlarını kullanarak o alanda bir nesne inşaa ettirerek o alanın pointer’ını döndürtmek için kullanılan bir c++ özelliği. [↑](#footnote-ref-85)
87. --dump-dts komut satırı argümanıyla çalıştırınca oluyor. anladığım kadarıyla simülasyonun device tree’sini bastırıp bitirme [↑](#footnote-ref-86)
88. Simülasyon konfigürasyonlarını paketleyen sınıf [↑](#footnote-ref-87)
89. Emin olmamakla birlikte “global scope kavramı olan dillerde” diye genelleyebilirim. [↑](#footnote-ref-88)
90. Static olursa yine data veya bss’te [↑](#footnote-ref-89)
91. Bir türden diğerine dönüştürme, bir türe diğeriymiş gibi muamele etme, bir türün elemanını diğer bir türdenmiş gibi algılamak [↑](#footnote-ref-90)
92. Kalıtan sınıftan ata sınıfa [↑](#footnote-ref-91)
93. Hiç bir nesneye işaret etmeyen bir referans [↑](#footnote-ref-92)
94. Int, shortint, longint, byte, real, shortreal gibi [↑](#footnote-ref-93)