İçindekiler

Spike ile Cosim

[1]

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]) ile Spike'ın fonksiyonlarından türettiğimiz fonksiyonlara erişim sağlıyor.

Bu arayüz hâlihazırda şunlardan ibaret:

```
// cosim_pkg
import "DPI-C" function void init();
```

[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

```
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] Spike'a verilecek komut satırı argümanlarını okur ve simülasyonu oluşturur.

`step` fonksiyonu simülasyonu bir adım[3] ilerletmeye yarar.

`simulation_completed` fonksiyonu, spike tarafında çalışan kodun çıkış sinyali üretip üretmediğini kontrol etmek için kullanılır. (bkz. baremetal programı sonlandırma)

---

[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

[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.

`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.

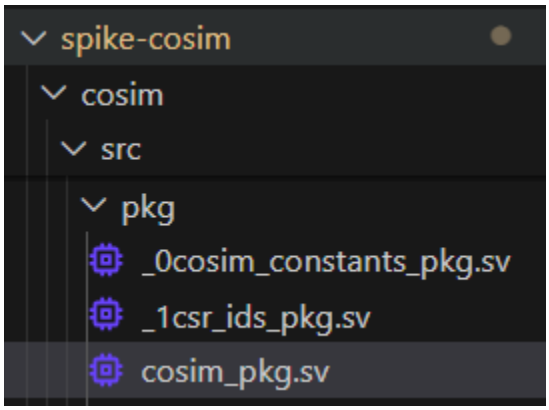
`get_log_mem_read` fonksiyonu benzer şekilde son adımda yapılan bellek okuma işlemlerini, `get_log_mem_write` fonksiyonu da belleğe yazma işlemlerini verilog tarafına çekmeye yarar.

`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]'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].

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

---

[4] Cosim'i kullanacak testbench'in **`import`**'layacağı paket

[5] *"herhangi bir nesnenin metotlarını, verilerini ve değiskenlerini diğer nesnelerden saklayarak ve bunlara erişimini sınırlandırarak yanlış kullanımlardan koruyan bir konsepttir."* [3]

lowRISC'e göre [2] paket A, paket B'yi kullanıyorsa B, A'yı kullanmamalı. Ama normalde SystemVerilog'da böyle bir kısıt yok. Dolayısıyla verilator'de[6] paketler arası bağımlılık ilişkisini[7] 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] 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], önceden belirtilmiş[10] bir dosyadan okuyor.

---

[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.

[7] Hangi paketin hangi paketi kullandığı ilişkisi.

[8] Cosim tasarımında önemli bir tasarım hatası yoksa **`cosim_pkg`**'ın **`import`**'lanması yeterli olmalı

[9] Mesela normalde spike'ı çalıştırırken verdiğimiz: **`spike <flag'ler, seçenekler> hello.elf`** gibi komut satırı argümanları.

[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.

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

`-d`[11]: (interactive mod[12]) devre dışı.

`--log-commits`[13]: her türlü aktif.

Bunlar kodun içinde macro[14] tanımlamalarıyla kontrol ediliyor.

`void step()`: simülasyonu bir "adım" ilerletir. Bu bir adım, bir buyruğun tamamlanması ve gerekli cihazların[15] 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.

---

[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.

[12] Spike'ta koşan kodumuzu komut satırından adım adım ilerleterek register'ların ve belleğin durumunu inceleyebileceğimiz etkileşimli mod.

[13] **`--log-commits`** seçeneği, simülasyon adımlarında gerçekleşen olayların kayıt altında tutulması için her türlü -girseniz de girmeseniz de- aktif oluyor.

[14] Bu özellikler, **`spike-cosim/cosim/src/cpp/cosim_conf.h`** dosyası içinde tanımlanan macro'larla kontrol ediliyor

[15] (atladıklarım olabilir) plic, clint, ns16550 ve komut satırı argümanı olarak belirtilmişse remote_bitbang

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] döndüremediğim için **CommitLogEntries**[17] şeklinde bir sabit olarak tanımladım.

`function bit simulation_completed();` Bu fonksiyon, `htif_t::exitcode != 0`[18] 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] tarafından hallediliyor. Spike üzerinde baremetal kod çalıştırırken bu işlemin nasıl yapılacağına baremetal programı sonlandırma bölümünde 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] 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] türü aslında bir listedir[22]. Bu listenin her bir elemanı[23] 1/2/4/8 byte'lık

---

[16] Uzunluğu değişebilen diziler. SystemVerilog Language Standard 7.5
[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.**
[18] Spike'ta simülasyonu bitirmek için kontrol edilen koşul.
[19] riscv-pk, spike'ta koştuğumuz kod ile host-os arasında tercümanlık yapan bir yazılım. Bkz. RISCV proxy-kernel
[20] Hart'ların durumlarıyla ilgili her türlü bilgiyi içeren bir **struct**. Register file'lar, csr'lar, pc,...
[21] "Bellek işlem kaydı" şeklinde çevirebiliriz.
[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)
[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).

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]. 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]; csr'lar için spike'ta makrolarla kodlandığı[26] 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,
```

[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. [] []
[25] **reg_id**, x0 için 0, x1 için 1, x2 için 2 gibi.
[26] **riscv-isa-sim/riscv/encoding.h** içerisinde

```
    FREG      = 'b0001,
    VREG      = 'b0010,
    VREG_HINT = 'b0011,
    CSR       = 'b0100
  } reg_key_type_e
```

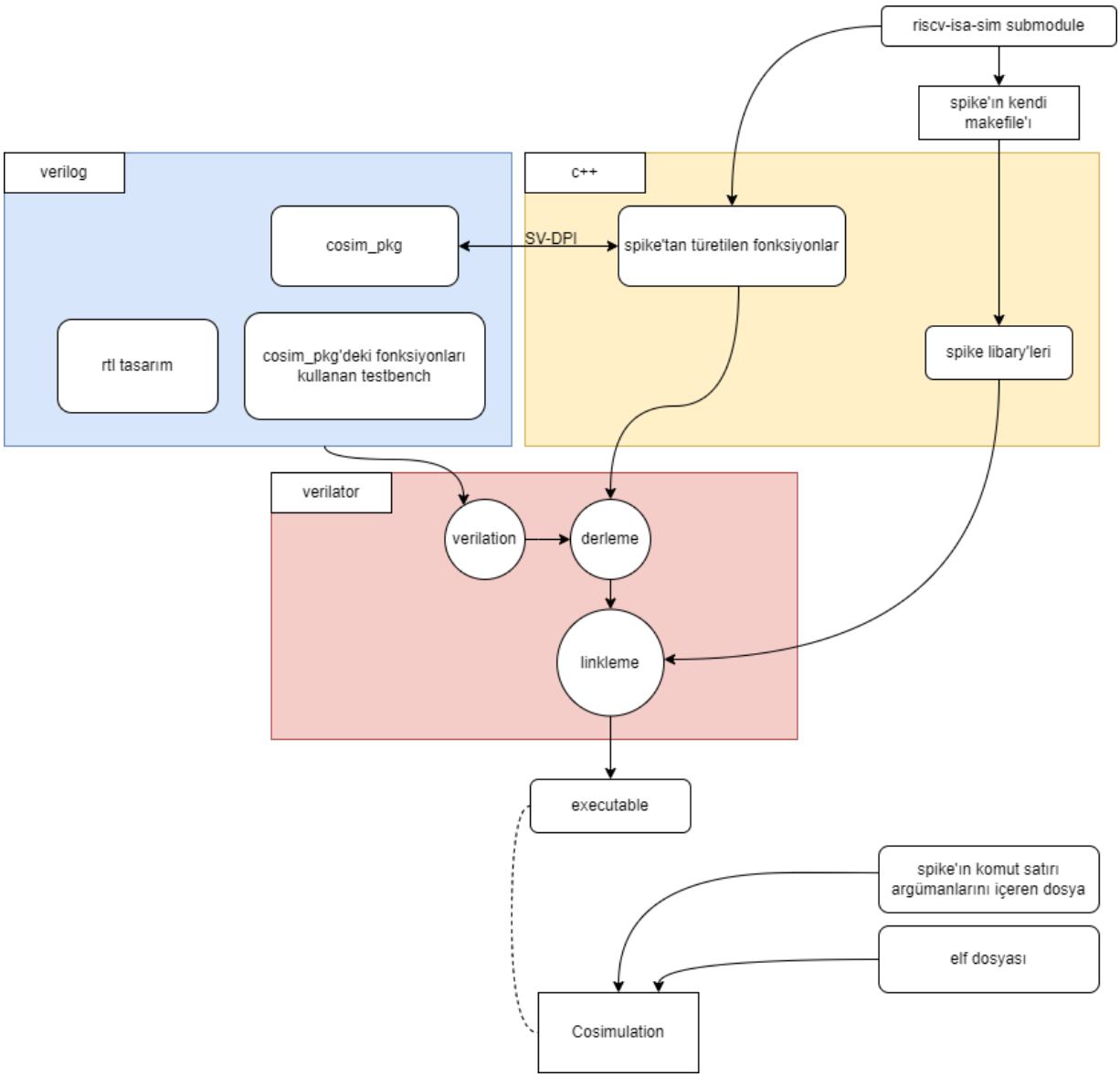
REG_KEY_TYPE_W = 4, **REG_KEY_ID_W** = 60, bu değerler spike'ta register yazma işlemlerini kaydederken[27] anahtar-değer ikililerinde anahtarın oluşturulma şeklinden geliyor.

```
  // 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
```

[27] **riscv-isa-sim/riscv/decode_macros.h**

Cosim Kullanımı



Özet: Riscv-isa-sim submodule'ünü derleyin. verilator ile **`ARGS_FILE_PATH`** makrosunu tanımlayarak ve gcc include path'ine gerekli eklemeleri[28] 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] mevcut)

---

[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`

[29] **`spike-cosim/cosim/makefile`**

ARGS_FILE_PATH makrosu ile gösterilen dosyanın[30] 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]de mevcut. Burada verilator'un kullanımıyla ilgili kısaca bahsedeceğiz.

Verilator

Aşağıda makefile[32] içindeki derleme kuralını çalıştırdığımda çalışan komutu açıklıyorum:

```
usr1@LENOVO:~/spike-cosim/cosim
> verilator -O2 -CFLAGS -DARGS_FILE_PATH=\\\"/home/usr1/spike-cosim/cosim/log/args.txt\\\" -
CFLAGS "-I/home/usr1/spike-cosim/riscv-isa-sim/build  -I/home/usr1/spike-cosim/riscv-isa-sim
/riscv -I/home/usr1/spike-cosim/riscv-isa-sim/fesvr -I/home/usr1/spike-cosim/riscv-isa-sim/
-I/home/usr1/spike-cosim/riscv-isa-sim/softfloat -I/home/usr1/spike-cosim/riscv-isa-sim/fdt"
--Mdir obj_dir_tb_spike_link --binary +1800-2017ext+sv src/pkg/_0cosim_constants_pkg.sv src
/pkg/_1csr_ids_pkg.sv [illegible] src/pkg/cosim_pkg.sv src/tb/tb_s
pike_link.sv src/cpp/args_reader.cc src/cpp/cosim_create_sim.cc src/cpp/cosim_htif.cc src/cp
p/cosim_sim.cc src/cpp/cosimif.cc /home/usr1/spike-cosim/riscv-isa-sim/build/libspike_dasm.a
 /home/usr1/spike-cosim/riscv-isa-sim/build/libriscv.so --top tb_spike_link --prefix tb_spik
e_link -o tb_spike_link.exe
```

🔴🔴 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] içerisinde **CURDIR**[34] değişkenini kullanarak makefile'ın konumuna göre hesaplanmasının daha evrensel olduğunu düşündüğüm için derleme kuralına bu şekilde ekledim.

```
compile_$(1):
  verilator -O2 -CFLAGS -DARGS_FILE_PATH=\\\"$(CURDIR)/log/args.txt\\\" -CFLAGS "$(INC_DIRS)" --M
  @echo "--compilation done--"
```

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] parse'lama olduğu için ters eğik çizgileri korumak için onları da \\ olarak veriyoruz.

---

[30] Cosim reposundaki örnekte **spike-cosim/cosim/log/args.txt**
[31] **spike-cosim/README.md**
[32] **spike-cosim/cosim/makefile**
[33] **spike-cosim/cosim/makefile**
[34] Makefile dosyasının konumunu taşıyan bir makefile değişkeni
[35] Birincisi verilator'u çağırırken yapılan parse'lama, ikincisi verilator'un gcc'yi çağırdığı komutta yapılan parse'lama

ARGS_FILE_PATH'i derleme komutunda tanımlamaya alternatif olarak cosimif.cc[36]de kullanıldığı yerde de tanımlanabilir.

```
22  void init()
23  {
24    #ifndef ARGS_FILE_PATH
25    #define ARGS_FILE_PATH "args.txt"
26    #warning ARGS_FILE_PATH is not defined. Using default value: "args.txt"
27    #endif
28
29    const char *args_filepath = ARGS_FILE_PATH;
30
```

```
usr1@LENOVO:~/spike-cosim/cosim
> verilator -O2 -CFLAGS -DARGS_FILE_PATH=\\\"/home/usr1/spike-cosim/cosim/log/args.txt\\\" -
CFLAGS "-I/home/usr1/spike-cosim/riscv-isa-sim/build  -I/home/usr1/spike-cosim/riscv-isa-sim
/riscv -I/home/usr1/spike-cosim/riscv-isa-sim/fesvr -I/home/usr1/spike-cosim/riscv-isa-sim/
-I/home/usr1/spike-cosim/riscv-isa-sim/softfloat -I/home/usr1/spike-cosim/riscv-isa-sim/fdt"
--Mdir obj_dir_tb_spike_link --binary +1800-2017ext+sv src/pkg/_0cosim_constants_pkg.sv src
/pkg/_1csr_ids_pkg.sv [redacted] src/pkg/cosim_pkg.sv src/tb/tb_s
pike_link.sv src/cpp/args_reader.cc src/cpp/cosim_create_sim.cc src/cpp/cosim_htif.cc src/cp
p/cosim_sim.cc src/cpp/cosimif.cc /home/usr1/spike-cosim/riscv-isa-sim/build/libspike_dasm.a
 /home/usr1/spike-cosim/riscv-isa-sim/build/libriscv.so --top tb_spike_link --prefix tb_spik
e_link -o tb_spike_link.exe
```

🟠🟡 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] sebep olabilecek herhangi bir nokta var mı kontrol et.

Tamamı için şuraya bakabilirsiniz: [5]

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

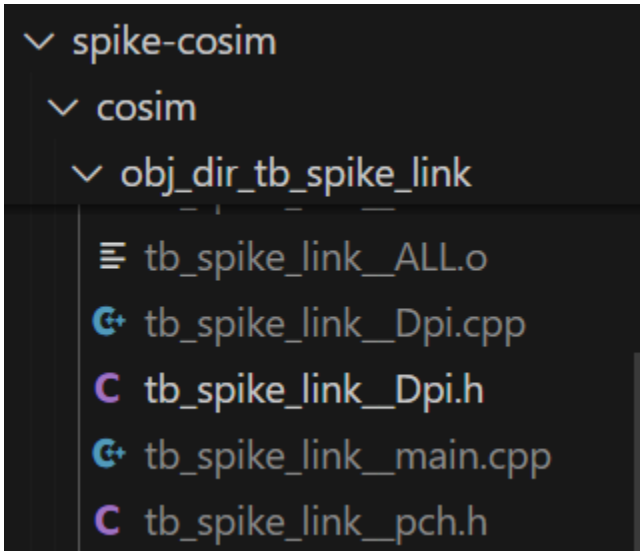
🔵🔵 SystemVerilog paketlerini barındıran dosyalar. Verilator'de A paketini kullanan paket/modülü derlerken Paket A'nın, komut satırı argümanı olarak daha önce yazılması gerekiyor. Paket A, paket B'yi kullanıyorsa B'nin tanımlandığı dosyayı A'nınkinden önce yazmamız gerekiyor.

🔵🟣 **cosim_pkg**'deki fonksiyonları kullandığım bir testbench.

[36] **spike-cosim/cosim/src/cpp/cosimif.cc**

```
usr1@LENOVO:~/spike-cosim/cosim
> verilator -O2 -CFLAGS -DARGS_FILE_PATH=\\\"/home/usr1/spike-cosim/cosim/log/args.txt\\\" -
CFLAGS "-I/home/usr1/spike-cosim/riscv-isa-sim/build  -I/home/usr1/spike-cosim/riscv-isa-sim
/riscv -I/home/usr1/spike-cosim/riscv-isa-sim/fesvr -I/home/usr1/spike-cosim/riscv-isa-sim/
-I/home/usr1/spike-cosim/riscv-isa-sim/softfloat -I/home/usr1/spike-cosim/riscv-isa-sim/fdt"
--Mdir obj_dir_tb_spike_link --binary +1800-2017ext+sv src/pkg/_0cosim_constants_pkg.sv src
/pkg/_1csr_ids_pkg.sv [redacted] src/pkg/cosim_pkg.sv src/tb/tb_s
pike_link.sv src/cpp/args_reader.cc src/cpp/cosim_create_sim.cc src/cpp/cosim_htif.cc src/cp
p/cosim_sim.cc src/cpp/cosimif.cc /home/usr1/spike-cosim/riscv-isa-sim/build/libspike_dasm.a
 /home/usr1/spike-cosim/riscv-isa-sim/build/libriscv.so --top tb_spike_link --prefix tb_spik
e_link -o tb_spike_link.exe
```

- **`cosim_pkg`**'de SV-DPI ile importlanan fonksiyonların ve bazı yardımcı fonksiyonların tanımlandığı c++ dosyaları.[37]
- Spike'ın cosim için değiştirilmiş hâlinin derlenmesiyle oluşturulmuş library'lerden ikisi[38]
- çıktı oluşturulacak top modül olarak **`tb_spike_link`**[39] modülünü seçiyoruz. Aslında benim örneğimde sadece bir tane top modül adayı[40] (**`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] 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.

---

[37] **`cosim/src/cpp`** içerisinde
[38] Cosim için sadece bu library'lere ihtiyaç duyuluyor.
[39] (yeni adı **`cosim_ornek_kullanim`**)
[40] Verilator, başka modüller tarafından örneklenmeyen modülleri top modül adayı olarak belirler.
[41] Bahsedilen çıktı: **`--binary`** flag'iyle kullandıysak çalıştırılabilir dosya, **`--cc`** flag'i ile çalıştırdıysak c++ dosyaları gibi.

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
```

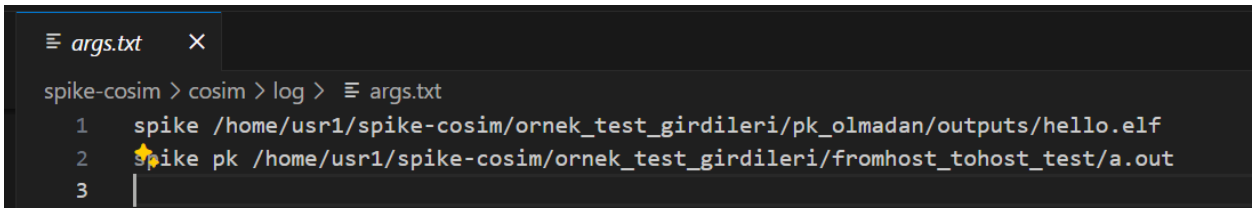
Bellek yazma olaylarına bakıyoruz.

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.

```
--compilation done--
./obj_dir_cosim_ornek_kullanim/cosim_ornek_kullanim.exe
../src/cpp/cosimif.cc:33: reading args from file: /home/u
sr1/spike-cosim/cosim/log/args.txt

--running cosim with arguments (the first line of the fil
e):--
 spike /home/usr1/spike-cosim/ornek_test_girdileri/pk_olm
adan/outputs/hello.elf
----------------------------------------
sim.cc/ sim_t::set_rom()/ start pc val: 2147483784
communication_available() is true
pc before execution: 1000
log_reg_write_from_c[0] reg_type: XREG
log_reg_write_from_c[0] reg_id: 5
log_reg_write_from_c[0].value: 1000
press any key to continue...

pc before execution: 1004
log_reg_write_from_c[0] reg_type: XREG
log_reg_write_from_c[0] reg_id: 11
log_reg_write_from_c[0].value: 1020
press any key to continue...

pc before execution: 1008
log_reg_write_from_c[0] reg_type: XREG
log_reg_write_from_c[0] reg_id: 10
log_reg_write_from_c[0].value: 0
press any key to continue...

pc before execution: 100c
log_reg_write_from_c[0] reg_type: XREG
log_reg_write_from_c[0] reg_id: 5
log_reg_write_from_c[0].value: 80000088
log_mem_read_from_c[0].addr: 1018
log_mem_read_from_c[0].wdata: 0
log_mem_read_from_c[0].len: 8
press any key to continue...
```

args.txt Dosyası

Daha önceki bölümlerde bahsedildiği gibi, bu dosya spike'ın komut satırı argümanlarını yazdığımız **`void cosim_pkg::init()`**[42] 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] derleme kuralında verilator'e **`-CFLAGS -DARGS_FILE_PATH=\\\"$(CURDIR)`**[44]**`/log/args.txt\\\"`** olarak ondan da gcc'ye **`-DARGS_FILE_PATH=\"<dosya yolu>\"`** olarak veriliyor. İstenilirse cosimif.cc[45]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], 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] 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.

---

[42] Fonksiyonun esas tanımı **`spike-cosim/coim/src/cpp/cosimif.cc`** dosyasında.
[43] spike-cosim/cosim/makefile
[44] makefile'ın konumunu tutan built-in bir makefile değişkeni.
[45] **`spike-cosim/cosim/src/cpp/cosimif.cc`**
[46] Bir uart
[47] Host target interface. Host: host işletim sistemi. Target: spike (spike'taki processor)

dosya.elf içerisinde sistem çağrısı yaptığımız yerlerde kontrol bizim kodumuzdan pk'ye geçiyor. pk, bellekte **magic_mem**[48] isimli global array'e gerekli mesajı yazıyor ve **magic_mem**'in adresini bellekte **tohost**[49] 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 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] test girdimizde[51] 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);
}
```

---

[48] Spike'ı pk ile çalıştırdığımızda bu alan pk tarafından tanımlanıyor.
[49] Spike'ı pk ile çalıştırdığımızda bu alan pk tarafından tanımlanıyor.
[50] **spike pk spike-cosim/ornek_test_girdileri/fromhost_tohost_test/a.out**
[51] Spike'ta koşan kod. Simülasyonun girdisi. .elf dosyası

baremetal_exit fonksiyonunu, test girdimizde[52] programı sonlandırmak istediğimiz yerde çağırarak htif'e programı sonlandırma mesajı göndermiş oluyoruz ve `htif_t::exitcode != 0`[53] 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] denilen iletişim kanalları tanımlanmışsa target[55] ile host[56] arasında iletişim[57] kurulacak şekilde, bu kanallar tanımlı değilse host ve target arasında iletişim kurulmadan gerçekleşir.

sim_t Sınıfı

sim_t[58] 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] sınıfını kalıtır[60]. **htif_t** temel olarak host ile target arasındaki iletişimi sağlamakla görevlidir.

---

[52] Spike'ta koşan kod. Simülasyonun girdisi. .elf dosyası
[53] **cosim_pkg::cosim_completed** tarafından kontrol edilen koşul.
[54] Spike'ı proxy-kernel ile çalıştırdığımızda bu kanallar proxy-kernel tarafından tanımlanıp işletiliyor.
[55] spike'taki processor'ler ve spike üzerinde koştuğumuz test girdisi
[56] host-os
[57] Host-os'e sistem çağrıları yapmak ve bu çağrıların dönüşlerini almak
[58] Tanımı şu dosyalarda: **riscv-isa-sim/riscv/sim.h** **riscv-isa-sim/riscv/sim.cc**
[59] Host target interface
[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.

İlklendirme İşlemleri

`main`[61] fonksiyonunda komut satırı argümanları çözümlenip belirtilen konfigürasyona[62] 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 ilişkin 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] gönderir. Daha sonra da **`htif_t::run`**[64] metodunu çağırır.
Burada **`htif_t::run`** metodu aslında bir **`sim_t`** nesnesi üzerinden çağrıldığı için mevzubahis **`htif_t`** nesnesi gerektiğinde polimorfizm[65] 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] bu .elf dosyasındaki sembol[67] isimlerini ve program giriş adresini kaydettikten sonra **`virtual void htif_t::reset()`** metodunu çağırır.
`reset` metodu, **`htif_t`**'de **`virtual`**[68] olarak tanımlandığı ve **`htif_t`**'yi kalıtan **`sim_t`**'de **`override`**[69] 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.

---

[61] **`riscv-isa-sim/spike_main/spike.cc`**'de tanımlı
[62] pmp granüleritesi, önbellek ayarları, processor sayısı, isa-spec, işlem kaydı yapılacak mı, hangi dosyaya yapılacak gibi ayarlar.
[63] integer register'ların bit genişliği.
[64] C++ syntax'ında **`htif_t`** türünün **`run`** metodu
[65] Polimorfizm: nesne yönelimli programlamada ata sınıfın nesnesinin, hem ata sınıf hem alt sınıf nesnesi olarak muamele görebilmesi
[66] **`mmap`** diye bir standard c library fonksiyonu ile bir dosya, içeriği kopyalanmak zorunda kalmadan sanal bir şekilde erişilebilir oluyor.
[67] Kodumuzun içerisindeki değişken, fonksiyon, sınıf, metot isimleri; kod derlendikten sonra binary dosyasının içinde <adres> <sembol_ismi> [<tanım>] şeklinde birer sembol olarak yer alır.
[68] ata sınıf'ın, tanımı kalıtan sınıflar tarafından "override" edilebilecek metodu.
[69] ata sınıftaki tanımı geçersiz kılıp yeniden tanımlamak

sim_t::reset'te pc'nin **DEFAULT_RSTVEC**[70] değerinden .elf entry değerine zıplamasını sağlayan boot kodlarını bus'ta[71] **boot_rom** isimlendirilen böylege **rom_device_t**[72] olarak ekleyen **sim_t::set_rom** metodu çağrılır.

htif_t::run'da, **htif_t::start**'tın tamamlanmasından[73] 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_callback** denilen bir callback[74] 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] sembolleri mevcut değilse diğer bir deyişle .elf dosyasını oluşturmak için derlediğimiz kodda fromhost ve tohost alanları tanımlı değilse[76] 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.

---

[70] **riscv-isa-sim/riscv/platform.h** dosyasında tanımlı. Processor'un state'inin içindeki pc değişkeni, **state_t::reset** fonksiyonunda bu değeri (0x1000) alıyor.
[71] Simülasyon cihazlarından bir tanesi
[72] **riscv-isa-sim/riscv/devices.h** dosyasında tanımlı
[73] .elf yükleme ve boot yükleme işlemleri bittikten sonra
[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.* []
[75] Tohost ve fromhost, bellekte target ve host arasında iletişimi sağlamakta kullanılan alanlar.
[76] Spike'ı riscv-pk ile çalıştırırsak pk bu işi bizim için yapar. []

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 bölümünde bahsettiğimiz fromhost_callback ile birlikte front end server'daki araçlara[77] 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 bölümündekiyle 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.

---

[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.

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, tohost-idle-fromhost döngüsü) 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] 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 metodu ile iletişimin mümkün olmadığını tespit ederse (bkz. Idle döngüsü) step_callback[79] olarak single_step_without_communication'u seçer.

---

[78] risc v-isa-sim/fesvr/htif.h'de
[79] Step [] tarafından çağrılan callback

single_step_without_communication'un tek yaptığı şey, idle_single_step'i (bkz. Idle'ın 5000 yerine …) 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 metodu ile iletişimin mümkün olduğunu tespit ederse (bkz. Tohost-idle-fromhost döngüsü) 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 …) 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 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] 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]
void sim_t::step_without_print(size_t n);[82]
static inline reg_t execute_insn_logged_without_print(processor_t* p, reg_t pc, insn_fetch_t fetch);[83]
Bunların spike'taki asıl muadillerinden tek farkı string'e dönüştürme ve log_file'a bastırma işlemlerinin kaldırılmış olmasıdır.

---

[80] cosim_pkg fonksiyonları bölümü, get_log_reg_write, get_log_mem_read, get_log_mem_write
[81] Imzası riscv-isa-sim/riscv/processor.h, tanımı processor.cc
[82] Imsası riscv-isa-sim/riscv/sim.h, tanımı sim.cc
[83] Imzası riscv-isa-sim/riscv/processor.h, tanımı riscv-isa-sim/riscv/execute.cc

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.

```
void sim_t::idle()
{
  if (done())
    return;

  if (debug || ctrlc_pressed)
    interactive();
  else{
    // std::cout << "sim.cc/ s
    step(INTERLEAVE);
  }
```

```
volatile bool ctrlc_pressed = false;
static void handle_signal(int sig)
{
  if (ctrlc_pressed)
    exit(-1);
  ctrlc_pressed = true;
  signal(sig, &handle_signal);
}
```

Handle_signal ise sim_t constructor'ında[84] spike process'ine gelen interrupt'larda ne yapılacağını bildirmek için signal diye bir sistem fonksiyonuna gönderiliyor.

```
signal(SIGINT, &handle_signal);
```

[84] Bir sınıfın nesnesini oluşturmak için var olan metod. kurucu.

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] 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 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 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:

[85] Bir programlama dilinin runtime'da nesnelere metod bağlamasını sağlayan yapı []

- 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] ile oluşturuluyor.
- main'in dump_dts yaptığı[87] 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" bölümünde bahsettiğimiz kurucusunu kullanıyor.
- cfg_t[88] 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 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 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);
```

[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.
[87] --dump-dts komut satırı argümanıyla çalıştırınca oluyor. anladığım kadarıyla simülasyonun device tree'sini bastırıp bitirme
[88] Simülasyon konfigürasyonlarını paketleyen sınıf

```
argv_argc_t *argc_argv = read_args_from_file(args_filepath);
```

2- okunan argümanları create_sim_with_args 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] 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] 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 ve sim_t'nin kalıttığı htif_t'nin start (bkz. Ilklendirme işlemleri)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] 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] 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.

---

[89] Emin olmamakla birlikte "global scope kavramı olan dillerde" diye genelleyebilirim.

[90] Static olursa yine data veya bss'te

[91] Bir türden diğerine dönüştürme, bir türe diğeriymiş gibi muamele etme, bir türün elemanını diğer bir türdenmiş gibi algılamak

[92] Kalıtan sınıftan ata sınıfa

4- htif::run'ın ilk kısımlarında yapılana benzer şekilde (bkz. Ilklendirme işlemleri) 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] oluşturmak yerine aslında "varsayılan ilklendirilmiş" [] bir nesne oluşturmaktadır.

5- host ile target arasında iletişimin mümkün olup olmamasına (bkz. communication_available) 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'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'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.

[93] Hiç bir nesneye işaret etmeyen bir referans

void step()

DPI ile importlanan void cosim_pkg::step() fonksiyonunun tanımlandığı kısımdır. Init 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 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 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 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] 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.

[94] Int, shortint, longint, byte, real, shortreal gibi

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) 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.

```
for (auto x: map_from_c_side){
  item_ptr[num_entries].key = x.first;
  item_ptr[num_entries].value = x.second;
  num_entries++;
}
```

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:

```
typedef struct packed {
  reg_key_t key;
  freg_t value;
} commit_log_reg_item_t;
```

c++ tarafında:

```
typedef struct {
  freg_t value;
  uint64_t key;
} commit_log_reg_item_t;
```

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 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;"

```
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,
  num_entries++;
}
```

Yandaki döngü ile spike'tan erişti̇ğimiz 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'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:

```
typedef struct packed {
  reg_t addr;
  reg_t wdata;
  bit [55:0] reserved; //
  byte unsigned len;
} commit_log_mem_item_t;
```

c++ tarafında:

```
typedef struct {
  uint8_t len;
  uint8_t reserved[7];
  uint64_t wdata;
  reg_t addr;
} commit_log_mem_item_t;
```

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%BCll eme

[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.