



**THE IMAGINATION UNIVERSITY PROGRAMME**

# **RVfpga Deney 9**

## **Kesinti-güdümlü Girdi/Çıktı**

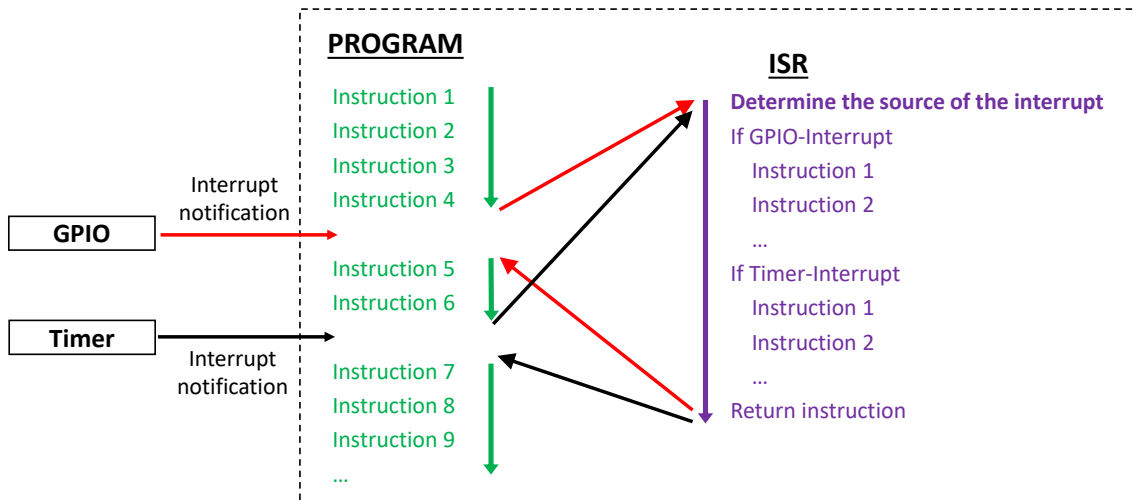
## 1. GİRİŞ

Bu deneyde kesintiler konseptini tanıtır, RVfpga’de kesintilerin nasıl kullanılacağını gösteriyoruz. Kesintileri yazılım da donanım da oluşturabilir. Bu deneyde fiziksel bir ucun değişimiyle tetiklenen donanım kesintilerine odaklanıyoruz. Bölüm 2’de **Programlanmış Girdi/Çıktı** ile **Kesinti-güdümlü Girdi/Çıktı** ayrımını tanımlayarak başlıyoruz. Ardından RVfpga’in Kesinti Denetleyicisinin, ki bu SweRV EH1 çekirdeğinin parçasıdır (Bölüm 3), işlemini açıklıyoruz. Bölüm 4’te Western Digital’in Peripherals Support Package’ini (PSP), Board Support Package’ini (BSP), ki bunlar donanım çevre birimlerinin sürücülerini içeren yazılımlardır, kullanarak dış kenstilerin nasıl yapılandırılacağını tanımlıyoruz. Son olarak (Bölüm 5) örnek programlar gösterip ardından (Bölüm 6) RVfpga’in donanım kesintilerini kullanıp genişletme için alıştırmalar öneriyoruz.

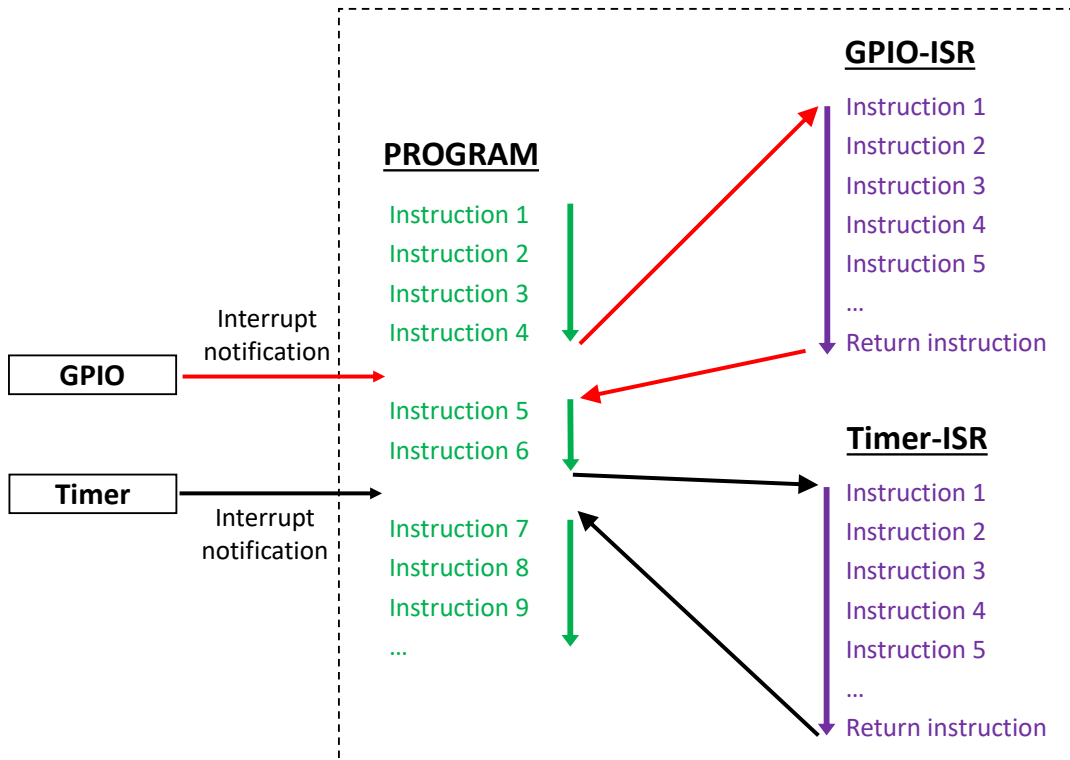
## 2. PROGRAMLANMIŞ I/O VS. KESİNTİ-GÜDÜMLÜ I/O

Çevre birimleriyle etkileşime geçmek için birden çok yöntem vardır: Programlanmış I/O, Kesinti-güdümlü I/O, Direkt Bellek Erişimi (DMA). 2-8 arası deneylerde çevre birimleriyle etkileşime geçmek için **Programlanmış I/O** kullandık. Programlanmış I/O’da kullanıcı program sürekli I/O’yu yoklayıp durumuna göre tepki verir. Örneğin Deney 6’daki *Temel Alıştırma* anahtarları sürekli yoklayıp programlanmış I/O kullandı. Programlanmış I/O’nun gerçekleştirmesi çok kolay olup, çok az donanım desteği gerektirir, ancak sürekli I/O yoklamasıyla işlemciyi kullanışsız işle dolu tutar.

**Kesinti güdümlü I/O** bu engeli aşıp programın yalnızca etkinlik çevre biriminde gerçekleşince tepki vermesini sağlar. Bu şemada çevre birimi, bir etkinlik olduğunda işlemciye sinyal yollamakla (**kesinti** denir) sorumludur – örneğin, bir zamanlayıcı değer aşımında, bir UART arayüzünde karakter alındığında, bir buton basıldığında, gibi gibi. Bir etkinlik gerçekleşmediğinde (bir diğer deyişle kesinti yokken), işlemci yararlı işler yapmayı sürdürür. İşlemci bir kesinti aldığında çalıştırdığı programı beklemeye alıp bir *kesinti hizmet yordamı* (ISR), *kesinti yöneticisi* de denir, çalıştırır. ISR özünde kesintileri yöneten, `void` argümanlı bir işlevdir – yani butonun yeni değerini okur, zamanlayıcı değer aşımıyla ilgili eylemler yapar, gibi gibi. İşlemciler genelde bir- ile çoklu-vektör modlarını destekler. Bir-vektör modunda (Figür 1), bütün kesintiler aynı ISR’ı çalıştırır. Dolayısıyla, bir kesinti gerçekleşince işlemci ana programı durdurup yaygın ISR’a sıçrar, burada da ilk kesinti kaynağı belirlenip, belirlenen kesinti nedenine karşılık gelen belirli ISR kodu yürütülür. Çoklu-vektör modunda (Figür 2), bütün kesintiler ayrı ISR’ları çalıştırır. Dolayısıyla bir kesinti oluşturulduğunda ilk kesinti kaynağı belirlenip ardından program bu nedene karşılık gelen ISR’a sıçrar.



**Figür 1. Bir-vektör modunda 2 kesintili örnek**



**Figür 2. Çoklu-vektör modunda 2 kesintili örnek**

İşlemciler genelde kesintilere öncelik verilmesini sağlar. Yüksek öncelikli kesintiler yalnızca ilk yönetilmez, üstüne yüksek öncelikli kesinti yönetilen düşük öncelikli kesintiyi bekletebilir. Örneğin, diyelim ki buton kesintisine 5 önceliği verildi, zamanlayıcı kesintisine 7 önceliği verildi, eşik ise 4'e ayarlandı (yani iki öncelik de eşğin yukarısında). Program normal akışını yürütürken buton basılırsa kesinti oluşup işlemci ISR'ı çağıracaktır, bu da butondan veriyi okuyup yönetir. Buton ISR'ı etkinken zamanlayıcı değer aşımı gerçekleşirse ISR'ın kendisi

kesilip işlemci anında zamanlayıcı değer aşımıyla ilgilenmeye başlayacaktır. Bitince program ana akışına dönmeyen buton kesintisine dönüp onu bitirecektir<sup>1</sup>.

### 3. SWERV EH1'İN SAĞLADIĞI PROGRAMLANABİLİR KESİNTİ DENETLEYİCİSİ

SweRV EH1 çekirdeği şu referanslarda tanımlanıp aşağıda özetlenen kesintileri destekler:

- **[PRM v1.7]** Revision 1.7 (June 25, 2020), Chapter 6, “RISC-V SweRV EH1 Programmer’s Reference Manual”  
[https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V\\_SweRV\\_EH1\\_PRM.pdf](https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf)
- **[ISM v1.11]** Version 1.11-draft (December 1, 2018), Chapter 7, “The RISC-V Instruction Set Manual – Volume II: Privileged Architecture”  
<https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20181201-2650e2a>

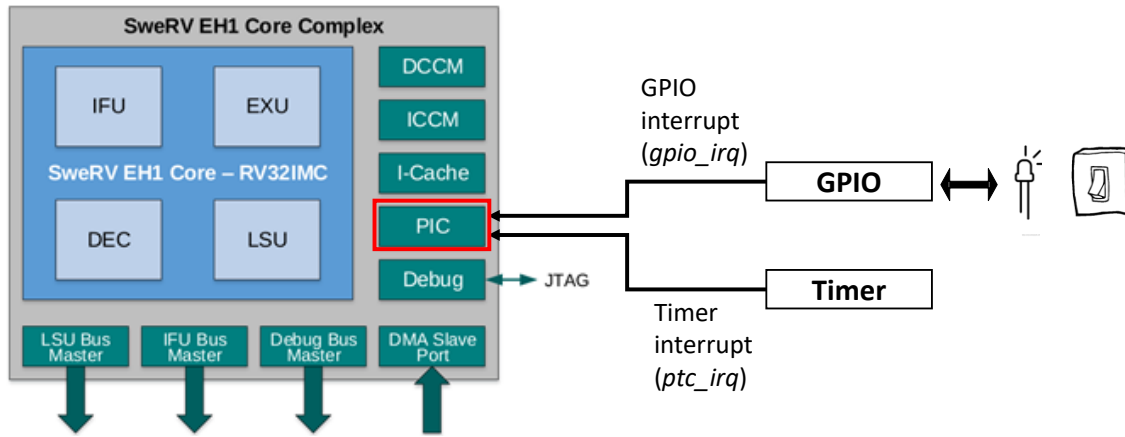
SweRV EH1 çekirdeğindeki dış kesintiler ([PRM v1.7]) büyük oranda RISC-V PLIC (Platform Düzeyli Kesinti Denetleyicisi) spesifikasyonu üzerine modellenmiştir ([ISM v1.11]). Ancak kesinti denetleyicisi çekirdekle ilişkilendirilmiştir, platformla değil. Dolayısıyla daha genel terim olan PIC (Programlanabilir Kesinti Denetleyicisi), SweRV EH1 çekirdeğindeki denetleyiciyi adlandırmak için kullanılır. PIC şu ana özellikleri sağlar:

- 255 dış kesinti kaynağına dek destekler (1 (en yüksek öncelikten) 255 (en düşük önceliğe)); bütün kaynakların kendi etkinleştiricisi vardır.
- Kaynak numaralandırmanın yanı sıra 15 ek öncelik düzeyi sağlar; iki öncelik şemasına erişilebilir: 1-15 (1 en düşük öncelik), ya da 0-14 (14 en düşük öncelik). Bütün kaynaklara bir öncelik atanabilir.
- Düşük öncelikli kesintileri etkisiz kılmak için programlanabilir öncelik eşiğini destekler.
- Vektörlü dış kesintileri, kesinti zincirlemeyi, iç içe geçmiş kesintileri destekler.

Figür 3 RVfpga’in kesinti sisteminin basitleştirilmiş bir sürümünü görselleştirir. Kesinti oluşturan bütün işlevsel ünitelere **dış kesinti kaynağı** denir. Dış kesinti kaynakları **PIC**’ye **\_irq** (kesinti isteğinin bir kısaltması) ile biten bir eş zamansız sinyal göndererek kesinti isteğini gösterir. Bu deneyde kesintilerin zamanlayıcı ile GPIO’dan nasıl kullanılacağını gösteriyoruz; bu üniteler **ptc\_irq** ile **gpio\_irq** sinyallerini kullanarak kesinti oluşturur.

Bütün dış kesinti kaynakları bir adanmış geçide bağlanır (PIC içerisindedir), bu çekirdeğin saat alanına kesinti isteğini senkronize etmek, istek sinyalini yaygın kesinti istek formatına (bir diğer deyişle etkin-yüksek/alçak ya da düzey-tetikli) PIC için dönüştürmekle sorumlu bir donanım yapısıdır. PIC kesinti kaynağı başına bir kezde yalnızca bir kesinti isteğini yönetebilir. Bütün bekleyen, etkin kesinti isteklerini değerlendirip en düşük kaynak ID’li, en yüksek öncelikli kesinti isteğini seçer. Ardından önceliği programlanabilir öncelik eşiğiyle kıyaslayıp, iç içe geçmiş kesintileri desteklemek için eğer çalışan bir kesinti yöneticisi varsa onun önceliğiyle kıyaslar. Eğer seçilen isteğin önceliği ikisinden de yüksekse PIC çekirdeğe bir kesinti bildirimi yollar, bu da ana programın yürütmesini durdurup karşılık gelen ISR’a sıçrar, Figür 1 (bir vektör modu) ile Figür 2 (çoklu vektör modu) içerisinde görselleştirildiği gibi.

<sup>1</sup> D. Harris and S. Harris. “*Digital Design and Computer Architecture*”. Second Edition – 2012. Morgan Kaufmann Publishers (San Francisco, CA, United States). ISBN:978-0-12-394424-5.



**Figür 3. RVfpga kesinti sistemi**

PIC'nin ana işlevleri şu basit adımlarla özetlenmiştir:

- 1) Etkinleştirme/Engelleme: PIC dış kesintilerin etkinleştirilmesine/engellenmesine izin verir
- 2) Yapılandırma: PIC değişik ucaylılık (etkin-yüksek/etkin-alçak) ya da tür (kenar-tetikli/düzyet-tetikli) dış kesintilere dinlemek üzere yapılandırılabilir. PIC ayrıca ISRların değişik bellek adreslerine ayrılmasına da olanak sağlar.
- 3) Filtreleme, öncelik atama: PIC kesintilere öncelik düzeyi atanmasını sağlar. Ana program çalışırken PIC etkinleştirilmiş, tetiklenmiş kesintilerin en yüksek önceliklisini seçer.
- 4) Bildirim: PIC en yüksek öncelikli kesintiyi seçince çekirdeğe kesintiye hizmet eden yordama sıçramak için ana programın yürütmesini durdurmasını bildirir.
- 5) Boşaltma: iç içe geçmiş kesintiler etkinse hizmet verilen kesinti, daha yüksek öncelikli bir diğer kesintiyle boşaltılabilir.

#### 4. SWERV EH1'DA DIŞ KESİNTİLERİ YAPILANDIRMA

Diğer çevre birimleri gibi PIC, kullanıcının yükle/depola yönergeleriyle erişebildiği bellek eşlenmiş yazmaçlarla yapılandırılır. Kesinti sistemi yazmaç düzeyinde kullanmak çok karmaşık olur; neyse ki WD'nin İşlemci Destek Paketi (PSP) ile Kart Destek Paketi (BSP) (<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>) kesinti kullanan programları gerçekleştirmeyi çok daha basitleştiren işlevler içerir. Tablo 1 dış kesintileri yapılandırmak için gereken ana işlevlerle makroları tanımlar. Bütünlük için bu dokümanın sonundaki Ek, diğer erişilebilir yazmaçlarla PIC'nin yazmaç düzeyinde yapılandırmasıyla kullanımının tanımını sağlar.

**Tablo 1. Dış kesintileri yapılandırmak için kullanılan basit işlevlerle makrolar**

Başlık	Tanım
<code>void pspInterruptsSetVectorTableAddress(void* pVectTable);</code>	Prepares vector-table address
<code>void pspExternalInterruptSetVectorTableAddress(void* pExtIntVectTable);</code>	Prepares external interrupts vector-table address
<code>void bspInitializeGenerationRegister(u32_t uiExtInterruptPolarity);</code>	Put the Generation-Register in its initial state
<code>void bspClearExtInterrupt(u32_t uiExtInterruptNumber);</code>	Clear the trigger that generates external interrupt
<code>void pspExtInterruptSetPriorityOrder(u32_t uiPriorityOrder);</code>	Sets Priority Order (Standard or Reserved)
<code>void pspExtInterruptsSetThreshold(u32_t uiThreshold);</code>	Sets the priority threshold of the external interrupts in the PIC
<code>void pspExtInterruptsSetNestingPriorityThreshold(u32_t uiNestingPriorityThreshold);</code>	Sets the nesting priority threshold of the external interrupts in the PIC
<code>void pspExtInterruptSetPolarity(u32_t uiIntNum, u32_t uiPolarity);</code>	Sets the polarity (active-high or active-low) of a specified interrupt line
<code>void pspExtInterruptSetType(u32_t uiIntNum, u32_t uiIntType);</code>	Sets the type (Level-triggered or Edge-triggered) of a specified interrupt line
<code>void pspExtInterruptClearPendingInt(u32_t uiIntNum);</code>	Clears the indication of pending interrupt for the specified interrupt line
<code>void pspExtInterruptSetPriority(u32_t uiIntNum, u32_t uiPriority);</code>	Sets the priority of a specified interrupt line
<code>void pspExternalInterruptEnableNumber(u32_t uiIntNum);</code>	Enables a specified interrupt line in the PIC
<code>void pspInterruptsEnable(void);</code>	Enable interrupts (in all privilege levels) regardless their previous state
<code>void pspInterruptsDisable(u32_t *pOutPrevIntState);</code>	Disables interrupts and return the current interrupt state in each one of the privileged levels

Örnek kesinti hizmet yordamları (ISR'lar) bu deneyin ilerisinde verilir. RVfpga kesintilerini yapılandırmak için aşağıda tanımlanan adımları izlerler, Tablo 1'deki işlevler üzerine. Önemli olarak, PIC'yi yapılandırmanın yanı sıra, dış kesinti oluşturan çevre birimleri de yapılandırılmalıdır (bu örneklerle alıştırmalarda kullanılan çevre birimleri için sonra tanımlanacaktır).

#### **KESİNTİ SİSTEMİNİN VARSAYILAN İLK DEĞERLENDİRMESİ:**

1. Çoklu vektör modunda dış vektörlü kesinti adres tablosunun taban adresini ayarla. `pspInterruptsSetVectorTableAddress`, `pspExternalInterruptSetVectorTableAddress` işlevlerini kullan.
2. Oluşturma Yazmacını ilk durumuna getir. `bspInitializeGenerationRegister` işlevini kullan.
3. Dış kesinti tetiklerinin sıfırlandığının sağlanmasını yap. `bspClearExtInterrupt` işlevini kullan.
4. Öncelik sırası için, (işlev `pspExtInterruptSetPriorityOrder`), eşik için (işlev `pspExtInterruptsSetThreshold`), iç içe geçme öncelik eşiği için (işlev `pspExtInterruptsSetNestingPriorityThreshold`) varsayılan değerleri ayarla.

### **KESİNTİ KAYNAKLARININ İLK DEĞERLENDİRMESİ:**

1. Bütün kesinti kaynakları için ucaylılığı (etkin-yüksek/etkin-düşük), türü (düzey-tetikli/kenar-tetikli) `pspExtInterruptSetPolarity`, `pspExtInterruptSetType` işlevlerini kullanarak ayarla.
2. Bekleyen kesintileri `pspExtInterruptClearPendingInt` işlevini kullanarak sıfırla.
3. `pspExtInterruptSetPriority` işlevini kullanarak bütün dış kesinti kaynakları için öncelik düzeyini ayarla.
4. `pspExternalInterruptEnableNumber` işlevini kullanarak uygun dış kesinti kaynakları için kesintileri etkinleştir.
5. Çoklu vektör modunda bütün dış kesinti kaynakları için karşılık gelen yöneticinin adresini dış vektörlü kesinti adres tablosunda yaz.

**İLERİ DÜZEY GÖREV:** Bu temel işlevleri daha derinden anlamak için şuradaki PSP kodunu `.platformio/packages/framework-wd-riscv-sdk/psp`, şuradaki BSP kodunu `.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/bsp` incele. Özellikle şu dosyalar ilgi çekicidir, birkaçı `api_inc` alt klasöründedir.

- `bsp_external_interrupts.h`: external\_interrupts creation in RVfpga
- `psp_interrupts_eh1.h`: it provides information and registration APIs for ISRs on the EH1 core
- `psp_ext_interrupts_eh1.h`: it defines the psp external interrupts interfaces for SweRV EH1
- `psp_macros_eh1.h`: it defines the psp macros for SweRV EH1
- `psp_csrs_eh1.h`: definitions of SweRV EH1 CSRs

Bu işlevlerden en az birini yazmaç düzeyine dek çözümleme önerilir. Bu amaç için Ek'te verilen bilgiyi kullanabilirsin, orada SweRV EH1 Çekirdeğinin PIC'sinin dış kesintileri yazmaç düzeyinde nasıl yapılandırıp yönettiği tanımlanır.

**İLERİ DÜZEY GÖREV:** Western Digital'in sağladığı dış kesintiler demosunu da çözümleyip yürütmeni öneriyoruz, şuradan <https://github.com/westerndigitalcorporation/riscv-fw-infrastructure> ya da PlatformIO projesinde şuradan `[RVfpgaPath]/RVfpga/Labs/Lab9/WD_demo_external_int_Original` erişilebilir. Eğer düzgün çalışırsa dizisel konsolda şu iletileri görmeyi gerekir:

```
Hello from SweRV core running on NexysA7
Core list:
  EH1 = 11
  EL2 = 16
Running demo on core 11...
-----
SweRVolf version 255.255255 (SHA 000000ef) (dirty 128)
-----
External Interrupts tests passed successfully
```

## 5. ÖRNEKLER

Bu bölümde programlanmış I/O programları kesinti güdümlü I/O programlarına dönüştürmenin örneklerini sağlıyoruz. Programlanmış I/O'nun değişik sorunlarını gösteren (ilk, ikinci örnekler), ardından Kesinti-güdümlü I/O şemasıyla bu sorunların nasıl kolayca çözülebileceğini gösteren (üçüncü örnek) üç örnek gösteriyoruz.

### A. LED-Switch C-Lang programı

*LED-Switch\_C-Lang* programı (Figür 4'e göz at) en sağdaki LEDin durumunu, en sağdaki anahtarda 0→1 geçişi gerçekleştirdiğinde tersie döndürür. Program şuradadır:

[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch\_C-Lang.c

Çevre birimlerinin ilk değerlendirmesinden sonra program şimdiki durumu önceki durumla kıyaslayan bir sonsuz döngüye girer, bir 0→1 geçişi bulunursa LED durumunu tersine döndürür (önemli olarak 1→0 geçişinde bir değişiklik ya da eylem gerçekleşmez).

C'de yazılan önceki örneklerle alıştırılmalarda I/O yazmaçlarını okumak için makrolar tanımladık (READ\_GPIO, READ\_Reg, WRITE\_GPIO, WRITE\_Reg, gibi.). Bu örnekte bu amaçla PSP'de tanımlanmış iki makroyu kullanıyoruz: M\_PSP\_READ\_REGISTER\_32, ki bu argüman olarak verilmiş 32-bitlik yazmacı okur, ile M\_PSP\_WRITE\_REGISTER\_32, ki bu ikinci argümanda verilmiş değerle bir 32-bitlik yazmaca yazar. Bu makroları kullanmak için `framework = wd-riscv-sdk` satırını *platformio.ini* dosyasında içermen, (bu bir proje RVfpga hedeflenerek oluşturulduğunda varsayılandır) `#include "psp_api.h"` satırını program başında içermen gerektiğini anımsa (Figür 4, satır 1).

```

1  #include "psp_api.h"
2
3  #define GPIO_SWs    0x80001400
4  #define GPIO_LEDs   0x80001404
5  #define GPIO_INOUT  0x80001408
6
7  int main ( void )
8  {
9      int LED_state, Sw_current_state, Sw_previous_state;
10
11      /* Configure LEDs and Switches */
12      M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF);
13
14      /* Init states */
15      LED_state = 0;
16      M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
17      Sw_previous_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
18
19      while (1) {
20          /* Invert LED-0 when SW-0 goes high */
21          Sw_current_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
22          if(Sw_current_state==1 && Sw_previous_state==0){
23              LED_state = !LED_state;
24              M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
25          }
26          Sw_previous_state = Sw_current_state;
27      }
28
29      return(0);
30  }

```

**Figür 4. LED-Switch\_C-Lang programı**

**GÖREV:** *LED-Switch\_C-Lang* programını detaylıca anlamak için çözümle. Gerekirse programı adım adım çözümlemek için ayıklayıcıyı kullanabilirsin.

Program düzgün çalışır ancak çok verimsizdir, program anahtarları/LEDleri okuyup/yazma dışında iş yapmaz. İşlemcimizin I/O aygıtlarıyla iletişim kurma dışında işler de yapmasını isteriz.



## B. LED-Switch 7SegDispl C-Lang programı

Bu ikinci örnekte, *LED-Switch\_7SegDispl\_C-Lang*, program *LED-Switch\_C-Lang*'i ikinci bir çevre birimiyle genişletir: 7-kesimli ekranlar. Program iki görevi yerine getirir:

- İlk örnekteki gibi en sağdaki LED'i 0→1 geçişi en sağdaki anahtarda olunca tersine döndürür.
- 8-sayı 7-kesimli ekranlarda yükselen bir sayı gösterir, ki bu saniyede bir artar. Önemli olarak, anlaşılmanın kolaylığı için, bir `for` döngüsüyle bir saniyelik geciktirme oluşturuyoruz (Alıştırma 1'de bu amaçlar Deney 8'deki zamanlayıcıyı kullanacaksınız).

Bu programı Figür 5'te görüp şurada bulabilirsin:

[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch\_7SegDispl\_C-Lang.c

İlk değerlendirmelerden sonra program şimdiki anahtar değerini öncekiyle kıyaslayıp 0→1 geçişi olursa LED durumunu tersine döndüren bir döngüye girer. Ardından 8-sayı 7-kesimli ekranlarda gösterilen değer artırılıp gecikme oluşturulur. Figür 5'te kırmızı kutuya göz at.

```

1  #include "psp_api.h"
2
3  #define SegEn_ADDR    0x80001038
4  #define SegDig_ADDR   0x8000103C
5
6  #define GPIO_SWs      0x80001400
7  #define GPIO_LEDs     0x80001404
8  #define GPIO_INOUT    0x80001408
9
10 int main ( void )
11 {
12     int i, LED_state, Sw_current_state, Sw_next_state, count=0;
13
14     /* Configure LEDs and Switches */
15     M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF);
16
17     /* Configure 7-Seg Displays */
18     M_PSP_WRITE_REGISTER_32(0x80001038, 0x0);
19
20     /* Init states */
21     LED_state = 0;
22     M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
23     Sw_current_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
24
25     while (1) {
26         /* Invert LED-0 when SW-0 goes high */
27         Sw_next_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
28         if(Sw_current_state==0 && Sw_next_state==1){
29             LED_state = !LED_state;
30             M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
31         }
32         Sw_current_state = Sw_next_state;
33
34         /* Increase 7-Seg Displays */
35         M_PSP_WRITE_REGISTER_32(SegDig_ADDR, count);
36         count++;
37
38         /* Delay */
39         for(i=0;i<1000000;i++);
40     }
41
42     return(0);
43 }

```

Figür 5. *LED-Switch\_7SegDispl\_C-Lang* programı

**GÖREV:** Detaylıca anlamak için *LED-Switch\_7SegDispl\_C-Lang* programını çözümü. Gerekirse programı adım adım çözümlemek için ayıklayıcıyı kullanabilirsin.

Önemli olarak bu durumda program kimi durumlarda doğru bile çalışmaz. Örneğin geciktirme döngüsü sırasındaki bir 0→1→0 geçişi tespit edilemez. Dahası, önceki örneklerdeki sorunlar yine bulunur: işlemci sürekli aygıtları okuma/yazma ya da gecikme oluşturmaya uğraşır.



```

114 int main(void)
115 {
116     int count=0, i;
117
118     /* INITIALIZE THE INTERRUPT SYSTEM */
119     DefaultInitialization(); /* Default initialization */
120     pspExtInterruptsSetThreshold(5); /* Set interrupts threshold to 5 */
121
122     /* INITIALIZE INTERRUPT LINE IRQ4 */
123     ExternalIntLine_Initialization(4, 6, GPIO_ISR); /* Initialize line IRQ4 with a priority of 6. Set GPIO_ISR as the Interrupt Service Routine */
124     M_PSP_WRITE_REGISTER_32(Select_INT, 0x1); /* Connect the GPIO interrupt to the IRQ4 interrupt line */
125
126     /* INITIALIZE THE PERIPHERALS */
127     GPIO_Initialization(); /* Initialize the GPIO */
128     M_PSP_WRITE_REGISTER_32(SegEn_ADDR, 0x0); /* Initialize the 7-Seg Displays */
129
130     /* ENABLE INTERRUPTS */
131     pspInterruptsEnable(); /* Enable all interrupts in mstatus CSR */
132     M_PSP_SET_CSR(D_PSP_MIE_NUM, D_PSP_MIE_MEIE_MASK); /* Enable external interrupts in mie CSR */
133
134     while (1) {
135         /* Increase 7-Seg Displays */
136         M_PSP_WRITE_REGISTER_32(SegDig_ADDR, count);
137         count++;
138
139         /* Delay */
140         for(i=0; i<50000000; i++);
141     }
142 }
143

```

**Figür 7. main işlevi.**

**DefaultInitialization** işlevi, Figür 8, aşağıdaki öge “DEFAULT INITIALIZATION OF THE INTERRUPT SYSTEM” Bölüm 4’ünde açıklanan adımları atar:

- It configures the vector-table (lines 53 and 56). Note that, in this example, array `G_Ext_Interrupt_Handlers` stores the vector-table.
- It initializes the register used for triggering the IRQs (line 59).
- It clears all external interrupts (in our case IRQ3 and IRQ4) at lines 61-65. Constants `D_BSP_FIRST_IRQ_NUM` and `D_BSP_LAST_IRQ_NUM` are defined by WD’s BSP to 3 and 4, respectively.
- It establishes the default threshold and priorities (lines 68, 71 and 74). Again, the constants used by these functions are defined by WD’s PSP.

```

48 void DefaultInitialization(void)
49 {
50     u32_t uiSourceId;
51
52     /* Register interrupt vector */
53     pspInterruptsSetVectorTableAddress(&M_PSP_VECT_TABLE);
54
55     /* Set external-interrupts vector-table address in MEIVT CSR */
56     pspExternalInterruptSetVectorTableAddress(G_Ext_Interrupt_Handlers);
57
58     /* Put the Generation-Register in its initial state (no external interrupts are generated) */
59     bspInitializeGenerationRegister(D_PSP_EXT_INT_ACTIVE_HIGH);
60
61     for (uiSourceId = D_BSP_FIRST_IRQ_NUM; uiSourceId <= D_BSP_LAST_IRQ_NUM; uiSourceId++)
62     {
63         /* Make sure the external-interrupt triggers are cleared */
64         bspClearExtInterrupt(uiSourceId);
65     }
66
67     /* Set Standard priority order */
68     pspExtInterruptSetPriorityOrder(D_PSP_EXT_INT_STANDARD_PRIORITY);
69
70     /* Set interrupts threshold to minimal (== all interrupts should be served) */
71     pspExtInterruptsSetThreshold(M_PSP_EXT_INT_THRESHOLD_UNMASK_ALL_VALUE);
72
73     /* Set the nesting priority threshold to minimal (== all interrupts should be served) */
74     pspExtInterruptsSetNestingPriorityThreshold(M_PSP_EXT_INT_THRESHOLD_UNMASK_ALL_VALUE);
75 }

```

**Figür 8. DefaultInitialization işlevi**

**ExternalIntLine\_Initialization** işlevi, Figür 9, aşağıdaki öge “INITIALIZATION OF EACH INTERRUPT SOURCE” bölüm 4’ünde açıklanan adımları atar:

- It configures the type and polarity of the IRQ4 interrupt (the constants used by these functions are defined by WD’s PSP) and it clears any potential pending interrupts at the corresponding gateway (lines 81, 84 and 87).

- It sets the priority for IRQ4 (line 90).
- It enables IRQ4 interrupts in the PIC at line 93.
- It registers the GPIO Interrupt Service Routine (GPIO\_ISR) in the vector-table (at line 96), which is stored in array `G_Ext_Interrupt_Handlers`.

```

78 void ExternalIntLine_Initialization(u32_t uiSourceId, u32_t priority, pspInterruptHandler_t pTestIsr)
79 {
80     /* Set Gateway Interrupt type (Level) */
81     pspExtInterruptSetType(uiSourceId, D_PSP_EXT_INT_LEVEL_TRIG_TYPE);
82
83     /* Set gateway Polarity (Active high) */
84     pspExtInterruptSetPolarity(uiSourceId, D_PSP_EXT_INT_ACTIVE_HIGH);
85
86     /* Clear the gateway */
87     pspExtInterruptClearPendingInt(uiSourceId);
88
89     /* Set IRQ4 priority */
90     pspExtInterruptSetPriority(uiSourceId, priority);
91
92     /* Enable IRQ4 interrupts in the PIC */
93     pspExternalInterruptEnableNumber(uiSourceId);
94
95     /* Register ISR */
96     G_Ext_Interrupt_Handlers[uiSourceId] = pTestIsr;
97 }

```

**Figür 9. *ExternalIntLine\_Initialization* işlevi**

**GPIO\_Initialization** işlevi, Figür 10, şu görevleri yerine getirir:

- Configure the GPIO pins as input/output and initialize the LEDs to 0 (lines 103 and 104).
- Configure the GPIO interrupts. (To further understand the functionality of each GPIO register, use the GPIO Core Specification, available at: [\[RVfpgaPath\]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/docs/gpio\\_spec.pdf](#))
  - o `RGPIO_INTE`: it determines which general-purpose pins generate an interrupt (line 107).
  - o `RGPIO_PTRIG`: it determines the edge that generates an interrupt (line 108).
  - o `RGPIO_INTS`: it clears the interrupts of all pins (line 109).
  - o `RGPIO_CTRL`: the least-significant bit of this register enables interrupt generation (line 110).

```

100 void GPIO_Initialization(void)
101 {
102     /* Configure LEDs and Switches */
103     M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF); /* GPIO_INOUT */
104     M_PSP_WRITE_REGISTER_32(GPIO_LEDS, 0x0); /* GPIO_LEDS */
105
106     /* Configure GPIO interrupts */
107     M_PSP_WRITE_REGISTER_32(RGPIO_INTE, 0x10000); /* RGPIO_INTE */
108     M_PSP_WRITE_REGISTER_32(RGPIO_PTRIG, 0x10000); /* RGPIO_PTRIG */
109     M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0); /* RGPIO_INTS */
110     M_PSP_WRITE_REGISTER_32(RGPIO_CTRL, 0x1); /* RGPIO_CTRL */
111 }

```

**Figür 10. *GPIO\_Initialization* işlevi.**

Son olarak ISR (bir diğer deyişle **GPIO\_ISR** işlevi, Figür 11) kesinti GPIO'da tetiklenince çalıştırılır. Bu ISR (Kesinti Hizmet Yordamı) şu görevleri yerine getirir:

- The current state of the LEDs is read (line 35).
- The LEDs are inverted and masked (lines 36-37).
- The LEDs are written with the new value (line 38).
- The GPIO interrupt is cleared (line 41).

- The IRQ4 external interrupt is cleared (line 44).

```

30 void GPIO_ISR(void)
31 {
32     unsigned int i;
33
34     /* Write the LED */
35     i = M_PSP_READ_REGISTER_32(GPIO_LEDS);          /* RGPI0_OUT */
36     i = !i;                                           /* Invert the LEDs */
37     i = i & 0x1;                                     /* Keep only the right-most LED */
38     M_PSP_WRITE_REGISTER_32(GPIO_LEDS, i);          /* RGPI0_OUT */
39
40     /* Clear GPIO interrupt */
41     M_PSP_WRITE_REGISTER_32(RGPI0_INTS, 0x0);        /* RGPI0_INTS */
42
43     /* Stop the generation of the specific external interrupt */
44     bspClearExtInterrupt(4);
45 }

```

**Figür 11. GPIO\_ISR işlevi.**

**GÖREV:** *LED-Switch\_7SegDispl\_Interrupts\_C-Lang* programını detaylıca anlamak için çözümü. Bölüm 4'ün gerçekleştirmesiyle kıyaslayıp, gerekirse programı adım adım çözümlemek için ayıklayıcıyı kullanabilirsin.

## 6. ALIŞTIRMALAR (İNGİLİZCE)

**Alıştırma 1.** Modify the *LED-Switch\_7SegDispl\_Interrupts\_C-Lang* program to include a second interrupt source, in this case generated by the timer. Recall that a timer can act as a PWM generator, timer, or counter, so it is generally referred to as a PTC unit.

- In RVfpga, the timer interrupt is connected to IRQ3 by setting bit 1 (*irq\_ptc\_enable*) of word 0x80001018 (see Figür 6).
- Create a function that initializes PTC interrupts, similar to *GPIO\_Initialization* in the previous example.
- Create a second ISR called *PTC\_ISR*. It should be similar to *GPIO\_ISR* in the *LED-Switch\_7SegDispl\_Interrupts\_C-Lang* program, but it should instead be invoked using IRQ3. *PTC\_ISR* should handle and clear the timer interrupt.

Once the program is implemented and debugged, use the PSP functions *pspExtInterruptsSetThreshold(threshold)* and *pspExtInterruptSetPriority(interrupt\_source, priority)* to analyse different combinations of the priorities and the threshold. Note that you can even change the priorities at execution time; for example, you can show the 7-segment displays count up to 10 and then stop counting by modifying the priority of the appropriate external interrupt source.

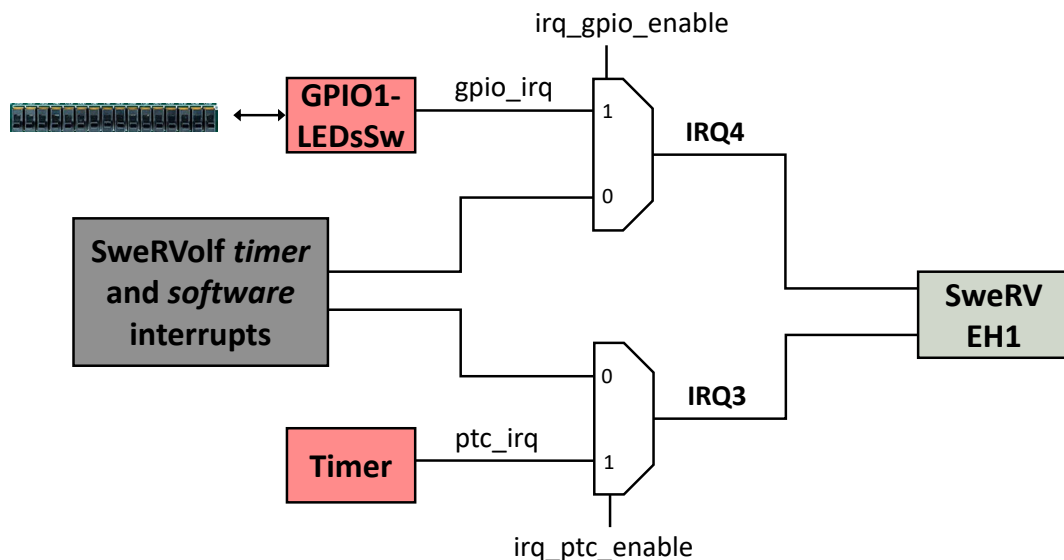
**Alıştırma 2.** Modify RVfpga to include a third interrupt source coming from the second GPIO that you designed in Lab 6 for controlling the on-board pushbuttons (GPIO2). Two approaches are possible for completing this exercise:

- You can connect the GPIO2 interrupt to an unused external interrupt source. SweRV EH1 provides up to 255 different interrupt lines and so far we have only used 2 of them. The drawback of this approach is that WD's libraries need to be modified.
- You can connect the GPIO2 interrupt to IRQ4, so that the GPIO module (that

connects to the LEDs and switches) and GPIO2 (that connects to the pushbuttons) use a single-vector interrupt mode. Although multi-vector mode is preferable under some situations, the advantage of this approach is that you can reuse the BSP.

We provide some guidance for the second approach by providing some details about the low-level implementation of interrupts in RVfpga.

Figür 12 shows the RVfpga circuit that connects the various interrupt sources (GPIO interrupt, timer interrupt – and the interrupt sources originally available in the SweRVolf core, which we do not analyse nor use here) with *IRQ4* and *IRQ3*. Specifically, *IRQ4* is connected to the GPIO when *irq\_gpio\_enable* = 1 (Figür 6), whereas *IRQ3* is connected with the timer when *irq\_ptc\_enable* = 1 (Figür 6). When *irq\_gpio\_enable* = *irq\_ptc\_enable* = 0, *IRQ4* and *IRQ3* are connected with the SweRVolf original interrupt sources, which we do not use in this lab (if you are interested in using these interrupt sources, you can view more information from <https://github.com/chipsalliance/Cores-SweRVolf>).



**Figür 12. Logic implementation: connection of GPIO and timer interrupts with IRQ4 and IRQ3 respectively**

Figür 13 shows the Verilog region of module **swervolf\_core** that implements the connection between the interrupt sources and *IRQ4* and *IRQ3*. The GPIO interrupt is connected with *IRQ4* when signal *irq\_gpio\_enable* is 1 (top part of the red box). The timer interrupt is connected to *IRQ3* when signal *irq\_ptc\_enable* is 1 (bottom part of the red box). When both signals are 0 (code not highlighted in the figure), the interrupt sources implemented in SweRVolf are connected to *IRQ3* and *IRQ4*.

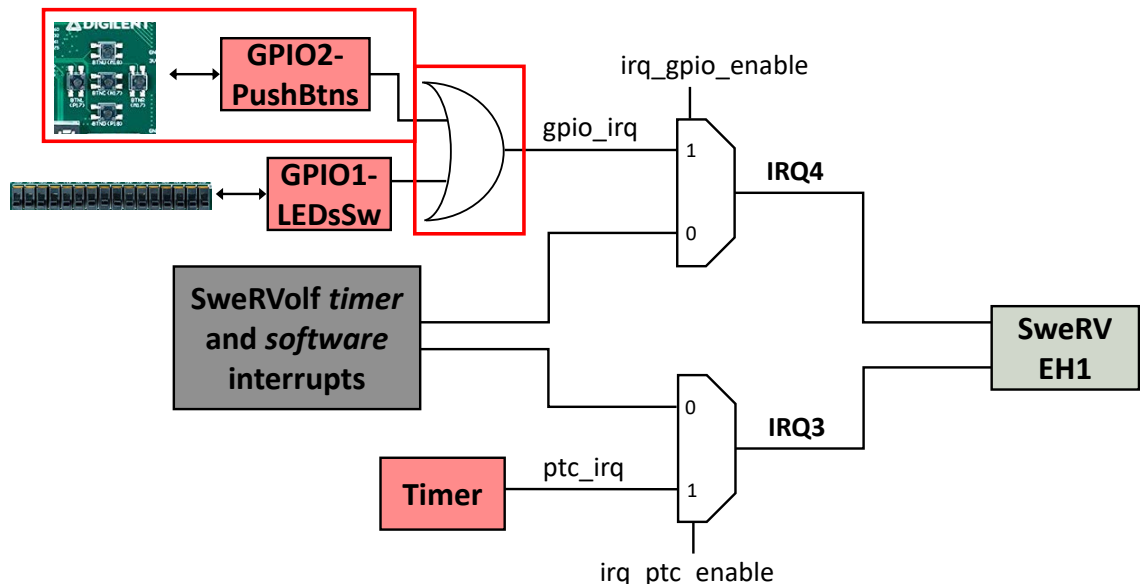
```

123 always @(posedge i_clk) begin
124     o_wb_ack <= i_wb_cyc & !o_wb_ack;
125
126     nmi_int    <= 1'b0;
127     nmi_int_r <= nmi_int;
128
129     // GPIO Interrupt through IRQ4. Enable by setting bit 0 of word 0x80001018
130     if (irq_gpio_enable & gpio_irq) begin
131         sw_irq4 <= 1'b1;
132     end
133
134     // Timer (PTC) Interrupt through IRQ3. Enable by setting bit 1 of word 0x80001018
135     if (irq_ptc_enable & ptc_irq) begin
136         sw_irq3 <= 1'b1;
137     end
138
139     // SweRVolf simple timer and software interrupts. Enable by resetting bits 0 and 1 of word 0x80001018
140     if (!irq_gpio_enable & !irq_ptc_enable) begin
141
142         if (sw_irq3_edge)
143             sw_irq3 <= 1'b0;
144         if (sw_irq4_edge)
145             sw_irq4 <= 1'b0;
146
147         if (irq_timer_en)
148             irq_timer_cnt <= irq_timer_cnt - 1;
149
150         if (irq_timer_cnt == 32'd1) begin
151             irq_timer_en <= 1'b0;
152             if (sw_irq3_timer)
153                 sw_irq3 <= 1'b1;
154             if (sw_irq4_timer)
155                 sw_irq4 <= 1'b1;
156             if (!(sw_irq3_timer | sw_irq4_timer))
157                 nmi_int <= 1'b1;
158         end
159     end
160 end

```

**Figür 13. Verilog gerçekleştirilmesi: highlighted in red, connection of GPIO and timer interrupts with IRQ4 and IRQ3, respectively.**

In this exercise you must extend the previous implementation (Figür 12) to include a new interrupt source connected to *IRQ4* as shown in Figür 14.



**Figür 14. Logic implementation: connection of a second interrupt source (provided by the GPIO that reads the pushbuttons) with IRQ4**

We highlight a few other Verilog regions that you should also understand, although you do not need to modify them in this example.

- The interrupt sources are inserted into the SweRV processor at line 599 of



the **swervolf\_core** module (Figür 15). Although four interrupt sources are available, in this lab we are only interested in sources *sw\_irq4*, and *sw\_irq3*.

```
599 .extintsrc_req ({4'd0, sw_irq4, sw_irq3, spi0_irq, uart_irq}),
```

**Figür 15. Interrupt sources sent to SweRV**

- The enable signals, *irq\_gpio\_enable* and *irq\_ptc\_enable* (accessible at address 0x80001018, see Figür 6), are written by the core at lines 192-196 of the **swervolf\_syscon** module (Figür 16).

```
192 6: begin //0x18-0x1B
193     if (i_wb_sel[0])
194         irq_gpio_enable <= i_wb_dat[0];
195         irq_ptc_enable <= i_wb_dat[1];
196     end
```

**Figür 16. Writing of register 0x80001018 from the SweRV core**

These enable signals, *irq\_gpio\_enable* and *irq\_ptc\_enable*, are read at lines 248-249 by the **swervolf\_syscon** module from the core (see Figür 17).

```
248 //0x18-0x1B
249 6 : o_wb_rdt <= {30'd0, irq_ptc_enable, irq_gpio_enable};
```

**Figür 17. Reading of register 0x80001018 into the SweRV core**

**Alıştırma 3.** Use the extended RVfpga version that you designed in the previous exercise to implement a C program that displays an increasingly incrementing binary count on the LEDs, starting at 1. Create a delay with the timer, using interrupts, for waiting between displaying each incremented value so that the values are viewable by the human eye. Read BTNC and use it to change the speed of the count, and read Switch[0] and use it to restart the count whenever it is pressed.

With your extended RVfpga from Exercise 2, you now have three possible interrupt sources:

- **GPIO** (interrupts from the switches)
- **GPIO2** (interrupts from the buttons, that you designed in the previous exercise, Exercise 2)
- **PTC** (the timer)

Given that the extended RVfpga implementation from Exercise 2 has two interrupt sources that share the same line (*IRQ4*), the corresponding Interrupt Service Routine (*GPIO\_ISR*) has to identify the device that generated the interrupt. You can obtain that information from the GPIO registers.



## EK (İNGİLİZCE)

This appendix describes how the SweRV EH1 Core's Programmable Interrupt Controller (PIC) manages external interrupts at a register level. The PIC uses the memory-mapped registers shown in Tablo 2. It must be noted that the PIC memory space starts at address 0xF00C0000; This address is referred to as *RV\_PIC\_BASE*. Addresses are given relative to this base address.

**Tablo 2. PIC Memory-mapped Register Address Map**

Name	Addresses (relative to <i>RV_PIC_BASE</i> )	Description	Location at the manual
meipIS	$0x0004 - 0x0004 + S_{max} * 4 - 1$	External interrupt priority level register	Tablo 6-2 of [PRM v1.7]
meipX	$0x1000 - 0x1000 + (X_{max} + 1) * 4 - 1$	External interrupt pending register	Tablo 6-3 of [PRM v1.7]
meieS	$0x2000 - 0x2000 + S_{max} * 4 - 1$	External interrupt enable register	Tablo 6-4 of [PRM v1.7]
mpiccfg	0x3000 – 0x3003	External interrupt PIC configuration register	Tablo 6-1 of [PRM v1.7]
meigwctrlS	$0x4004 - 0x4004 + S_{max} * 4 - 1$	External interrupt gateway configuration register (for configurable gateways only)	Tablo 6-11 of [PRM v1.7]
meigwclrS	$0x5004 - 0x5004 + S_{max} * 4 - 1$	External interrupt gateway clear register (for configurable gateways only)	Tablo 6-12 of [PRM v1.7]

All registers are 32 bits wide and are accessible through load and store instructions, as usual for memory-mapped I/O. The access type depends on the specific bits that we want to access (this can be viewed at [PRM v1.7]).

Some of the registers have parameterized names, which end in S or X. Several instances of these registers can exist. The parameter S refers to the number of external interrupt sources, which in SweRV EH1 is equivalent to the number of gateways. Thus, registers ending with 'S' have 1 to 255 register instances available. In this lab we only use 2 external interrupt sources: **IRQ3** (associated with the timer), and **IRQ4** (associated with the GPIO). The parameter X refers to a group of 32 gateways. This does not mean that the gateways are grouped, but grouping them reduces the size of required memory for certain 32-bit registers where 1 bit is enough for performing an action on a group of external interrupt sources. Such is the case of the external interrupt pending register, where one bit is enough to distinguish whether or not the interrupt has been serviced. In order to get more information about these registers, the rightmost column of Tablo 1 points to the place within [PRM v1.7] where the bit-level (specific interrupt) description is contained.

Besides the registers shown in Tablo 2, the PIC contains Control and Status Registers (CSRs). The standard RISC-V ISA establishes a 12-bit encoding space (*csr[11:0]*) for up to 4,096 CSRs. By convention, the upper 4 bits of the CSR address (*csr[11:8]*) are used to encode the read and write accessibility of the CSRs according to privilege level. The top two bits (*csr[11:10]*) indicate whether the register is read/write (00, 01, or 10) or read-only (11). The next two bits (*csr[9:8]*) encode the lowest privilege level that can access the CSR. More information about the CSRs is available in [PRM v1.7] and [ISM v1.11]. Tablo 3 lists those CSRs that are useful for managing the external interrupts in the SweRV EH1 core. These are accessible through dedicated load and store instructions such as *csrrw* or *csrrs* (CSR read/write and CSR read/set).

**Tablo 3. PIC Non-standard RISC-V CSR Address Map.**

Name	Number	Description	Location
meivt	0xBC8	External interrupt vector table register	Tablo 6-6 of [PRM v1.7]
meipt	0xBC9	External interrupt priority threshold register	Tablo 6-5 of [PRM v1.7]
meicpct	0xBCA	External interrupt claim ID / priority level capture trigger register	Tablo 6-8 of [PRM v1.7]
meicidpl	0xBCB	External interrupt claim ID's priority level register	Tablo 6-9 of [PRM v1.7]
meicurpl	0xBCC	External interrupt current priority level register	Tablo 6-10 of [PRM v1.7]
meihap	0xFC8	External interrupt handler address pointer register	Tablo 6-7 of [PRM v1.7]
mie	0x304	Machine interrupt enable register	Tablo 11-1 of [PRM v1.7]
mstatus	0x300	Machine status register	Figür 3.7 of [ISM v1.11]

The right-most column on Tablo 3 points to the place in [PRM v1.7] or [ISM v1.11] where bit-level information is described for the given CSR (note that the *mstatus* bits description is not provided in [PRM v1.7] but in [ISM v1.11] instead).

## A. External Interrupt Configuration

In this subsection we summarize the basic steps needed to configure an external interrupt using the aforementioned registers:

1. Disable all external interrupts by clearing bit *miep* within the *mie* CSR.
2. Configure the priority order by writing the *prord* bit of the *mpiccfg* register.
3. In multi-vector mode, if not configured, set the base address of the external vectored interrupt address table by writing the base field of the *meivt* register.
4. Set the priority threshold by writing the *prithresh* field of the *meipt* register.
5. Initialize the nesting priority thresholds by writing '0' (or '15' for reversed priority order) to the *clidpri* field of the *meicidpl* and the *currpri* field of the *meicurpl* registers.
6. For each configurable gateway *S*, set the polarity (active-high/active-low) and type (level-triggered/edge-triggered) in the *meigwctrlS* register and clear the IP bit by writing to the gateway's *meigwclrS* register.
7. In multi-vector mode, for each external interrupt source *S*, write the address of the corresponding handler in the external vectored interrupt address table.
8. Set the priority level for each external interrupt source *S* by writing the corresponding priority field of the *meiplS* registers.
9. Enable interrupts for the appropriate external interrupt sources by setting the *inten* bit of the *meieS* registers for each interrupt source *S*.
10. Activate the *mei* bit within the *mstatus* CSR.
11. Enable all external interrupts by setting bit *miep* within the *mie* CSR.

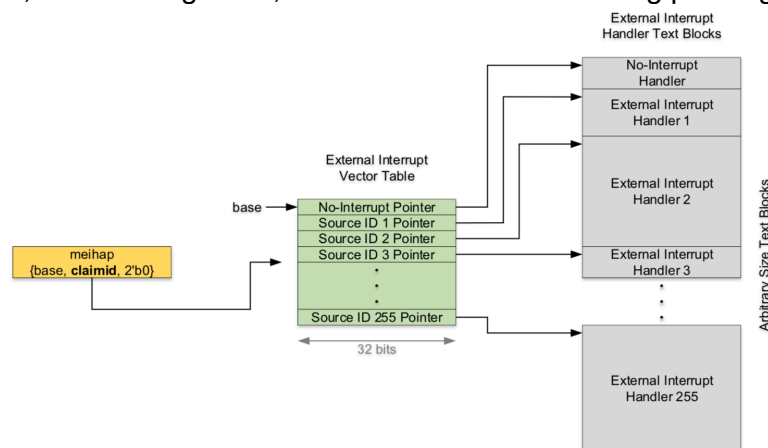
These are the general steps for *S* gateways. However, in RVfpga we only use 2 interrupt sources (IRQ3 and IRQ4), each of which has its own gateway. Furthermore, it must be noted that the order is not fully strict, as some actions are interchangeable (e.g., step 4 can be

completed prior to step 2). Besides, because each function calls `pspInterruptsDisable` upon entry, step 1 is not strictly needed.

## B. External Interrupt Operating Mode

In this subsection we describe how the PIC operates once an external interrupt is triggered. Once the desired event occurs on the external interrupt line (wire), the following actions take place:

1. The PIC decides which pending interrupt possesses the highest priority.
2. When the target hart (hardware thread) takes the external interrupt, it disables all interrupts (i.e., it clears the *mie* bit in the RISC-V hart's *mstatus* register) and jumps to the external interrupt handler.
3. The external interrupt handler writes to the *meicpct* register to trigger the capture of the interrupt source ID of the highest priority external interrupt that is pending (in the *meihap* register) and its corresponding priority (in the *meicidpl* register).
4. The handler then reads the *meihap* register to obtain the interrupt source ID provided in the *claimid* field. Based on the contents of the *meihap* register, the external interrupt handler jumps to the handler specific to this external interrupt source. This can be observed in Figür 18.
5. The source-specific interrupt handler (ISR) services the external interrupt, and then:
  - a. For level-triggered interrupt sources, the interrupt handler clears the state in the SoC IP which initiated the interrupt request.
  - b. For edge-triggered interrupt sources, the interrupt handler clears the IP bit in the source's gateway by writing to the *meigwclrS* register.
 This deasserts the source's interrupt request.
6. Meanwhile, in the background, the PIC continues evaluating pending interrupts.



**Figür 18. Vectored External Interrupts (taken from [PRM v1.7])**

It must be noted that this is regular operation mode. Nested interrupts (a maximum of 15) are also supported in the SweRV EH1 core. For more information, please consult the [PRM v1.7].