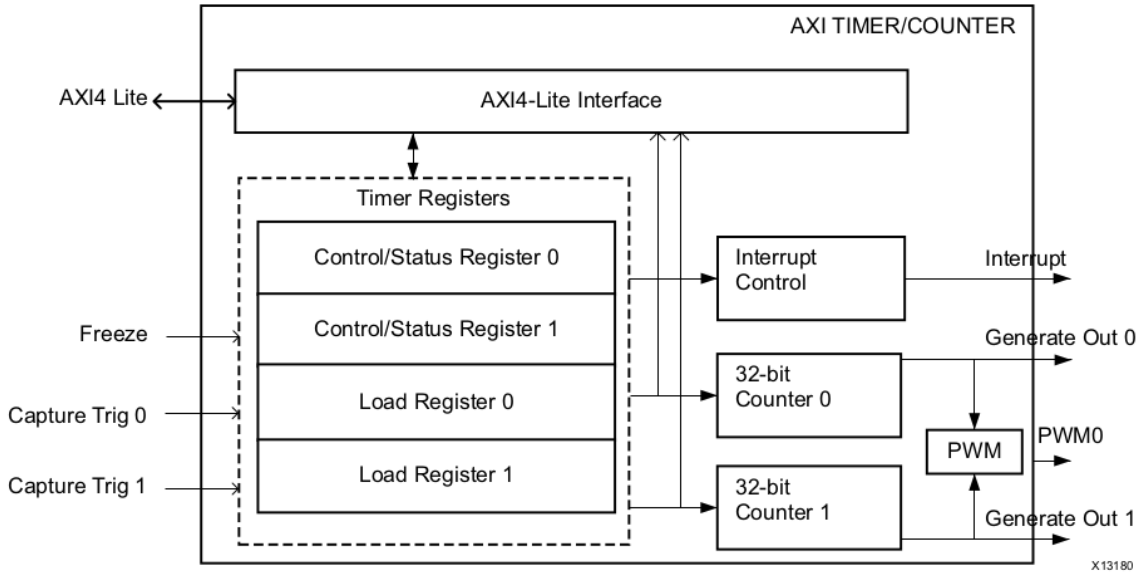
AXI Timer/Counter IP

UVOD

AXI Timer/Counter je jezgro koje predstavlja dva 32-bitna ili jedan 64-bitni brojački modul koji komunicira sa ostatkom sistema putem AXI4-Lite interfejsa. AXI Timer je uređen kao dva identična brojačka modula kao što je prikazano na blok šemi (slika 1). Svi registri su 32-bitni, te svaki brojač ima sopstveni Load, Control/Status i Counter registar. **Control/Status** se koristi za podešavanja brojača, kao i očitavanje statusnih signala (flags). Svaki bit unutar ovog registra ima posebnu funkcionalnost. **Counter** registar čuva trenutnu vrednost brojača, koji može biti konfigurisan da broji na više ili na niže. Brojač ima više režima rada pri čemu se uloga **Load** registra menja u zavisnosti od odabranog režima. Interrupt signali brojača su dovedeni na ILI (OR) logičko kolo tako da iz IP jezgra izlazi samo jedana, univerzalna prekidna linija. PWM blok pravi impulsno širinsku modulaciju na izlaznom portu PWM0, gde se frekvencija i faktor ispune mogu proizvoljno podesiti. PWM mod koristi brojač 0 (Counter 0) za generisanje periode, a brojač 1 za generisanje impulsa.



Slika 1: Blok dijagram brojačkog jezgra

1. Režimi rada

Oba brojača podržavaju četiri različita režima rada:

1.1.1. Generate mod

U *Generate* modu, vrednost *Load* registra je inicijalna vrednost brojača koja se kopira u Counter registar. Brojač kreće da broji od te vrednosti na više ili na niže u zavisnosti od UDT bita u Control/Status registru. Kada brojač dosegne maksimalnu ili minimalnu vrednost (*overflow, underflow*), izlazni signal *Generate out* se postavlja

na visoku vrednost za dužinu jednog takta, te se prekidni (*interrupt*) signal postavi na visoku vrednost (*TINT* bit u *Control/Status* registru). Prekidni signal je potrebno ručno vratiti na nulu upisom jedinice u *TINT* bit. Ukoliko je *AutoReload/Hold* bit u *Control/Status* registru postavljen na 1, vrednost iz *Load* registra se ponovno prenosi u *Counter* registar, te brojač kreće da broji iznova. Ovaj mod se može koristiti ukoliko je potrebno generisati periodične prekide ili eksterne signale.

1.1.2. Capture mod

U Capture modu, brojač broji u punom opsegu 2^{32}, te u trenutku kada se postavi ulazni signal Capture, vrednost brojača *Counter* se sačuva u *Load* registar. Kada se detektuje ulazni signal, generiše se prekidni signal (*TINT* u *Control/Status* registru se postavlja na 1). *AutoReload/Hold* bit u *Control/Status* registru kontroliše da li je moguće prebrisati vrednost Load registra pre nego što se resetuje TINT signal. Ovaj mod se može koristiti ukoliko je potrebno zapamtiti vreme kada se desio neki eksterni dogadjaj.

1.1.3. PWM mod

U PWM modu, oba brojača se koriste kako bi se generisala impulsno širinska modulacija na izlaznom portu PWM0. Moguće je postaviti željenu frekvenciju i faktor ispune. Brojač 0 definiše dužinu periode, dok brojač 1 definiše dužinu impulsa.

1.1.4. Cascade mod

U kaskadnom modu, dva 32-bitna brojača se spajaju u jedan 64-bitni. Kaskadni brojač može raditi u *Generate* ili *Capture* modu. Control/Status registar brojača 0 se koristi za kontrolu kaskadnog brojača. Ovaj mod se može koristiti kada je potreban neki od prva dva moda ali nad dužim vremenskim intervalom.

1.2. Opis registara

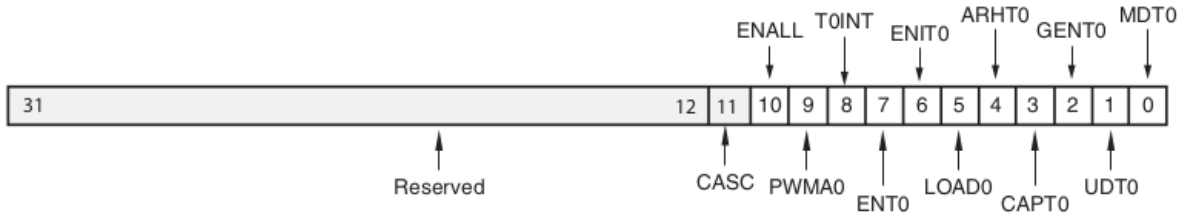
Adresni opseg registara ovog IP jezgra je dat u tabeli na slici 2.

Address Offset	Register Name	Description
0h	TCSR0	Timer 0 Control and Status Register
04h	TLR0	Timer 0 Load Register
08h	TCR0	Timer 0 Counter Register
0Ch-0Fh	RSVD	Reserved
10h	TCSR1	Timer 1 Control and Status Register
14h	TLR1	Timer 1 Load Register
18h	TCR1	Timer 1 Counter Register
1Ch-1Fh	RSVD	Reserved

Slika 2. Registri brojačkog IP jezgra

1.2.1. *Control/Status* registri 0 (TCSR0 i TSCR1)

Sledeća slika prikazuje raspored i imena bitova kod *Control/Status* registra za brojač 0 (*TCSR0*).

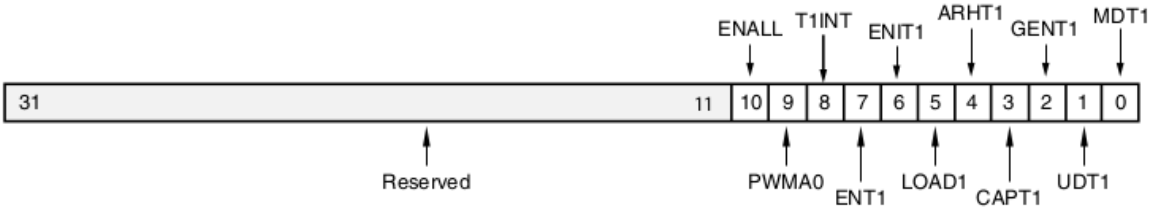


Slika 3. Raspored bita TCSR0 registra

U sledećoj tabeli su naznačene funkcionalnosti svakog bita:

BIT	IME	Funkcija
31:12	Rezervisani	Ne koriste se
11	CASC	1 = Aktivira kaskadni mod dva brojača
10	ENALL	1 = Signal dozvole za oba brojača, postavlja ENT bite
9	PWMA0	1 = Aktivira PWM mod brojača 0
8	T0INT	Čitanje : 1 = Prekid se desio 0 = Prekid se nije desio Upis: 1 = Postavi na nula (izbriši prekid) 0 = Ništa se ne desi
7	ENT0	1 = Signal dozvole (*enable*) za brojač 0
6	ENIT0	1 = Signal dozvole (*enable*) za prekid brojača 0
5	LOAD0	1 = Prebaci vrednost iz *Load* registra u *Counter* registar
4	ARHT0	1 = Automatsko upisivanje vrednosti Load registra u Counter registar kada se desi *overflow*/*underflow*
3	CAPT0	1 = Signal dozvole (*enable*) za eksterni *capture* signal
2	GENT0	1 = Signal dozvole (*enable*) za eksterni *generate* signal
1	UDT0	1 = Brojanje na više; 0 = Brojanje na niže
0	MDT0	1 = *Generate* mod 0 = *Capture* mod

TCSR 1 ima iste funkcionalnosti bita kao i TCSR0 registar. Jedina razlika je što se u kaskadnom modu ovaj registar koristi samo za upisivanje u TLR1 registar. Raspored bita se može videti na sledećoj slici:



Slika 4. Raspored bita TCSR 1 registra

1.2.2. *Load* registri (*TLR0* i *TLR1*)

Load registri sadrže 32-bitnu vrednost koja se menja u zavisnosti od režima rada. U generate modu je to inicijalna vrednost od koje brojač kreće da broji. U capture modu ona čuva vrednost koja je bila u *Counter* registru kada se desio eksterni signal *capture*. U kaskadnom modu, TLR0 sadrži niža 32 bita dok TLR1 sadrži viša 32 bita.

1.2.3. *Timer/Counter* registri (*TCR0* i *TCR1*)

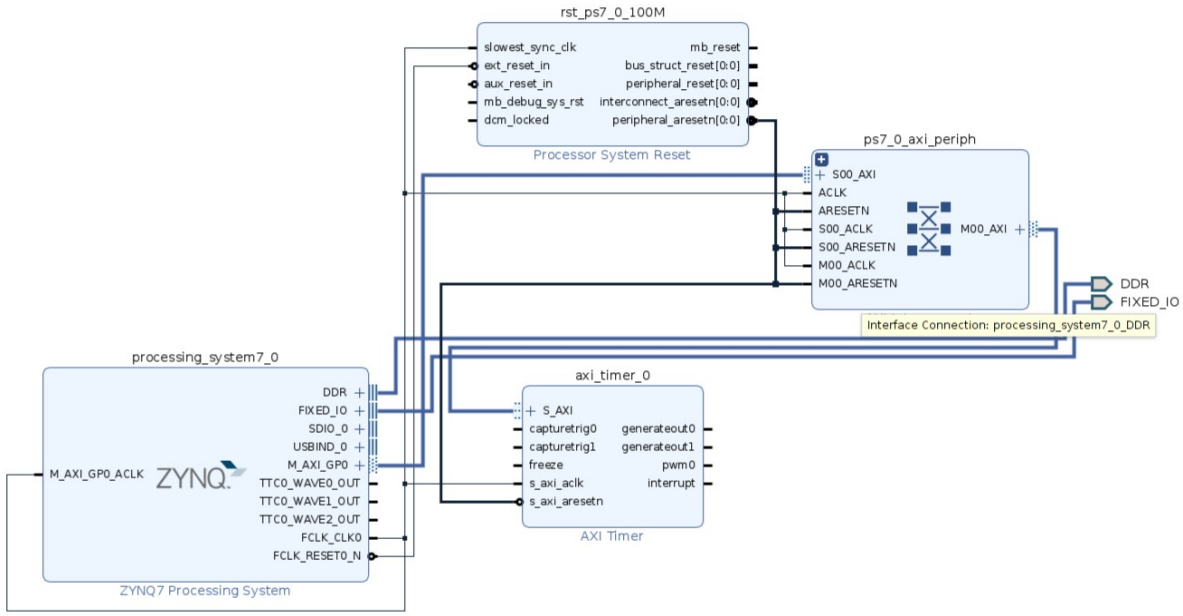
Timer/Counter su 32-bitni registri koji čuvaju trenutnu vrednost brojača. U kaskadnom modu TCR0 sadrži niža 32 bita dok TCR1 sadrži viša 32 bita.

2 Vivado IP integrator

AXI timer IP se dodaje na isti način kao i jezgra iz prethodnih materijala:

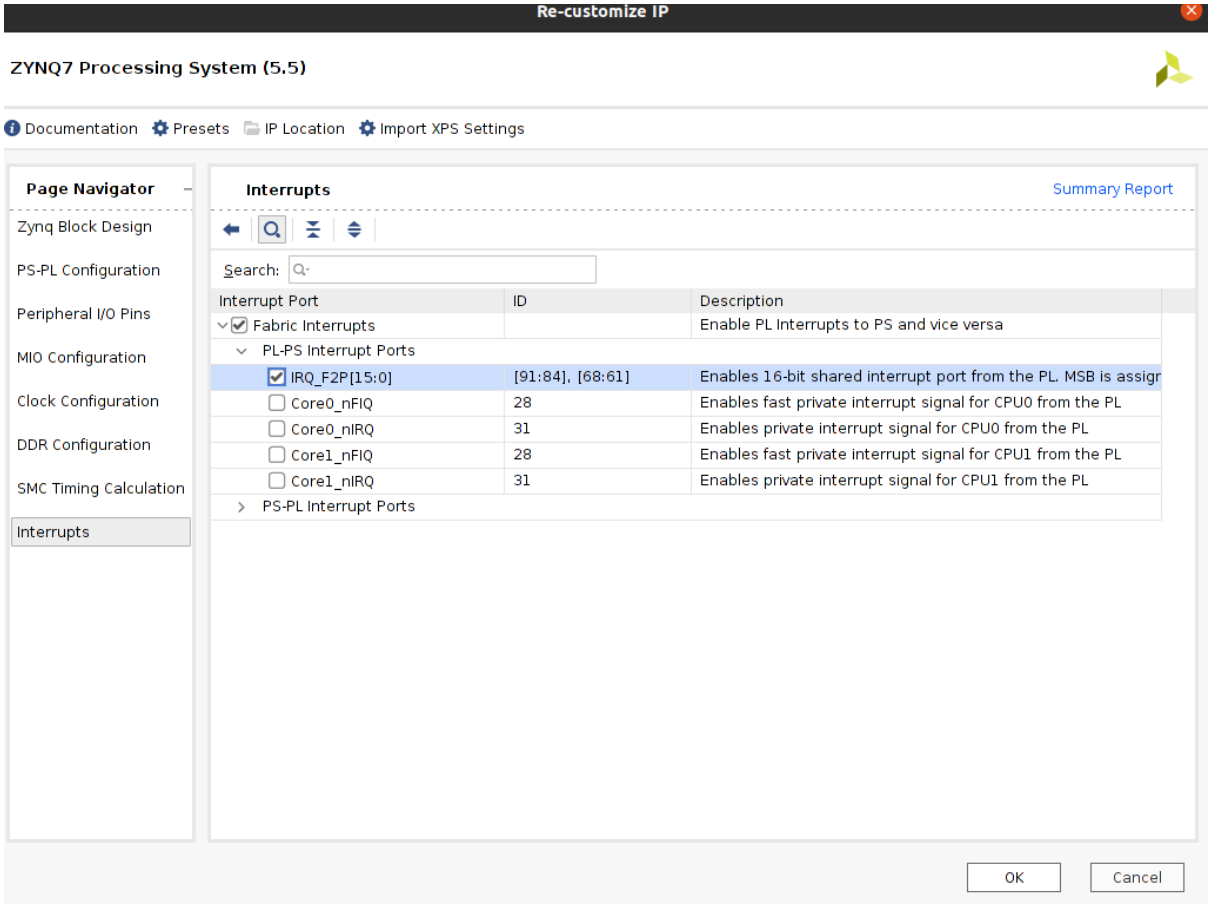
Desni klik -> Add IP -> AXI Timer

Kada se pokrenu opcije za automatskom povezivanje (takođe prikazane u prethodnom materijalu) IP integrator treba da ima sledeći izgled:



Slika 5. Izgled IP integratora nakon dodavanja AXI Timer jezgra

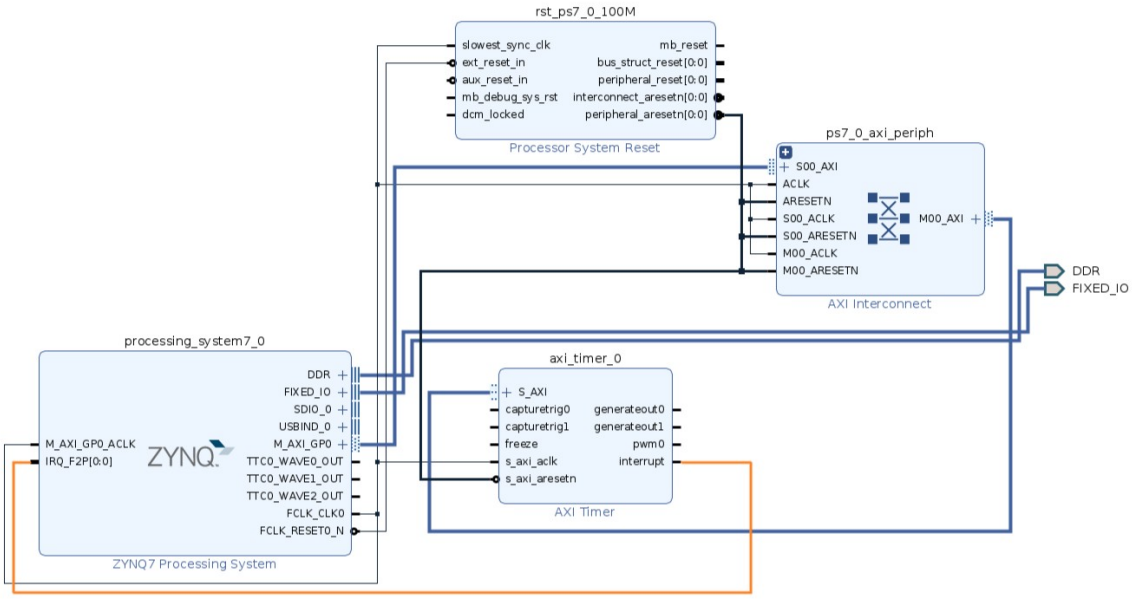
Može se primetiti da AXI timer modul ima prethodno pomenute ulaze i izlaze, ali za sada samo treba povezati njegov interrupt izlaz sa procesorom. Kako bi se to omogućilo neophodno je dodatno konfigurisati procesorsko jezgro na sledeći način:



Slika 6. Konfigurisanje Zynq procesorskog bloka

Na procesorskom jezgru će se sada pojaviti ulazni port za interapte na koji treba povezati interapt AXI timer modula:

-



Slika 7. Povezivanje interapt izlaza AXI timer modula sa procesorom

3 VITIS

U nastavku je dat kod pomoću koga se Tajmersko jezgro konfiguriše tako da generiše prekid nakon 5000ms.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xil_exception.h"
#include "xscugic.h"
#include "sleep.h"

// Timer base Addres, can also be found in address editor in vivado
// but this way it is automaticaly generated in xparameters.h file
// when platform project is created
#define TIMER_BASEADDR           XPAR_AXI_TIMER_0_BASEADDR

// Can also be found in xparameters.h file. Every interrupt has its number
#define TIMER_INTERRUPT_ID             XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR

// Processor core also has its own device ID which is used to enable
// interrupts
#define INTC_DEVICE_ID           XPAR_PS7_SCUGIC_0_DEVICE_ID

// REGISTER CONSTANTS used to address correct registers
// inside AXI timer IP
#define XIL_AXI_TIMER_TCSR_OFFSET       0x0
#define XIL_AXI_TIMER_TLR_OFFSET        0x4
#define XIL_AXI_TIMER_TCR_OFFSET        0x8

// Constants used to setup AXI Timer registers
#define XIL_AXI_TIMER_CSR_CASC_MASK            0x00000800
#define XIL_AXI_TIMER_CSR_ENABLE_ALL_MASK      0x00000400
#define XIL_AXI_TIMER_CSR_ENABLE_PWM_MASK      0x00000200
#define XIL_AXI_TIMER_CSR_INT_OCCURED_MASK     0x00000100
#define XIL_AXI_TIMER_CSR_ENABLE_TMR_MASK      0x00000080
#define XIL_AXI_TIMER_CSR_ENABLE_INT_MASK      0x00000040
#define XIL_AXI_TIMER_CSR_LOAD_MASK            0x00000020
#define XIL_AXI_TIMER_CSR_AUTO_RELOAD_MASK     0x00000010
#define XIL_AXI_TIMER_CSR_EXT_CAPTURE_MASK     0x00000008
#define XIL_AXI_TIMER_CSR_EXT_GENERATE_MASK 0x00000004
#define XIL_AXI_TIMER_CSR_DOWN_COUNT_MASK      0x00000002
#define XIL_AXI_TIMER_CSR_CAPTURE_MODE_MASK 0x00000001

// Funtion Prototypes
u32 Setup_Interrupt(u32 DeviceId, Xil_InterruptHandler Handler, u32
interrupt_ID);
void Timer_Interrupt_Handler();
static void Setup_And_Start_Timer(unsigned int milliseconds);
// Global Variables
volatile int timer_intr_done = 0;
int main()
{
```

```
    int status;
    int data;

    init_platform();
    status = Setup_Interrupt(INTC_DEVICE_ID,
(Xil_InterruptHandler)Timer_Interrupt_Handler, TIMER_INTERRUPT_ID);

    print("Hello Timer\n\r");

    Setup_And_Start_Timer(5000); // Timer will generate an interrupt after 5
seconds

    while(!timer_intr_done)
    {
       sleep(1);
       print("Waiting... \n\r");
    }

       // Clear Interrupt Flag
    data = Xil_In32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET);
    Xil_Out32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET, (data |
XIL_AXI_TIMER_CSR_INT_OCCURED_MASK));
       // Disable Timer
    data = Xil_In32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET);
    Xil_Out32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET, data &
~(XIL_AXI_TIMER_CSR_ENABLE_TMR_MASK));

    print("Successfully ran Hello Timer application");
    cleanup_platform();
    return 0;
}


// Initialize Interrupt Controller
u32 Setup_Interrupt(u32 DeviceId, Xil_InterruptHandler Handler, u32 interrupt_ID)
{
   XScuGic_Config *IntcConfig;
   XScuGic INTCInst;
   int status;
   // Extracts informations about processor core based on its ID, and they are
used to setup interrupts
   IntcConfig = XScuGic_LookupConfig(DeviceId);
   // Initializes processor registers using information extracted in the previous
step
   status = XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig-
>CpuBaseAddress);
   if(status != XST_SUCCESS) return XST_FAILURE;
   status = XScuGic_SelfTest(&INTCInst);
   if (status != XST_SUCCESS) return XST_FAILURE;
   // Connect Timer Handler And Enable Interrupt
   // The processor can have multiple interrupt sources, and we must setup
trigger and   priority
   // for the our interrupt. For this we are using interrupt ID.
   XScuGic_SetPriorityTriggerType(&INTCInst, interrupt_ID, 0xA8, 3);
   // Connects out interrupt with the appropriate ISR (Handler)
   status = XScuGic_Connect(&INTCInst, interrupt_ID, Handler, (void *)&INTCInst);
   if(status != XST_SUCCESS) return XST_FAILURE;
```

```
   // Enable interrupt for out device
   XScuGic_Enable(&INTCInst, interrupt_ID);
   //Two lines bellow enable exeptions
   Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
(Xil_ExceptionHandler)XScuGic_InterruptHandler,&INTCInst);
   Xil_ExceptionEnable();

   return XST_SUCCESS;
}
// Timer Interrupt Handler
void Timer_Interrupt_Handler()
{
   timer_intr_done = 1;
   return;
}

static void Setup_And_Start_Timer(unsigned int milliseconds)
{
       // Disable Timer Counter
   unsigned int timer_load;
   unsigned int zero = 0;
   unsigned int data = 0;
   // Line bellow is true if timer is working on 100MHz
   timer_load = zero - milliseconds*100000;

   // Disable timer/counter while configuration is in progress
   data = Xil_In32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET);
   Xil_Out32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET, (data &
~(XIL_AXI_TIMER_CSR_ENABLE_TMR_MASK)));
   // Set initial value in load register
   Xil_Out32(TIMER_BASEADDR + XIL_AXI_TIMER_TLR_OFFSET, timer_load);
   // Load initial value into counter from load register
   data = Xil_In32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET);
   Xil_Out32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET, (data |
XIL_AXI_TIMER_CSR_LOAD_MASK));
   // Set LOAD0 bit from the previous step to zero
   data = Xil_In32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET);
   Xil_Out32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET, (data &
~(XIL_AXI_TIMER_CSR_LOAD_MASK)));
   // Enable interrupts and autoreload, reset should be zero
   Xil_Out32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET,
(XIL_AXI_TIMER_CSR_ENABLE_INT_MASK | XIL_AXI_TIMER_CSR_AUTO_RELOAD_MASK));
   // Start Timer by setting enable signal
   data = Xil_In32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET);
   Xil_Out32(TIMER_BASEADDR + XIL_AXI_TIMER_TCSR_OFFSET, (data |
XIL_AXI_TIMER_CSR_ENABLE_TMR_MASK));
}
```

Kodni listing 1. Primer programiranja tajmer jezgra

U ovom primeru je bitno obratiti pažnju na 3 funkcije čiji su prototipi dati u nastavku:

```
u32 Setup_Interrupt(u32 DeviceId, Xil_InterruptHandler Handler, u32
interrupt_ID);
void Timer_Interrupt_Handler();
static void Setup_And_Start_Timer(unsigned int milliseconds);
```

Setup_Interrupt konfiguriše procesorsko jezgro tako da može da prepozna prekid izazvan od strane komponenti iz FPGA dela čipa. Ona prima 3 parametra:

- Device ID - Jedinstveni broj procesorskog jezgra definisan u xparameters.h fajlu preko XPAR_PS7_SCUGIC_0_DEVICE_ID makroa.
- Xil_InterruptHandler - pokazivač na prekidnu rutinu (funkciju)
- Interrupt ID - jedinstveni broj prekida usled koga treba da se pozove prekidna rutina. Svaki uređaj povezan na *interrupt* port procesorskog jezgra dobija ovaj broj koji može da se izvuče iz xparameters.h fajla. Za slučaj tajmera, on se automatski generiše prilikom pravljenja platformskog projekta i pristupa mu se preko XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR makroa

Ova funkcija koristi XSugic biblioteku kompanije Xilinx koja omogućava konfigurisanje procesorskog jezgra tako da može da prepozna prekid. **Nju ne treba modifikovati već samo prilikom njenog poziva menjati parametre koji joj se prosleđuju kako bi se podesilo detektovanje određenog prekida.**

Timer_interrup_Handler funkcija je prekidna rutina koja se poziva kada se detektuje prekid. **Setup_And_Start_Timer** je funkcija koja konfiguriše AXI timer jezgro. U ovom primeru konfiguriše se load registar AXI timer jezgra kako bi se podesilo odakle tajmer broji, omogućavaju se prekidi i pokreće se tajmer.