

**THE IMAGINATION UNIVERSITY PROGRAMME**

**RVfpga Lab 3**

**Chamadas de Funções**

# INTRODUÇÃO

As chamadas a funções são uma parte crítica de qualquer programa pois permitem modularidade e reutilizar código, e por consequência, facilitar escrever e depurar programas mais facilmente. A linguagem de programação C também inclui bibliotecas padrão, como também bibliotecas específicas para processadores/placas, de funções usadas frequentemente em C, tal como geração de números aleatórios, e funções matemáticas típicas. Funções em alto-nível são traduzidas para Assembly seguindo uma convenção de chamada ou *Calling Convention*. Este lab mostra como escrever e usar funções em programas em C – tanto as funções escritas pelo programador como também as funções contidas nas bibliotecas C. Também mostra como as funções são implementadas em linguagem Assembly. No final deste lab, propomos exercícios sobre escrita de programas que usam funções e chamadas a bibliotecas.

# Escrever um programa em C que usa funções

Uma função – também pode ser chamada de subrotina ou procedimento – é código que é agregado num bloco de código e que tem um uso (entradas e saídas) e comportamento bem definidos. Esta modularidade aumenta a eficiência através da redução da complexidade e promover a reutilização de código. Uma função pode ser invocada em qualquer parte do programa de tal forma que, quando a unção termina, a execução do programa é retomada imediatamente após a chamada da função. As funçõe spodem ser chamadas por outras funções (funções encadeadas ou *nested*), ou até pela mesma função (chamadas funções *recursivas*).

Para escrever um programa RISC-V com funções, deve seguir os mesmos passos descritos nos Labs 2 e 3:

1. Crie um projeto RVfpga
2. Escreva um programa em C
3. Configurar o RVfpgaNexys na placa FPGA Nexys A7 (lembre-se que também pode corer estes programas via simulação, usando o Verilator ou o Whisper)
4. Compile, descarregue, e corra/depure o programa

Refira-se ao Lab 2 para instruções detalhadas para estes passos. Em baixo encontra-se uma breve descrição de cada passo.

Passo 1. Criar um projeto RVfpga

Crie um projeto chamado project1 na seguinte pasta:

[RVfpgaPath]/RVfpga/Labs/Lab03

Passo 2. Escrever um programa em C

Agora irá adicionar um ficheiro com um programa em C ao projeto. Crie um novo ficheiro e escreva ou copie o seguinte programa em C no projeto. Este programa também está disponível no seguinte ficheiro:

[RVfpgaPath]/RVfpga/Labs/Lab03/LedsSwitches\_functions.c

// Endereços de E/S mapeados em memória

#define GPIO\_SWs 0x80001400

#define GPIO\_LEDs 0x80001404

#define GPIO\_INOUT 0x80001408

#define READ\_GPIO(dir) (\*(volatile unsigned \*)dir)

#define WRITE\_GPIO(dir, value) { (\*(volatile unsigned \*)dir) = (value); }

void IOsetup();

unsigned int getSwitchVal();

void writeValtoLEDs(unsigned int val);

int main ( void )

{

unsigned int switches\_val;

IOsetup();

while (1) {

switches\_val = getSwitchVal();

writeValtoLEDs(switches\_val);

}

return(0);

}

void IOsetup()

{

int En\_Value=0xFFFF;

WRITE\_GPIO(GPIO\_INOUT, En\_Value);

}

unsigned int getSwitchVal()

{

unsigned int val;

val = READ\_GPIO(GPIO\_SWs); // read value on switches

val = val >> 16; // shift into lower 16 bits

return val;

}

void writeValtoLEDs(unsigned int val)

{

WRITE\_GPIO(GPIO\_LEDs, val); // display val on LEDs

}

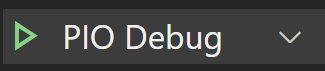
Guarde este ficheiro na pasta src do seu projeto e dê-lhe o nome LedsSwitches\_Functions.c.

Passo 3. Configure o RVfpgaNexys na placa FPGA Nexys A7

Configure o RVfpgaNexys na placa Nexys A7 tal como fez nos Labs 2 and 3.

Passo 4. Compile, descarregue, e execute o programa

Agora está apto a compilar, descarregar, e correr/depurar o programa no RVfpgaNexys.

Depois de carregar nos botões de *Run*  e *Start Debugging* , clique no botão *Step Over*  (localizado na barra de ferramentas superior) ou prima F10 duas vezes até chegar à linha 19 que chama a função getSwitchVal(). Depois, pressione o botão *Step Into*  (ou F11). Isto fará entrar dentro da função getSwitchVal(). Se não estiver já visível, expanda o campo das variáveis VARIABLES → na barra de ferramentas à esquerda para ver a variável val. A variável val poderá estar listada como otimizada/“optimized out” neste ponto do programa. Avance um passo (quer seja Step Over ou Step Into) e observe a mudança do valor da variável val para o valor dos interruptores (switches), tal como mostrado na Figura 1.

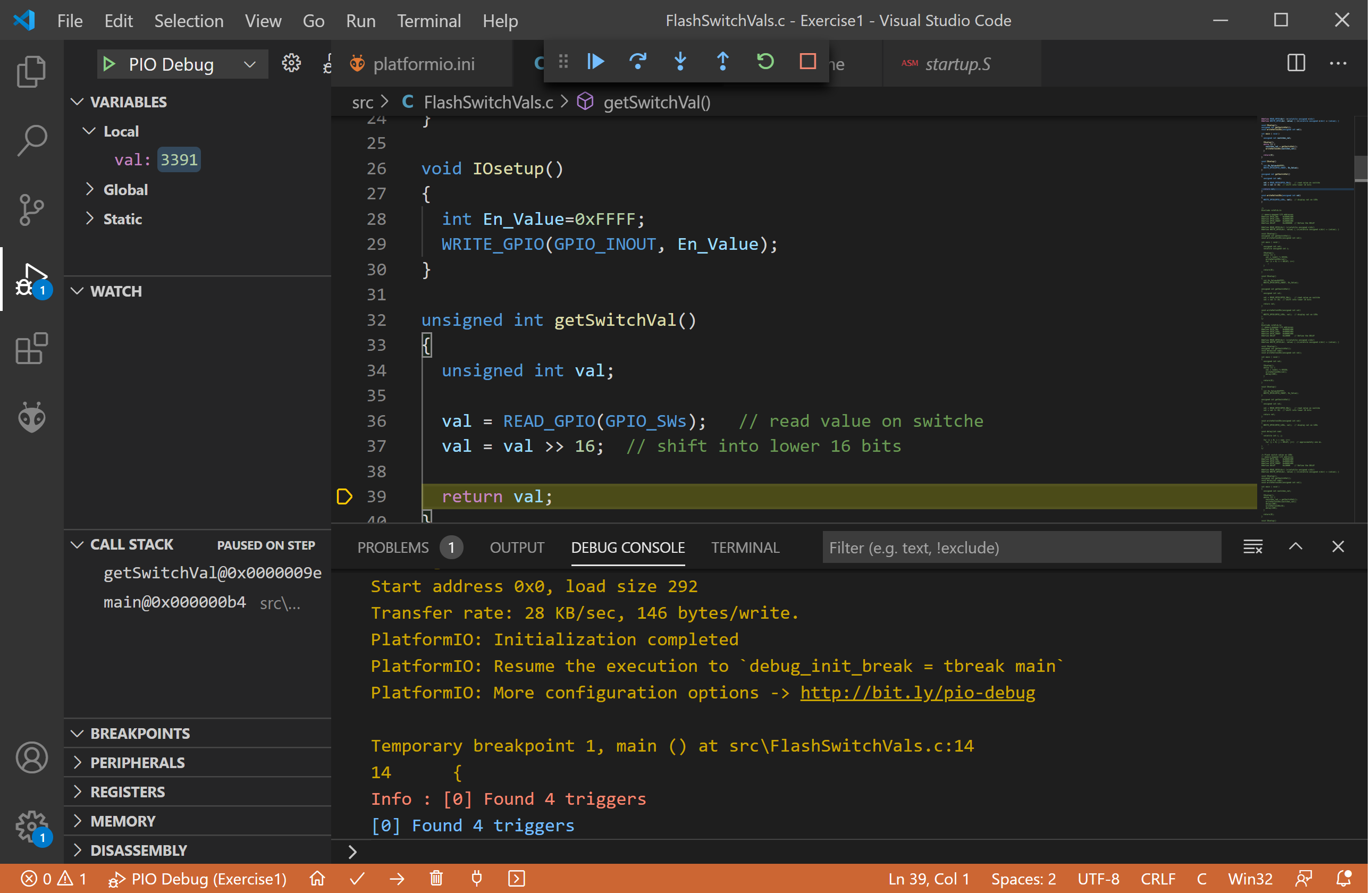


Figura 1. Entrada dentro da função getSwitchVal()

Agora coloque um ponto de paragem (*breakpoint*) na linha 19 clicando à esquerda do número da linha. Irá ver um ponto vermelho aparecer à esquerda indicando que existe um ponto de paragem nessa linha, tal como ilustrado na Figura 2.

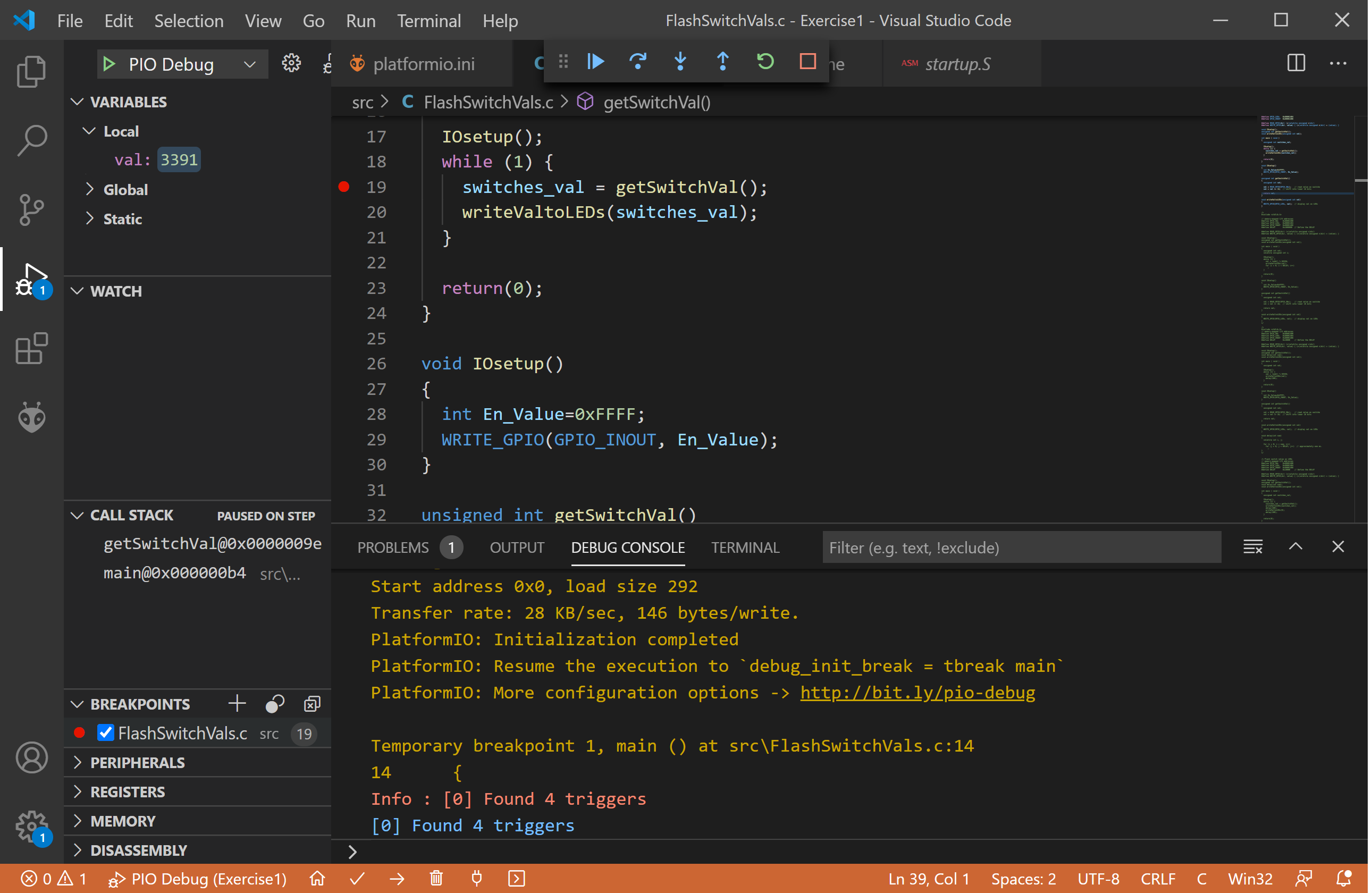


Figura 2. Definição de um ponto de paragem (*breakpoint*)

Pressione o botão *Continue*  (ou F5). O programa irá parar na linha 19 assim que o ponto de paragem seja alcançado. De seguida, pressione o botão *Step Over*  (ou F10). A função irá executar, mas o depurador não irá entrar dentro da função. Apenas os efeitos da função são apresentados. Em particular, a variável switches\_val irá tomar o valor dos interruptores, tal como ilustrado na Figura 3.

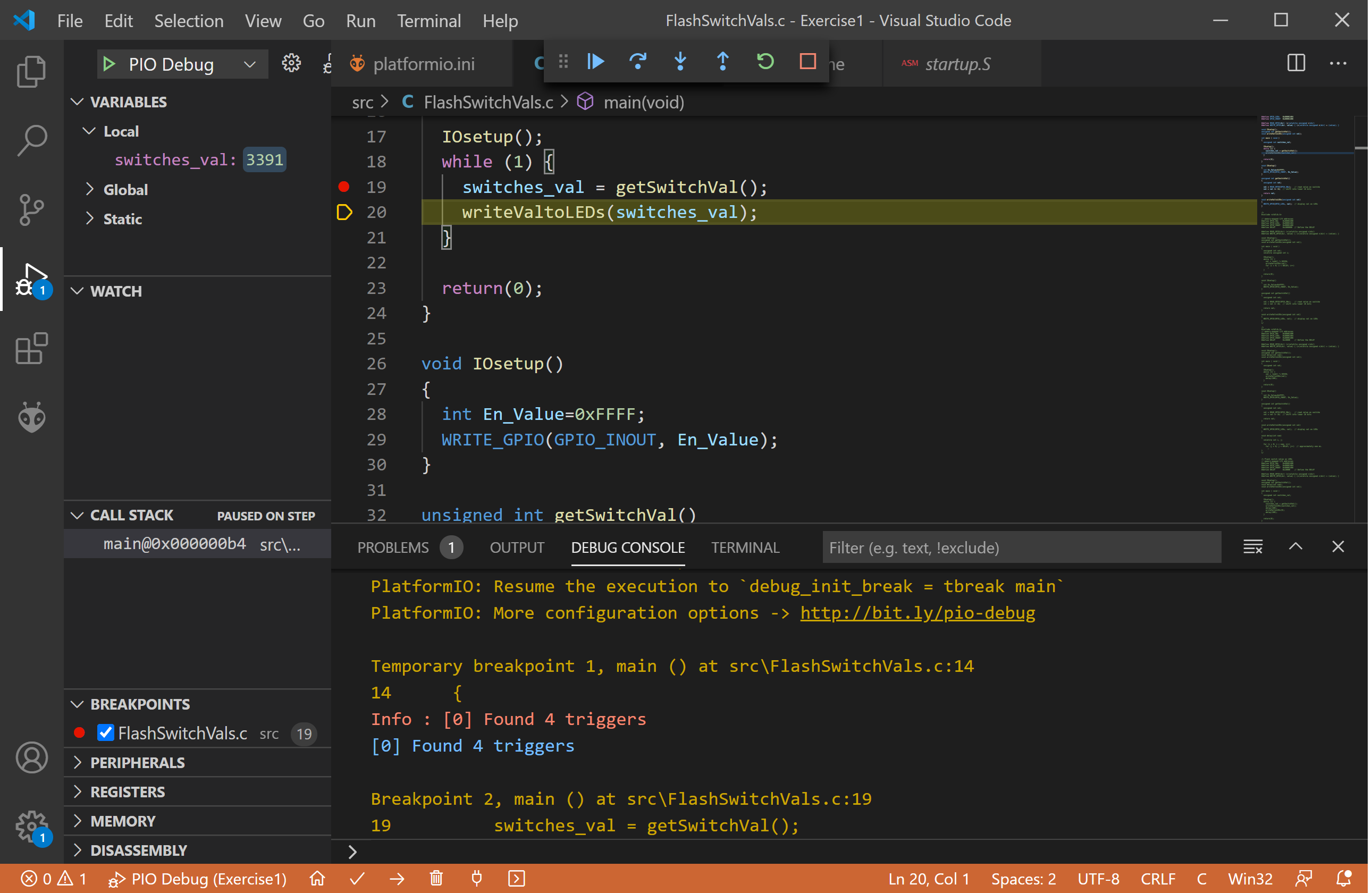


Figura 3. Saltando uma função

# Escrever um programa em C com chamadas a funções de bibliotecas

Linguagens de programação de alto-nível tal como C, incluem bibliotecas de funções que são usadas frequentemente por programadores. Pesquise na Internet por “C standard libraries” para procurar pelo nome das funções habitualmente usadas. Estas bibliotecas de funções podem ser usadas através da inclusão de um ficheiro “cabeçalho” (header file) que contém a declaração das funções. Isto é conseguido escrevendo a seguinte linha no topo do programa em C:

#include <libraryname>

“libraryname” é para ser substituído pelo nome da biblioteca a usar. Por exemplo, a biblioteca de matemática (math.h) oferece funções comuns tais como fabs(), que calcula o valor absoluto de um número em virgula flutuante, fmax(), que devolve o máximo de dois valores em virgula flutuante, etc.

Outra biblioteca comum é a biblioteca C standard (stdlib.h). Uma das funções incluídas nesta biblioteca gera valores aleatórios. Por exemplo, o programa seguinte mostra um valor aleatório nos LEDs incluindo o ficheiro stdlib.h (#include <stdlib.h>) e invocando a função rand() que devolve um número aleatório. Copie o programa em baixo para o projeto no PlatformIO RVfpga e corra-o no RVfpgaNexys na placa FPGA Nexys A7 FPGA.

#include <stdlib.h>

// Endereços de E/S mapeados em memória

#define GPIO\_SWs 0x80001400

#define GPIO\_LEDs 0x80001404

#define GPIO\_INOUT 0x80001408

#define DELAY 0x1000000 // Define the DELAY

#define READ\_GPIO(dir) (\*(volatile unsigned \*)dir)

#define WRITE\_GPIO(dir, value) { (\*(volatile unsigned \*)dir) = (value); }

void IOsetup();

unsigned int getSwitchVal();

void writeValtoLEDs(unsigned int val);

int main(void)

{

unsigned int val;

volatile unsigned int i;

IOsetup();

while (1) {

val = rand() % 65536;

writeValtoLEDs(val);

for (i = 0; i < DELAY; i++)

;

}

return(0);

}

void IOsetup() {

int En\_Value=0xFFFF;

WRITE\_GPIO(GPIO\_INOUT, En\_Value);

}

unsigned int getSwitchVal() {

unsigned int val;

val = READ\_GPIO(GPIO\_SWs); // read value on switches

val = val >> 16; // shift into lower 16 bits

return val;

}

void writeValtoLEDs(unsigned int val) {

WRITE\_GPIO(GPIO\_LEDs, val); // display val on LEDs

}

Este programa encontra-se também disponível no ficheiro:

[RVfpgaPath]/RVfpga/Labs/Lab03/RandomNumberLEDs.c

Para além das bibliotecas padrão em C, a Western Digital (WD) fornece, no Firmware Package (<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>), bibliotecas específicas para o processador SweRV EH1 (PSP, que pode encontrar na pasta ~/.platformio/packages/framework-wd-riscv-sdk/psp/) e para a placa Nexys A7 (BSP, que pode encontrar na pasta ~/.platformio/packages/framework-wd-riscv-sdk/board/nexys\_a7\_eh1/bsp/). Tal como explicado no Guia de Introdução (Secção 6.F – programa *HelloWorld\_C-Lang*), estas bibliotecas são incluídas no projeto adicionando a linha correspondente no platformio.ini e incluindo os ficheiros no início do programa em C.

Estas bibliotecas oferecem funções e macros que permitem os programadores usarem interrupções, mostrar uma cadeia de caracteres, ler/escrever registos individuais, entre outras coisas. No Guida de Introdução ao RVfpga e nestes labs, irá usar várias destas funções nos exemplos e exercícios.

# Convenção de Chamadas no RISC-V

Esta secção descreve a Convenção de Chamadas (*Calling Convention*) no RISC-V, que define como as funções de alto-nível são traduzidas em linguagem Assembly RISC-V. Esta convenção de chamadas faz parte da *Application Binary Interface* (ABI). Ao definir uma convenção, as funções escritas por diferentes programadores ou contidas em bibliotecas podem ser utilizadas entre programas. No RISC-V, a instrução jump and link (jal) invoca uma chamada para uma função. Por exemplo, o seguinte código chama a função func1:

jal func1

Esta instrução salta para a etiqueta func1 *e* guarda o endereço da instrução após jal no registo do endereço de retorno *return address* (ra = x1). A função retorna então usando a pseudo-instrução (ret) ou instrução de registo de salto: jr ra, que salta para o endereço armazenado em ra.

As funções podem ser chamadas com argumentos de entrada e podem também devolver um valor à função de chamada. Pela convenção RISC-V, os argumentos de entrada são passados para a função em registos *a0*–*a7*. Se forem necessários argumentos adicionais, estes são colocados na pilha. Mais uma vez, por convenção, os valores de retorno são colocados nos registos *a0* e *a1*. O acordo sobre os registos são utilizados para passar argumentos e os valores de retorno são definidos pelo Convenção de Chamada RISC-V.

Para invocar com segurança uma função a partir de qualquer local do programa, é essencial que a função preserve o estado arquitetónico da máquina (ou seja, o conteúdo desses registos do que pode ser visto pelo programador). Suponha-se que temos um programa com uma função main que tem um ciclo que utiliza registo t0 para o armazenamento do índice do ciclo. No corpo do ciclo, uma função chamada SortVector é chamada, e esta função SortVector usa o registo t0 para armazenar o endereço do vetor A (ver Figura 4). Então, o registo t0 é substituído na função SortVector, que tem o efeito secundário indesejável de modificar o índice do laço e fazer com que a sua execução seja incorreta.

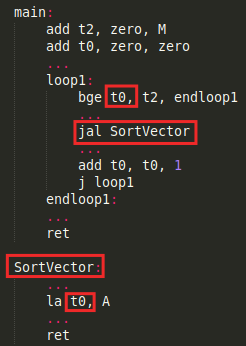


Figura 4. Exemplo do conflito na utilização de um registo entre o programa principal e uma função

Obviamente, isto não teria acontecido se o programador da função main tivesse escolhido outro registo para implementar o índice do ciclo (por exemplo, t1). Contudo, não é razoável (e em alguns casos, nem sequer é possível) forçar o programador a conhecer todos os detalhes internos da implementação de uma função antes de a chamar.

Uma solução mais prática é que cada função crie uma cópia temporária em memória de todos os registos que serão modificados, e restabeleça os seus valores originais antes de regressar à função *chamadora*. Esta solução é implementada por meio do Call Stack, que é uma região de memória que é acedida utilizando uma pilha com política LIFO (Last-In-First-Out). Esta região é utilizada para armazenar toda a informação relacionada com as funções vivas do programa (ou seja, as funções que já começaram, mas não terminaram a sua execução), e que começa no fim da memória disponível (ou seja, nos endereços mais altos), e cresce para endereços mais baixos.

Uma função é normalmente estruturada em três partes:

* Código de entrada (Prólogo)
* Função Corpo
* Código de saída (Epílogo)

O *Prólogo* deve criar a stack frame da função e guardar os registos na pilha, se necessário. A *stack frame* é a região de memória utilizada por uma função durante a sua execução. O *Epilogo* restaura o estado arquitetónico da função chamadora (ou *caller*) e liberta o espaço de memória ocupado pela *stack frame*, deixando assim a pilha exatamente como estava antes de executar o *Prólogo*.

Os acessos à pilha são geridos por meio de um ponteiro, chamado *stack pointer* (sp *=* x2), que armazena o endereço do último local ocupado da pilha. Antes do início de um programa, sp deve ser iniciado com o endereço da base da pilha (ou seja, o endereço mais alto da região da pilha). No Sistema RVfpga, o registo sp é inicializado pela função \_start, que é implementado no ficheiro *~/.platformio/packages/framework-wd-riscv-sdk/board/nexys\_a7\_eh1/startup.S*. Na inicialização, a pilha está vazia. Um segundo ponteiro, o *frame pointer* (fp = x8) aponta para o endereço base (ou seja, o endereço mais alto) da *stack frame* da função ativa.

As funções utilizam o *stack frame* como região de memória privada, que só pode ser acedida a partir da própria função. Uma parte da *stack frame* é dedicada a guardar uma cópia dos registos arquitetónicos que devem ser modificados pela função e, em alguns casos, também pode ser utilizado como forma de passar parâmetros para a função através de locais de memória.

A Tabela 1 descreve o objetivo que a convenção RISC-V atribui a cada registo de números inteiros. Tal como ilustrado na Tabela 1, alguns registos devem ser preservados por uma função chamada, enquanto outros podem ser substituídos pela função (ou seja, não são preservados).

* Se a função precisar de substituir qualquer registo preservado, deve primeiro fazer uma cópia de tal registo no seu *stack frame* e restaurar o valor antes de regressar à *caller* (i.e., a função que a chamou). Para além do stack pointer (sp) e do registo *return address* (ra), doze registos inteiros s0–s11 são preservados em todas as chamadas e devem ser salvos pela função chamadora (*callee*) se forem usados por ele.
* Por outro lado, o *caller* deve estar consciente de que alguns registos não precisam de ser preservados pela função chamadora (*callee*) e, portanto, pode perder-se após a chamada. Note-se que, para além do argumento e dos registos de valor de retorno (a0–*a7*), sete registos inteiros t0–t6 são registos temporários que são voláteis através de chamadas e devem ser guardados pelo *caller* se usado novamente após a invocação da função.

**Tabela 1. Registos de Inteiros RISC-V**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nome** | **Número de Registo** | **Uso** | **Preservado** |
| **zero** | **x0** | Valor constante 0 | - |
| **ra** | **x1** | Return address | Sim |
| **sp** | **x2** | Stack pointer | Sim |
| **gp** | **x3** | Global pointer | - |
| **tp** | **x4** | Thread pointer | - |
| **t0-2** | **x5-7** | Variáveis temporárias | Não |
| **s0/fp** | **x8** | Registo guardado/Frame pointer | Sim |
| **s1** | **x9** | Registo guardado | Sim |
| **a0-1** | **x10-11** | Argumentos de funções/Valores de retorno | Não |
| **a2-7** | **x12-17** | Argumentos de funções | Não |
| **s2-11** | **x18-27** | Registos guardados | Sim |
| **t3-6** | **x28-31** | Variáveis temporárias | Não |

No exemplo da Figura 4, haveria duas soluções de acordo com esta convenção:

* A função main poderia utilizar um registo para o índice do ciclo que é garantido para ser preservado pela função SortVector (tal como s0) em vez de t0.
* A função main poderia continuar a usar t0, mas depois tem de preservar o seu conteúdo na pilha antes de chamar SortVector e restaurá-lo depois de regressar de SortVector.

A pilha expande-se à medida que é necessária mais memória pelas estruturas da pilha de funções e contrai à medida que essas funções se completam. A pilha cresce para baixo (para endereços mais baixos) e o ponteiro da pilha é alinhado a um limite de 16 bytes na entrada do procedimento. Na ABI padrão, o ponteiro da pilha deve permanecer alinhado durante toda a execução do procedimento.

**Exemplo**

O exemplo seguinte implementa um algoritmo de ordenação, primeiro em C (Figura 5) e depois na linguagem Assembly RISC-V (Figura 6). A entrada é um vetor A de N elementos, sendo cada um deles um inteiro maior que 0. A saída é outro vetor, B, que armazena os elementos de A em ordem decrescente.

Em C, a função main chama a função SortVector, que recebe os endereços das matrizes A e B, e o seu tamanho (N), e armazena os elementos de A em B elemento por elemento, em ordem decrescente. Esta função SortVector chama outra função, MaxVector, que recebe o endereço da matriz A e o seu tamanho, e devolve o valor máximo da matriz A e repõe esse valor, para que não seja mais considerado nas iterações seguintes.

|  |
| --- |
| #define N 8  int MaxVector(int A[], int size)  {  int max=0, ind=0, j;  for(j=0; j<size; j++){  if(A[j]>max){  max=A[j];  ind=j;  }  }  A[ind]=0;  return(max);  }  int SortVector(int A[], int B[], int size)  {  int max, j;  for(j=0; j<size; j++){  max=MaxVector(A, size);  B[j]=max;  }  return(0);  }  int main ( void )  {  int A[N]={7,3,25,4,75,2,1,1}, B[N];  SortVector(A, B, N);  return(0);  } |

Figura 5. Algoritmo de ordenação em C

A Figura 6 ilustra o mesmo algoritmo escrito em Assembly. Analisamos o programa tendo em conta os conceitos explicados nas secções anteriores.

* **função main** 
  + Prólogo
    - Primeiro, é reservado espaço na pilha para armazenar os registos conservados que são utilizados na função: add sp, sp, -16. Note-se que, de acordo com a convenção, o registo sp deve ser sempre manter o alinhado a 16 bytes para manter a compatibilidade com a versão de 128 bits do RISC-V, RV128I.
    - Dado que esta função não utiliza nenhum registo guardado, os registos s0-s11 não precisam de ser armazenados na pilha. No entanto, o registo ra deve ser guardado, dado que main chama a função SortVector, que atualiza o valor armazenado em ra.
  + Corpo da Função
    - A função SortVector é invocada usando a instrução jal SortVector. Antes de chamar a função, de acordo com a Convenção de Chamada, os 3 argumentos de entrada são colocados em registos a0 (endereço de A), a1 (endereço de B), e a2 (dimensão dos vetores A e B).
  + Epilogo
    - O registo que foi guardado na pilha no prólogo (ra) é agora restaurado.
    - O stack pointer (sp) é também restaurado à sua posição inicial: add sp, sp, 16.
* **função SortVector** 
  + Prólogo
    - Primeiro, é reservado espaço na pilha para armazenar os registos conservados que são utilizados na função: add sp, sp, -32.
    - Depois, os registos guardados utilizados pela função (s1-s3) são armazenados na pilha, um por um.
    - O registo ra também deve ser salvo, porque SortVector invoca a função MaxVector, que substitui o valor armazenado em ra.
  + Corpo da Função
    - Primeiro, os parâmetros de entrada (a0, a1 e a2) são copiados para registos conservados (s1, s2 and s3), para que possam ser utilizados após a execução da função MaxVector.
    - Para calcular o vetor B, é implementado um ciclo que, em cada iteração, calcula o valor máximo de A e armazena-o em B. Para calcular o valor máximo de A, a função MaxVector é chamda em cada iteração do ciclo: jal MaxVector. Antes de chamar a função, de acordo com a Convenção de Chamada, os parâmetros de entrada para esta função são movidos para registos a0 e a1. Quando a função termina a execução, devolve o valor máximo de A no registo a0.
    - Note-se que o ciclo utiliza principalmente os registos guardados para armazenar variáveis. Estes registos são garantidos pela Convenção de Chamada RISC-V para preservar o seu valor após a execução do MaxVector (i.e. a função deve preservar os seus valores).
    - Os registos a0 e a1 podem ser modificados pela função. Assim, devem ser preparados antes de cada invocação.
    - O registo t1 precisa de ser reutilizado após MaxVector retornos. Assim, deve ser preservado na pilha de SortVector antes de chamar a função (sw t1, 16(sp)) e restaurado após a sua execução (lw t1, 16(sp)).
  + Epílogo
    - Os registos que foram guardados na pilha durante o prólogo, são agora restaurados.
    - O stack pointer (sp) é também restaurado à sua posição inicial: add sp, sp, 32.
* **Função MaxVector** 
  + Prólogo
    - Primeiro, é feito espaço na pilha para armazenar os registos conservados que são utilizados na função: add sp, sp, -16.
    - Depois, o registo guardado utilizado pela função (ou seja, O registo s1) é guardado na pilha: sw s1, 0(sp). Note-se que, se este registo não fosse guardado por esta função, a execução da função chamadora/*caller* (SortVector) falharia, uma vez que também está a utilizar este registo para armazenar o endereço do vetor A.
    - Porque esta função não invoca outra (é uma função folha/*leaf*), ra não precisa de ser guardado neste caso.
  + Corpo da Função
    - A função usa s1 e alguns registos temporários para calcular o valor máximo da matriz A.
  + Epílogo
    - A função deve preparar o valor de retorno antes de regressar à função chamadora/*caller*: mv a0, t2.
    - O registo que foi guardado na pilha durante o prólogo (s1), é agora restaurado.
    - O stack pointer (sp) é também restaurado para a sua posição inicial: add sp, sp, 16.

|  |
| --- |
| .globl main  .equ N, 8  .data  A: .word 7,3,25,4,75,2,1,1  .bss  B: .space 4\*N  .text  MaxVector:  add sp, sp, -16  sw s1, 0(sp)  mv s1, zero  mv t2, zero  loop2:  beq s1, a1, endloop2  lw t1, (a0)  ble t1, t2, else2  mv t2, t1  mv t3, a0  else2:  add a0, a0, 4  add s1, s1, 1  j loop2  endloop2:  sw zero, (t3)  mv a0, t2  lw s1, 0(sp)  add sp, sp, 16  ret  SortVector:  add sp, sp, -32  sw s1, 0(sp)  sw s2, 4(sp)  sw s3, 8(sp)  sw ra, 12(sp)  mv s1, a0 # Address of vetor A  mv s2, a1 # Address of vetor B  mv s3, a2 # Size of vetors A and B  mv t1, zero  loop1:  beq t1, s3, endloop1  mv a0, s1  mv a1, s3  sw t1, 16(sp)  jal MaxVector  lw t1, 16(sp)  sw a0, (s2)  add s2, s2, 4  add t1, t1, 1  j loop1  endloop1:  lw s1, 0(sp)  lw s2, 4(sp)  lw s3, 8(sp)  lw ra, 12(sp)  add sp, sp, 32  ret  main:  add sp, sp, -16  sw ra, 0(sp)  la a0, A  la a1, B  add a2, zero, N  jal SortVector  lw ra, 0(sp)  add sp, sp, 16  ret  .end |

Figura 6. Algoritmo de ordenação em linguagem de Assembly

AFigura 7ilustra o estado da pilha no ponto de execução do corpo da função MaxVector.

* O *stack frame* da função main é mostrado em azul, e inclui o endereço de retorno /returning address (ra) para essa função.
* O *stack frame* da função SortVector é mostrado em verde, e inclui os registos guardados utilizados por esta função (s1-s3), e os registos t1, e *ra*.
* Finalmente, o *stack frame* da função MaxVector, que é o *active stack frame* (o *stack frame* da função que está em execução), é mostrado em amarelo, e inclui o registo guardado utilizado por esta função (s1).



Figura 7. Estado de pilha no corpo da função MaxVector para o programa Assembly da Figura 6.

**TAREFA:** O programa Assembly da Figura 6 é fornecido num projeto Platformio disponível em: *[RVfpgaPath]/RVfpga/Labs/Lab03/SortingAlgorithm\_Functions*. Executar este programa na placa (ou no simulador ISS) usando a opção de depuração passo a passo para analisar o valor armazenado nos vários registos (s, ra, a, etc.) bem como os valores armazenados na pilha, de acordo com a Convenção de Chamada RISC-V.

- O ficheiro *.pio/build/swervolf\_nexys/firmware.dis*, gerado pelo PlatformIO após a compilação do seu programa, pode ser útil para conhecer os endereços de cada instrução do seu programa.

- Pode utilizar a Consola de Memória para analisar a evolução da pilha, bem como o conteúdo das matrizes A e B.

- Neste projeto utilizamos um script *link.lds* modificado no qual o registo *sp* é alinhado a 16-byte. Pode encontrar o script em *[RVfpgaPath]/RVfpga/Labs/Lab03/SortingAlgorithm\_Functions/ld/link.lds*. O alinhamento do registo sp é forçado utilizando o comando ALIGN():

.stack :

{

\_heap\_end = .;

. = . + \_\_stack\_size;

/\* Force 16-B alignment of SP register \*/

**. = ALIGN(16);**

\_sp = .;

} > ram : ram\_load

# Exercícios

Agora crie os seus próprios programas de C/Assembly que incluem chamadas de funções, completando os seguintes exercícios.

Lembre-se que se deixar a placa Nexys A7 ligada ao seu computador e alimentada, não precisa de recarregar o RVfpgaNexys na placa entre diferentes programas. No entanto, se desligar a placa Nexys A7, terá de recarregar o RVfpgaNexys na placa usando o PlatformIO.

Lembre-se também que pode executar estes programas em simulação, usando Verilator ou Whisper.

Exercício 1. Escrever um programa em C que mostra o inverso dos interruptores nos LEDs. Nomear o programa DisplayInverse\_Functions.c.

Por exemplo, se os interruptores estão (em binário): 0101010101010101, então os LEDs devem exibir: 1010101010101010; se os interruptores forem: 1111000011110000, então os LEDs devem exibir: 0000111100001111; e assim por diante. Incluir a função getSwitchesInvert() que devolve o valor invertido dos interruptores. A declaração da função é:

unsigned int getSwitchesInvert();

Exercício 2. Escrever um programa em C que pisca o valor dos interruptores nos LEDs. Nomear o programa FlashSwitchesToLEDs\_Functions.c

O valor deve ser ligado e desligado aproximadamente a cada dois segundos. Incluir uma função chamada delay(num) que provoca um atraso de num milissegundos. Isto pode ser feito empiricamente e não precisa de ser exato. A declaração da função tem o seguinte aspeto:

void delay(int num);

Exercício 3. Escrever um programa em C que meça o tempo de reação. O programa deve cronometrar o tempo que uma pessoa demora a ligar o interruptor mais à direita (SW[0]) depois de todos os LEDs se acenderem. Irá utilizar a função rand() da biblioteca stdlib.h para gerar uma quantidade aleatória de tempo de atraso entre cada vez que o utilizador tenta testar o seu tempo de reação. Nomear o programa ReactionTime.c.

O programa deve funcionar da seguinte forma.

1. O utilizador desliga o interruptor mais à direita (para baixo) para indicar que gostaria de começar.
2. O programa desliga todos os LEDs, depois espera por uma quantidade aleatória de tempo (mas não mais do que cerca de 3 segundos). Pode reutilizar a função delay() do Exercício 2.
3. Depois todos os LEDs se ligam e o programa começa a contar o número de milissegundos até um utilizador ligar o interruptor mais à direita.
4. 4. Quando o utilizador liga o interruptor mais à direita (SW[0]), o número de milissegundos necessários para ligar o interruptor (ON) é apresentado em binário nos LEDs e em decimal na consola de série.
5. O jogo repete-se então pelo utilizador comutando o interruptor mais direito para baixo (desligado).

Exercício 4. Um problema com a função rand() é que utiliza uma sequência previsivelmente aleatória de números. Ou seja, cada vez que se executa o programa, este começa com o mesmo número aleatório e segue a mesma sequência de números aleatórios. Execute o seu programa a partir do Exercício 3 várias vezes para ver que começa com o mesmo número aleatório e segue a mesma sequência de números aleatórios.

No entanto, se utilizar primeiro a função srand(), ela semeará a função rand() com um ponto de partida aleatório. A única questão é que a função srand() deve receber um argumento de entrada, um número natural, que em si mesmo é aleatório. Dê srand() um número aleatório, por exemplo, o número de milissegundos até o utilizador desligar o interruptor para iniciar o jogo.

Reescrever o Exercício 3 para produzir uma sequência verdadeiramente aleatória de vezes antes de os LED acenderem. Utilizar funções sempre que possível. Nomear o programa ReactionTimeTrulyRandom.c.

Exercício 5. Reescrever o exercício 4 para que os LEDs apresentem uma barra crescente de LEDs, proporcional ao tempo de reação. Desta forma, a pessoa visualiza o seu tempo de reação pode mais facilmente dizer se está a ficar mais rápido - sem ter de interpretar a representação binária do número de milissegundos. Pode escolher a gama de tempos de reação correspondente a cada gama de LEDs acesos. Por exemplo, para tempos de reação rápidos, apenas alguns LEDs à direita devem acender-se. Um número crescente de LEDs à esquerda deverá acender-se à medida que os tempos de reação aumentam. Um tempo de reação muito lento acenderia todos os LEDs. Nomear o programa ReactionTimeBar.c.

Exercício 6. Escrever um programa em C que implemente um jogo "Simon says" / “O reizinho manda”. Deve acontecer o seguinte:

1. O programa pisca um padrão nos três LEDs mais à direita e espera que o utilizador prima a sequência correspondente de interruptores usando os três interruptores mais à direita. Os Switches[2:0] correspondem ao LED[2:0], sendo o LED[0] o LED mais direito e os Switches[0] o interruptor mais direito.
2. Os padrões aleatórios devem começar por acender 1 LED, 2 LEDs, e depois 3, etc.
3. O utilizador tenta então repetir a sequência usando os três interruptores mais à direita. O LED correspondente deverá acender-se quando o utilizador comutar os interruptores para cima (e desligar-se quando o utilizador comutar o interruptor para baixo).
4. Se o utilizador introduzir a sequência correta, após uma pausa, o padrão seguinte deve ser apresentado, com mais um LED na sequência.
5. 5. Se o utilizador introduzir a sequência errada, os LEDs ficam acesos e não é reproduzida qualquer sequência nova.
6. O jogo é reiniciado movendo o interruptor mais à esquerda (Switches[15]) para cima (ON) e depois para baixo (OFF).

Utilize funções à sua escolha para modular o programa e facilitar a escrita, a depuração e a compreensão. Lembre-se de utilizar as bibliotecas C padrão, conforme desejado, para escrever o seu programa. Dê um nome ao programa SimonSays.c.

Exercício 7. Dado um vetor, A, de 3\*N elementos, queremos obter um novo vetor, B, de N elementos, para que cada elemento de B seja o valor absoluto da soma um trio de elementos consecutivos de A. Por exemplo:

B[0] = |A[0]+A[1]+A[2]|, B[1] = |A[3]+A[4]+A[5]|, ...

Escreva um programa Assembly RISC-V chamado Triplets.S (o programa deve estar em conformidade com a convenção de chamada RISC-V):

* A função main implementa o cálculo de B, de acordo com o seguinte pseudo-código de alto nível:

#define N 4

int A[3\*N] = {a list of 3\*N values};

int B[N];

int i, j=0;

void main (void)

{

for (i=0; i<N; i++){

B[i] = res\_triplet(A,j);

j=j+3;

}

}

* A função res\_triplet devolve o valor absoluto da soma de 3 elementos consecutivos do vetor V, começando na posição p. É implementado de acordo com a especificação dada pelo seguinte pseudo-código de alto nível:

int res\_triplet(int V[ ], int pos)

{

int i, sum=0;

for (i=0; i<3; i++)

sum = sum + V[pos+i];

sum=abs(sum);

return sum;

}

* A função abs(int x) devolve o valor absoluto do seu argumento de entrada.

Exercício 8. Escreva um programa Assembly RISC-V chamado Filter.S (o programa deve estar em conformidade com a norma de gestão de funções estudada anteriormente). É possível utilizar o seguinte pseudo-código:

#define N 6

int i, j=0, A[N]={48,64,56,80,96,48}, B[N];

for (i=0; i<(N-1); i++){

if( (myFilter(A[i],A[i+1])) == 1){

B[j]=A[i]+ A[i+1] + 2;

j++;

}

}

* Escreva o código Assembly RISC-V equivalente, incluindo quaisquer diretivas necessárias para reservar espaço de memória, e declarar as secções correspondentes (.data, .bss e .text). A função myFilter devolve o valor 1 se o primeiro argumento for um múltiplo de 16 e o segundo for maior do que o primeiro; caso contrário, devolve um 0.
* Escrever o código Assembly da função myFilter.

Exercício 9. Queremos construir um programa de Assembly RISC-V chamado Coprimes.S (o programa deve estar em conformidade com o padrão de gestão de funções estudado anteriormente), de modo que, dada uma lista de pares de inteiros (>0), se descubra quais os pares que são compostos por números coprimos (ou mutuamente primos). Entende-se que dois números são coprimos se o único divisor comum que eles têm for 1.

Assumimos que os dados de entrada estão contidos num vetor, D, da forma:

D=(x0, y0, c0, x1, y1, c1, ... , xN-1, yN-1, cN-1)

Cada trio (xi, yi, ci) é interpretado da seguinte forma: xi e yi representam um par de números, e ci é inicialmente 0. Depois de executar o programa, o valor de cada ci deve ter sido modificado de tal forma que ci = 2, se xi e yi são coprimos; e ci = 1, caso contrário.

Por exemplo:

Para o vetor de entrada: D = (3,5,0, 6,18,0, 15,45,0, 13,10,0, 24,3,0, 24,35,0)

O resultado final deve ser: D = (3,5,2, 6,18,1, 15,45,1, 13,10,2, 24,3,1, 24,35,2)

* Escreva um programa Assembly RISC-V que atravessa o vetor D e gera o resultado de acordo com a especificação dada na caixa esquerda abaixo. O programa chama a função check\_coprime (int D [], int i), cujos argumentos de entrada são o endereço de partida de D e o número do par que queremos verificar (de 0 a M-1). A função verifica se os números do i-ésimo par do vetor D são coprimos e armazena o resultado na localização da memória correspondente.
* Escreva o código para as funções check\_coprime, de acordo com a especificação dada na caixa direita abaixo. Lembre-se que a função gcd(int a, int b) foi implementada no Lab 3 de acordo com o algoritmo euclidiano, e devolve o maior divisor comum (gcd) dos dois argumentos de entrada. Se o gcd é 1, então os números são coprimos.

|  |  |
| --- | --- |
| #define M 6  int D[]= {a list de M\*3 int values}  void main ( ) {  int i;  for (i=0; i<M; i++)  check\_coprime(D,i);  } | void check\_coprime (int A[ ], int pos) {  int res;  res= gcd( A[3\*pos], A[(3\*pos)+1] );  if (res == 1)  A[(3\*pos)+2]=2;  else  A[(3\*pos)+2]=1;  } |