



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 2

Linguagem Assembly RISC-V

1. INTRODUÇÃO

A programação em linguagens de alto-nível de abstração como C, Java, e Python são eficientes para o programador. Estas linguagens de alto-nível são traduzidas em linguagem Assembly, que é um grupo de instruções simples. Por vezes, secções críticas, em termos de desempenho ou de tempo, são escritas em Assembly para garantir um tempo específico ou reduzir o tempo de computação. Este laboratório mostra-lhe como criar um programa em linguagem Assembly RISC-V que pode ser executado no sistema RVfpga utilizando o PlatformIO. Primeiro, damos uma breve visão geral do Assembly do RISC-V e depois mostramos como criar e executar um programa em Assembly no RVfpgaNexys (lembre-se que também pode executar os programas em simulação usando o Verilator ou o Whisper). Depois fornecemos exercícios para que possa praticar a escrita dos seus próprios programas de montagem do RISC-V.

2. Resumo da Linguagem Assembly RISC-V

A linguagem Assembly do RISC-V inclui instruções simples que são utilizadas para implementar um código de alto-nível. Por exemplo, algumas instruções comuns do RISC-V incluem as instruções `add`, `add`, `sub`, e `mul` que adicionam, subtraem ou multiplicam dois operandos.

Os tipos básicos de instruções RISC-V são: instruções computacionais (aritméticas, lógicas, e de deslocamento), operações de memória, e saltos (in)condicionais. As instruções mais comuns do RISC-V encontram-se na Tabela 1. As instruções utilizam operandos que estão em registos ou memória ou que são codificados como uma constante (ou seja, valor imediato). O RISC-V tem 32 registos de 32 bits. A Tabela 2 lista os nomes dos 32 registos RISC-V. Podem ser especificados pelo seu nome (por exemplo, `zero`, `s0`, `t5`, etc.) ou o seu número de registo (ou seja, `x0`, `x8`, `x30`). Os programadores utilizam normalmente nomes de registo que retêm alguma informação sobre a finalidade típica do registo. Por exemplo, os registos guardados, `s0-s11`, são tipicamente utilizados para variáveis de programa, enquanto os registos temporários, `t0-t6` são utilizados para cálculos temporários. O registo `zero` (`x0`) contém sempre o valor 0, uma vez que este é um valor normalmente necessário nos programas. Os outros registos têm também usos específicos, como se mostra na Tabela 2, mas neste laboratório, só é necessário utilizar o registo `zero` e os registos temporários e guardados.

Tabela 1. Instruções comuns do Assembly RISC-V

	Assembly RISC-V	Descrição	Operação
Computacional	<code>add s0, s1, s2</code>	Adição	$s0 = s1 + s2$
	<code>sub s0, s1, s2</code>	Subtração	$s0 = s1 - s2$
	<code>addi t3, t1, -10</code>	Adição imediata	$t3 = t1 - 10$
	<code>mul t0, t2, t3</code>	Multiplicação 32-bits	$t0 = t2 * t3$
	<code>div s9, t5, t6</code>	Divisão	$t9 = t5 / t6$
	<code>rem s4, s1, s2</code>	Resto da divisão	$s4 = s1 \% s2$
	<code>and t0, t1, t2</code>	AND lógico (bit)	$t0 = t1 \& t2$
	<code>or t0, t1, t5</code>	OR lógico (bit)	$t0 = t1 t5$
	<code>xor s3, s4, s5</code>	XOR lógico (bit)	$s3 = s4 \wedge s5$
	<code>andi t1, t2, 0xFFB</code>	AND lógico (bit) imediato	$t1 = t2 \& 0xFFFFFFF$
	<code>ori t0, t1, 0x2C</code>	OR lógico (bit) imediato	$t0 = t1 0x2C$

	xori s3, s4, 0xABC	XOR lógico (bit) imediato	$s3 = s4 \wedge 0\text{FFFFFFABC}$
	sll t0, t1, t2	Deslocamento esquerda lógico	$t0 = t1 \ll t2$
	srl t0, t1, t5	Deslocamento direita lógico	$t0 = t1 \gg t5$
	sra s3, s4, s5	Deslocamento direita aritmético	$s3 = s4 \ggg s5$
	slli t1, t2, 30	Deslocamento esquerda lógico imediato	$t1 = t2 \ll 30$
	srli t0, t1, 5	Deslocamento direita lógico imediato	$t0 = t1 \gg 5$
	srai s3, s4, 31	Deslocamento direita lógico aritmético	$s3 = s4 \ggg 31$
Memória	lw s7, 0x2C(t1)	Carregar palavra	$s7 = \text{memory}[t1+0x2C]$
	lh s5, 0x5A(s3)	Carregar meia-palavra	$s5 = \text{SignExt}(\text{memory}[s3+0x5A]_{15:0})$
	lb s1, -3(t4)	Carregar byte	$s1 = \text{SignExt}(\text{memory}[t4-3]_{7:0})$
	sw t2, 0x7C(t1)	Guardar palavra	$\text{memory}[t1+0x7C] = t2$
	sh t3, 22(s3)	Guardar meia-palavra	$\text{memory}[s3+22]_{15:0} = t3_{15:0}$
	sb t4, 5(s4)	Guardar byte	$\text{memory}[s4+5]_{7:0} = t4_{7:0}$
Salto	beq s1, s2, L1	Salto se igual	if (s1==s2), PC = L1
	bne t3, t4, Loop	Salto se diferente	if (s1!=s2), PC = Loop
	blt t4, t5, L3	Salto se menor	if (t4 < t5), PC = L3
	bge s8, s9, Done	Salto se maior ou igual	if (s8>=s9), PC = Done
Pseudoinstruções	li s1, 0xABCDEF12	Carregar valor imediato	$s1 = 0\text{xABCDEF12}$
	la s1, A	Carregar endereço	s1 = Endereço de memória onde a variável A é armazenada
	nop	Nop	sem operação / operação vazia
	mv s3, s7	Mover	$s3 = s7$
	not t1, t2	Not (Inversão a bit)	$t1 = \sim t2$
	neg s1, s3	Negação	$s1 = -s3$
	j Label	Salto	PC = Endereço de Label
	jal L7	Salto e ligar	PC = L7; ra = PC + 4
	jr s1	Salto para endereço no registo	PC = s1

Para além das instruções RISC-V reais, RISC-V inclui pseudoinstruções (mostradas no fundo da Tabela 1), instruções que não são realmente instruções RISC-V, mas que são normalmente utilizadas pelos programadores. Pseudoinstruções são implementadas usando uma ou mais instruções RISC-V reais. Por exemplo, a pseudo-instrução de cópia (mv s1, s2) copia o conteúdo de s2 para s1. É implementado usando a instrução RISC-V real: addi s1, s2, 0.

Tabela 2. Registos RISC-V

Nome	Número do Registo	Uso
zero	x0	Valor constante 0
ra	x1	Endereço de retorno
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Variáveis temporárias
s0/fp	x8	Registo guardado / Frame pointer
s1	x9	Registo guardado
a0-1	x10-11	Argumentos de função / Valores de retorno

a2-7	x12-17	Argumentos de função
s2-11	x18-27	Registos guardados
t3-6	x28-31	Variáveis temporárias

Os comandos que começam com um ponto são diretivas de Assembler. São comandos para o Assembler e não código a ser traduzido por ele. Dizem ao Assembler onde colocar o código e dados, especificar instruções e constantes de dados para utilização no programa, e assim por diante. Tabela 3 mostra as principais diretivas Assembly do RISC-V (*The RISC-V Reader: An Open Architecture Atlas, Patterson & Waterman, © 2017*).

Tabela 3. Directivas Principais RISC-V

Directiva	Descrição
.text	Os itens subsequentes são armazenados na secção <code>text</code> (código da máquina).
.data	Os itens subsequentes são armazenados na secção <code>data</code> (variáveis globais).
.bss	Os itens subsequentes são armazenados na secção <code>bss</code> (variáveis globais iniciadas a 0).
.section .foo	Os itens subsequentes são armazenados na secção <code>.foo</code> .
.align n	Alinha os próximos dados com o limite 2^n -byte. Por exemplo, <code>.align 2</code> alinha o próximo valor com o limite da palavra.
.balign n	Alinha os próximos dados com o limite n -bytes. Por exemplo, <code>.balign 4</code> alinha o próximo valor com o limite da palavra.
.globl sym	Declara que a Label <code>sym</code> é global e pode ser referenciada a partir de outros ficheiros.
.string "str"	Guardar a String <code>str</code> em memória e termina-a com valor nulo (zero).
.word w1,...,wn	Guarda n valores de 32-bits em palavras de memória consecutivas.
.byte b1,...,bn	Guarda n valores de 8-bits em bytes consecutivos de memória.
.space	Reserva espaço de memória para armazenar variáveis sem um valor inicial. É normalmente utilizado para declarar as variáveis de saída, quando não estão também a servir como variáveis de entrada. O espaço que queremos reservar deve ser sempre expresso como um número de bytes. Por exemplo, a diretiva <code>RES: .space 4</code> reserva quatro bytes (i.e. uma palavra) que não são inicializados.
.equ name, constant	Definir o símbolo <code>name</code> com o valor <code>constant</code> . Por exemplo, <code>.equ N, 12</code> , define o símbolo <code>N</code> com o valor 12.
.end	O montador concluirá o seu trabalho quando chegar à diretiva <code>.end</code> . Qualquer texto localizado após esta diretiva será ignorado.

Os exemplos abaixo (ver Tabela 4 - Tabela 5) mostram como codificar algumas instruções de alto-nível em Assembly RISC-V. Repare que as instruções de salto (`beq`, `bne`, `blt`, e `bge`) saltam para uma Label condicionalmente; ao passo que a instrução `jump` (`j`) salta para uma Label incondicionalmente. Comentários de uma linha são indicados com `//` em C e `#` no Assembly RISC-V.

No primeiro exemplo (implementando uma instrução `if/else`, ver Tabela 4), repare que o código C e Assembly RISC-V verificam as condições opostas: o código C code verifica a condição menor que (`<`) e o equivalente em Assembly verifica a condição maior ou igual que (`>=`).

Tabela 4. Exemplo Assembly RISC-V 1: instrução `if/else`

// Código C	# Assembly RISC-V
<code>int a, b, c;</code>	<code># s0 = a, s1 = b, s2 = c</code>
<code>if (a < b)</code>	<code>bge s0, s1, L1 # if (a >= b) goto L1</code>
<code> c = 5;</code>	<code>addi s2, zero, 5 # c = 5</code>
<code>else</code>	<code>j L2 # jump over else block</code>
<code> c = a + b;</code>	<code>L1: add s2, s0, s1 # c = a + b</code>
	<code>L2:</code>

No segundo exemplo (manipulando um vector de inteiros, ver Tabela 5), o código Assembly RISC-V usa registos temporários (`t0-t3`) para guardar valores temporários, tais como a constante 100 e o endereço base do vector. Após a inicialização dos registos nas primeiras 3 instruções, o código Assembly RISC-V verifica se `i >= 100` usando a instrução `bge` (saltar se maior ou igual que); mais uma vez, isto é o caso oposto do código C. Se a condição for verificada, o ciclo `for` é executado. Se o salto **não** for realizado, `i` é menor do que 100 e o resto do código é executado. Repare que o índice `i` é multiplicado por 4 (usando a instrução `slli t2, s0, 2`) antes de ser adicionado ao valor do endereço base, isto porque os inteiros (números de 32-bit codificados em complemento a 2) ocupam 4 bytes em memória. No RISC-V, a memória é endereçada ao byte (isto é, cada byte tem o seu endereço). Se o vector fosse de caracteres (isto é, `char data[100];`), então cada elemento do vector ocuparia um byte e `i` podia ser adicionado diretamente ao valor do endereço base para gerar o endereço do valor indexado no vector `i`, isto é., `array[i]`. Depois do elemento do vector ser lido, decrementado por 10, e escrito (via instruções `lw`, `addi`, e `sw`, respectivamente), o índice do vector `i` (isto é, `s0`) é incrementado e o programa salta de volta ao início do ciclo `for` (usando a instrução `j L5`).

Tabela 5. Exemplo Assembly RISC-V 2: manipulando um conjunto de inteiros

// Código C	# RISC-V Assembly
<code>int i;</code>	<code># s0 = i, t1 = endereço base de dados (assumido</code>
<code>int data[100];</code>	<code># estar em 0x300)</code>
	<code>addi s0, zero, 0 # i = 0</code>
	<code>addi t0, zero, 100 # t0 = 100</code>
	<code>li t1, 0x300 # endereço base do vector</code>
<code>for (i=0; i<100; i++)</code>	<code>L5: bge s0, t0, L7 # if (i>=100) sair ciclo</code>
	<code>slli t2, s0, 2 # t2 = i*4</code>
	<code>add t2, t1, t2 # endereço de data[i]</code>
	<code>lw t3, 0(t2) # t3 = array[i]</code>
	<code>addi t3, t3, -10 # t3 = array[i]-10</code>
<code> array[i] = array[i]-10;</code>	<code>sw t3, 0(t2) # array[i] = array[i]-10</code>

	<pre> addi s0, s0, 1 # i++ j L5 # loop L7: </pre>
--	---

Para mais detalhes sobre a linguagem Assembly RISC-V, refere-se ao Manual do Conjunto de Instruções (Instruction Set) RISC-V (disponível aqui: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>) ou a um livro tal como o *Digital Design and Computer Architecture*, Harris & Harris, Elsevier, © 2021 ou *The RISC-V Reader: An Open Architecture Atlas*, Patterson & Waterman, © 2017.

3. Escrita de um programa Assembly RISC-V para RVfpga

Agora está apto a explorar e praticar a escrita de programas Assembly RISC-V autonomamente. Antes de escrever os seus próprios programas, siga os passos de preparação do projeto PlatformIO e criar e correr um programa Assembly no RVfpgaNexys (lembrar que também pode correr estes programas em simulação, usando o Verilator ou o Whisper):

1. Criar um projecto RVfpga
2. Escrever um programa em linguagem Assembly RISC-V
3. Configurar o RVfpgaNexys na placa Nexys A7 FPGA
4. Compilar, descarregar, e correr o programa Assembly

Passo 1. Criar um projeto RVfpga

Siga o passo 1 do Lab 2 RVfpga – aqui repetido para sua conveniência. Abra o VSCode clicando no botão Iniciar e escrevendo VSCode e depois clicar em Virtual Studio Code (ver Figura 1).

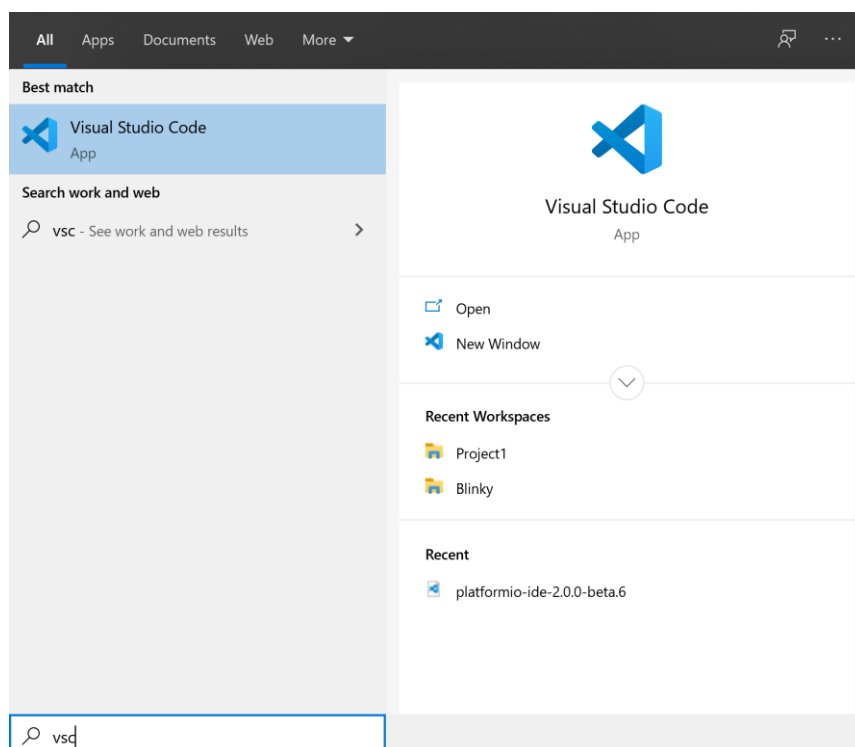


Figura 1. Abrir VSCode

Se o PlatformIO não abrir automaticamente aquando do arranque do VSCode, clique no ícone PlatformIO na faixa lateral e depois clique em PIO Home → Open (ver Figura 2).

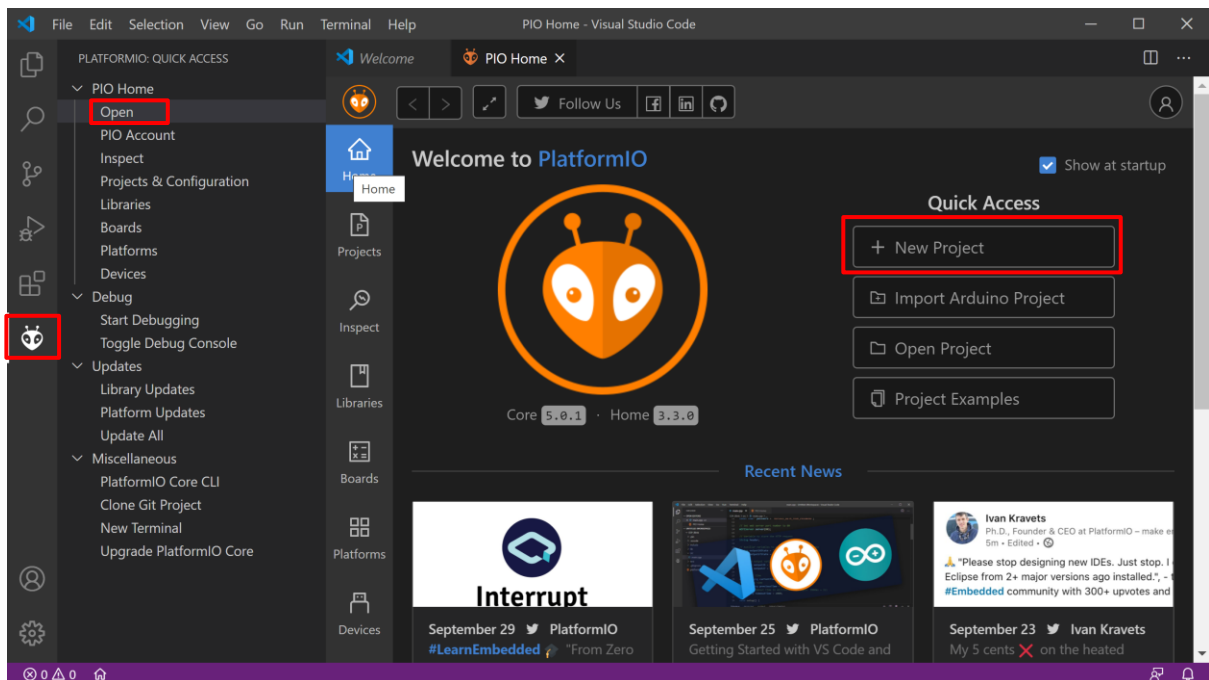


Figura 2. Abrir o PlatformIO e criar novo projeto

Agora, na janela de boas-vindas do PIO Home, clique em New Project (ver Figura 2).

Tal como mostrado na Figura 3, nomeie o projeto como Project1 e escolha a placa (Board) RVfpga: Diligent Nexys A7 (comece a escrever RVfpga e o nome da placa surgirá). Deixe o nome das ferramentas (Framework) como WD-framework (Western Digital framework – que inclui o Freedom-E SDK com o gcc e o gdb). Desmarque o uso da localização por omissão e guarde o programa em:

```
[RVfpgaPath]/RVfpga/Labs/Lab2
```

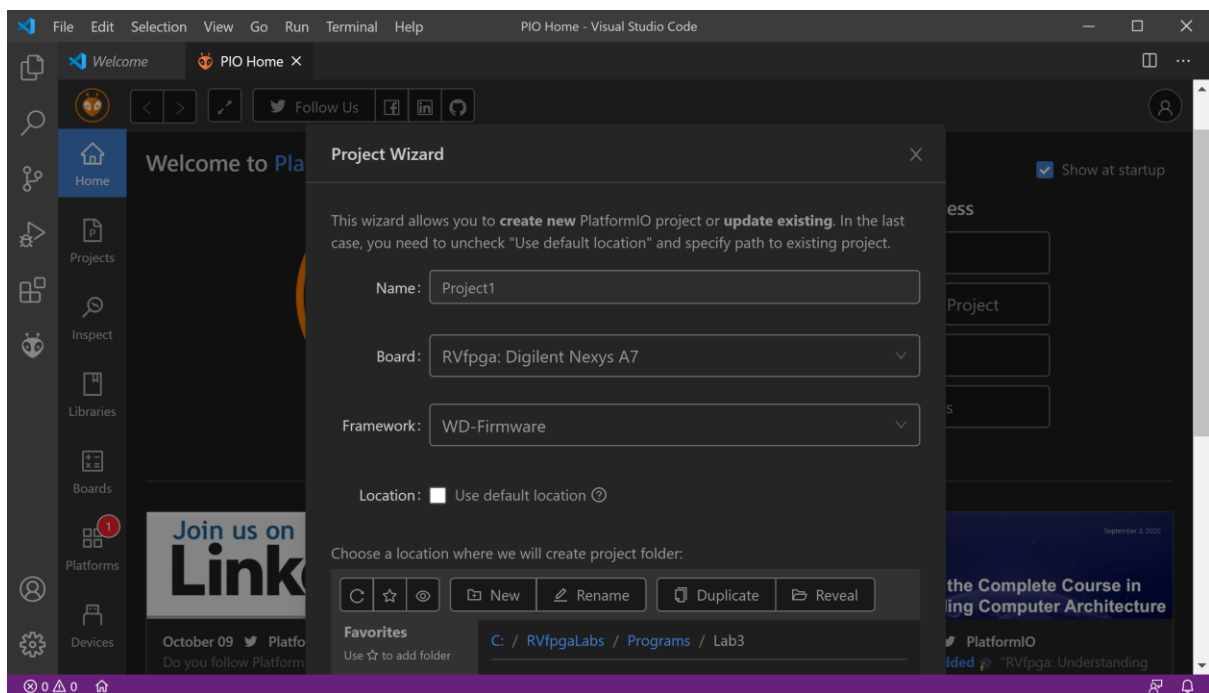


Figura 3. Nomear projeto e selecionar placa e pasta de projeto

Depois clique em Finish no fundo da janela (ver Figura 4).

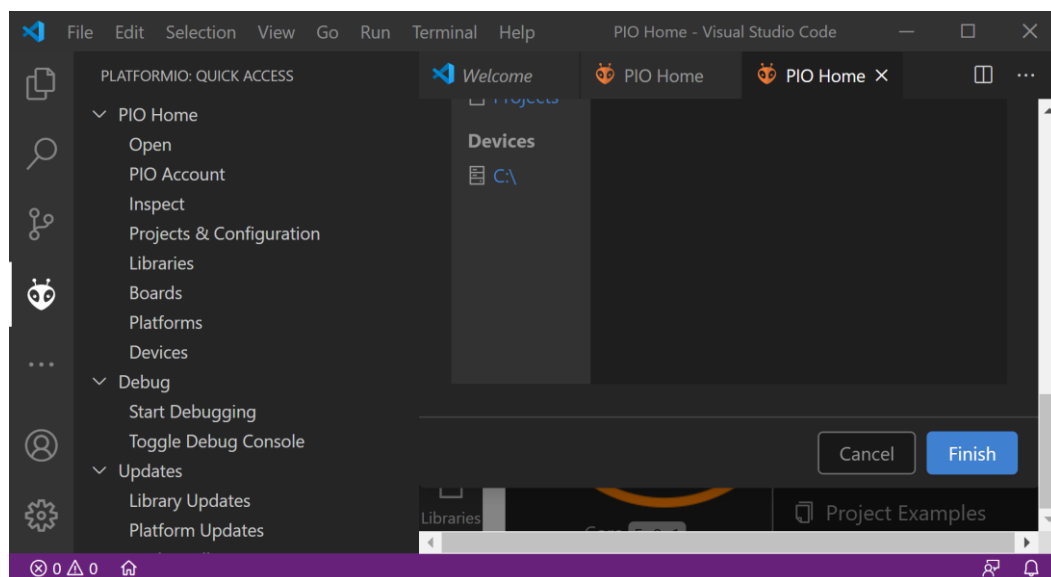


Figura 4. Acabar de criar o projeto

No painel Explorer à esquerda, debaixo de PROJECT1 (poderá ter de o expandir), faça duplo-clique em platformio.ini para o abrir (ver Figura 5). Este é o ficheiro de inicialização do PlatformIO.

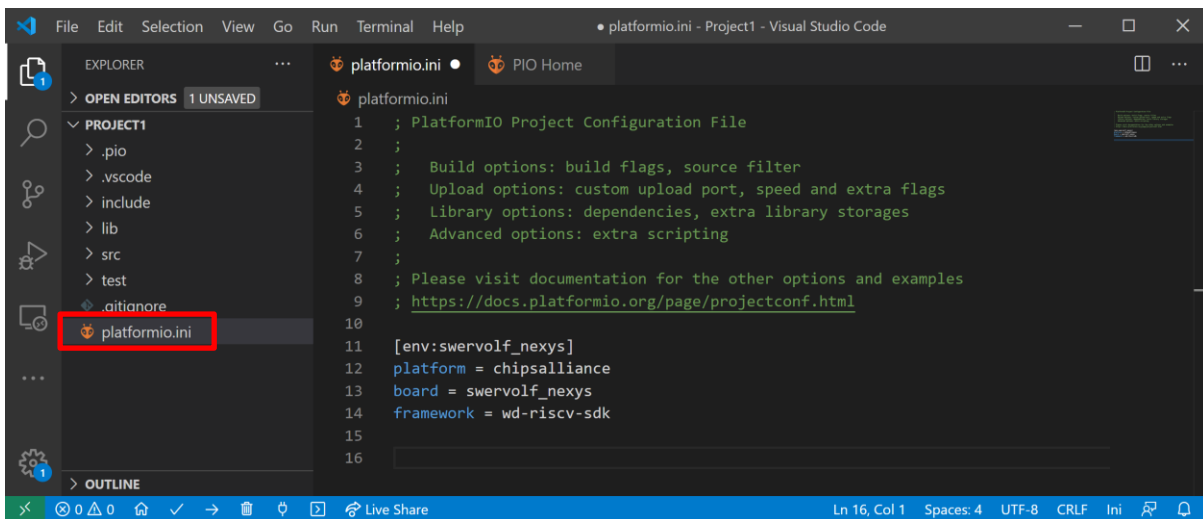


Figura 5. Ficheiro de inicialização do PlatformIO: platformio.ini

Adicione a seguinte linha ao ficheiro platformio.ini, tal como ilustrado na Figura 6:

```
board_build.bitstream_file =
[RVfpgaPath]/RVfpga/Labs/Lab1/Project1/Project1.runs/impl_1/rvfpganexys.bit
```

Esta linha indica onde é que o PlatformIO deverá encontrar o ficheiro binário (bitstream) para configurar a FPGA. O caminho acima é a localização da configuração (bitstream) criado no Lab 1. (Caso não tenha completado o Lab 1, pode usar a configuração RVfpgaNexys distribuída com o Guia de Iniciação em: *[RVfpgaPath]/RVfpga/src/rvfpganexys.bit*.) Faça Ctrl-s para guardar o ficheiro platformio.ini.

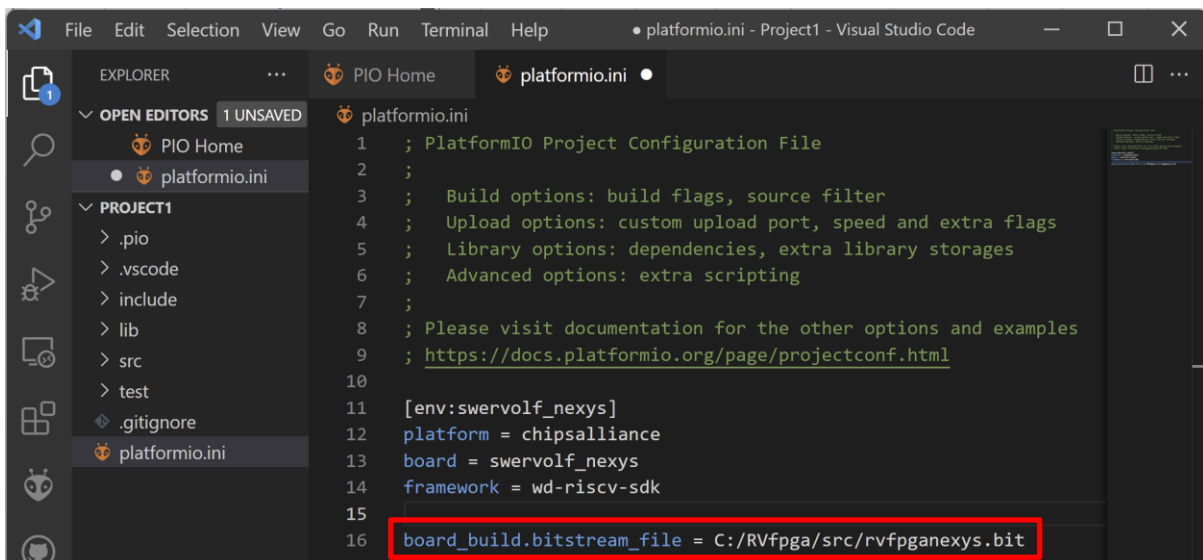


Figura 6. Adicionar localização do ficheiro bitstream do RVfpgaNexys (rvfpganexys.bit)

Lembre-se que um ficheiro *platformio.ini* mais completo foi usado nos exemplos do Guia de Iniciação. Caso queira usar alguma funcionalidade que necessite de comandos extra (tal como o caminho para o simulador Verilator, a configuração da consola de porta série, o depurador whisper, etc.), pode usar o ficheiro *platformio.ini* desses exemplos.

Passo 2. Escrever um programa de linguagem Assembly RISC-V

Agora irá escrever um programa em Assembly RISC-V. Clique em File → New File (ver Figura 7).

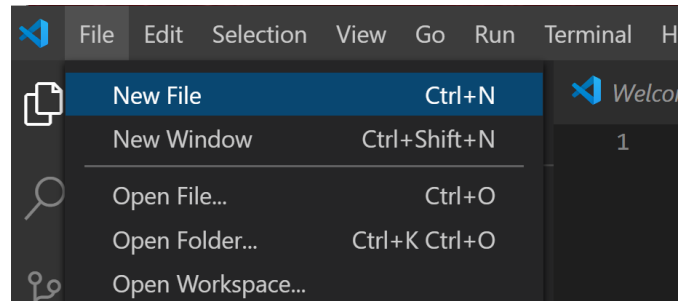


Figura 7. Adicionar um ficheiro ao projeto

Uma janela vazia irá surgir. Escreva (ou faça copy/paste) do seguinte programa Assembly RISC-V nessa janela (ver Figura 8). Este programa também está disponível em:

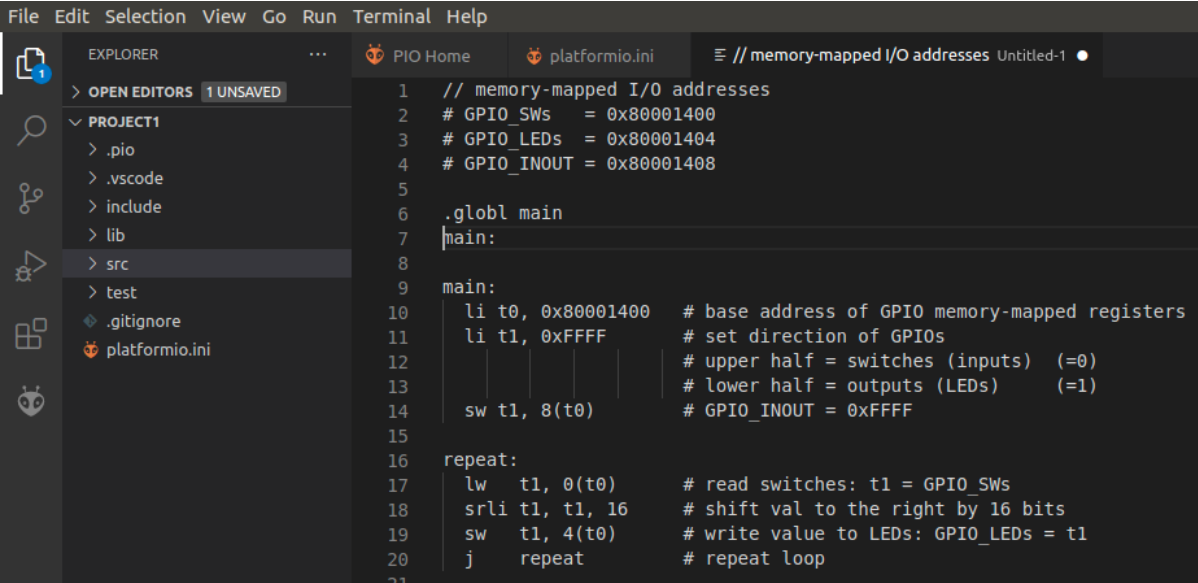
```
[RVfpgaPath]/RVfpga/Labs/Lab2/ReadSwitches.S

// Endereços E/S mapeados em memoria
# GPIO_SWs   = 0x80001400
# GPIO_LEDs  = 0x80001404
# GPIO_INOUT = 0x80001408

.globl main
main:

main:
    li t0, 0x80001400 # base address of GPIO memory-mapped registers
    li t1, 0xFFFF    # set direction of GPIOs
                    # upper half = switches (inputs) (=0)
                    # lower half = outputs (LEDs)   (=1)
    sw t1, 8(t0)      # GPIO_INOUT = 0xFFFF

repeat:
    lw t1, 0(t0)      # read switches: t1 = GPIO_SWs
    srli t1, t1, 16    # shift val to the right by 16 bits
    sw t1, 4(t0)      # write value to LEDs: GPIO_LEDs = t1
    j repeat          # repeat loop
```



```

1 // memory-mapped I/O addresses
2 # GPIO_SWS = 0x80001400
3 # GPIO_LEDS = 0x80001404
4 # GPIO_INOUT = 0x80001408
5
6 .globl main
7 main:
8
9 main:
10     li t0, 0x80001400 # base address of GPIO memory-mapped registers
11     li t1, 0xFFFF    # set direction of GPIOs
12                     # upper half = switches (inputs) (=0)
13                     # lower half = outputs (LEDs) (=1)
14     sw t1, 8(t0)      # GPIO_INOUT = 0xFFFF
15
16 repeat:
17     lw t1, 0(t0)      # read switches: t1 = GPIO_SWS
18     srli t1, t1, 16    # shift val to the right by 16 bits
19     sw t1, 4(t0)      # write value to LEDs: GPIO_LEDS = t1
20     j repeat          # repeat loop
21

```

Figura 8. Escrever o programa Assembly RISC-V

O código Assembly tem de ter as seguintes linhas no seu início:

```
.globl main
main:
```

A diretiva Assembly `.globl` faz com que a Label seja visível em todos os ficheiros ligados (linked). O código de arranque (`~/platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/startup.S`) irá configurar o sistema e saltar para a Label (`main`). O depurador irá definir um ponto de paragem (breakpoint) no mesmo local quando arrancar.

O programa Assembly RISC-V é o mesmo programa do Lab 2, mas desta vez escrito em Assembly RISC-V. Ele define a direção das entradas e saídas das E/S genéricas (GPIO) e repetidamente lê o valor dos interruptores (Switches) e escreve o seu valor nos LEDs.

Depois de introduzir o programa na janela de texto guarde o programa via Ctrl-s. Dê o nome `ReadSwitches.S` e guarde-o na pasta `src` de `Project1` (ver Figura 9).

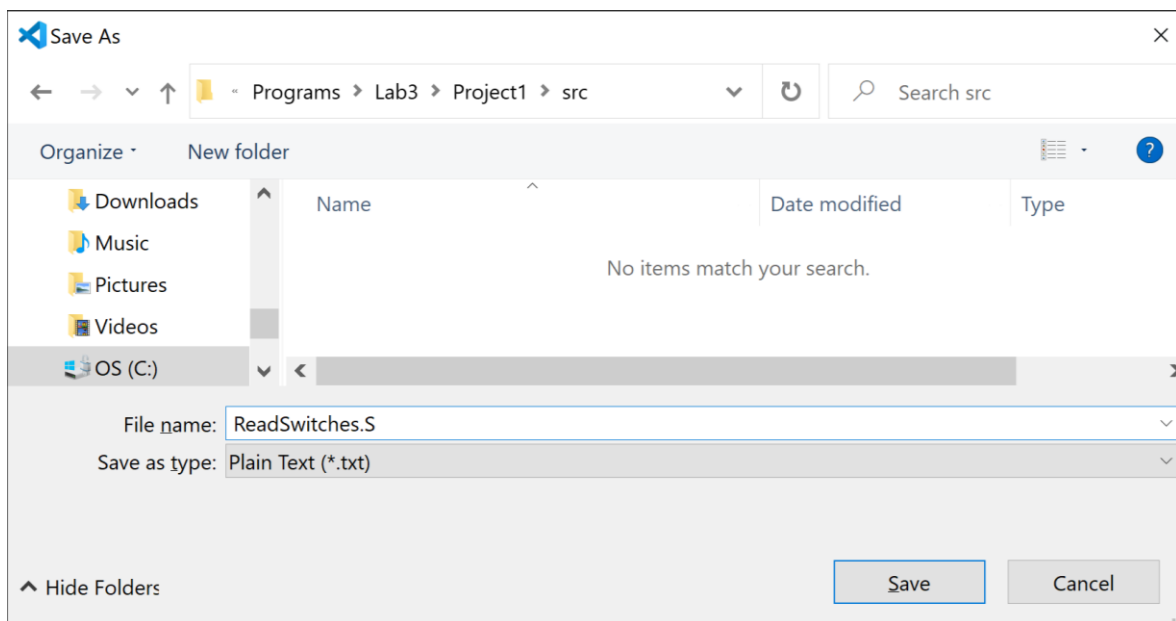




Figura 9. Guardar ficheiro como ReadSwitches.S

Passo 3. Descarregar o RVfpgaNexys para placa FPGA Nexys A7

Irá agora configurar o RVfpgaNexys na placa FPGA Nexys A7 FPGA. Siga as instruções descritas no Guia de Iniciação (GSG) e no Lab 2 – e repetido aqui para a sua conveniência.

Configure o RVfpgaNexys na placa Nexys A7 clicando no ícone na faixa da esquerda do ícone do PlatformIO , de seguida expanda *Project Tasks* → *env:swervolf_nexys* → *Platform* e clique em *Upload Bitstream*.

Como alternativa pode configurar o RVfpgaNexys usando a janela do terminal do PlatformIO clicando no PlatformIO em: botão *New Terminal*  no fundo do menu da janela do PlatformIO, e depois escreva (ou copie) o seguinte texto no terminal do PlatformIO:

```
pio run -t program_fpga
```

Passo 4. Compilar, descarregar, e executar o programa Assembly RISC-V

Agora que o RVfpgaNexys está a correr na placa, irá compilar o seu programa, descarregá-lo para o RVfpgaNexys, e corrê-lo/depurá-lo. Se o VSCode não está em execução, abra-o. O seu último projeto, Project1, deverá abrir automaticamente. Caso contrário, garanta que a extensão do PlatformIO está aberta e clique em *File* → *Open Folder* e selecione (mas não abra) Project1, que criou anteriormente neste lab.

Clique no botão *Run* na barra do menu da esquerda e depois clique no botão *Start Debugging* (ver Figura 10).

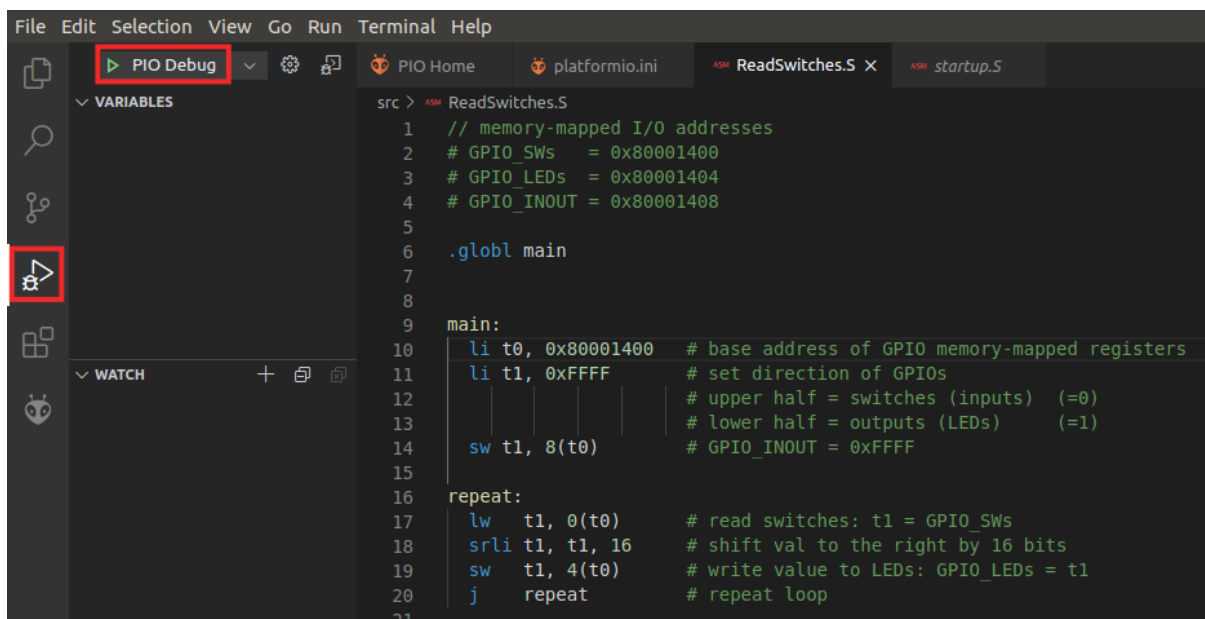


Figura 10. Executar o programa no RVfpgaNexys

O programa será descarregado para a RVfpgaNexys, que está a correr na FPGA na placa Nexys A7. Agora pode começar a correr e depurar o programa (ver Figura 11).

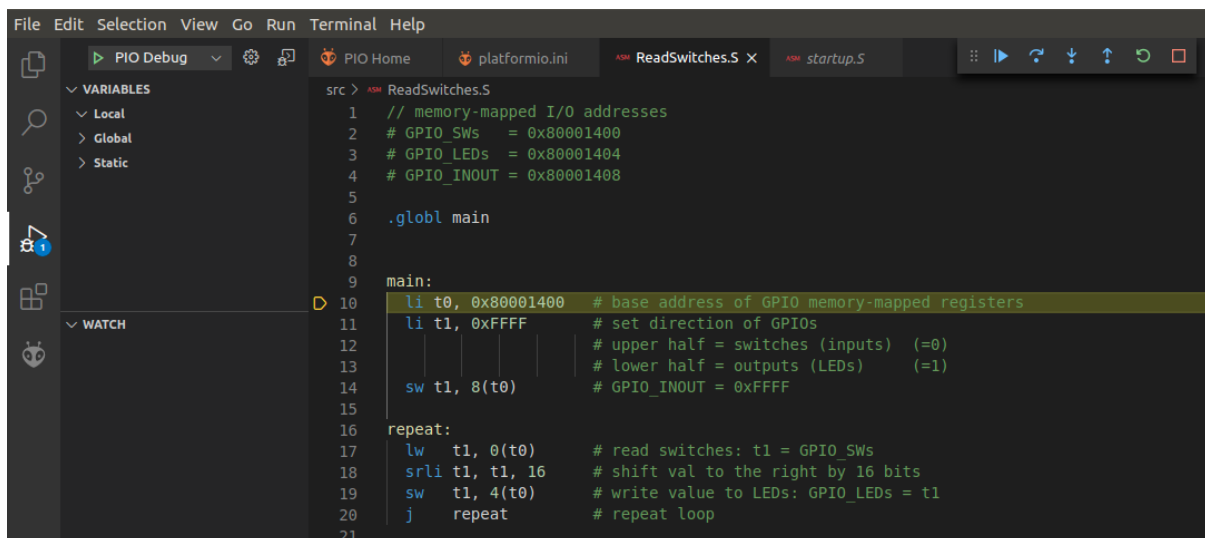


Figura 11. Programa em execução no RVfpgaNexys

Tal como descrito no Guia de Introdução ao RVfpga e no Lab 2, use a barra de ferramentas de depuração e opções de depuração para correr e controlar o programa. Por exemplo, pode definir um breakpoint na linha 17 (clicando imediatamente à esquerda do número da linha) e observe o valor do registo `t1` como o valor lido dos interruptores é escrito nele.

Quando para a sessão de depuração, pressionando o botão Stop



(ou Shift - F5), a sessão de depuração termina, mas o programa continua a correr no RVfpgaNexys.

4. Exercícios

Agora crie os seus programas Assembly RISC-V completando os mesmos exercícios que no Lab 2, mas desta vez com Assembly RISC-V em vez de C. As descrições dos exercícios encontram-se repetidas em baixo para a sua conveniência.

Lembre-se que se deixar a placa Nexys A7 ligada no seu computador e alimentada, não necessita de reconfigurar o RVfpgaNexys para correr programas diferentes. Contudo, se desligar a placa Nexys A7, terá de a reconfigurar com o RVfpgaNexys usando o PlatformIO.

Lembre-se também que pode correr estes programas via simulação, usando o Verilator ou o Whisper.

Exercício 1. Escreva um programa Assembly RISC-V que faça piscar o valor dos interruptores (switches) nos LEDs. O valor deverá acender e apagar de tal forma que se perceba que está a piscar. Nomeie este programa **FlashSwitchesToLEDs.S**.

Exercício 2. Escreva um programa Assembly RISC-V que mostre o valor contrário (inverso) que o valor dos interruptores (switches) nos LEDs. Por exemplo, se o valor dos interruptores for (em binary): 01010101010101, então os LEDs deverão mostrar: 10101010101010; se os interruptores tiverem: 1111000011110000, então os LEDs deverão mostrar: 0000111100001111; e por aí adiante. Nomeie este programa **DisplayInverse.S**.

Exercício 3. Escreva um programa Assembly RISC-V desloque um número crescente de LEDs acessos para trás e para a frente até que todos os LEDs estejam acesos. Depois o padrão deverá ser repetido. Nomeie este programa **ScrollLEDs.S**.

O programa originar o seguinte:

1. Primeiro, um LED deverá acender e deslocar da direita para a esquerda.
2. Quando chega ao LED na posição mais à esquerda, dois LEDs deverão acender e deslocarem-se da esquerda para a direita e depois da direita para a esquerda.
3. Quando os dois LEDs chegam à posição do LED mais à esquerda, três LEDs deverão acender e deslocarem-se da esquerda para a direita e depois da direita para a esquerda.
4. Depois quatro LEDs deverão deslocarem-se.
5. E por aí adiante, até que todos os LEDs estejam acesos.
6. Depois disso o padrão deverá repetir-se.

Exercício 4. Escreva um programa Assembly RISC-V que mostra o resultado da soma do valor inteiro positivo a 4 bits, dos 4 bits menos significativos dos interruptores com os 4 bits mais significativos dos interruptores. Mostre o resultado nos 4 bits menos significativos (mais à direita) dos LEDs. Nomeie o programa **4bitAdd.S**. O quinto bit dos LEDs deverá acender quando ocorrer excesso na soma (ou seja quando o carry out é 1).

Exercício 5. Escreva um programa Assembly RISC-V que calcula o máximo divisor comum entre dois números, a e b , de acordo com o algoritmo Euclideano. Os valores de a e b devem ser definidos como variáveis constantes no programa. Nomeie este programa **GCD.S**. Informação adicional sobre o algoritmo Euclideano:

<https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm>. Também pode pesquisar na Internet por “Euclidean algorithm”.

Exercício 6. Escreva um programa Assembly RISC-V que calcula os primeiros 12 números da série de Fibonacci, e guarda o resultado num vector (ou array), V , com 12 elementos. Os números da série de Fibonacci são calculados pela expressão:

$$V(0)=0, \quad V(1)=1, \quad V(i)=V(i-1)+V(i-2) \quad (\text{onde } i=0,1,2,\dots)$$

Por outras palavras, um determinado elemento i da série de Fibonacci é obtido através da soma dos dois valores anteriores da série. A Tabela 6 mostra os números da série de Fibonacci para $i = 0$ até 8.

Tabela 6. Série de Fibonacci

i	0	1	2	3	4	5	6	7	8
V	0	1	1	2	3	5	8	13	21

A dimensão do vector, N , deve ser definida no programa como uma constante. Nomeie o programa **Fibonacci.S**.

Exercício 7. Dado um vector com N -elementos, A , crie outro vector, B , tal que B apenas contenha os elementos de A que são pares e maiores que 0. Por exemplo: supondo $N = 12$ e $A = [0,1,2,7,-8,4,5,12,11,-2,6,3]$, então B deverá ser: $B = [2,4,12,6]$. Nomeie este programa **EvenPositiveNumbers.S**.

Exercício 8. Dados dois vectores com N -element, A e B , crie outro vector, C , que é definido como:

$$C(i) = |A[i] + B[N-i-1]|, \quad i = 0, \dots, N-1.$$

Escreva um programa Assembly RISC-V que calcula o novo vector. Use vectores de 12 elementos no seu programa. Nomeie o programa **AddVectors.S**.

Exercício 9. Implemente o algoritmo de ordenação *bubble sort* em Assembly RISC-V. Este algoritmo ordena os elementos de um vector por ordem crescente através das seguintes acções:

1. Itere sobre o vector até terminar.
2. Torque qualquer par de valores adjacentes caso: $V(i) > V(i+1)$.
3. O algoritmo pára quando todos os pares de valores consecutivos estiverem ordenados.

Use vectores com 12 elementos para testar o seu programa. Nomeie o programa **BubbleSort.S**.

Exercício 10. Escreva um programa Assembly RISC-V que calcula o fatorial de um dado número positivo, n , através de multiplicações sucessivas. Deverá testar o seu programa para diversos valores de n , mas na submissão final deverá usar $n = 7$. n deve ser uma variável definida estaticamente no programa. Nomeie o programa **Factorial.S**.