LSDI 2013/14 – Laboratório 5

Sequenciador de instruções para o Caminho de Dados

Nota importante: As secções 1 a 3 devem ser lidas e analisadas em casa, assim como a secção 4 referente à componente de projeto. A secção 5 descreve as etapas de realização do trabalho na aula laboratorial.

1. Introdução

O circuito usado no trabalho laboratorial anterior implementa um "caminho de dados" para a realização de um conjunto de operações entre valores armazenados em determinados registos. Essas operações, os operandos intervenientes e o registo de destino eram especificados com recurso a uma aplicação que corria no computador. O conjunto dos 8 bits que definem esses valores, OPR, SEL e CE, respetivamente, vai ser agora designado por "instrução". Assim sendo, a realização daquilo que designámos por uma "tarefa" exigia a introdução manual de uma sequência de "instruções", formando o que vulgarmente se chama um "programa". Por exemplo, a tarefa "colocar em R1 o produto do valor X (definido nos interruptores) por 10", é realizada pela sequência de instruções apresentada na tabela 1, tendo em consideração que X×10 = X×2 + X×8.

Tabela 1 – Programa que implementa a tarefa de colocar em R1 o produto de X por 10.

Linguagem simbólica	OPR	SEL	CE	Instrução	
				binário	hexadecimal
A ← X	000	100	00	00010000	10
R1 ← A << 1	101	000 *	01	10100001	A1
A ← A << 1	101	000 *	00	10100000	A0
A ← A << 1	101	000 *	00	10100000	A0
A ← A << 1	101	000 *	00	10100000	A0
R1 ← A + R1	010	001	01	01000101	45

* este valor pode de facto ser qualquer, pois a instrução de deslocamento não necessita do operando B

2. Programa residente em memória

Para executar o programa, o utilizador introduzia a primeira instrução, pressionando em seguida o botão (btn3) do sinal de relógio `clock` para executá-la, repetindo este procedimento para as instruções restantes. Este processo pode ser automatizado através de um circuito que consiga fornecer ao caminho de dados essa sequência de instruções, uma a uma, armazenadas previamente numa memória.

2.1 Memória ROM

Uma memória só de leitura, ou ROM (*Read-Only Memory*), é um circuito digital combinacional que tem uma entrada com N bits chamada `endereco` e uma saída com M bits chamada `dados`. Para cada valor colocado na entrada `endereco` o circuito apresenta na sua saída `dados` um valor que foi previamente programado naquele endereço (figura 1).

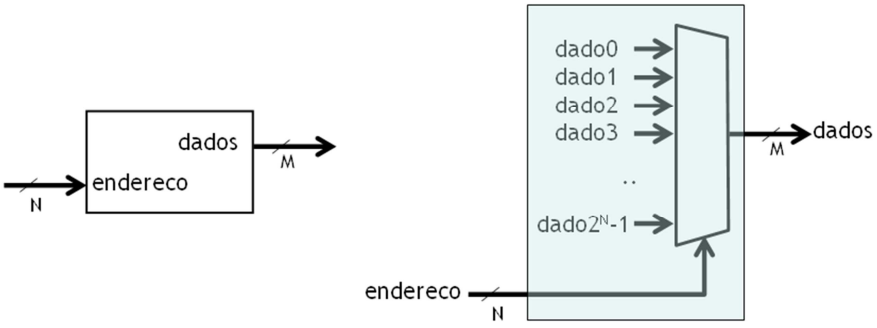

Figura 1 – Memória ROM.

Na figura 1 mostra-se também o circuito equivalente a uma memória ROM: a entrada `endereco` é ligada às linhas de seleção de um multiplexador que escolhe para a saída `dados` o valor (dado0, dado1,...) que corresponde a esse endereço. Pode-se comparar um circuito deste tipo com um arquivo com 2^N "gavetas". A cada endereço (com N bits) corresponde uma única "gaveta" (posição de memória) e dentro de cada uma está guardado um valor com M bits que é designado por dado ou conteúdo da memória naquele endereço.
Por exemplo, uma memória deste tipo, com 4 posições de 8 bits cada uma, pode ser construída em Verilog da seguinte forma:

```
module memoriaROM4x8( endereco, dado );
input [1:0] endereco;
output [7:0] dado;
reg [7:0] dado;
always @*
  case ( endereco )
    2'd0: dado = 8'b1101_0001;   // dado guardado no endereço 0, em binário
    2'd1: dado = 8'hAB;          // dado guardado no endereço 1, em hexadecimal
    2'd2: dado = 8'd29;          // dado guardado no endereço 2
    2'd3: dado = 8'b1010_0001;   // dado guardado no endereço 3
  endcase
endmodule
```

É numa memória como esta que irá ser guardada a sequência de instruções que constituirá um certo "programa", dispensando o utilizador de fornecer o código de cada instrução (OPR, SEL e CE) em cada passo. Se as diversas instruções forem colocadas em endereços consecutivos da memória, a execução do programa obriga a que seja aplicado na entrada `endereco` da ROM uma sequência de números inteiros (endereços), iniciada com o endereço onde está guardada a primeira instrução (em geral, zero).

2.2 Circuito

A figura 2 mostra o circuito base que será implementado neste trabalho laboratorial e que utiliza uma memória ROM contendo o programa a executar. Um contador binário produz o endereço para a memória ROM, constituindo o chamado "contador de programa", ou PC (*Program Counter*). Assim, para executar o programa guardado na ROM basta fazer *reset* para colocar o contador em zero, ou seja, o endereço igual a 0, e premir sucessivamente o botão de relógio que fará incrementar o endereço aplicado na ROM.

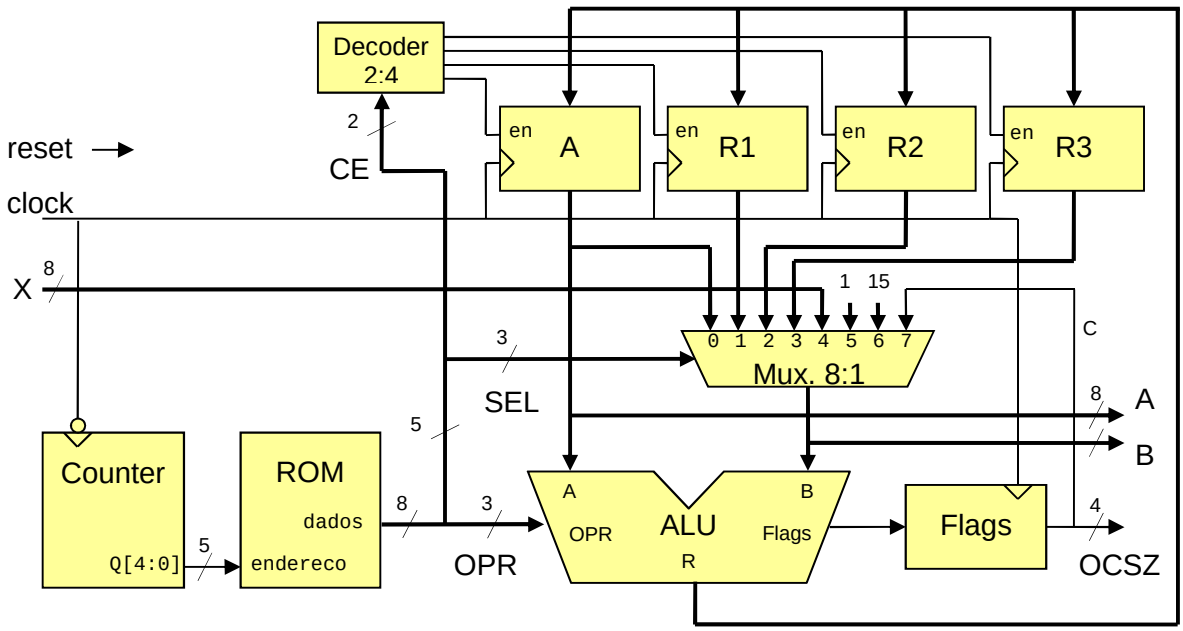

Figura 2 – Circuito que permite a execução de um programa residente em memória.

Tal como já acontecia no trabalho anterior, os registos mudam de estado quando a entrada `clock` do circuito transitar de zero para um, ou seja, quando é premido o botão de pressão btn3 que está ligado ao sinal `clock`. No entanto, o sinal de relógio que entra no contador aparece negado, significando que o incremento do contador

só se verifica na transição descendente do `clock`, ou seja, quando se liberta o respetivo botão btn3 (figura 3). Desta forma consegue-se executar uma instrução em cada ciclo de relógio.

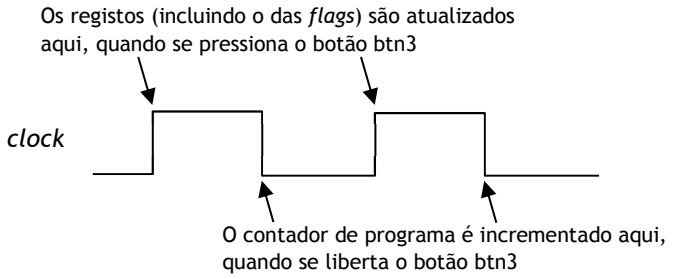

Figura 3 – O sinal de relógio atualiza os registos (dados e *flags*) na transição positiva e o contador de programa na transição negativa.

Tal como se pode ver na figura 4, as restantes entradas e saídas continuam a ser as mesmas do trabalho anterior.

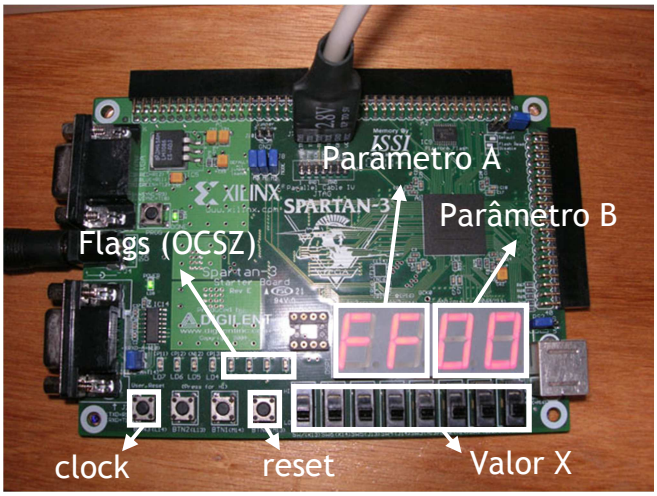

Figura 4 – Entradas e saídas do circuito na placa S3board.

Note que neste circuito o utilizador não pode mudar diretamente o sinal SEL, pois agora é a memória a responsável pela sua geração. Torna-se pois impossível ver o conteúdo dos registos que não o registo A. Contudo, existe uma aplicação a correr no computador (versão LSDI_Lab5b) que permite conhecer em qualquer instante o valor de todos os registos do caminho de dados, assim como o endereço e a próxima instrução a ser executada. A figura 5 mostra a interface desta aplicação. O botão Hex, ativo por omissão, permite escolher a visualização de todos os valores em hexadecimal ou no formato decimal.

Figura 5 – Interface da aplicação LSDI_Lab5b.exe.

Para o correto funcionamento da aplicação é importante que sempre que se faça um *reset* na placa se carregue em seguida no botão "Após fazer reset na placa carregue aqui !!!".

3. Execução de instruções de salto

O circuito descrito na secção anterior permite a execução de pequenos programas (até 32 instruções, porque o contador usado tem 5 bits) guardados numa memória ROM. Após ativar o sinal `reset` e ao ritmo do sinal de relógio aplicado manualmente, é executada a sequência de instruções previamente codificada na memória ROM. Essa sequência é iniciada no endereço zero, continuando nos endereços consecutivos seguintes, não havendo forma de alterar a ordem de execução das instruções. Assim, vai de seguida mostrar-se o interesse de permitir a alteração desta ordem e expandir o circuito anterior de forma a permitir essa funcionalidade.

3.1 Salto incondicional

Considere-se um programa que coloca no registo A o dobro do valor presente em X, cuja "listagem" está apresentada na tabela 2.

Tabela 2 – Programa que coloca em A o dobro de X.

Endereço	Instrução	Descrição
0	000_100_01	R1 ← X
1	000_001_00	A ← R1
2	010_001_00	A ← A + R1

E se agora se pretendesse colocar no registo A o triplo de X? Uma solução possível seria repetir a última instrução tal como mostra o programa da tabela 3.

Tabela 3 – Programa que coloca em A o triplo de X.

Endereço	Instrução	Descrição
0	000_100_01	R1 ← X
1	000_001_00	A ← R1
2	010_001_00	A ← A + R1
3	010_001_00	A ← A + R1

E se o objetivo fosse colocar no registo A uma sequência de múltiplos de X (2X, 3X, 4X,…)? Embora fosse possível ir acrescentando instruções A ← A + R1, este procedimento não só seria pouco "elegante", como também iria rapidamente encher a memória disponível.
Uma solução mais interessante será repetir a execução da instrução no endereço 2 (A ← A + R1), em vez de seguir para a execução da instrução no endereço 3. Para tal, vai criar-se uma nova instrução que permita efetuar um salto para o endereço 2: `JMP endereco`, onde `endereco` representa o endereço para o qual se pretende "saltar" (tabela 4).

Tabela 4 – Programa que coloca em A os múltiplos de X.

Endereço	Instrução	Descrição
0	000_100_01	R1 ← X
1	000_001_00	A ← R1
2	010_001_00	A ← A + R1
3	101_00010	JMP 2

Como as oito instruções que o valor de OPR permite codificar já foram todas definidas (ver tabela 1 do guião do trabalho nº 3), não é possível acrescentar uma nova instrução. Contudo, como a instrução R = A << 1 (código OPR=101), usada para multiplicar por 2, pode ser alternativamente realizada como R = A + A, o seu código vai passar a ser usado para definir a nova instrução de salto.

A instrução `JMP endereco` tem a seguinte codificação:

- Os bits 7 a 5 da instrução (OPR) são iguais a 101.
- Os bits 4 a 0 definem o parâmetro `endereco` que representa o endereço (de 0 a 31) da instrução seguinte a executar.

Circuito para execução de salto incondicional

Vejam-se na figura 6 as alterações ao circuito anterior (figura 2) necessárias à execução desta nova instrução. O contador responsável pela geração dos endereços terá agora que possuir uma funcionalidade adicional: poder ser "carregado" (*load*) com um determinado valor presente nas suas entradas `I[4:0]`. Assim, sempre que se quiser efetuar um salto, bastará indicar o endereço para onde saltar na entrada `I` do contador e ativar o sinal `jump` ligado à respetiva entrada de `load`. Quando isto acontece, o contador é carregado com os 5 bits menos significativos da instrução, passando a ser esse o endereço da próxima instrução a ler da memória.

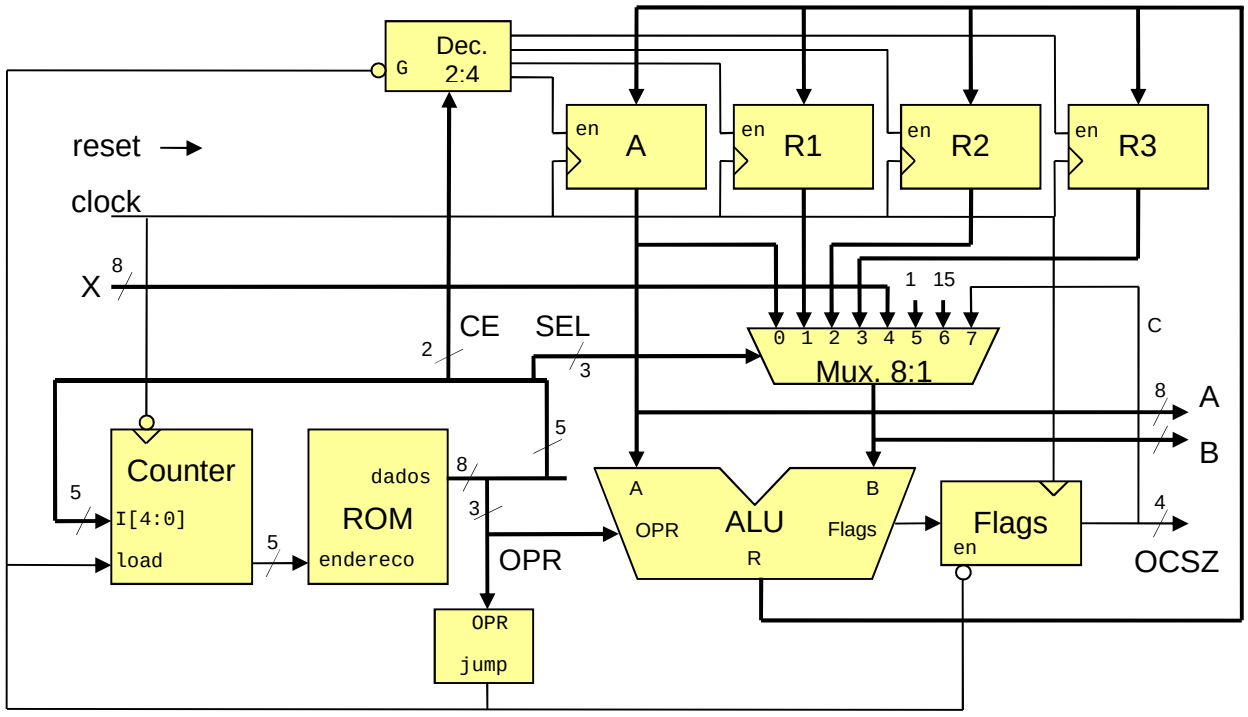

Figura 6 – Circuito para a implementação de um salto incondicional (`JMP endereço`).

A função lógica que gera o sinal `jump` é muito simples: `jump` é 1 se OPR=101 e 0 para os restantes valores de OPR. Contudo, quando OPR=101 a ALU continuará a produzir na sua saída R o resultado do deslocamento do registo A de 1 bit para a esquerda. Se for mantida a estrutura do circuito anterior, esse resultado será guardado num dos quatro registos em função do valor de CE aplicado ao descodificador de 2 para 4. Além disso, também o registo Flags será atualizado. Como numa instrução de salto só interessa mesmo realizar o salto e não modificar qualquer registo, essa ação deverá ser inibida. Isto pode ser feito acrescentando um sinal de *enable* (`G`) no descodificador, permitindo inibir a ativação de qualquer uma das suas saídas, e no registo Flags. Assim, como se pretende que o descodificador não ative as suas saídas nem o registo Flags seja atualizado quando `jump=1`, essas entradas de *enable* deverão ser controladas pelo sinal `jump` negado, conforme indicado (figura 6).

3.2 Salto condicional

O tipo de salto anterior designa-se por incondicional porque o salto realiza-se sempre. Contudo, pode haver interesse em que um salto só se realize caso uma determinada condição seja verdadeira: por exemplo, uma das *flags* esteja ativa. Neste caso o salto diz-se "condicional". Por exemplo, para implementar um salto em que a condição seja "*flag* de zero não activa" (`JNZ`, ou *jump if not zero*) basta modificar o sinal que atua o *load* do contador conforme apresentado na figura 7. O salto só é realizado se a instrução corrente é OPR=101 e a *flag* de zero é nula.

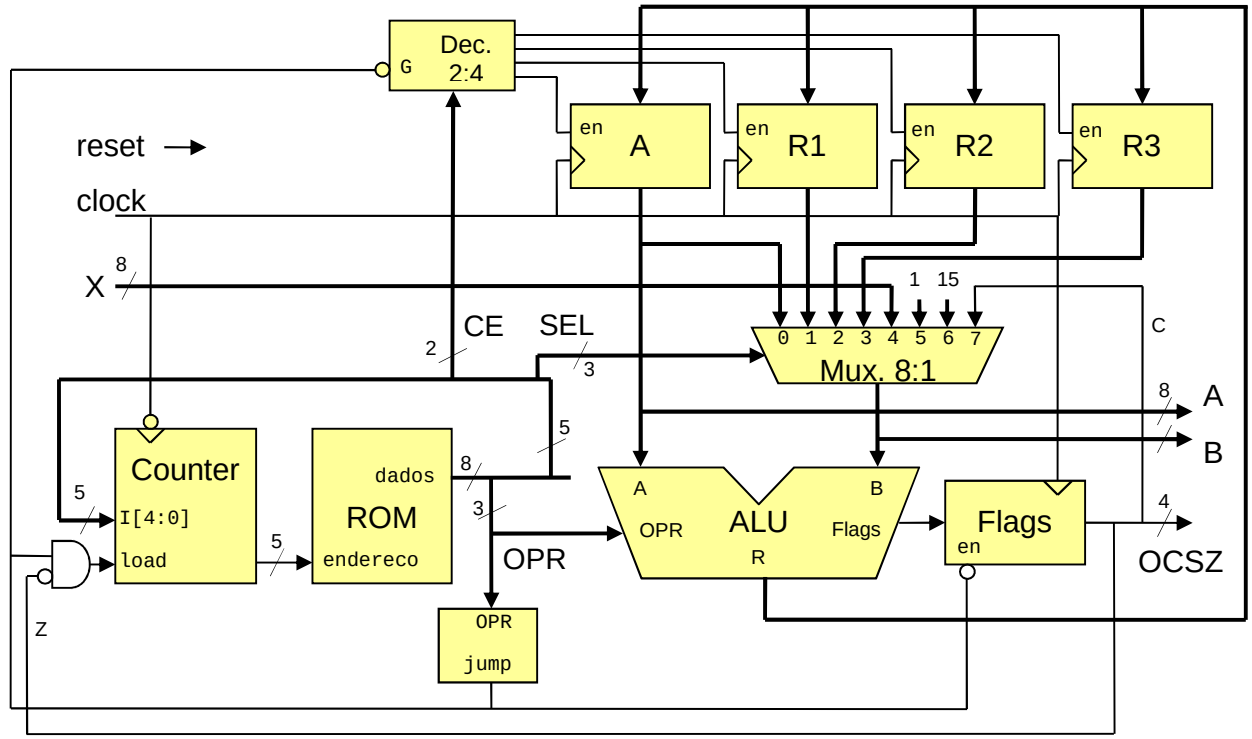

Figura 7 – Circuito para a implementação de um salto condicional (`JNZ endereco`).

Note que embora as instruções de salto incondicional e salto condicional sejam diferentes, elas podem coexistir num mesmo programa e serem implementadas pelo mesmo circuito, desde que se recorra a `JNZ` (salto condicional) para executar ambas as formas. Efetivamente, `JMP` (salto incondicional) é equivalente a `JNZ` se a condição de salto for verdadeira. Portanto, um salto incondicional pode ser implementado recorrendo a `JNZ` se imediatamente antes de `JNZ` se executar uma instrução que cause um resultado diferente de zero, por exemplo, R←1 (OPR=000, SEL=101 e CE=código de um registo R livre).

4. Projeto

Antes da aula laboratorial deve preparar os programas a ensaiar na aula.

a) Projete o módulo da memória ROM.v de modo a ficar nela armazenado o programa exemplificado na tabela 4 (secção 3.1), que calcula os múltiplos de X. Utilize como ponto de partida o exemplo dado na secção 2.1, nomeadamente a zona sombreada em que se define o programa como um conjunto de instruções em endereços sucessivos. Utilize o ficheiro ROM.v incluído no arquivo do projeto ISE LSDI2013_lab5.

b) Escreva um programa que permita calcular o *n*-ésimo ($n \geq 1$) termo de uma progressão aritmética $u_n = u_1 + (n-1) \times r$. O programa deve começar por colocar o termo inicial u_1 no registo A, o valor da razão r em R1 e a ordem n do termo pretendido no registo R2. Estes três valores devem ser definidos através da entrada X, proveniente dos interruptores da placa, e o resultado u_n deve ficar no registo A.

c) A série de Fibonacci é uma sequência de números em que cada valor é obtido somando os dois últimos valores da sequência. Por convenção, os dois primeiros números da sequência são iguais a 1. Assim sendo, a sequência é: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, … . Elabore um programa que obtenha no registo A o X-ésimo valor da série de Fibonacci. Por exemplo, se X for 10, A deverá ficar com o valor 55 após a execução do programa.

5. Implementação e ensaio

Na aula laboratorial vai implementar e ensaiar o circuito final apresentado na figura 7, devendo para isso abrir o projeto LSDI2013_lab5 e efetuar os passos a seguir indicados.

- Comece por editar o módulo ROM.v com o conteúdo que preparou em casa referente a **4.a)**: programa que gera os múltiplos de X (exemplo da tabela 4).
- Implemente o circuito na placa S3board e faça o respetivo ensaio executando o programa contido em memória, verificando se de facto é realizada a tarefa pretendida (mostrar em A os sucessivos múltiplos de X, sendo este definido com os interruptores da placa antes de provocar a primeira transição do sinal de relógio). Para tal, tenha em consideração as entradas e saídas da placa S3board (figura 4), e a utilização da aplicação LSDI_Lab5b.exe (figura 5). Não esqueça que se fizer alguma alteração ou correcção ao programa, deverá alterar o conteúdo de ROM.v e implementar de novo o circuito.

- Modifique o módulo ROM.v de modo a conter o programa que permite obter um termo de uma progressão aritmética, conforme descrito em **4.b)**. Note que o circuito que está a usar é o da figura 7, pelo que necessitando de executar uma instrução de salto incondicional `JMP` deverá assegurar-se que a flag Z é diferente de zero, nem que para isso tenha de executar uma instrução que não altere nenhum conteúdo importante para o programa mas que origine R diferente de zero (por exemplo, se R3 não estiver a ser usado, pode ser `R3 ← 1`).
- Implemente o circuito na placa e faça o respetivo ensaio executando o atual programa contido em memória.

- Altere o conteúdo da memória ROM de acordo com o programa que preparou em casa, conforme descrito em **4.c)**, para gerar números da série de Fibonacci.
- Volte a implementar o circuito na placa e efetue o ensaio verificando se o programa funciona adequadamente.