Instituto Superior Técnico

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Arquitecturas Avançadas de Computadores

Simulação de um processador μRisc com funcionamento multi-ciclo

Guilherme Branco Teixeira	n.º 70214
Maria Margarida Dias dos Reis	n.º 73099
Nuno Miguel Rodrigues Machado	n.º 74236

Lisboa, 29 de Março 2014

Índice

[1] É de referir que as imagens e tabelas não foram colocadas em anexo de modo a permitir uma melhor compreensão do relatório.

1 Introdução

Com este trabalho laboratorial pretende-se projectar um processador μRisc, de 16 *bits* com arquitectura RISC, com um funcionamento multi-ciclo. O processador possui 8 registos de uso geral e 42 instruções, sendo que cada instrução demora quatro ciclos a completar, um ciclo por cada andar do processador. O processador é feito com recurso a uma linguagem de descrição de *hardware* - VHDL.

2 Características do Processador

O processador elaborado foi simulado para uma placa Artix 7 e tem as seguintes características: 16 *bits*; 8 registos de uso geral de 16 *bits* de largura (R0, . . . , R7); 42 instruções; instruções de 3 operandos; organização de dados na memória do tipo *big endian*; uma memória ROM de 8 KBytes (4k endereços $\times$ 2 *bytes*) endereçada com palavras de 12 *bits* utilizada para as instruções/programa e uma memória RAM de 8 KBytes (4k endereços $\times$ 2 *bytes*) endereçada com palavras de 12 *bits* utilizada para os dados.

3 Estrutura do Processador

O processador μRisc que foi projectado encontra-se dividido em quatro andares - num primeiro andar é feito o *instruction fetch* (IF), no segundo andar é feito o *instruction decode* (ID) e o *operand fetch* (OF), no terceiro andar são executadas operações da ALU (EX) e de acesso à memória de dados (MEM) e, por fim, no quarto e último andar é feita a escrita no banco de registos, o *write back* (WB).

3.1 Primeiro Andar - IF

No primeiro andar obtem-se a instrução a ser executada a cada ciclo. Como todas as instruções do programa são armazenadas na memória ROM, o *instruction fetch* tem a função de endereçar a ROM com o *program counter* (`PC`) e ler a instrução desse endereço.

FIGURA 1

O *instruction fetch* é simplesmente um somador que, em cada ciclo, soma `1` ao endereço actual e armazena o resultado no registo `PC`, como se pode ver na Figura 1. O endereço actual, além de ser um operando do somador, também endereça a memória ROM.

Podem ocorrer duas situações que alteram o funcionamento sequencial do *instruction fetch* - a primeira ocorre quando há uma transferência de controlo do tipo condicional ou incondicional, seleccionando o sinal `destino_cond` no `MUX_1` e o resultado do somador no `MUX_2`, ou seja, o sinal `s_cond` está a *high* e `s_jump` a *low*. A última situação ocorre quando existe uma transferência de controlo do tipo *jump and link* ou *jump register*, selecionando o sinal `destino_jump` no `MUX_2`, ou seja, o sinal `s_jump` está a *high*.

se calhar e melhor explicar o significado destes sinais

Tabela 1: Caracterização do registo de saída do andar de *instruction fecth*.

Bits do registo de saída do andar IF	Sinal correspondente
27 downto 12	instrução
11 downto 0	PC + 1

3.2 Segundo Andar - ID e OF

3.2.1 ID

No segundo andar é realizada a descodificação da instrução a ser executada, sendo a principal descodificação feita neste andar, com um só sinal descodificado no último, que será explicado mais à frente no relatório.

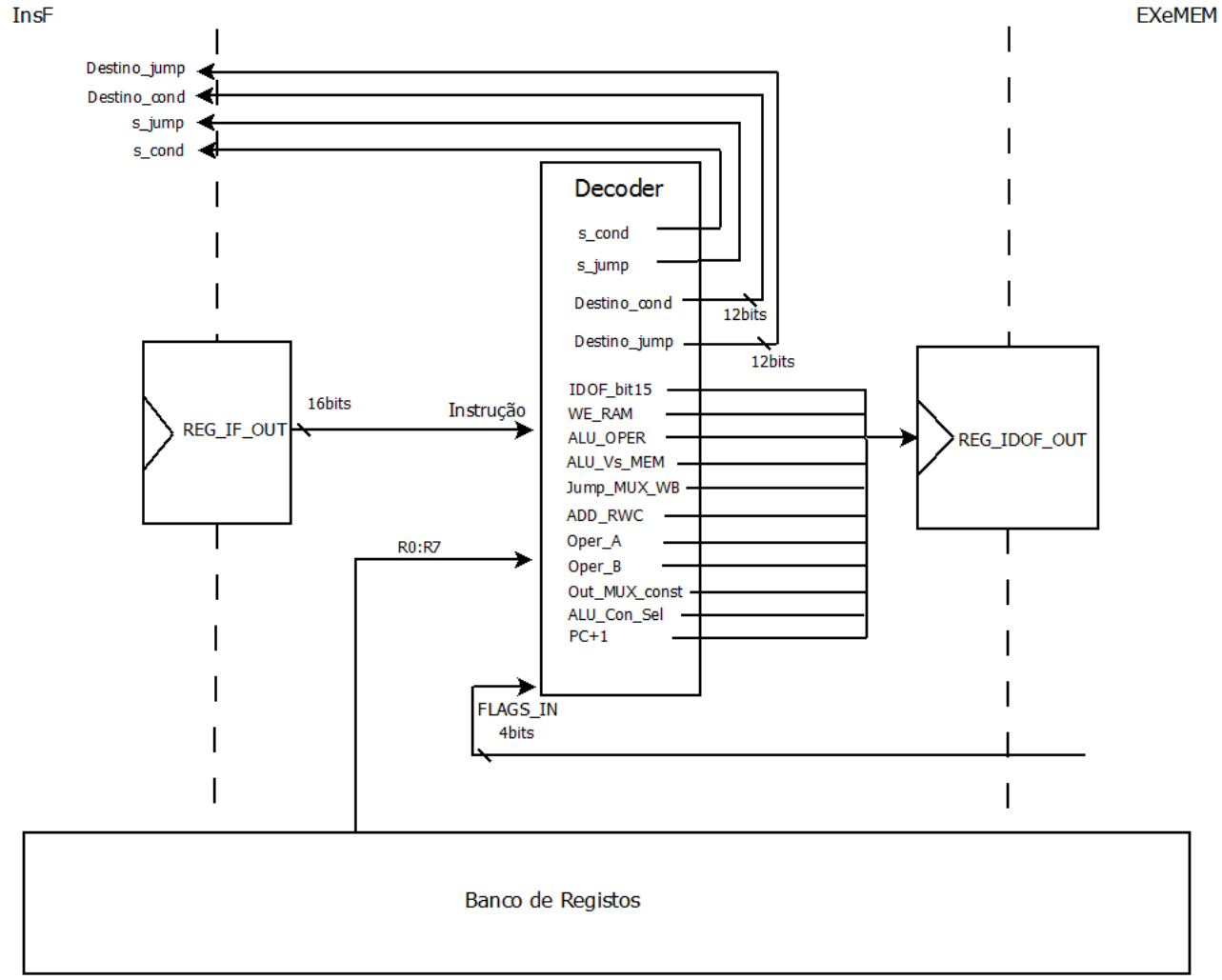

Figura 1: Selecção dos operando `A` e `B`.

Analisando a figura anterior, existe três grupos de sinais de entrada.

- O sinal `instrução` proveniente do primeiro andar, *instruction fetch*, é um sinal de 16 *bits* que representa a instrução a ser descodificada;

- Os sinais `R0 ... R7`, provenientes do banco de registos, devolvem o valor do registo respectivo, sendo posteriormente selecionados os registos indicado pela instrução para operandos;
- O sinal `FLAGS_IN` é o resultado da actualização das *flags* realizado no terceiro andar.

Os sinais de saída do descodificador são distribuídos pelos 4 andares do processador, como se verá.

3.2.1.1 Sinais de controlo para o primeiro andar - IF

Os sinais de controlo para o primeiro andar são referentes às instruções do tipo de transferência de controlo. Há dois tipos principais de operações - salto relativo ao `PC + 1` ou salto absoluto.

Para poder haver um salto é necessário detectar se a instrução é do tipo de transferência de controlo e, com essa ideia em mente, foi criado o sinal auxiliar `Active_FlagTest`. A figura abaixo mostra a função lógica de verificação.

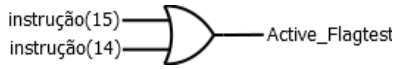

Figura 2: Construção do sinal Active_Flagteste

De seguida, é preciso saber qual o tipo de salto a ser executado.

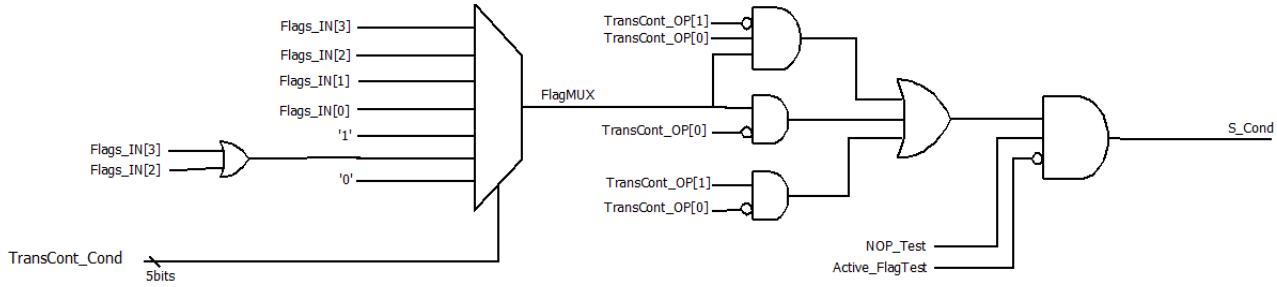

Figura 3: Selecção da constante a carregar.

A figura anterior representa a função lógica do sinal `s_cond` que controla o salto relativo ao `PC + 1`. É de referir que o sinal `TransCont_OP` corresponde aos *bits* `12` e `13` da instrução, ou seja, é um sinal de 2 *bits* que identifica as operações de transfêrencia de controlo. O sinal `TransCont_Cond` corresponde aos *bits* `8` a `11`, ou seja, é um sinal de 4 *bits* que identifica qual a *flag* a ser testada.

Há três tipos diferentes de operações em que pode ocorrer este tipo de salto.

- Quando há um *jump* incondicional: `TransCont_OP = "10"`, `Active_FlagTest = 0` e `NOP_Test = 1`;
- Quando há um *jump false*: `TransCont_OP = "00"`, `Active_FlagTest = 0`, `NOP_Test = 1` e o resultado do MUX de 8:1 é *false*, ou seja, a *flag* escolhida do registo `MSR` é *false*;
- Quando há um *jump true*: `TransCont_OP = "00"`, `Active_FlagTest = 0`, `NOP_Test = 1` e o resultado do MUX de 8:1 é *true*, ou seja, a *flag* escolhida do registo *MSR* é *true*.

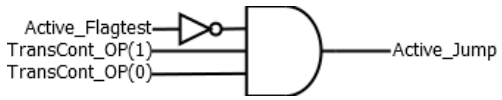

Figura 4: Selecção da constante a carregar.

A figura anterior representa a função lógica do sinal `s_jump` que controla o salto absoluto. O sinal `s_jump` é activado, `s_jump = 1`, quando `active_FlagTest = 0` e o `TransCont_OP = "11"`.

3.2.1.2 Sinais de controlo para o segundo andar - ID e OF

Os sinais de controlo para o segundo andar são referentes ao controlo do *operand fetch*. Para que o *operand fetch* defina os operandos A e B é necessário identificar os endereços desses operandos.

O sinal `ADD_RA_C` define o endereço do operando `A`. Como a instrução do tipo constantes reutiliza o endereço de escrita como leitura, `ADD_RWC`, é necessário acrescentar lógica para poder resolver o problema destas operações. A solução encontrada foi utilizar um MUX de 2:1 de forma a escolher o endereço `ADD_RA` (sinal de 3 *bits* que é igual ao sinal `instrução(5 downto 3)`) ou `ADD_RWC` (sinal de 3 *bits* que é igual ao sinal `instrução(13 downto 11)`). A figura abaixo define como se obtém o endereço.

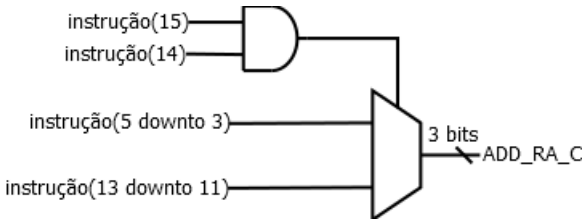

Figura 5: Mux que permite selecionar

O sinal `ADD_RB` (sinal de 3 *bits* que é igual ao sinal `instrução(2 downto 0)`) é simplesmente o endereço do registo para obter o operando `B`.

O sinal `select_mux_constantes` (sinal com dimensão de 2 *bits*) é o *bit* `15` do sinal `instrução` concatenado com o *bit* `10` do mesmo sinal. Este sinal tem a função de selecionar se a operação de constantes é do tipo *lcl*, `select_mux_constantes = "10"`, do tipo *lch*, `select_mux_constantes = "11"` ou um carregamento directo de uma constante de 11 *bits* com o *bit* de sinal replicado, `select_mux_constantes = "00"` e `select_mux_constantes = "01"`.

3.2.1.3 Sinais de controlo para o terceiro andar - EX e MEM

O sinal de controlo `oper_ALU` de 5 *bits* tem como objectivo ajudar na descodificação de todas as operações da ALU, e posteriormente na criação de outros sinais de controlo. Este sinal de 5 *bits* é retirado do sinal referente à instrução recebida.

O sinal de controlo `ALU_vs_MEM` tem como objectivo destinguir entre instuções que são efectuadas na ALU e instruções de memória, este sinal será útil especialmente para controlar as escritas na memória, tal como exemplificado na tabela 7. Este sinal é obtido como demonstrado na figura:

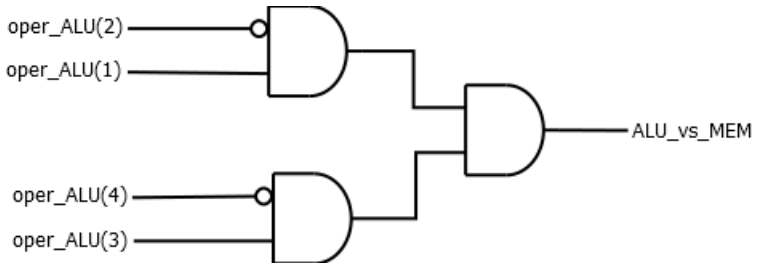

Figura 6: Construção do sinal ALU_vs_MEM.

O sinal de controlo `WE_RAM` tem como objectivo permitir a escrita na RAM, e utiliza o sinal `ALU_vs_MEM` representado na figura 6, a sua construção pode ser observada na seguinte figura:

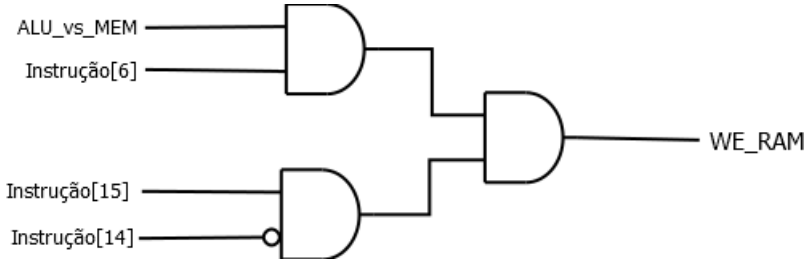

Figura 7: Construção do sinal WE_RAM.

3.2.1.4 Sinais de controlo para o quarto andar - WB

IDOF_bit15 ALU_Con_Sel ADD_RWC PC+1

Os dois sinais de controlo de um *bit*, `IDOF_bit15` e `ALU_Con_Sel`, correspondem respectivamente ao *bit* 15 e *bit* 14 do sinal referente à instrução recebida. Estes dois sinais, como se pode verificar na tabela 7, servem para descodificar o sinal overwrite, a única descodificação fora do andar ID (Primeiro Andar).

O sinal de controlo `ADD_RWC`, como explicado na secção 3.4. Este sinal irá servir como sinal de selecção do mux da figura 16 que selecciona a informação que vai ser escrita nos registos.

No caso da instrução recebida ser *jump and link*, o sinal `PC+1` guarda o valor PC+1 que terá como destino ser escrito no registo 7, como tal este sinal tem entrada no mux apresentado na figura 16.

LEGENDA BOAS

3.2.2 OF

É também no segundo andar que é feito o *operand fetch*. Numa primeira fase é preciso definir os operandos `A` e `B` da ALU, feito de acordo com a seguinte lógica.

explicar WE da RAM, que eu depois uso quando explico a MEM. nao esta no desenho do decoder?

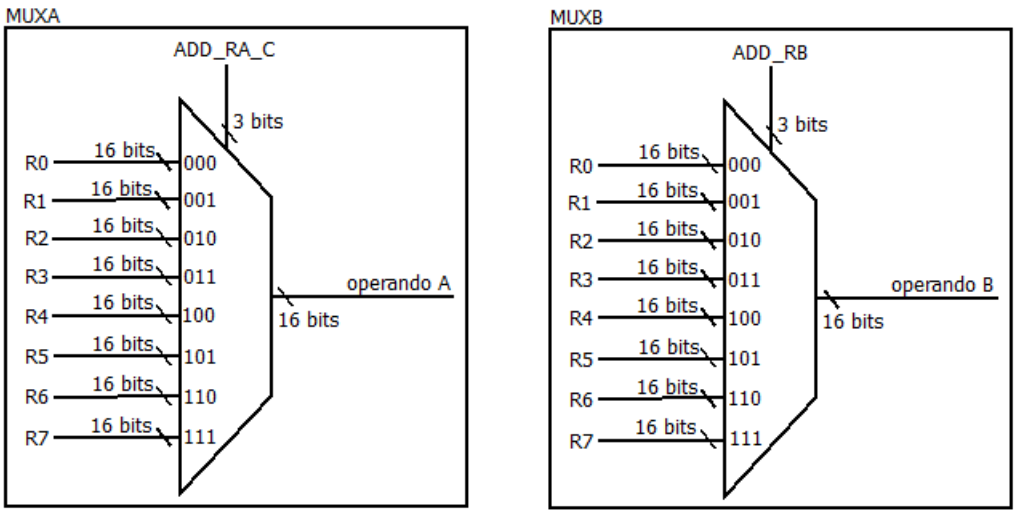

Figura 8: Selecção dos operando `A` e `B`.

Para fazer a selecção é necessário recorrer a alguns sinais que o *decoder* explicado anteriormente fornece - `ADD_RA_C` (sinal 3 *bits* que permite fazer a selecção do operando `A` no `MUXA`) e `ADD_RB` (sinal 3 *bits* que permite fazer a selecção do operando `B` no `MUXB`).

Os sinais que definem o operando `A` e o operando `B` são depois passados para o terceiro andar para que a ALU possa fazer operações com o seu valor.

É também aqui que se faz a selecção das constantes que depois serão carregadas nos registos do banco de registos, algo que é feito de acordo com a próxima figura.

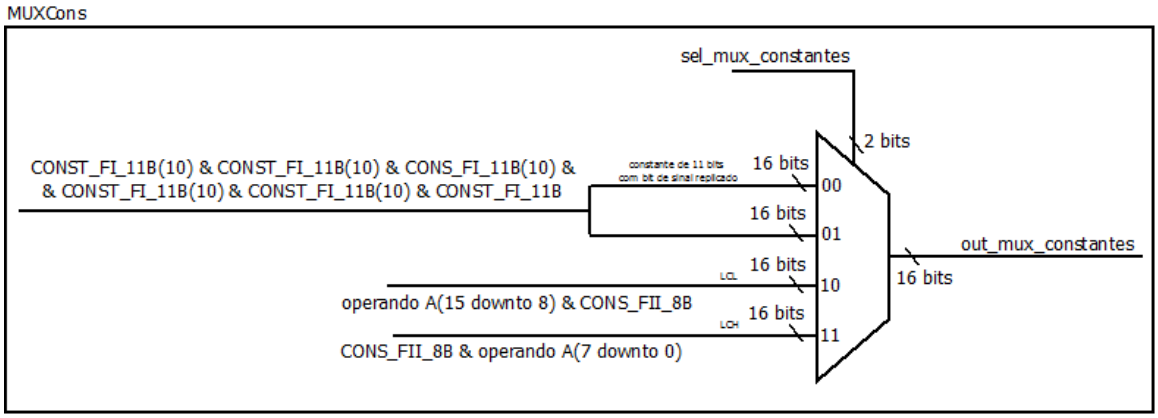

Figura 9: Selecção da constante a carregar.

Do *decoder* são fornecidos os seguintes sinais - `CONS_FI_11B` (constante de 11 *bits* que é carregada directamente), `CONS_FII_8B` (constante de 8 *bits* com que é feita uma operação de *lch* ou *lcl*) e `select_mux_constantes` (sinal de 2 *bits* que permite fazer a selecção dos três casos definidos anteriormente no `MUXCons`).

este sinal nao esta no decoder?

De referir que,

explicar que não faço ands, é so fios

Tabela 2: Caracterização do registo de saída do andar de *instruction decoding* e *operand fetch*.

Bits do registo de saída do andar ID e OF	Sinal correspondente
73	instrução(15)
72	WE_RAM
70 downto 66	oper_ALU (instrução(10 downto 6))
65	ALU vs MEM
64 downto 53	PC + 1
52	JUMP_MUXWB_OUT (JAL)
51 downto 49	ADD_RWC (instrução(13 downto 11))
48 downto 33	operando A
32 downto 17	operando B
16 downto 1	out_mux_constantes
0	instrução(14)

3.3 Terceiro Andar - EX e MEM

Neste andar trata-se de executar operações da ALU bem com operações da memória, sendo que, ao contrário do MIPS, em que é possível utilizar a ALU e a memória na mesma instrução, no processador μRisc projectado tal não é possível. A memória RAM está colocada no mesmo andar que a execução porque não é necessário fazer cálculos dos endereços, se tal fosse necessário, a memória teria de estar no andar seguinte àquele que contém a ALU.

3.3.1 ALU (EX)

No terceiro andar o bloco da ALU é responsável pelas operações aritméticas e lógicas. Este bloco recebe como entrada os sinais dos operandos `A` e `B`, um sinal de 4 *bits* com as *flags* actuais para posterior actualização, se for esse o caso, e também um sinal de 5 *bits* que representa a operação que a ALU vai efectuar. Como saída tem-se um sinal de 16 *bits* que representa o resultado da ALU e um sinal de 4 *bits* que representa as *flags*.

Analisando primeiramente as seis operações aritméticas a realizar concluiu-se que algumas podiam ser simplificadas de modo a que todas pudessem ser efectuadas com recurso a apenas um somador. A seguinte tabela demonstra como todas as operações aritméticas a realizar podem ser calculadas apenas com um somador.

Tabela 3: Caracterização somador utilizado nas operações aritméticas.

Operação	A + B	A + B + 1	A + 1	A - B - 1	A - B	A - 1
oper_ALU(2 downto 0)	000	001	011	100	101	110
Operando P	A	A	A	A	A	A
Operando Q	B	B	0	!B	!B	-1
Carry in	0	1	1	0	1	0

Esta solução é mais eficiente pois os somadores têm um tempo de propagação elevado. As seis operações aritméticas podem ser feitas usando um somador com *carry-in*, que efectua o seguinte cálculo:

`out_ARI` = `P` + `Q` + `cIN`.

O operando `P` corresponde sempre ao operando `A`, o operando `Q` corresponde a um sinal diferente dependendo da operação aritmética a realizar, tal como o *carry-in* que é usado para operações como incrementos, podendo também completar o `!B` em complemento para dois. Os dois sinais de entrada do somador recebem uma concatenação com o bit `0` como o bit mais significativo, sendo isto feito para que a saída do somador tenha 17 *bits*, tornando possível a actualização da *flag* de *cary*. Este somador foi projectado tal como representado na figura abaixo.

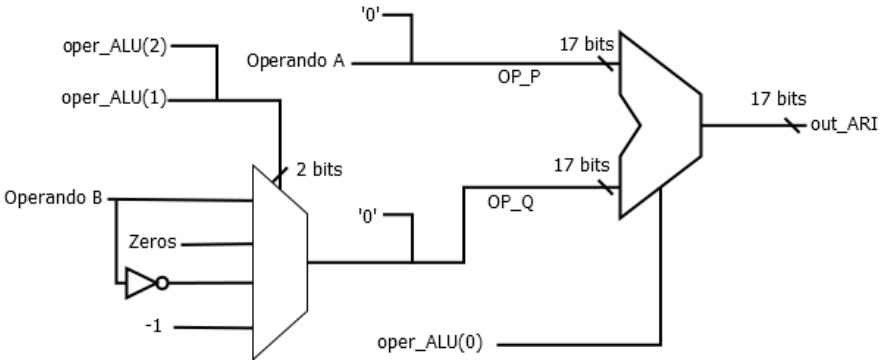

Figura 10: Esquema do bloco aritmético da ALU.

No caso das operações de *shift*, tal como nas operações aritméticas, a saída é representada com 17 *bits*, pela mesma razão do *cary*. Para escolher entre as operações de *shift* é usado um MUX de 2:1 tal como está *pseudo*-representado na figura.

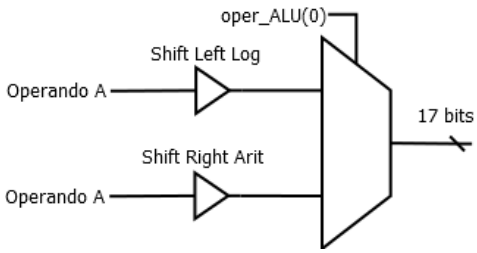

Figura 11: Esquema do bloco de operações de *shift* da ALU.

No caso das operações lógicas, que representam 16 operações, fez-se um esforço para reduzir um MUX de 16:1 para um MUX de 8:1 devido à diferença de tempo gasto entre os dois MUXs. Após uma análise cuidada das operações a realizar, foi possível establecer uma relação entre as operações, tal como se pode observar na tabela abaixo.

Tabela 4: Descrição das operações lógicas a realizar.

Operação	0	A&B	!A&B	B	A&!B	A	A XOR B	A|B
oper_ALU(3 downto 0)	0000	0001	0010	0011	0100	0101	0110	0111
!Operação	-1	!A|!B	A|!B	!B	!A|B	!A	!(A XOR B)	!A&!B
!oper_ALU(3 downto 0)	1111	1110	1101	1100	1011	1010	1001	1000
Select_LOG	000	001	010	011	100	101	110	111

Ao observar o sinal `oper_ALU(2 downto 0)` e as suas operações podemos perceber que quando temos um valor específico de `oper_ALU(2 downto 0)` com a sua respectiva operação, o valor negado desse valor

representa a negação da operação. Podendo assim criar um mux apenas com 8 entradas entradas e com um sinal de selecção que permite selecionar a operação X quando de facto desejamos a operação !X, como representa a seguinte figura.

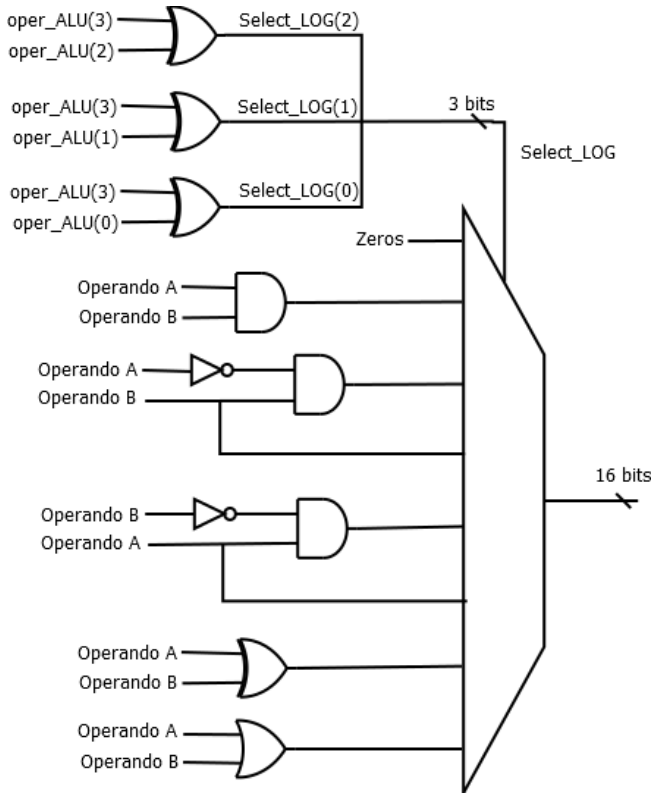

Figura 12: Esquema do bloco de operações lógicas da ALU.

Será no entanto, quando necessário, negar a operação, esta negação será efectuada no mux final da ALU como demonstrado na figura

De notar que a saída das operações lógicas necessita apenas de 16 *bits* e não 17 pois uma operação lógica não produz *carry*.

Finalmente, após a verificação das *flags*, tem-se um MUX de 4:1, onde as entradas de 17 *bits* são reduzidas para 16 *bits*, retirando-lhes o *bit* mais significativo que, relembre-se, tinha como objectivo a verificação da flag de *cary*. A saída deste MUX é um sinal de 16 *bits* que representa o resultado da operação da ALU, tal como se pode verificar na figura seguinte.

Na Figura 13 encontra-se um esquema completo da ALU.

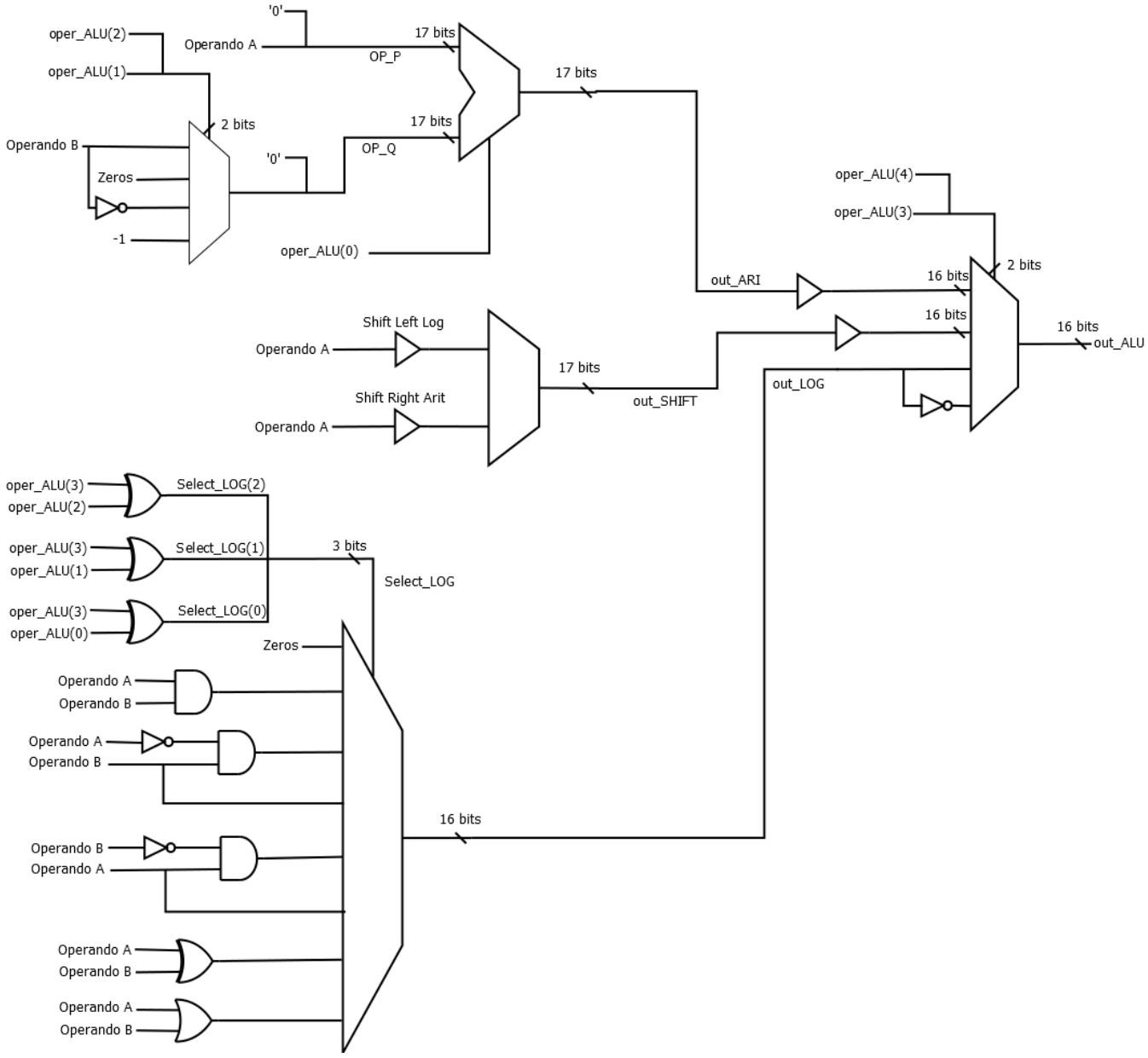

Figura 13: Esquema do bloco de operações lógicas da ALU.

Para cada um dos três sinais de saída (`out_ARI`, `out_SHIFT` e `out_LOG`) é criado um sinal que corresponde às *flags* que cada operação pode actualizar. No caso das operações aritméticas é criado um sinal de 4 *bits* com as quatro *flags* atualizadas indiscriminadamente. No caso das operações de *shift* é criado um sinal de 3 *bits* apenas pois, as *flags* que podem vir a ser actualizadas nessas duas operações, são a flag de `Zero`, de `Negative` e de `Carry`. No caso das operações lógicas é criado um sinal com apenas dois *bits* que representam as *flags* que poderão ser atualizadas neste tipo de operações, a *flag* de `Zero` e de `Negative`.

Estes três sinais criados serão utilizados no bloco de actualização de *flags*, tal como explicado na secção 3.3.2.

3.3.2 Actualização das *Flags*

Este bloco recebe como entrada o sinal de 4 *bits* que representa as *flags* da operação realizada na instrução anterior, os três sinais criados na ALU que representam as *flags* atualizadas indiscriminadamente e também o sinal de 5 *bits* que representa a operação que a ALU efectuou de modo a que seja possível discriminar que *flags* actualizar. A saída é um sinal de 4 *bits* que representa as *flags* atualizadas discriminadamente.

Ao analisar o quadro de operações da ALU, é possível reparar que existem apenas quatro tipos de atualizações de *flags*:

- Nenhuma;
- `Zero` e `Negative`;
- `Zero`, `Negative` e `Carry`;
- `Zero`, `Negative`, `Carry` e `Overflow` (Todas).

Foi então criado um sinal que tem como objectivo discernir de entre todas as *flags* quais a actualizar. Este sinal foi criado com a lógica representada na seguinte figura.

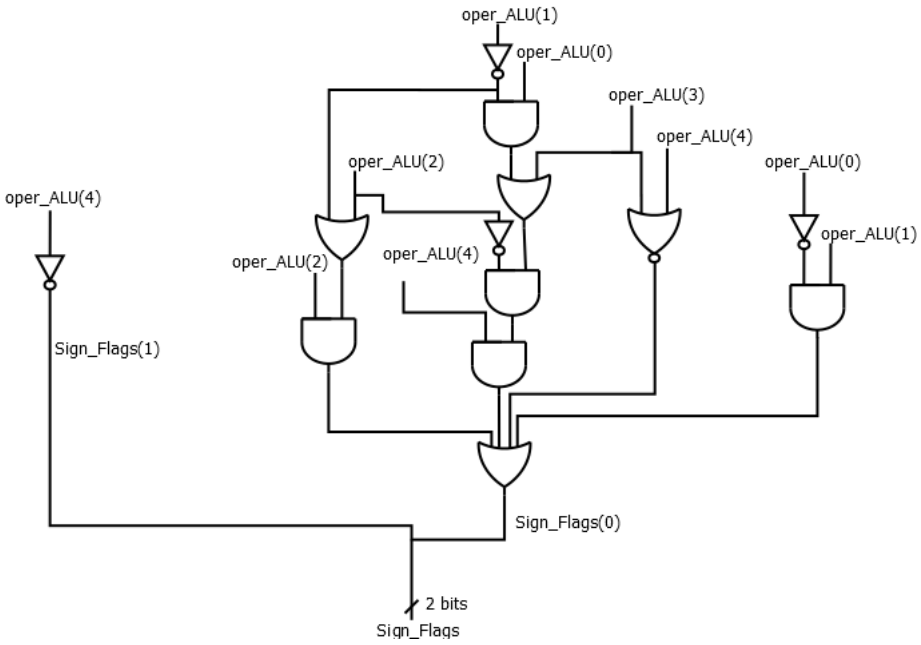

Figura 14: Lógica que calcula o sinal de selecção de quais as *flags* a actualizar.

Com o sinal `Sign_Flags` como sinal de selecção é então possível criar um MUX de 2:1 que tem em cada entrada o que está descrito na seguinte tabela.

Tabela 5: Actualização de *flags* consoante a operação realizada.

Actualização	Nenhuma	Z, N	Z, N, C	Z, N, C, O
Sign_Flags	00	01	10	11
Flags [Z, N, C, O]	[O, O, O, O]	[NL, NL, O, O]	[NS, NS, NS, O]	[NA, NA, NA, NA]

Onde `O` representa um bit de *flags* não actualizado, e `NL,S,A` representa um novo *bit* actualizado retirado do sinal de entrada referente às operações lógicas, de *shift* ou aritméticas. Dependendo de `Sign_Flags` tem-se actualizações diferentes nas *flags*, podendo assim ter uma saída do MUX que será também a saída deste bloco de atualização das *flags*.

3.3.3 MEM

Relativamente às operações de memória é necessário tratar de *loads* e *stores*. Em ambos os casos o endereçamento à RAM é feito com o valor guardado no registo `A`, especificado pelos *bits* `3` a `5` da instrução. Para o caso de um *load* o valor que estiver nessa posição de memória é guardado no registo `WC`, especificado pelo *bits* `11` a `13` da instrução, estando o *write enable* da RAM a *low*. Para o caso de um *store* pretende-se escrever o conteúdo do registo `B`, especificado pelos *bits* `0` a `2` da instrução, na posição de memória anteriormente endereçada, sendo necessário colocar o *write enable* da RAM a *high*.

Uma vez que o conteúdo dos registos é de 16 *bits*, para endereçar a memória RAM recorre-se apenas ao 12 menos significativos. O sinal de *write enable* da RAM é, como já se viu, calculado no andar anterior, mas só neste terceiro andar é que é ligado à RAM. Optou-se por fazer desta maneira pois o cálculo desse sinal depende apenas de *bits* específicos da instrução, como se pode ver na Figura TAL, fazendo então parte do andar que trata de fazer o *decoding* da instrução.

É também neste andar que se liga a saída de dados da RAM ao sinal que depois será escrito no registo `WC` do banco de registos (para o caso do *load*) e liga-se também o valor que estiver no registo `B` à entrada de dados da RAM, para que depois possa ser escrito na posição de memória especificada (para o caso do *store*).

De referir que as leituras da RAM são feitas assincronamente e as escritas são feitas nos flancos positivos de relógio.

Na figura apresentada de seguida encontra-se o esquema de acesso à memória RAM.

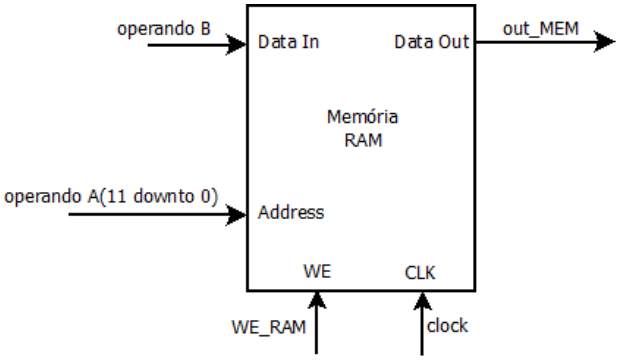

Figura 15: Representação da memória de dados, RAM.

Na tabela abaixo está a descrição do registo de saída deste andar.

Tabela 6: Caracterização do registo de saída do andar de *execute* e *memory access*.

Bits do registo de saída do andar EX e MEM	Sinal correspondente
67	instrução(6)
66	instrução(15)
65 downto 50	out_MEM
49	ALU vs MEM
48 downto 37	PC + 1
36	JUMP_MUXWB_OUT (JAL)
35 downto 33	ADD_RWC (instrução(13 downto 11))
32 downto 17	out_ALU
16 downto 1	out_mux_constantes
0	instrução(14)

3.4 Quarto Andar - WB

No último andar os diversos resultados possíveis são escritos no banco de registos - pode ser o resultado de uma operação da ALU, o resultado de uma operação sobre a memória (*load*), o carregamento de uma constante ou guardar em `R7` o valor do próximo *program counter*. Como se pode ver na figura seguinte, a seleção de qual os resultados deve ser escrito é feita com recurso a um MUX de 4:1.

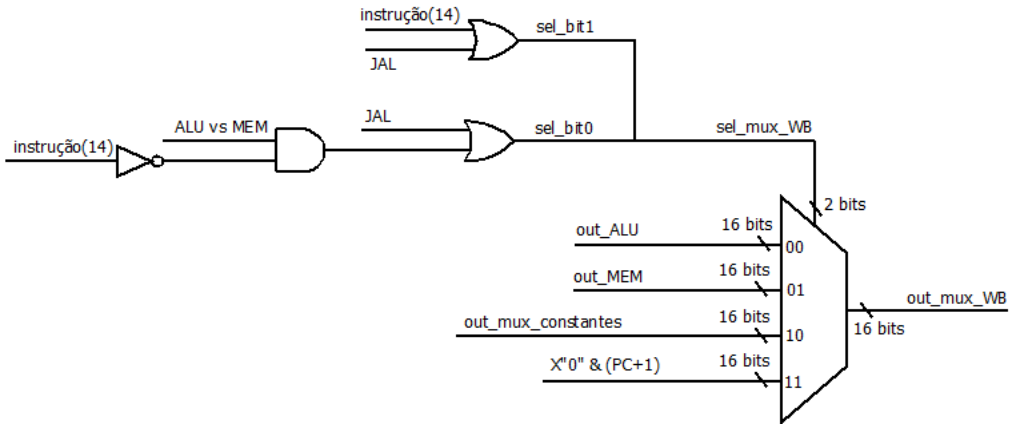

Figura 16: MUX para selecção do que vai ser escrito num dos registos do banco de registos.

Uma vez seleccionado o resultado a escrever é preciso escolher qual o registo onde se pretende escrever esse mesmo resultado, o registo `WC`.

Para instruções da ALU a escolha do registo onde se quer escrever o resultado final é feita com recurso aos *bits* `11` a `13` da instrução, assim como para operações de carregamento de constantes. Originalmente pensou-se em utilizar os 3 *bits* referidos anteriormente para controlar um MUX de 8:1 que colocasse a *high* um dos 8 *enables* (que estão armazenados nos 8 *bits* de um vector).

Relativamente à escrita no registo `R7` para quando se está numa operação de *jump and link*, verifica-se, com recurso a uma porta AND, quando é que o sinal de selecção do MUX de 4:1 está a `11`, ou seja, quando se vai escrever num dos registos o valor de `PC` + 1, e coloca-se o sinal de selecção do MUX de 8:1 com o valor `111`, ou seja, apenas o enable de `R7` fica a *high*.

Esta solução pode ser vista na figura abaixo.

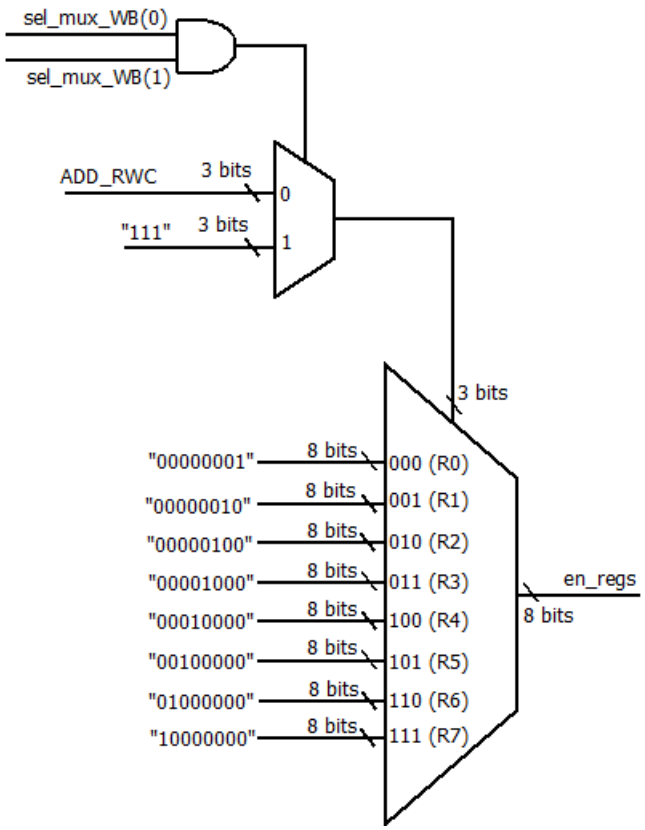

Figura 17: Ideia original para o MUX de selecção do sinal que controla os *enables* dos registos do banco de registos.

No entanto, a solução acima tem um problema - suponha-se o caso da instrução `1401` (HEX) que corresponde a um *jump if true* mediante a condição do resultado da ALU ser negativo. Os *bits* `11` a `13` da instrução são `010` e, como tal, o *enable* do registo `R2` ficaria activo. Porém, não se pretende escrever nesse registo. O mesmo decorre para uma operação de *store* na RAM e NOP.

Assim, para resolver o problema é necessário criar um sinal que faça *overwrite* ao *enable* que o MUX colocou a *high*, permitindo o sinal de *overwrite* colocar o *enable* a *low*, tal como pretendido, para que não se escreva em nenhum registo. De notar que este sinal corresponde àquele único que não é descodificado no andar de ID, decisão tomada para que os sinais que controlam a escrita no banco de registos possam estar no andar de WB, andar que corresponde de facto à escrita do resultado final.

O sinal de *overwrite* foi obtido com recurso à seguinte tabela.

A lógica que permite implementar o sinal é demonstrada na figura seguinte.

Como se pode ver, para o caso de operações da ALU, operações de *load*, carregamento de constantes e o caso de *jump and link*, o sinal de *overwrite* fica a *high*. Para o caso de *store* na memória, transferências de controlo que não *jump and link* e NOP, o sinal de *overwrite* fica a *low*, tal como pretendido. De notar também que a escrita nos registos é feita no flanco positivo do relógio.

Na figura abaixo encontra-se o esquema completo do andar de *write back*.

Tabela 7: Sinais que permitem obter o sinal de *overwrite* pretendido para cada operação.

	Sinais a utilizar para calcular sinal de *overwrite*					Sinal de *overwrite*
Operação	instrução(15)	instrução(14)	instrução(6)	ALU ou MEM?	JAL?	
ALU	1	0	X	0	0	1
MEM (*load*)	1	0	0	1	0	1
MEM (*store*)	1	0	1	1	0	0
Constantes	0	1	X	X	0	1
	1	1				
Controlo (não JAL)	0	0	X	X	0	0
Controlo (JAL)	0	0	X	X	1	1
NOP	0	0	0	X	0	0

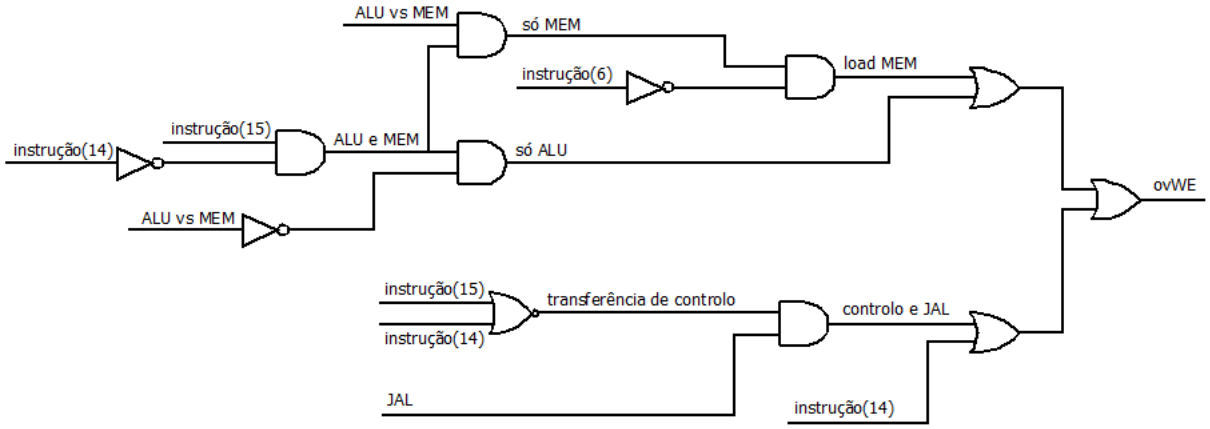

Figura 18: Lógica que permite calcular o sinal de *overwrite*.

4 Controlo do Processador

explicar ai os registos entre andares e maquina de estados

5 Conclusões

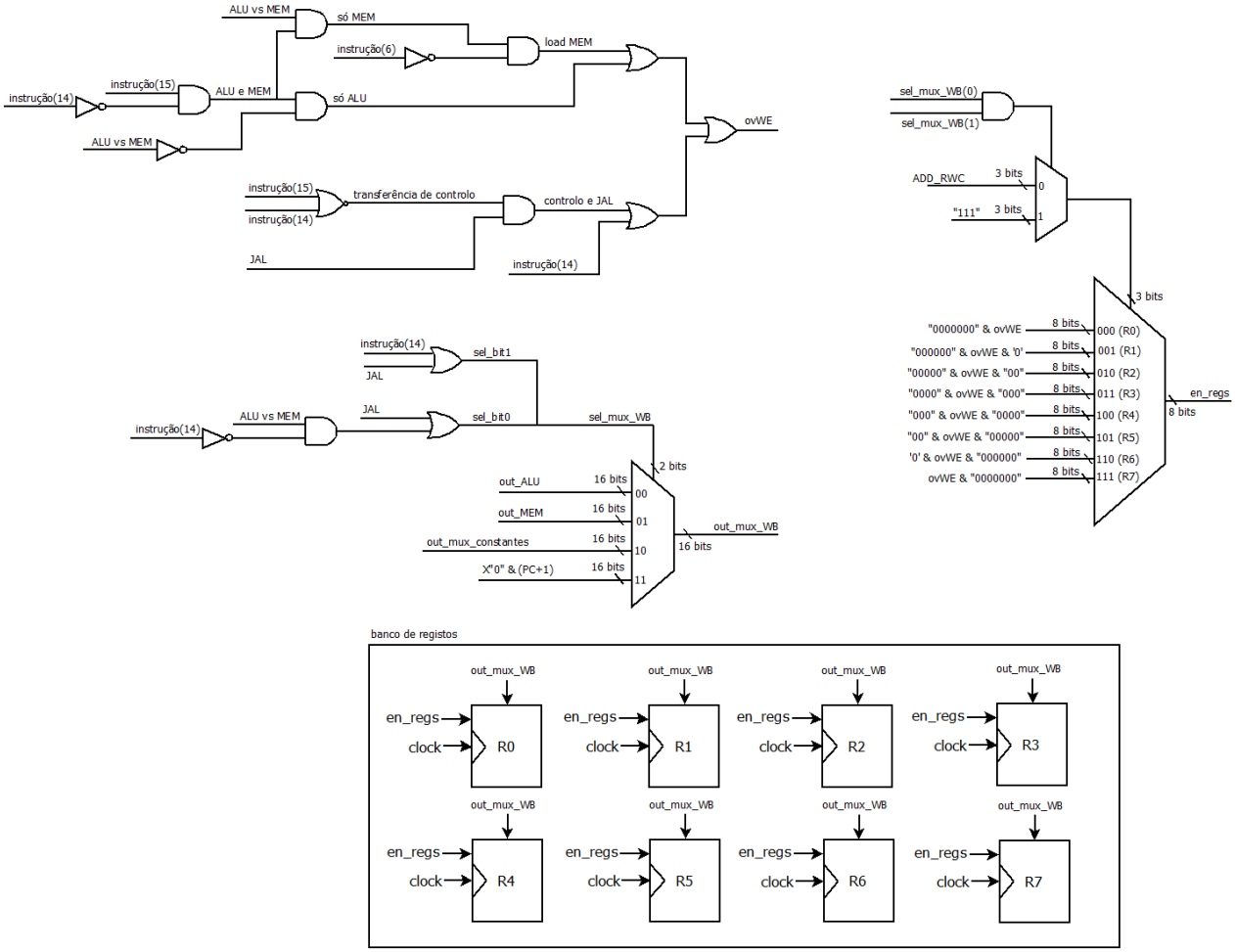

Figura 19: Esquema que representa o andar de WB e o banco de registos.

Todo list