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 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 projecto do processador é feito com recurso a uma linguagem de descrição de *hardware* - VHDL.

operaçoes a demorar 4 ciclos!

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).

fazemos em cada andar uma tabela a dizer que sinais passam para o andar seguinte?

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

3.2 Segundo Andar - ID e OF

3.2.1 ID

3.2.2 OF

referir se fazemos todo o decoding neste andar ou se passamos sinais e fazemos algum decoding depois

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.

FIGURA

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

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`).

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

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.

estes sinais serao explicados no ID ou explico

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

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

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

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:

TABELA

Ao observar o sinal `oper_ALU(2 downto 0)` 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.

FIGURA

Para cada um dos três sinais de saída (`out_ARI`, `out_SHIFT` e `out_LOG`) é criado um sinal que

ALU - falar das operações logicas e o mux de 16-¿8 entradas e a figura para as flags

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

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 1: 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]	[N$_L$, N$_L$, O, O]	[N$_S$, N$_S$, N$_S$, O]	[N$_A$, N$_A$, N$_A$, N$_A$]

Onde `O` representa um bit de *flags* não actualizado, e `N_L,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.

nao sei se é preciso explicar melhor o porquê

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

FIGURA

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

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 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

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.

Tabela 2: 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

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

FIGURA

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*.

FIGURA

4 Controlo do Processador

explicar ai os registos entre andares e maquina

5 Conclusões

Todo list