



INSTITUTO SUPERIOR TÉCNICO
MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE
COMPUTADORES

ARQUITECTURAS AVANÇADAS DE COMPUTADORES

Simulação de um processador μ Risc

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	1
2	Características do Processador	1
3	Estrutura do Processador	1
3.1	Primeiro Andar - IF	1
3.2	Segundo Andar - ID e OF	2
3.3	Terceiro Andar - EX e MEM	2
3.4	Quarto Andar - WB	3
4	Controlo do Processador	4
5	Conclusões	4

1 Introdução

Com este trabalho laboratorial pretende-se projectar um processador μ Risc, de 16 *bits* com arquitectura RISC. O processador possui 8 registos de uso geral e 42 instruções. O projecto do 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 μ Risc 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

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

`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*, seleccionando o sinal `destino_jump` no MUX_2, ou seja, o sinal `s_jump` está a *high*.

3.2 Segundo Andar - ID e 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.

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).

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 μ Risc 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.

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

se calhar e melhor explicar o significado destes sinais

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

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

estes sinais serao explicados no ID ou explico eu?

quem explica 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

nao sei se
é preciso
explicar
melhor o
porquê

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 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. Porém, para operações de transferência de controlo esses mesmos *bits* representam a operação a realizar, pelo que não basta utilizar os 3 *bits* referidos anteriormente para controlar um *decoder* que colocasse a *high* um dos 7 *enables* (que estão armazenados nos 7 *bits* de um vector), tal como pensado originalmente e como pode ser visto 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 *decoder* colocou a *high*, permitindo o sinal de *overwrite* colocar o *enable* a *low*, tal como pretendido,

para que não se escreva em nenhum registo.

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

TABELA

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

5 Conclusões

explicar ai
os regis-
tos entre
andares e
maquina
de estados

Todo list

fazemos em cada andar uma tabela a dizer que sinais passam para o andar seguinte?	1
se calhar e melhor explicar o significado destes sinais	2
referir se fazemos todo o decoding neste andar ou se passamos sinais e fazemos algum decoding depois	2
explicar WE da RAM, que eu depois uso quando explico a MEM. nao esta no desenho do decoder?	2
estes sinais serao explicados no ID ou explico eu?	2
quem explica a reciclagem do sinal do MUXA/MUXC - eu ou o teddy?	2
este sinal nao esta no decoder?	2
ALU - nao esquecer da simplificação de MUXs	2
nao sei se é preciso explicar melhor o porquê	3
explicar ai os registos entre andares e maquina de estados	4