Instituto Superior Técnico

Engenharia Eletrotécnica e de Computadores

Arquitecturas Avançadas de Computadores

2015/2016 2º Semestre

Desenvolvimento de um processador RISC

João Lages nº 75286

Guilherme Pires nº

Índice

[Introdução 3](#_Toc446282927)

[IF/ID Enable 3](#_Toc446282928)

[Andar ID 4](#_Toc446282929)

[ID/EX *enable* e andar EX 5](#_Toc446282930)

[EX/MEM *enable* e andar MEM 5](#_Toc446282931)

[MEM/WB *enable* e andar WB 6](#_Toc446282932)

[Conflitos de Dados e Alterações 7](#_Toc446282933)

[Conflitos de Controlo e Alterações 8](#_Toc446282934)

[Comparação de performance e conclusões 9](#_Toc446282935)

# Introdução

O objectivo deste trabalho era o desenvolvimento de um processador que suportasse o conjunto de Instruções de Arquitectura Microblaze. Para este efeito, foi-nos dado o código VHDL de um processador *5-stage multi-cycle* que suportava todas as operações deste conjunto de instruções, menos as operações relativas a *branch delay* (que teriam de ser implementadas mais tarde)*,* visto que não fazia sentido ter este tipo de operações funcional no *multi-cycle*.

Para além do que já foi referido, o propósito deste mini-projecto era também o de meter o processador a funcionar em *5-stage pipeline,* tendo para isso que resolver conflitos de dados, conflitos de controlo e fazer algumas alterações justificadas na arquitectura.

Depois da análise do código VHDL, conseguimos chegar a um desenho da arquitectura pipeline do *multi-cycle*. Explicaremos agora esta arquitectura com mais detalhe, por passos:

## IF/ID Enable

Como podemos ver pela Figura 1, este *Enable* apenas faz a transição de um sinal, sendo este sinal o PC, que indica a posição da instrução actual na memória, sendo usado para escolher a instrução do ciclo seguinte, depois de passar pelo BR Ctrl, como veremos mais à frente.

O outro sinal que vai “à volta” do registo *enable* é o I, relativo à instrução a carregar no decoder, carregada da memória (quadrado na figura). Este sinal foi aqui representado pois em várias arquitecturas este passa pelo *enable* também. Neste caso, como a memórica é síncrona, não é necessário pois esta variável já só fica activa depois de um ciclo de relógio ascendente, estando sincronizada com o PC.

Na passagem para o andar ID, o PC vem com a informação da posição da instrução do ciclo actual e o I vem com a informação da instrução actual (carregada antes pelo mesmo valor de PC ).

Figura 1 - IF/ID ENABLE

## Andar ID

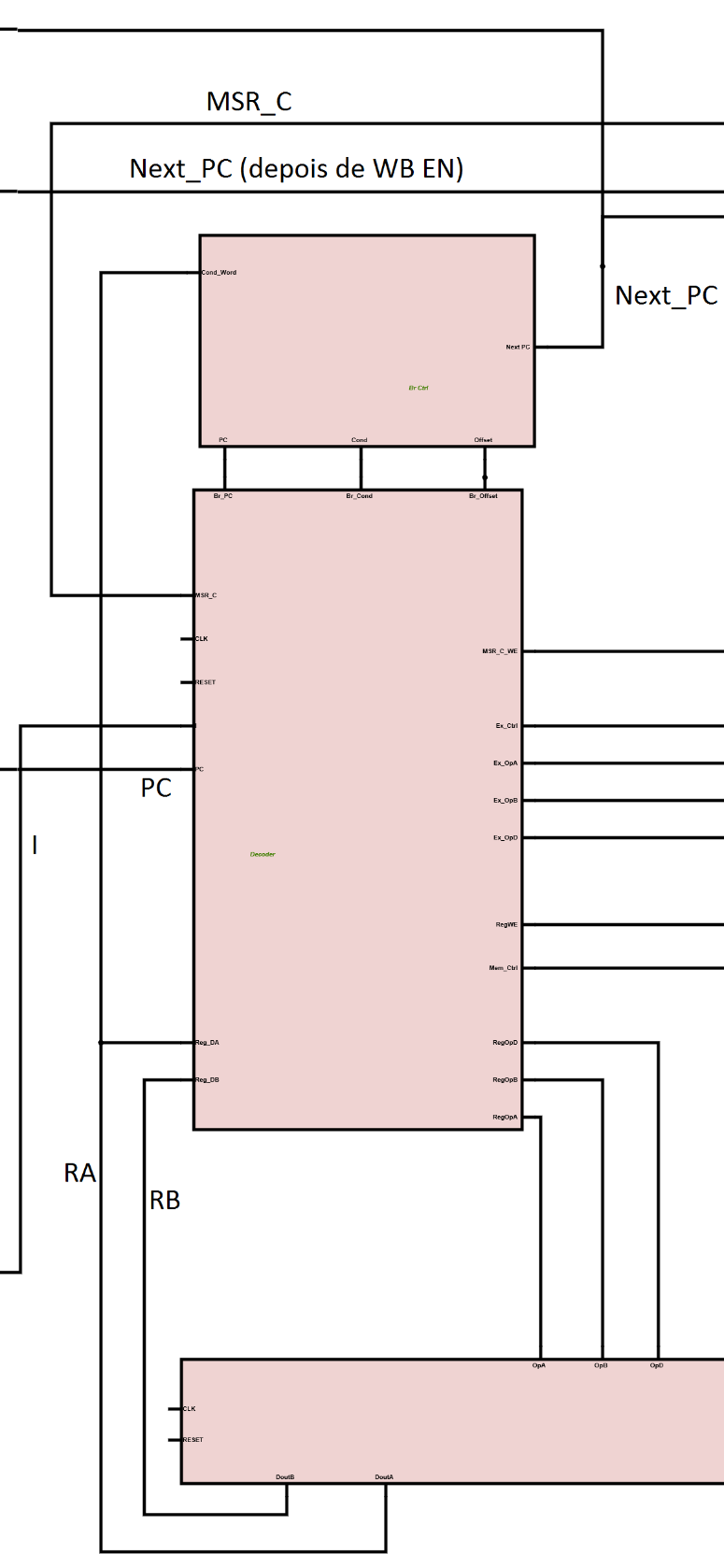


Figura 2 - Andar ID

Neste andar, o que acontece, muito resumidamente, é o I a ser descodificado no Decoder e a gerar todos os sinais que serão mais à frente utilizados para realizar a instrução. O decoder envia algum dados à unidade de Branch Control, para que esta realize o cálculo do PC seguinte (Next\_PC), que vai ter dois destinos distintos: vai à memória buscar o sinal I da próxima instrução (que vimos na transição IF/ID) e vai até ao MEM/WB *enable* para depois passar também pelo *enable* do IF/ID, como também vimos.

Esta passagem do Next\_PC por 2 enables era completamente desnecessária e impossível de realizar quando mudámos o circuito para *5-stage pipeline.* Apenas funcionava porque estávamos a trabalhar em *multi-cycle,* não fazendo qualquer diferença, uma vez que apenas um dos 4 *enables* de transição de estágios está activo.

É de notar os seguintes detalhes:

* O BR CTRL tem como entrada o RA também, que serve para verificar se este é maior, menor ou igual a zero, que é uma condição de salto;
* O decoder tem à entrada a flagC, vinda da unidade de ALU, que passa por um registo no meio (esta funcionalidade teve de ser modificada mais tarde);
* O decoder tem um sinal clock à sua entrada, que serve para a atribuição de sinais relacionados com o Registo Imm(*Immediate*);
* RA e RB são pedidos e lidos do *Register File* (RF) no mesmo ciclo de relógico, visto que funciona como uma memória de leitura assíncrona e escrita síncrona.

## ID/EX *enable* e andar EX

Os sinais saídos do decoder passam para o andar EX pelo *enable* ID/EX. Alguns destes sinais (os operandos e parte da instrução/Ctrl que decide qual a operação a fazer, *opcode*) passam pera ALU que faz todos os cálculos desta arquitectura.

O resultado (Res) é passado para o andar MEM, juntamente com os outros 3 sinais representados na Figura 3, que vão ser discutidos mais tarde.

A FlagC (*carry*) é depois passada para a entrada do decoder, depois de passar por um registo de estado. Esta funcionalidade foi posteriormente alterada, como já foi dito, devido aos conflitos de dados que podiam ocorrer pela espera desta sinal no andar ID.

## EX/MEM *enable* e andar MEM

O andar MEM trata dos acessos de leitura na memória síncrona da arquitectura, a mesma memória utilizada no andar IF para carregar a próxima instrução.

Figura - Andar EX

Este processador não tem escritas e leituras da memória na mesma instrução. Apesar disso, como a escrita tanto da memória como do *register file* ser sempre feito no andar WB, podem haver duas instruções seguidas que colidam com uma leitura e escrita da memória no mesmo ciclo de relógio, ao funcionar em *pipeline*. Este foi um dos conflitos de dados que mais tarde tivemos em conta.

No fundo, este andar recebe o Resultado do andar anterior e faz um acesso de leitura à memória. Se for uma operação de escrita, esta vai ser só efectuada no andar WB, que vai enviar um sinal à memória (WE\_B) com o endereço de escrita (DI\_B). O resultado da leitura da memória é apresentado à saída do andar MEM, como DO\_B.

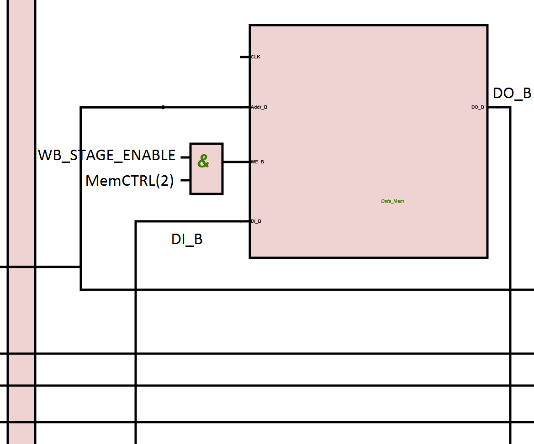
 É de acentuar o facto de os sinais de saída da memória não passam pelo próximo enable, tal como acontecia no andar IF, pois a memória é síncrona.

Figura - Andar MEM

O *enable* do WB e o bit mais significativo do MemCTRL decidem se haverá uma escrita na memória.

## 

## MEM/WB *enable* e andar WB

Neste andar, vamos finalmente utilizar os 3 sinais que saíram do andar IF e ainda não foram utilizados:

* O sinal RegWE, um bit que vai decidir se vai haver uma escrita no registo ou não. Esta escrita pode ser do resultado da ALU ou do acesso de leitura à memória do ciclo anterior, dada pelo sinal DinD (escolhido pelo MemCTRL e pelo resultado do andar andar EX, sendo alguns bits do Res também uma entrada de selecção). ;
* O sinal MemCTRL, 3 bits que decidem que tipo de leitura ou escrita vai ser efectuada na memória e no RF (*read/write byte, half-word and word*);
* O sinal DoutD, que contém o conteúdo de um registo, retirado logo no andar ID.

O andar WB, tal como o nome indica, trata de todas as escritas em registos, seja este o *Register File* ou a memória. A caixa de cima da Figura 5 selecciona o que escrever no RF e a de baixo o que escrever na memória. Quando não há escritas os WE’s vêm a zero, sendo irrelevante a saída de cada uma das caixas anteriores.

O sinal DI\_B, que contém o conteúdo do que vai ser escrito na memória, tanto pode tomar o valor (ou parte do valor) do DoutD ou do DO\_B, sendo o resto das entradas, entradas de selecção.

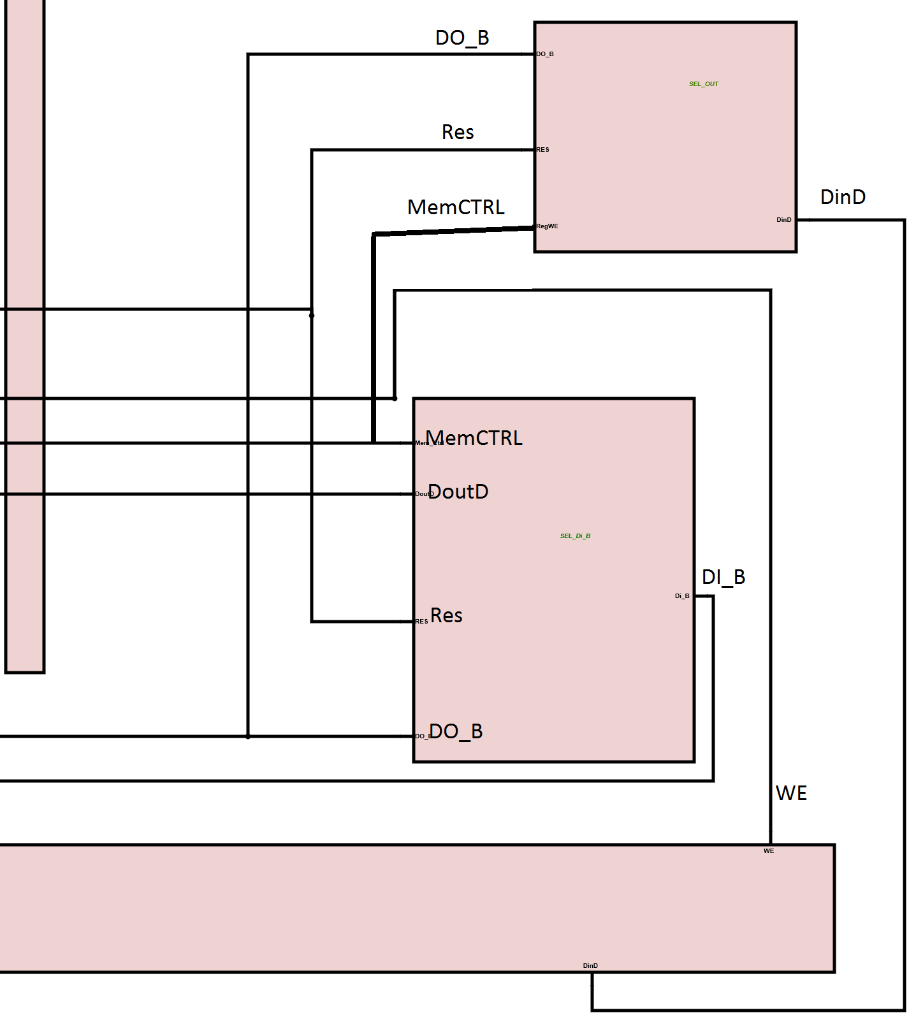
No entanto, o endereço de escrita é dado pelo Resultado da ALU, no andar MEM. Apesar de não ocorrerem problemas no funcionamento *multi-cycle,* ao funcionar em pipeline, este sinal já tem um endereço de escrita na memória diferente quando a escrita ocorrer no andar WB. Este conflito de dados teve de ser resolvido posteriormente.

Figura - Andar WB

# Conflitos de Dados e Alterações

# Conflitos de Controlo e Alterações

Nesta arquitectura, como a unidade de *Branch Control* está no andar ID, a resolução dos conflitos de controlo acabou por ser relativamente simples, uma vez que o endereço do resultado do *Branch* é logo actualizado no ciclo de relógio a seguir. Isto porque, o ID\_Next\_PC foi alterado, de modo a ir do andar ID para o IF directamente, sem passar pelos registos *enable,* como acontecia anteriormente (Figura 6).

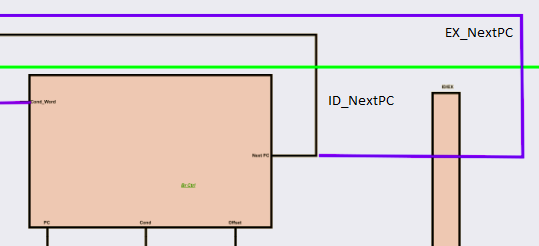


Figura 6 - Unidade BR CTRL

Desta forma, a unidade da Figura 6 calculava sempre os endereços correctos do PC da seguinte instrução , excepto quando era uma operação de *Branch Delayed*. Neste caso, o endereço de salto (se o branch for *taken*) só deve ser tomado 2 instruções a seguir, isto é, deixar a instrução a seguir ao *BR delayed* ser executada e só depois alterar o PC.

Para resolver este conflito, o ID\_NextPC foi propagado para o andar EX (EX\_NextPC), de modo a guardarmos o endereço de salto e uma lógica (espécie de *MUX*) foi adicionado no andar IF, para seleccionarmos o PC correcto da próxima instrução.

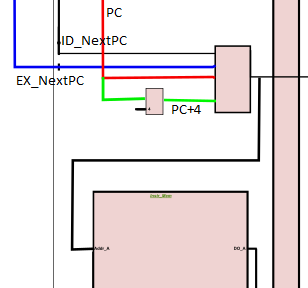
Na Figura 7, vê-se parte da implementação desta lógica que, com base no opcode da instrução no andar ID e EX, escolhe o PC pretendido:

Figura 7 - Seleccionador do PC

* EX\_NextPC, para escolher o endereço de salto do *BR delayed*;
* ID\_NextPC, para escolher o PC em todas as operações, excepto no caso de ocorrer *BR delayed,* visto que a saída do *BR ctrl* não tem o PC seguinte;
* PC+4, para executar a instrução a seguir ao *BR delayed*;
* PC, no caso de ocorrer um stall mais à frente (do andar ID para o andar EX).

# Comparação de performance e conclusões

No início do projecto, o processador que nos foi dado funcionava em *multicycle* (1 instrução em cada 5 ciclos de relógio) com um periodo de 9.904ns (Tmulticycle), em que o caminho crítico se situava no andar EX, passando pelo multiplicador.

Depois de metermos o processador a funcionar em *pipeline*, o periodo de relógio aumentou para 37.448ns (Tpipeline, o que corresponde a uma frequência de aproximadamente 27MHz). No entanto, há que ter em conta que o processador anterior demora 5 ciclos a executar uma instrução (5\*9.904ns), fazendo com que o processador melhorado seja mais rápido.

Para fazer uma análise mais em concreto do *speed*-*up* alcançado, calculámos o CPI (*Cycles per Instruction*) do novo processador, usando para isso um dos programas disponibilizados na memória. Isto porque, operações de Store seguidas de Load não são suportadas nesta arquitectura e têm um Stall pelo meio, daí o número de ciclos não ser igual ao número de instruções (sem contar com o número de ciclos que temos de esperar quando o pipeline é inicializado, claro).

Como se pode constatar, o novo processador é apenas aproximadamente 32.2% mais rápido que o *multicycle*. Apesar de as alterações terem sido implementadas sem ter em atenção o aumento de performance do processador, existe uma razão principal para o speed-up ter sido tão baixo: deve-se ao caminho crítico do processador pipeline. Este caminho, está assinalado na Figura 8, que passa pela unidade ALU (multiplicador) e vai até ao *BR Control* que depois selecciona o PC da instrução seguinte.

Apesar de os conflitos de *Branch* serem resolvidos no andar ID, que nos facilita muito o trabalho pois a instrução seguinte vem sempre correcta, o caminho crítico fica excessivamente grande, limitando muito a frequência de relógio.

Deste modo, no próximo trabalho de laboratório, a nossa primeira prioridade será mudar a unidade de *BR Control* de andar e resolver os conflitos de outra maneira (com *branch prediction*).

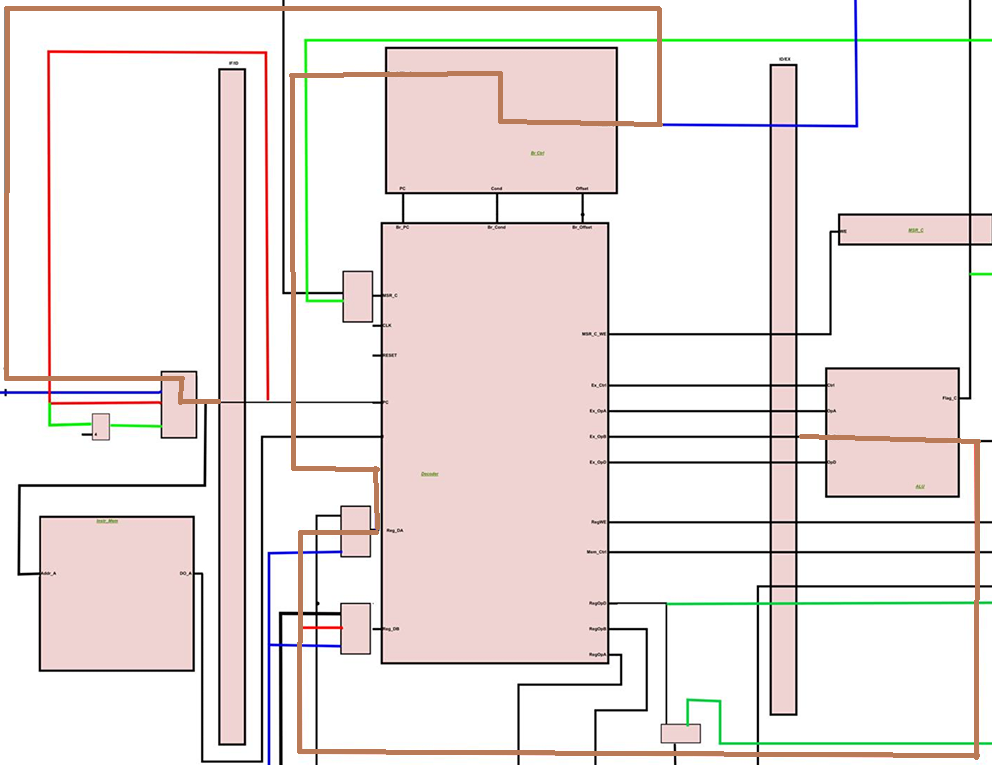


Figura - Caminho crítico (assinalado a castanho)