ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Graduação em Engenharia Computação

PCS3612 - Organização e Arquitetura de Computadores I

Professora Cintia Borges

Etapa 1 - Planejamento do Projeto ARM Pipeline

Arthur Pires da Fonseca NUSP: 10773096
Guilherme Elias Setter Bauab NUSP: 9900383
Sergio Ariel Gonzales Fuentes NUSP: 10770200

Introdução

Este é o planejamento para a implementação do processador ARMv4 na versão pipeline.

O projeto será desenvolvido em VHDL 2008, suas versões serão controladas em um repositório do GitHub e os testes serão feitos na máquina virtual disponibilizada pela disciplina, visto que é o mesmo ambiente onde será feita a correção desta descrição VHDL.

1.1 a) Entendimento

Para este projeto, o trio optou por basear-se no circuito de referência (*arm_single.vhd*), que é uma implementação de processador monociclo baseado em uma subparte da ISA da arquitetura ARM. Esta seção descreve o funcionamento desse circuito, suas unidades principais e como se comunicam.

O projeto monociclo é composto por 16 entidades: *top, dmem, imem, arm, decoder, condlogic, condcheck, regfile, adder, extend, flopenr, flopr, mux2, alu, datapath, controller*.

A camada de abstração de mais alto nível definida pelo livro de referência divide o processador em basicamente 3 partes: unidade de controle, fluxo de dados e memórias (de instrução e de dados).

Unidade de Controle: A entidade controller é composta por um *decoder* e um *condlogic*. O *decoder* é responsável por determinar o tipo de instrução lida, o tipo de operação a ser realizada pela ULA e se a próxima instrução é sequencial ou um *branch*. Para isso, lê os sinais *OpCode*, *Funct* e *Rd* da instrução e elabora os sinais de controle *RegSrc*, *ImmSrc*, *AluSrc*, *MemtoReg* e *AluControl* que serão enviados para o fluxo de dados.

O *condlogic* determina se a instrução deve ser executada pelo campo condicional e pelo valor da *flag* respectiva. Dessa forma, fica responsável por determinar os sinais *PCSrc*, *MemWrite* e *RegWrite*.

Fluxo de Dados: O *datapath* é formado pelas entidades *alu*, *regfile*, *adder*, *flopr*, *extend*, e *mux2*. A *alu* é a unidade lógico-aritmética, que realiza as operações de soma, subtração, AND e ORR, baseado no sinal de controle *ALUControl* (vindo da unidade de controle). Possui duas saídas: o resultado da operação e o sinal *ALUFlags*, correspondente às flags de controle enviadas à unidade de controle. Possui duas entradas de operandos, um deles corresponde a um dos registradores do banco e o outro pode ser outro registrador ou um imediato na instrução, determinado por um MUX na entrada com sinal de controle *ALUSrc*. O operando imediato é estendido para o tamanho de 32 bits através do componente Extend, que faz um *padding* de zeros à esquerda, de forma que o tamanho do padding é determinado pelo sinal de controle *ImmSrc*.

O componente *regfile* corresponde ao banco de registradores, composto de 16 registradores de 32 bits, sendo um deles o PC, que frequentemente é representado fora do componente para facilitar a visualização. Possui como *inputs*:

- Dois sinais de 4 bits correspondentes a dois registradores a serem lidos, um desses sinais correspondente ao registrador onde será realizada a escrita
- Um sinal de 32 bits correspondente ao conteúdo da escrita a ser realizada

- Uma entrada R15 referente ao registrador PC
- Uma entrada de controle *RegWrite* correspondente ao enable de escrita.

A entidade *mux2* é utilizada para selecionar qual sinal será utilizado em determinado momento. Ela foi instanciada de 4 formas diferentes:

- Escolhendo se a próxima instrução do PC vai ser (PC + 4), ou seja, a próxima da sequência, ou um valor de branch determinado pela instrução.
- Selecionando as entradas de leitura do RegFile, um deles com o código estático 15 (referente ao valor do registrador de PC) e o outro determinando o trecho da instrução correspondente ao endereço do registrador, que varia entre instruções de diferentes tipos.
- Na entrada da *ALU*, que determina se o segundo operando é um registrador ou imediato.
- Determinando, ao fim da execução do *pipeline*, se o valor a ser escrito no registrador é o resultado da *ALU* ou o valor lido da memória de dados.

O último componente é o *flopr*, que funciona como o *Program Counter* determinando a posição na memória da instrução executada em determinado ciclo.

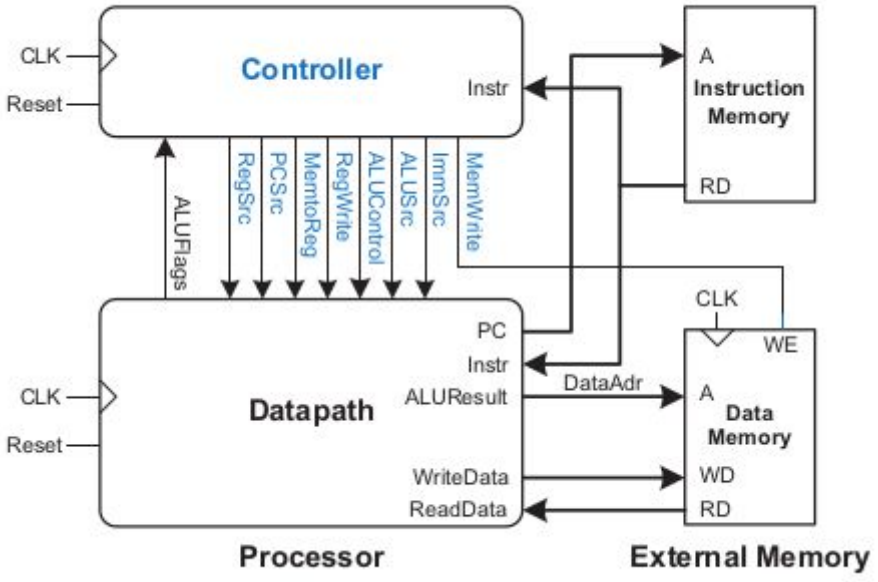

Figura 1 - Divisão principal do processador (página 443 do livro)

Por último, deve-se observar que a arquitetura ARM utiliza palavras de 32 bits. As memórias de instrução e dados são endereçados em *bytes* (8 bits), portanto os incrementos de posição são realizados somando 4 ao valor atual de PC, de forma a incrementar uma posição de 4 x 8 = 32 bits.

1.1 b) Demonstração

O código do processador monociclo foi compilado e executado usando-se o *ModelSim* instalado na máquina virtual disponibilizada e no computador de um dos membros do grupo. Os resultados são mostrados a seguir.

```
arthur@arthurpfonseca:~/intelFPGA_lite/20.1/modelsim_ase/linuxaloem/ARMv4_pipeline$ ../vcom -check
_synthesis -2008 arm_single.vhd
Model Technology ModelSim - Intel FPGA Edition vcom 2020.1 Compiler 2020.02 Feb 28 2020
Start time: 19:22:11 on Nov 21,2020
vcom -check_synthesis -2008 arm_single.vhd
-- Loading package STANDARD
-- Loading package TEXTIO
-- Loading package std_logic_1164
-- Loading package NUMERIC_STD_UNSIGNED
-- Compiling entity testbench
-- Compiling architecture test of testbench
** Warning: arm_single.vhd(52): (vcom-1400) Synthesis Warning: Signal "MemWrite" is read in the pr
ocess but is not in the sensitivity list.
** Warning: arm_single.vhd(53): (vcom-1400) Synthesis Warning: Signal "DataAdr" is read in the pro
cess but is not in the sensitivity list.
** Warning: arm_single.vhd(54): (vcom-1400) Synthesis Warning: Signal "WriteData" is read in the p
rocess but is not in the sensitivity list.
** Warning: arm_single.vhd(56): (vcom-1400) Synthesis Warning: Signal "DataAdr" is read in the pro
cess but is not in the sensitivity list.
-- Compiling entity top
-- Compiling architecture test of top
-- Compiling entity dmem
-- Compiling architecture behave of dmem
-- Compiling entity imem
-- Compiling architecture behave of imem
-- Compiling entity arm
-- Compiling architecture struct of arm
-- Compiling entity controller
-- Compiling architecture struct of controller
-- Compiling entity decoder
-- Compiling architecture behave of decoder
-- Compiling entity condlogic
-- Compiling architecture behave of condlogic
-- Compiling entity condcheck
-- Compiling architecture behave of condcheck
-- Compiling entity datapath
-- Compiling architecture struct of datapath
-- Compiling entity regfile
-- Compiling architecture behave of regfile
-- Compiling entity adder
-- Compiling architecture behave of adder
-- Compiling entity extend
-- Compiling architecture behave of extend
-- Compiling entity flopenr
-- Compiling architecture asynchronous of flopenr
-- Compiling entity flopr
-- Compiling architecture asynchronous of flopr
-- Compiling entity mux2
-- Compiling architecture behave of mux2
-- Compiling entity alu
-- Compiling architecture behave of alu
End time: 19:22:12 on Nov 21,2020, Elapsed time: 0:00:01
Errors: 0, Warnings: 4
```

Figura 2 - Análise e elaboração do circuito *arm_single.vhd*

```
arthur@arthurpfonseca:~/intelFPGA_lite/20.1/modelsim_ase/linuxaloem/ARMv4_pipeline$ ../vsim -c -do
 "run -all" testbench
Reading pref.tcl

# 2020.1

# vsim -c -do "run -all" testbench
# Start time: 19:22:26 on Nov 21,2020
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading ieee.numeric_std_unsigned(body)
# Loading work.testbench(test)
# Loading work.top(test)
# Loading work.arm(struct)
# Loading work.controller(struct)
# Loading work.decoder(behave)
# Loading work.condlogic(behave)
# Loading work.flopenr(asynchronous)
# Loading work.condcheck(behave)
# Loading work.datapath(struct)
# Loading work.mux2(behave)
# Loading work.flopr(asynchronous)
# Loading work.adder(behave)
# Loading work.regfile(behave)
# Loading work.extend(behave)
# Loading work.alu(behave)
# Loading work.imem(behave)
# Loading work.dmem(behave)
# run -all
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 0  Instance: /testbench/dut/i_dmem
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
```

```
#    Time: 0 ps  Iteration: 0  Instance: /testbench/dut/i_dmem
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 0  Instance: /testbench/dut/i_imem
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 0 ps  Iteration: 0  Instance: /testbench/dut/i_arm/dp/i_alu
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 0  Instance: /testbench/dut/i_arm/dp/rf
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 0  Instance: /testbench/dut/i_arm/dp/rf
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 0  Instance: /testbench/dut/i_arm/dp/rf
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 0  Instance: /testbench/dut/i_arm/dp/rf
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 1  Instance: /testbench/dut/i_dmem
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 0 ps  Iteration: 1  Instance: /testbench/dut/i_arm/dp/i_alu
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 1  Instance: /testbench/dut/i_arm/dp/rf
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 1  Instance: /testbench/dut/i_arm/dp/rf
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 1  Instance: /testbench/dut/i_arm/dp/rf
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 1  Instance: /testbench/dut/i_arm/dp/rf
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ps  Iteration: 2  Instance: /testbench/dut/i_dmem
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 0 ps  Iteration: 2  Instance: /testbench/dut/i_arm/dp/i_alu
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 30 ns  Iteration: 9  Instance: /testbench/dut/i_dmem
```

```
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 30 ns  Iteration: 9  Instance: /testbench/dut/i_arm/dp/i_alu
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 100 ns  Iteration: 9  Instance: /testbench/dut/i_dmem
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 100 ns  Iteration: 9  Instance: /testbench/dut/i_arm/dp/i_alu
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 140 ns  Iteration: 9  Instance: /testbench/dut/i_dmem
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 140 ns  Iteration: 9  Instance: /testbench/dut/i_arm/dp/i_alu
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 190 ns  Iteration: 9  Instance: /testbench/dut/i_dmem
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 190 ns  Iteration: 9  Instance: /testbench/dut/i_arm/dp/i_alu
# ** Failure: NO ERRORS: Simulation succeeded
#    Time: 205 ns  Iteration: 1  Process: /testbench/line__51 File: arm_single.vhd
# Break in Process line__51 at arm_single.vhd line 55
# Stopped at arm_single.vhd line 55
VSIM 2>
```

Figura 3 - Execução do *testbench* presente no código (sucesso na simulação, conforme anunciado pela 4ª linha de baixo para cima)

1.2 Projeto

Esta seção descreve brevemente a versão *pipeline* do processador ARM e descreve as modificações e adições que serão realizadas no código de base do processador ARMv4 monociclo para transformá-lo em um processador ARM *pipeline*.

O *pipeline* divide a computação feita pelo processador ciclo único em 5 etapas que podem ser executadas simultaneamente por instruções diferentes. Dessa forma, a latência de cada instrução idealmente não muda, porém o fluxo total de instruções aumenta significativamente, melhorando a performance geral do processador.

Os cinco estágios são:

1. *Fetch* - Ler instrução da *instruction memory*
2. *Decode* - Ler os operandos do *Register FIle* e decodificar a instrução para produzir os sinais de controle
3. *Execute* - Performar a computação da ULA
4. *Memory* - Ler ou escrever dados na memória de dados (RAM)
5. *Writeback* - Escrever o resultado no registrador (quando aplicável)

No processador monociclo, cada instrução é executada individualmente até completar todas essas etapas, apenas ao fim de todas a próxima instrução se inicia.

No processador *pipeline* a próxima instrução se inicia um ciclo após a anterior ter começado, de forma que todos os estágios do processamento podem executar uma instrução diferente ao mesmo tempo, como evidenciado pela imagem abaixo.

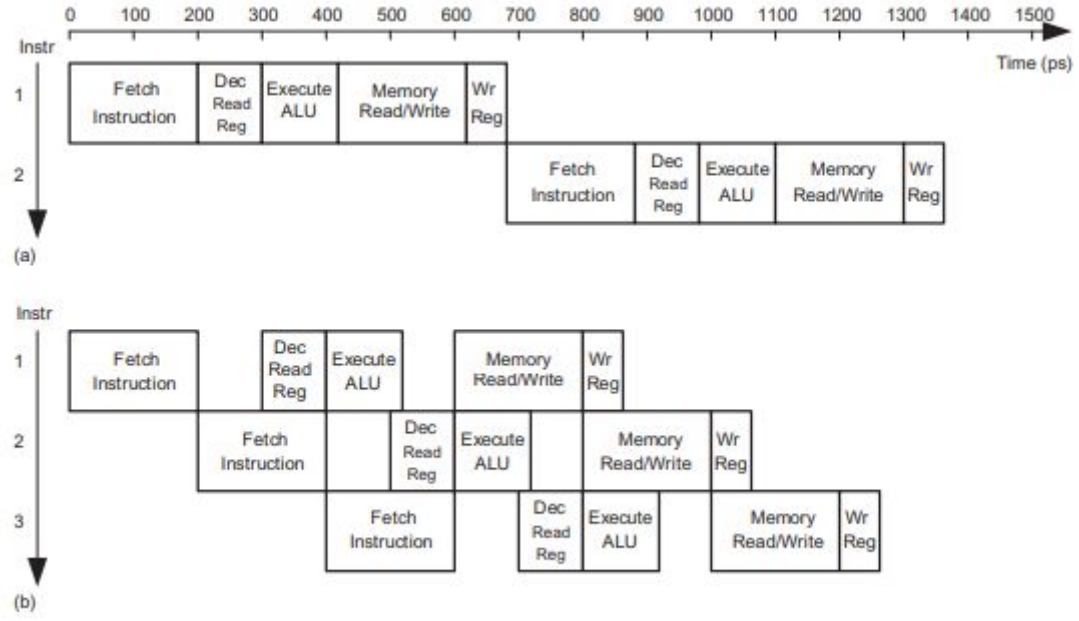

Figura 4 - Distribuição temporal das instruções na versão monocíclica e com *pipeline* (página 426 do livro)

Iremos partir do circuito disponibilizado pelo livro, descrito brevemente na seção anterior deste documento.

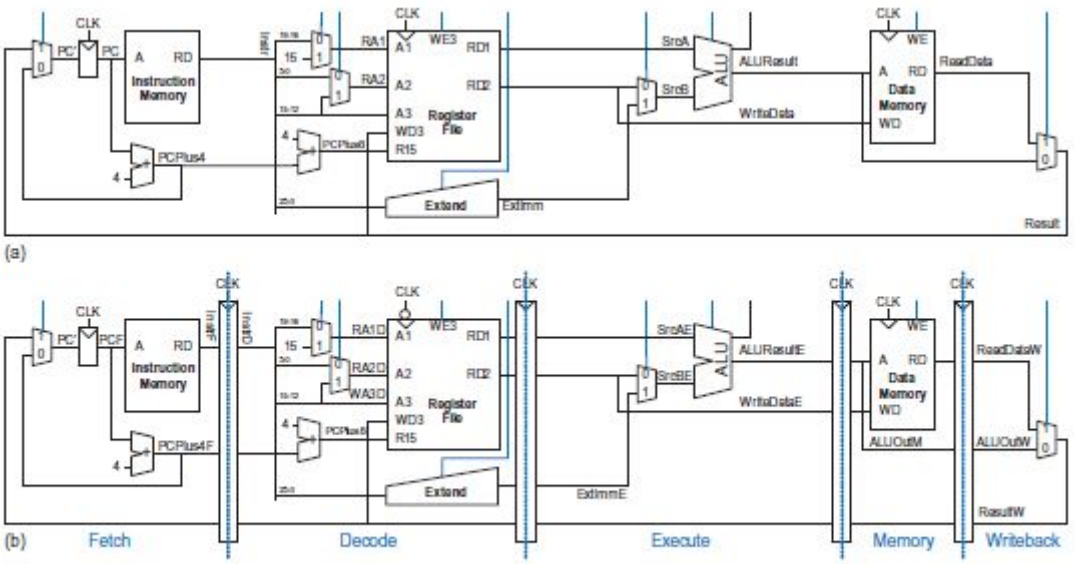

Figura 5 - Fluxo dados da versão monocíclica em comparação com uma primeira adaptação para a versão *pipeline* do processador (página 428 do livro)

A implementação do *pipeline* consiste em dois conjuntos de modificações: alterações no fluxo de dados e implementação da *Hazard Unit*, detalhadas abaixo.

A unidade de controle continua a mesma já que não há alteração nos sinais de controle, a única diferença é que esses sinais de controle passam pelas divisórias do *pipeline* junto com os outros sinais (resultados parciais) do fluxo de dados, de forma a permanecer sincronizados com o resto da instrução.

<u>Fluxo de Dados:</u>

1. A primeira modificação é separar os cinco estágios de processamento por 4 conjuntos de registradores de *pipeline*, que também serão chamados de “registradores divisores” ou “divisórias”. A função desses registradores é armazenar os resultados parciais de um estágio para quando for necessário inserir uma bolha no *pipeline*, ou seja, quando instruções tiverem que esperar 1 ou 2 ciclos para continuarem sua execução, como forma de resolução de conflitos de dados ou de controle.

   É importante garantir que o sinal de endereço de escrita WA3 (*write address 3*) acompanhe a execução da instrução correspondente até o final do *pipeline*, pois a divisória ID e EX descartará esses bits assim que a instrução seguinte for decodificada, então o endereço dos registradores deve ser repassado juntamente com o processamento da instrução para que essa informação persista e para que esse sinal não seja incorretamente definido por outro comando que esteja no estágio de decodificação (como aconteceria na versão *pipeline* definida na figura acima).

2. A segunda modificação é uma otimização sugerida pelo livro para economizar um registrador de *pipeline* e um somador de 32 bits: ligar a saída do primeiro somador no estágio de *fetch* diretamente com a entrada R15 do banco de registradores. Isto é

possível devido ao fato de que o valor que deve estar armazenado no registrador R15 no estágio ID de uma certa instrução é, quando não há um *branch* no *pipeline*, logicamente equivalente ao valor que será armazenado no registrador PC naquele mesmo ciclo (PC + 8 = (PC da instrução seguinte) + 4).

Nas palavras do livro, o valor de PC já terá sido incrementado duas vezes de 4, por isso o segundo somador de 32 bits pode ser retirado e o valor de (PC + 4) do estágio de *fetch* pode ser ligado diretamente à entrada R15 do banco de registradores.

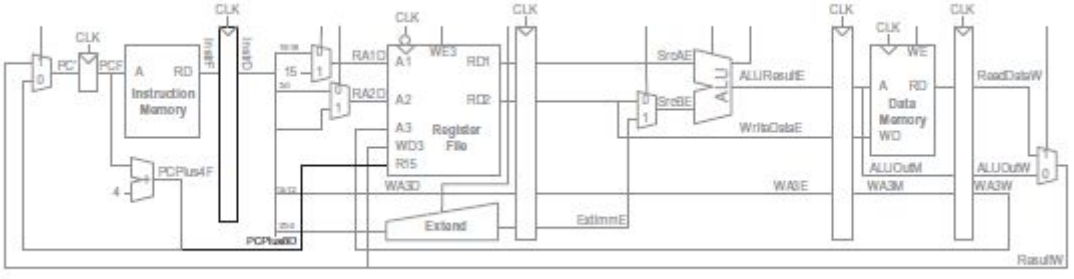

Figura 6 - Otimização sugerida pelo livro, já com o sinal WA3D corrigido, ou seja, sendo repassado ao longo do *pipeline* junto da instrução a que ele corresponde (página 430 do livro)

***Hazard Unit*:**

Quando existe dependência entre instruções, podem ocorrer dois tipos de *hazard*: de dados e de controle. O *hazard* de dados ocorre quando uma instrução tenta ler um registrador que ainda não foi escrito por uma instrução anterior. O *hazard* de controle ocorre quando a decisão de qual será a próxima instrução a ser buscada no estágio IF ainda não foi tomada pela instrução de salto em execução.

De forma a evitar alguns *hazards* de dados, os registradores são escritos na primeira metade do ciclo (borda de descida do *clock*) e lidos na segunda metade (borda de subida), de forma que é possível realizar as duas operações (escrita e leitura) no mesmo ciclo, mas ainda não evitando os conflitos do tipo RAW (*read after write*).

1. ***Forwarding*:**

   Uma forma alternativa à inserção de bolhas no pipeline quando existe conflito de dados é o uso de encaminhamento (*forwarding*). Seu funcionamento consiste em encaminhar um resultado do estágio de Memória ou *Writeback* para uma instrução no de execução. Para isso, adicionamos dois MUX na frente da ULA para selecionar os operandos a partir de 3 origens possíveis: valor armazenado em um registrador (o definido na instrução como o de origem para a operação), estágio de memória ou estágio de *writeback*. Esses dois MUX são controlados pelos sinais de controle ForwardAE e ForwardBE gerados pela *Hazard Unit*.

   É importante observar que pode acontecer de um dos operandos da instrução depender de um registrador que será escrito por ambas as instruções que estão em MEM e WB. Neste caso, o resultado encaminhado deverá ser,

obviamente, o da instrução executada mais recentemente, que é a que está em MEM.

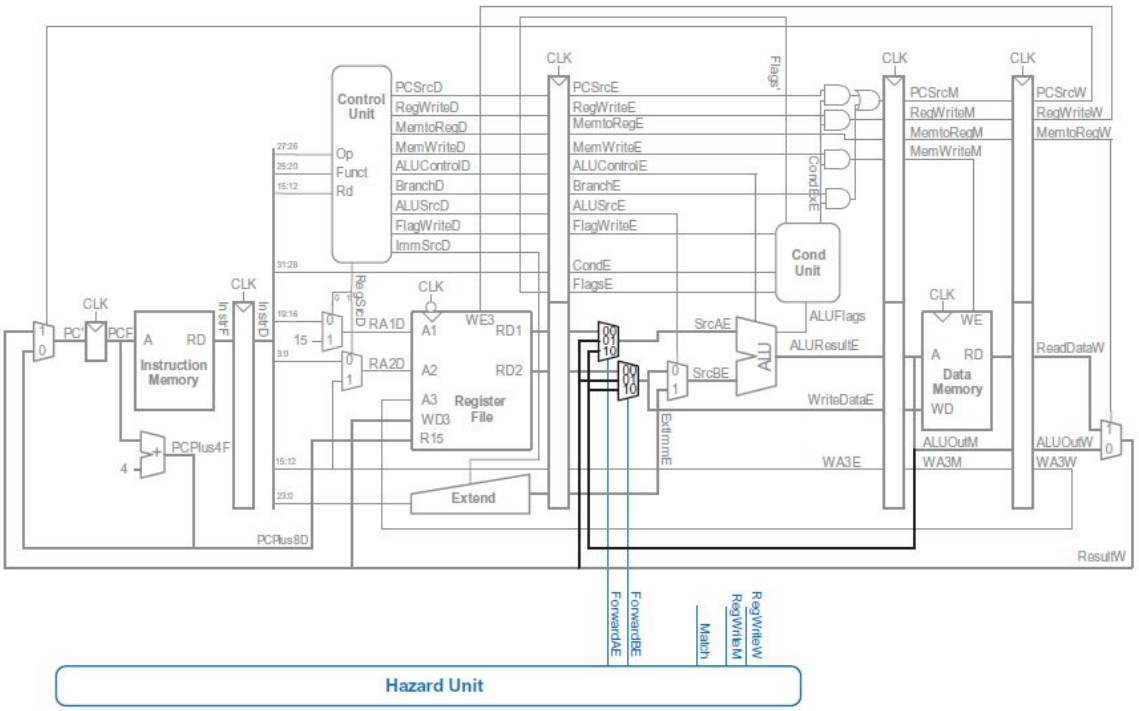

Figura 7 - Implementação de *Forwarding*

2. ***Stalls*:**

Alguns *hazards* não podem ser resolvidos apenas com *forwarding*. Por isso, são necessárias outras técnicas para mitigar o problema.

Podem-se utilizar *stalls* por exemplo, conhecidos também como inserção de bolhas. Isso pode ser usado nos casos em que o registrador de destino do estágio MEM é o mesmo que um registrador de origem do estágio ID ou ainda quando em EX há uma instrução LDR (um *load* da memória).

A instrução LDR, por exemplo, é denominada como de latência de dois ciclos de *clock*, pois o resultado da operação só fica pronto no estágio WB. Para uma instrução subsequente que dependa do valor armazenado no registrador de destino do LDR ter a garantia de que obterá o valor correto do banco de registradores, é necessário a adição de um *stall*. O *stall* consiste em fazer com que os registradores de *pipeline* do estágio atual e anteriores mantenham seus valores armazenados, assim como o valor do PC, que não deve mudar numa situação como essa.

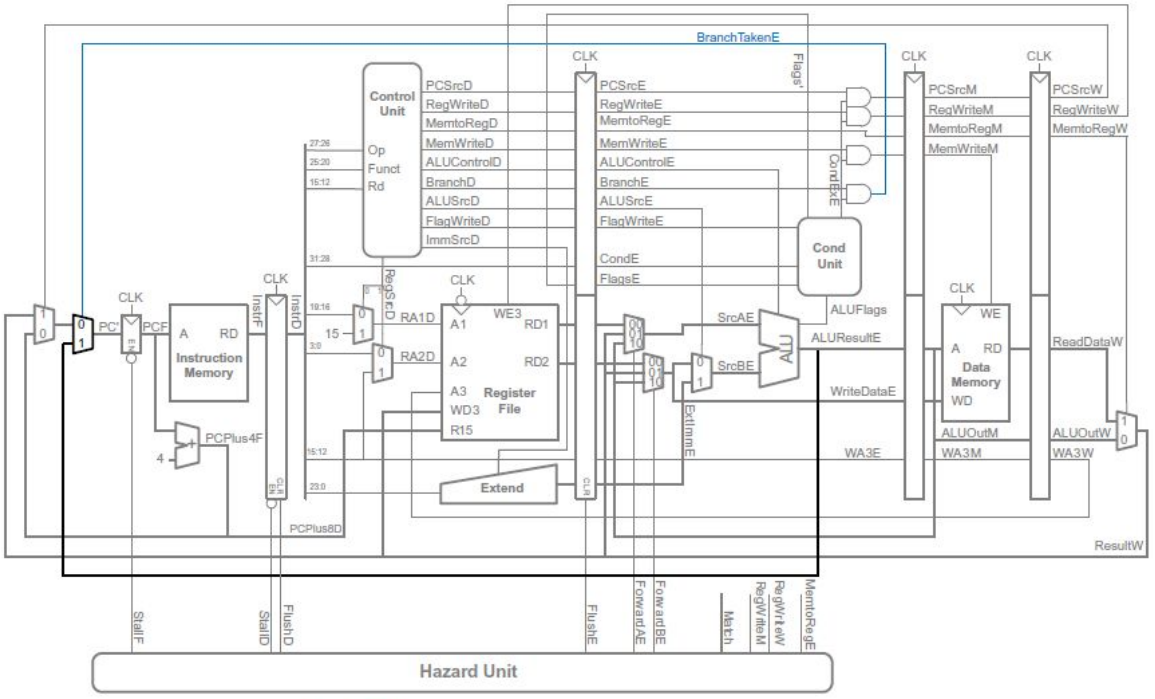

Figura 8 - Implementação dos *enables* e *reset* dos registradores divisores e MUX do PC para *hazard* de controle (página 439 do livro)

3. ***Branches:***

A unidade de controle do processador *pipeline* usará o mecanismo de previsão *nottaken* para a execução das instruções, ou seja, será sempre suposto que não ocorrerá um salto quando uma instrução de *branch* adentrar o *pipeline.*

Essa abordagem falhará quando o programa armazenado na memória possuir saltos que devem ser tomados. Quando for esse o caso, deve-se fazer um *flush* em todas as instruções que tinham começado a ser executadas depois do *branch*.

Isso será implementado com um MUX antes do registrador PC, optando por transferir o controle para a próxima entrada regular (instrução presente em PC + 4) ou para a entrada calculada no estágio WB.

Quando é identificado que o salto deve ocorrer, também é necessário fazer um *flush* no registrador divisor do estágio seguinte para garantir que não se escreva indevidamente na memória.

Para que tudo isso seja possível, precisa-se adicionar *enables* no registrador de PC e na divisória entre ID e IF e um *reset* no registrador de execução; todos sob comando da *Hazard Unit*.

4. ***Hazard Unit***

A unidade de identificação e resolução de *hazard* será criada de forma a gerar os sinais de controle que definirão o comportamento descrito nos itens anteriores.

Com todas essas modificações e adições temos o processador ARM *pipeline* completo, ilustrado pelo diagrama fornecido pelo livro:

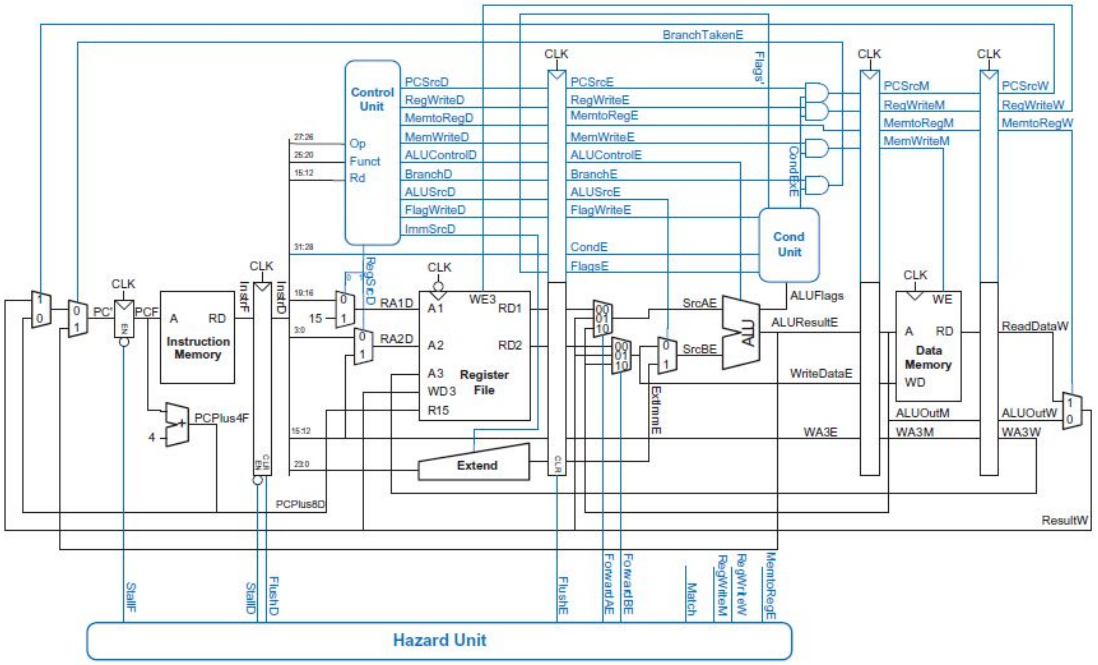

Figura 9 - Processador ARM *pipeline* completo (página 441 do livro)

1.3 Planejamento

Seguindo-se a mesma sequência lógica definida pelo livro, o trio tem a intenção de implementar em 2 semanas a descrição VHDL do processador.

A primeira semana será marcada pela implementação das divisórias do *pipeline* (unidades de armazenamento de dados parciais) e a unidade de controle de *hazard*.

Cada um dos registradores divisores será uma entidade VHDL separada, seus nomes serão “partial-IF-ID”, “partial-ID-EX”, “partial-EX-MEM”, “partial-MEM-WB”. Cada um dos bits de entrada, à exceção do *clock* e sinais de controle, deverão ser interceptados por um registrador (definido comportamentalmente) o qual passará adiante os resultados parciais desde que a unidade de controle libere isso.

A unidade de identificação de *hazards* deverá auxiliar a unidade de controle para que as instruções sejam executadas sequencialmente, conforme descrito no código de máquina armazenado na memória de dados do circuito.

A unidade de controle deverá ser implementada para primeiramente inserir apenas bolhas no *pipeline* quando algum conflito ocorrer. O grupo pretende deixar o projeto nesse estado até que o conjunto todo funcione corretamente.

A segunda semana será marcada pelos testes de unidade e de integração do que tiver sido implementado na semana anterior.

Uma vez estando o pipeline pronto e funcional, comprovado pelos testes feitos na segunda semana, o trio deverá modificar o circuito para incluir na unidade de controle de *hazard* o *forwarding* conforme descrito na seção anterior deste documento. É importante observar que, como esta funcionalidade é opcional para o funcionamento do processador, o grupo só irá implementá-la se houver tempo hábil para tal.

Paralelamente ao desenvolvimento das entidades novas que comporão o fluxo de dados e a unidade de controle do projeto, o grupo deve fazer *testbenchs* para teste de unidade sempre que achar pertinente, de forma que não haja grandes surpresas durante os testes finais de integração, dentre os quais o próprio *testbench* fornecido no documento “arm_single.vhd”.

Usando o GitHub, o grupo irá dividir as tarefas proporcionalmente e de forma que os desenvolvimentos de cada membro do trio sejam independentes entre si.

Uma divisão que poderia ser feita a princípio é:

- Elaboração das divisórias do pipeline referentes apenas aos dados parciais e otimizações no fluxo de dados (retirada do somador que gera (PC + 8), por exemplo).
- Elaboração das divisórias do pipeline referentes aos sinais de controle que devem ser repassados através dos estágios.
- Criação da unidade de controle de *hazard* e inclusão de *hardware* controlado por ela, bem como elaboração dos *testbenchs* para as entidades dos outros dois membros do grupo.

Agora, durante a fase de planejamento, essa divisão parece fazer sentido, mas pode ser mudada caso algum dos integrantes do trio ache estar sobrecarregado ou ocioso demais, de forma que o projeto possa progredir com rapidez.

Embora a execução de testes esteja planejada para a segunda semana, é normal que o grupo já consiga adiantar essa etapa na primeira semana à medida que os códigos fiquem prontos. Quanto a isso, o projeto é bastante maleável.