MODELAÇÃO DE UM PROCESSADOR

Relatório

Laboratório de Sistemas Digitais

Universidade de Aveiro

Bruno Lemos 98221

blemos@ua.pt

Departamento de Eletrónica, Telecomunicações e Informática

universidade de aveiro

INTRODUÇÃO

Este trabalho tem como objetivo validar por simulação um processador simplificado. Um Processador é capaz de realizar um conjunto de operações aritmé-ticas/lógicas sobre dois operandos guardados em registos de 8 bits e armazenar o resultado também num registo ou na memória de dados.

Datapath

- Mux, Pc, IMemory, Register, Alu, SignExtend, DMemory.

ControlUnit

- Controla o datapath.

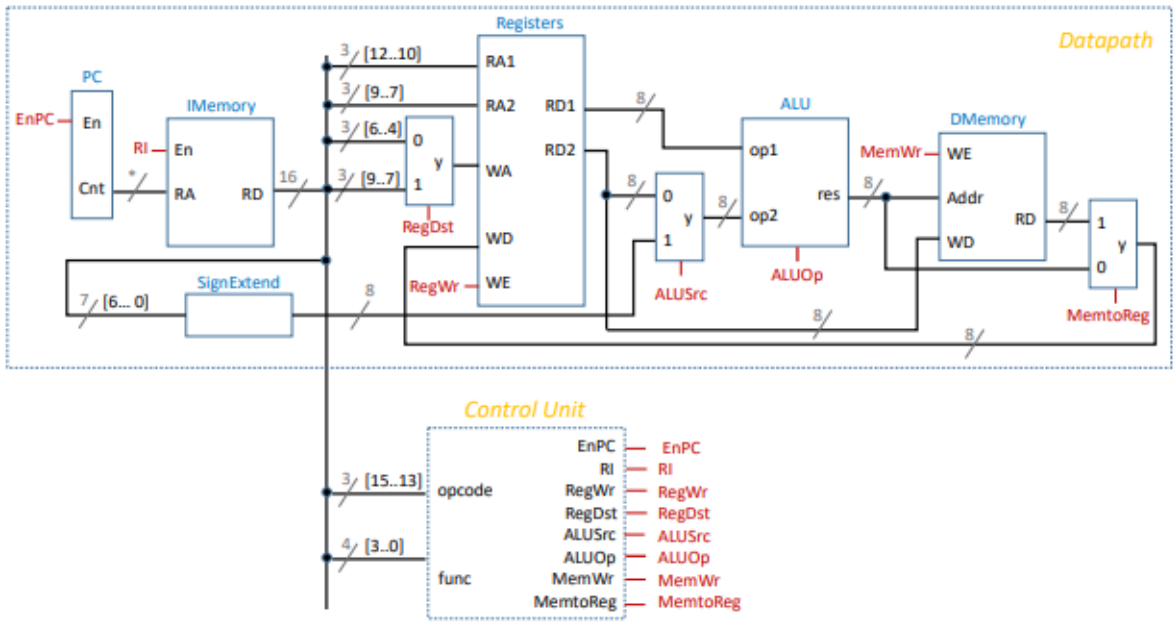


Figura 1 – Diagrama de blocos do processador.

Fase 1

Implementação e teste da Unidade de Execução (datapath)

Nesta fase foram sintetizados e validados com ficheiros de validação (*.wdf) todos os módulos que compõe o datapath. Alguns módulos são parametrizáveis de acordo com o que é explicado a seguir. Em alguns módulod foi acrescentada uma entrada de reset e clock para sincronizar operações de escrita em memórias e os sianis de fetch (RI e EnPC.

MUX 2:1

Este módulo é parametrizável, uma vez que são necessários 3 multiplexadores com diferentes tamanhos de barramento.

PROGRAM COUNTER (PC)

Este módulo vai permitir gerar os endreços para a memória de instruções, a utilizar durante a fase de fetch. Foi adicionada uma entrada de reset e clock. A entrada EnPC vai ser registada com transição ascendente do clock (rising_edge).

MEMÓRIA DE INSTRUÇÕES (IMEMORY)

Este módulo vai armazenar o programa no formato código máquina. É constituido por uma ROM com 8 palavras de 16 bits.
Foi adicionada uma entrada de clock que vai permitir registar a entrada EN com o transição ascendente do relógio (riding_edge).

MÓDULO DE REGISTOS (REGISTER)

Este modulo, não é mais do que uma memória com 8 palavras de 8 bits (8 registos de 8 bits)
O registo 0 nunca poderá ser escrito e será sempre “00000000”. Essa validação foi colocada no código VHDL.
Foi adicionada uma entrada de clk e uma entrada de reset.
O WE e o reset são validados pela transição ascendente do clk. Sempre que ocorre um reset todos os registos são colocados com o valor “00000000”.

ALU

Módulo que executa operações sobre dois operandos (op1, op2) de 8 bits. Este módulo suporta 16 operações (ALUop é constituída por 4 bits).

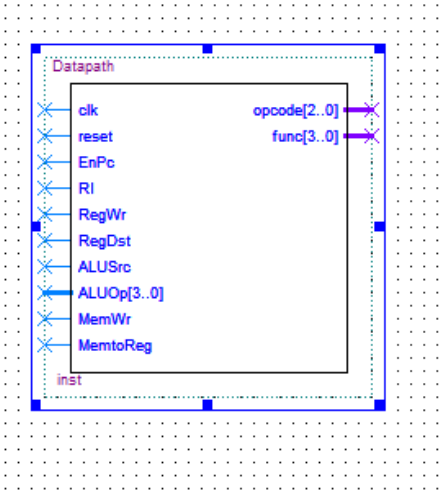
SIGNEXTENDED

Este módulo vai permitir converter uma palavra de 7 bits para 8 bits. É utilizado nas instruções tipo II. Como a instrução apenas suporta um address de 7 bits e a ALU (onde este sinal é utilizado) suporta 8 bits, é necessário fazer esta conversão.

MEMORIA DE DADOS

Este módulo implementa a memória de dados. É implementada tendo por base um array de 256 palavras de 8 bits.
Foi adicionado uma entrada de clock.
A entrada de WE da memória é registada com a transição ascendente do clk.

O Passo seguinte foi interligar todos os módulos. Para isso foi criado um novo módulo VHDL que designei por datapath.vhd. O simbolo resultante está representado a seguir:



Tal como pedido nesta primeira fase, a memoria de dados foi criada com o valor X"04" na primeira posição e e X"05" na segunda. Todos as outras posições ficam a X"00".

O programa foi traduzido para código máquina tal como é descrito a seguir:

ASSEMBLY	TIPO DE INSTRUÇÃO	CÓDIGO MAQUINA
LW $0, $1, 1	Tipo II	1110000010000001
LW $0, $2, 0	Tipo II	1110000100000000
ADD $1, $2, $3	Tipo I	0010010100110000
SLS $1, $2, $4	Tipo I	0010010101001100
SW $1, $3, 0	Tipo II	1100010110000000
SW $1, $4, 1	Tipo II	1100011000000001

Figura1 - tradução do código assembly

Na figura seguinte está o código máquina introduzido na memória de intruções (IMemory).

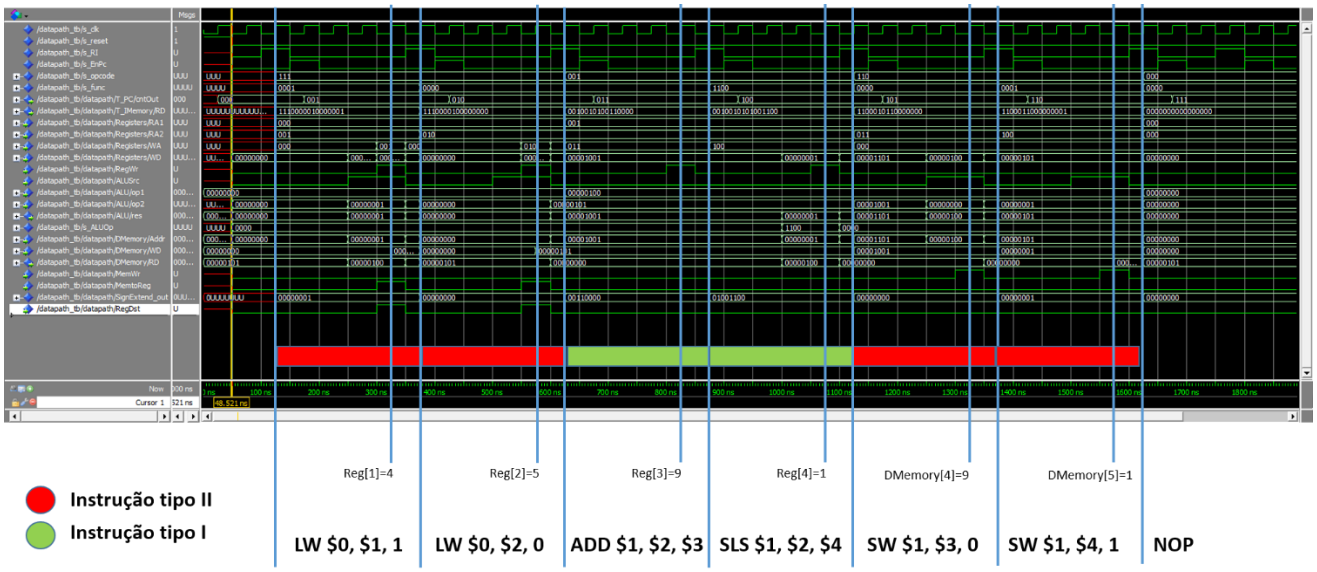
```
type TypeROM is array (0 to ((2 ** N)-1)) of TypeDataWord; -- size do array é dimencionado com base no numero de bits do RA
constant rom_data: TypeROM := (
"1110000010000001",   -- LW  $0, $1,  1 -> carregar no registo 1 dados que se encontram no endereco [registo0 +1] na memoria de dados
"1110000100000000",   -- LW  $0, $2,  1 -> carregar no registo 2 dados que se encontram no endereco [registo0 +0] na memoria de dados
"0010010100110000",   -- ADD $1, $2, $3 -> realizar a operacao registo3 = registo1  +  registo2
"0010010101001100",   -- SLS $1, $2, $4 -> realizar a operacao registo4 = registo1 SLS registo2
"1100010110000000",   -- SW  $1, $3,  0 -> escrever no endereco[registo1+0] na memoria de dados o conteudo do registo 3
"1100011000000001",   -- SW  $1, $4,  1 -> escrever no endereco[registo1+1] na memoria de dados o conteudo do registo 4
"0000000000000000",   -- NOP            -> nao fazer nada
"0000000000000000"    -- NOP            -> nao fazer nada
);
```

Figura2 – Implementação das instruçoes na Imemory

Foi criado um testbench para simular este módulo. Esse testbench gera os seguintes sinais:
- processo que gera um clk com um período de 50 ns
- processo que gera os sinais de controlo (EnPc, RI, RegWr, RegRst, ALUSrc, ALUOp, MemWr, MemtoReg) com base numa maquina de estados stim_proc. Este processo corre sempre na transição descendente do clk. A razão disto tem a ver com o facto de todos os blocos que utilizam sinais registados, estarem a fazer na transição ascendente. Desta forma na máquina de estímulos garantimos que os sinais de controlo já estão no nível pretendido quando a transição ascendente acontece . O sinal têm de estar estável durante a transição.

Na imagem seguinte é mostrada a execução do programa pretendido no simulador. Nestas imagens são identificadas as várias fases em que cada instrução é executada.

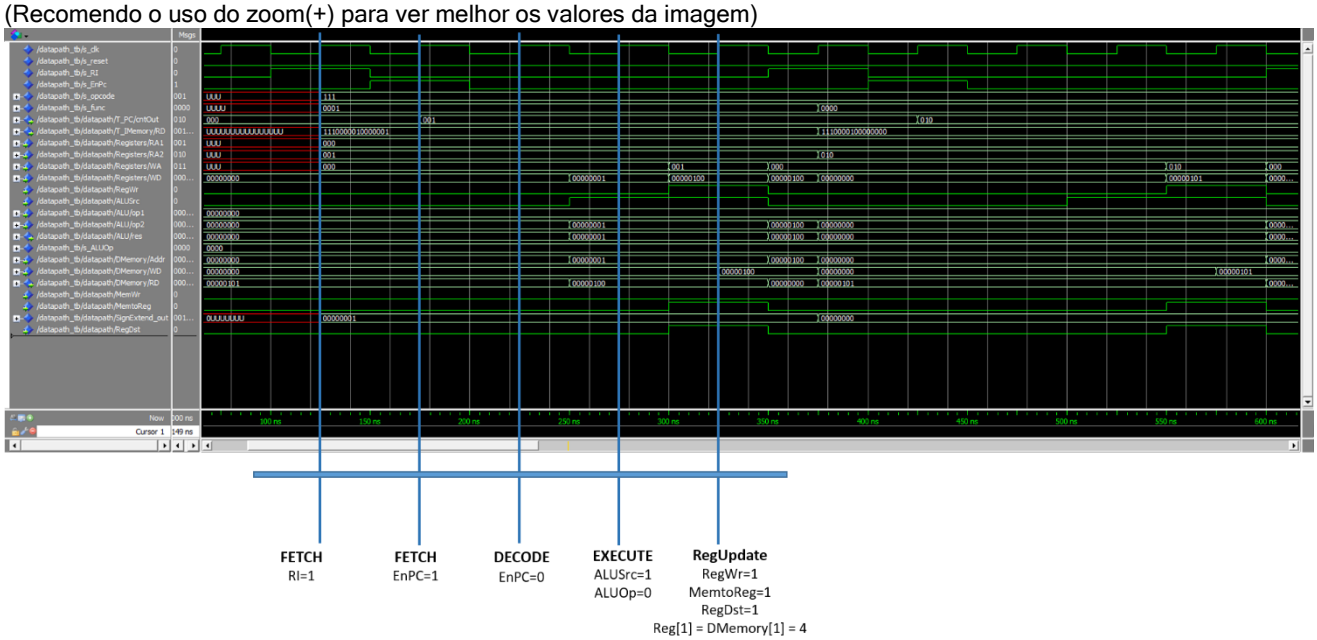
Programa dividido por instruções:
(Recomendo o uso do zoom(+) para ver melhor os valores da imagem)



Este programa calcula a soma dos valores que estão na posição 0 e 1 da memória de dados, (5 e 4 respetivamente) e de seguida testa se 4 é menor que 1. Estas operações são feitas depois de carregar o valores 5 e 4 para os registos 1 e 2 respetivamente. O resultado da soma (9) é guardado na posição 4 da memória de dados. O resultado da comparação (1) é guardado na posição 5 da memória de dados.

Programa dividido por estados (ex: execução da instrução: LW $0, $1, 1)

(Recomendo o uso do zoom(+) para ver melhor os valores da imagem)



FETCH
RI=1

FETCH
EnPC=1

DECODE
EnPC=0

EXECUTE
ALUSrc=1
ALUOp=0

RegUpdate
RegWr=1
MemtoReg=1
RegDst=1
Reg[1] = DMemory[1] = 4

Fase 2

Implementação e teste da Unidade de Controlo

Nesta fase foi desenvolvido o modo de controlo. Este módulo é responsável por gerar todos os sinais de controlo para o datapath de forma a permitir implementar uma instrução previamente grava na memória de instruções.

Implementa a máquina de estados representada na figura, onde se pode indentificar 4 estados principais. Alguns desses estados foram subdivididos de forma a refletir o facto de algumas variáveis de controlo terem valores diferentes dependentes do opcode da instrução.

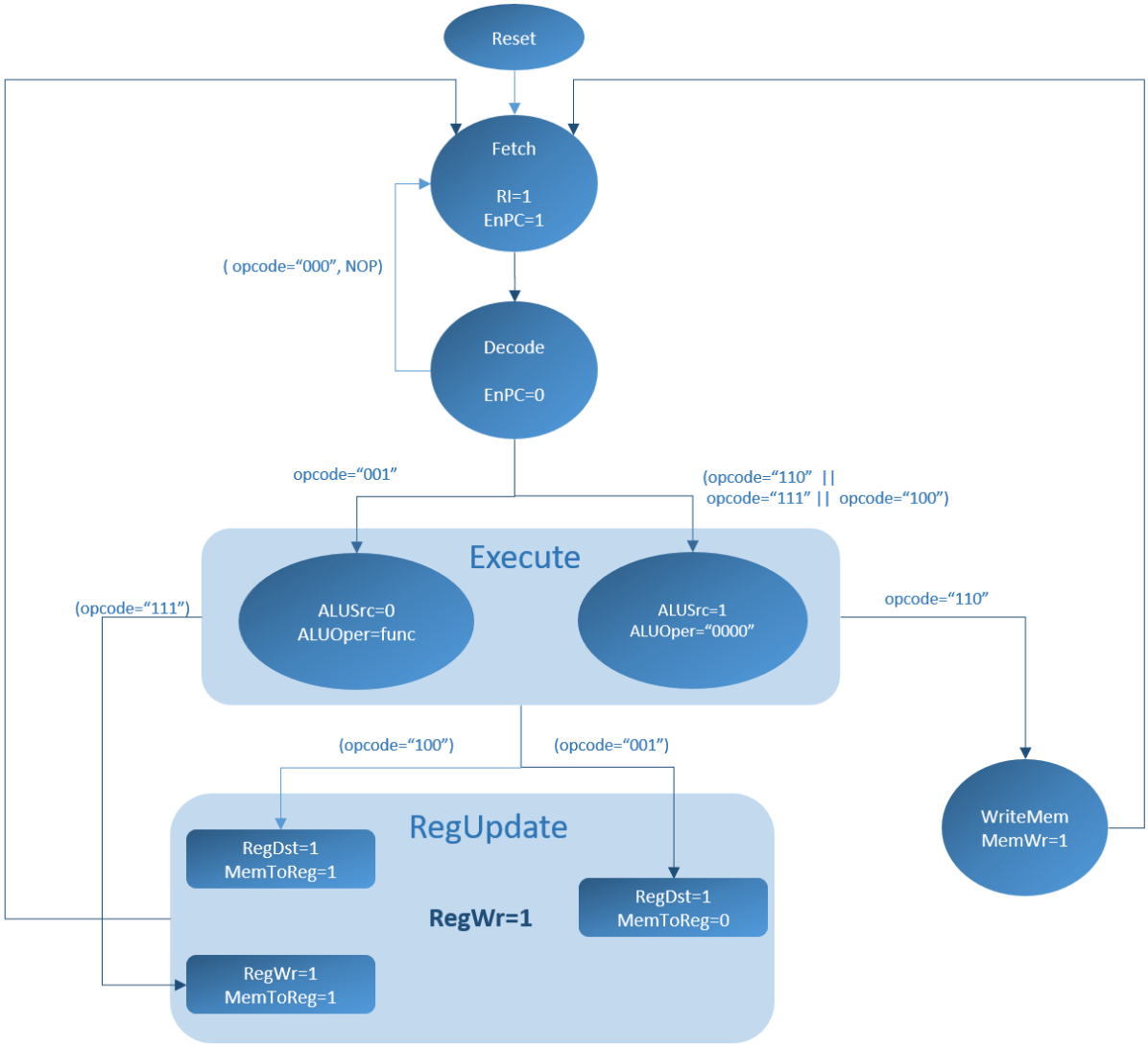
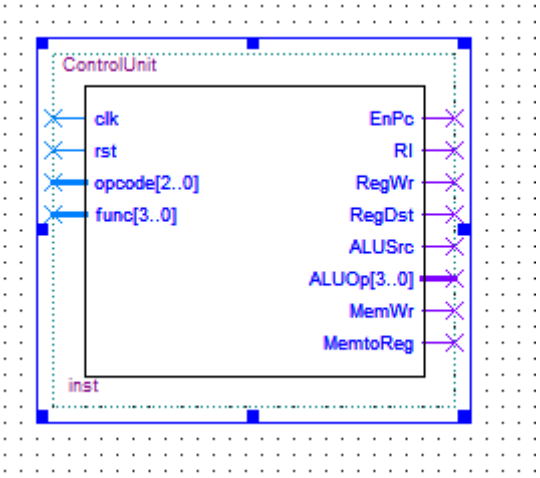


Diagrama de blocos

Na implementação deste módulo foram adicionados os sinais de reset e clock. Foi criado um processo que implementa a máquina de estados. Este processo é síncrono com a transição descendente do clock. A razão para esta decisão tem a ver com o facto de todos os blocos do datapath que possuem sinais registados, estarem síncronos com a transição ascendente. Desta forma garantimos que os sinais de controlo estão estáveis no datapath quando a transição ascente ocorrer.



Estados da máquina

FETCH:

O estado de fech foi decomposto em dois estados (FETCH0 e FETCH1) na máquina para garantir que o EnPC só é ativado quando a instrução tiver sido lida da memória de intruções (ativação do RI). Desta forma não teemos qualquer interferência entre a leitura e o incremento do Program Counter. Neste caso são precisos dois ciclos de relógio para esta operação. No final deste ciclo temos disponível à saída da IMemory o código máquina da instrução que deve ser processada(16 bits).

DECODE:

Este estado precisa apenas de 1 clico de relógio e atua no sinal do EnPC=0 de forma a que o contador não avance mais nehuma posição. A função destes foi referida no guião como sendo a de leitura dos registos. Uma vez que o RD1 e RD2 são colocados logo à saida quando os endereços estão à entrada do modulo Registos, nehum sinal de controlo é atuado nesta fase. Seria necessário se existisem sinais de leitura registados por clock, o que não é ocaso.

EXECUÇÃO:

Este estado tem como objetivo garantir a operação na ALU. Para isso são controlados dois sinais: ALUSrc e ALUOpcode. O seu valor depende mais uma vez do opcode da instrução que está a ser processada.

UPDATE_REG:
Neste estado é feito a escrita do resultado da operação na memória de registos. Para isso é necessário ativar o RegWr e os dados a escrever deverão ser encaminhados da ALU ou da memoria de dados, dependendo do tipo de opcode. Esse encaminhamento é configurado com o sinal MemRegDst (escolhe entre saída da ALU ou memeoria de dados) e RegDst (0 para instruções tipo I e 1 para instruções tipo II).

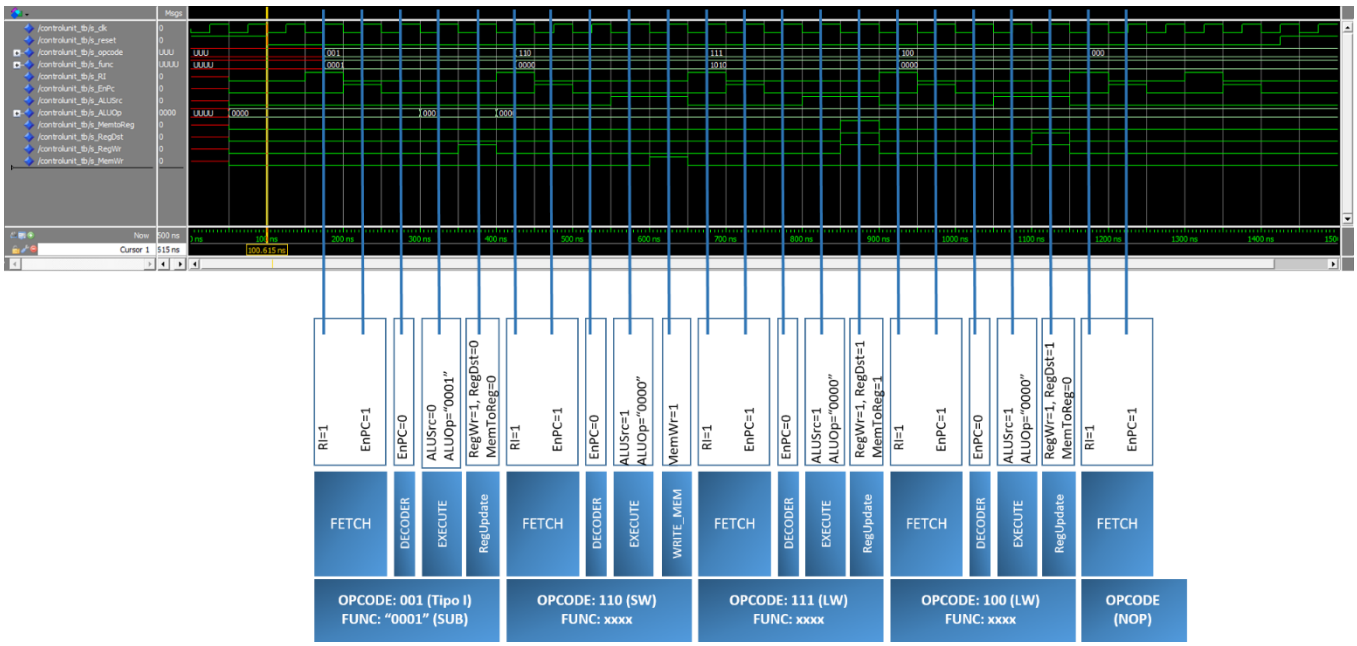
UPDATE_MEM:
Neste estado é feito a escrita do resultado da operação na memória de dados. Para isso deve ser ativado o sinal MemWr da memoria de dados.

NOTAS :

- Sempre que ocorre um opcode NOP, decidiu-se no DECODE que se passa para o estado de FETCH de forma a processar a instrução seguinte.

SIMULAÇÃO:

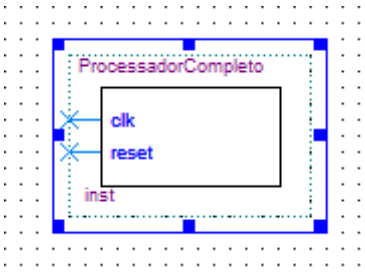
Foi criado um testebeanch com o nome datapath_TB.vhd que permite testar os vários opcodes e analisar os sinais de controlo gerados.
Na figura seguinte está a simulação dos vários opcodes.



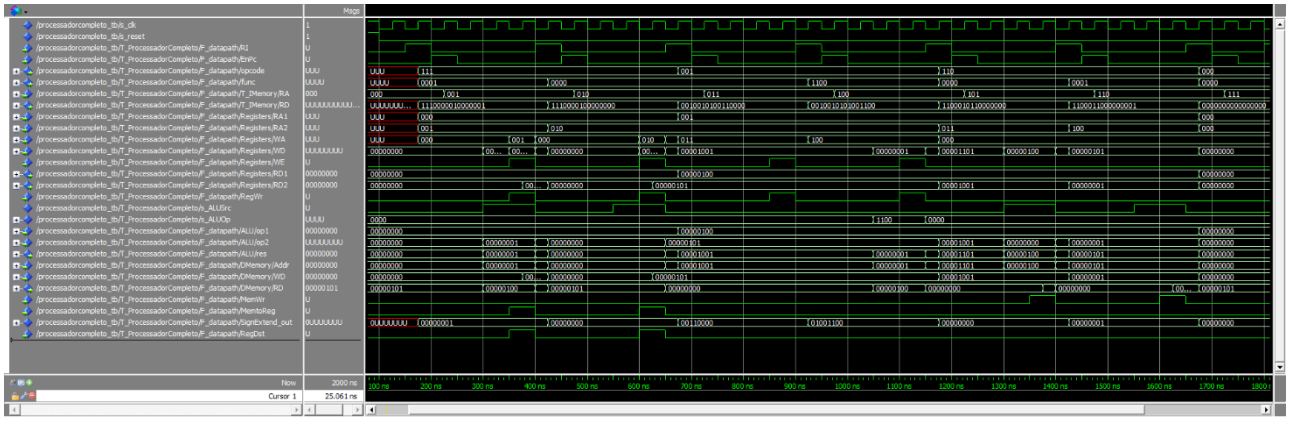
Fase 3

Interligação e teste do processador completo

Nesta fase o bloco desenvolvido na fase 1 (datapath) e na fase 2 (Unidade de controlo) foram interligados. Para isso foi criado o módulo ProcessadorCompleto.vhd. Este módulo terá apenas como entradas, o sinal de clock e reset.



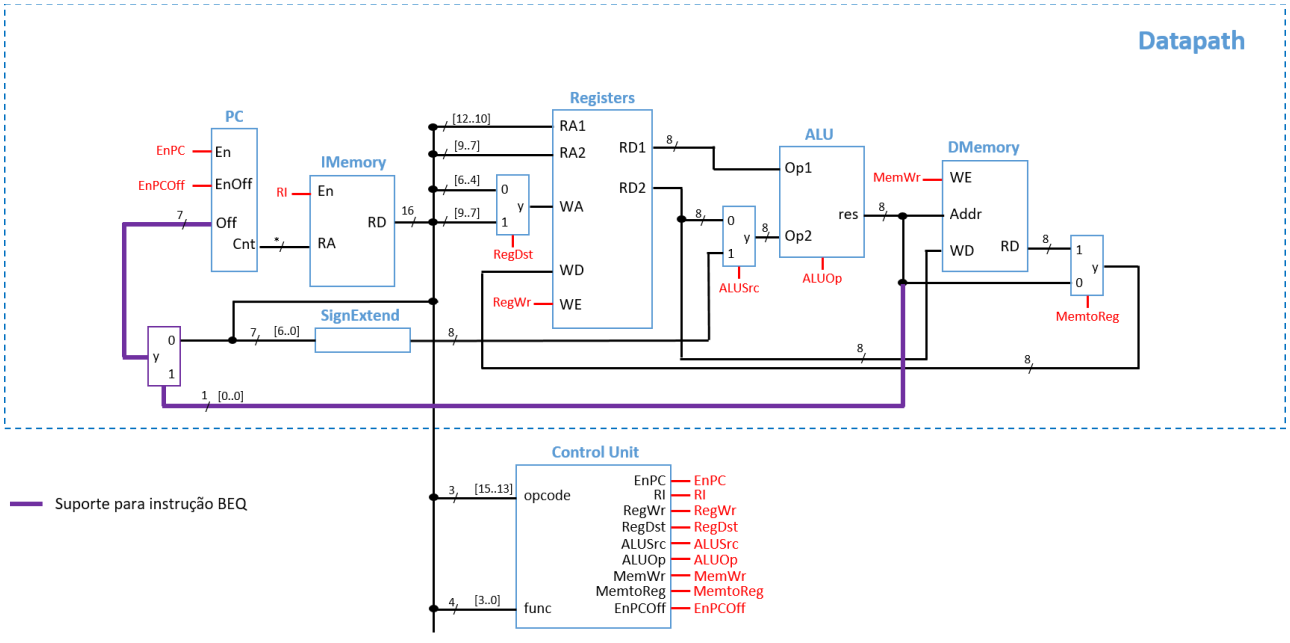
Foi criado um módulo de testbench; ProcessadorCompleto_tb.vhd, que permite simular o sistema, tendo apenas como inputs o reset e o clk. Foi utilizado programa carregado na fase 1. O resultado é apresentado na figura seguinte. Como seria de esperar o resultado é equivalente ao já demonstrado na simulação do datapath (fase 1).



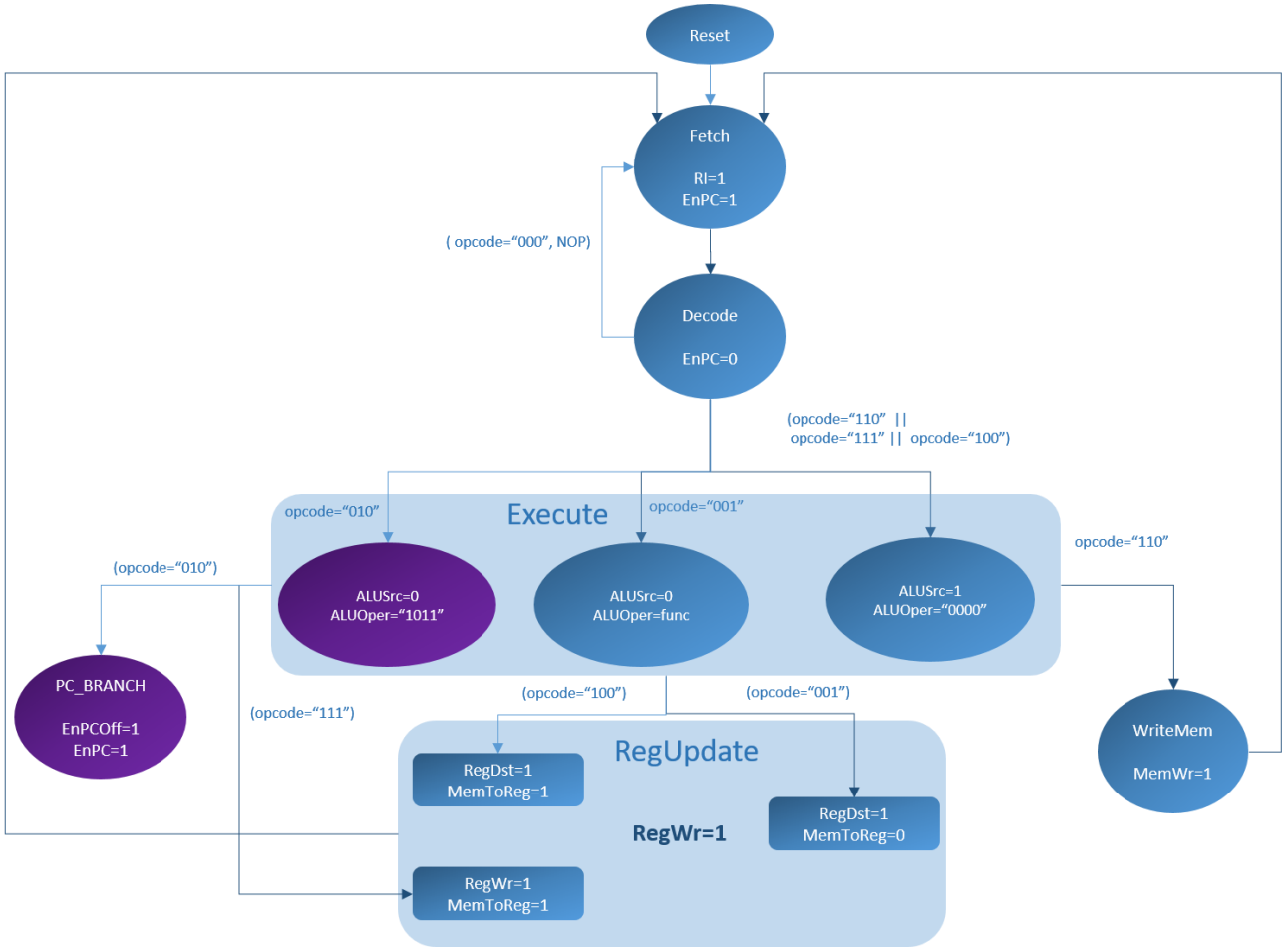
Fase 4.1

Adicionar instrução BEQ :

A instrução a adicionar é do tipo:
BEQ $rs, $rt, address, em que é comparado o valor ques está no registo $rs com $rt, e se forem iguais salta address instruções.
Com base neste pressuposto, esta intrução envolve uma operação da ALU (EQ) e dependendo do resultado (0 ou 1), o program counter deve avançar adress instruções antes do próximo fetch.
Para isso foi necessário alterar o módulo de Program Counter para suportar um incremento configurável de address posicoes. Foram adicionadas duas entradas:
Off: que indica quantas posições o contador deve saltar, e o EnOff que indica se a entrada off é para usar ou não.
O valor de Off depende da operação resultante da ALU e do valor que se encontra nos bits [7..0] da instrução do tipo II. Para isso foi introduzido um mux 2:1 para contemplar as operações possiveis.
Pretendemos que se a operação da ALU for 0, a entrada Off seja 0 (o PC não salta). Se for 1, o PC salta adress posições. Assim foi colocado na entrada 0 do mux o valor 0 e na entrada 1 o address descodificado da instrução tipo II.
Esta entrada é contrrolada pelo EnOff que é por sua vez controlado pela unidade de controlo.
Foi necessário colocar um novo estado na máquina da unidade de controlo para suportar este novo Opcode (“010”). Na figura seguinte está a unidade de execução e unidade de controlo atrualizadas.



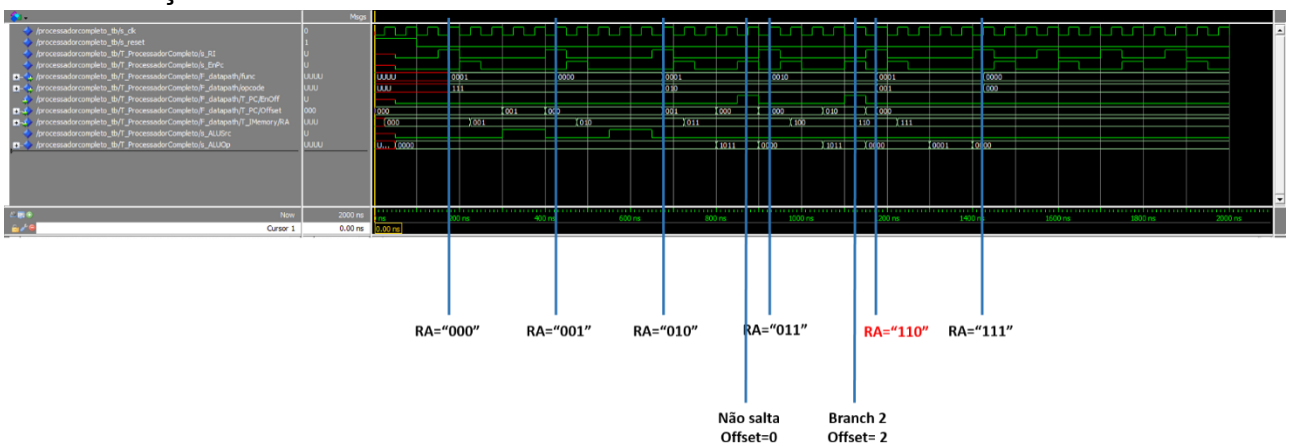
Na figura seguinte encontra-se o novo diagrama de estados que contempla a instrução BEQ (novo estado PC_BRANCH e alteração no estado de execução).



Para testar a implementação foi introduzido o seguinte código da Imemory:

```
    subtype TypeDataWord is std_logic_vector(15 downto 0);
    type TypeROM is array (0 to ((2 ** N)-1)) of TypeDataWord; -- size do array é dimencionado com base no numero de bits do RA
    constant rom_data: TypeROM := (
    "1110000010000001",   -- LW  $0, $1,  1 -> carregar no registo 1 dados que se encontram no endereco [registo0 +1] na memoria de dados
    "1110000100000000",   -- LW  $0, $2,  1 -> carregar no registo 2 dados que se encontram no endereco [registo0 +0] na memoria de dados
    "0100010100000001",   -- BEQ $1, $2,  1 -> salta 1 instrução se registo 1 = registo 2
    "0100010010000010",   -- BEQ $1, $1,  2 -> salta 2 instrução se registo 1 = registo 1 (VAI SALTAR)
    "0010010100110000",   -- ADD $1, $2, $3 -> realizar a operacao registo3 = registo1 ADD registo2
    "0010100100110000",   -- ADD $2, $2, $4 -> realizar a operacao registo4 = registo2 ADD registo2
    "0010100100110001",   -- ADD $2, $2, $4 -> realizar a operacao registo4 = registo2 SUB registo2
    "0000000000000000"    -- NOP            -> nao fazer nada
    );
```

Segundo este programa, o PC deverá saltar da instrução do address "011" para o "110" (salta duas instruções, porque a condição (***BEQ $1, $1, 2)*** vai dar falsa e assim vai salta duas instruções.



Fase 4.2

Programa para calcular o mais de 3 numeros :

Pretende-se criar um programa que calcule o maior de 3 número com sinal armazenados na memória de dados. Foram escolhidas as posições 0, 1 e 2 da Dmamemory para armazenar esses números.
Na tabela seguinte estão representadas as instruções, bem como a conversão para código máquina do programa. Como se pode verificar foram precisas 9 instruções. Uma vez que a memória de instruções inicialmente desenvolvida tem apenas 8 posições, foi preciso aumentar o tamanho dessa memoria. Para isso parametrizámos, tanto a memória de instruções de forma a que o RA passe a ter 4bits. O mesmo aconteceu para o Program Counter que passou a contar com 4 bits (16 posições).

ASSEMBLY	TIPO DE INSTRUÇÃO	CÓDIGO MAQUINA
LW $0, $1, 0	Tipo II	1110000010000000
LW $0, $2, 1	Tipo II	1110000100000001
SLS $1, $2, $3	Tipo I	0010010100111100
LW $3, $2, 0	Tipo II	1110110100000000
LW $0, $1, 2	Tipo II	1110000010000010
SLS $1, $2, 3	Tipo I	0010010100111100
SW $0, $2, 3	Tipo II	1100000100000011
LW $3, $4, 2	Tipo II	1110111000000010
SW $0, $4, 3	Tipo II	1100000110000011

Na figura seguinte está o conteúdo da IMemory com o programa carregado.

```
"1110000010000000",  -- LW  $0, $1,  0 -> carregar no registo 1 dados que se encontram no endereco [registo0 + 0] na memoria de dados
"1110000100000001",  -- LW  $0, $2,  1 -> carregar no registo 2 dados que se encontram no endereco [registo0 + 1] na memoria de dados
"0010010100111100",  -- SLS $1, $2, $3 -> registo3 = 1 se (signed)registo1 < (sigend)registo2 , se não dá 0.
"1110110100000000",  -- LW  $3, $2,  0 -> carregar no registo 2 dados que se encontram no endereco [registo3 + 0] na memoria de dados
"1110000010000010",  -- LW  $0, $1,  2 -> carregar no registo 1 dados que se encontram no endereco [registo0 + 2] na memoria de dados
"0010010100111100",  -- SLS $1, $2, $3 -> registo3 = 1 se (signed)registo1 < (sigend)registo2 , se não dá 0.
"1100000100000011",  -- SW  $0, $2,  3 -> escrever no endreço [registo0 + 3] na memória de dados o conteudo do registo 2
"1110111000000010",  -- LW  $3, $4,  2 -> carregar no registo 4 dados que se encontram no endereco [registo3 + 2] na memoria de dados
"1100000110000011",  -- SW  $0, $4,  3 -> escrever no endreço [registo0 + 3] na memória de dados o conteudo do registo 4
others => "0000000000000000"
```

Instrução nº1>

Carrega o primeiro número da memória de dados para o registo1

Instrução nº2>

Carrega o segundo número da memória de dados para o registo 2

Instrução nº3>

Se o valor do registo 1 < valor do registo 2, escreve 1 no registo 3. Caso contrário escreve 0. Este indice serve para indica na memória qual o valor maior.

Nesta fase já encontrámos o maior número das duas primeiras posições.

Instrução nº4:

Carrega o indice registo3+0 da memória de dados para o registo 2. Carrega o maior dos dois primeiros números no registo 2.

Instrução nº5:

Carrega o terceiro numero da memória de dados para o registo 1

Instrução nº6:

Se o valor do registo 1 < valor do registo 2, escreve 1 no registo 3. Caso contrário escreve 0. Este indice serve para indicar na memória qual o valor maior (posição 2 de memória + indice).
Nesta fase já encontrámos o maior número.

É necessário agora que esse número seja copiado para a posição 3 da memória.

Instrução nº7:

Carrega na memória de dados (registo3 + 2), o maior dos dois primeiros números.

Instrução nº8:

Carrega o indicie registo3+2 da memória de dados para o registo 4.

Instrução nº9 ->

Carrega o registo4 para a memória de dados (registo0 + 3).

Em conclusão: O maior dos 3 primeiros números armazenados na DMemory e armazenado na posição 3 de DMemory.