

LABORATÓRIO DE ARQUITETURA DE COMPUTADORES

Projeto Final

Descrição Simplificada do Processador MIPS em VHDL

Grupo: 5 Turma: B

Caroline Aparecida de Paula Silva	726506
Gabriel Rodrigues Rocha	726518
Henrique Shinki Kodama	726537
Isabela Sayuri Matsumoto	726539

Sumário

1. Introdução	3
R-format	4
I-format	5
J-format	6
2. Módulos do Projeto	7
I-fetch	7
I-decode	7
Control	7
Execute	8
3. Simulação	10
AND	10
OR	11
BNE - Branch Not Equal	11
JAL e JR - Jump and Link / Jump Register	12
SLL - Shift Left Logical	13
SRL - Shift Right Logical	14
SLT - Set on Less Than	15
4. Vetor - Implementação e Simulação	18
5. Construção do Projeto	21
6. Considerações Finais	22
Referências	23
Apêndice A	24
SLL	24
SRL	24
SLT	25
ADDI	25
Vetor	26

1. Introdução

O microprocessador MIPS foi criado na década de 80 e é amplamente utilizado, principalmente em sistemas embarcados. Possui uma arquitetura RISC baseada no uso de registradores e é capaz de realizar um número compacto e consistente de instruções de tamanho fixo de 32 bits e que variam apenas entre três formatos. O MIPS possui 32 registradores de propósito geral, e cada um deles corresponde a uma palavra (32 bits).

Esse projeto representa a parte final da proposta de realizar uma descrição simplificada, em VHDL, do microprocessador MIPS. Anteriormente, foram realizados três experimentos nos quais elaborou-se a implementação de módulos básicos do microprocessador: uma unidade de busca de instruções (*Ifetch*), uma de decodificação (*Idecode*), uma de controle (*Control*) e uma de execução das instruções (*Execute*). A memória é organizada separadamente, com um módulo para a memória do programa (*program.mif*) e um para a memória de dados (*dmemory.mif*).

Considere que a implementação usada como base do presente projeto já suporta e executa corretamente as seguintes instruções:

- **Add:** realiza a soma de dois registradores e coloca o resultado em um registrador destino.
- **Sub:** realiza a subtração de dois registradores e coloca o resultado em um registrador destino.
- **And:** realiza a operação de “E” lógico entre os bits de dois registradores e coloca o resultado em um registrador destino.
- **Or:** realiza a operação de “OU” lógico entre os bits de dois registradores e coloca o resultado em um registrador destino.
- **Lw (*Load Word*):** carrega uma palavra de uma posição específica da memória e coloca em um registrador.
- **Sw (*Store Word*):** escreve uma palavra contida em um registrador em uma posição específica da memória.
- **Beq (*Branch on equal*):** salta para o endereço de um rótulo se dois registradores tiverem mesmo valor.

Será descrito neste documento as modificações realizadas para o suporte de novas instruções: **jr** (*jump register*), **sll** (*shift left logical*), **srl** (*shift right logical*), **j** (*jump*), **jal** (*jump and link*), **bne** (*branch on not equal*), e **addi**.

R-format

As instruções R-format possuem o seguinte formato:

31	26	25	21	20	16	15	11	10	6	5	0
opcode	rs	rt	rd	shamt	funct						

Jr – Jump Register:

000000	src	00000	00000	00000	001000	JR rs
--------	-----	-------	-------	-------	--------	-------

A instrução jr salta para o endereço contido em um registrador (rs). Para a codificação são reservados, além dos 6 bits para o código da instrução, 5 bits para o registrador que contém o endereço destino, e os 6 últimos bits para o código da função. Note que os bits de 20 a 6 são zerados.

Srl – Shift Right Logical:

000000	00000	src	dest	shamt	000010	SRL rd, rt, shamt
--------	-------	-----	------	-------	--------	-------------------

A instrução srl desloca o valor do registrador de origem para a direita um determinado número de vezes (shamt), completa o número de bits necessários com zero, e coloca o resultado em um registrador destino. Para a codificação são reservados os 6 primeiros bits para o código da operação, os bits 25 a 21 são indiferentes, 5 bits para o registrador de origem, 5 bits para o registrador destino, 5 bits para o deslocamento e os 6 últimos bits para o código da função.

Sll – Shift Left Logical:

000000	00000	src	dest	shamt	000000	SLL rd, rt, shamt
--------	-------	-----	------	-------	--------	-------------------

A instrução sll desloca o valor do registrador de origem para a esquerda um determinado número de vezes (shamt), completa o número de bits necessários com zero, e coloca o resultado em um registrador destino. A codificação é análoga a instrução srl.

Slt - Set on Less Than:

000000	src1	src2	dest	00000	101010	SLT rd, rs, rt
--------	------	------	------	-------	--------	----------------

A instrução slt realiza uma comparação entre dois registradores (rs e rt de acordo com a figura) e atribui um valor verdadeiro (diferente de zero) ao registrador destino (rd) caso $rs < rt$ e zero caso contrário. Para a codificação são reservados os bits 25 a 21 para o primeiro registrador, os bits 20 a 16 para o segundo registrador e por fim, os bits de 15 a 11 para o registrador destino. Por padrão os bits 10 a 6 são zerados e os últimos 5 são reservados para o código da função.

I-format

As instruções do tipo I-format possuem o seguinte formato:

31	26	25	21	20	16	15	11	10	6	5	0
opcode						rs		rt		immediate	

Addi – Add immediate value:

001001	src	dest	signed immediate	ADDIU rt, rs, signed-imm.
--------	-----	------	------------------	---------------------------

A instrução *Addi* realiza a soma de um registrador com um valor imediato e coloca o resultado em um registrador destino. Para a codificação são reservados, além dos 6 primeiros bits para o código da operação, 5 bits para o registrador de origem, 5 bits para o registrador destino e os 16 últimos bits para o valor imediato inteiro, o qual pode ser, positivo ou negativo.

Bne – Branch on not equal:

000101	src1	src2	signed offset	BNE rs, rt, offset
--------	------	------	---------------	--------------------

A instrução *Bne* verifica a igualdade entre dois registradores, e caso não sejam iguais, salta para o rótulo *offset*. Para a codificação são reservados, além dos 6 primeiros bits para o código da operação, 5 bits para o primeiro registrador a ser comparado, 5 bits para o segundo registrador a ser comparado e os 16 últimos bits para o endereço do rótulo. Vale ressaltar que, no caso do salto ser realizado o PC já incrementado é somado ao valor do rótulo, se não, o PC continua normalmente, sem alterações, para a próxima instrução.

Beq - Branch on equal:

000100	src1	src2	signed offset	BEQ rs, rt, offset
--------	------	------	---------------	--------------------

A instrução *Beq* é semelhante ao *Bne*, sua codificação é igual, mudando apenas seu *opcode*, porém o salto é realizado quando os registradores são iguais.

J-format

As instruções do tipo J-format possuem o seguinte formato:

31	26	25	21	20	16	15	11	10	6	5	0
opcode						target					

J - Jump:

000010	target	J target
--------	--------	----------

A instrução j salta para a instrução contida no endereço do rótulo *target*. Para a codificação são reservados 6 bits para o código da operação e 26 bits para o endereço do rótulo. Vale ressaltar que o salto é incondicional, ou seja, sempre o PC já incrementado é somado ao valor do rótulo.

Jal - Jump and Link:

000011	target	JAL target
--------	--------	------------

A instrução jal salta para a instrução contida no endereço do rótulo *target* e armazena o endereço de retorno no registrador \$31. A codificação é análoga a da instrução jump. O salto também é realizado incondicionalmente, ou seja, o PC é incrementado é sempre somado ao valor do rótulo.

2. Módulos do Projeto

Cada módulo do projeto sofreu alterações para admitir o funcionamento de novas instruções. A síntese dessas mudanças e as convenções adotadas encontram-se a seguir:

I-fetch

O módulo *Ifetch* contém as descrições necessárias para o incremento do PC de forma a adquirir da memória as instruções que serão decodificadas. Há um *process* que zera o contador de programas quando o *reset* é ativo alto. A cada subida do clock o sinal do PC é incrementado. Como o projeto estrutura a memória de uma forma diferente em relação ao padrão MIPS, utilizando palavras ao invés de bytes, o incremento do contador de programas é realizado unitariamente, ao invés de 4 em 4.

Além disso, a unidade de *Ifetch* foi modificada para a alteração do valor do contador de programas (PC) nas instruções de salto, através da adição de um multiplexador. Para o **beq**, **bne** e o **jr**, o PC recebe o valor do *ADDResult* que contém o endereço da próxima instrução, cujo valor foi adquirido na ULA (*Execute*). Já na instrução **jump**, o PC recebe o valor específico do endereço da nova instrução (*instrJump*). Na instrução **jal**, o PC recebe o valor contido no sinal *instrJump*, que contém o valor do endereço de retorno que foi guardado no registrador \$31. Caso não seja uma instrução de pulo, o multiplexador seleciona o valor do *PC_inc*.

I-decode

A unidade de decodificação separa os bits de cada instrução conforme foi descrito, e atribui os valores dos registradores a sinais que serão utilizados ao longo da execução das instruções. Há um multiplexador que seleciona qual dos registradores deve ser utilizado para escrita. Instruções de *R-format* utilizam o registrador rd e instruções de *I-format* utilizam o registrador rt.

A extensão de sinal é realizada para converter sinais de instruções de *I-format* de 16 para 32 bits. Apesar da extensão sempre ser realizada, ela é usada, efetivamente, somente nas instruções desse formato.

Além disso, há um *process* que permite a escrita da memória na subida do *clock* e inicializa o banco de registradores de acordo com seu índice. Uma alteração realizada, foi que, agora, quando a instrução pretendida for **jump and link**, é necessário que se altere o valor do registrador \$31 para o valor atual do PC, e desse modo, o endereço de retorno seja armazenado.

Control

A unidade de controle ativa os sinais específicos para cada formato de instrução. Sinais de escrita e leitura da memória e de registradores são habilitados de acordo com o código da operação (*opcode*) de cada instrução. É importante ressaltar que, agora, o sinal *Branch* possui dois bits, caso contrário, o sinal para a realização do salto seria habilitado

sempre que fosse instrução de **beq** ou **bne**, independente do resultado da verificação de igualdade entre os dois registradores. Convencionou-se que 10 refere-se a instrução **beq** e 01 a instrução **bne**.

Foi acrescentado sinais de controle para as instruções de pulo incondicional, **jump**, **jal**. Além disso, modificou-se o multiplexador que seleciona o bit menos significativo do sinal *AluOp* (diferencia qual operação será realizada no ULA), para ficar ativo alto nas instruções **beq** e **bne**.

Execute

A unidade de execução é aquela que decide quais operações serão feitas na unidade lógica aritmética (ALU), tendo como saída os sinais *Alu_Result*, *Zero*, *JumpReg* e *ADDResult* (o sinal *Zero* é ativo alto quando *Read_data1* for igual a *Read_data2*, utilizado para **bne/beq**, e o sinal *JumpReg* é utilizado como auxílio para a instrução **jr**, sendo ativo alto quando for uma instrução desse tipo).

O sinal interno *Alu_ctl*, de 5 bits, determina qual operação (**and**, **or**, **add**, **sub**, **slt**, **jr**, **srl**, **sll**) será realizada e o que será passado para *Alu_Result*. Abaixo segue a tabela de *Alu_ctl*:

Instrução	<i>Alu_ctl</i> (4)	<i>Alu_ctl</i> (3)	<i>Alu_ctl</i> (2)	<i>Alu_ctl</i> (1)	<i>Alu_ctl</i> (0)
AND	0	0	0	0	0
OR	0	0	0	0	1
ADD	0	0	0	1	0
SUB	0	0	1	1	0
SLT	0	0	1	1	1
JR	0	1	0	0	0
SLL	0	0	1	0	0
SRL	1	0	0	1	0

Os bits do *Alu_ctl* são determinados pelos sinais *AluOp* (que vem do controle) e *Funct* (os seis bits menos significativo da instrução).

Vale notar que o sinal *JumpReg* é passado para o componente I-fetch, para que *Next_PC* receba o registrador destino (presente em *Alu_Result*), comumente o *\$ra*, e a

instrução **jr** seja feita corretamente. Além disso, para as demais instruções, a ALU sempre realiza a operação de soma, o que é conveniente para a execução da instrução **addi**.

3. Simulação

Para cada instrução foi realizada uma simulação separada, com objetivo de mostrar suas funcionalidades individualmente. Por padrão, todos os registradores são inicializados com seus valores de 0 a 31 e as *waves* serão apresentadas apenas nas instruções que forem necessárias. As simulações encontram-se abaixo:

AND

A instrução utilizada foi And \$9 \$D \$A. Na primeira imagem no banco de registradores temos \$9 = 1001(registrador destino), \$A = 1010 e \$D = 1101. Na segunda imagem, o banco de registradores representa o estado dos registradores após a instrução utilizada, temos \$9 = 1000 e sem mudanças nos demais registradores. Portanto a simulação foi bem-sucedida, já que o *and* bit a bit resultou no valor correto e o resultado obtido foi escrito no registrador destino.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	01AA4824	00000000	00000000	00000000	00000000	00000000	00000000	00000000
008	00000000	00000000	00000000	08000000	00000000	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figura 1.1 - and.mif

Figura 1.2 - Banco de registradores antes

Figura 1.3 - Banco de registradores depois

readData2, e portanto deve acontecer o pulo. No próximo pulso de clock, o PcAddr = 0x04 mostrando que houve o pulo e comprovando que a simulação foi bem-sucedida.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	14000002	14040002	00000000	00000000	00000000	00000000	00000000	00000000	-----
008	00000000	00000000	00000000	08000000	00000000	00000000	00000000	00000000	-----
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	-----
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	-----
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	-----

Figura 3.1 - bne.mif



Figura 3.2 - waves do bne

JAL e JR - Jump and Link / Jump Register

Para simular **jal** e **jr** foram utilizados o mesmo program.mif. Nota-se, através da variável PCAddr que a simulação foi bem-sucedida, pois o **jal** faz o salto para a posição 0xC da memória, que possui a instrução **jr**, voltando para a posição inicial contida no registrador \$ra.

Nas *waves*, o sinal de controle auxJump fica ativo alto quando o PcAddr é 0x0, indicando que deve haver o pulo e o sinal auxLink também fica alto, o que indica que deve ser escrito no registrador \$31 o endereço da próxima instrução na sequência. No próximo pulso de *clock* o PcAddr é 0xC (que possui o jr), ou seja, o pulo foi feito com sucesso. O sinal auxJumpReg fica ativo alto sinalizando que deve haver o pulo para instrução contida no registrador \$31. No próximo pulso de clock o PcAddr volta para 0x1, comprovando que **Jr** foi realizado como o esperado.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	0C00000C	00000000	00000000	00000000	00000000	00000000	00000000	00000000
008	00000000	00000000	00000000	00000000	03E00008	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figura 4.1 - jaljr.mif

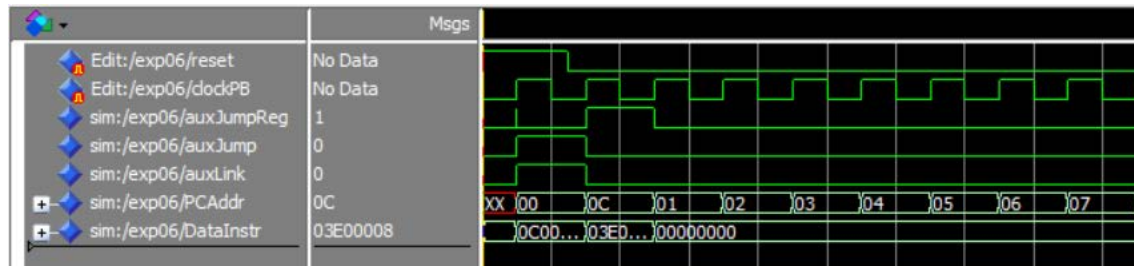


Figura 4.2 - waves do jal e do jr

SLL - Shift Left Logical

Foi utilizado a operação sll \$0x9 \$0xA 0x2 e os resultados estão corretos, uma vez que o valor inicial de \$0xA era de 1010 e o \$0x9 era 001001, e ao deslocar duas vezes para a esquerda e completar a parte menos significativa com zeros, chegamos ao valor de 101000, exatamente o conteúdo que foi escrito no registrador \$0x9 (registrador destino), após a execução da operação.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	000A4880	00000000	00000000	00000000	00000000	00000000	00000000	00000000
008	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figura 5.1 - sll.mif

00000000	00000000000000000000000000000000
00000001	00000000000000000000000000000001
00000002	00000000000000000000000000000010
00000003	00000000000000000000000000000011
00000004	00000000000000000000000000000100
00000005	00000000000000000000000000000101
00000006	00000000000000000000000000000110
00000007	00000000000000000000000000000111
00000008	00000000000000000000000000001000
00000009	00000000000000000000000000001001
0000000a	00000000000000000000000000001010
0000000b	00000000000000000000000000001011
0000000c	00000000000000000000000000001100
0000000d	00000000000000000000000000001101
0000000e	00000000000000000000000000001110
0000000f	00000000000000000000000000001111
00000010	00000000000000000000000000010000
00000011	00000000000000000000000000010001
00000012	00000000000000000000000000010010
00000013	00000000000000000000000000010011
00000014	00000000000000000000000000010100
00000015	00000000000000000000000000010101
00000016	00000000000000000000000000010110
00000017	00000000000000000000000000010111
00000018	00000000000000000000000000011000

Address: hexadecimal Data: symbolic

Figura 5.2 - Banco de registradores antes

00000000	00000000000000000000000000000000
00000001	00000000000000000000000000000001
00000002	00000000000000000000000000000010
00000003	00000000000000000000000000000011
00000004	00000000000000000000000000000100
00000005	00000000000000000000000000000101
00000006	00000000000000000000000000000110
00000007	00000000000000000000000000000111
00000008	00000000000000000000000000001000
00000009	00000000000000000000000000010100
0000000a	00000000000000000000000000001010
0000000b	00000000000000000000000000001011
0000000c	00000000000000000000000000001100
0000000d	00000000000000000000000000001101
0000000e	00000000000000000000000000001110
0000000f	00000000000000000000000000001111
00000010	00000000000000000000000000010000
00000011	00000000000000000000000000010001
00000012	00000000000000000000000000010010
00000013	00000000000000000000000000010011
00000014	00000000000000000000000000010100
00000015	00000000000000000000000000010101

Address: hexadecimal Data: symbolic

Figura 5.3 - Banco de registradores depois

SRL - Shift Right Logical

Foi utilizado a operação `srl $0x9 $0xA 0x2`. O resultado está de acordo com o esperado, uma vez que o valor inicial de `$0xA` era de 1010 e o `$0x9` era 1001, e ao deslocar duas vezes para a direita, e completar a parte mais significativa com zeros, chegamos ao valor de 0010, exatamente o conteúdo que foi escrito no registrador `$0x9` (registrador destino), após a execução da operação.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	000A4882	00000000	00000000	00000000	00000000	00000000	00000000	00000000
008	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figura 6.1 - srl.mif

Address	Data
00000000	00000000000000000000000000000000
00000001	00000000000000000000000000000001
00000002	00000000000000000000000000000010
00000003	00000000000000000000000000000011
00000004	000000000000000000000000000000100
00000005	000000000000000000000000000000101
00000006	000000000000000000000000000000110
00000007	000000000000000000000000000000111
00000008	0000000000000000000000000000001000
00000009	0000000000000000000000000000001001
0000000a	0000000000000000000000000000001010
0000000b	0000000000000000000000000000001011
0000000c	0000000000000000000000000000001100
0000000d	0000000000000000000000000000001101
0000000e	0000000000000000000000000000001110
0000000f	0000000000000000000000000000001111
00000010	0000000000000000000000000000010000
00000011	0000000000000000000000000000010001
00000012	0000000000000000000000000000010010
00000013	0000000000000000000000000000010011
00000014	0000000000000000000000000000010100
00000015	0000000000000000000000000000010101
00000016	0000000000000000000000000000010110
00000017	0000000000000000000000000000010111
00000018	0000000000000000000000000000011000

Figura 6.2 - Banco de registradores antes

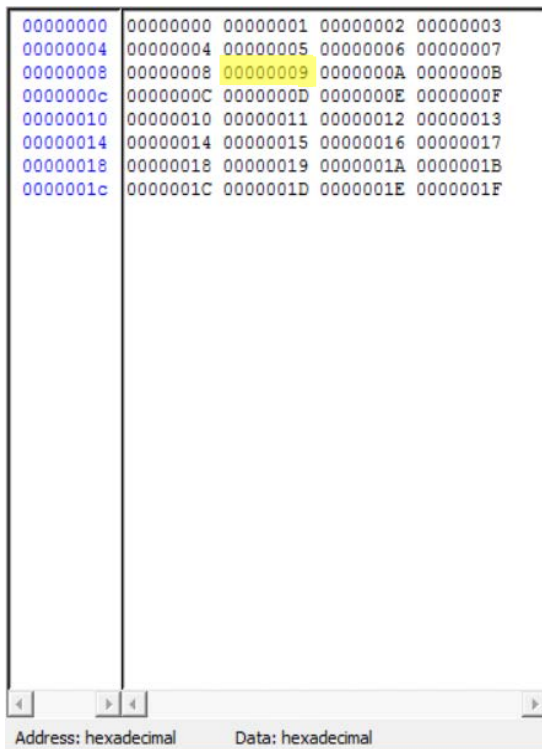
Figura 6.3 - Banco de registradores depois

SLT - Set on Less Than

Foram utilizadas as instruções `slt $0x9 $0xD $0xA` e `slt $0x9 $0xA $0xD`. A simulação ocorreu corretamente pois é escrito no registrador \$0x9 o valor 0x0 após a primeira instrução, visto que o conteúdo de \$0xD > \$0xA. Após a terceira instrução é escrito o valor 0x1, visto que o conteúdo \$0xA < \$0xD. A segunda instrução está nula (nop) para facilitar a visão do resultado no banco de registradores. Portanto, a operação e a escrita foram executadas de maneira correta.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	01AA482A	00000000	014D482A	00000000	00000000	00000000	00000000	00000000
008	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

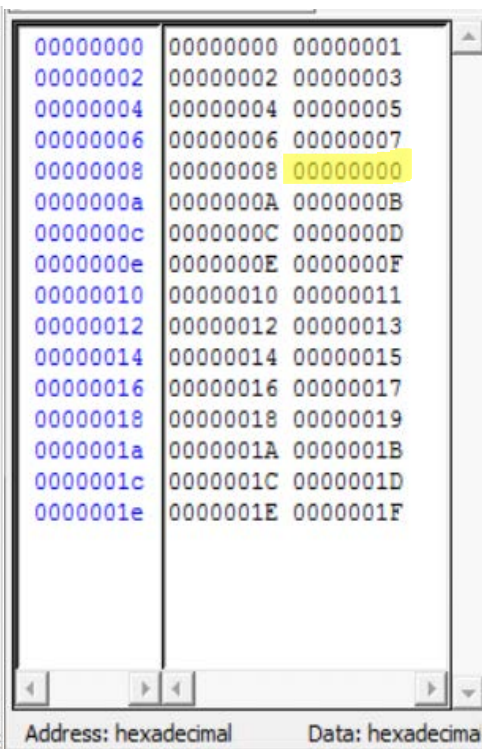
Figura 7.1 - slt.mif



Initial state of the register bank. The registers are initialized with values from 0x0 to 0xF. The register \$0x9 (index 9) contains the value 0x9.

Address (hex)	Data (hex)
00000000	00000000
00000004	00000004
00000008	00000008
0000000c	0000000c
00000010	00000010
00000014	00000014
00000018	00000018
0000001c	0000001c
00000001	00000001
00000002	00000002
00000003	00000003
00000005	00000005
00000006	00000006
00000007	00000007
0000000a	0000000a
0000000b	0000000b
0000000d	0000000d
0000000e	0000000e
0000000f	0000000f
00000011	00000011
00000012	00000012
00000013	00000013
00000015	00000015
00000016	00000016
00000017	00000017
00000019	00000019
0000001a	0000001a
0000001b	0000001b
0000001d	0000001d
0000001e	0000001e
0000001f	0000001f

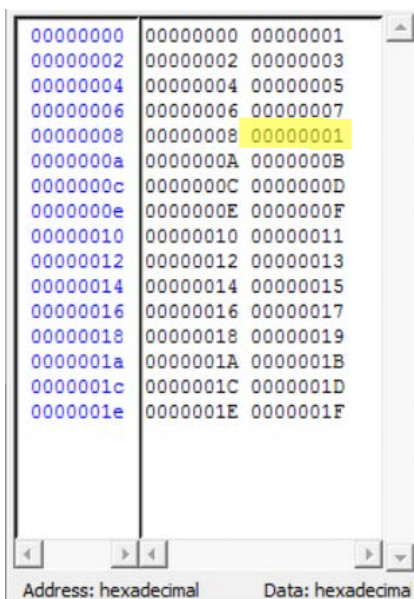
Figura 7.2 - Banco de registrador inicialmente



State of the register bank after executing instruction 0x0. The register \$0x9 (index 9) now contains the value 0x0.

Address (hex)	Data (hex)
00000000	00000000
00000002	00000002
00000004	00000004
00000006	00000006
00000008	00000008
0000000a	0000000a
0000000c	0000000c
0000000e	0000000e
00000010	00000010
00000012	00000012
00000014	00000014
00000016	00000016
00000018	00000018
0000001a	0000001a
0000001c	0000001c
0000001e	0000001e
00000001	00000001
00000003	00000003
00000005	00000005
00000007	00000007
0000000b	0000000b
0000000d	0000000d
0000000f	0000000f
00000011	00000011
00000013	00000013
00000015	00000015
00000017	00000017
00000019	00000019
0000001b	0000001b
0000001d	0000001d
0000001f	0000001f

Figura 7.3 - Banco de registradores após a instrução 0x0



State of the register bank after executing instruction 0x2. The register \$0x9 (index 9) now contains the value 0xB.

Address (hex)	Data (hex)
00000000	00000000
00000002	00000002
00000004	00000004
00000006	00000006
00000008	00000008
0000000a	0000000a
0000000c	0000000c
0000000e	0000000e
00000010	00000010
00000012	00000012
00000014	00000014
00000016	00000016
00000018	00000018
0000001a	0000001a
0000001c	0000001c
0000001e	0000001e
00000001	00000001
00000003	00000003
00000005	00000005
00000007	00000007
0000000b	0000000b
0000000d	0000000d
0000000f	0000000f
00000011	00000011
00000013	00000013
00000015	00000015
00000017	00000017
00000019	00000019
0000001b	0000001b
0000001d	0000001d
0000001f	0000001f

Figura 7.4 - Banco de registradores após a instrução 0x2

Addi - Add Immediate

A instrução utilizada foi `addi $0x9 $0x9 0x2` e a simulação está correta, pois nota-se na figura do banco de registradores que após a instrução ser executada foi somado valor imediato 0x2 com o conteúdo do registrador \$0x9, e foi escrito tal resultado nesse mesmo registrador destino (inicialmente continha o valor 0x9) e após a operação contém o valor de 0xB. Portanto, a soma está correta e a escrita no registrador também.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	21290002	00000000	00000000	00000000	00000000	00000000	00000000	00000000
008	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figura 8.1 - addi.mif

Address	Data
00000000	00000000
00000004	00000004
00000008	00000009
0000000c	0000000c
00000010	00000010
00000014	00000014
00000018	00000018
0000001c	0000001c

Figura 8.2 - Banco de registradores inicialmente

Address	Data
00000000	00000000
00000004	00000004
00000008	0000000b
0000000c	0000000c
00000010	00000010
00000014	00000014
00000018	00000018
0000001c	0000001c

Figura 8.3 - Banco de registradores após a operação

Observações:

Como as instruções *and*, *or*, *sll*, *slt* e *srl* são do tipo r-format, as imagens das waves foram omitidas, para o relatório não ficar extenso e porque os formatos de ondas são parecidas, apenas mudando o AluResult e o AluOp, e os sinais RegDst e RegWrite ficam ativo alto para indicar escrita no banco de registradores. Em caso de dúvidas, as imagens estarão no Apêndice.

4. Vetor - Implementação e Simulação

O vetor que foi utilizado para esta atividade foi o seguinte:

6	2	3	5	7	4
---	---	---	---	---	---

Para a execução desta etapa foram utilizados, como auxiliares, os registradores \$0x8 - \$0xE, em que:

- \$0x8 - recebe sempre o valor do vetor lido da memória.
- \$0x9 - recebe o tamanho do vetor, no caso 6.
- \$0xA - guarda o somatório.
- \$0xB - guarda o valor mínimo.
- \$0xC - guarda o valor máximo
- \$0xD - contém o contador para a quantidade de posições lidas.
- \$0xE - recebe apenas 1 ou 0 para comparações de mínimo e máximo.

Abaixo seguem as imagens das instruções, estado dos registradores e memória; comentários mais detalhados em relação a cada operação estão presentes no arquivo **vetor.mif**.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
000	20090006	AC090000	20090002	AC090001	20090003	AC090002	20090005	AC090003
008	20090007	AC090004	20090004	AC090005	00000000	20090006	00006820	8DA80000
010	11A90014	11A00008	010B702A	15C0000B	010C702A	11C0000C	01485020	21AD0001
018	0800000F	00000000	00005020	00085820	00086020	08000012	00000000	00085820
020	08000014	00000000	00086020	08000016	00000000	00000000	00000000	00000000
028	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
030	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figura 9 - vetor.mif

00000035	0	0
00000033	0	0
00000031	0	0
0000002f	0	0
0000002d	0	0
0000002b	0	0
00000029	0	0
00000027	0	0
00000025	0	0
00000023	0	0
00000021	0	0
0000001f	0	0
0000001d	0	0
0000001b	0	0
00000019	0	0
00000017	0	0
00000015	0	0
00000013	0	0
00000011	0	0
0000000f	0	0
0000000d	0	0
0000000b	0	0
00000009	0	0
00000007	0	0
00000005	4	7
00000003	5	3
00000001	2	6

Address: hexadecimal Data: decimal

Figura 10 – Memória

00000000	0
00000001	1
00000002	2
00000003	3
00000004	4
00000005	5
00000006	6
00000007	7
00000008	8
00000009	6
0000000a	10
0000000b	11
0000000c	12
0000000d	13
0000000e	14
0000000f	15
00000010	16
00000011	17
00000012	18
00000013	19
.....	--

Address: hexadecimal Data: decimal

Figura 11 - Banco de registrador inicial

00000000	0
00000001	1
00000002	2
00000003	3
00000004	4
00000005	5
00000006	6
00000007	7
00000008	0
00000009	6
0000000a	27
0000000b	2
0000000c	7
0000000d	6
0000000e	1
0000000f	15
00000010	16
00000011	17
00000012	18
00000013	19
.....	--

Address: hexadecimal Data: decimal

Figura 12 - Banco de registradores após a execução do programa

Primeiramente, foi utilizado a instrução *addi* para colocar os valores do vetor em um registrador temporário e a instrução *store* para escrever na memória esses valores, a Figura – 10 mostra o estado da memória após a escrita de todos os valores.

Adiante, é feito um loop que irá percorrer o vetor na memória. Em cada execução do laço é carregado o valor da memória com a instrução *lw*, são feitas as devidas comparações com *beq*, *bne* e *slt* para achar o valor máximo e mínimo, é usado o *beq* para

sair do loop no momento que o registrador \$0xD for igual ao \$0x9 (possui o tamanho do vetor), é somado em \$0xA o valor da posição atual do vetor, e é incrementado com o *addi* o valor do registrador \$0xD que faz papel de contador.

No final, temos os valores indicados na Figura 12 - Banco de registradores após a execução do programa. Dessa forma, o programa e as instruções usadas nele estão corretas, pois no fim temos os valores esperados nos registradores \$0x8 ao \$0xE. Observação, a imagem das *waves* está no Apêndice porque foram usadas muitas instruções e o como foram muitos ciclos de clock, não cabe em uma única imagem.

5. Construção do Projeto

O projeto pode ser reconstruído adicionando ao *Quartus* todos os arquivos de tipo VHD. Para a simulação geral das instruções, adicione também o arquivo *program.mif*, no qual, há instruções em código de máquina para verificar o funcionamento de todas as novas instruções criadas. Caso queira simular as instruções separadamente, ao invés do *program.mif*, adicione o arquivo de formato *.mif* cujo nome seja da instrução de interesse (por exemplo, para a instrução *bne*, deve-se adicionar o arquivo *bne.mif*). Caso queira a simulação do programa em *assembly* que realiza operações a partir de um vetor, adicione o arquivo *vetor.mif*. Em seguida, é necessário importar o arquivo de tipo *.csv* que mapeia os pinos da placa corretamente, visto que foi adicionado uma chave que funciona como entrada de um multiplexador que seleciona o dado a ser apresentado no *display*. A partir daí, já é possível executar a simulação normalmente. Todas as outras convenções adotadas são equivalentes a aquelas passadas em sala de aula.

6. Considerações Finais

A partir das simulações é possível observar que os resultados estão coerentes com o funcionamento esperado. Devido ao formato consistente e regular das instruções MIPS, foi possível sua descrição em VHDL de maneira factível, apesar de algumas dificuldades encontradas.

Inicialmente, houve certo embaraço em relação à linguagem VHDL, pois esta consiste em uma linguagem de baixo nível e não sequencial. O restante dos contratempos baseou-se na adaptação da lógica para a implementação de algumas instruções, visto que este projeto é uma versão simplificada no processador MIPS e há limitações e distinções em comparação com o formato padrão proposto originalmente. Todos os obstáculos foram contornados, com auxílio de livros e materiais passados em aula [1][2].

Além das mudanças no *Ifetch*, *Idecode*, *Control* e *Execute*, a entidade top level também foi alterada com relação ao último experimento entregue. Foram adicionados sinais internos para o auxílio dos *port maps*, os quais também foram modificados para a integração dos módulos. Ademais, foram acrescentados uma chave e um multiplexador para selecionar a saída no display da placa.

Desse modo, com os módulos *Ifetch*, *Idecode*, *Control* e *Execute*, a entidade top level, a memória de dados *Dmemory* e a memória de instrução *program.mif* foi realizada a descrição do processador MIPS simplificado, de modo a cumprir com objetivo do projeto.

Referências

- [1] D'Amore, R. VHDL: Descrição e Síntese de Circuitos Digitais. LTC. 2005.
- [2] Hamblen, J.O; Hall, T.S. & Furman, M.D – Rapid Prototyping of Digital Systems: SOPC Edition. Springer, 2008.

Apêndice A

As waves das simulações das operações *sll*, *srl*, *sllt*, *addi*, assim como a simulação completa do exercício proposto relacionado à criação e leitura do vetor encontram-se abaixo. Vale ressaltar que devido a extensa simulação do exercício, as *waves* encontram-se separadas em 4 figuras distintas.

SLL

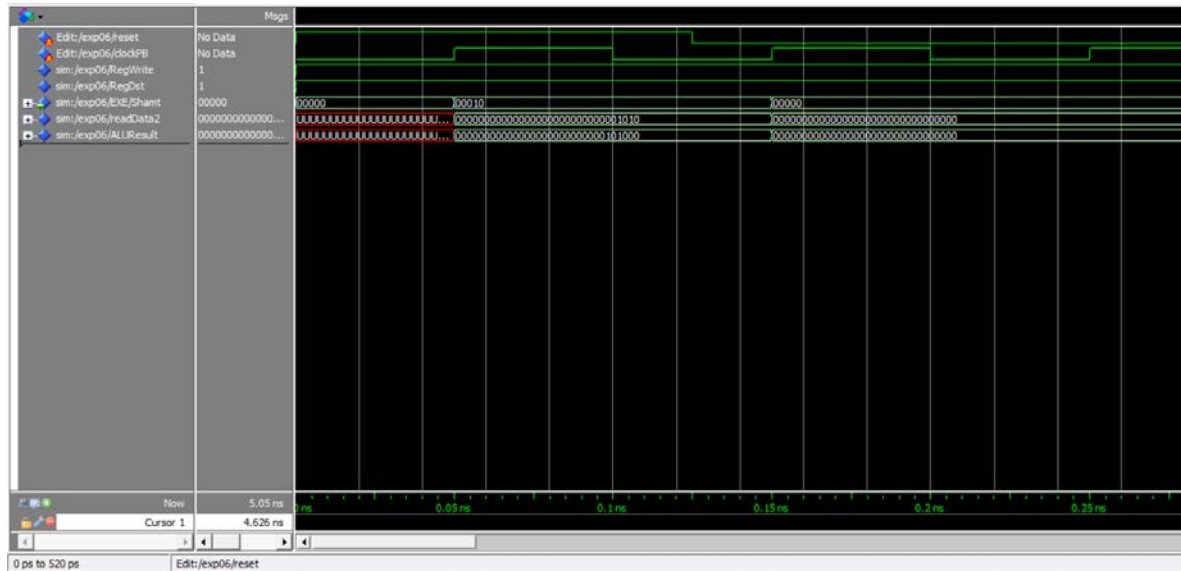


Figura 13 - Simulação da instrução SLL

SRL

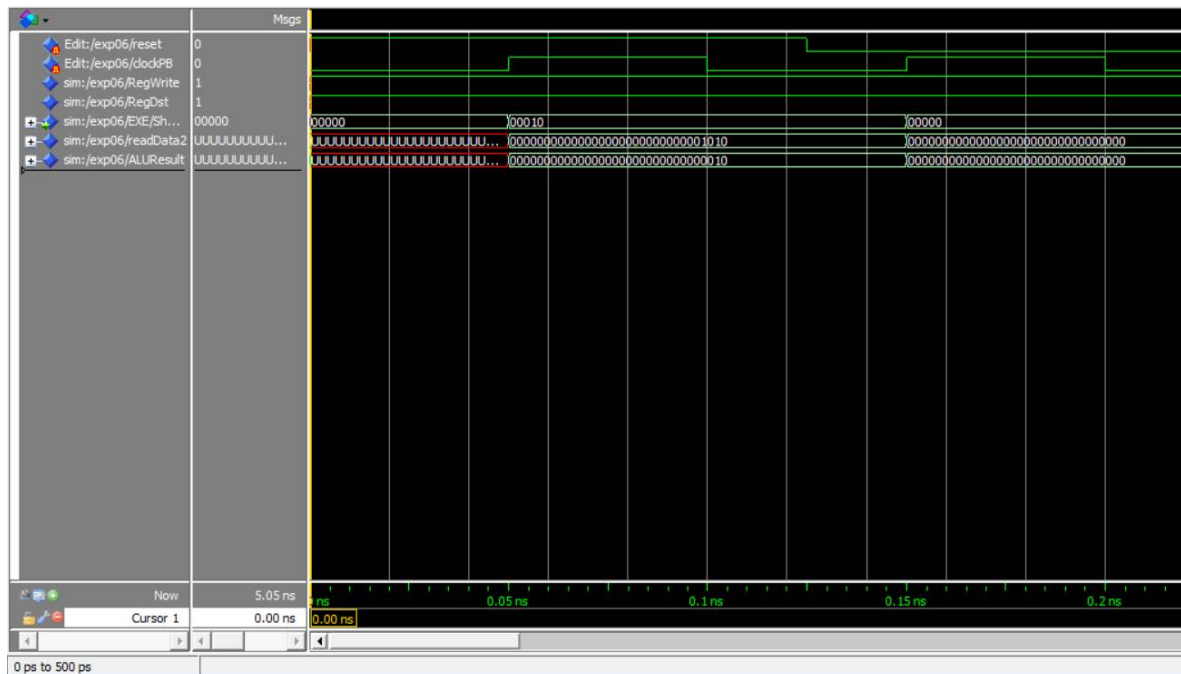


Figura 14: Simulação da instrução SRL

SLT

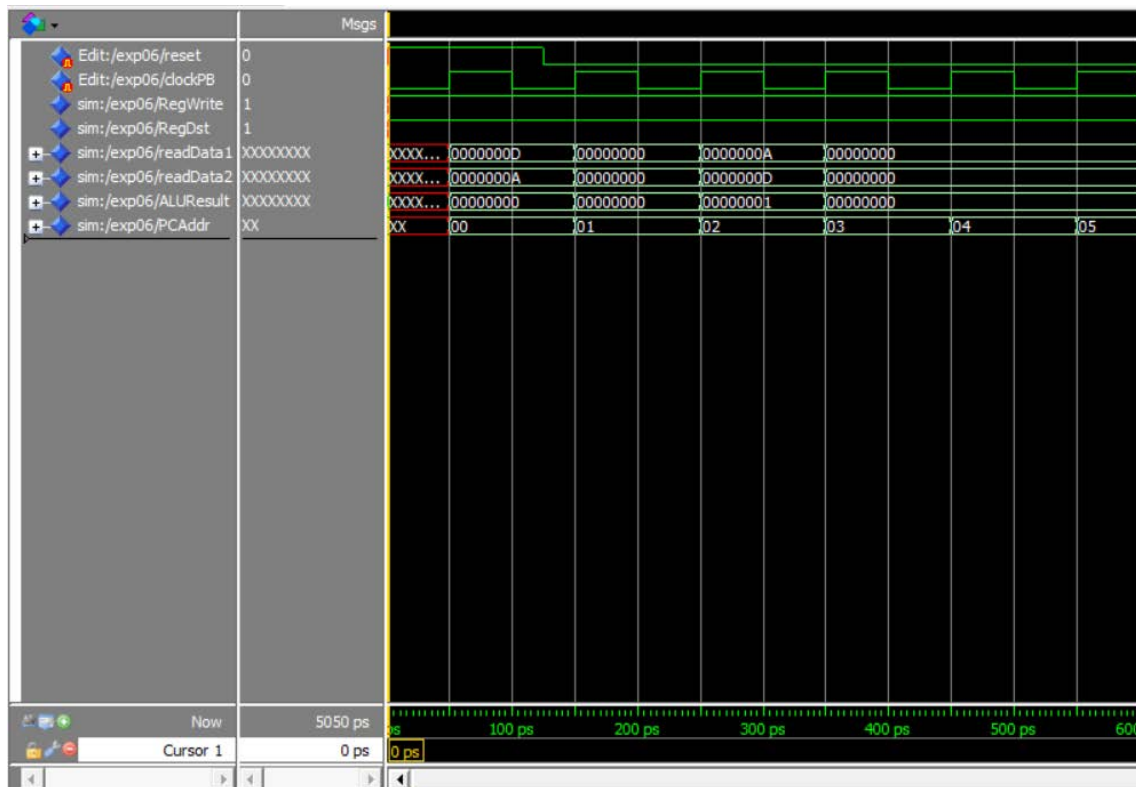


Figura 15- Simulação da instrução SLT

ADDI

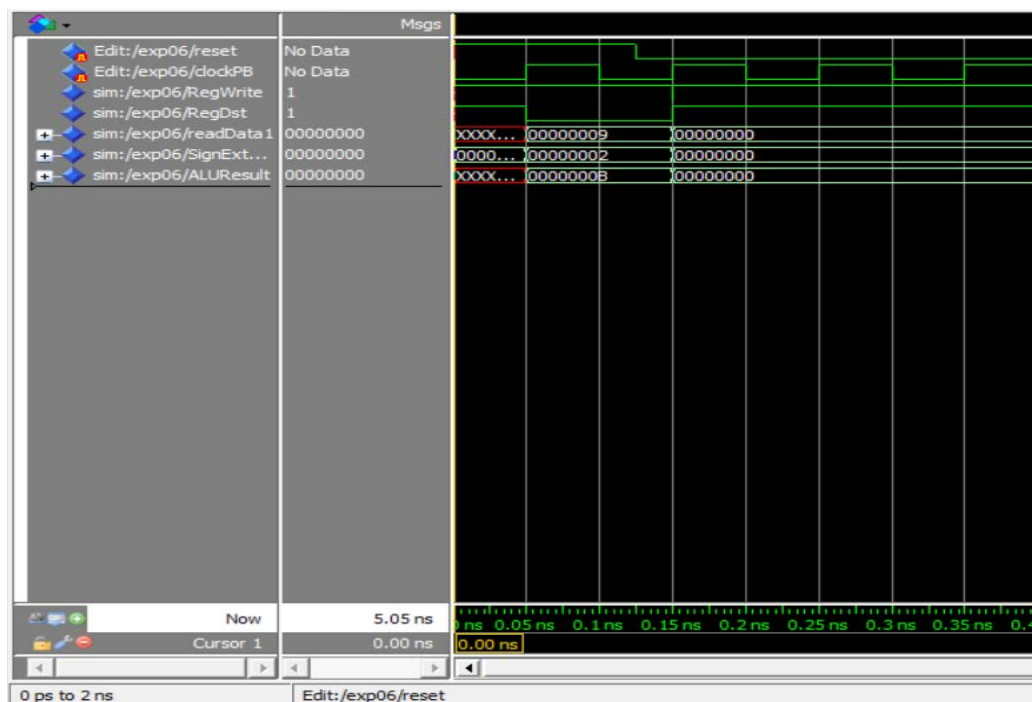


Figura 16: Simulação da instrução ADDI

Vetor

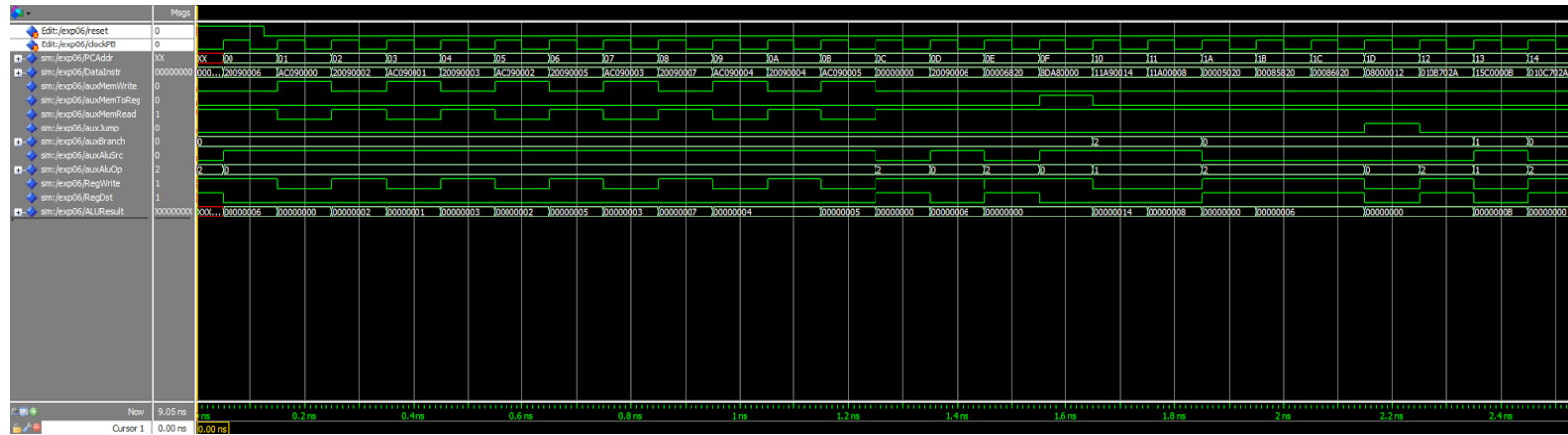


Figura 17 - Simulação do programa (1)

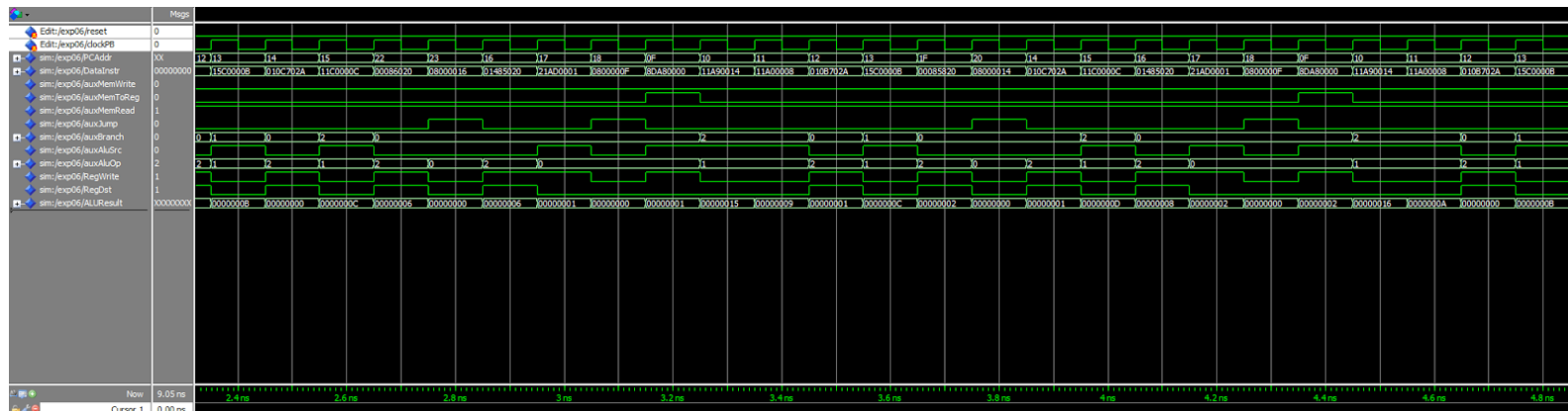


Figura 18 - Simulação do programa (2)

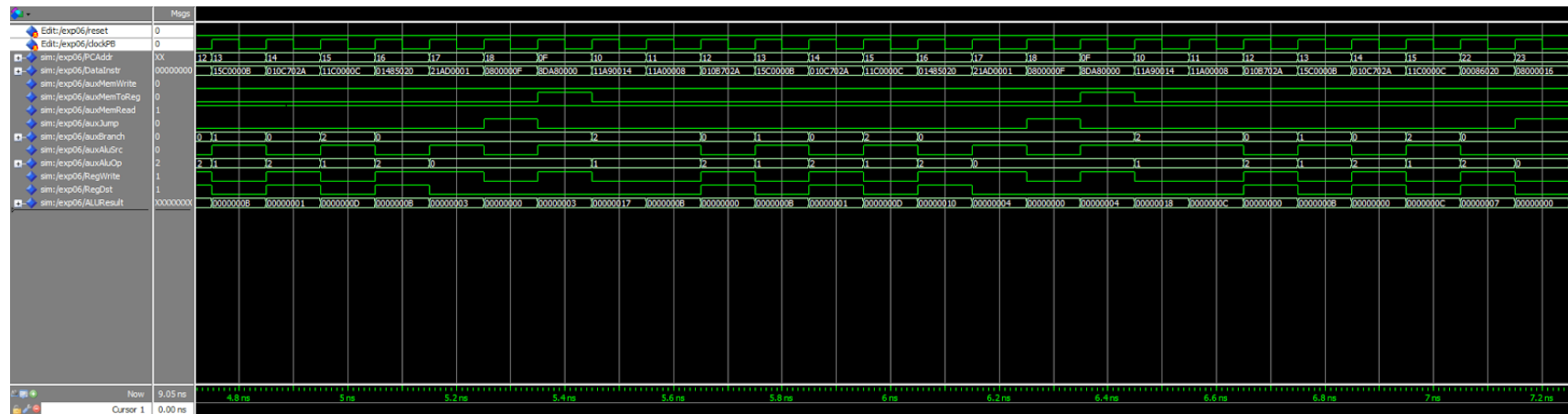


Figura 19 - Simulação do programa (3)

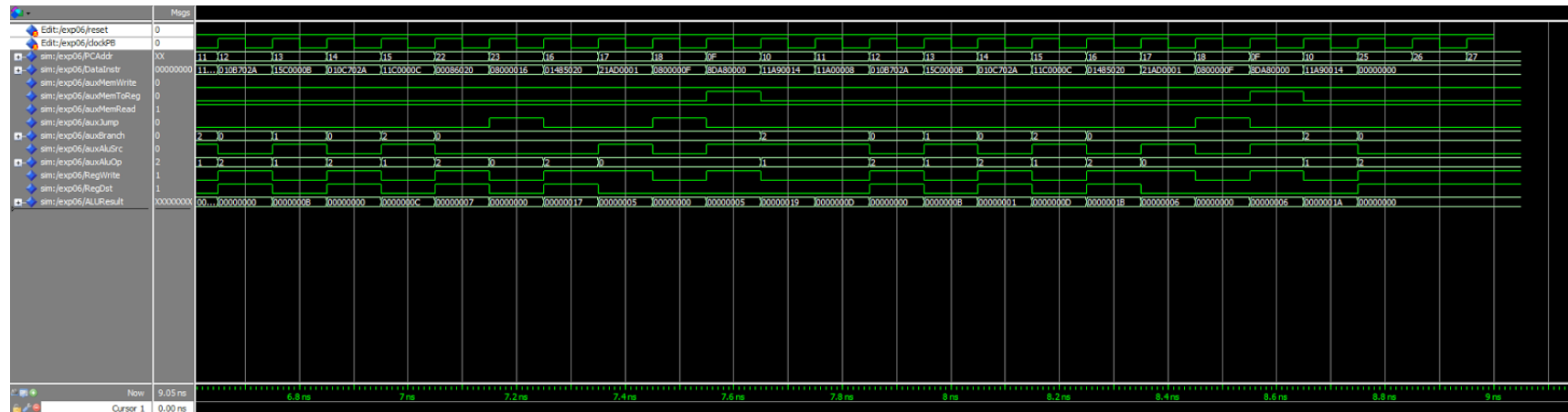


Figura 20 - Simulação do programa (4)