Universidade de Brasília

Departamento de Ciências da Computação – CIC

Organização e Arquitetura de Computadores

Relatório do Projeto Final

MIPS multiciclo

Para a implementação do MIPS multiciclo, foi necessário elaborar os módulos **ALU** (Arithmetic Logic Unit, ou em português Unidade Lógica e Aritmética), **BREG** (Banco de Registradores), **RAM** (a memória do MIPS multiciclo) e o **cntrMIPS** (Controle do MIPS multiciclo) separadamente. Além de serem associados utilizando o *port map* na entidade principal, chamada **MIPS_multiciclo**.

O módulo **ALU.vhd** (Arithmetic Logic Unit, ou em português Unidade Lógica e Aritmética) contém uma entidade, que por sua vez, possui os seguintes sinais como entrada: o código da operação (*operation*) e os operandos (*A* e *B*). E os sinais de saída são o resultado (*Z*), *zero* (ativo quando os operandos são iguais), o indicador de overflow (*ovfl*) e o indicador do último carry out (*vai*).

A execução das operações da ALU é feita a partir do código de operação, ou seja, dependendo do sinal *operation* pode ser feita as operações soma (*add*), subtração (*sub*), and, or, nor, xor, shift left logical (*sll*), shift right logical (*srl*), shift reight arithmetic (*sra*) e set on less than (*slt*).

O módulo **BREG.vhd** (Banco de Registradores) contém uma entidade, que por sua vez, possui os seguintes sinais como entrada: o sinal do clock(*clk*), o habilitador de escrita no banco de registradores (*EscreveReg*), os endereços dos registradores a serem lidos (*radd1* e *radd2*), o endereço do registrador a ser escrito (*wadd*) e o conteúdo a ser escrito (*wdata*). E os sinais de saída são o conteúdo dos registradores lidos (*r1* e *r2*).

A escrita no banco de registradores só é realizada no flanco de subida do clock e se o sinal *EscreveReg* estiver em ‘1’. Enquanto isso, a leitura é realizada independente do sinal do clock, ou seja, a partir do momento que é fornecido os endereços dos registradores a serem lidos (*radd1* e *radd2*) é obtido o conteúdo dos registradores lidos em (*r1* e *r2*).

O módulo **RAM.vhd** (a memória do MIPS multiciclo) foi implementado utilizando a ferramenta MegaWizard Plug-In Manager do Quartus II – Web Edition, versão 13.0 da ALTERA. A entidade possui os seguintes sinais de entrada: o endereço ou índice da memória (*address*), o sinal de clock (*clock*), o conteúdo a ser escrito na memória (*data*) e o habilitador de escrita na memória (*wren*). Como sinal de saída, contém apenas o conteúdo lido da memória (*q*).

O módulo **RAM.vhd** faz uso de um arquivo de inicialização da memória em formato hexadecimal, chamado apresentacao.mif, esse arquivo contém a memória de código e a memória de dados. Além de possuir 256 posições de memória disponíveis,

onde as primeiras 128 posições são destinadas ao código e o restante aos dados. Cada posição de memória, ou seja, cada palavra possui o tamanho de 32 bits (ou se preferir 4 bytes).

Na implementação real do MIPS multiciclo, o endereço de acesso a memória tanto interna quanto externa é múltiplo de 4. Entretanto o módulo **RAM.vhd**, que implementa a memória de código e dados do MIPS, possui apenas 256 posições. Então para contornar esse problema, o cálculo do endereço de acesso das instruções é obtido da seguinte forma:

```
address <= outPC1(9 downto 2);
```

Já o endereço de acesso a memória de dados é obtida da seguinte forma:

```
address <= ('1' & sALU(8 downto 2));
```

Onde o sinal *sALU* é o resultado da ALU armazenado no registrador Saída da ALU. Dessa forma, a ALU calcula o endereço de acesso da memória de dados e é concatenado com '1', pois desse modo irá acessar a parte de dados do arquivo **apresentacao.mif**, contida no endereço maior ou igual a 128.

O módulo **cntrMIPS.vhd** (Controle do MIPS multiciclo), possui como sinais de entrada: o sinal do clock (*clk*), *reset* (serve para forçar a máquina a começar de um estado conhecido, chamado *Fetch0*), e o sinal de *Opcode* da instrução. E como sinais de saída, possui os sinais de controle do MIPS: *OpALU, OrigBALU, OrigAALU, OrigPC, EscreveReg, RegDst, MemparaReg, EscrevePC, EscrevePCCond, IouD, EscreveMem, EscreveRI* e por último o sinal chamado *state*, que indica em qual estado a máquina se encontra.

O controle do MIPS multiciclo foi implementado utilizando uma máquina de estados finitos, onde em sua arquitetura possui dois processos. O primeiro, chamado de síncrono, força a máquina a começar de um estado conhecido, pois quando a máquina é ligada ela pode oscilar e começar de um estado indefinido. Então para resolver esse problema, foi definido o sinal de entrada *reset*, caso esteja desativado em '0' força a máquina a começar de um estado definido chamado *Fetch0*. Além disso, no mesmo processo caso haja o flanco de subida do clock, o próximo estado (*PE*) é atribuído ao estado presente (*EP*). O segundo processo utilizado em sua arquitetura, chamado de combinacional, apenas define como serão os sinais de saída para cada estado, além de definir qual será o próximo estado.

Durante os testes feitos na implementação do **MIPS multiciclo**, percebemos que há um atraso entre o controle e a leitura/escrita da memória **RAM** e do **BREG**. Então para contornar esse problema, foram duplicados cada estado de execução das instruções no **cntrMIPS**, pois a mudança de estado ocorre no flanco de subida do clock e dessa forma, precisávamos manter os sinais de controle inalterados durante um certo tempo.

Além disso, foram feitos alguns adiantamentos de dados na arquitetura da entidade principal, chamada **MIPS_multiciclo.vhd**, com relação ao endereço lido da memória (o

sinal *address*) para as instruções load word (lw), store word (sw), branch if equal (beq), branch if not qual (bne), jump (j) e jump and link (jal). Para que funcionassem corretamente.

Na entidade principal **MIPS_multiciclo.vhd** além da associação dos módulos já descritos anteriormente, utilizando a ferramenta *port map*. Foram criados processos com lista de sensitividade para simular os multiplexadores de 2 para 1 e de 4 para 1 existentes na arquitetura principal. Além disso, foi criado um processo que simula o controle da ALU, que decide qual operação será realizada pela ALU por meio do código de operação.

O **controle da ALU** é necessário porque as instruções do tipo R, possuem o mesmo *Opcode* e o que as distingue só é o campo *funct*. Então por meio do campo *funct* de cada instrução, é realizada é definida qual código de operação é atribuído ao sinal *operação*, caso o sinal de controle *OpALU* for igual a "10". E dependendo desse código, é realizada uma operação específica na ALU.

O código utilizado nos testes da entidade principal **MIPS_multiciclo.vhd** por meio do TestBench, chamado **MIPS_multiciclo_TB.**vhd é o mesmo utilizado na apresentação e está no arquivo **apresentacao.asm**.

Code	Basic	Source		
0x8c180000	lw $24,0($0)	4:	lw $t8,0($zero)	# $t8 = 4
0x03008020	add $16,$24,$0	5:	add $s0,$t8,$zero	#$s0 = 4
0x20080002	addi $8,$0,2	6:	addi $t0,$zero,2	# 0 + 2 = 2
0x11000016	beq $8,$0,22	7:	beq $t0,$zero, exit	# não executa o salto
0x35090008	ori $9,$8,8	8:	ori $t1,$t0,8	# 8 or 2 = 10
0x01285824	and $11,$9,$8	9:	and $t3,$t1,$t0	# 10 and 2 = 2
0x01286025	or $12,$9,$8	10:	or $t4,$t1,$t0	# 10 or 2 = 10
0x01286027	nor $12,$9,$8	11:	nor $t4,$t1,$t0	# 10 nor 2 = -11
0x01286026	xor $12,$9,$8	12:	xor $t4,$t1,$t0	# 10 xor 2 = 8
0x14000010	bne $0,$0,16	13:	bne $zero,$zero,exit	# não executa o salto
0x01286822	sub $13,$9,$8	14:	sub $t5, $t1, $t0	# 10 - 2 = 8
0x01a9502a	slt $10,$13,$9	15:	slt $t2, $t5, $t1	# 8 < 10 = 1
0x012d502a	slt $10,$9,$13	16:	slt $t2, $t1, $t5	# 10 < 8 = 0
0x000d50c0	sll $10,$13,3	17:	sll $t2, $t5, 3	# 8 << 3 = 64
0x000a68c2	srl $13,$10,3	18:	srl $t5, $t2, 3	# 64 >> 3 = 8
0x11ac0001	beq $13,$12,1	19:	beq $t5, $t4, L1	# Salta para L1 se 8 = 8
0x01087020	add $14,$8,$8	20:	add $t6, $t0, $t0	# nao deve ser executado
0x15a80001	bne $13,$8,1	21: L1:	bne $t5, $t0, L2	# Salta para L2 se 8 != 2
0x010a7020	add $14,$8,$10	22:	add $t6, $t0, $t2	# nao deve ser executado
0x0c000019	jal 100	23: L2:	jal L3	#
0x01097020	add $14,$8,$9	24:	add $t6, $t0, $t1	# 2 + 10 = 12
0xac0e0004	sw $14,4($0)	25:	sw $t6, 4($zero)	# guarda 12 na memoria
0x8c0f0004	lw $15,4($0)	26:	lw $t7, 4($zero)	# recupera 12 da memoria
0x01e0c020	add $24,$15,$0	27:	add $t8,$t7,$zero	# 12 + 0 = 12
0x0800001a	j 104	28:	j exit	# salta para o fim do programa
0x03e00008	jr $31	29: L3:	jr $ra	# retorna para depois do jal

Figura 1: Código nomeado como apresentação.asm

```
-- Quartus II generated Memory Initialization File (.mif)

WIDTH=32;
DEPTH=256;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
    00  :   8c180000;
    01  :   03008020;
    02  :   20080002;
    03  :   11000016;
    04  :   35090008;
    05  :   01285824;
    06  :   01286025;
    07  :   01286027;
    08  :   01286026;
    09  :   14000010;
    0A  :   01286822;
    0B  :   01a9502a;
    0C  :   012d502a;
    0D  :   000d50c0;
    0E  :   000a68c2;
    0F  :   11ac0001;
    10  :   01087020;
    11  :   15a80001;
    12  :   010a7020;
    13  :   0c000019;
    14  :   01097020;
    15  :   ac0e0004;
    16  :   8c0f0004;
    17  :   01e0c020;
    18  :   0800001a;
    19  :   03e00008;
    [1A..7F]    :   00000000;
    80  :   00000004;
    [81..FF]    :   00000000;
END;
```

Figura 2: imagem retirada do arquivo apresentação.mif

A seguir serão mostradas as telas da simulação no ModelSim, mostrando os sinais de saída do registrador PC (*saidaPC*, que armazena o endereço da próxima instrução), a saída do registrador de Dados da Memória (*saidaRDM*), a saída do registrador de Instruções (*saidaRI*) e a saída do registrador que armazena o resultado da ALU (*saidaALU*).

Uma observação importante e que deve ser feita é que a execução de cada instrução só ocorre a partir do momento em que a instrução é carregada no Registrador de Instruções, isso só acontece quando o sinal de controle de escrita nesse registrador *EscreveReg* for ‘1’. Já o Registrador de Dados da Memória é atualizado sempre quando é feita uma leitura da memória e quando ocorre o flanco de descida do clock.

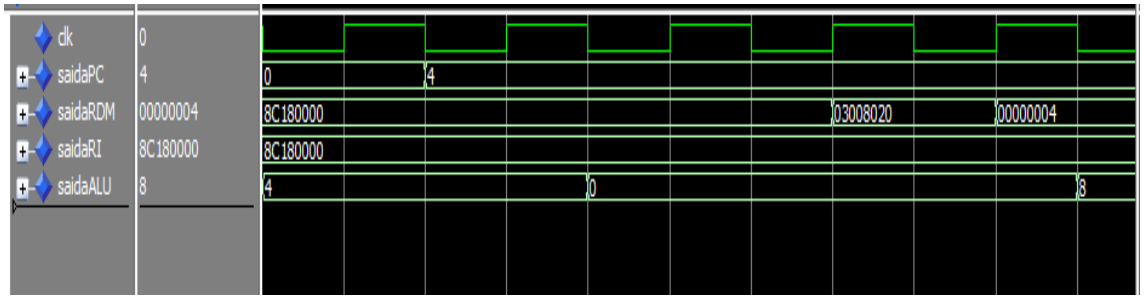

Figura 3: execução da instrução lw $t8, 0($zero)

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, ou seja, o endereço da próxima instrução, além disso no segundo ciclo de clock a saída PC é atualizada com o endereço da próxima instrução. No terceiro ciclo de clock, a saída da ALU armazena o resultado da soma do imediato que é 0 + o conteúdo do registrador 0 = 0. No quinto ciclo de clock, é acessado o endereço de memória calculado pela ALU e o

conteúdo é armazenado no Registrador de Dados da Memória. No sexto ciclo de clock a saída da ALU contém o valor PC + 4.

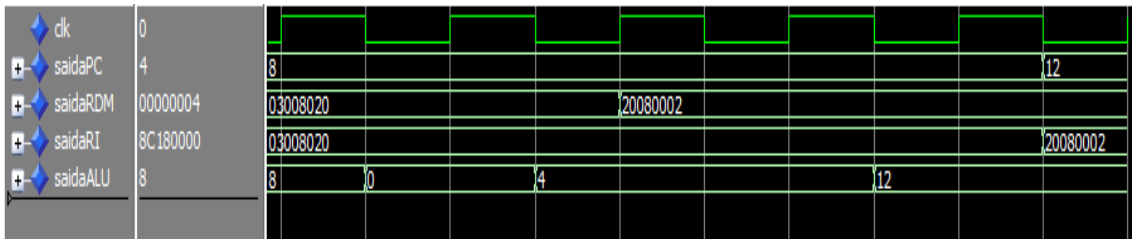

Figura 4: execução da instrução add $s0, $t8, $zero

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. No terceiro ciclo de clock, a saída da ALU armazena o resultado da instrução *add* de 4 + 0. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No quinto ciclo de clock a saída da ALU contém o valor PC + 4 e o registrador PC é atualizado.

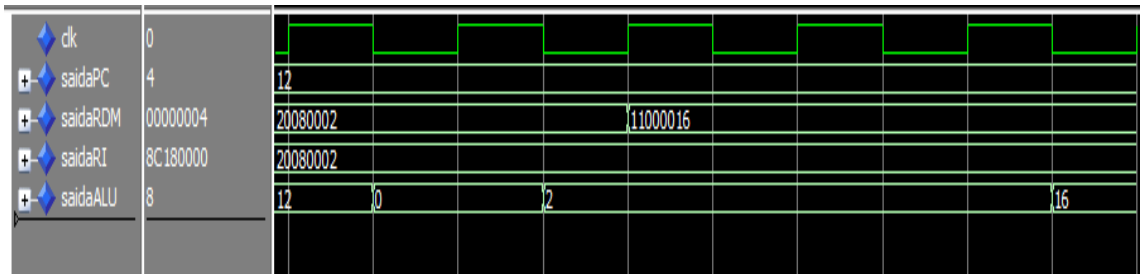

Figura 5: execução da instrução addi $t0,$zero, 2

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, ou seja, o endereço da próxima instrução, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. No terceiro ciclo de clock, a saída da ALU armazena o resultado da instrução *addi* de 2 + 0. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No sexto ciclo de clock a saída da ALU contém o valor PC + 4.

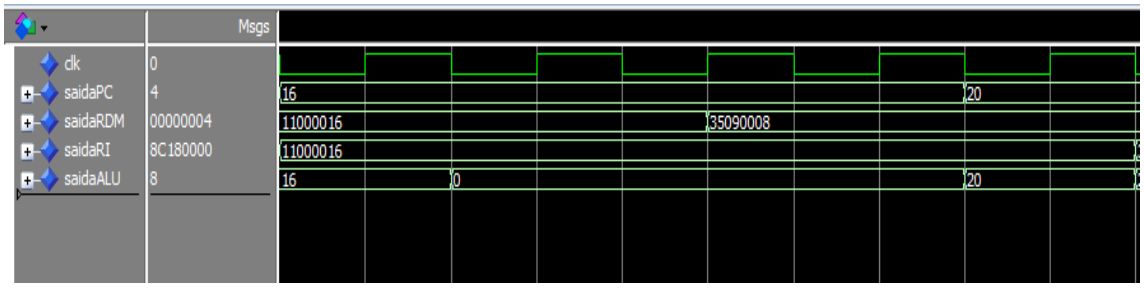

Figura 6: Execução da instrução beq $t0, $zero,exit

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, ou seja, o endereço da próxima instrução, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução de um salto condicional *beq*, onde a condição para o salto é falsa. Dessa forma, executa a próxima instrução. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

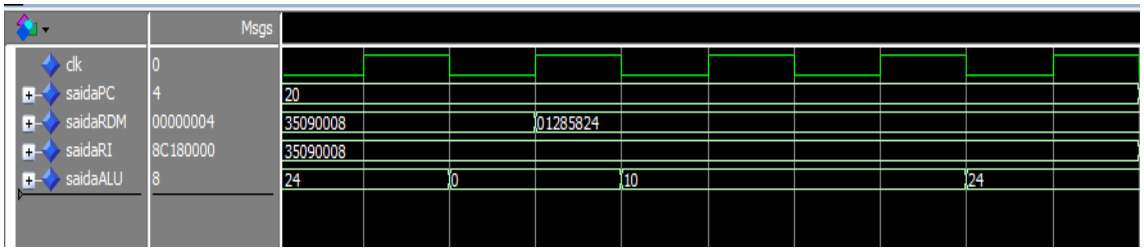

Figura 7: Execução da instrução ori $t1, $t0, 8

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, ou seja, o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *ori*. No segundo ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 2 *ori* 8 = 10. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

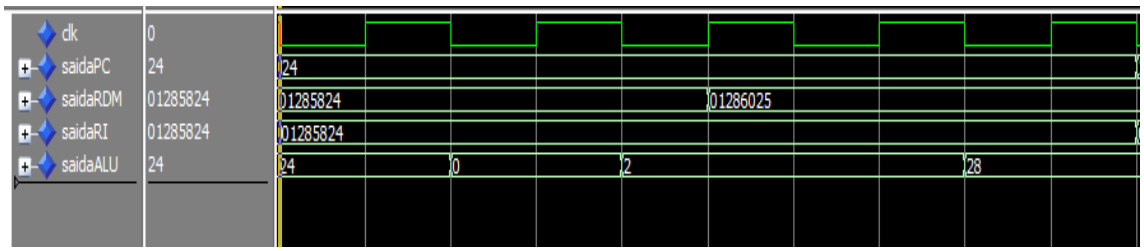

Figura 8: Execução da instrução and $t3, $t1, $t0

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *and*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 10 *and* 2 = 2. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

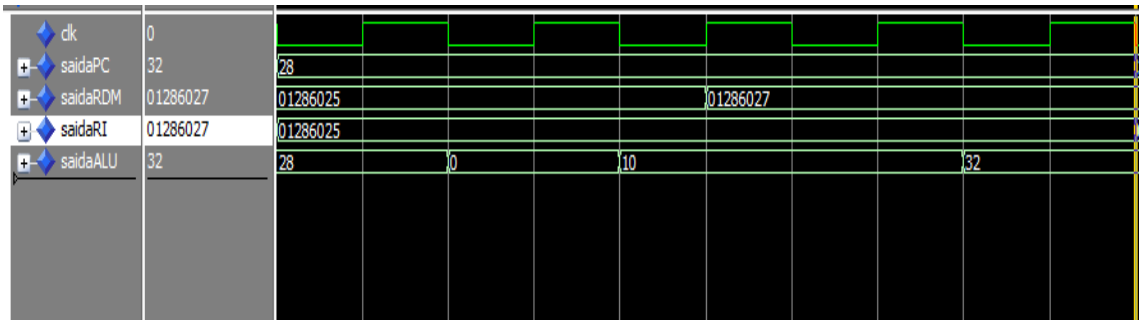

Figura 9: Execução da instrução or $t4, $t1, $t0

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, ou seja, o endereço da próxima instrução, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *or*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 10 *or* 2 = 10. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

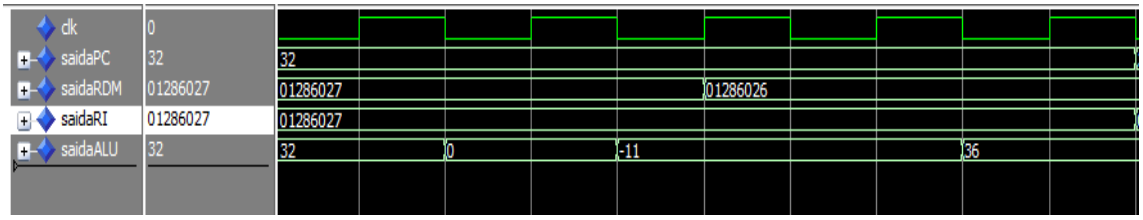

Figura 10: Execução da instrução nor $t4, $t1, $t0

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, , além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *nor*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 10 *nor* 2 = -11. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

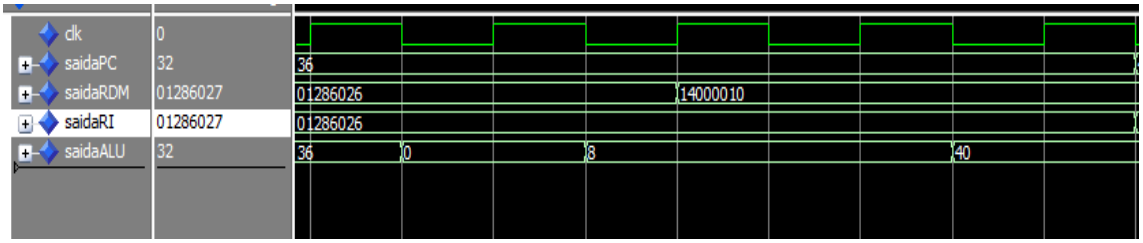

Figura 11: Execução da instrução xor $t4, $t1, $t0

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4 , além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está

indefinida. Essa é a execução da instrução *xor*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 10 *xor* 2 = 8. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

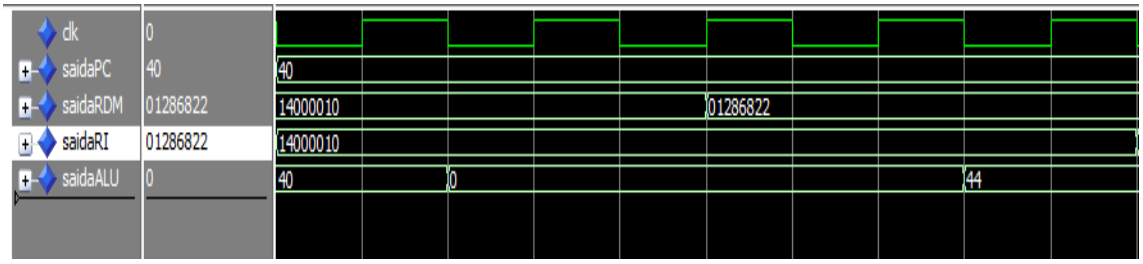

Figura 12: Execução da instrução bne $zero, $zero, exit

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, , além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução de um salto condicional *bne*, onde a condição para o salto é falsa. Dessa forma, executa a próxima instrução. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

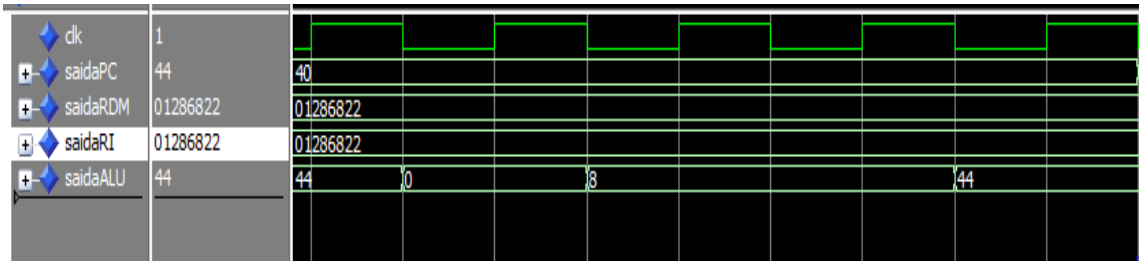

Figura 13: Execução da instrução sub $t5, $t1, $t0

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, ou seja, o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *sub*. No terceiro ciclo de clock, a saída da ALU armazena o resultado da instrução de 10 - 2 = 8. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

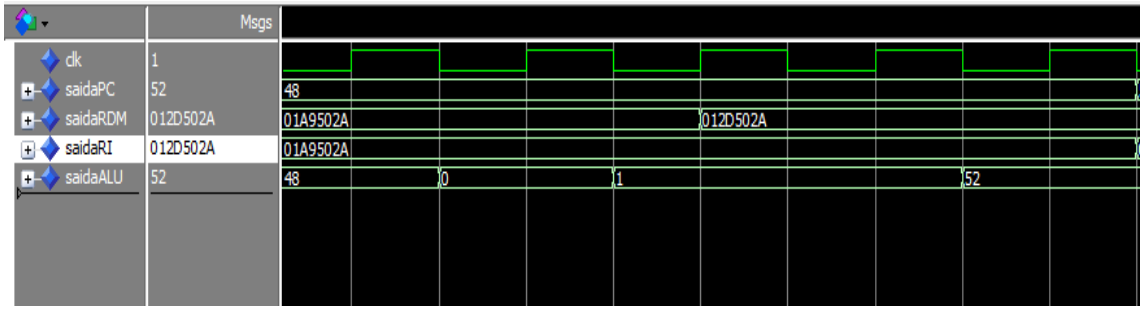

Figura 14: Execução da instrução slt $t2, $t5, $t1

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4 , além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *slt*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado da instrução *slt* e como 8 é menor que 10, então o resultado é 1. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

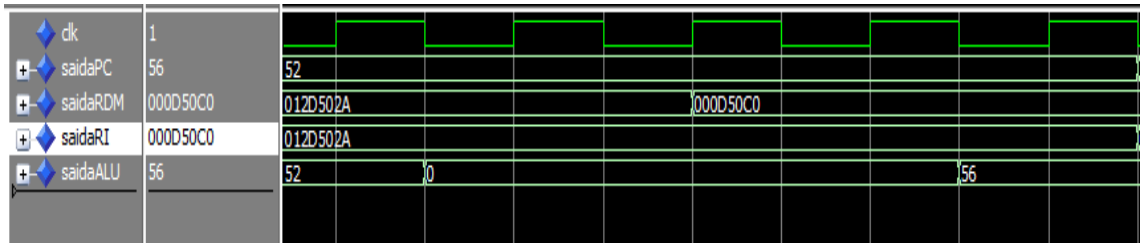

Figura 15: Execução da instrução slt $t2, $t1, $t5

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *slt*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado da instrução *slt* e como 10 não é menor que 8, então o resultado é 0. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

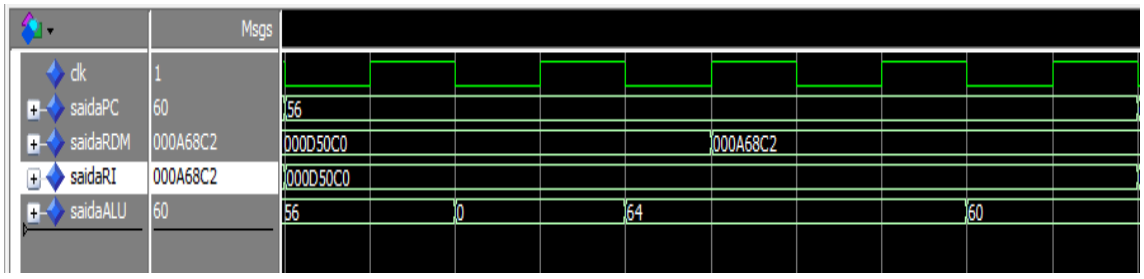

Figura 16: Execução da instrução sll $t2, $t5, 3

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de

clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *sll*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 8 << 3 = 64, pois deslocar 3 bits para a esquerda é equivalente a multiplicar por 8. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

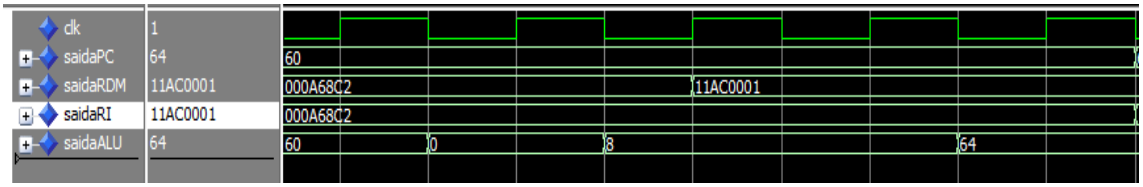

Figura 17: Execução da instrução srl $t5, $t2, 3

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *srl*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 64 >> 3 = 8, pois deslocar 3 bits para a direita é equivalente a dividir por 8. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

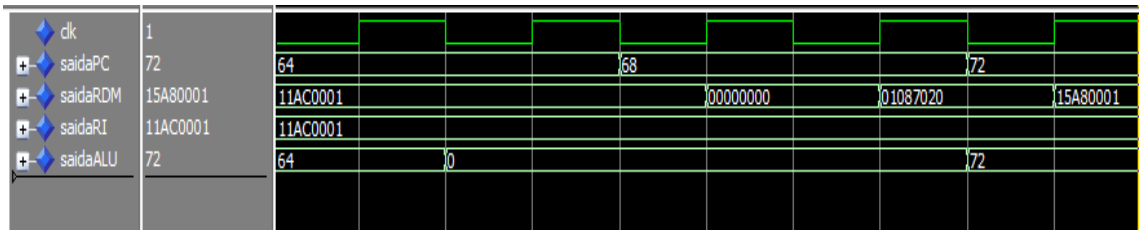

Figura 18: Execução da instrução beq $t5, $t4, L1

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução de um salto condicional *beq*, onde a condição para o salto é verdadeira. Dessa forma, o salto é executado. No terceiro ciclo de clock o registrador PC é atualizado com o endereço do salto. No sexto ciclo de clock, o Registrador de Dados da Memória é atualizado com a instrução armazenada no endereço do salto, além da saída da ALU conter o valor de PC + 4.

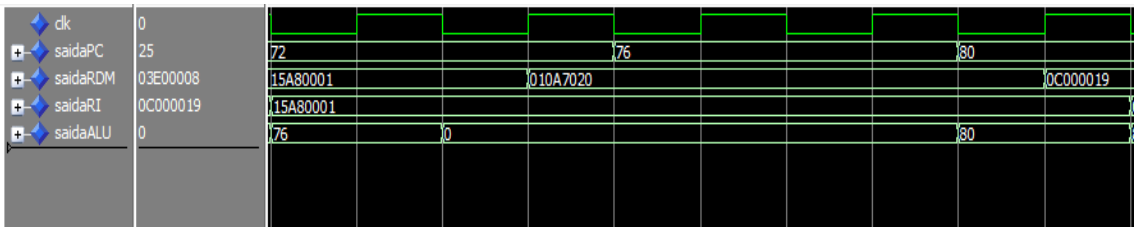

Figura 19: Execução da instrução bne $t5, $t0, L2

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução de um salto condicional *bne*, onde a condição para o salto é verdadeira. Dessa forma, o salto é executado. No terceiro ciclo de clock o registrador PC é atualizado com o endereço do salto. No quinto ciclo de clock, o Registrador de Dados da Memória é atualizado com a instrução armazenada no endereço do salto, além da saída da ALU conter o valor de PC + 4 e atualizar o registrador PC com o endereço da próxima instrução.

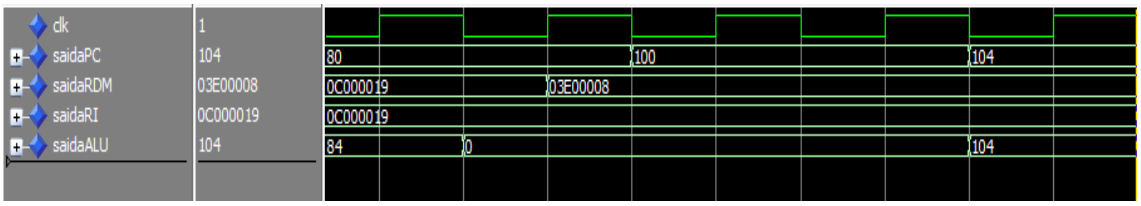

Figura 20: Execução da instrução jal L3

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, ou seja, o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução de um salto incondicional *jal*. Dessa forma, o salto é executado. No segundo ciclo de clock o registrador de Dados da Memória é atualizado com a instrução contida no endereço do salto. No terceiro ciclo de clock o registrador PC é atualizado com o endereço do salto. No quinto ciclo a saída da ALU contém o valor de PC + 4 e atualiza o registrador PC com o endereço da próxima instrução.

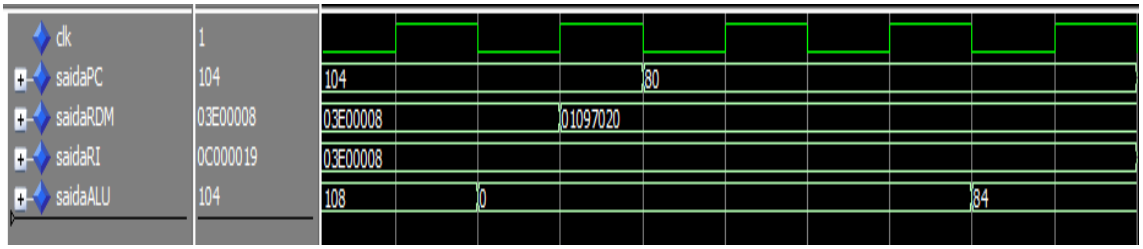

Figura 21: Execução da instrução jr $ra

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, ou seja, o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução de um salto incondicional *jr*, que retorna para a próxima instrução depois de *jal*. Dessa forma, o

salto é executado. No segundo ciclo de clock a instrução contida no endereço do salto é lida da memória e armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock o registrador PC é atualizado com o endereço do salto. No quinto ciclo a saída da ALU contém o valor de PC + 4.

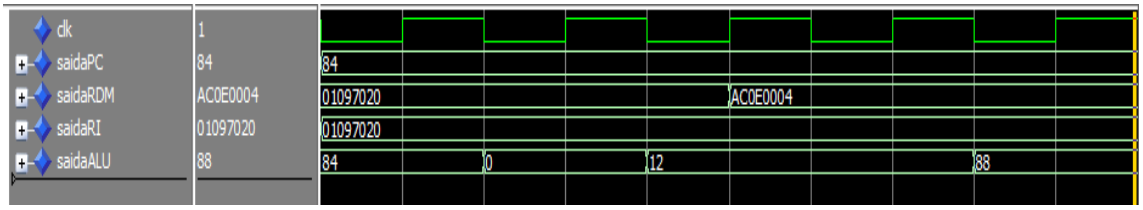

Figura 22: Execução da instrução add $t6, $t0, $t1

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *add*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 2 + 10 = 12. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

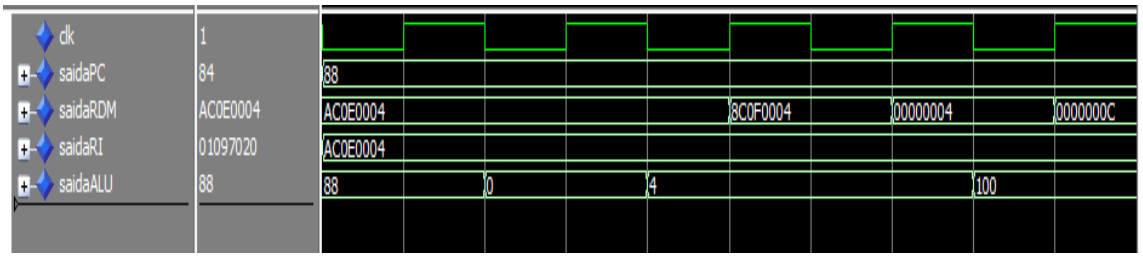

Figura 23: Execução da instrução sw $t6, 4($zero)

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *sw*. No terceiro ciclo de clock, a saída da ALU armazena o resultado da soma do imediato que é 4 + o conteúdo do registrador 0 = 4. No quinto ciclo de clock a saída da ALU contém o valor PC + 4 e o Registrador de Dados da Memória é atualizado com o conteúdo escrito na posição de memória calculada pela ALU, ou seja, é gravada na memória o valor 12.

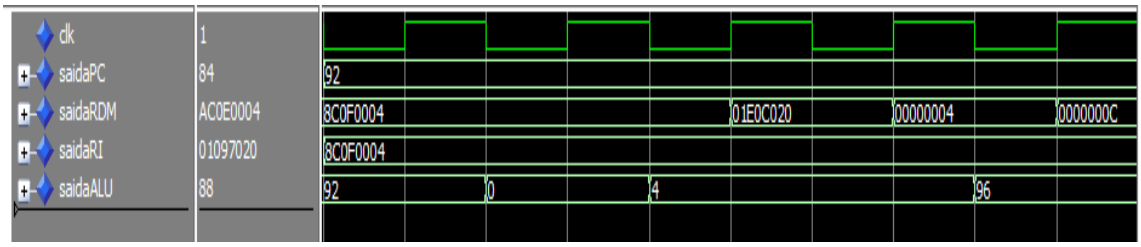

Figura 24: Execução da instrução lw $t7, 4($zero)

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *lw*. No terceiro ciclo de clock, a saída da ALU armazena o resultado da soma do imediato que é 4 + o conteúdo do registrador 0 = 4. No quinto ciclo de clock a saída da ALU contém o valor PC + 4 e o Registrador de Dados da Memória é atualizado com o conteúdo lido da posição de memória calculada pela ALU, ou seja, é lida da memória o valor 12.

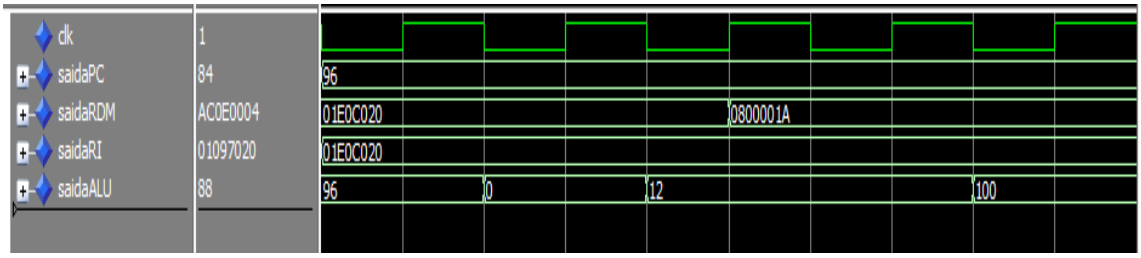

Figura 25: Execução da instrução add $t8, $t7, $zero

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida. Essa é a execução da instrução *add*. No terceiro ciclo de clock a próxima instrução é armazenada no Registrador de Dados da Memória. No terceiro ciclo de clock, a saída da ALU armazena o resultado de 12 + 0 = 12. No quinto ciclo de clock a saída da ALU contém o valor PC + 4.

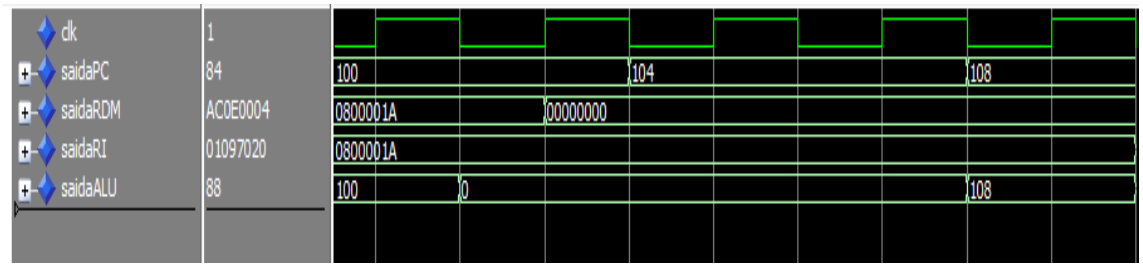

Figura 26: Execução da instrução j exit

No primeiro ciclo de clock a saída da ALU armazena o valor de PC + 4, além disso a saída PC é atualizada com o endereço da próxima instrução. No segundo ciclo de clock, a saída da ALU armazena o valor 0, pois nesse ciclo a operação da ALU está indefinida e nesse mesmo ciclo é carregada a instrução contida no endereço do salto no

registrador de Dados da Memória. Essa é a execução da instrução j. No quinto ciclo de clock a saída da ALU contém o valor PC + 4 e atualiza o registrador PC.

Como a próxima instrução carregada no registrador de Dados da Memória é zero, significa que a execução do código acabou. Eu não fiz condição de parada, mas se continuar executando a simulação então verá que as próxima instruções serão sempre zero.