UNIVERSIDADE DO VALE DO ITAJAÍ

Ciência da Computação
Arquitetura e Organização de Computadores
Douglas Rossi de Melo

Avaliação 5:
Organização do MIPS Monociclo

Fernando Concatto
Itajaí, 14 de junho de 2016

1. Introdução

A atividade desenvolvida consistiu na implementação de três novas instruções em uma organização preexistente do MIPS monociclo, utilizando a ferramenta de design Quartus II, da Altera. As instruções implementadas foram **j** (jump), **jal** (jump and link) e **jr** (jump register); todas elas realizam desvios incondicionais, com algumas características individuais.

2. Implementação

As três novas instruções exigiram alterações tanto no caminho de dados quanto no controle. Foram adicionados três novos sinais: *Jump*, que é ativado nas instruções j e jal; *Link*, ativado apenas na instrução jal; e *JumpReg*, que é acionado na instrução jr.

2.1. Instrução j

Para a implementação da instrução j, a unidade de controle foi alterada para gerar o sinal *Jump* quando o opcode, expressado pelos 6 bits mais significativos da palavra de instrução, possuir a forma

$$\overline{x_5} \wedge \overline{x_4} \wedge \overline{x_3} \wedge \overline{x_2} \wedge x_1$$

onde x_5 é o bit mais significativo e x_0 é o bit menos significativo do opcode. Desta forma, o sinal é gerado tanto pelo código 000010, que indica a instrução j, quanto pelo código 000011, que denota a instrução jal, apresentada na próxima seção.

No caminho de dados, foi adicionado um módulo para realizar a concatenação dos 4 bits mais significativos do PC com os 26 menos significativos da palavra de instrução deslocados duas casas à esquerda. Além disso, foi adicionado um multiplexador para realizar a seleção entre o PC incrementado em 4 e a saída do módulo descrito previamente, com base no sinal *Jump*.

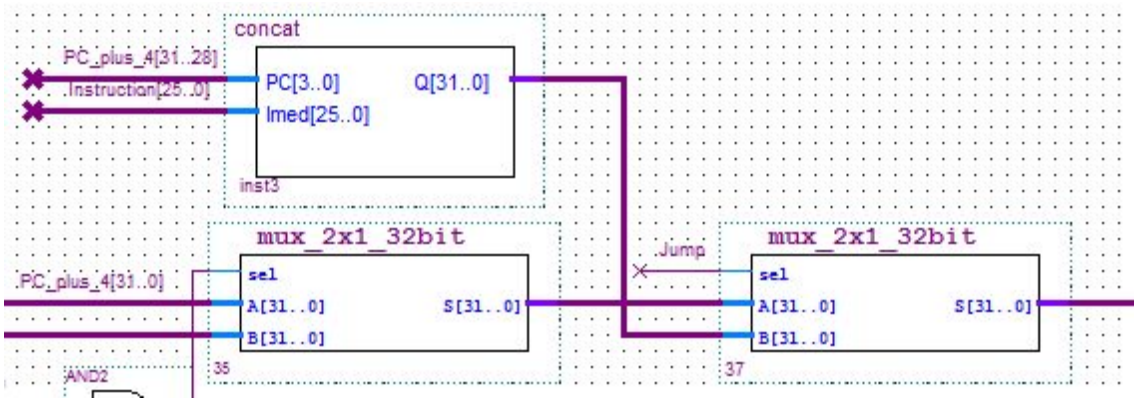

Figura 1. Alterações no caminho de dados: *mux 37* e *concat*

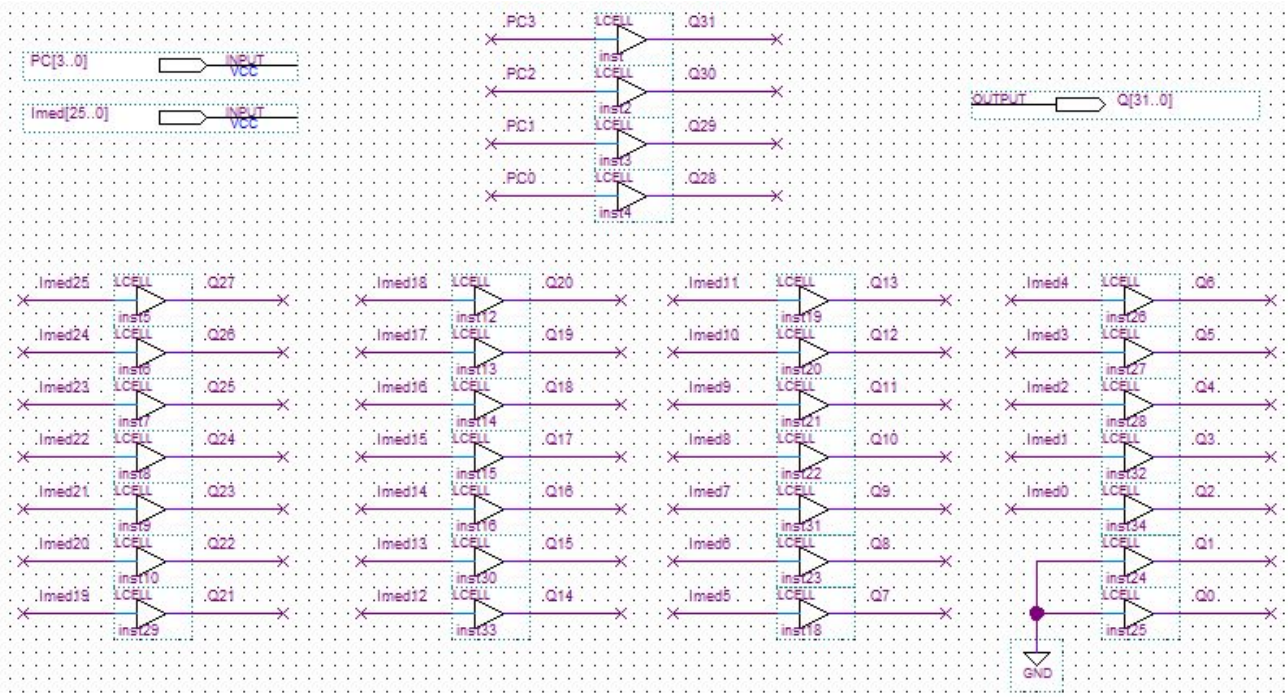

Figura 2. Módulo de concatenação e deslocamento *(concat)*.

2.2. Instrução jal

A instrução jal utiliza exatamente o mesmo caminho da instrução j; é por este motivo que o sinal *Jump* também é ativado por seu opcode. Entretanto, esta instrução possui uma funcionalidade extra: além de desviar o fluxo do programa, o endereço da instrução que seria executada em seguida (PC + 4) é salvo no registrador **$ra**, cujo índice é 31.

Para implementar essa funcionalidade, o sinal *Link* foi adicionado, que é ativado quando o sinal *Jump* e o bit menos significativo do campo opcode estão ligados. A figura 3 apresenta o circuito responsável pelo controle dos sinais *Link* e *Jump*.

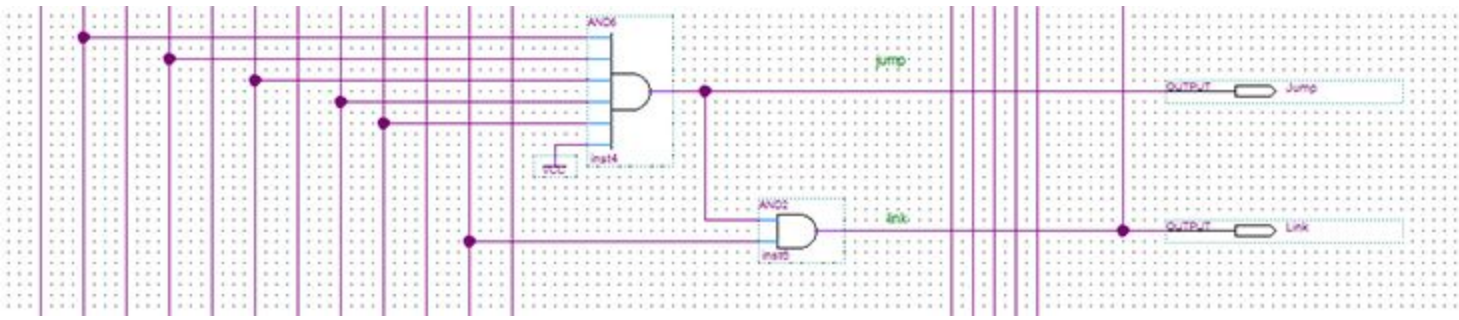

Figura 3. Circuito de controle dos sinais Link e Jump.

O sinal *Link* controla dois multiplexadores que influenciam na escrita no banco de registradores, além de acionar o sinal *RegWrite*, previamente existente, para fazer com que o registrador $ra seja atualizado. O primeiro multiplexador seleciona entre o endereço de escrita padrão, se o sinal *Link* estiver desligado, e a constante 31, que indica o registrador $ra, se estiver ligado; a saída deste multiplexador é conectada com a entrada Wr_Addr do banco de registradores. O segundo controla o dado que será escrito no banco: se o sinal *Link* estiver desligado, o dado padrão é escolhido; se estiver ligado, o valor do PC somado a 4 é selecionado. Sua saída é conectada com a entrada Wr_Data do banco de registradores.

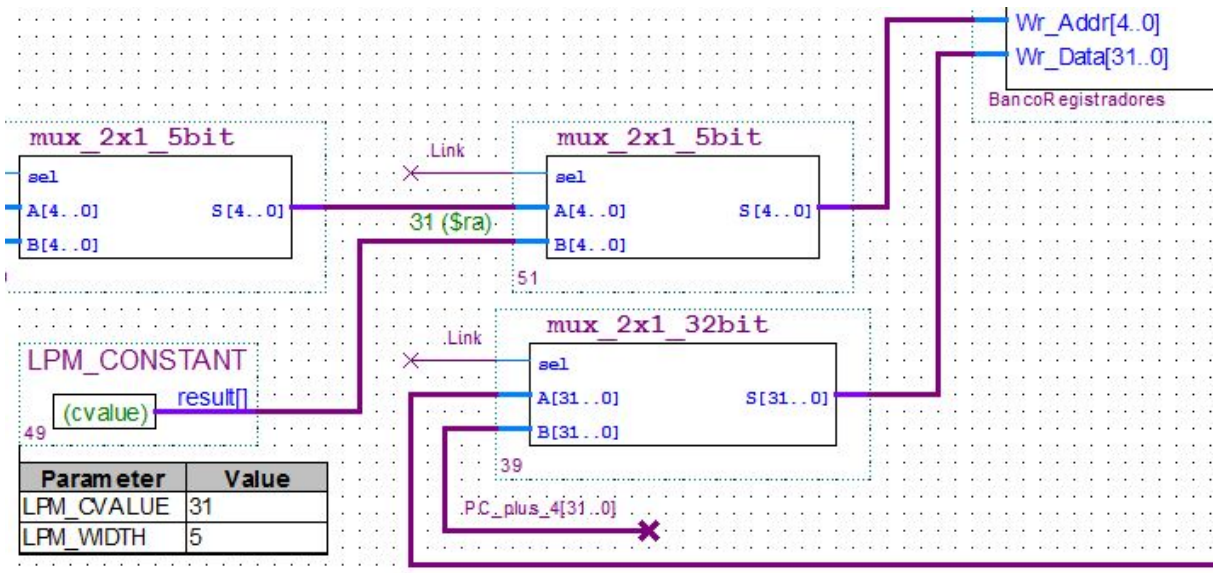

Figura 4. Multiplexadores controlados pelo sinal Link.

2.3. Instrução jr

A instrução Jump Register difere significantemente na implementação em comparação com as duas instruções previamente apresentadas. Ao invés de concatenar o PC com os 26 bits menos significativos da palavra de instrução, a instrução jr substitui o PC diretamente pelo valor presente no registrador indicado pelo campo **rs** da palavra de instrução. Dessa forma, o formato da instrução jr é R, ao contrário das instruções j e jal, que possuem formato J.

A implementação desta instrução envolveu o desenvolvimento do sinal *JumpReg*, que é acionado quando o campo opcode possui o valor 000000, indicando que o formato é R, e quando o campo function, expressado pelos 6 bits menos significativos da palavra de instrução, possuir a seguinte forma:

$$\overline{x_5} \wedge \overline{x_4} \wedge x_3 \wedge \overline{x_2} \wedge \overline{x_1} \wedge \overline{x_0}$$

O circuito lógico que controla este sinal é apresentado na figura 5.

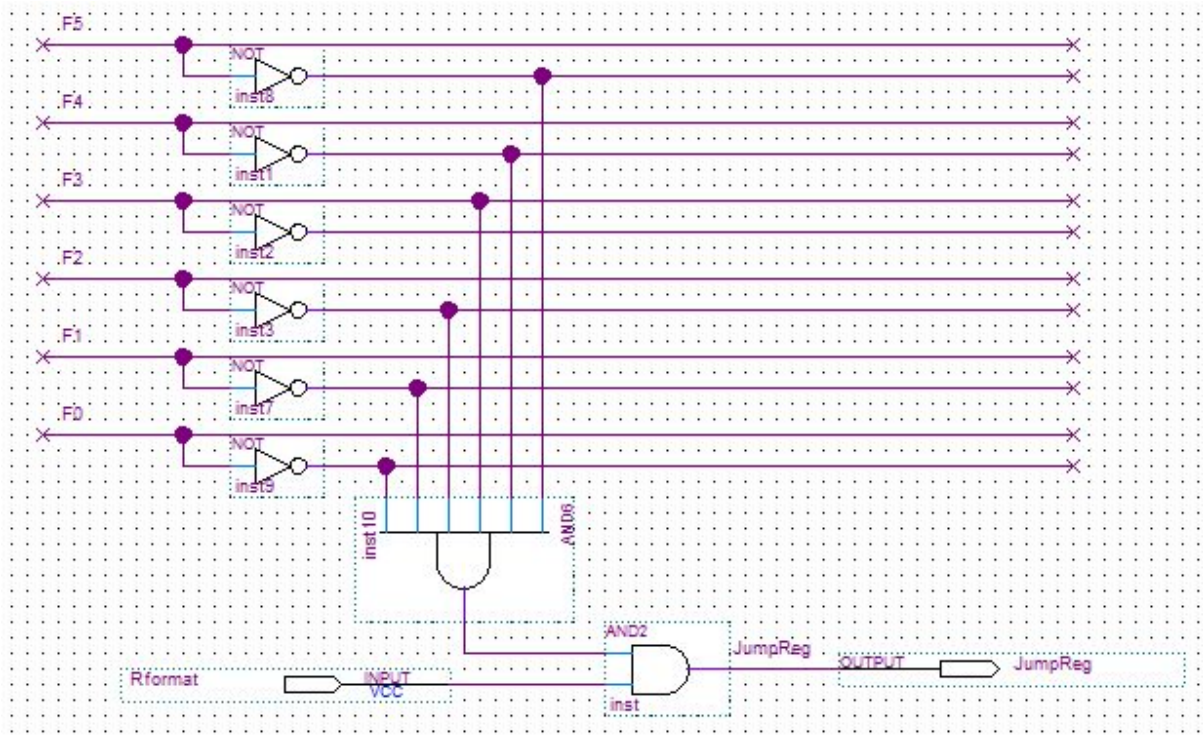

Figura 5. Lógica de controle do sinal JumpReg.

No caminho de dados, a única alteração foi a adição de um multiplexador que influencia o valor do PC. Se o sinal *JumpReg* estiver ativo, a saída do banco de registradores que contém o valor do registrador indicado pelo campo rs (Rd_Data_A) é selecionado, causando o desvio; caso contrário, o nenhuma alteração é feita. Este multiplexador é mostrado na figura 6.

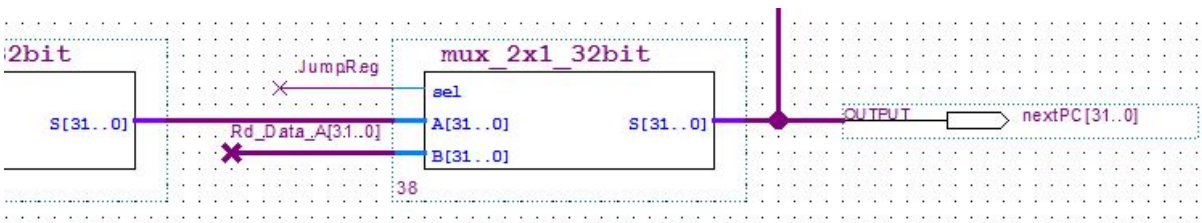

Figura 6. Multiplexador controlado pelo sinal JumpReg.

3. Simulação

Para realizar a simulação do circuito construído, foi utilizado um simples programa que utiliza todas as três instruções. O programa é composto por um procedimento chamado *leaf_example* e uma seção chamada *main*. A primeira instrução desvia o fluxo para a seção main, utilizando a instrução **j**; após isso, os valores 4, 3, 2 e 1 são carregados nos registradores $a0, $a1, $a2 e $a3, respectivamente. Em seguida, o procedimento *leaf_example* é chamado, utilizando a instrução **jal**. O procedimento realiza as operações $t0 = $a0 + $a1, cujo resultado é

7; $t1 = $a2 + $a3, cujo resultado é 3; e $v0 = $t0 - $t1, cujo resultado é 4. Na sequência, o fluxo retorna para a seção main, através da instrução **jr** $ra. A figura 7 apresenta o estado dos registradores após a simulação do programa no MARS.

$zero	0	0x00000000	$s0	16	0x00000000
$at	1	0x00000000	$s1	17	0x00000000
$v0	2	0x00000004	$s2	18	0x00000000
$v1	3	0x00000000	$s3	19	0x00000000
$a0	4	0x00000004	$s4	20	0x00000000
$a1	5	0x00000003	$s5	21	0x00000000
$a2	6	0x00000002	$s6	22	0x00000000
$a3	7	0x00000001	$s7	23	0x00000000
$t0	8	0x00000007	$t8	24	0x00000000
$t1	9	0x00000003	$t9	25	0x00000000
$t2	10	0x00000000	$k0	26	0x00000000
$t3	11	0x00000000	$k1	27	0x00000000
$t4	12	0x00000000	$gp	28	0x10008000
$t5	13	0x00000000	$sp	29	0x7fffeffc
$t6	14	0x00000000	$fp	30	0x00000000
$t7	15	0x00000000	$ra	31	0x00400028

Figura 7. Valores nos registradores após a execução do programa.

O programa foi exportado e adaptado para o formato Memory Initialization File (.mif) para ser utilizado na simulação através do Quartus II. Foi executada uma simulação do tipo Timing Simulation, utilizando o arquivo de configuração fornecido com a organização parcialmente desenvolvida. Os resultados em forma de onda são apresentados a seguir.

Na figura 8, é possível observar o comportamento correto da instrução j. O PC, que possuia valor 0x00400000, foi deslocado para 0x00400014, endereço que expressa o início da seção *main*, ao invés de ser incrementado em 4. Além disso, é possível observar o sinal *Jump* se tornando ativo.

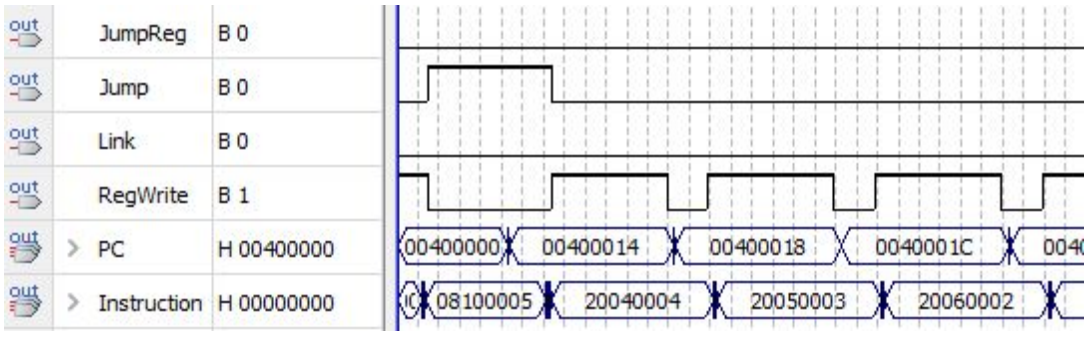

Figura 8. Execução da instrução j.

Observando a figura 9, é possível identificar as duas ações da instrução jal: o PC é deslocado, como na instrução j, desta vez para 0x00400004, e o registrador $ra (31) recebe o valor do PC atual somado com 4. Novamente, é possível ver os sinais *Jump* e *Link* se tornando ativos.

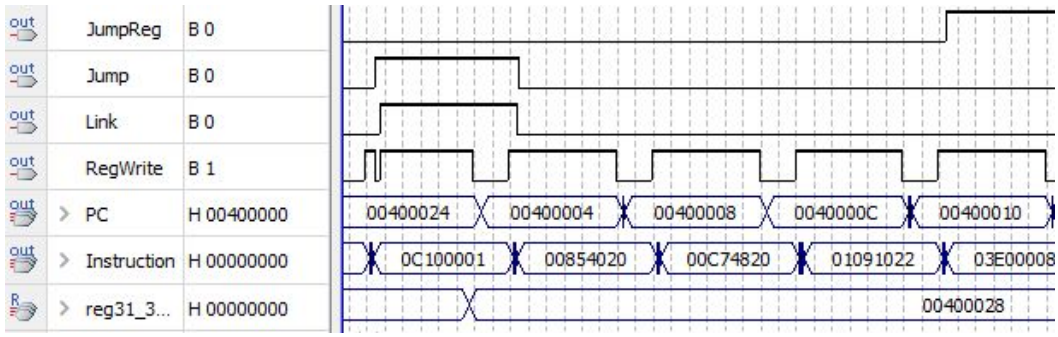

Figura 9. Execução da instrução jal.

Na figura 10, a execução da instrução jr $ra é apresentada. O valor 0x00400028, contido no registrador $ra (observável na figura 9), é atribuído diretamente ao PC, realizando o desvio. Mais uma vez, a ativação do sinal *JumpReg* pode ser observada.

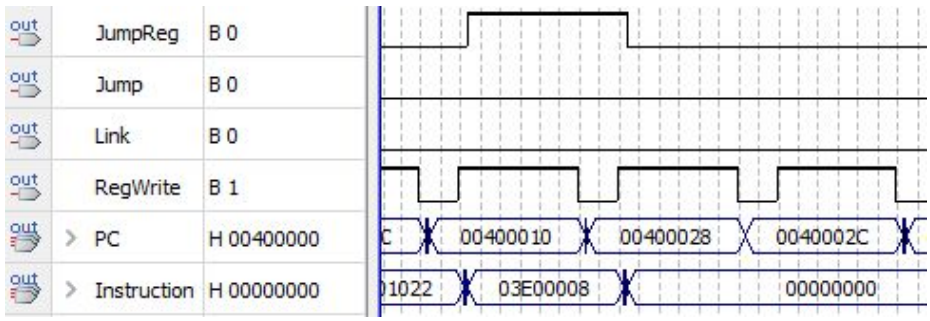

Figura 10. Execução da instrução jr.

Concluindo, a figura 11 apresenta a forma de onda completa, com os registradores relevantes, os sinais adicionados e o PC dispostos.

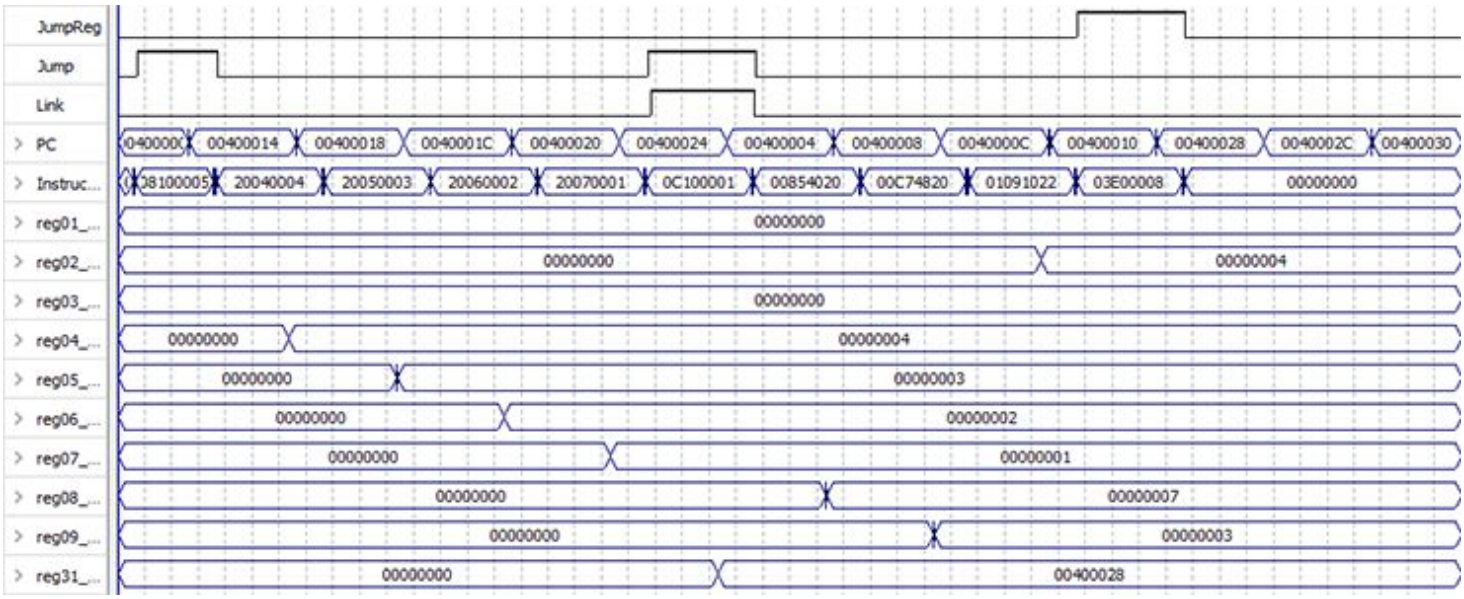

Figura 11. Forma de onda completa.

Apêndice - Memory Initialization File e código fonte em assembly

```
DEPTH = 256; % Number of positions    %
WIDTH = 32;        % Position size    %

ADDRESS_RADIX = HEX;
DATA_RADIX    = HEX;

CONTENT
  BEGIN
% The following addresses considers an offset that equals 0x04000020. In
other words, the first position (0x00000000) points to 0x04000020, an so on.
%
      00000000 : 08100005;  % j main               %
      00000001 : 00854020;  % add $t0, $a0, $a1   %
      00000002 : 00C74820;  % add $t1, $a2, $a3    %
      00000003 : 01091022;  % sub $v0, $t0, $t1    %
      00000004 : 03E00008;  % jr $ra         %
      00000005 : 20040004;  % addi $a0, $zero, 4  %
      00000006 : 20050003;  % addi $a1, $zero, 3  %
      00000007 : 20060002;  % addi $a2, $zero, 2  %
      00000008 : 20070001;  % addi $a3, $zero, 1  %
      00000009 : 0C100001;  % jal leaf_example     %
      [0000000A..000000FF] : 00000000;
END ;


%-------------------------------------------------------------------------%
% Original assembly source code:
      .text                   # segmento de código (programa)
            j     main
leaf_example:
            add   $t0, $a0, $a1      # $t0 = a + b
            add   $t1, $a2, $a3      # $t1 = i + j
            sub   $v0, $t0, $t1      # f = $t0 - $t1
            jr    $ra                # retorna do procedimento
main:
            addi  $a0, $zero, 4      # inicializa 1° parâmetro (g)
            addi  $a1, $zero, 3      # inicializa 2° parâmetro (h)
            addi  $a2, $zero, 2      # inicializa 3° parâmetro (i)
            addi  $a3, $zero, 1      # inicializa 4° parâmetro (j)
            jal   leaf_example # chama o procedimento
%
```