UNIVERSIDADE DO VALE DO ITAJAÍ - UNIVALI

GUSTAVO COPINI DECOL
JOÃO PAULO ROSLINDO

RELATÓRIO SOBRE PROCESSADOR MONOCICLO MIPS

Itajaí
2017

GUSTAVO COPINI DECOL

JOÃO PAULO ROSLINDO

RELATÓRIO SOBRE PROCESSADOR MONOCICLO MIPS

Relatório apresentado ao curso de Engenharia de Computação na Universidade do Vale do Itajaí - UNIVALI, na disciplina de Arquitetura de Computadores, como requisito parcial para obtenção de nota.

Professor: Douglas Rossi de Melo

Itajaí

2017

SUMÁRIO

1 INTRODUÇÃO

Após todos os estudos sobre a linguagem *assembly* aplicada ao processador MIPS, nesta atividade utilizamos de uma abordagem um pouco mais voltada ao hardware. Com todos os conhecimentos adquiridos até o momento sobre os modos de endereçamento, os tipos de instruções, formas de onda e toda a forma como um processador MIPS monociclo funciona, neste último trabalho, com o auxílio do *software* Quartus II, implementamos novas funcionalidades ao modelo monociclo do mesmo.

Modificando e adicionando novos componentes, implementamos ao hardware já apresentado, a capacidade de lidar com instruções como: *jump, jal* e *jr*.

2 INTRODUÇÃO AO MIPS MONOCICLO

Neste trabalho, tivemos como objetivo, a implementação de suporte em hardware, para algumas novas instruções no processador MIPS monociclo. São elas: *jump, jal* e *jr*.

Para esta implementação, antes de qualquer coisa, foi preciso realizar uma análise completa sobre o funcionamento deste processador, e quais componentes seriam necessários adicionar ou modificar. Para chegar a essa conclusão foi necessário um bom entendimento sobre o funcionamento das próprias instruções devem ser adicionadas. Como o nosso conhecimento sobre a linguagem *assembly* adquirido ao longo das aulas anteriores, esse processo tornou-se um pouco mais fácil.

Abaixo uma imagem do processador MIPS monociclo sem a implementação das instruções no software Quartus II:

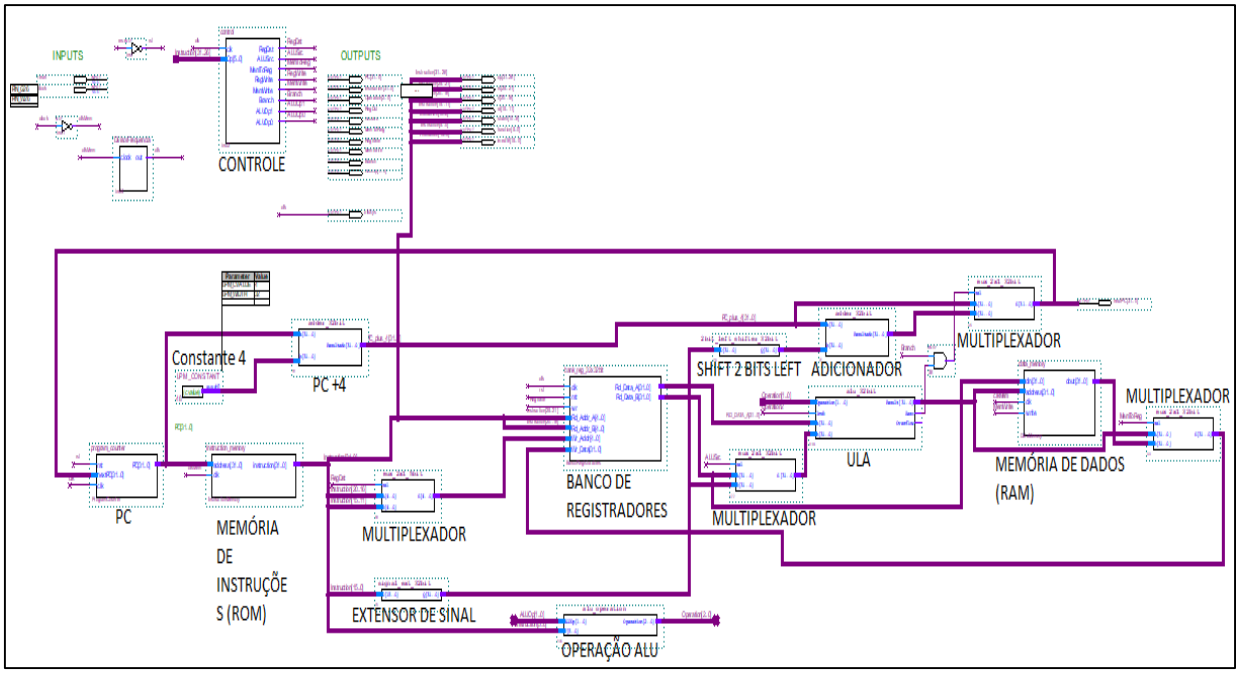

Figura 1-Processador MIPS monociclo no software Quartus II.

Na Figura 1, podemos observar todo o processador MIPS, sem as instruções sugeridas implementadas. Foi a partir deste projeto, que iniciamos as modificações necessárias para a primeira instrução, o *jump*.

3 IMPLEMENTAÇÃO DA INSTRUÇÃO JUMP

Por se tratar de uma instrução de desvio incondicional, sua implementação é extremamente simples. Como todas as outras instruções, para a sua aplicação foi necessário adicionar o seu *opcode* no controle, e modificar o caminho de dados. Sua implementação será descrita passo a passo a seguir:

3.1 PRIMEIRO PASSO

A primeira coisa a ser feita, foi a modificação do bloco de controle. Dentro do mesmo, implementamos, de forma análoga às outras instruções, um suporte para o *opcode* do *jump* para que quando, uma instrução utilizando sua funcionalidade fosse utilizada, ele fosse ativado.

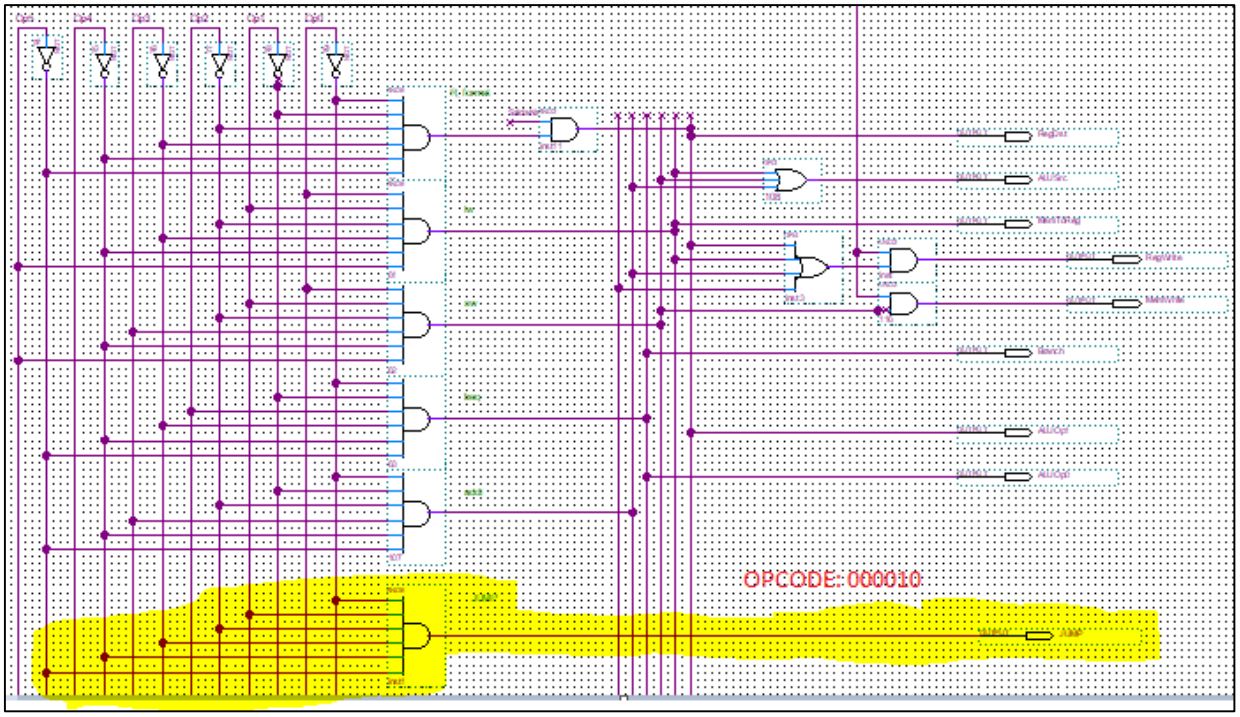

Figura 2- Controle do processador MIPS monociclo, com o opcode da instrução jump implementado.

Como podemos observar na Figura 2, com o auxílio de uma porta *and* de 6 portas, adicionamos o *opcode* correspondente da instrução *jump* ao controle, para quando a instrução for chamada, sua saída entrar em estado 1.

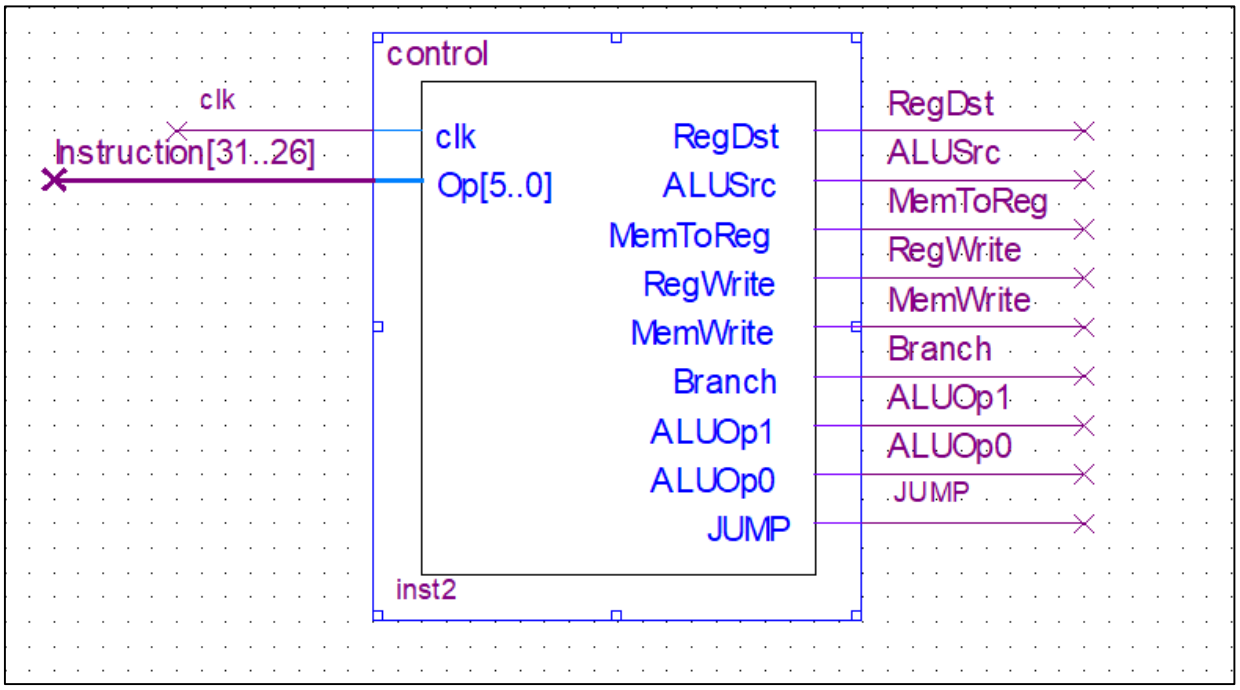

Figura 3 -Controle com o jump implementado.

3.2 SEGUNDO PASSO

Com o controle modificado para a instrução *jump*. Nos resta implementar o suporte no caminho de dados. Tivemos que adicionar dois componentes para isso: um concatenador que nós mesmos criamos, e um multiplexador.

Por conta do *jump* calcular o endereço de destino de uma forma diferente do *branch*, foi necessária a implementação de um concatenador, que em sua saída, retorna o endereço pronto. Abaixo uma imagem do formato da instrução *jump* com suas respectivas *"ranges"* de barramentos:

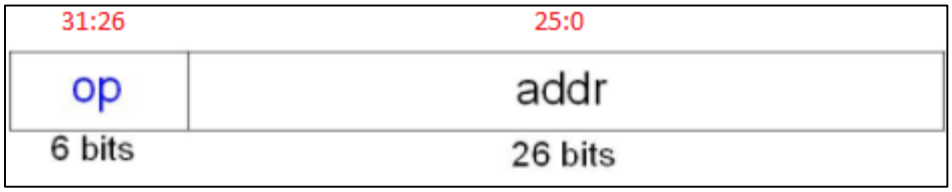

Figura 4- Formato da instrução jump

Sabendo o formato, e os respectivos barramentos, é necessário apenas saber como é calculado o endereço de destino da instrução.

No *jump* os 4 bits mais significativos do PC atual +4 (31:28) são concatenados com os 26 bits do imediato da instrução (Figura 4) deslocados dois bits para a esquerda (00bin).

Dessa forma, criamos um concatenador que realiza essas alterações, para que a instrução funcione da forma como deve.

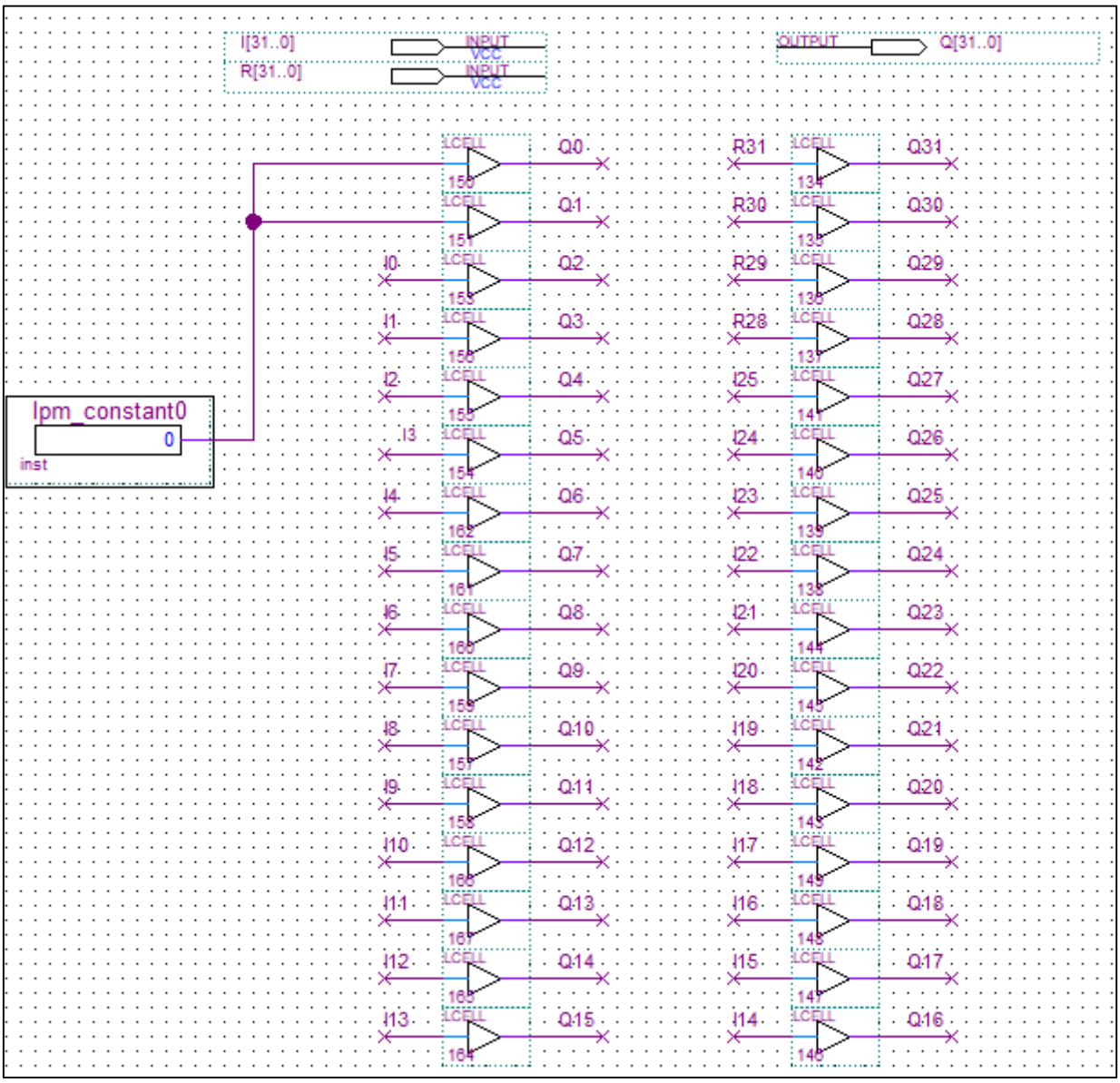

Figura 5-Concatenador por dentro.

Observando a Figura 5 acima, note que no concatenador entram os 32 bits do PC+4 (R) e os 32 bits da Instrução (I), e saem 32bits (Q). A concatenação foi feita de modo bem trivial. Utilizando buffers, pegamos os 00 bits do deslocamento para à esquerda e fomos inserindo bit a bit, todos os 26 bits do imediato da instrução (25:0), com os 4 bits do PC+4 (31:26), no final o que obtivemos é uma saída de 32 bits (Q) com a concatenação pronta.

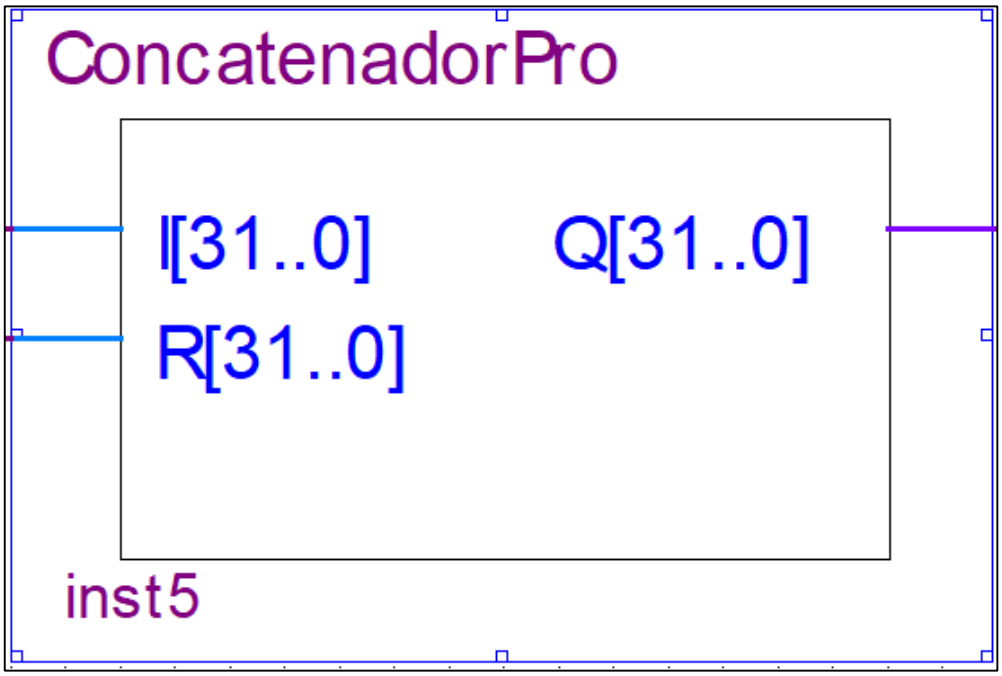

Figura 6- Bloco do concatenador.

Com o concatenador pronto, tudo o que tivemos que fazer foi: adicionar um multiplexador antes do *nextPC*, para caso a saída *jump* do controle seja 1, ele receba a saída desse concatenador, caso seja 0, receba o PC normalmente.

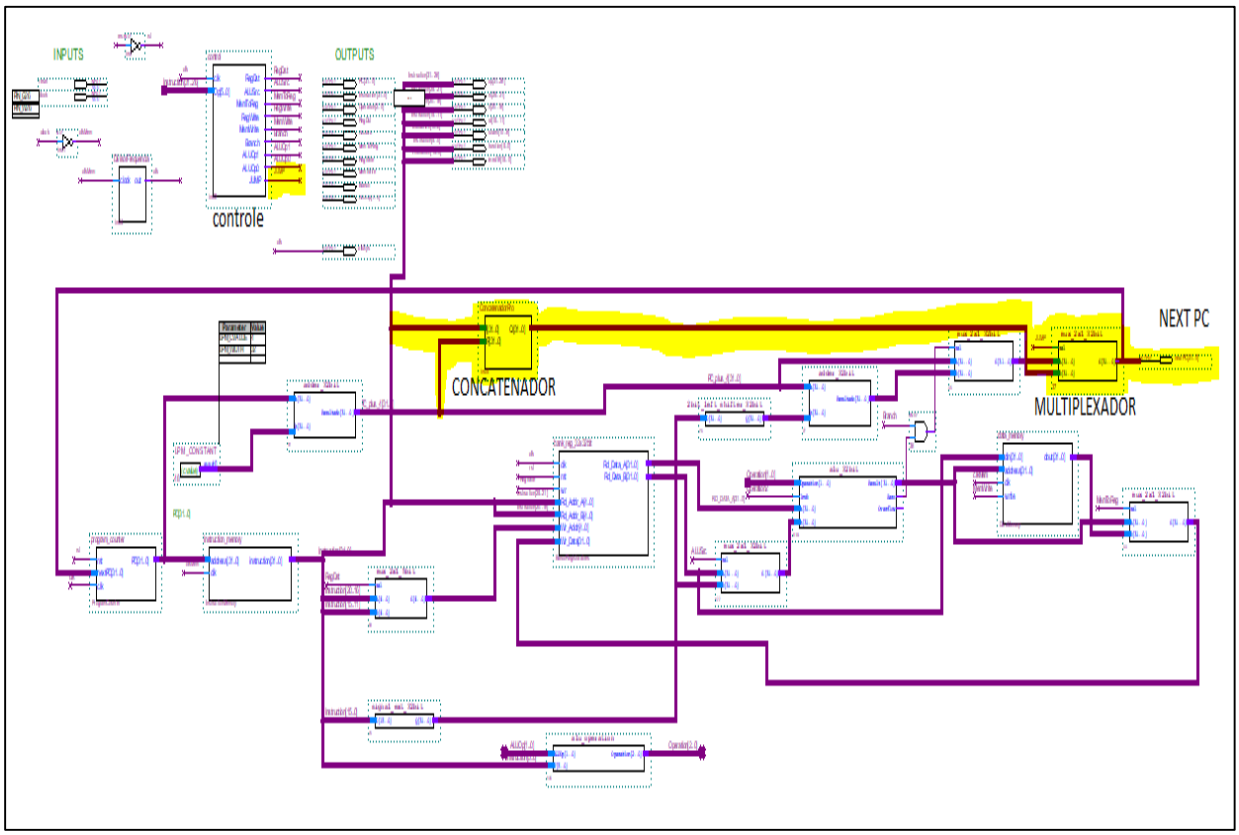

Figura 7- Processador MIPS monociclo com a instrução jump implementada

3.3 VALIDAÇÃO DA INSTRUÇÃO

Após toda a implementação ser concluída, nos resta um último passo, verificar o seu funcionamento. A validação consiste na utilização de códigos gerados em hexadecimal, salvos em arquivos .MIF, que são inseridas na ROM (memória de instruções) do nosso processador. Com o arquivos gerado e salvo, iremos analisar o seu funcionamento através de diagramas de forma de onda.

O programa a ser testado, foi a rom 4, já inclusa no projeto. O arquivo contém o seguinte código em hexadecimal, e abaixo do mesmo podemos ver a sua tradução.

```
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 : 20100001;  % addi $s0, $zero, 1  %
    00000001 : 22100002;  % addi $s0, $s0, 2    %
    00000002 : 08100009;  % j    loop           %
    [00000003..0000000F] : 00000000;
END ;


%-------------------------------------------------------------------------------%
% Original assembly source code:
      .text
main:
      addi $s0, $zero, 1
loop: addi $s0, $s0, 2
      j    loop
%
```

Figura 8-Rom 4, utilizada para testar a instrução jump.

Durante os testes, vimos que os resultados não estavam batendo conforme o esperado, e analisando as nossas modificações em hardware, não conseguimos identificar o erro. Com isso em mente, decidimos analisar a rom, e descobrimos uma pequena incompatibilidade no código em hexadecimal do jump. Na Figura 8, a linha grifada é onde encontramos o possível erro.

Utilizando da ferramenta MARS, colamos o código em assembly da Figura 8, e vimos que realmente o código em hexadecimal possuía um erro. Invés de ser 08100009, segundo o MARS, deveria ser 08100001, como mostra na imagem abaixo:

Bkpt	Address	Code	Basic	Source
	0x00400000	0x20100001	addi $16,$0,1	3: addi $s0, $zero, 1
	0x00400004	0x22100002	addi $16,$16,2	4: loop: addi $s0, $s0, 2
	0x00400008	0x08100001	j 0x00400004	5: j loop

Figura 9- Imagem com os códigos das instruções da ROM 4 no MARS

Após alterarmos o 08100009 para 08100001, conseguimos os resultados que eram esperados no diagrama de forma de ondas.

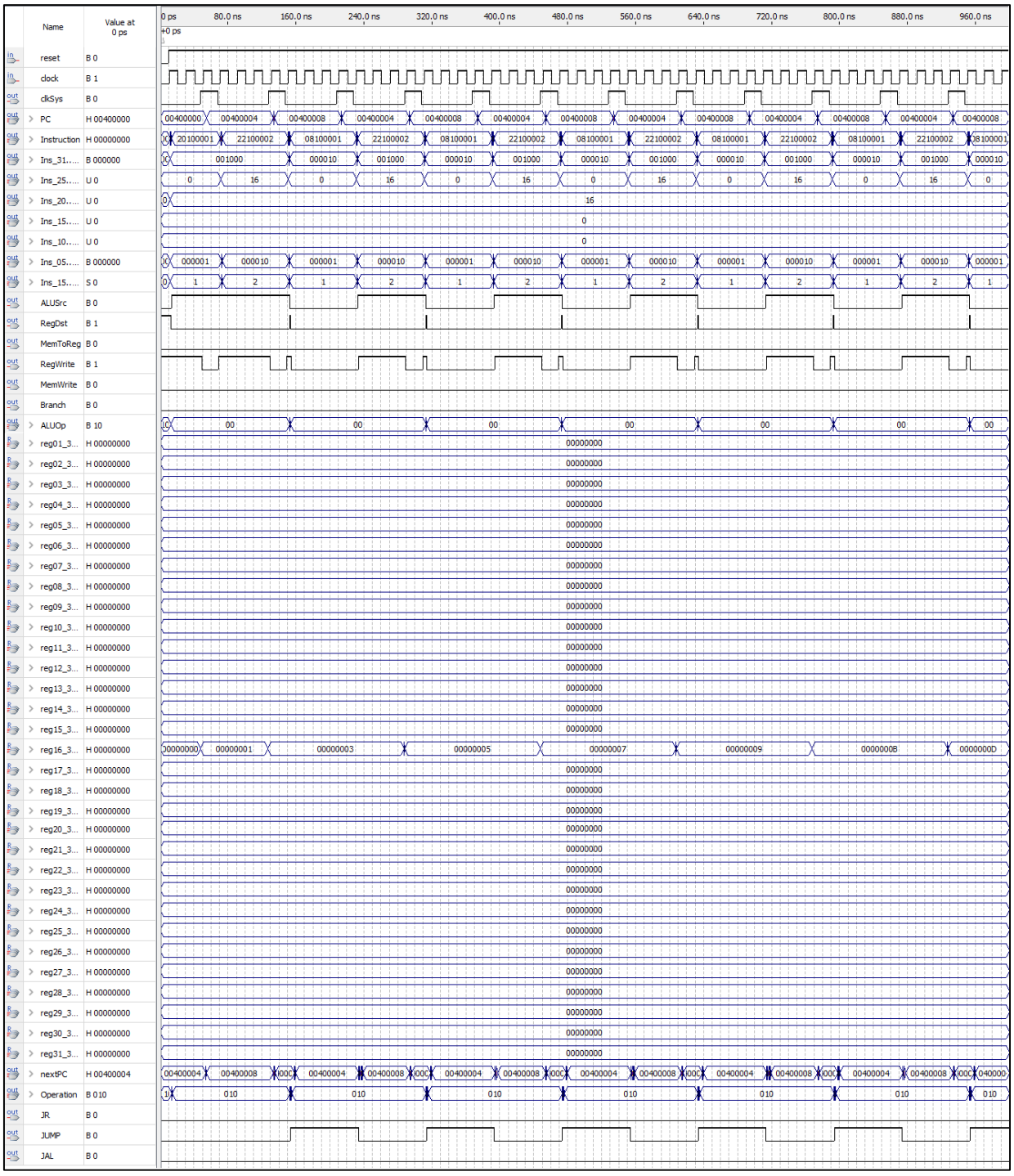

Figura 10- Diagrama de forma de ondas da ROM 4.

Podemos observar na Figura 10, como o funcionamento ocorreu da forma desejada, o loop que o *jump* gera (observado na saída *JUMP*), vai incrementando o registrador $s0, de 2 em 2, exceto na sua primeira utilização, que ele recebe apenas o 1.

4 IMPLEMENTAÇÃO DA INSTRUÇÃO JAL E JR

Para a implementação das instruções *jal* e *jr*, assim como no *jump*, tivemos que realizar algumas mudanças no controle e no caminho de dados. Ao contrário do *jump*, não criamos nenhum bloco novo, apenas inserimos alguns multiplexadores que, juntamente com os seletores corretos, resolveriam as nossas necessidades do projeto. As suas implementações serão explicadas passo a passo a seguir:

4.1 PRIMEIRO PASSO

Como já fizemos anteriormente, o primeiro passo para a implementação de uma nova instrução é entender como ela funciona, logo, conhecendo o seu devido *opcode*, inserimos no controle, suporte para ambas as instruções:

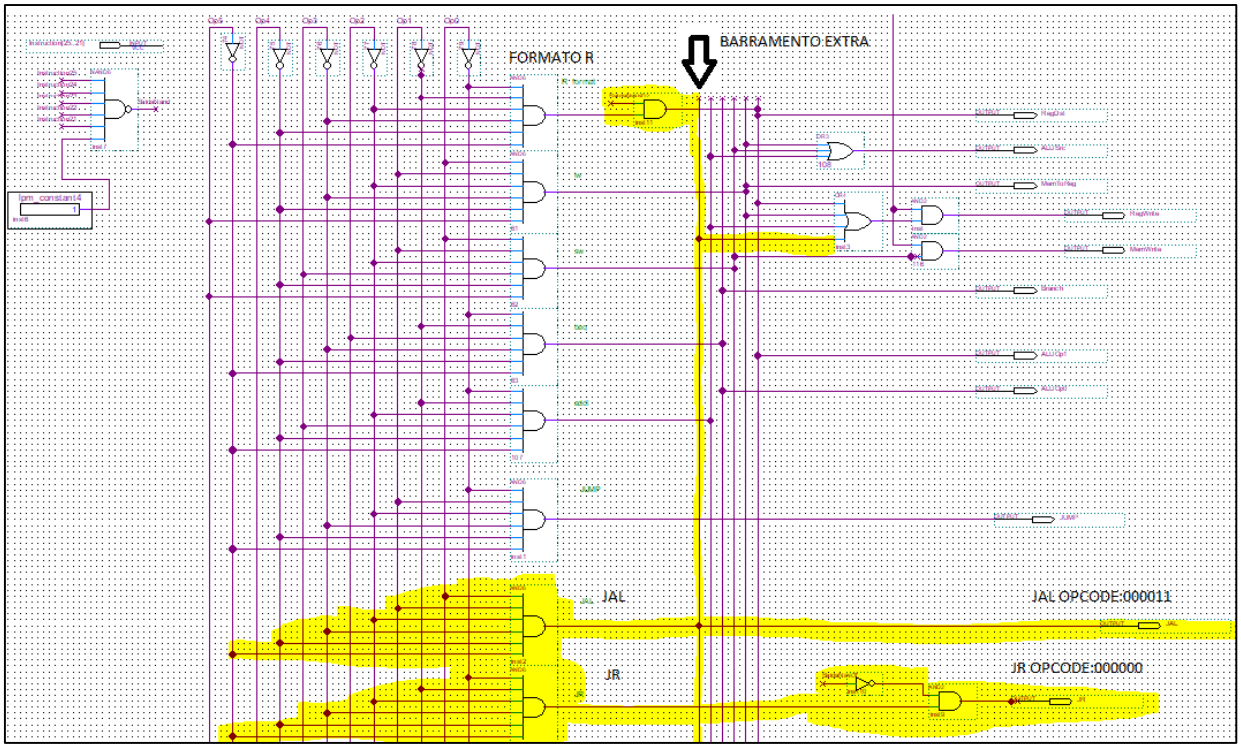

Figura 11-Controle modificado para as instruções jal e jr.

Seguindo a mesma lógica anteriormente descrita, conhecendo os *opcodes* de cada instrução, no caso do *jal* (primeiro pedaço grifado) inserimos uma porta and de 6 entradas, com o seu *opcode* entrando, e uma saída que será setada para 1 quando for ativado. Além disso, inserimos um barramento extra (grifado na vertical), pois como o *jal* é uma instrução que precisa escrever no banco de registradores, tivemos que habilitar mais algumas funções do controle além dele mesmo.

Em relação ao *jr*, notamos que o seu *opcode* é o mesmo do formato R, e portanto, poderia causar problemas na hora do seu funcionamento por conta dos multiplexadores. Com isso em mente, adicionamos uma lógica que reconhece quando a instrução é do tipo *jr*, para que esse problema não ocorra.

A lógica para evitar esse problema foi resolvida utilizando alguns campos da própria instrução dentro do controle.

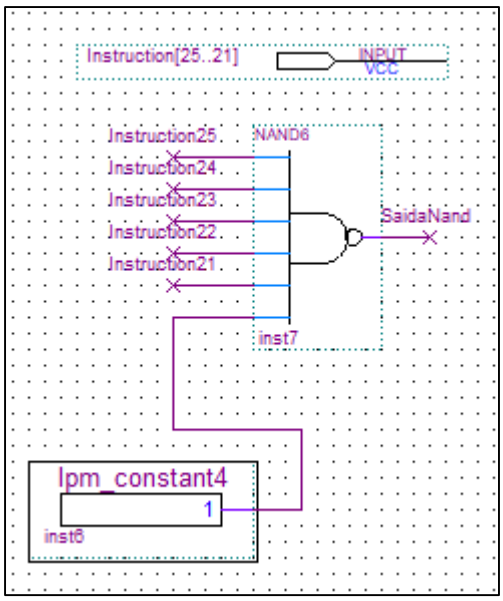

Figura 12-Entrada extra inserida no controle.

Como a instrução jr utiliza de um registrador $ra, que no banco de registradores é o último, ou seja, seu código é 1111, com uma nand, podemos obter um sinal equivalente a 0, caso a instrução seja realmente um *jr* e não uma instrução do formato R. Se esse for realmente o caso então, a saída *jr,* retornará 1 do controle, enquanto a saída R, retornará 0.

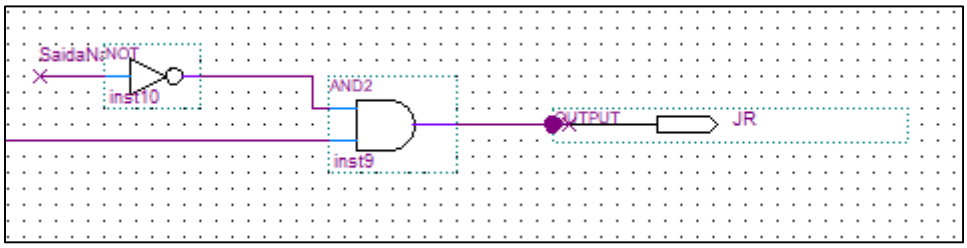

Figura 13-Porta and com a saidaNand negada entrando, para que JR retorne 1 caso a instrução seja do tipo jr.

Na figura 13, podemos observar essa lógica aplicada.

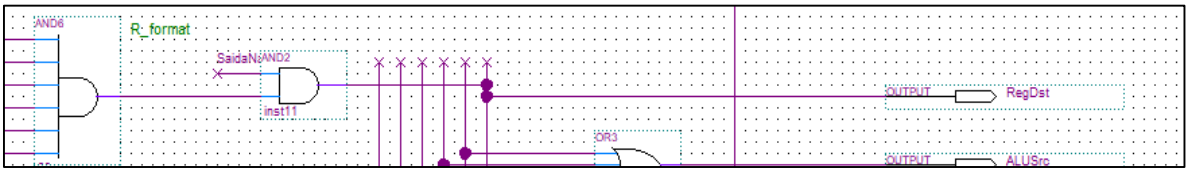

Figura 14-Porta and com a saidaNand entrando, para que RegDest retorne 0 caso a instrução seja um jr.

Na figura 14, a outra parte da lógica é aplicada ao formato R.

4.2 SEGUNDO PASSO

A segunda e última parte da implementação trata do caminho de dados. Nessa parte foi preciso um pouco a mais de atenção do que a instrução *jump*, porém não criamos nenhum novo componente. A forma de implementação foi apenas completada com alguns multiplexadores e seus seletores nos lugares certos.

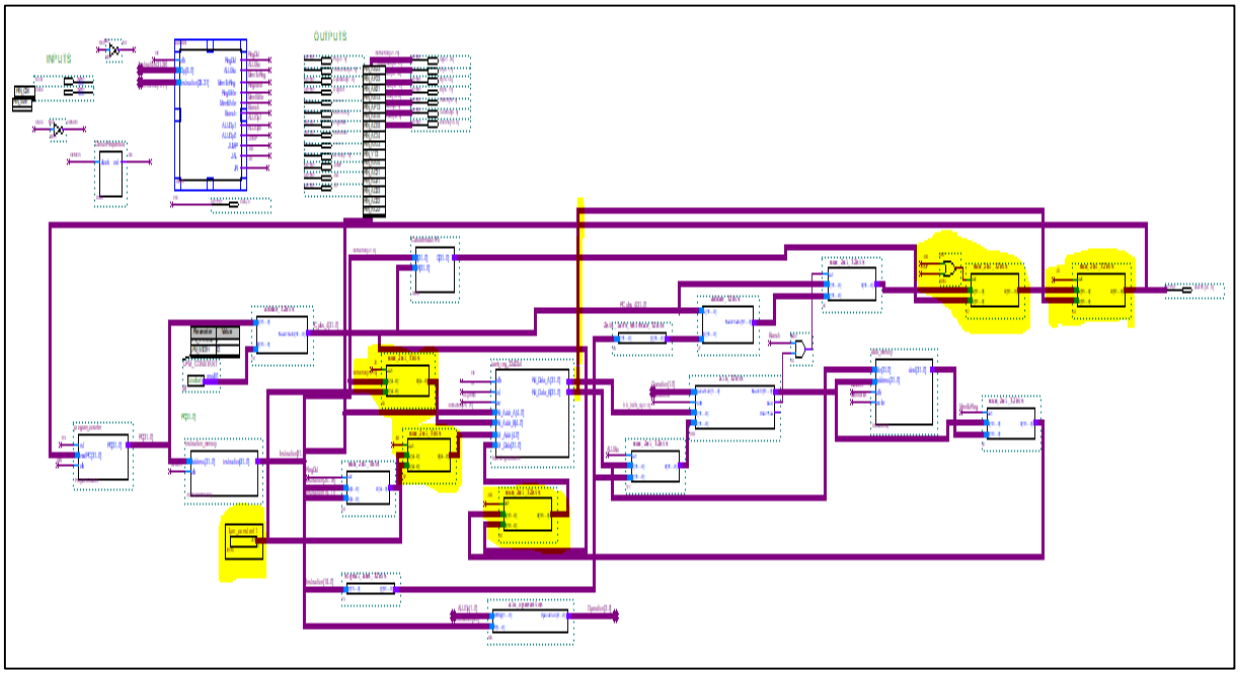

Figura 15-Processador MIPS monociclo com as alterações destacadas para as instruções jal e jr.

Na Figura 15 acima, podemos observar todas as alterações feitas nessa última etapa, todas elas serão explicadas a seguir.

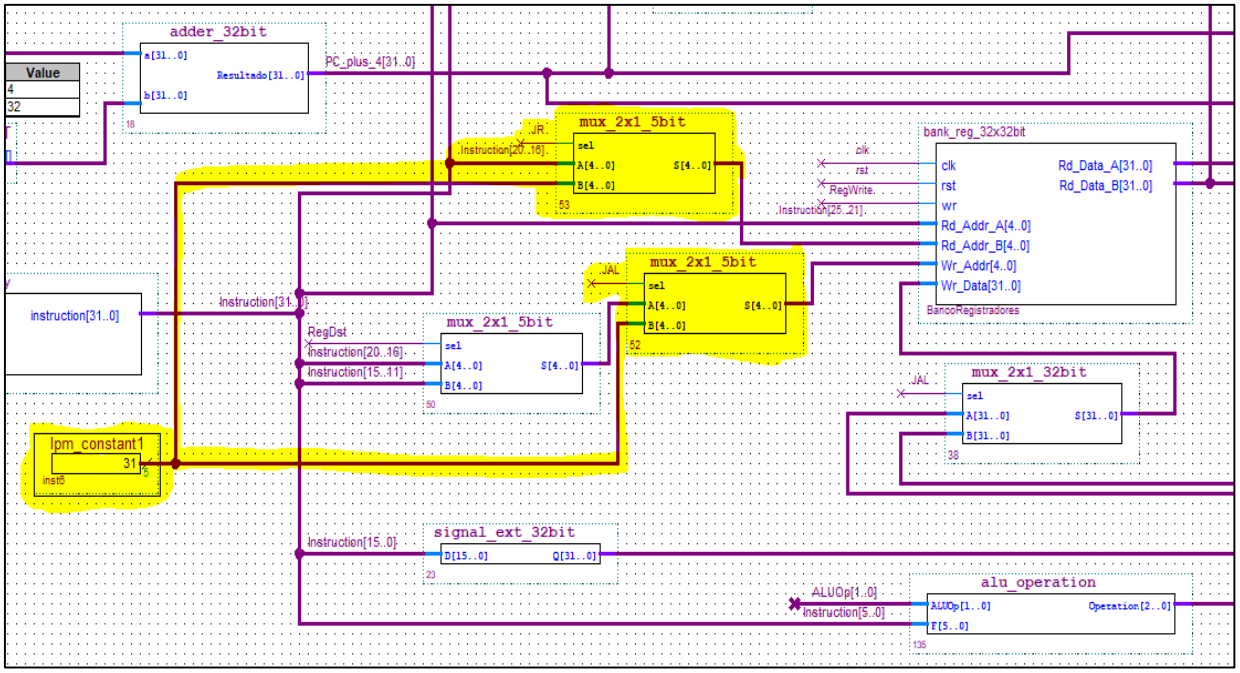

Figura 16- Processador MIPS monociclo modificado.

Primeiramente inserimos dois multiplexadores que são destacados na Figura 16. O multiplexador de baixo, serve para escrever no registrador equivalente à constante 31, que no nosso caso seria 1111, ou seja, o registrador $ra caso *jal* seja ativo. O multiplexador de cima pega a mesma constante do $ra, juntamente com alguns bits da instrução, caso *jr* seja ativo o Rd_Addr_B do banco de registradores receberá essa constante e a sua saída Rd_Data_B irá retornar o endereço armazenado em $ra.

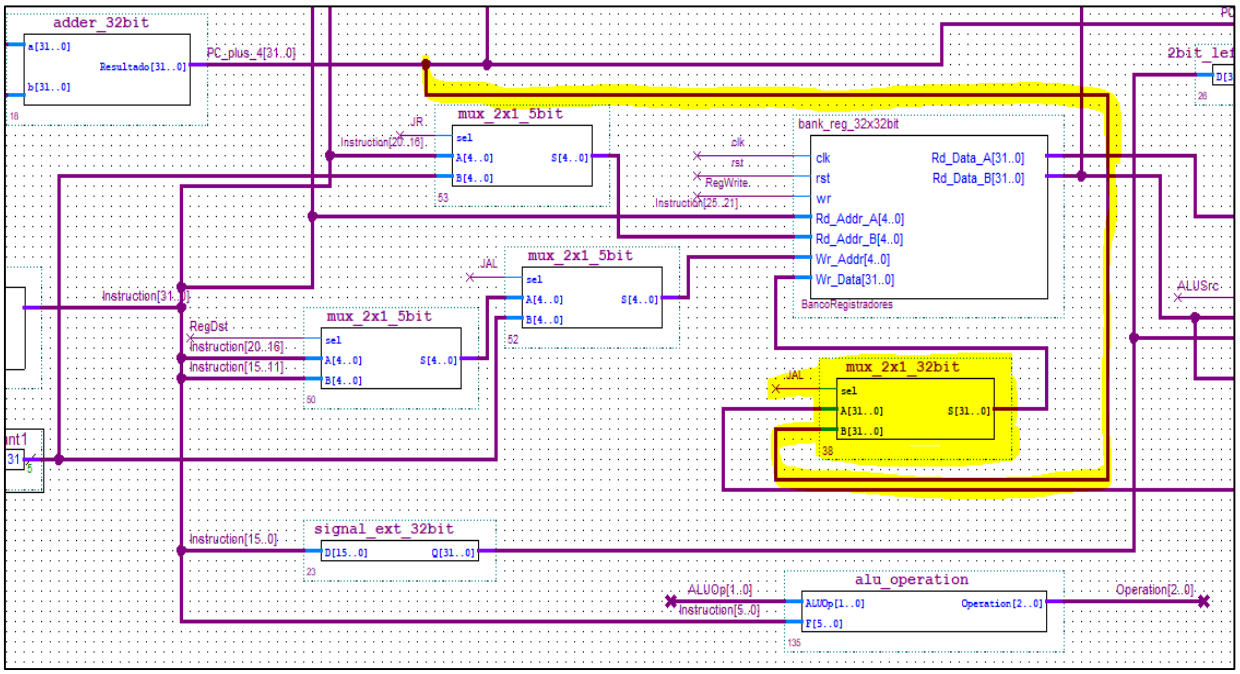

Figura 17- Processador MIPS monociclo modificado.

Na Figura 17, podemos observar a continuação da implementação da instrução *jal*, caso o mesmo seja setado para 1, o banco de registradores irá receber em 1111($ra), o endereço de PC+4, para ser utilizado mais tarde no *jr.*

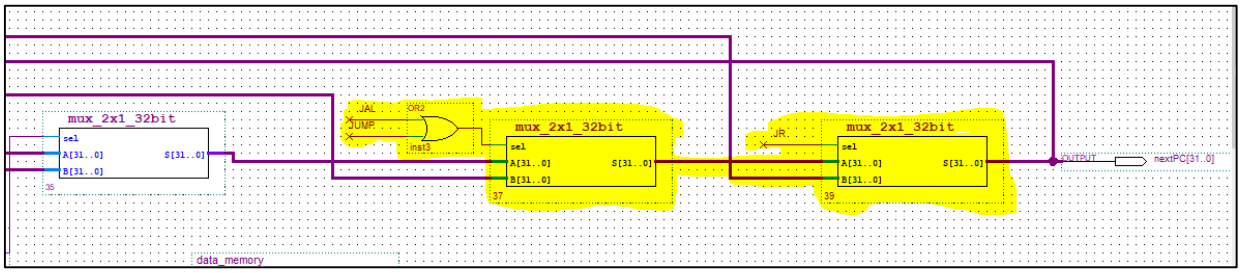

Figura 18- Processador MIPS monociclo modificado.

Na Figura 18, foi feita a última etapa das modificações. O antigo multiplexador que havia sido inserido para o *jump* recebeu uma pequena modificação no seu seletor. Como a forma que o endereço é calculado pelo *jal* e pelo *jump* são as mesmas, colocamos uma porta or para caso qualquer uma das duas instruções esteja ativa, o que passará pela sua saída será o endereço do salto. Em seguida o novo multiplexador que foi inserido para o *jr*. Caso ele seja ativo, o *nextPC* irá receber o endereço que vem lá do banco de registradores pela saída Rd_Data_B.

4.3 VALIDAÇÃO DA INSTRUÇÃO

Para validar essas alterações realizadas no modelo MIPS monociclo, utilizamos uma ROM que nós mesmo geramos a partir de um código pronto publicado no enunciado do trabalho.

```
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;
    00000001 : 00854020;
    00000002 : 00c74820;
    00000003 : 01091022;
    00000004 : 03E00008;
    00000005 : 20040004;
    00000006 : 20050003;
    00000007 : 20060002;
    00000008 : 20070001;
    00000009 : 0C100001;
    0000000A : 00000000;
    [0000000B..0000000F] : 00000000;
END ;
%-------------------------------------------------------------------------------%
% Original assembly source code:

# Trecho em C para validação do jr e do jal:
# int leaf_example (int g, int h, int i, int j) {
#     int f;
#     f = (g + h) - (i + j);
#     return f;

    .text   # segmento de código (programa)
    j   main
leaf_example:
    add $t0, $a0, $a1   # $t0 = g + h
    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 1º parâmetro (g)
    addi $a1, $zero, 3  # inicializa 2º parâmetro (h)
    addi $a2, $zero, 2  # inicializa 3º parâmetro (i)
    addi $a3, $zero, 1  # inicializa 4º parâmetro (j)

    jal leaf_example    # chama o procedimento

    nop             # não faz nada. $v0 tem o resultado do procedimento
```

Figura 19-ROM 5 utilizada para testar o MIPS.

Com essa ROM apresentada na Figura 19, criada com a ajuda do MARS, obtivemos a seguinte forma de onda no teste através do Quartus II.

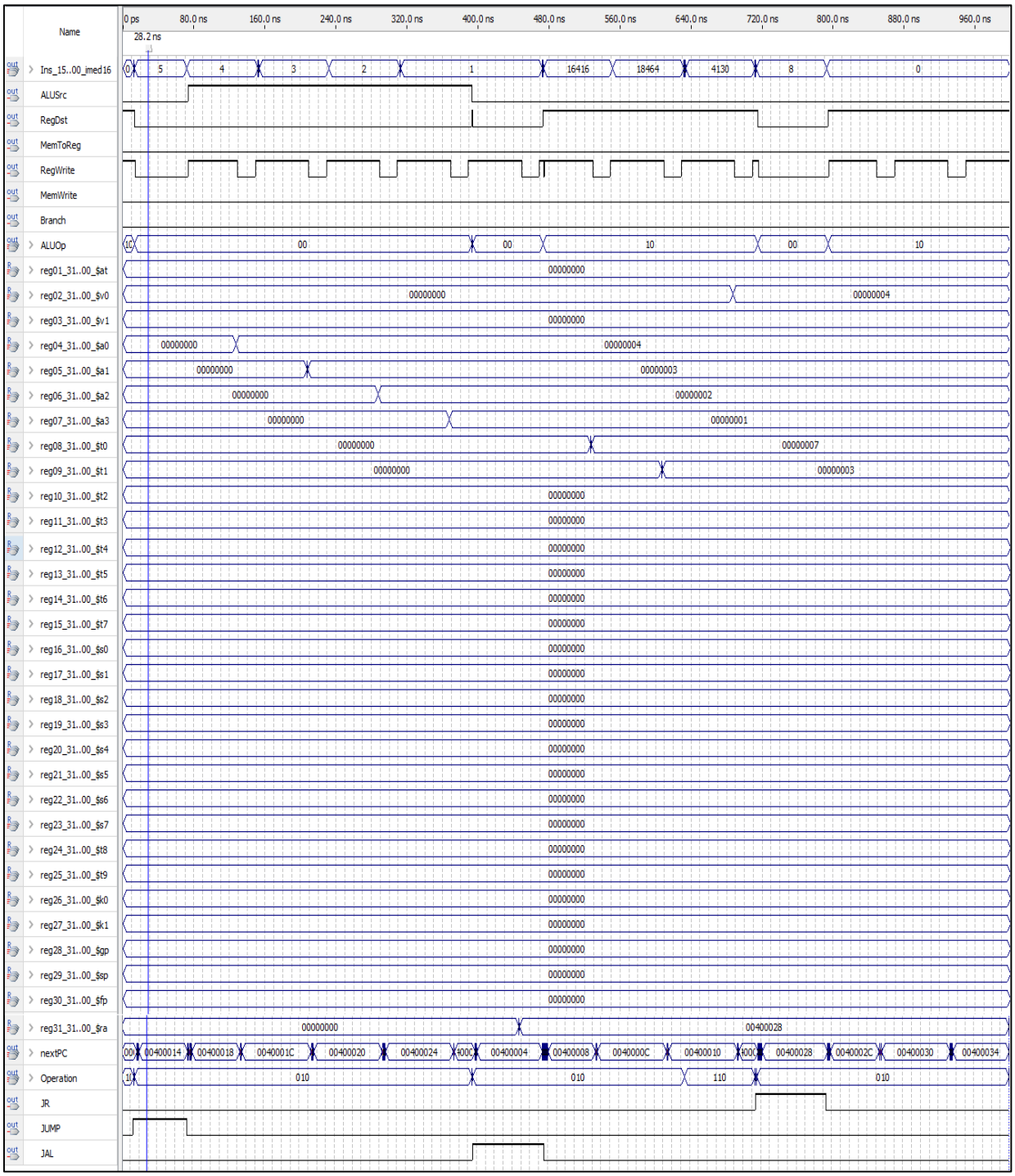

Figura 20- Forma de onda obtida na ROM 5.

Como podemos visualizar na Figura 19, a ROM 5 se trata de algumas operações básicas entre registradores, observando a Figura 20, podemos notar que todas essas operações foram feitas corretamente, além de podermos analisar o *jr, jal* e o *jr*, funcionando com as suas devidas ondas.

5 CONCLUSÃO

A partir deste trabalho conseguimos entender como são feitas as implementações em hardware de instruções em processadores e aplica-las, mais especificamente, em um modelo monociclo MIPS. Além de toda a lógica de projeto, conseguimos aplicar alguns conceitos vistos em aula, que nos auxiliaram no desenvolvimento do mesmo. Com isso podemos ter uma breve noção de como funcionam, mesmo que de forma bem básica, os processadores mais atuais que existem no mercado atualmente.