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

Contents

[1. Introdução 3](#_Toc486439399)

[R-format 4](#_Toc486439400)

[I-format 5](#_Toc486439401)

[J-format 6](#_Toc486439402)

[2. Módulos do Projeto 7](#_Toc486439403)

[I-fetch 7](#_Toc486439404)

[I-decode 7](#_Toc486439405)

[Control 7](#_Toc486439406)

[Execute 7](#_Toc486439407)

[3. Simulação 7](#_Toc486439408)

[AND 7](#_Toc486439409)

[OR 7](#_Toc486439410)

[BNE - Branch Not Equal 7](#_Toc486439411)

[JAL e JR - Jump and Link / Jump Register 7](#_Toc486439412)

[SLL - Shift Left Logical 7](#_Toc486439413)

[SRL - Shift Right Logical 7](#_Toc486439414)

[SLT - Set on Less Than 7](#_Toc486439415)

[4. Vetor - Implementação e Simulação 7](#_Toc486439416)

[5. Construção do projeto 7](#_Toc486439417)

[6. Considerações Finais 7](#_Toc486439418)

[Referências 7](#_Toc486439419)

[Apêndice 7](#_Toc486439420)

# 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*).

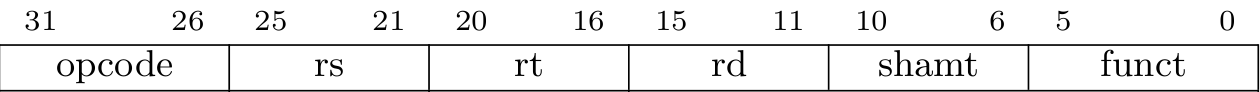
Considere que a implementação usada como base do presente projeto já suporta e executa corretamente as seguintes instruções:

* **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 e 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:



**Jr – Jump Register:**

https://lh4.googleusercontent.com/fEXEYmazGFJkSMveNK4KfkGWIxUQ5y-1eL0Y6pIl7D7eMtdGEs1H-OI7G9XmAlUJsoXiuSoxKwOZIvjz-VzFLcuqoYnJ0Yqr0I2YZlcHMBrHF8XRM0orIg9t4LPayYtOIVOyHsPr

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:**

https://lh5.googleusercontent.com/HHK_KWXLBN6_35XeSbL8GBSqxRbMlcfY_9L8oIJ4jBRXydrTp9cz2EN-kK8742xazYVkFQG9iERIYs1mCYUGNlounuLcbz5im0u-Yop9VUCeRRP0Sqp5MyqfNqz9yHJRAzBJsUaV

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:**

https://lh6.googleusercontent.com/ypJ9rP5FmilyrRx9DFvBFmisQFsbUJHlyOoYZSfH_4OnLwm8xod3v-FG1OPYaTjEjEkkyBKUCohdgSJ36Jsl6Bm-9H501DFKkyBkbyC5p6T_iUn6ujBCYm2JW0WO3n48PsCCIJlP

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.

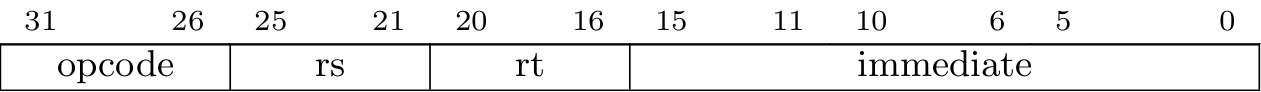
**Slt - Set on Less Than:**

https://lh6.googleusercontent.com/F90uKXJ19-9paWnbH7b7IImLOk8mZN8BG7jbNki4j7cmBbw0N99n9wa5SNw9rAMTmvXgOZHILBr0Toq26CqOk_ycWqYxINsBVYQqFfreV3MHCFu7G2-q3HRlBR6ZRI_0cT8Vml95

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:



**Addi – Add imediate value:**

https://lh5.googleusercontent.com/ZpQqrtG1mgVQKaUx_la_xqTpAlX4g_pZzr59mh7K8gQxlAVUlk1Eda_HsDlnQcV2eKsIfNtI3ufw0ldNibUugzOgRaEoNCVfEBHNd9Utm7E0RMPCtpj8-RBoViZ0xCMbNVNBqlZI

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:**

https://lh5.googleusercontent.com/CzSZRQertJB8OmAK_fpa2TTLrxCrHOIcTAhGlgS967j23cIYfiwbkZ2CVXtabdfOr7UDVulWW328TEzR5zUyTk8sLNE5EK6tueXkx-gn6vuF3Ydk3C4XRrQyte7DhoF7wD3fuWdj

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.

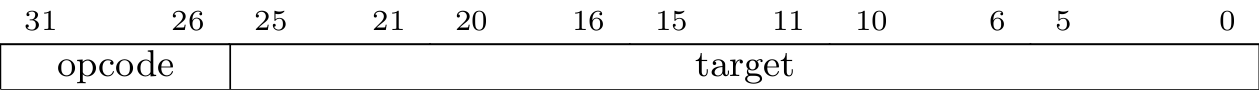
**Beq - Branch on equal:**

https://lh5.googleusercontent.com/Cp4tGnZ-MUsfdWMlfO-Nie0IxgBi-UG-rn0hux_OfodeGsNuivUPAvu64YtH6WbLxJBSH7wIZZLYq33BeY1o8L7Y3OcgaI4NNXD1rdHtlfh6zIWzWvvyy15oIF5GrlXi7MN0b-iA

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:



**J - Jump:**

https://lh5.googleusercontent.com/LhGquSfYLItB7ExfOrFkXd3tboR_EZ5FUkHj7_d0_3QeR7Ds0vbHO9jE8koVCETWnOiK31FJ7uu6VVKXKsuaXS0VFWRYvI-xP9jDHQ8z2vTZGFLNZOpFwJp3ucGEqnd7O4jgSIDZ

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:**

https://lh4.googleusercontent.com/nsamQ5ntl3BLqdeV66OwdP9w5DFqQjIu0V6gZiipBYCs1761zxXAizeXdjyhOc7u0HpFrGyPFK5ojwRUx8B_wZ9oJVERb_yegFCY6GlErcsVUSh5PFSmtmjf9QNQSMWjXSh5GUip

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.

# 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, isso foi feito adicionando 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 de **jump**, o PC recebe o valor específico do endereço da nova instrução (*instrJump)*. Na instrução de **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 de **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** e01 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 equivalente 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 deste 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 | Alu\_ctl(4) | Alu\_ctl(3) | Alu\_ctl(2) | Alu\_ctl(1) | Alu\_ctl(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).

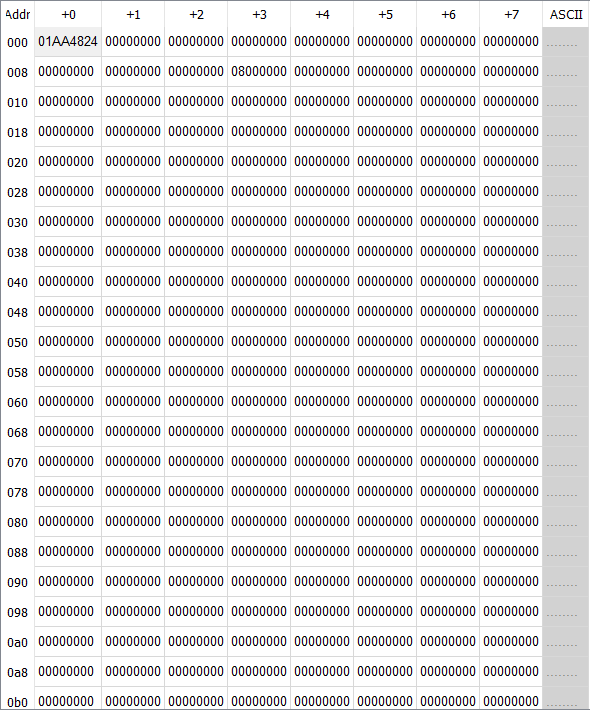
Vale notar que o sinal *JumpReg* é passado para o componente I-fetch, para que Next\_PC receba o registrador destino (presente em Alu\_Result), comumente o $ra, e a instrução **jr** seja feita corretamente. Além disso, para as demais instruções, a ALU sempre realiza a operação de soma.

# 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.



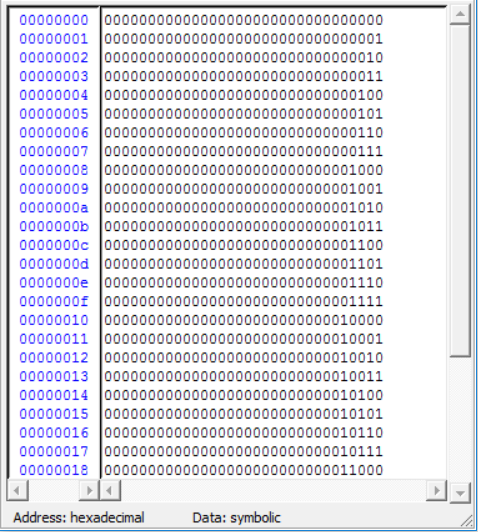


Figura 1.2 - Banco de registradores antes

Figura 1.1 - and.mif

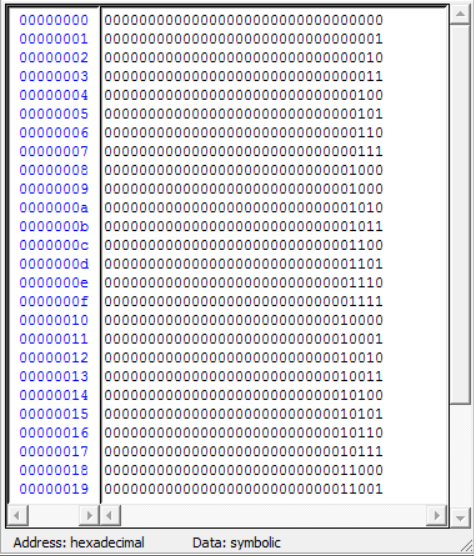


Figura 1.3 - Banco de registradores depois



## OR

Foram utilizados os mesmos registradores, porém com a instrução Or $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 = 1111 e sem mudanças nos demais registradores. Portanto a simulação foi bem-sucedida, já que o *or* bit a bit resultou no valor correto e o resultado obtido foi escrito no registrador destino.

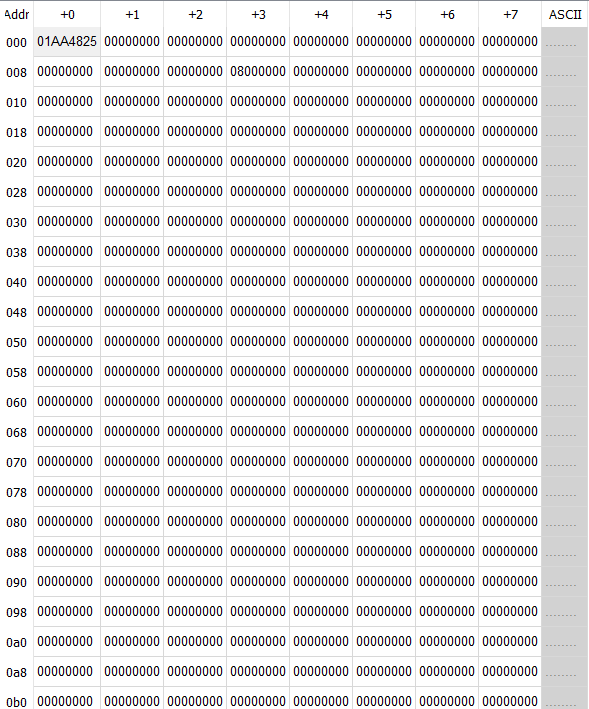


Figura 2.1 - or.mif

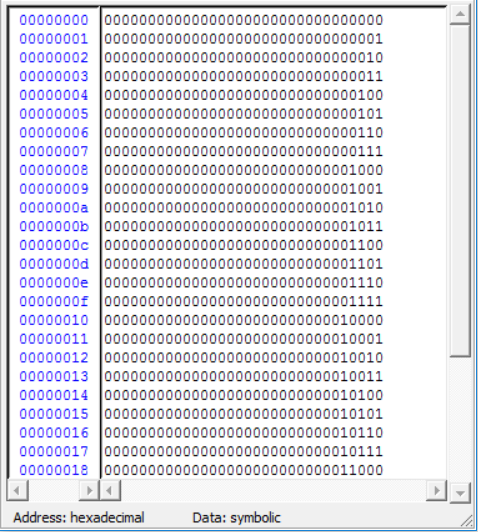
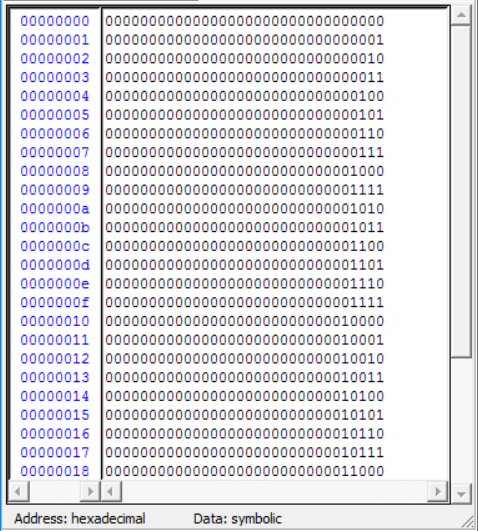


Figura 2.3 - Banco de Registradores depois

Figura 2.2 - Banco de Registradores antes



## BNE - Branch Not Equal

As instruções utilizadas foram Bne $zero $zero 0x0002(não fará o pulo) e Bne $zero $4 0x0002 (fará o pulo). Na imagem com as *waves,* na primeira instrução (PcAddr = 0x00) o readData1 e o readData2 contém o valor 0x0, ou seja, são iguais e por isso o sinal Zero está ativo alto, portanto não fará o pulo, e o sinal auxBranch indica 0x1 denotando que a instrução é um Beq (convenção, explicada na sessão do Control). No próximo pulso de clock, o PcAddr = 0x01, logo não houve pulo como o previsto, o readData1 contém o valor 0x0 e o readData2 o valor 0x4, o auxBrench indica que é um Bne, o sinal Zero está baixo pois readData1 é diferente de readData2, e portanto deve haver o pulo. No próximo pulso de clock, o PcAddr = 0x04 mostrando que houve o pulo. Desse modo, a simulação foi bem-sucedida.

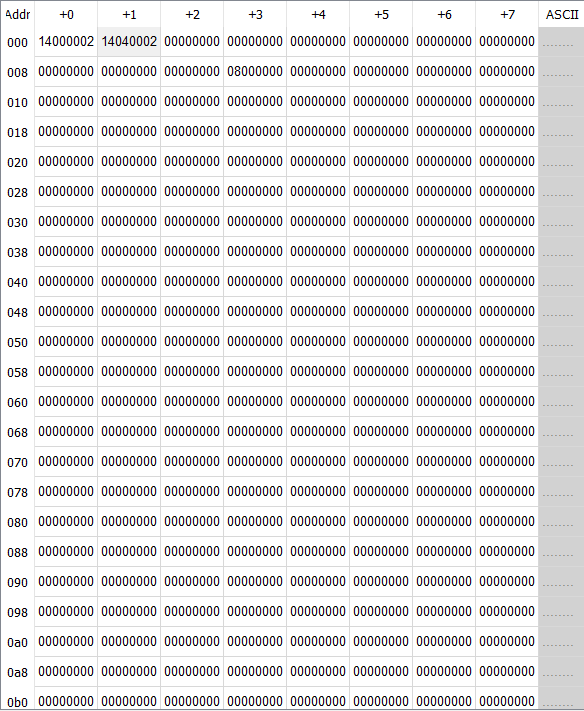


Figura 3.1 - bne.mif

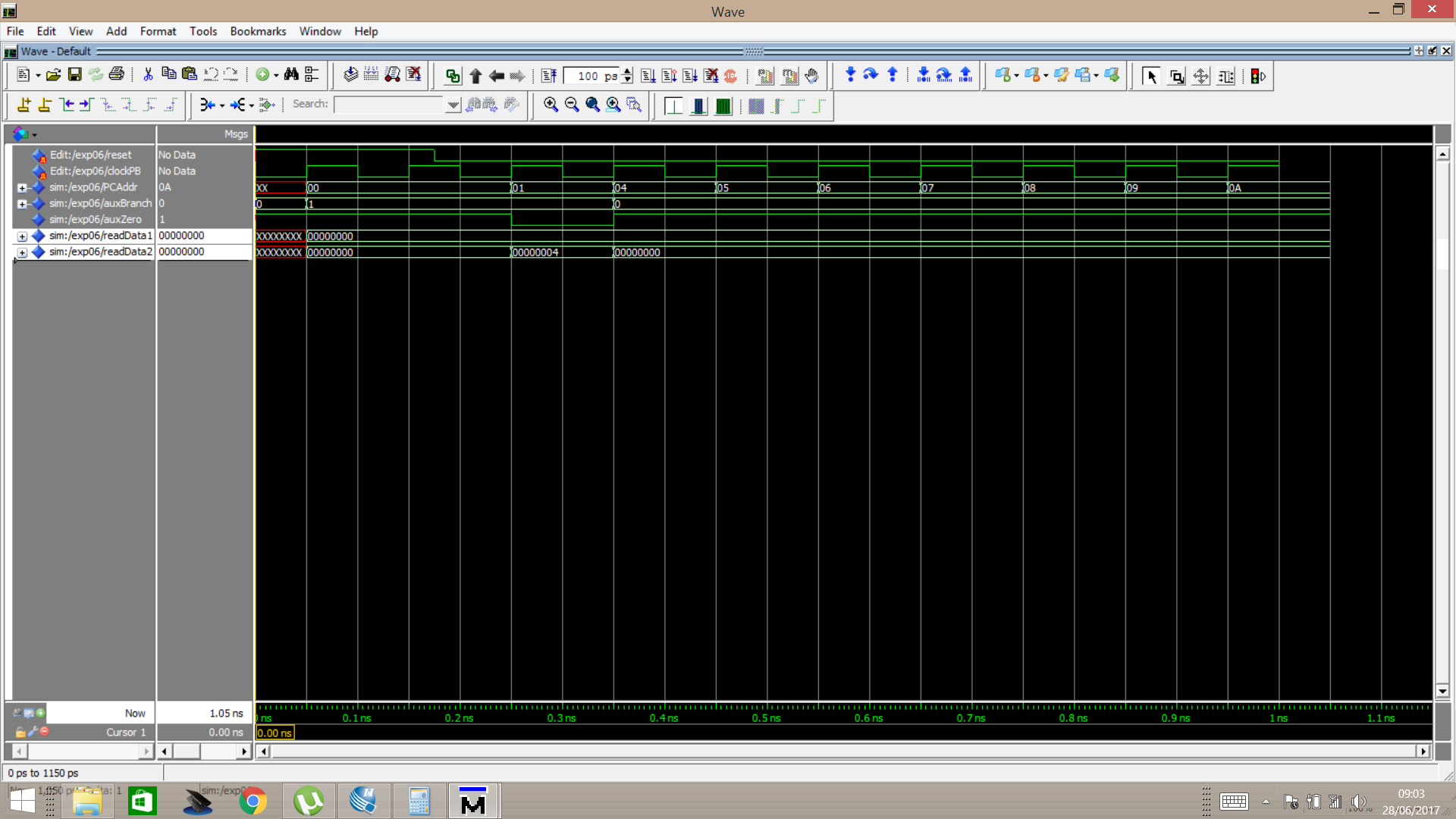


Figura 3.2 - waves do bne

## 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, e 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, ou seja, o Jr foi realizado como o esperado.

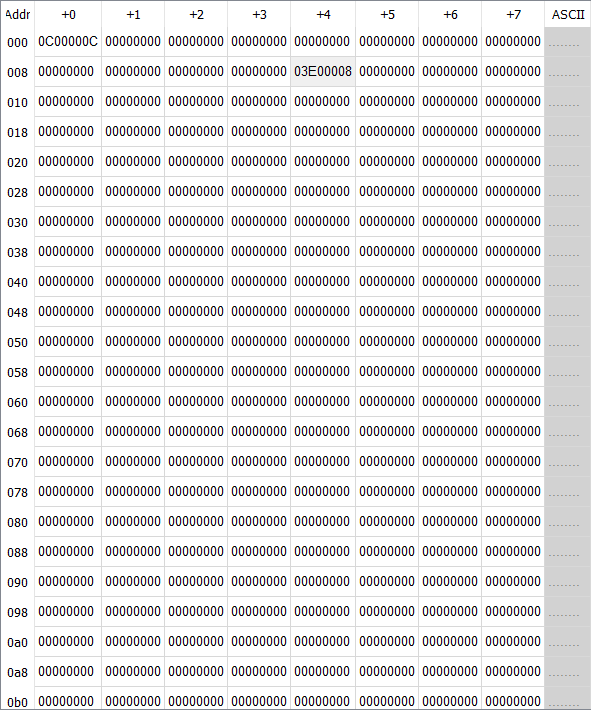


Figura 4.1 - jaljr.mif

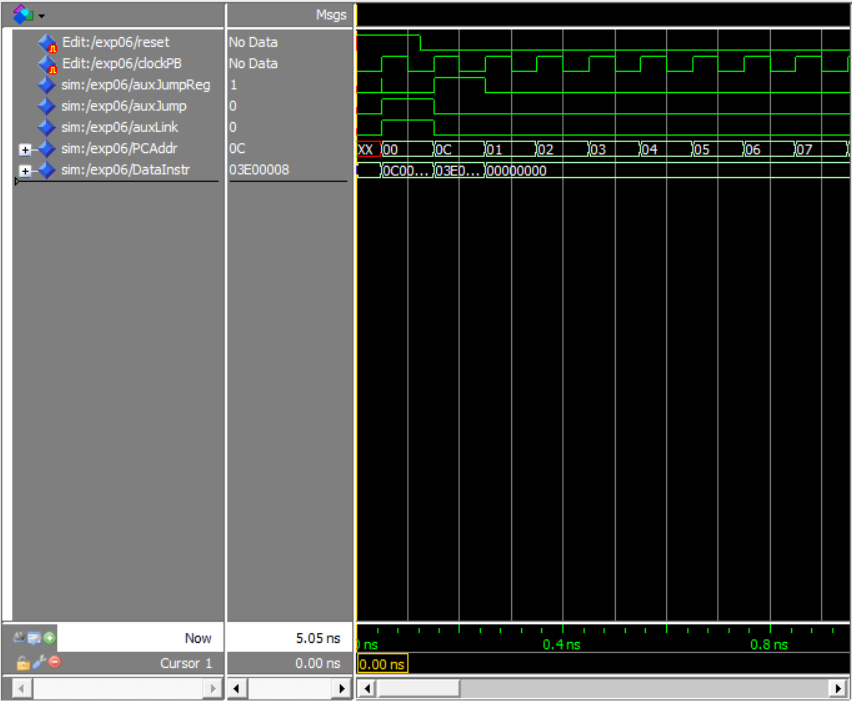


Figura 4.2 - waves do jal e do jr

## 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 1001, e ao deslocar duas vezes para a direita, 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.

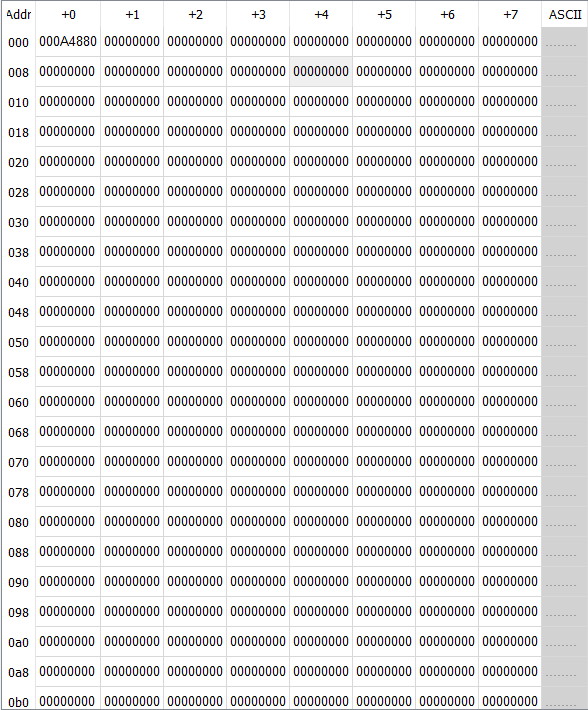


Figura 5.1 - sll.mif

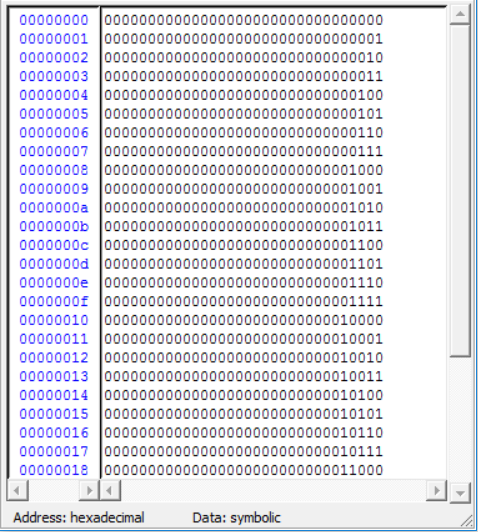
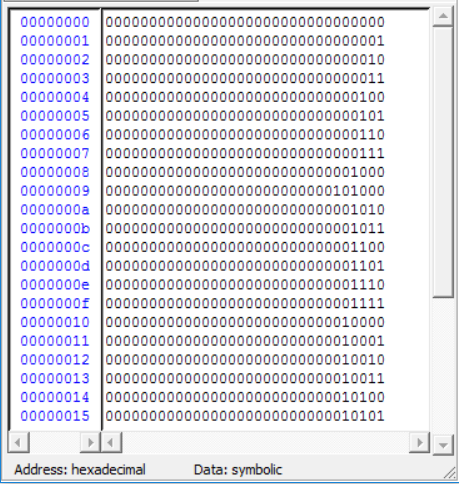
 



Figura 5.2 - Banco de registradores antes Figura 5.3 - Banco de registradores depois

## 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 esquerda, 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.

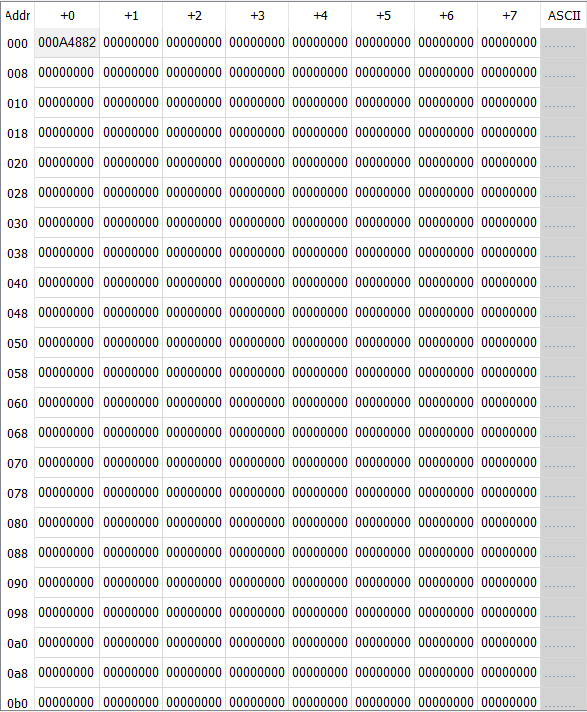


Figura 6.1 - srl.mif

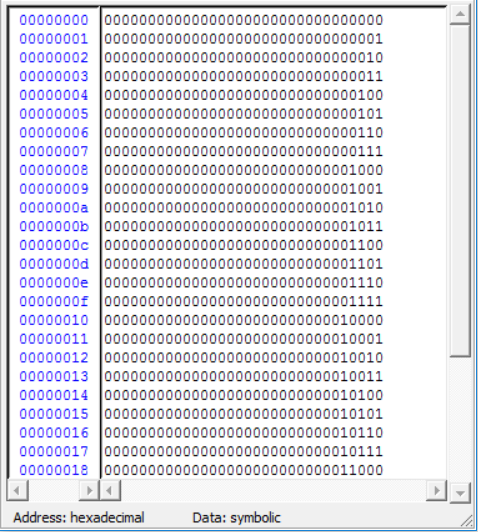
 



Figura 6.2 - Banco de registradores antes Figura 6.3 - Banco de registradores depois

## 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, porque o conteúdo de $0xD > $0xA, e após a terceira instrução é escrito o valor 0x1, pois o conteúdo $0xA < $0xD. A segunda instrução está nula (nop) para facilitar a visão do resultado no banco de registradores. Portanto, a operação executou de maneira correta e também foi escrito no registrador certo.

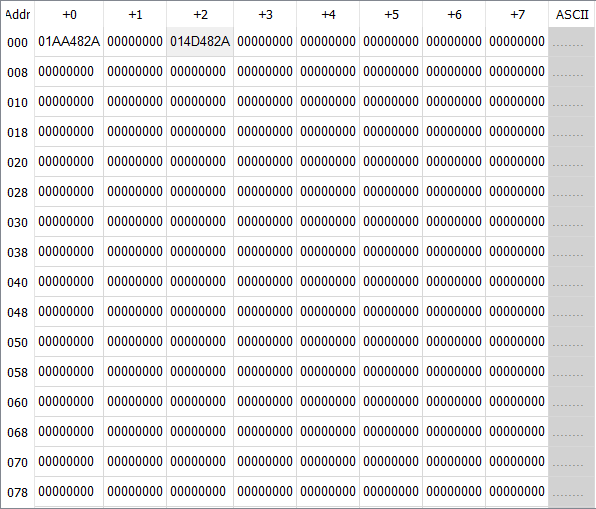


Figura 7.1 - slt.mif

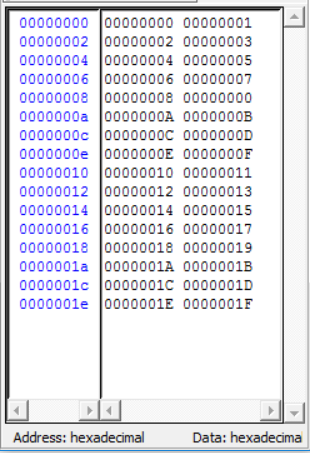
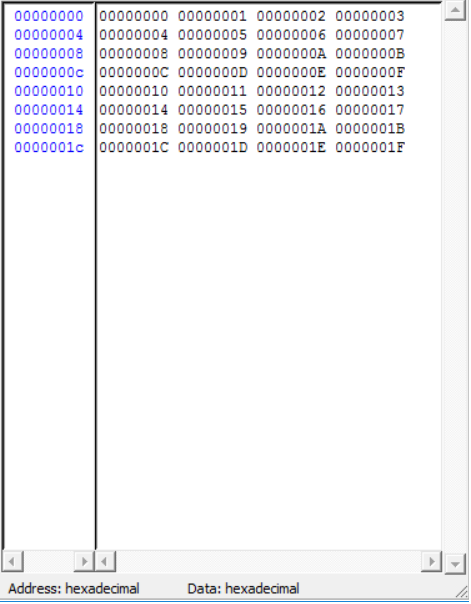




Figura 7.2 - Banco de registrador inicialmente Figura 7.3 - Banco de registradores após a instrução 0x0

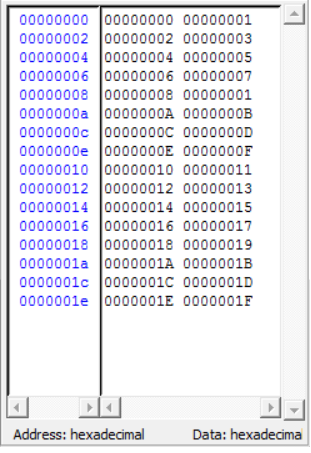




Figura 7.4 - Banco de registradores após a instrução 0x2

#### 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 esse resultado no registrador destino $0x9, que 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.

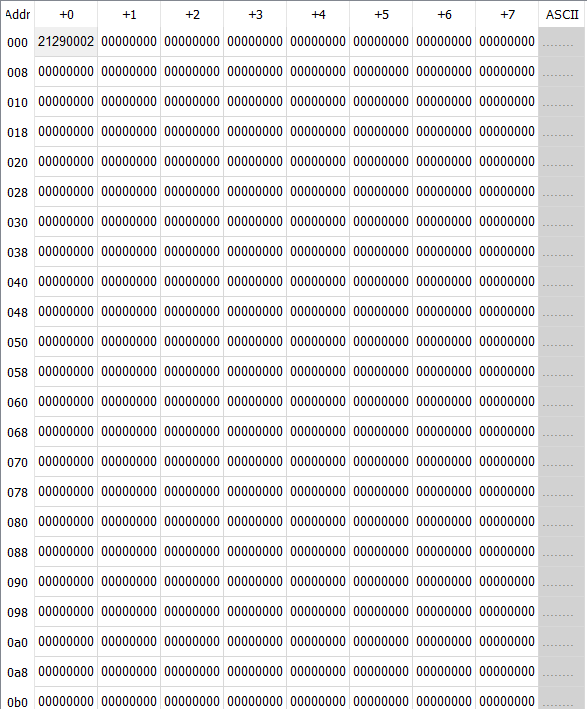


Figura 8.1 - addi.mif

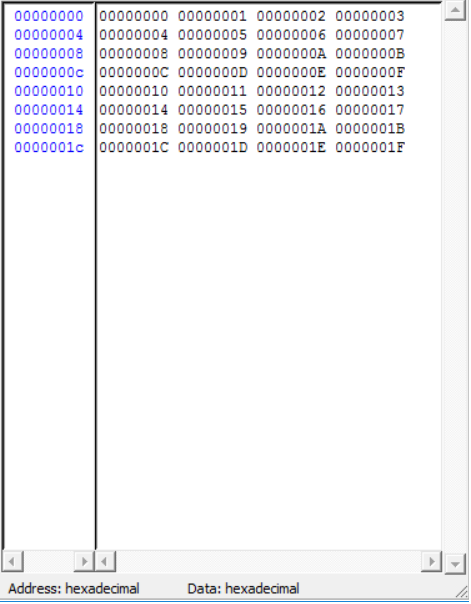
 



Figura 8.2 - Banco de registradores inicialmente Figura 8.3 - Banco de registradores após a operação

**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.

# 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**.

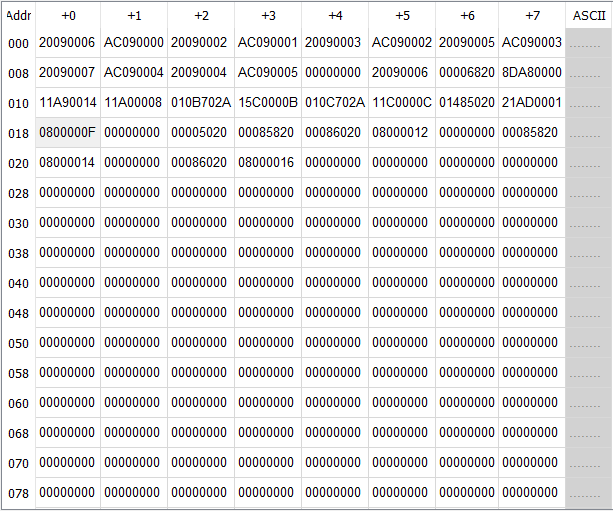


Figura 9 - vetor.mif

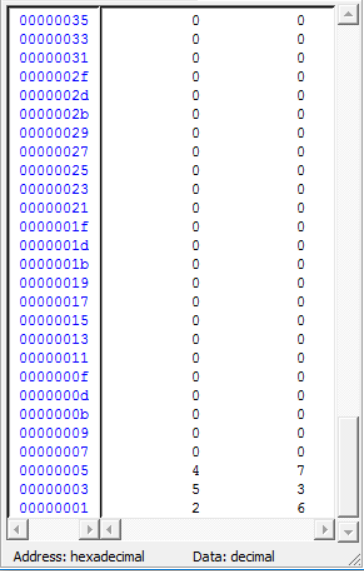
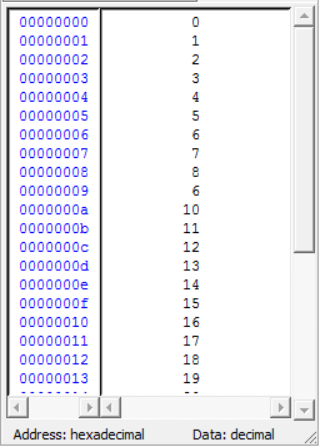
 

Figura 10 – Memória Figura 11 - Banco de registrador inicial

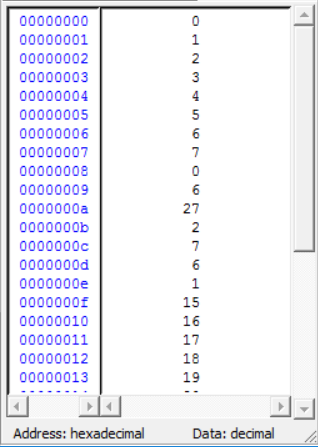


Figura 12 - Banco de registradores após a execução do programa

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 monto 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 indicado 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.

# 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.

# 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 MIPS proposto originalmente. Todos os obstáculos foram contornados, com auxílio do livro “VHDL: Descrição e Síntese de Circuitos Digitais. ” [1].

Além das mudanças no *Ifetch*, *Idecode*, *Control* e *Execute,* aentidade 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* realizado 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