Organização e Arquitetura de Computadores
Universidade de Brasília - 2020/2
João Tito do Nascimento Silva
Matrícula: 18/0123301

Trabalho final

Implementação de um processador RISC-V uniciclo em VHDL

Conteúdo

1 Objetivos

O objetivo desse trabalho é obter maior conhecimento sobre a estrutura do processador RISC-V e seu funcionamento por meio de uma implementação em VHDL.

2 Metodologia

A metodologia de desenvolvimento utilizada foi majoritariamente behavioral. No entanto, a entidade que representa o RISC-V completo é constituída apenas de sinais ligando os componentes, em um estilo mais estrutural. Inicialmente, foram criados alguns componentes mais básicos, como o PC, o somador e o multiplexador. Após isso, os componentes desenvolvidos fariam mais sentido de serem testados junto ao RISC-V completo, como a unidade de controle. Assim, foi criado um testbench e uma entidade representando o RISC-V completo que foi, aos poucos, moldada para executar o código de exemplo fornecido pelo professor. Para manter o rastro do desenvolvimento também foi utilizado o Git(https://github.com/titosilva/riscv-uniciclo).

3 Sobre este documento

Este documento contém a descrição do que foi implementado e como foi implementado. Há várias seções, sendo que cada uma delas descreve um componente e problemas diferentes e como foram resolvidos. Em especial, a seção "Notas de implementação"contém uma descrição dos maiores problemas encontrados.

4 Contador de programa (PC)

4.1 Descrição

O contador de contém uma referência para a instrução a ser executada. É um registrador de 32 bits. Entretanto, pelas restrições de memória adotadas, apenas o número de bits necessário será enviado à memória de instruções, sendo o restante ignorado.

4.2 Implementação

Para a implementação, foi criado um componente no VHDL com as seguintes portas:

1. Clock (clock)
2. Write Enabled (we): habilita escrita
3. DataIn (datain): entrada de dados
4. DataOut (dataout): saída de dados

O sinal Write Enabled não seria realmente necessário no RISC-V, pois não há sinal de controle correspondente, no entanto foi mantido na implementação para ficar mais semelhante a um registrador comum.

4.3 Testes

Para testar o componente de contador de programa, foi criado um testbench simples aplicando alguns valores possíveis.

5 Somador (SUM)

5.1 Descrição

O somador é um componente que recebe duas entradas e as soma. Nesta implementação, serão utilizados 2 somadores de 32 bits para operar com endereços.

5.2 Implementação

Para a implementação, foi criado um componente no VHDL com as seguintes portas:

1. DataIn0 (datain0): entrada de dados
2. DataIn1 (datain1): entrada de dados
3. DataOut (dataout): saída de dados

Foram utilizadas as conversões e operações definidas pela biblioteca numeric_std.

5.3 Testes

Para testar o componente de contador de programa, foi criado um testbench simples aplicando alguns valores possíveis.

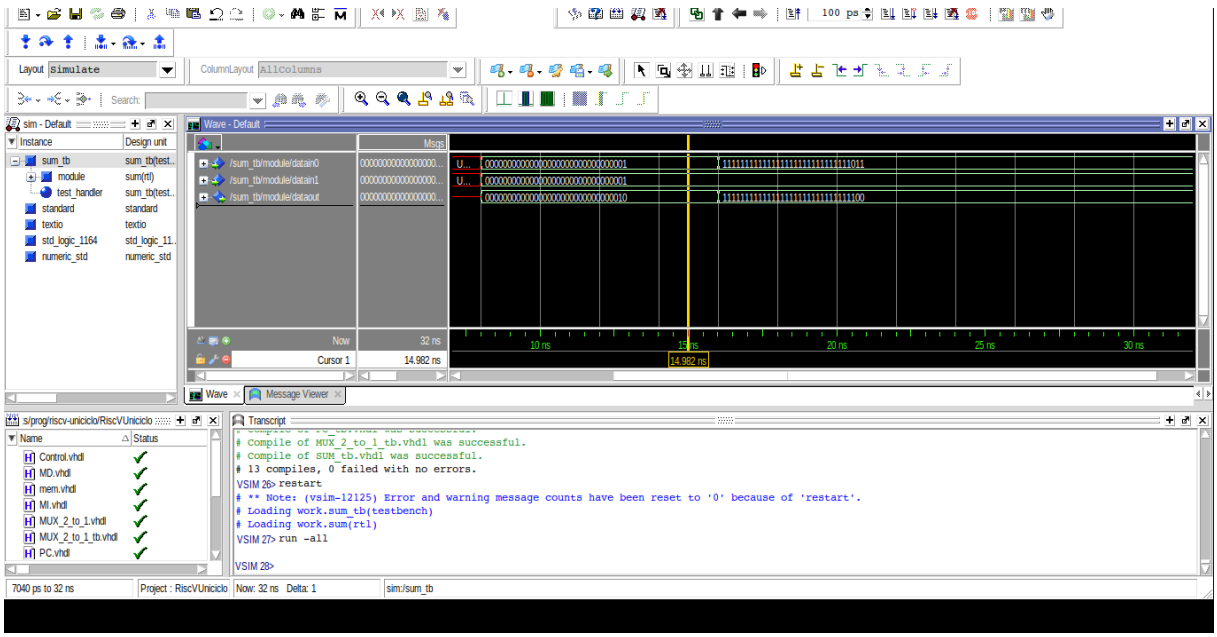

Figura 1: Somador - simulação do somador com soma de valores positivos e soma de um valor positivo com um negativo.

6 Multiplexador (MUX)

6.1 Descrição

O multiplexador é um componente que seleciona, entre duas entradas, a entrada selecionada a partir de uma terceira entrada seletora. Nesta implementação serão utilizados 4 multiplexadores com 2 entradas de 32 bits e uma saída de 32 bits.

6.2 Implementação

Para a implementação, foi criado um componente no VHDL com as seguintes portas:

1. DataIn0 (datain0): entrada de dados
2. DataIn1 (datain1): entrada de dados
3. Seletor (seletor): entrada de seleção
4. DataOut (dataout): saída de dados

6.3 Testes

Para testar o componente de contador de programa, foi criado um testbench simples aplicando níveis alto e baixo no seletor e também trocando os valores nas entradas.

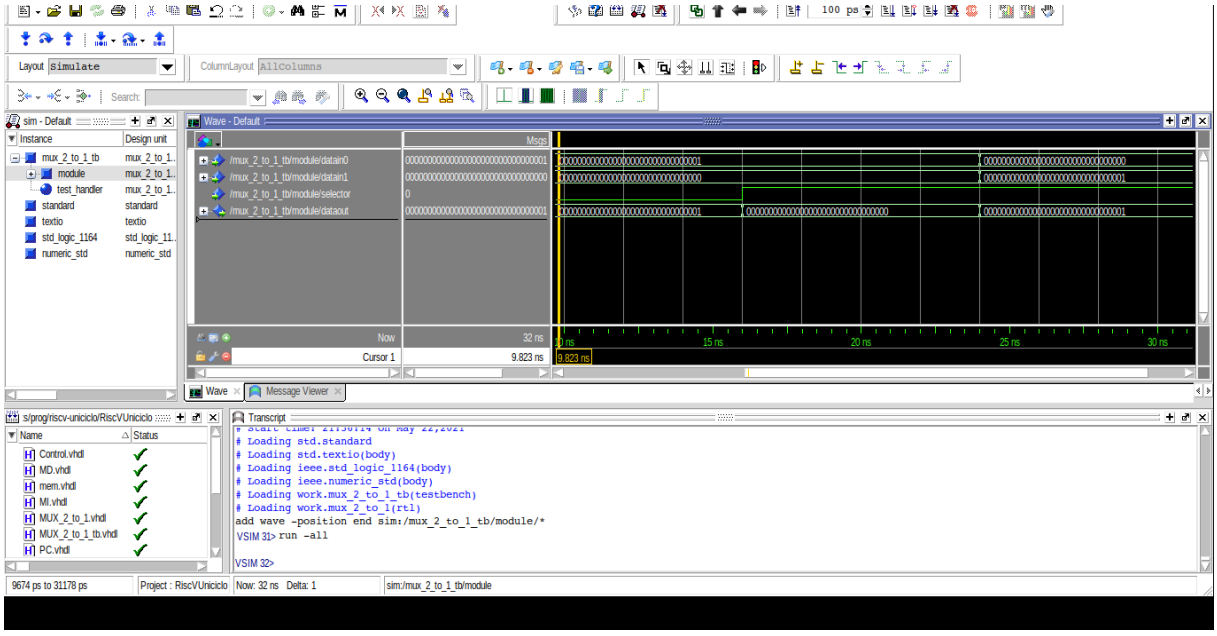

Figura 2: Multiplexador - simulação do multiplexador com diferentes seleções e entradas

7 Memória de instruções (Mem - MI)

7.1 Descrição

A memória de instruções armazena as intruções a serem executadas pelo processador. Ela é acessada por meio de um endereço fornecido pelo PC (contador de programa).

7.2 Implementação

A memória foi implementada como uma ROM por meio de um componente no VHDL com as seguintes portas:

1. Clock (clock)
2. Address (address): endereço a ser lido
3. DataOut (dataout): saída de dados

Esse componente já havia sido implementado em outro trabalho e será reutilizado com alterações, caso necessário. Note a adição do sinal de trigger, que serve para resolver o problema de propagação de dados descrito na seção de notas de implementação do RISC-V completo.

7.3 Testes

Como o componente já havia sido testado em outro trabalho, os testes não serão refeitos.

8 Memória de dados (Mem - MD)

8.1 Descrição

Consiste de uma memória que armazena dados utilizados na execução do programa, e pode tanto ser lida quanto escrita. A memória de dados também deve ser inicializada de acordo com o modelo de memória compacta do RARS, no qual o endereço de base é 0x2000.

8.2 Implementação

A memória foi implementada como um componente no VHDL com as seguintes portas:

1. Clock (clock)
2. Write Enabled (we): habilita a escrita
3. DataIn (datain): entrada de dados
4. Address (address): endereço a ser lido
5. DataOut (dataout): saída de dados

A implementação foi mais complexa que a de outros componentes. A inicialização da memória estava falhando muito em uma primeira implementação. Aparentemente, o problema estava relacionado à interpretação do ambiente de runtime sobre instruções de atribuição de sinais. Os sinais estavam sendo atribuídos quando não deviam, causando bugs na inicialização. Assim, para evitar esse problema, a memória de dados foi dividida em duas partes. A primeira parte é inicializada, mas não pode ser escrita. A segunda parte não é inicializada, e pode ser escrita. Assim, as operações de escrita eram todas

realizadas sobre a segunda parte. Para que o componente soubesse de qual das duas partes deveria pegar um dado, era necessário manter uma lista dos endereços onde haviam ocorrido escritas. Isso foi feito por meio de uma array de std_logic na qual os endereços onde havia bit 1 já haviam sido escritos.
Note, também, a adição do sinal de trigger, que serve para resolver o problema de propagação de dados descrito na seção de notas de implementação do RISC-V completo.

8.3 Testes

Como o componente já havia sido testado em outro trabalho, os testes não serão refeitos.

9 Banco de registradores (XREG)

9.1 Descrição

O banco de registradores consiste de uma estrutura com 32 registradores. Cada registrador pode ser lido ou escrito por meio de um endereço. O registrador no endereço 0 sempre retorna 0. Além disso, para a execução de instruções que envolvem mais de um registrador, é possível ler dois registradores ao mesmo tempo.

9.2 Implementação

O banco de registradores foi implementado de forma similar à memória de dados, apenas fazendo as modificações necessárias para compatibilidade. As seguintes portas foram criadas no componente em VHDL:

1. Clock (clock)
2. Write Enabled (we): habilita a escrita
3. Endereço 1 (address1): endereço de um dos registradores a ser lido
4. Endereço 2 (address2): endereço de um dos registradores a ser lido
5. Endereço de escrita (addressWrite): endereço do registrador onde escrever
6. Dados de escrita (writeData): dados a serem escritos no registrador com endereço dado por addressWrite
7. Saída de dados 1 (dataout1): saída para a leitura do registrador dado pelo endereço address1
8. Saída de dados 2 (dataout2): saída para a leitura do registrador dado pelo endereço address2

Note a adição do sinal de trigger, que serve para resolver o problema de propagação de dados descrito na seção de notas de implementação do RISC-V completo.

9.3 Testes

Para realizar testes, foi criado um testbench com algumas possibilidades de uso.

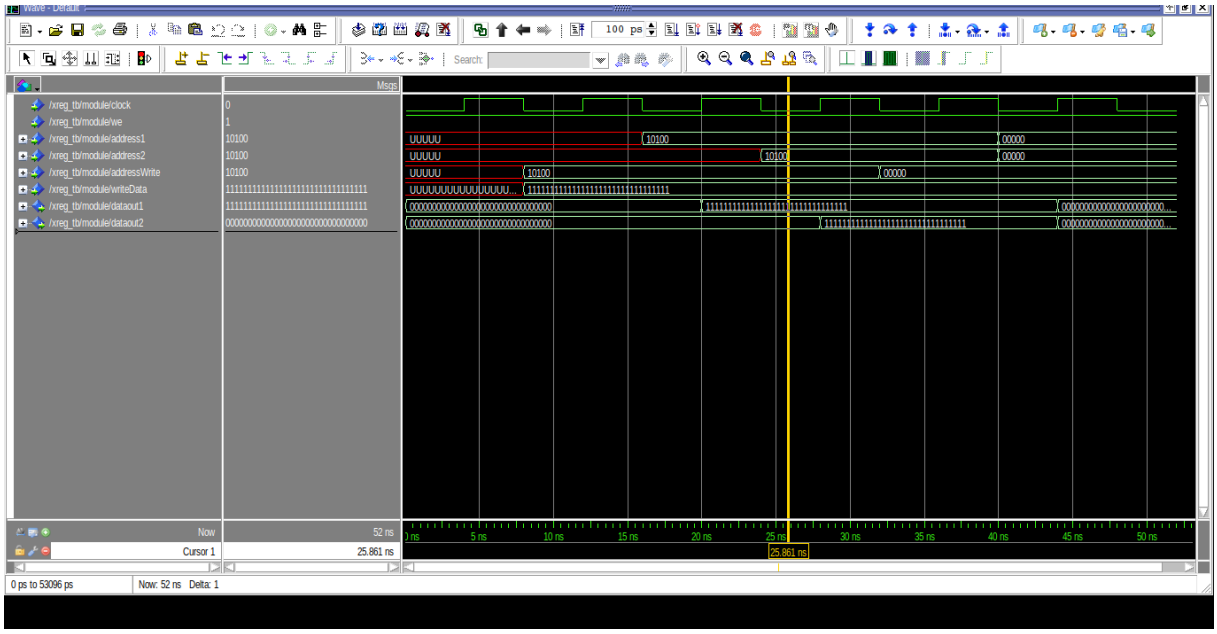

Figura 3: Banco de registradores - simulação do banco de registradores com uma escrita, uma leitura em cada saída disponível, uma tentativa de escrita em 0 e uma leitura em 0.

10 ULA

10.1 Descrição

A Unidade lógico-aritmética realiza operações necessárias à execução de uma instrução.

10.2 Implementação

A implementação já havia sido feita em outro trabalho, e será reutilizada aqui. As portas são:

1. Código de operação (opcode): entrada que indica o tipo da instrução sendo executada
2. Entradas de dados (A, B): são as entradas para os dados a serem operados
3. Resultado (Z): saída do resultado da operação
4. Zero (zero): saída que indica se o resultado foi nulo

10.3 Testes

Esse componente já havia sido testado em outro trabalho e, por isso, não será testado novamente aqui por meio de um testbench.

11 Unidade de controle (Control)

11.1 Descrição

A unidade de controle é o componente que gera os sinais de controle para cada componente, regindo a forma como se comportam com o objetivo de realizar a execução de uma instrução corretamente.

11.2 Implementação

A unidade de controle foi implementada como um componente de VHDL com uma porta de entrada, que corresponde ao Opcode da operação sendo executada, e com muitas portas de saída, sendo cada porta destas um sinal de controle para algum componente. A implementação é relativamente simples e consiste de cases aninhados.

11.3 Testes

Esse componente tem uma implementação relativamente simples, apesar de extensa. Por isso, não foi feito um testbench. A visualização será feita juntamente com os testes do RISC-V completo.

12 Controle da ULA (ULAControl)

12.1 Descrição

Para controlar as operações realizadas pela ULA, é utilizado um componente que, dado um Opcode e os campos funct7 e funct3 da operação sendo realizada, gera um sinal novo que é utilizado pela ULA como seletor da operação a ser executada. O opcode recebido é gerado pela unidade de controle.

12.2 Implementação

O controle da ULA foi implementado como um componente no VHDL com as seguintes portas:

1. Opcode (aluop): opcode gerado pela unidade de controle
2. Funct3 (funct3)
3. Funct7 (funct7)
4. Controle da ULA (aluctr): sinal gerado que é utilizado pela ULA para saber qual operação realizar

Note que, apesar de não ser necessário utilizar os campos funct3 e funct7 completos, estes foram mantidos completos como entradas para tornar a implementação e o entendimento do código mais simples. Um código dado em sala de aula já mostrava uma implementação desse componente, que foi modificava para o caso específico desse projeto. Note que foi utilizado o código "11"para instruções de tipo I, devido à necessidade de tratar casos mais específicos.

12.3 Testes

Dado que o componente tem funcionalidade simples, não foi feito um testbench para esse componente. A visualização será feita juntamente com os testes do RISC-V completo.

13 Gerador de Imediatos (GenImm32)

13.1 Descrição

Para gerar os imediatos das operações que envolvem uso de imediatos é necessário um componente. Esse componente deve receber a instrução e gerar os imediatos conforme o tipo de instrução.

13.2 Implementação

Esse componente havia sido implementado em outro trabalho no VHDL, e possui as seguintes portas:

1. Instrução (instr): entrada para a instrução em execução
2. Saída de imediato (imm32): saída para o imediato gerado

Foi necessário realizar a implementação mais específica para as instruções srli, srai e slli.

13.3 Testes

Para esse componente não foi criado um testbench, pois sua funcionalidade é simples e pode ser facilmente testada e verificada ao executar os testes do RISC-V completo.

14 Entidade RISC-V

14.1 Descrição

O processador Risc-v em si foi implementado como uma entidade no Modelsim com apenas uma entrada, que corresponde ao clock. Conforme a implementação era feita, um testbench auxiliava na verificação dos códigos. Esse testbench foi construído para executar o código abaixo:

```
.data
vetx: .word 15 63

# os valores ao lado sao tomados da saida dos modulos:
# PC - saida do PC
# MI - saida do modulo de memoria (e nao de IF/ID)
# ULA - saida direta da ULA
# MD - saida direta da memoria de dados

.text                       # PC         MI          ULA        MD

  ori  t0, zero, 0xFF       # 00000000   0ff06293    000000ff   00000000 OK
  andi t0, t0, 0xF0         # 00000004   0f02f293    000000f0   00000000 OK
  lui  s0, 2                # 00000008   00002437    00002000   00000000 OK
```

```
15   lw   s1, 0(s0)           # 0000000c  00042483   00002000  0000000f OK
16   lw   s2, 4(s0)           # 00000010  00442903   00002004  0000003f OK
17   add  s3, s1, s2          # 00000014  012489b3   0000004e  00000000 OK
18   sw   s3, 8(s0)           # 00000018  01342423   00002008  0000004e OK
19   lw   a0, 8(s0)           # 0000001c  00842503   00002008  0000004e OK
20   addi s4, zero, 0x7F0     # 00000020  7f000a13   000007f0  00000000 OK
21   addi s5, zero, 0x0FF     # 00000024  0ff00a93   000000ff  00000000 OK
22   and  s6, s5, s4          # 00000028  014afb33   000000f0  00000000 OK
23   or   s7, s5, s4          # 0000002c  014aebb3   000007ff  00000000 OK
24   xor  s8, s5, s4          # 00000030  014acc33   0000070f  00000000 OK
25   slli  t1, s5, 4          # 00000034  004a9313   00000ff0  00000000 OK
26   lui  t2, 0xFF000         # 00000038  ff0003b7   ff000000  00000000 OK
27   srli  t3, t2, 4          # 0000003c  0043de13   0ff00000  00000000 OK
28   srai  t4, t2, 4          # 00000040  4043de93   fff00000  00000000 OK
29   slt  s0, t0, t1          # 00000044  0062a433   00000001  00000000 OK
30   slt  s1, t1, t0          # 00000048  005324b3   00000000  00000000 OK
31   sltu s3, zero, t0        # 0000004c  005039b3   00000001  00000000 OK
32   sltu s4, t0, zero        # 00000050  0002ba33   00000000  00000000 OK
33
34   jal  ra, testasub        # 00000054  008000ef   00000000  00000000 => 5
      c -> Verificar saida da ULA. JUMP
35
36   jal  x0, next            # 00000058  00c0006f   00000000  00000000 =>
      64 JUMP OK!
37 testasub:
38   sub t3, t0, t1           # 0000005c  40628e33   fffffffe  00000000 ->
      Aparentemente, essa linha tem um erro. O RARS resulta em fffff100. OK
      !
39   jalr x0, ra, 0           # 00000060  00008067   00000058  00000000 =>
      58 JUMP OK!
40 next:
41   addi t0, zero, -2        # 00000064  ffe00293   fffffffe  00000000 OK!
42 beqsim:
43   addi t0, t0, 2           # 00000068  00228293   0000000*  00000000 * t0
       = 0(OK), 2(OK)
44   beq  t0, zero, beqsim    # 0000006c  fe028ee3   00000000  00000000 =>
      68(JUMP OK!), 70(JUMP OK!)
45 bnesim:
46   addi t0, t0, -1          # 00000070  fff28293   0000000*  00000000 * t0
       = 1(OK), 0(OK)
47   bne  t0, zero, bnesim    # 00000074  fe029ee3   00000000  00000000 =>
      70(OK), 78(OK)
48
49
50
```

14.2 Notas de implementação

Alguns problemas foram observados durante a implementação e estão descritos abaixo.

14.2.1 Deslocamento do imediato para instruções JAL

As instruções JAL e JALR exigem que o PC seja alterado conforme um *offset*, que é calculado pelo gerador de imediatos. Pelo modelo dado em sala de aula, esse resultado do gerador de imediatos deveria ainda ser deslocado antes de ser somado ao PC para gerar o novo endereço de instrução, mas durante a implementação isso gerou problemas.

Aparentemente, esse deslocamento não era realmente necessário por causa do modo como o gerador de imediatos foi codificado. O gerador de imediatos automaticamente adiciona um bit 0 ao final do imediato, tornando-o um múltiplo de 2. O deslocamento a mais causou saltos mais longos que o esperado, e por isso, foi removido.

14.2.2 Insuficiência do modelo para execução de instrução de salto incondicional

Para a correta execução das instruções JAL e JALR, foi necessário acrescentar alguns multiplexadores à arquitetura. Um multiplexador seleciona a entrada de dados do banco de registradores entre o resultado de *write-back* e o valor de PC+4. No caso de instruções JAL e JALR, o valor PC+4 deve ser selecionado para salvar o endereço de retorno no banco de registradores. Além disso, outro multiplexador seleciona a origem do valor a ser somado com o *offset* para gerar o novo valor de PC. Para a instrução JAL, esse valor é o próprio PC. Já para a instrução JALR, esse valor é rs1.

14.2.3 Valores de controle

Para a execução de algumas instruções, os valores de controle apresentados em sala de aula tiveram que ser alterados. Esse foi o caso das instruções tipo B, tipo I e tipo R, para as quais os valores dos códigos de operação gerados para a ULA tiveram que ser alteraos. Além disso, foram adicionados os controles *is_jalx* e *is_jalr* para implementação das instruções de JAL e JALR conforme a insuficiência no modelo discutida anteriormente.

14.2.4 Propagação de dados

Quando uma nova instrução era fornecida pela memória de instruções, os outros componentes deveriam reagir ao novo dado e executar a instrução corretamente. No entanto, alguns componentes executam suas mudanças no tempo de subida do clock, de modo que utilizavam dados antigos para realizar essas mudanças. Como exemplo, o banco de registradores estava retornando um valor com base no endereço vindo do clock anterior pois reagia logo na subida do clock ao invés de aguardar a mudança da instrução pela memória de instruções. Como solução para esse problema, foi adicionado um sinal de "trigger"para cada componente com lógica síncrona. Esse sinal reage à subida do clock alterando a si mesmo, porém com um pequeno atraso e é utilizado na lista de sensibilidade dos processos principais do componente. Esse pequeno atraso faz com que o componente reaja com um pequeno atraso com relação ao clock, porém com um dado atualizado.

14.3 Simulação no VHDL

A simulação foi realizada por meio da visualização de onda do ModelSim utilizando o código em Assembly apresentado anteriormente.

14.3.1 Sequência de instruções ori, andi, lui, lw

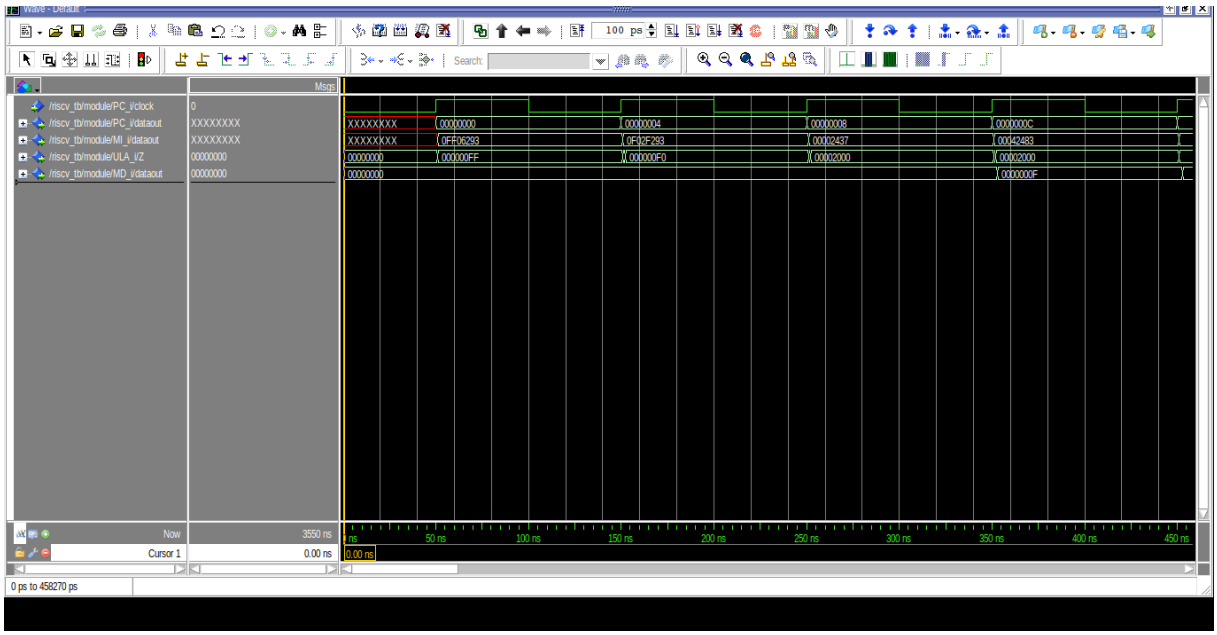

Figura 4: Simulação das instruções ori, andi, lui, lw

14.3.2 Sequência de instruções lw, add, sw, lw

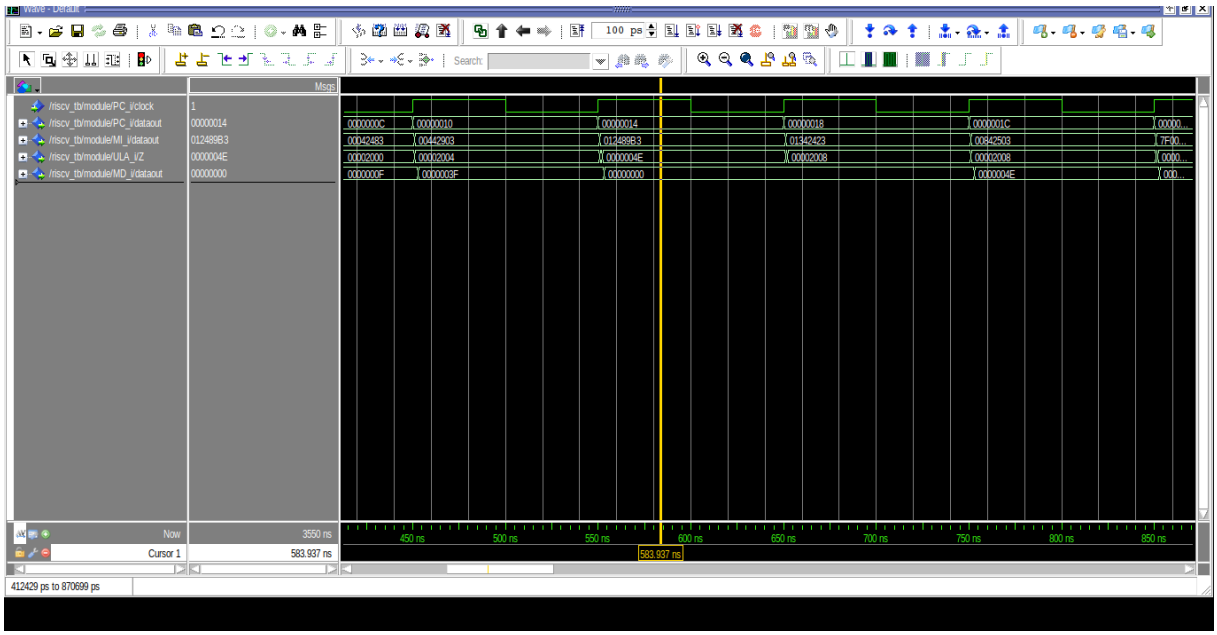

Figura 5: Simulação das instruções lw, add, sw, lw

14.3.3 Sequência de instruções addi, addi, and, or

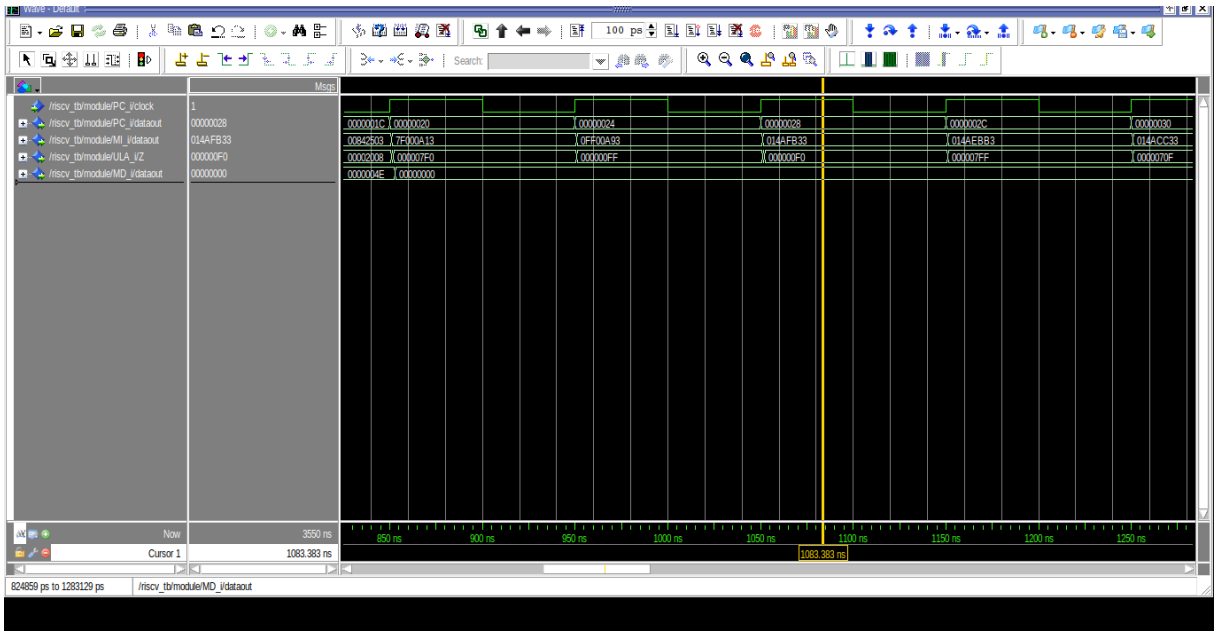

Figura 6: Simulação das instruções addi, addi, and, or

14.3.4 Sequência de instruções xor, slli, lui, srli

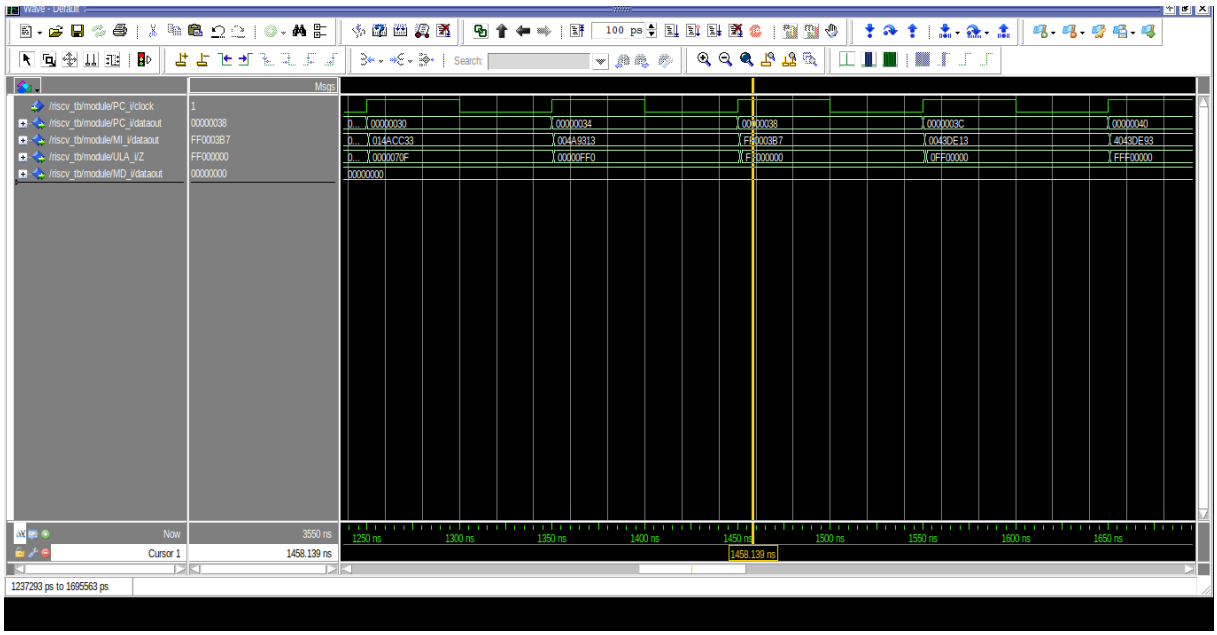

Figura 7: Simulação das instruções xor, slli, lui, srli

14.3.5 Sequência de instruções srai, slt, slt, sltu

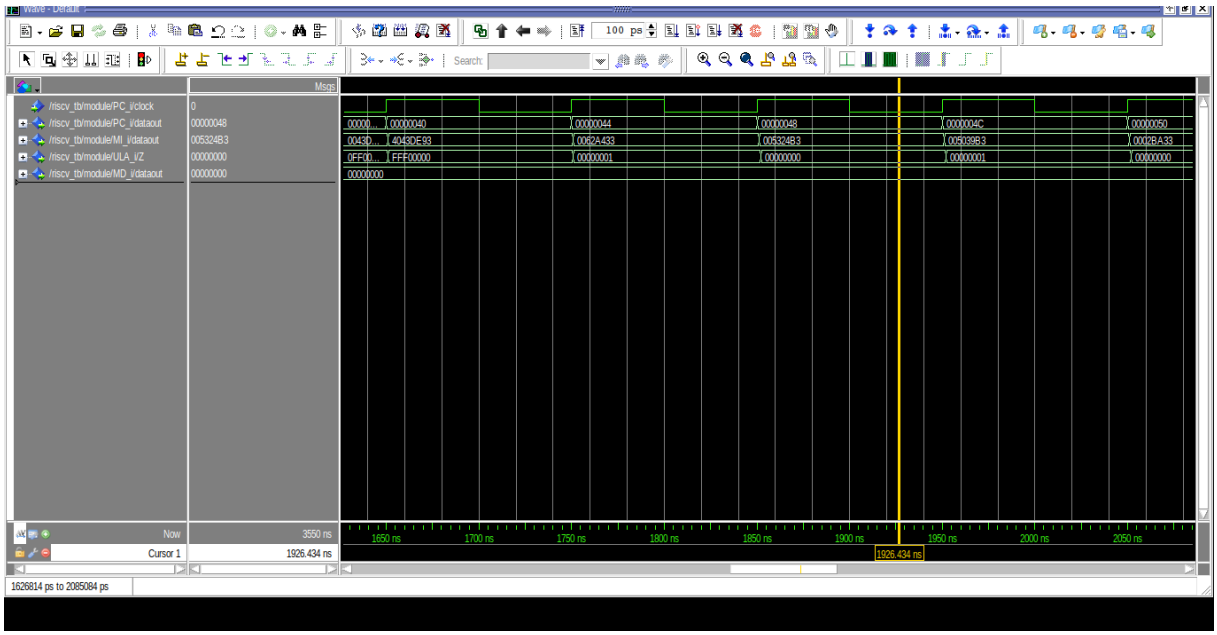

Figura 8: Simulação das instruções srai, slt, slt, sltu

14.3.6 Sequência de instruções sltu, jal, sub com salto

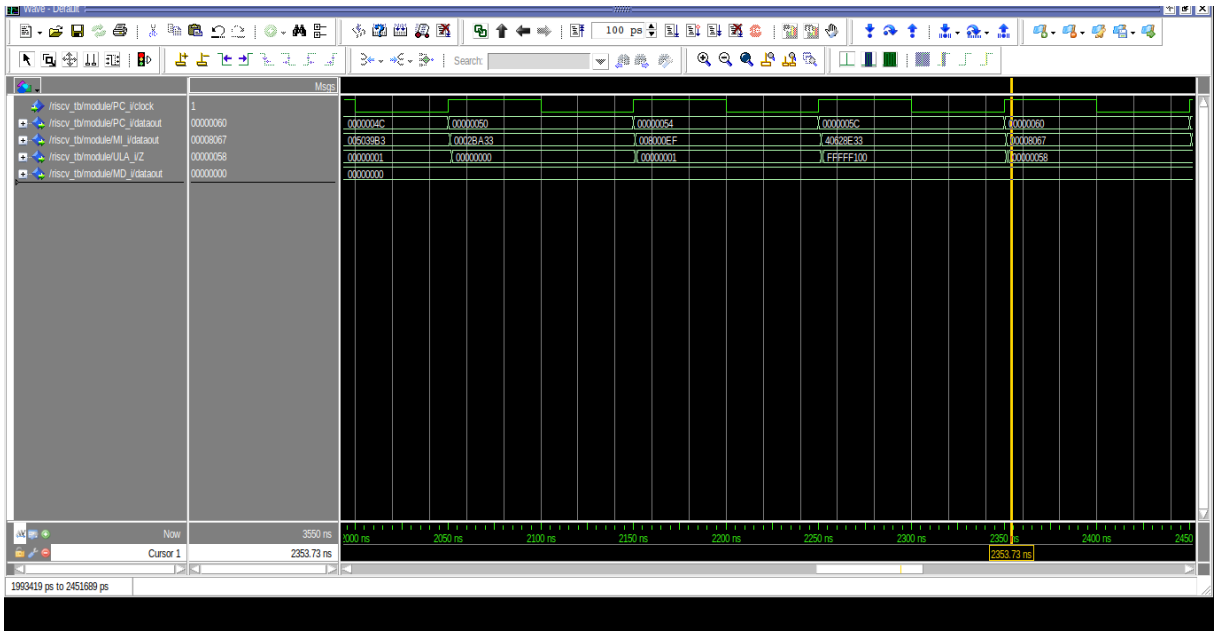

Figura 9: Simulação das instruções sltu, jal, sub com salto. Note o salto de 0x054 para 0x05C devido à instrução jal.

14.3.7 Sequência de instruções jalr, jal, addi, addi com saltos

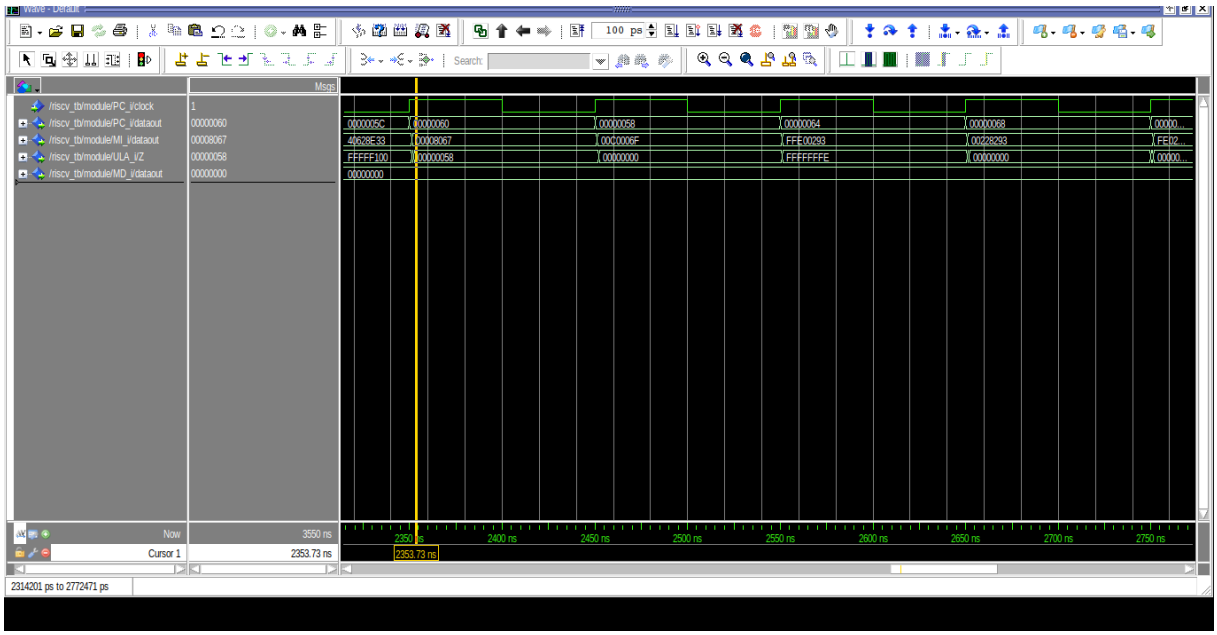

Figura 10: Simulação das instruções jalr, jal, addi, addi com saltos. Note o salto de 0x060 para 0x058 devido à instrução jalr e o salto de 0x058 para 0x064 devido à instrução jal.

14.3.8 Sequência de instruções beq, addi, beq, addi com saltos condicionais

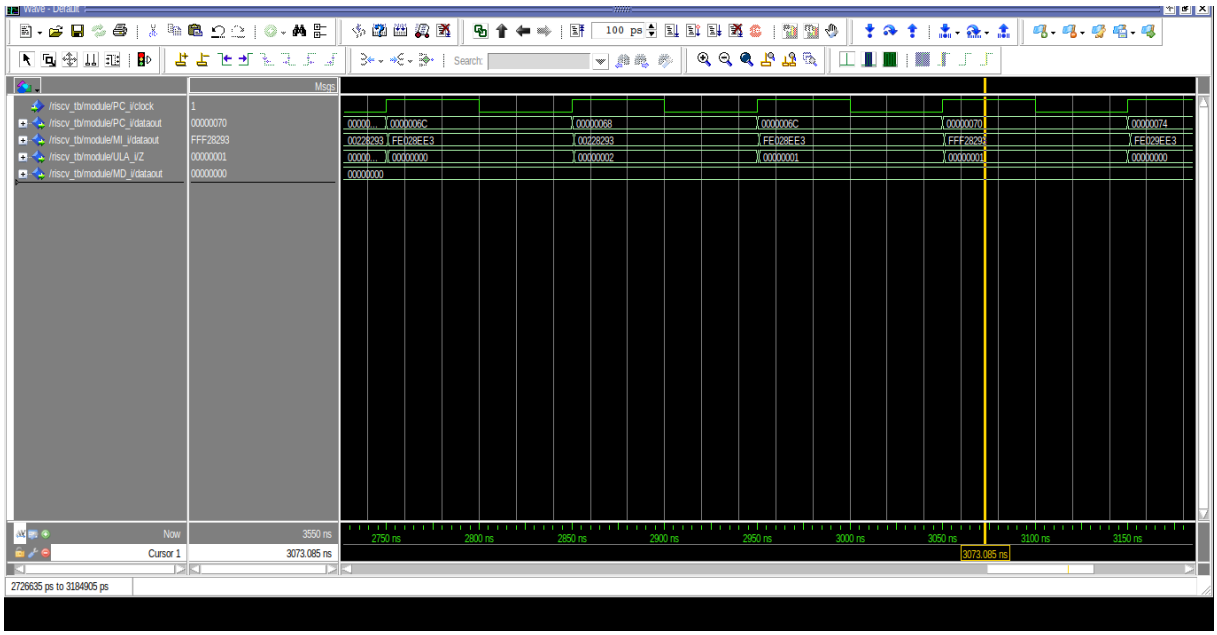

Figura 11: Simulação das instruções beq, addi, beq, addi com saltos condicionais. Note o salto de 0x06C para 0x068 devido à instrução beq, tomado uma primeira vez quando t0=0, mas não tomado na segunda vez, quando t0=2.

14.3.9 Sequência de instruções bne, addi, bne com saltos condicionais

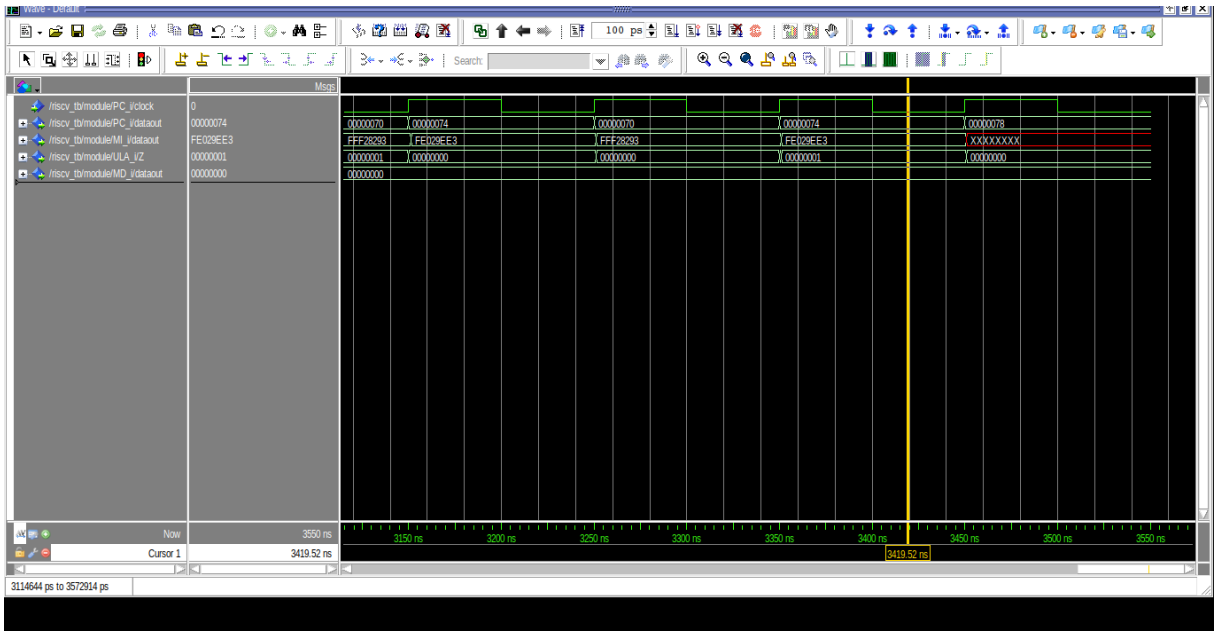

Figura 12: Simulação das instruções bne, addi, bne saltos condicionais. Note o salto de 0x074 para 0x070 devido à instrução bne, tomado uma primeira vez quando t0=1, mas não tomado na segunda vez, quando t0=0. A instrução 0x78 não existe, e resulta em um valor indefinido