Universidade Federal de Pernambuco – UFPE
Centro de Informática – CIn

Infraestrutura de Hardware – if674cc
Relatório de Projeto

Recife
05 de maio de 2011

Infraestrutura de hardware – if674cc
Especificação de Projeto

Documento desenvolvido pelo grupo de hardware sob orientação do monitor Húgaro Bernardino.

Escrito por:
Eric Ândersen (eafs),
Luana Martins (lms7),
Lucas Harada (lmfh),
Maria Fernanda (mfcc),
Ricardo Souza (rbs5).

Sumário

Introdução

Entender o funcionamento da arquitetura do computador e do seu processador é um importante pilar na formação de estudantes de Ciência da Computação. Para o conhecimento básico de como um processador executa programas, é fundamental estudar sua arquitetura interna, para a partir daí compreender a atuação e necessidade de softwares.

O desenvolvimento deste projeto objetiva proporcionar aos estudantes uma maior vivência, facilitando assim os estudos na área da Infraestrutura de Hardware, através de cartolinas, onde desenhamos e projetamos a arquitetura de nosso processador e seus estados de controle, e da implementação em Verilog, no Quartus II 8.0.

O processador projetado é composto de três blocos básicos:

* Unidade de processamento - formado basicamente por uma unidade lógica e aritmética (ULA), responsável pela manipulação de dados e realização das operações aritméticas e lógicas; e registradores responsáveis pelo armazenamento de informações para escrita e leitura;
* Unidade de controle - responsável pela interpretação do código da instrução e a ativação dos sinais de controle. Cada instrução é quem norteia os sinais do controle que serão ativados e qual a ordem deverá ser seguida.
* Memória - espaço destinado ao armazenamento temporário dos dados durante a execução de alguma instrução.

Este relatório é uma espécie de manual do funcionamento do processador desenvolvido pela equipe, deixando de maneira clara como cada entidade atua, dos circuitos e registradores utilizados, além de descrever o comportamento delas diante das instruções executadas.

Descrição das Entidades

1) **Store**

a) Entradas

Clk (1 bit): representa o clock do sistema.

Valor da memória (offset + $rs): calculado da instrução como endereço na memória a ser inserido o store byte.

Valor de rs (32 bits): valor advindo da memória, a partir da instrução, correspondendo aos bits [25:21] da instrução de store.

StoreControl(1 bit): sinal que representa o controle referido ao tipo de store requerido pelo sistema.

Tipos:
Store Byte (LoadControl = 0)
Store HalfWord (LoadControl = 1)

b) Saídas

StoreOut (32 bits): Saída para escrita na memória.

c) Objetivo

Essa entidade tem a função de ler algum dado no Banco de Registradores com a finalidade de guardar um byte ou uma halfword desse dado, em offset + $rs, advindo da instrução. Sendo assim, armazenado na memória.

Considerando apenas que o valor lido será uma parte do valor do endereço de memória para guardar no valor calculado por (offset + $rs) na memória.

d) Algoritmo

O StoreControl informará que tipo de store será feito. Assim sendo o valor a ser escrito na memória, ou seja, offset + $rs é somado na ULA.e o endereço em $rs é enviado ao Banco de Registradores para leitura. Assim, tanto o valor de offset +

$rs quanto o valor lido do registrador $rs é enviado ao módulo Store, e o controle irá enviar um sinal definindo o tipo de store que será feito. Então haverá uma concatenação dos bits menos significativos do registrador B (com o valor do registrador $rs)

2) Load

a) Entradas

Clk (1 bit): representa o clock do sistema.

Valor de rs (32 bits): valor advindo da memória, a partir da instrução, correspondendo aos bits [25:21] da instrução de load.

LoadControl (1 bit): sinal que representa o controle referido ao tipo de load requerido pelo sistema.

Tipos:
Load Byte (LoadControl = 0)
Load HalfWord (LoadControl = 1)

b) Saídas

LoadOut (32 bits): Saída para escrita no banco de registradores.

c) Objetivo

Essa entidade tem a função de ler algum dado na memória com a finalidade de guardar um byte ou uma halfword desse dado, em algum dos registradores do Banco. Assim, o valor poderá fica disponível para alguma operação na ULA por exemplo.

Considerando apenas que o valor lido será uma parte do valor do endereço de memória para guardar no registrador definido pela instrução lida da memória.

d) Algoritmo

O LoadControl informará que tipo de load será feito.

Assim, os bits menos significativos serão zerados. Sua quantidade variará de 24 bits ou 16 bits, correspondendo à um load byte e à um load halfword respectivamente.

3) Lui

a) Entradas

Clk (1 bit): representa o clock do sistema.

Imediato (16 bits) : Retira o imediato da instrução [15:0] vinda do registrador de instruções.

b) Saídas

LUIout (32 bits): Envia o resultado da operação de LUI para o multiplexador 11, para ser escrito no banco de registradores.

c) Objetivo

A LUI recebe uma constante (vinda do registrador de instruções contendo bits de 0 a 15) e a transforma na parte mais significativa de um vetor de 32 bits. Os bits menos significativos são preenchidos com zeros.

d) Algoritmo

Recebe um vetor de 16 bits como input, cria um vetor de 16 bits formado por zeros e concatena à entrada, deixando-a como bits mais significativos. Devolve o vetor de 32 bits como saída.

4) Signal Extend

a) Entradas

Imediato (16 bits): Representa os 16 primeiros bits da instrução armazenada no registrador de instruções.

b) Saídas

Signal_Extend_Out (32 bits): Vetor concatenado de 16 zeros + imediato (Se o valor de imediato [15] for igual a zero.) senão extende o 1.

c) Objetivo

O objetivo do Signal Extend é tornar o imediato um vetor de 32 bits para que possa ser usado em operações na ULA.

A entidade realiza a concatenação com zeros nos bits mais significativos.

d) Algoritmo

Assim como a LUI, o Signal Extend recebe um vetor de 16 bits como input, cria um vetor de 16 bits formado por zeros e concatena à entrada, deixando-a como bits menos significativos. Devolve o vetor de 32 bits como saída.

5) Shift Left 2 (versão 1)

a) Entradas

Vetor (26 bits): Retira o imediato da instrução [25:0] vinda do registrador de instruções.

b) Saídas

Vetor (28 bits): Envia o resultado da operação para o multiplexador 6 sendo possível selecioná-lo no multiplexador que escreve no PC.

c) Objetivo

Aumentar o número de bits do imediato e concatená-lo com os 4 bits mais significativos de PC, calculando assim um possível desvio incondicional em um mesmo clock, aumentando o desempenho do processador.

d) Algoritmo

Recebe um vetor de 26 bits como input, cria um vetor de 28 bits.

6) Shift Left 2(versão 2)

a) Entradas

Signal_Extend_out (32 bits) : Vetor de 32 bits que representa o valor do imediato (16 bits) da instrução concatenado com 16 zeros (que estarão nos bits mais significativos).

b) Saídas

Vetor (32 bits): Representa o vetor de entrada após ter seu valor multiplicado por 4 (shift left 2).

c) Objetivo

Durante a instrução J, é preciso calcular o endereço da próxima instrução. Para isso, o offset é retirado e estendido no Signal Extend e segue para o Shift Left 2 (versão 2) para ser multiplicado por 4 e finalmente carregado na ULA. Assim, obtemos o valor a ser setado no registrador de PC após a realização de uma instrução J.

d) Algoritmo

Recebe um vetor de 32 bits como input, realiza um shift left 2 no mesmo (move seus bits um a um para a esquerda em duas posições, o que representa uma multiplicação por 4) e o devolve como output.

7. **Div**

a) **Entradas**

Clk (1bit): representa o clock do sistema.
Reset (1 bit): Sinal que, quando ativado zera o conteúdo do registrador.
Valor do registrador A (32 bits)
Valor do registrador B (32 bits)
Controle_Div (1 bit): sinal que representa o controle.

b) Saídas

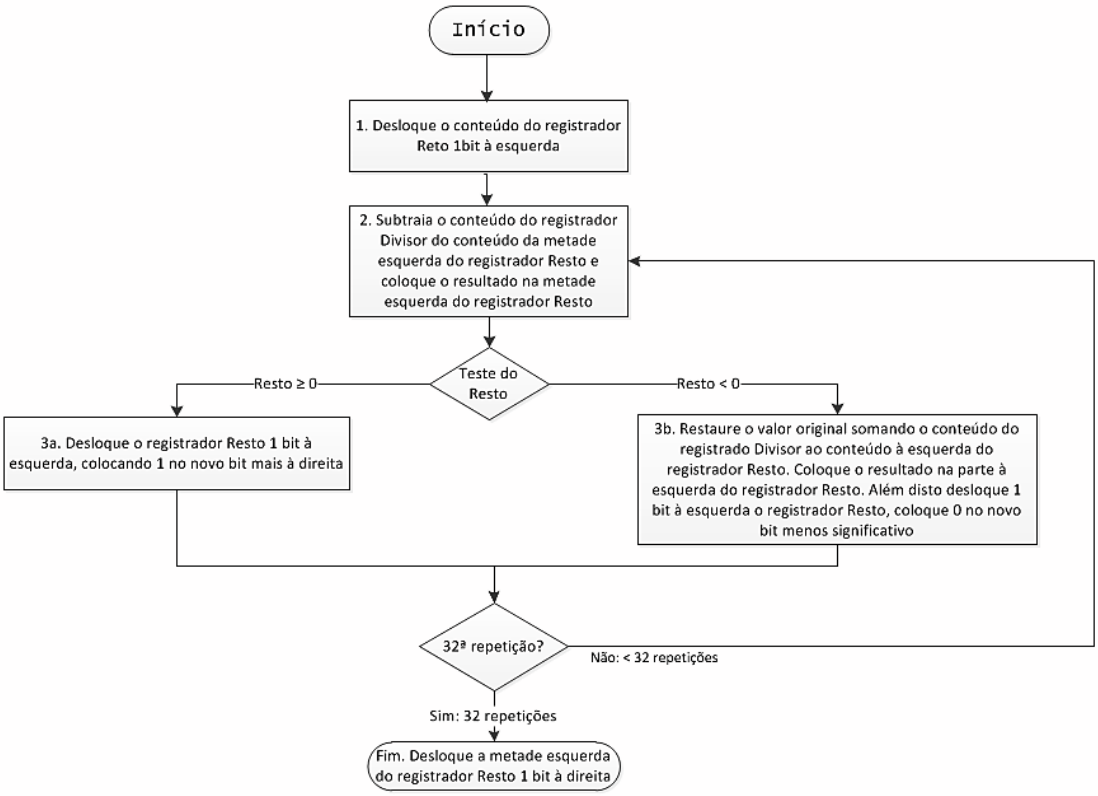
HI (32 bits): Saída para escrita no registrador hi representando o resto.
LO(32 bits): Saída para escrita no registrador lo representando o quociente.

c) Objetivo

Essa entidade tem a função de ler dados no Banco de Registradores a partir dos endereços de $rs e $rt, enviando tais informações ao módulo “Div” e o controle envia sinal para a dita-cuja divisão. O valor de quociente é escrito no registrador hi e o resto em lo. Finalizando a operação.

d) Algoritmo

Temos em seguida, o diagrama de estados da divisão:



Descrição das Operações

- Operações Tipo R

1. **ADD $rd, $rs, $rt**

Após a fase de decodificação da instrução, os valores dos registradores $rs e $rt é enviado ao banco de registradores e é carregado na ULA. O controle manda sinal positivo para a adição, e a ULA computa o valor de $rs + $rt. A soma, que será o output da ULA, é guardada no registrador de destino $rd.

2. **SUB $rd, $rs, $rt**

Após a fase de decodificação da instrução, os valores dos registradores $rs e $rt é enviado ao banco de registradores e é carregado na ULA. O controle manda sinal positivo para a subtração, e a ULA computa o valor de $rs - $rt. Esse resultado, que será o output da ULA, é guardada no registrador de destino $rd.

3. **AND $rd,$ rs,$ rt**

Após a decodificação da instrução, os valores de $rs e $rt são carregados na ULA. Esta, recebe do controle o sinal para a operação lógica AND ($rs & $rt). O resultado, booleano, sai no formato da flag Z, que é recebida pelo controle como um input. Analisando Z, o controle decide qual sinal passar ao multiplexador 11, que possui as constantes 1 e 0 à sua disposição. O multiplexador 11 deixa passar um dos sinais, que é armazenado no registrador de destino $rd.

4. **RTE**

Quando ocorre alguma exceção durante a execução de uma instrução, o valor de PC é armazenado na entidade EPC, registrador específico para este fim. A instrução RTE carrega o valor de EPC novamente em PC. Após a etapa de decodificação, o conteúdo de EPC é liberado através do multiplexador 6 e atualiza PC.

5. **NOP**

O NOP apenas faz com que o controle retorne ao seu estado inicial e busque a próxima instrução. O próprio mnemônico nop quer dizer “no operation”. Quando o controle identifica o NOP, retira uma nova instrução da memória.

6. **BREAK**

A instrução BREAK serve para parar a execução do programa. Ou seja, é um estado vazio do controle, que fica num eterno loop sobre si mesmo. Após a identificação da instrução pelo controle, este não envia mais sinais para os componentes do hardware e mantém-se sempre neste estado.

7. **SLT $rd,$ rs,$ rt**

Após a fase de decodificação, os valores de $rs e $rt são carregados em A e B e seguem para ser carregados na ULA. Lá, o controle seleciona a operação de comparação e analisa a flag LT (que retornará 1 caso $rs < $rt). Em caso positivo, o multiplexador 11 deixará passar a constante 1, que será salva no registrador $rd (selecionado pelo multiplexador 3). Em caso negativo, será salvo o valor 0.

8. **JR $rs**

O controle decodifica a instrução. Na presença de um JR, carrega o valor de $rs na ULA, que é liberado através do multiplexador 6 e sobrescreve o valor de PC. Esta instrução executa um jump definido previamente por $rs, sendo útil para marcar o retorno de subrotinas.

9. **MFHI $rd**

Abreviação de “Move from high”, permite recuperar o resto de uma divisão e armazená-lo no registrador $rd. Após a decodificação, o controle envia através do

multiplexador 11 o conteúdo relativo ao resto da divisão (que está armazenada no registrador específico encontrado dentro da entidade DIV) e atualiza $rd com esse valor.

10. **MFLO $rd**

Similar ao MFHI, porém corresponde ao quociente da divisão, também sendo armazenado no em $rd. Sua sigla significa “Move from low”.

11. **SLL $rd,$rt, shamt**

A instrução define que no registrador $rd seja armazenado o valor de $rt após receber um shift left 2 elevado ao valor de shamt. Para isso, os multiplexadores 7 e 8 recebem shamt como entrada N (5 bits) e o valor de $rt como vetor de entrada. A entidade SHIFTER realiza a operação (regida pelo controle, que identifica qual shift está sendo operado) e retorna o vetor de saída para o multiplexador 11, que deixa passar o sinal, salvando o valor no registrador $rd.

12. **S RL $rd,$rt, shamt**

Similar ao comportamento de SLL, porém realiza um shift right 2 elevado ao valor de shamt no vetor de entrada (importante lembrar que nesse caso será utilizado como input o registrador $rs).

13. **SLLV $rd,$rs, $rt**

Possui o mesmo comportamento do SLL, mas ao invés de utilizar shamt como referência ao número de bits deslocados, utiliza o valor encontrado no registrador $rt para realizar shift left no vetor $rs.

14. **SRAV $rd,$rs, $rt**

Apesar de se parecer com a roteiro tomado pela instrução SLLV, SRAV dá um shift right no vetor $rs do valor estendido do registrador $rt. O controle deve selecionar este tipo de shift na entidade SHIFTER.

15. **SRA $rd, $rt, shamt**

Muito parecido com o SRL, o SRA diferencia-se por dar um shift right no valor de $rt usando o valor do shamt com extensão de sinal.

16. **DIV rs, $rt**

Esta operação busca o valor de $rs e $rt para dividi-los como o valor lido no endereço $rs do banco de registradores como sendo o dividendo e o valor do endereço $rt como sendo o divisor. Estes valores são encaminhados ao módulo "Div" e calculados seguindo o algoritmo mostrado na seção das Entidades.

- Operações Tipo I

1. **ADDI $rt, $ rs**

A instrução é decodificada e o conteúdo de $rs é direcionado ao multiplexador 3. Seu sinal passa ao banco de registradores, pois o controle envia ao multiplexador sinal positivo para sua passagem. Assim, $rs é carregado na entrada da ULA. O imediato recebe um Signal Extend, tornando-se um vetor de 32 bits e é carregado na entrada da ULA através do multiplexador 5. A ULA recebe sinal do controle para realizar uma adição ($rs + imediato) e o resultado é armazenado no registrador de destino $rt.

2. **ADDIU $rt, $ rs**

Esta instrução é similar ao addi, utilizando dos mesmos registradores, apenas ignorando o sinal de overflow recebido pelo controle, para apenas armazenar o valor da soma em módulo.

3. **LUI $rt , imediato**

Após a instrução ser carregada no registrador de instruções, seu imediato [15:0] é enviado à entidade LUI, onde recebe uma concatenação de 16 bits, tornando-se um vetor de 32 bits onde o imediato representa os bits mais significativos.

Esta saída é direcionada ao multiplexador 11, que recebe sinal do controle para permitir a saída dos dados (após a fase de decodificação). O multiplexador 3 recebe sinal do controle para carregar o valor do registrador $rt [20:16]. Finalmente, o banco de registradores recebe sinal positivo para escrever dados, e guarda o vetor de 32 bits em $rt.

4. **LB $rt , byte[imediato + $rs]**

A instrução é carregada no banco de instruções, o imediato[15:0] é somado ao valor do registrador na posição rs [25: 21] (esta soma obtida da ULA), obtendo o valor a ser guardado no registrador. Porém neste caso, apenas um byte “mais significativo” é realmente carregado no registrador, sendo portanto enviado à unidade Load pra respectiva função de enviar o byte para o banco de registradores para escrita.

5. **LH $rt , halfword[imediato + $rs]**

Essa instrução é bastante similar à instrução de load byte, bastando ser apena o sinal da unidade de Load referir ao carregar de uma halfword.

6. **LW $rt ,imediato + $rs**

Instrução similar às instruções lh e lb, carrega uma word (32 bits) no registrador $rt.

Diferentemente das outras instruções (lh e lb), esta é executada no próprio circuito, sem um módulo. Esta instrução tem o objetivo de carregar uma word da memória para o banco de registradores, podendo o seu valor, usado em alguma instrução mais tarde, em outra operação. Assim que a instrução está no IR(Instruction Register) o valor do registrador $rs é somado na ULA com o imediato para calculo do endereço a ser lido na memória. O ULAOut é selecionado no mux1, e o sinal de controle envia um sinal de leitura para a memória. Assim que o dado da memória for salvo em MDR (Memory Data Register) esse dado é enviado para escrita no Banco de Registradores através do mux11 e o endereço em $rt é enviado como o endereço a ser escrito o valor de MDR através do mux3.Sinal é enviado ao Banco de Registradores para escrita desse dado.

7. BEQ $rt, $ rs, LABEL

É basicamente uma instrução que irá verificar se os valores dos registradores $rt e $rs são iguais em caso afirmativo, temos que calcular o valor desse desvio para escrita no PC. Assim, o valor do offset é extendido para 32 bits e posteriormente multiplicado por 4 e enviado ao multiplexador 5. O endereço guardado por PC é enviado ao multiplexador 4, esses valores sendo selecionados pelo controle e somados na ULA, reservando a soma no ULAOut.

Os valores dos registradores $rs e $rt são enviados à ULA para comparação, assim são lidos seus valores no Banco de Registradores. Um sinal do controle à ULA para comparar, acaba por enviar uma flag ao controle informando-o se realmente aconteceu de uma igualdade nesses valores. Em caso afirmativo, temos o desvio calculado e será enviado para escrita no registrador PC, selecionado pelos multiplexadores 6 e o muxPC.

8. BNE $rt, $ rs, LABEL

Essa instrução é bastante similar ao beq, diferindo apenas no sinal enviado ao

controle, em saída da ULA identificando se os valores de $rt e $rs são diferentes, podendo assim fazer o devido desvio, enviado ao PC.

9. BLT $rt, $ rs, LABEL

Essa instrução é bastante similar ao beq, diferindo apenas no sinal enviado ao controle, em saída da ULA identificando se os valores de $rt e $rs são diferentes, podendo assim fazer o devido desvio, enviado ao PC.

10. BGT $rt, $ rs, LABEL

Essa instrução é bastante similar ao beq, diferindo apenas no sinal enviado ao controle, em saída da ULA identificando se os valores de $rt é maior que o valor do endereço $rs, podendo assim fazer o devido desvio, enviado ao PC.

11. SB $rt, offset($rs)

Essa instrução é o inverso do loadByte(lb). Assim, deverá ler do Banco de Registradores e escrever o dado na memória, através do módulo ‘store’, definindo o byte que deverá ser escrito na memória.

O valor de $rs é colocado no registrador A e o valor do imediato é estendido, estes são somados e colocados em ULAOut. O endereço em ULAOut é lido na memória como “espécie” de load para não ocorrer perda de informação (este conceito é válido tanto para os store de byte ou de halfword). Assim, tendo essa informação, é enviado ao módulo “Store”, o dado da memória e o valor do registrador B.

No módulo, ocorre a concatenação com o byte menos significativo do registrador B (para manter a informação) com o restante dos bits do dado da memória.

Escreve esta concatenação na posição do offset + $rs da memória.

12. SH $rt, offset($rs)

Esta instrução é semelhante à store byte (sw) apenas diferindo quanto ao sinal do controle para definição do tipo de store e quanto ao número de bits para concatenação (evitando perda de informação), sendo neste caso, a concatenação com 16 bits do valor da memória e 16 bits do registrador B.

13. SLTI $rt, $rs, imediato

Essa instrução irá comparar o valor do registrador $rs com o valor do imediato da instrução e verificar se tal valor é menor que o outro, em caso afirmativo, iremos colocar a constante 1 no registrador $rt e caso contrário, colocaremos o valor 0 no referido registrador. O imediato é estendido e selecionado no mux5 e o valor do registrador $rs está no registrador B é selecionado no mux4.Assim podem ser comparados na ULA. A ULA envia um sinal ao controle que definirá qual valor será escrito no endereço $rt, caso seja positivo(o valor de $rs é menor que o valor do imediato), o mux11 selecionado o valor 1 e o mux3 seleciona o endereço de $rt, caso contrário, o valor 0 será selecionado pelo mux11. Sinal de escrita é enviado ao Banco de Registradores para poder realizar efetivamente a escrita.

14. SW $rt,offset($rs)

Essa instrução serve de inverso da instrução loadWord(lw). De forma que o valor que está no Banco de Registrador deverá ser enviado à memória para escrita.

O Banco de Registradores lê e coloca o valor de $rs no registrador A, selecionando-o no mux4 e o imediato da instrução é extendido e selecionado no mux5.

O controle envia à ULA um sinal de soma, colocando imediato + valor do registrador A em ULAOut.

O valor do registrador B é enviado ao WriteData da memória sendo o dado a ser guardado no endereço calculado no ULAOut. Agora, o controle envia um sinal de escrita à memória guardando a word do registrador $rt.

15. ADDM $rt, offset($rs)

Esta instrução calcula o offset somado ao valor do endereço em $rs, lido na memória tal endereço que será posteriormente somado ao dado do enderço $rt e tal soma será armazenada também na memória, utilizando portanto a memória para leitura e escrita assim como a ULA, duas vezes, porém para a mesma operação (add).

O Datapath: o valor do registrador $rs é salvo no registrador A, enviado ao mux4 e o imediato é estendido e enviado ao mux5.É enviado um sinal à ULA, para soma dos valores(valor da soma guardado em ULAOut para próximo clock).

ULAOut é salvo no regADDM para posteriormente, identificar o endereço a ser guardado o valor final da instrução. Este valor é lido na memória com o sinal do controle de maneira que o valor lido fique guardado em MDR (Memory Data Register). Assim temos o valor a ser somado ao dado do endereço $rt, na ULA,sendo os valores selecionados pelos mux4 e mux5.O ULAOut é enviado ao WriteData da memória no mux2 e o valor do registrador regADDM é enviado para Address da memória como o valor (offset + $rs), sinal de escrita é enviado à memória de guardar o dado no endereço calculado.

- Operações Tipo J

1. J Label

Essa instrução Jump corresponde ao desvio incondicional, assim o valor do imediato da intrução é multiplicado por 4(shift left 2) passando de 26 bits para 28 bits acrescentado dos 4 bits mais significativos de PC em uma concatenação. Este sendo enviado ao mux6 para ser escrito no PC, posteriomente sendo lido como a próxima instrução.

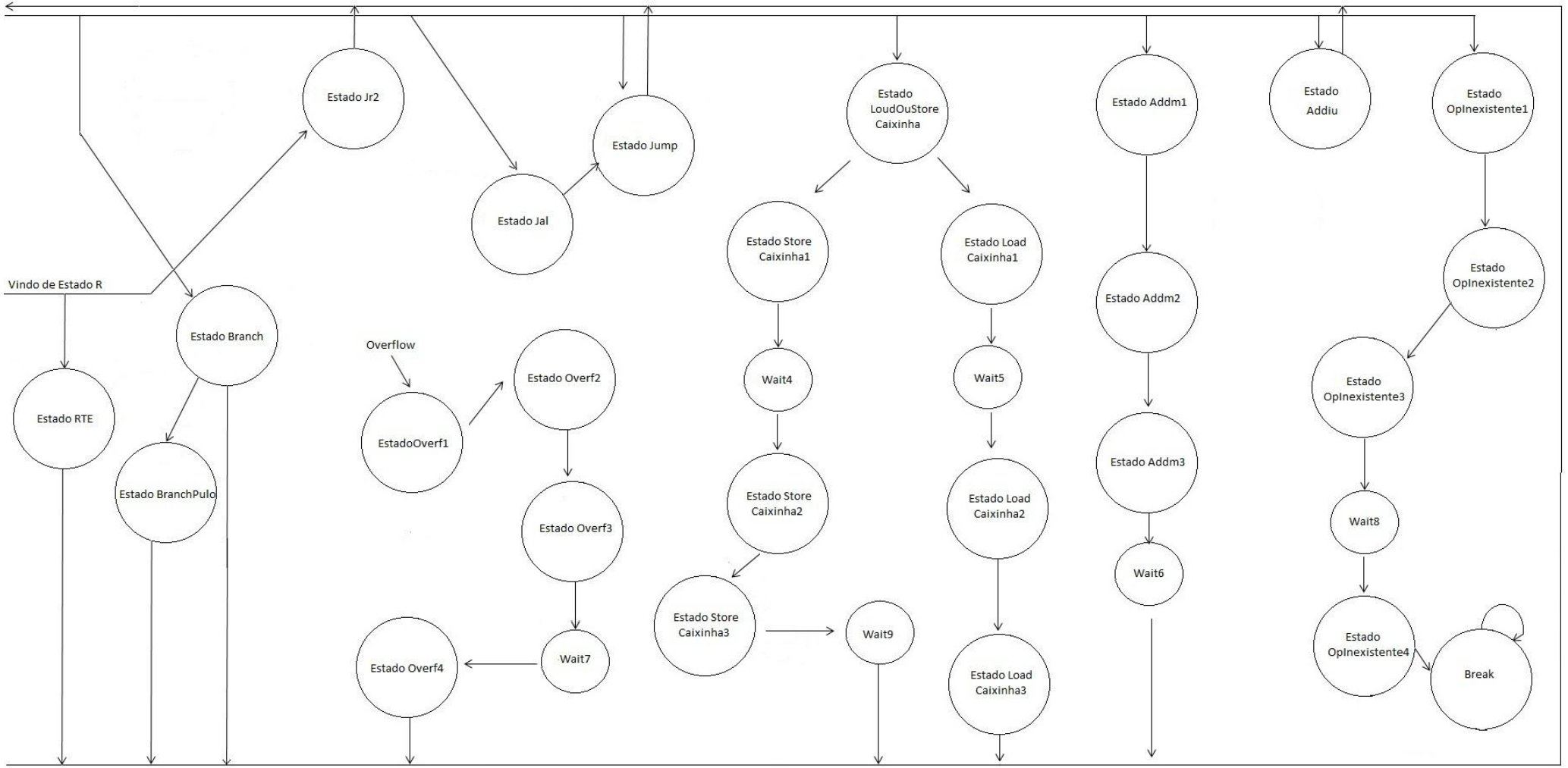
2. JAL Label

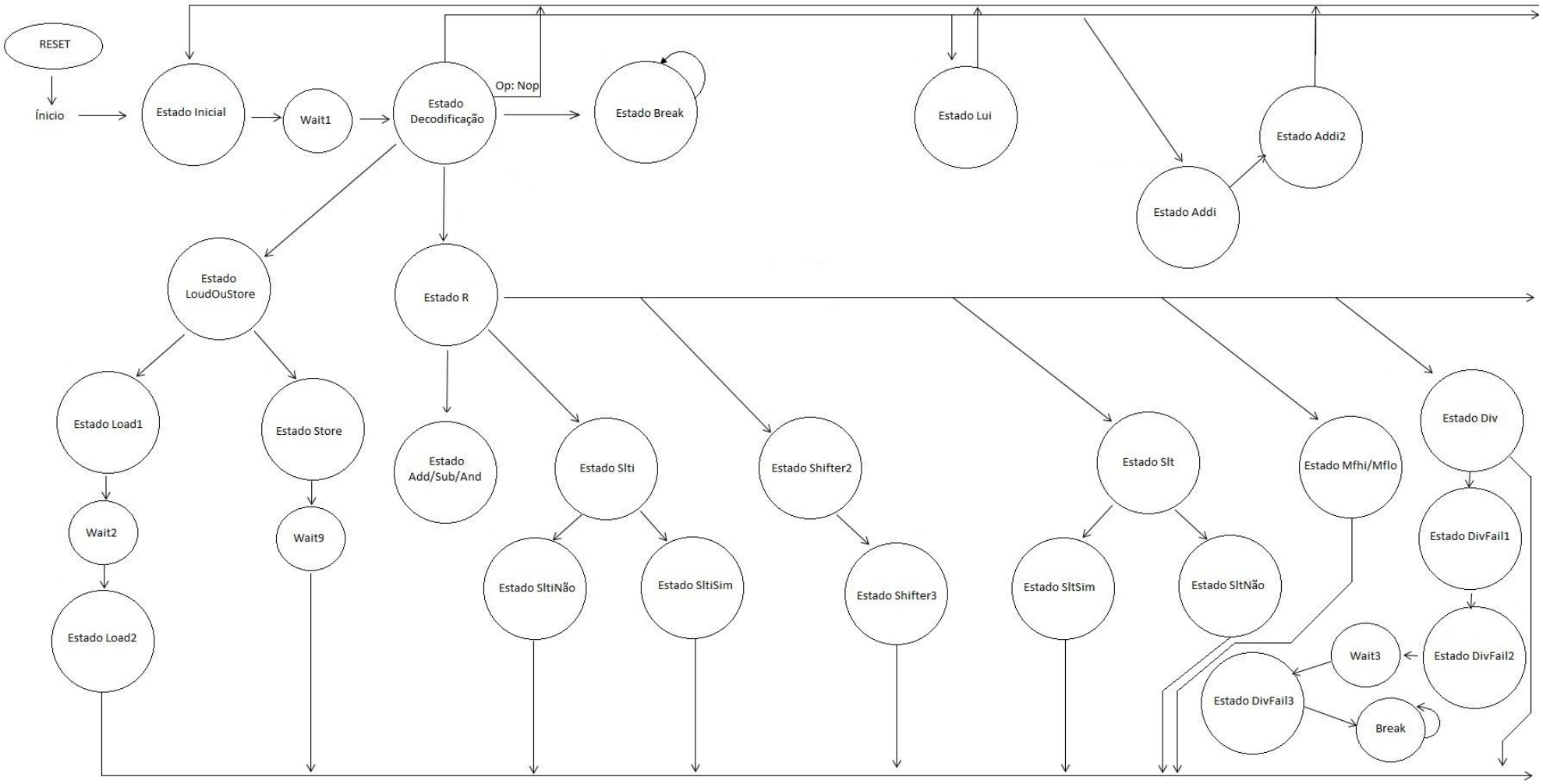
Essa instrução corresponde ao desvio incondicional, e com o cálculo fornecido pela especificação. Para armazenamento da instrução consequente, colocamos o valor de PC na ULA para ser carregado (sinal do controle). A saída da ULA vai para o multiplexador 11 para ser escrita no banco de registradores. O multiplexador 3 permite a passagem do endereço do registrador 31, específico para o armazenamento desta informação. É importante lembrar que o valor de PC já está incrementado de 4,ou seja, corresponde à instrução seguinte, pois isto é calculado antes mesmo da identificação do opcode da instrução pelo controle do processador.

Descrição dos estados de controle

Nas próximas duas páginas, encontraremos uma representação gráfica dos estados de controle.

Em seguida, a descrição detalhada de cada uma delas.


Vindo de Estado R
Estado RTE
Estado Branch
Estado BranchPulo
Estado Jr2
Estado Jal
Estado Jump
Overflow
EstadoOverf1
Estado Overf2
Estado Overf3
Estado Overf4
Wait7
Estado LoudOuStore Caixinha
Estado Store Caixinha1
Wait4
Estado Store Caixinha2
Estado Store Caixinha3
Wait9
Estado Load Caixinha1
Wait5
Estado Load Caixinha2
Estado Load Caixinha3
Estado Addm1
Estado Addm2
Estado Addm3
Wait6
Estado Addiu
Estado OpInexistente1
Estado OpInexistente2
Estado OpInexistente3
Wait8
Estado OpInexistente4
Break


RESET
Ínicio
Estado Inicial
Wait1
Estado Decodificação
Op: Nop
Estado Break
Estado Lui
Estado Addi
Estado Addi2
Estado LoudOuStore
Estado R
Estado Load1
Estado Store
Wait2
Wait9
Estado Load2
Estado Add/Sub/And
Estado Slti
Estado SltiNão
Estado SltiSim
Estado Shifter2
Estado Shifter3
Estado Slt
Estado SltSim
Estado SltNão
Estado Mfhi/Mflo
Estado Div
Estado DivFail1
Estado DivFail2
Wait3
Estado DivFail3
Break

Estado: **Reset**:
Este estado carrega o valor 227 no registrador 29.

Estado: **Estado_Inicial**
Neste estado, a memória é lida para busca da instrução e enviada para o registrador de instruções para decodificação no próximo estado (além do estado de ‘wait’ não descrito neste relatório, por ser apenas um estado de espera de leitura da memória). O valor de PC é somado com a constante 4 na ALU para cálculo da instrução seguinte.

Estado: **Estado_Decodificacao**:
Neste estado, a instrução é decodificada, ou seja, o valor do opcode é enviado ao controle para identificar qual é a instrução, seu tipo e ações a serem tomadas, como os próprios sinais colocados nas entidades para execução. Simultaneamente, o campo rs e rt da instrução é lido no banco de registradores e escrito nos registradores auxiliares A e B (facilitando a execução de instruções do formato R, aritmética ou lógica), assim como o valor do imediato é estendido e multiplicado por 4(shift left 2) correspondendo ao cálculo do ‘branch’, guardado no ALUOut. Sendo portanto, um clock que facilita alguns cálculos, não causando nenhum prejuízo à CPU, caso não seja um tipo dos que foram calculados.

Estado: **Estado_Break**:
Este estado não realiza nenhuma operação, e mantém o controle sempre em loop, retornando ao mesmo estado. Assim, a execução é paralisada.

Estado: **Estado_Rte**:
Responsável por carregar o valor de EPC em PC.

Estado: **Estado_Nop**:
Apenas retorna a execução do controle para o Estado_Inicial (não realiza operação).

Estado: **Estado_Lui**:
Realiza um shift left 16 no imediato da instrução.

Estado: **Estado_Jal**:
Responsável por guardar o endereço da instrução de retorno no registrador 31. Direciona o controle para o Estado_Jump (onde será realizado o desvio incondicional).

Estado: **Estado_Jump**:
Controla o desvio incondicional. Seta o valor de PC (após concatenar seus bits [31:28] com [25:0] da instrução). Após execução, controle retorna ao Estado_Inicial.

Estado: **Estado_Wait**:
Neste projeto, foram criados vários estados de espera. Porém, podemos descrevê-los no mesmo tópico, pois estes apenas diferem em direcionar o controle para estados diferentes. O wait é usado sempre que precisamos acessar a memória, que precisa de um ciclo em espera para realizar as suas operações.

Estado: **Estado_Addi**:
Realiza a soma entre o valor do registrador A e do imediato da instrução (após passar pelo Signal Extend). O resultado é escrito em ULAout e o controle é enviado para o Estado_AddiDois.

Estado: **Estado_AddiDois**:
Guarda o resultado da operação Addi no banco de registradores. Controle retorna para Estado_Inicial.

Estado: **Estado_OpInexistenteUm / Estado_OverfUm / Estado_DivFailUm**:
Decrementa PC (afinal, deve ser guardado o endereço da instrução que causou exceção) e salva em EPC. Na memória, é carregado o endereço 253 ou 254 ou 255 (dependerá da exceção a ser tratada).

Estado: **Estado_OpInexistenteDois**:
A entidade load recebe sinal do controle para realizar um load byte no valor enviado pela memória. (Este estado é usado para as outras exceções).

Estado: **Estado_OpInexistenteTres**:
PC recebe o valor enviado pela entidade load. (Este estado é utilizado para as outras exceções).

Estado: **Estado_R**:
Analisa o funct das instruções de opcode referente ao formato R. Realiza as operações de ULA para as instruções que a requisitam e redireciona o controle ao próximo estado correspondente, que varia de acordo com o funct.

Estado: **Estado_AddSubAnd**:
Escreve em $rd o resultado das operações add, sub e and.

Estado: **Estado_SltSim / Estado_SltNao**:
Escreve em $rd o resultado da comparação feita pela ULA para a instrução SLT. A flag de saída da ULA faz com que o controle analise para qual estado deve seguir: Estado_SltSim (para flag = 1, escreve 1 em $rd) ou Estado_SltNao (para flag = 0, escreve 0 em $rd).

Estado: **Estado_Div**:
Se a divisão foi realizada com sucesso, direciona o controle ao Estado_Inicial. Se não, o controle é preparado para estado de tratamento de exceção (no caso, divisão por zero).

Estado: **Estado_Addiu**:
Realiza a soma entre o valor encontrado em A com o imediato. Porém, despreza o sinal de overflow, cuidando apenas do resultado obtido.

Estado: **Estado_Slti**:
Semelhante ao slt, porém realiza a comparação utilizando o imediato (após signal extend). A flag de saída direciona o controle para o estado correspondente.

Estado: **Estado_SltiSim / Estado_SltiNao**:
Escreve 1 caso a flag retorne positiva à comparação (Estado_SltiSim) e 0 caso seja negativa (Estado_SltiNao). O controle retorna ao Estado_Inicial.

Estado: **Estado_AddmUm**:
Salva o endereço [offset+rs] no registrador especial para addm, já que ULAout será sobrescrito nas operações seguintes. O controle segue para Estado_AddmDois.

Estado: **Estado_AddmDois**:
[offset+rs] é somado a $rt. O valor é armazenado em ULAout. O controle segue para Estado_AddmTres.

Estado: **Estado_AddmTres**:
A memória recebe um load no endereço [offset+rs] (armazenado no registrador especial para addm) do valor [offset+rs] + rt. O controle segue para um estado de espera.

Estado: **Estado_JrDois**:
Atualiza PC com o valor do desvio condicional, calculado a partir do registrador fornecido.

Estado: **Estado_Branch**:
Realiza as comparações que irão definir se o branch será realizado ou não. Este estado direciona os parâmetros para a ULA e analisa as flags recebidas.

Estado: **Estado_BranchPulo**:
Caso as flags da ULA tenham resultado positivo para a comparação, o controle atualiza PC com o valor do branch.

Estado: **Estado_ShifterDois**:
Seleciona qual tipo de shift a unidade Shifter irá realizar com o vetor recebido.

Estado: **Estado_ShifterTres**:
Escreve o resultado do Shifter no registrador $rd. O controle retorna ao Estado_Inicial.

Estado: **Estado_LoadOuStore**:
Calcula [offset+rs] na ULA e salva em ULAout.

Estado: **Estado_Store**:
Em caso de store, guarda na memória o valor contido em B no endereço [offset+rs]. O controle entra em estado de espera.

Estado: **Estado_LoadUm**
Retira da memória o dado solicitado pelo endereço fornecido. O controle segue para estado de espera, e depois para o Estado_LoadDois.

Estado: **Estado_LoadDois**:
O registrador é escrito com o valor que foi carregado pela memória.

Estado: **Estado_LoadOuStoreCaixinha**:

Calcula [offset+rs] e guarda valor na ULA. Identifica se temos um load ou um store, direcionando o controle para o estado correto.

Estado: **Estado_StoreCaixinhaUm**:
A memória é solicitada e carrega o valor de [offset+rs]. O controle fica em estado de espera.

Estado: **EstadoStoreCaixinhaDois**:
Seleciona que tipo de store será realizado pela entidade Store.

Estado: **EstadoStoreCaixinhaTres**:
A memória escreve o dado fornecido pelo Store na posição [offset+rs]. É importante lembrar que os dados da memória não serão sobrescritos.

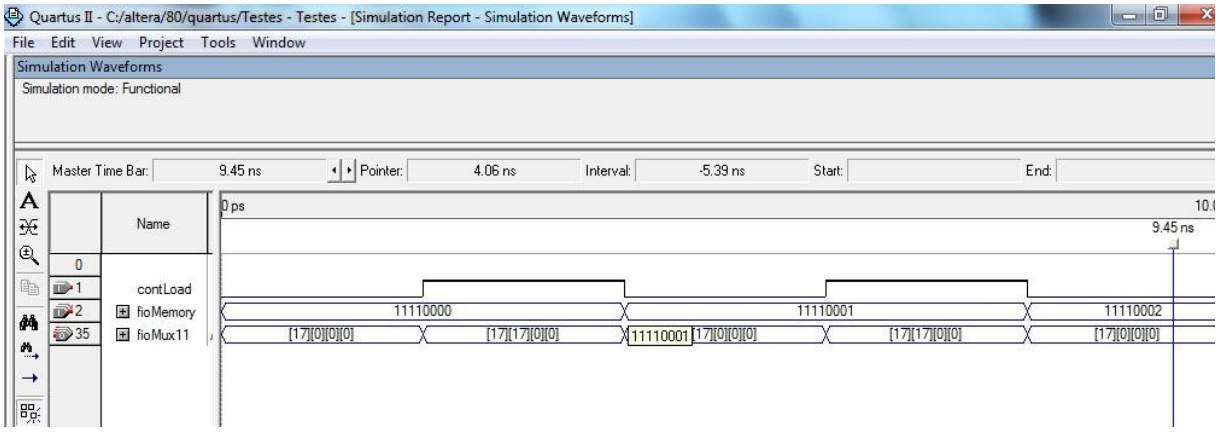
Estado: **Estado_LoadCaixinhaUm**:
A memória é requisitada a carregar o valor contido no endereço [offset+rs]. O controle permanece em estado de espera até o próximo ciclo, quando irá para o Estado_LoadCaixinhaDois.

Estado: **Estado_LoadCaixinhaDois**:
Seleciona qual o tipo de load utilizado na entidade Load.

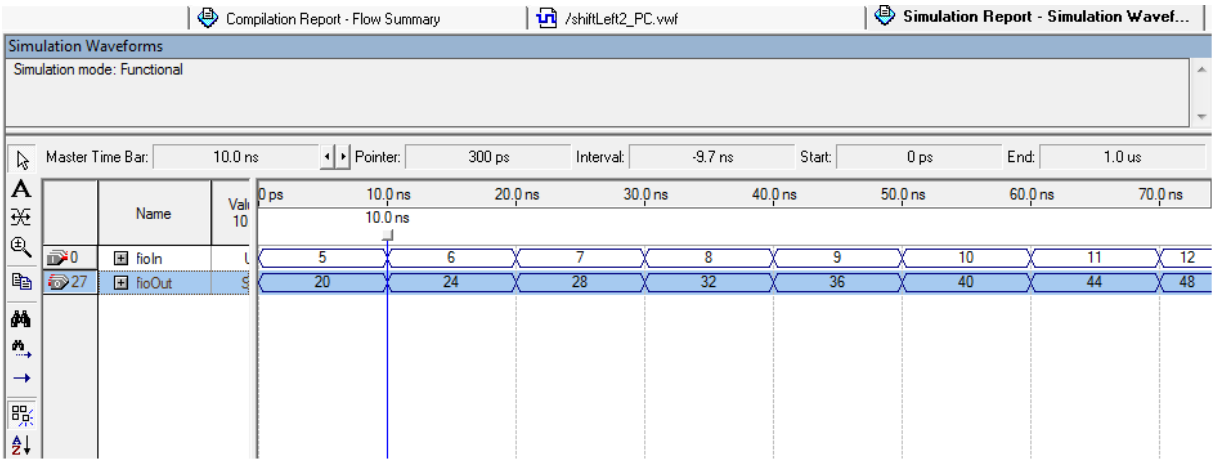
Estado: **Estado_LoadCaixinhaTres**:
Escreve no registrador o valor relativo ao output da entidade Load.

Conjunto de simulações

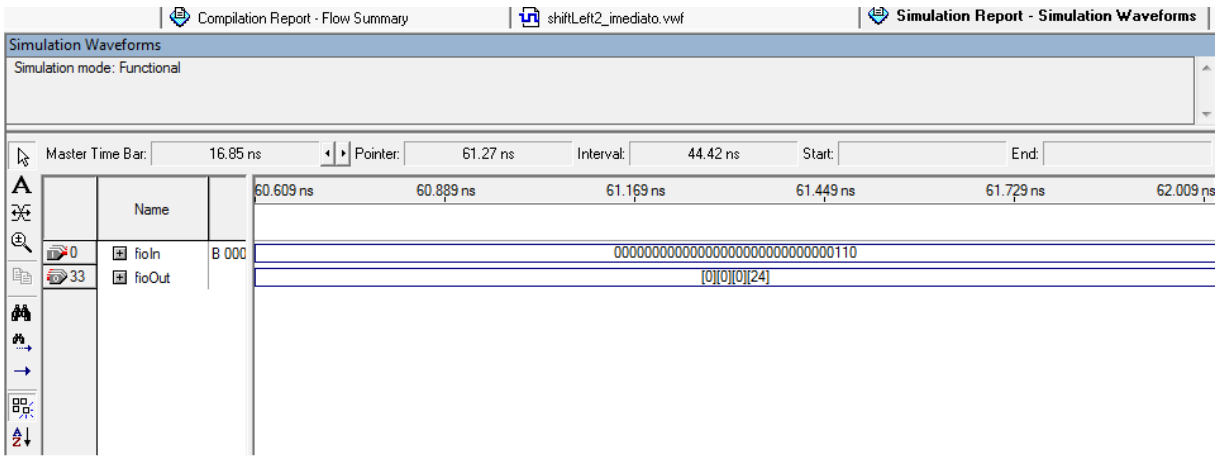
1) Entidade Load



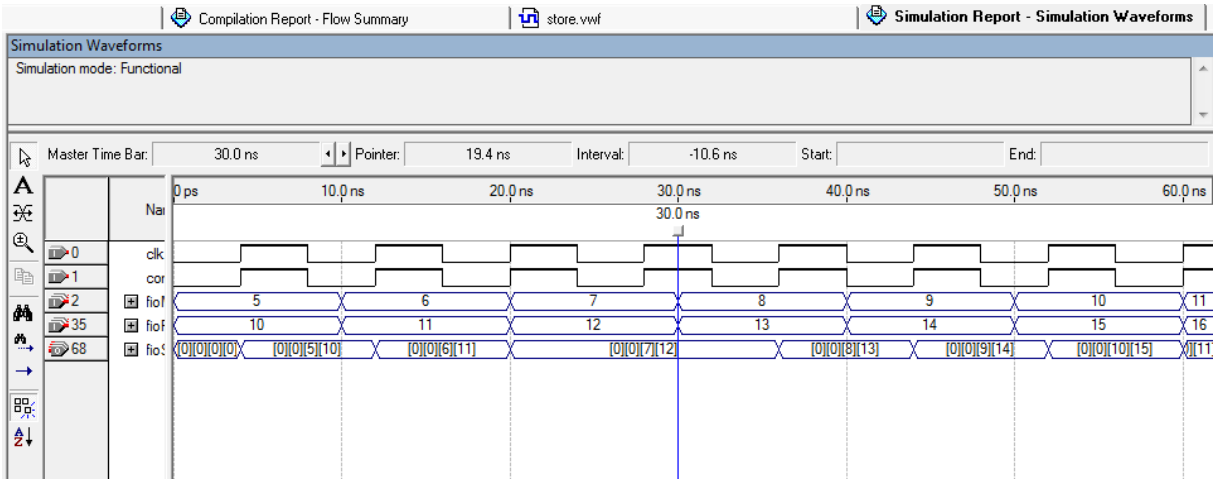
2) Entidade Shift-Left2_PC



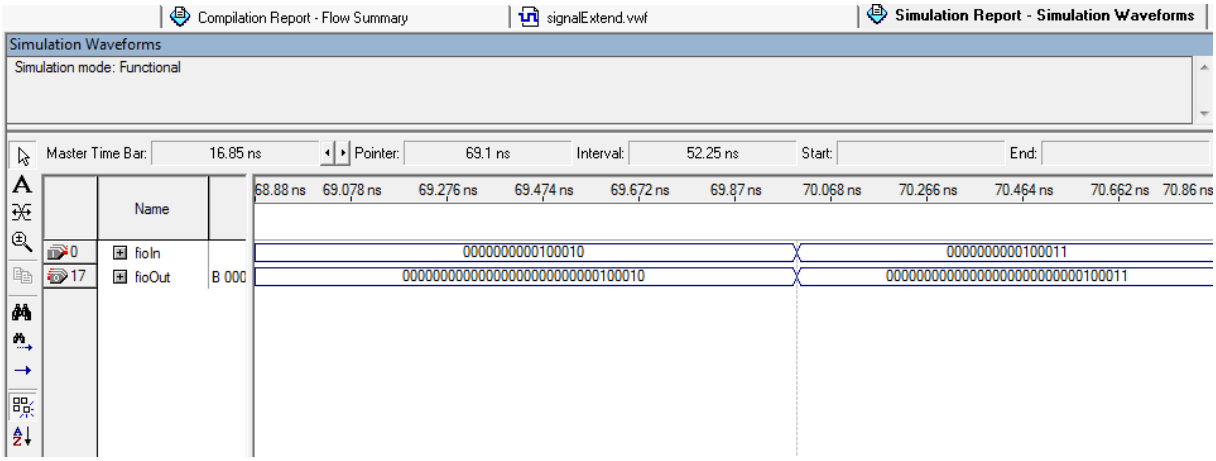
3) Entidade Shift-Left2_Imediato



4) Entidade Store



5) Entidade SignalExtend



Conclusões

Este projeto teve o intuito de demonstrar a arquitetura de máquina sob a ótica de um nível mais baixo, diminuindo o nível de abstração.

A construção do processador deixa clara a importância do conhecimento da área do hardware por parte de um cientista da computação. Conhecendo melhor sua organização, o profissional encontra-se mais preparado para usufruir e preparar programas e sistemas, de forma a extrair o máximo do desempenho de sua máquina.