UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE ELETRÔNICA
LABORATÓRIO DE ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES - 2022/1

Professor: Ricardo de Oliveira Duarte
Alunos: Arthur Coelho Ruback - 2019021140 e Gabriel Pimentel Gomes - 2018020271

Documentação do processador desenvolvido e dos trabalhos realizados

BELO HORIZONTE, JULHO DE 2022

Índice

1. Introdução

Primeiramente, é importante descrever como esta documentação se apresenta. Este documento está numa pasta com vários outros, sendo eles:

- Este texto (Documentacao_LAOC_GabrielArthur);
- Mapa de memória e banco de registradores (mapa_de_memoria);
- Discriminação do conjunto de instruções (Discriminacao_instrucao_banco);
- Discriminação das instruções SYSCALL (Discriminacao_SYSCALL)
- Tabela com os sinais de controle para cada instrução (Tabela_Controle);
- Tabela com o programa fatorial utilizado (Tabela_criacao_programas);
- Imagens do caminho de dados Ciclo Único, Pipeline e Interrupção+GPIO;
- Imagens da arquitetura da controladora de interrupções e do GPIO.

Uma vez descrito o conteúdo da pasta, agora será feita uma breve apresentação do trabalho. Esta disciplina tem como objetivo guiar os alunos no desenvolvimento de um processador de uso geral, tendo como inspiração os processadores MIPS ou RISCV. Além de fazer o núcleo do processador tanto em ciclo único quanto em pipeline, foram feitos também periféricos para interagir com esse núcleo.
Esta documentação apresentará, nesta ordem, o processador ciclo único, o processador pipeline, a controladora de interrupções e o GPIO.

2. Conjunto de instruções, mapa do banco de registradores, mapa de memória

Primeiramente, é importante dizer que as instruções, as palavras na memória e o tamanho dos registradores é de **16** bits. O número de registradores no banco também é 16. Essas e outras decisões de projeto estão no arquivo “mapa_de_memoria_banco”.
O arquivo “Discriminacao_instrucao” já apresenta uma descrição detalhada das instruções implementadas para o processador. Neste texto será feita uma descrição mais breve.
Existem apenas dois formatos de instrução no nosso conjunto, operações de imediato e operações de registrador, vistas nas figuras 1 e 2.

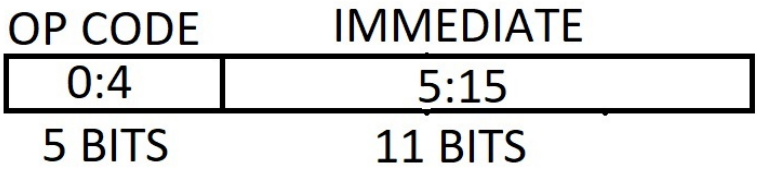

Figura 1: Formato de instrução imediato

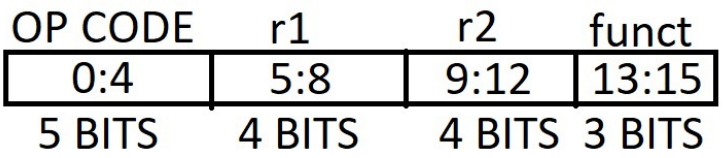

Figura 2: Formato de instrução de registrador

No conjunto de instruções possuímos operações de acesso à memória, de

operações lógicas e aritméticas e também de desvio do fluxo de execução, todas seguindo esses dois formatos. A lista de instruções é a seguinte: LI, NOP, JUMPI, JUMPL, JUMPR, BEQ, LOP, ADD, SUB, AND, OR, NOT, MUL, DIV, SLT, EQ.

Os 16 registradores do banco podem ser numerados de 0 a 15, mas eles também receberam nomes de acordo com a sua função. Essa nomeação aparece no arquivo “discriminacao_instrucao_banco”. Nela temos registradores de argumentos de função, registradores “saved”, registradores com endereço de retorno, etc.

O mapa de memória montado aparece no arquivo “mapa_de_memoria”, mas ele não foi muito relevante para o desenvolvimento do processador por vários motivos. Primeiramente, as simulações foram feitas em caráter de teste, com controle total sobre os dados, de maneira que o aluno poderia escrever em qualquer posição de memória que não afetaria o funcionamento do programa. Em segundo lugar, os programas feitos para testes utilizaram instruções de imediato, não precisando acessar a memória de dados em nenhuma ocasião. Por fim, os periféricos foram desenvolvidos no formato “entrada e saída dedicados”, fazendo com que novamente não fosse preciso acessar a memória de dados.

3. Processador Ciclo Único

Na figura 3 uma imagem do processador ciclo único projetado.

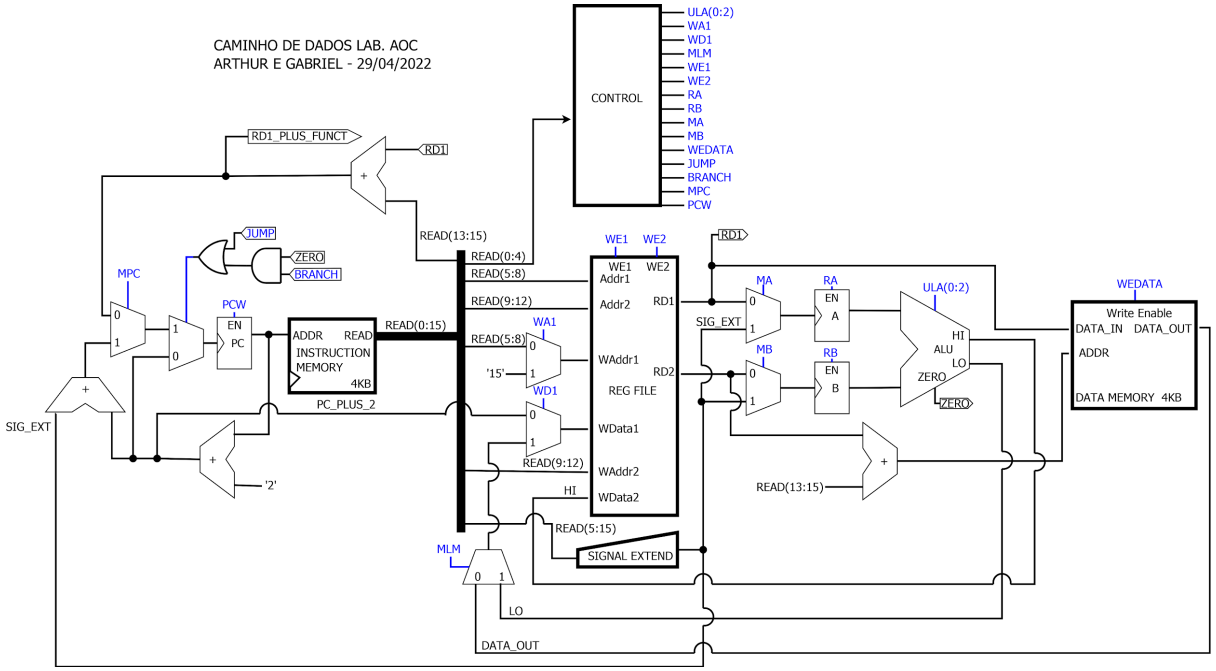

Figura 3: Processador Ciclo Único desenvolvido. Veja uma imagem de melhor qualidade na pasta.

a. Memória de Instrução e Barramento de Instrução

A memória de instrução é simplesmente uma memória assíncrona que recebe o endereço de PC. O barramento de instrução é marcado em negrito e ele distribui as partes da instrução para todo o caminho de dados. Veja na figura 4.

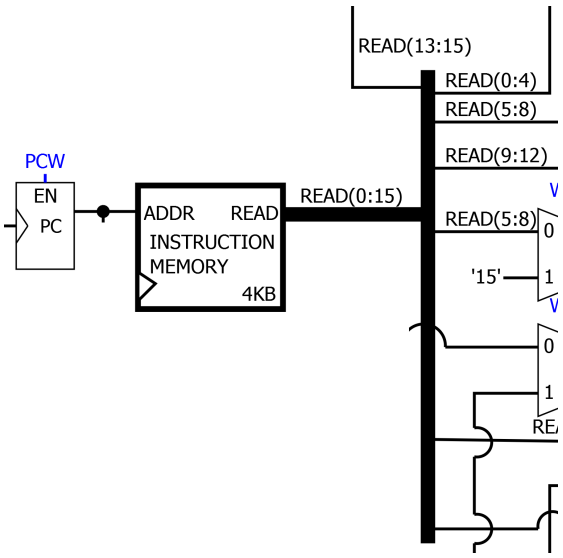

Figura 4: Recorte da memória de instrução, do PC e do barramento de instrução.

Observação: o PC endereça a memória por byte (cada instrução de 16 bits está distante duas posições de memória, PC incrementa de 2 em 2) mas a memória em VHDL de fato foi implementada com endereçamento para 16 bits. Ou seja, enquanto PC incrementa 2 tentando chegar na próxima instrução, a memória alcança a nova instrução com um incremento de 1 no endereço. Por isso, o endereço que chega na memória é PC/2. Assim, o PC aumenta de 2 em 2 enquanto o endereço na entrada da memória incrementa de 1 em 1. Veja o recorte do código onde isso acontece (processador_ciclo_unico.vhd).

```
Endereco => aux_endereco(4 to 14), --memoria de instrucao tem bloco de 16 bits e pula de 1 em 1,
                                   -- entao ignora o ultimo bit de PC, que pula de 2 em 2
```

b. Banco de Registradores

Veja o banco de registradores na figura 5.

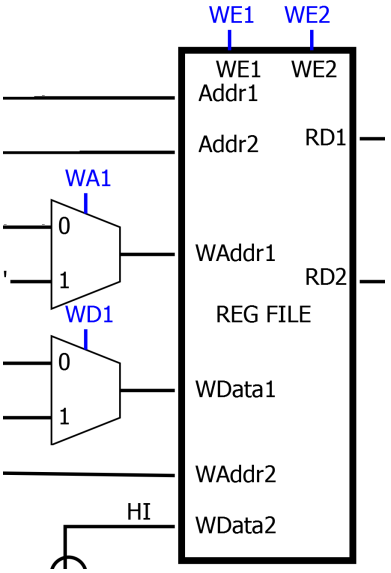

Figura 5: Recorte do banco de registradores

O banco de registradores utilizado foi uma modificação do componente fornecido pelo professor. Ele possui duas portas de leitura (RD1 e RD2) e duas portas de escrita (WD1 e WD2). A segunda porta de escrita, que foi adicionada ao componente inicial, serve apenas para guardar os bits mais significativos (HI) das operações de multiplicação e divisão. Foi decidido que era preferível guardar os dados no banco em vez de preparar um registrador específico para esse fim.

Os multiplexadores na entrada do banco servem apenas para a ocasião em que uma instrução JUMPL é chamada, na qual o endereço de retorno deve ser guardado no registrador de número 15.

c. Registradores de Operando

Um problema que surgiu quando foi tomada a decisão de usar palavras e registradores de 16 bits foi a falta de espaço nos argumentos das instruções. Enquanto arquiteturas de 32 bits podiam colocar 3 endereços de registradores em apenas uma instrução (dois de origem e um de destino), as instruções do nosso processador só tem espaço para dois endereços.

A solução encontrada foi dividir as instruções de operação em duas etapas, sendo a primeira de carregar os dados em registradores especiais A e B (onde aparecem os dois endereços de origem) e a segunda de operar com os valores guardados e devolver ao banco de registradores (onde aparece o endereço de destino). Veja os registradores logo na entrada da ULA na figura 6.

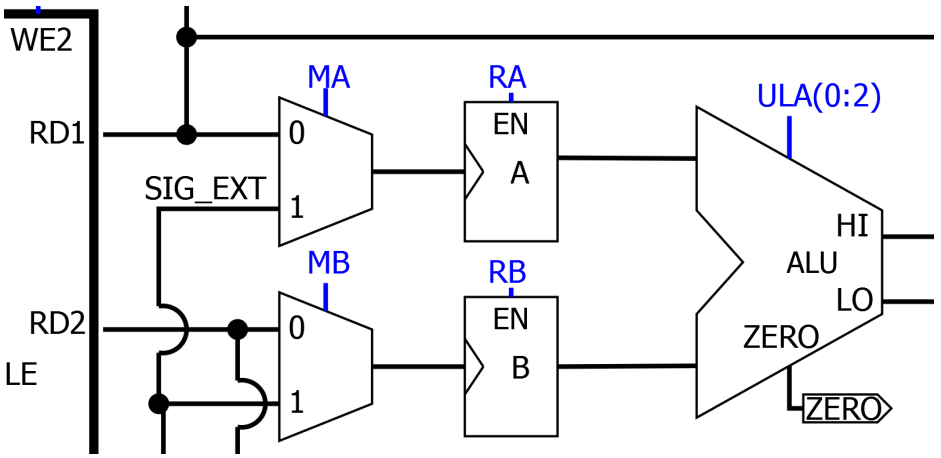

Figura 6: Recorte dos Registradores de operando A e B.

Outro ponto a se notar é que os registradores A e B podem receber dados tanto do banco de registradores quanto imediato estendido, de acordo com a configuração dos multiplexadores.

d. Unidade Lógico Aritmética

A ULA aparece na figura 7, com duas portas de entrada de 16 bits cada uma, duas saídas de dados HI e LO, e uma saída ZERO.

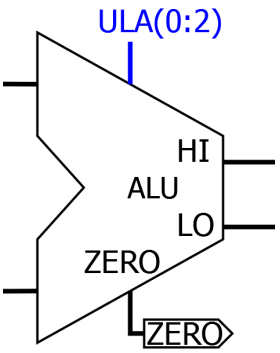

Figura 7: Recorte da Unidade Lógico Aritmética

A ULA faz operações de soma, subtração, operações lógicas e outras já descritas no conjunto de instruções. A operação a ser realizada é indicada pelos 3 bits de controle em azul. O sinal LO entrega a maior parte dos resultados da ULA, enquanto o sinal HI é utilizado apenas em operações de multiplicação e divisão. Ele entrega os 16 bits mais significativos da multiplicação e o resto da divisão. O sinal ZERO é utilizado na instrução de BRANCH para comparação.

e. Memória de Dados

Veja a memória de dados na figura 8.

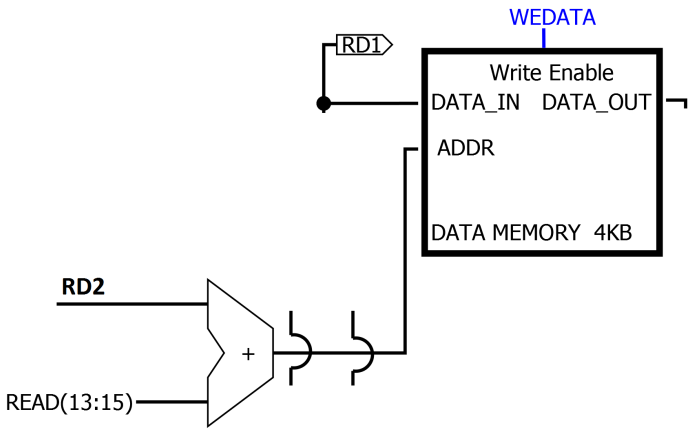

Figura 8: Recorte da memória de dados

A memória de dados é uma memória assíncrona que recebe como endereço o valor em no registrador RD2 mais um deslocamento dado pelo campo funct (bits 13:15 da instrução). Os dados vêm diretamente do registrador RD1.

f. Lógica de Nova Instrução

Existem 4 instruções de desvio do fluxo de execução do programa, sendo elas JUMPI, JUMPR, JUMPL e BEQ. Elas precisam de trabalhar com dados de caminhos diferentes para fornecer o novo valor de PC. A lógica para selecionar qual o novo PC é feita com multiplexadores, como se pode ver na figura 9. Eles foram instanciados diretamente no caminho de dados.

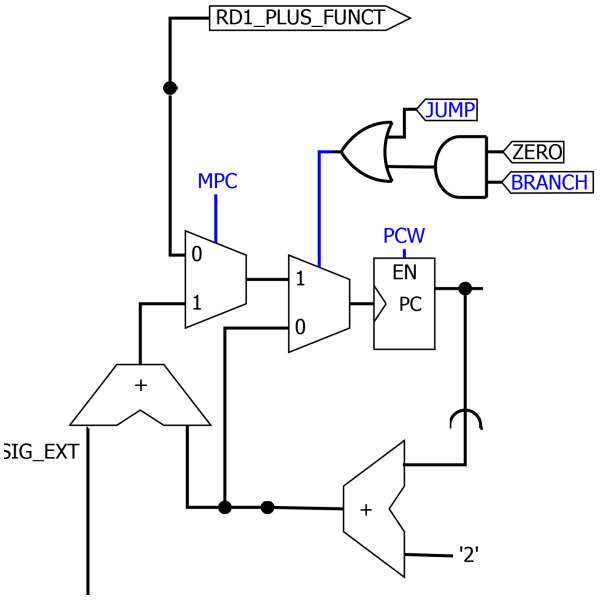

Figura 9: Recorte da Lógica de novo PC

Existem 3 opções de novo PC: $PC = PC + 2$ (próxima instrução), $PC = PC + SIG\,EXT$ (salto relativo ao PC, utilizado em JUMPI e BEQ) e $PC = RD1 + FUNCT$ (salto direto, utilizado em JUMPR e JUMPL).
O primeiro multiplexador, com "MPC", seleciona se o salto será direto ou relativo ao PC. O segundo multiplexador, com os sinais e portas lógicas no controle, decide se haverá salto ou não. Como se pode ver, sempre que há uma instrução JUMP ocorrerá salto. Se a instrução for BRANCH, o salto só ocorrerá se o sinal ZERO estiver em nível alto.

g. Unidade de Controle

A partir do OPCODE e do campo funct decide os sinais de controle. Cada instrução tem um conjunto de sinais de controle.
A unidade de controle é simplesmente um decodificador que recebe o campo OPCODE da instrução e gera os sinais a partir dele. Veja a unidade de controle na figura 10.

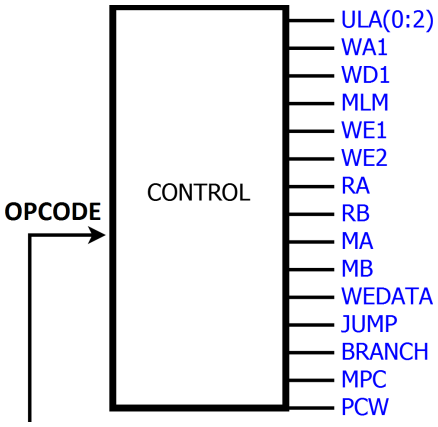

Figura 10: Recorte da unidade de controle

Como cada instrução possui um conjunto de sinais de controle, foi feita uma tabela com os sinais para cada uma das instruções. Essa tabela está no arquivo “Tabela_Controle”.

h. Programa Fatorial

Para testar o nosso processador, foi feito um programa que calcula o fatorial de um número dado. O programa com o código de máquina correspondente aparece no arquivo “Tabela_criacao_programas”. O programa em C e em Assembly é o seguinte:

Em C	Assembly
n = 8	li1 8
	lop2 r0
	add $s0
aux = 1	li1 1
	lop2 r0
	add $s1
one = 1	li1 1
	lop2 r0
	add $t0
#for_label	
if(n==0)	lop12 $s0 r0
goto(#end)	beq 5
aux = aux*n	lop12 $s0 $s1
	mul $s1
n - -	lop12 $s0 $t0
	sub $s0
goto(#for_label)	jumpi -7
#end	
	nop

O programa foi escrito diretamente no código da memória de instrução (memi.vhd), como se pode ver na figura 11. Ele calcula o fatorial de 8.

```
    variable tmp : memory_t := (others => (others => '0'));
begin
    tmp := (
            0 => X"4008",
            1 => X"5800",
            2 => X"0C80",
            3 => X"4001",
            4 => X"5800",
            5 => X"0D00",
            6 => X"4001",
            7 => X"5800",
            8 => X"0A00",
            9 => X"6480",
            10 => X"1805",
            11 => X"64D0",
            12 => X"7500",
            13 => X"64A0",
            14 => X"9C80",
            15 => X"2FF9",
            16 => X"7800",
            others => X"0000"
            );
```

Figura 11: Programa fatorial escrito diretamente na memória de instrução.

Agora veja o resultado final da simulação no ModelSIM na figura 12.

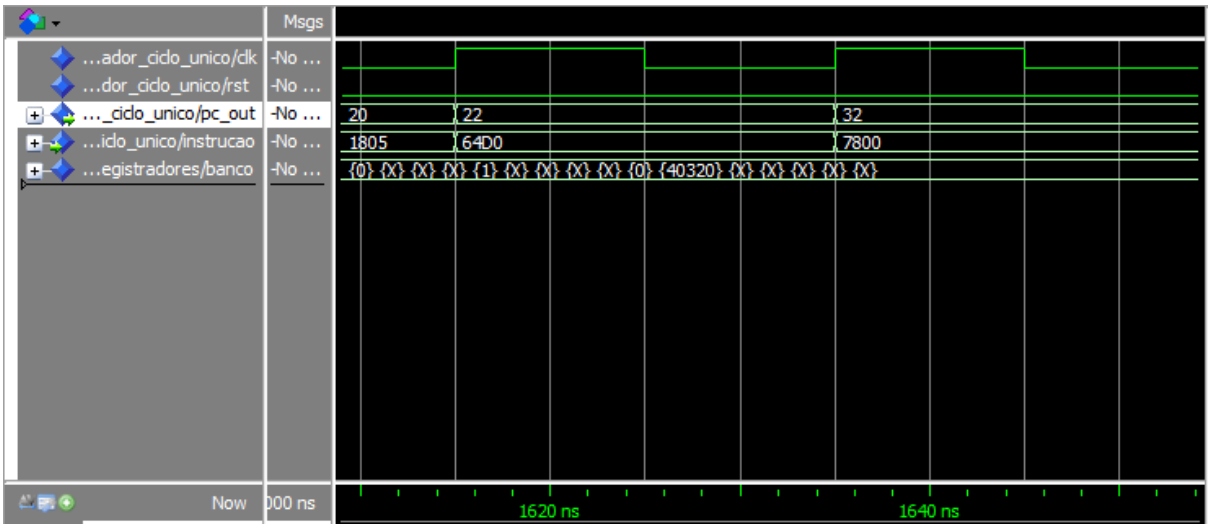

Figura 12: Resultado do programa fatorial no processador ciclo único

Note o número 8! = 40320 no banco de registradores, além do sinal pc_out, que salta de 22 para 32, finalizando o programa com uma instrução NOP (x7800).

4. Processador Pipeline

Após o projeto e validação do processador ciclo único, foi feito um processador pipeline.

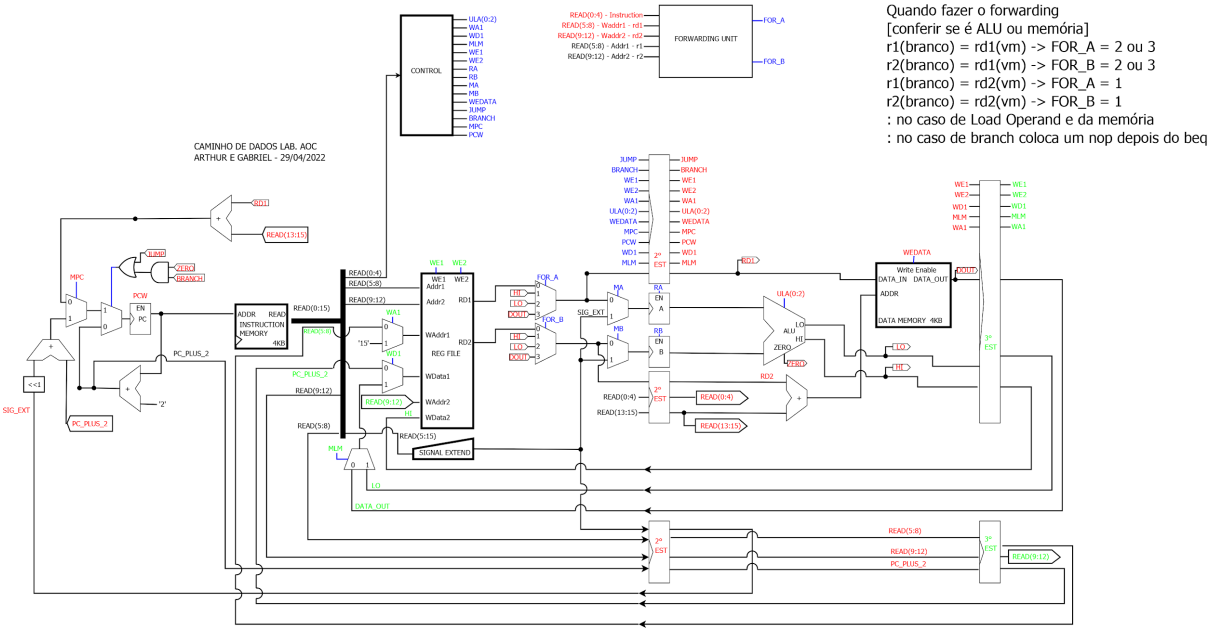

Figura 13: Processador pipeline. Uma imagem maior está na pasta de arquivos.

Ele possui três estágios, que foram marcados com cores diferentes para facilitar a identificação dos sinais de controle. O primeiro estágio não tem cor, sendo chamado de branco, o segundo estágio é vermelho e o terceiro estágio é verde. Foram adicionados vários registradores para que o pipeline pudesse funcionar. Veja na figura 13.

a. Primeiro Estágio (Fetch + Decode) - Branco

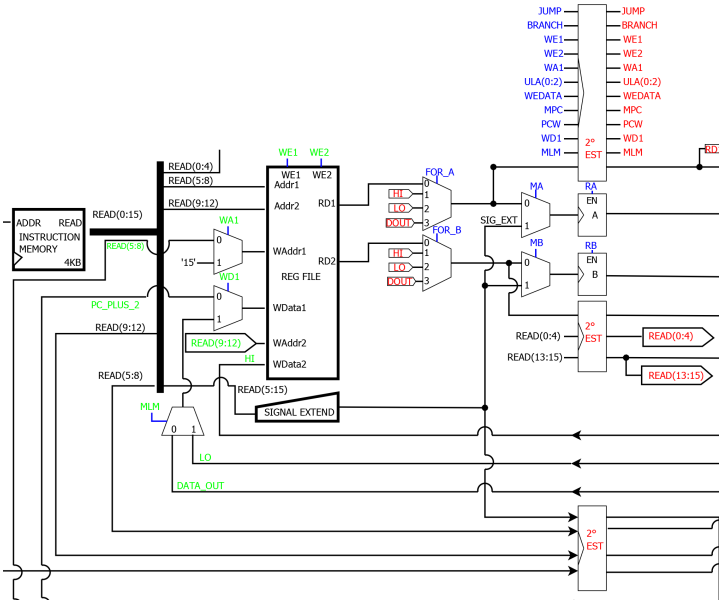

Figura 14: Primeiro estágio do pipeline - estágio branco

Quando se compara o pipeline desenvolvido com o pipeline MIPS, pode-se considerar que o primeiro estágio é equivalente aos estágios de Fetch e Decode juntos. Nele, a instrução é lida e decodificada, o banco de registradores é acessado e ao final do ciclo os registradores especiais A e B são carregados. Todos os dados gerados nesse estágio vão para um grande registrador vermelho. Os sinais de controle desse estágio vêm diretamente da unidade de controle. Veja as partes que compõem esse estágio na figura 14.

b. Segundo Estágio (Execute + Memory) - Vermelho

O segundo estágio desse pipeline é equivalente aos estágios Execute e Memory do pipeline MIPS juntos, pois nesse estágio pode ser feita uma operação de ULA ou de memória, mas não as duas ao mesmo tempo. Os dados utilizados e os sinais de controle são todos vermelhos, vindo do grande registrador logo atrás. Veja na figura 15.

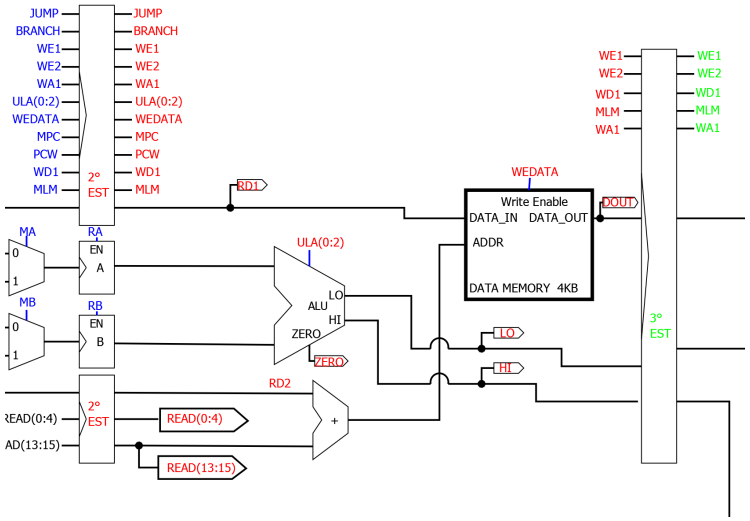

Figura 15: Segundo estágio do pipeline - estágio vermelho (Parte 1)

É importante notar que o BRANCH e o JUMP ocorrem nesse estágio, então os controles dos muxes de desvio de fluxo são vermelhos também. Veja na figura 16.

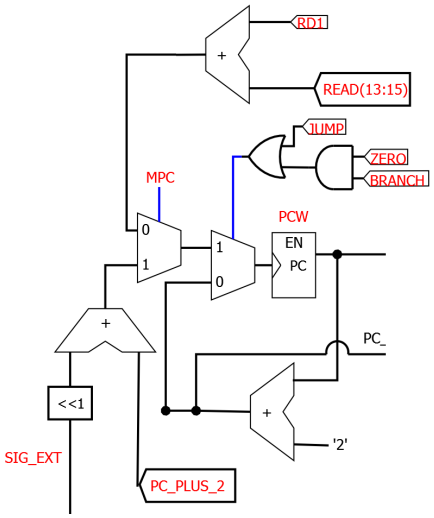

Figura 16: Segundo estágio do pipeline - estágio vermelho (Parte 2)

c. Terceiro Estágio (Writeback) - Verde

O terceiro estágio, da cor verde, corresponde ao estágio de writeback, que guarda os valores processados de volta no banco de registradores. Assim, os muxes que controlam a entrada do banco, e os sinais de enable do banco são todos da cor verde. Os dados utilizados vem de um grande registrador verde.

Veja o terceiro estágio na figura 17.

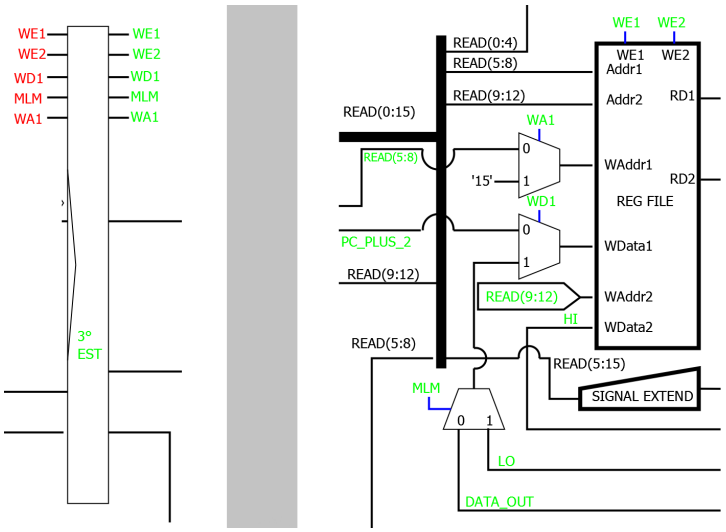

Figura 17: Terceiro estágio do pipeline - estágio verde

d. Unidade de Adiantamento

Um problema muito comum em processadores pipeline são os hazards. Nesta implementação do processador, tivemos dois tipo de hazards: de dados e de controle.

Para resolver o hazard de dados foi feita uma unidade de adiantamento, que busca o dado em estágios posteriores do pipeline antes de ele ser armazenado no banco de registradores no estágio de writeback. Veja o bloco na figura 18.

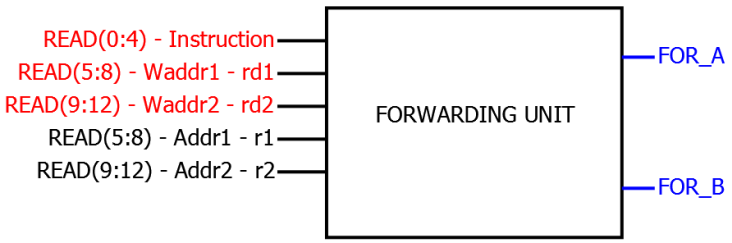

Figura 18: Unidade de adiantamento para o pipeline.

A unidade de adiantamento controla dois multiplexadores que decidem o que será usado pelo caminho de dados: a própria saída do banco de registradores, uma das saídas da ULA ou a saída da memória de dados. Veja na figura 19.

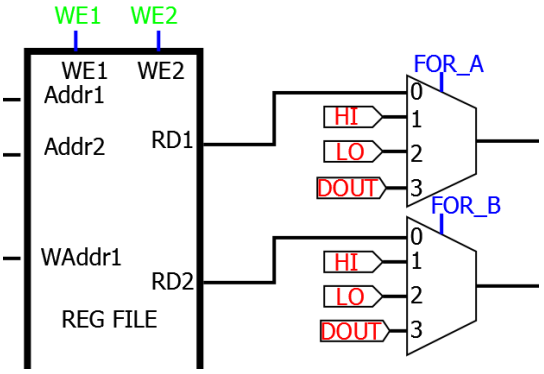

Figura 19: Multiplexadores de adiantamento para o pipeline

O funcionamento desta unidade é descrito a seguir. Primeiramente, é preciso saber se a instrução sendo executada no estágio vermelho é uma instrução da ULA ou da memória, para saber de onde o dado deve ser adiantado. Depois, é preciso seguir uma lógica para decidir se o dado deve ser adiantado ou não.

Quando o registrador a ser lido no banco de dados no primeiro estágio (r1 branco ou r2 branco) for igual ao registrador destino 1 do segundo estágio deve ser feito o adiantamento. Nesse caso, ou o dado vem da saída LO da ULA ou o dado vem da memória.

$$r1(branco) = rd1(vermelho) => FOR\_A = 2\ ou\ 3$$

$$r2(branco) = rd1(vermelho) => FOR\_B = 2\ ou\ 3$$

Caso o registrador a ser lido seja o registrador de destino 2, o dado deve vir da saída HI da ULA.

$$r1(branco) = rd2(vermelho) => FOR\_A = 1$$

$$r2(branco) = rd2(vermelho) => FOR\_B = 1$$

A unidade de adiantamento faz essas comparações e gerencia os multiplexadores de acordo.

e. Observação sobre BRANCH e JUMP

O outro hazard a ser tratado é o hazard de controle. Instruções BRANCH e JUMP mudam o valor do PC apenas no segundo estágio do pipeline, de

maneira que a instrução logo após o B. ou J. seja executada, contra a vontade do programador.
Como não foi possível fazer nenhum tipo de adiantamento ou criar outra solução para esse hazard, optamos por colocar uma instrução NOP depois de cada BRANCH ou JUMP que houver no código. Assim, a instrução NOP executada não afetará o programa.

f. Programa Fatorial

O programa utilizado para fazer o fatorial foi exatamente o mesmo usado no processador ciclo único. Veja o resultado na figura 20.

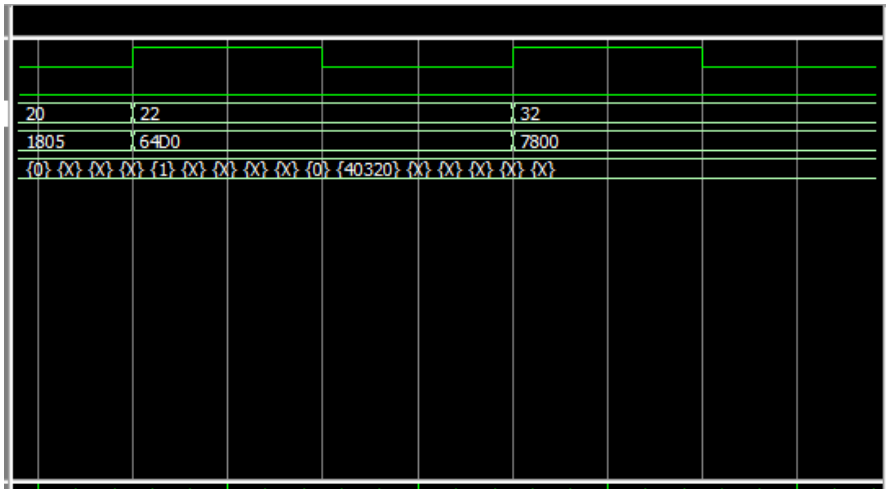

Figura 20: Resultado da simulação do pipeline

Também é possível ver que a instrução **lop12 $s0 $s1** (x"64D0") é executada mesmo quando há o desvio pelo BEQ (x"1805"). Basta conferir os registradores de operando A e B. A instrução carrega o índice n $s0 e o resultado $s1 em A e em B. Veja o registrador B carregando com 40320 no momento marcado pelo cursor na figura 21.

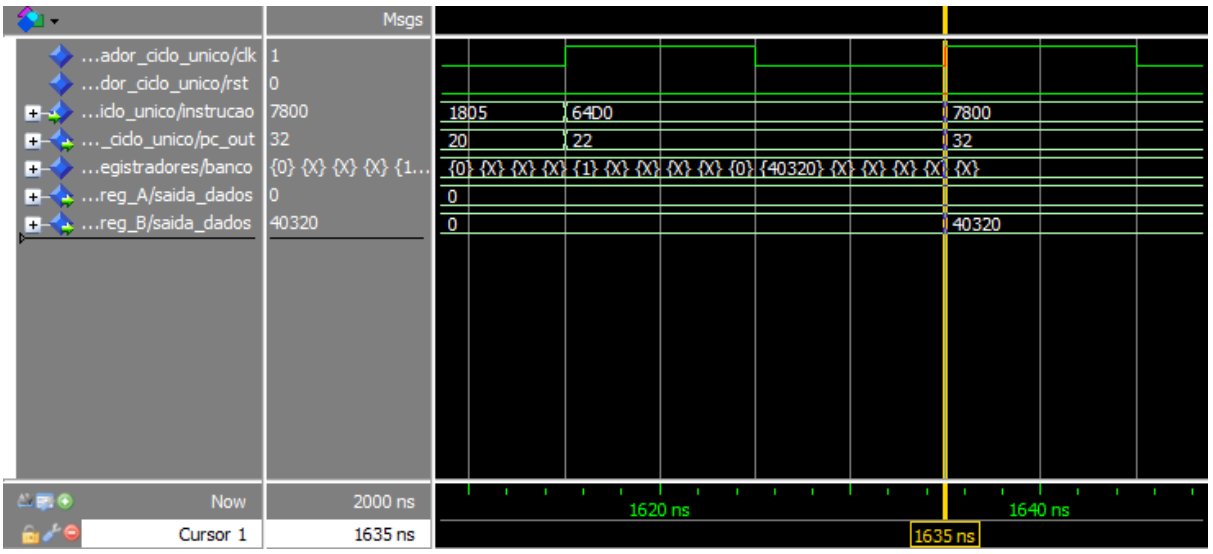

Figura 21: Instrução após o BRANCH sendo executada

5. Controladora de interrupções

O próximo passo do projeto foi fazer um processador capaz de lidar com interrupções. Veja o novo processador na figura 22.

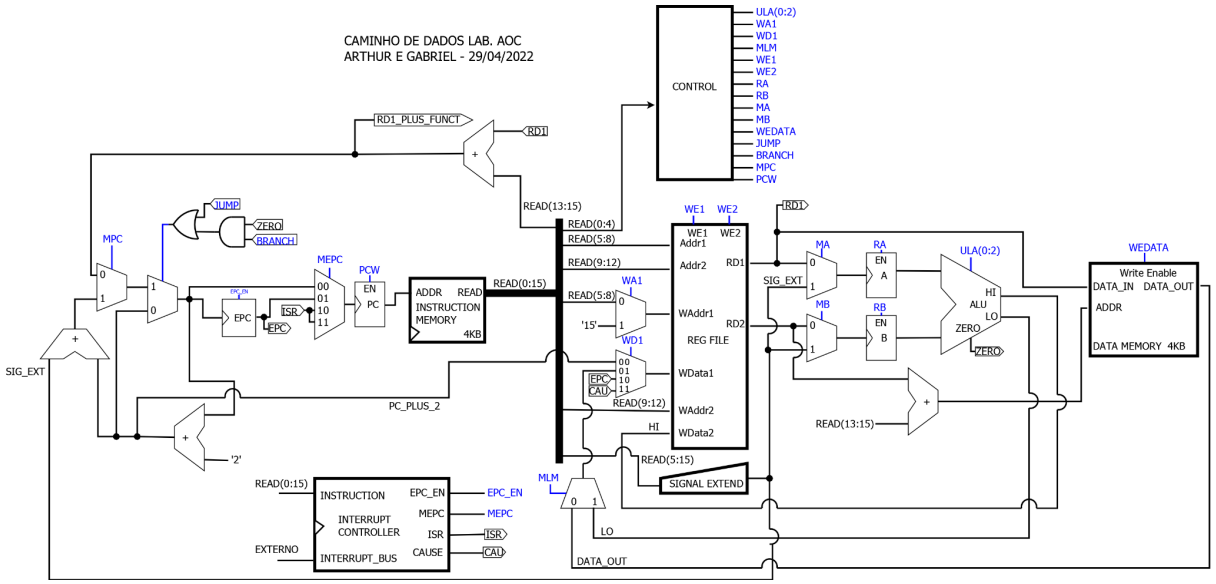

Figura 22: Processador com controladora de interrupções. Imagem maior na pasta.

Em uma frase, a controladora de interrupções desvia o fluxo de execução para uma subrotina quando algum evento ocorre e, após o fim da subrotina, retorna o fluxo ao normal. Para adicionar a controladora e cumprir esse objetivo, é preciso alterar o caminho de dados e criar instruções novas. Como tal controladora aumenta consideravelmente o nível de complexidade do sistema, optou-se por utilizar o processador ciclo único no projeto.

a. Funcionamento de Alto Nível

A controladora de interrupções monitora um barramento de interrupções INTERRUPT_BUS, como se pode ver na figura 23. Ela gera os sinais de controle EPC_EN e MEPC correspondentes quando ocorre uma interrupção, desviando o fluxo de execução. Além disso, ela também entrega o endereço da subrotina que deve ser executada para aquela interrupção por meio do sinal ISR. O sinal CAUSE é uma cópia de ISR para o caso de armazenar a causa da interrupção. A entrada INSTRUCTION serve para dar comandos à controladora, ela recebe a própria instrução do processador.

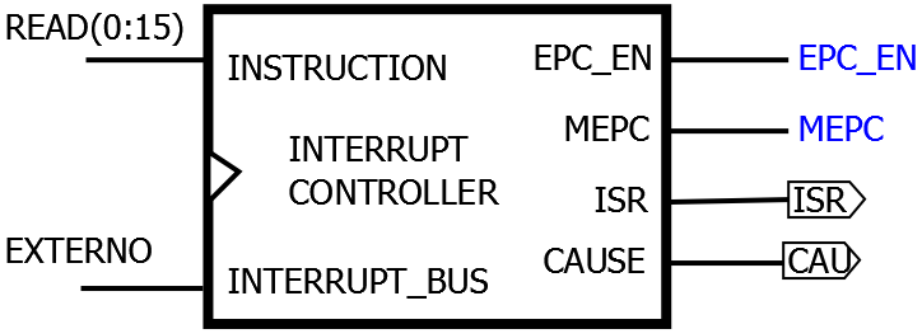

Figura 23: Controladora de interrupções utilizada

A controladora possui vários comandos como o de limpar flags de interrupção e o de habilitar/desabilitar interrupções de maneira geral. Uma descrição completa está no arquivo “Discriminacao_SYSCALL”. De maneira geral, o procedimento para tratar uma interrupção é o seguinte:
Primeiro, uma interrupção ocorre e automaticamente é feito o desvio do fluxo de execução. Dentro da rotina, deve ser chamada uma instrução de desabilitar as interrupções de maneira geral, para que o fluxo não seja desviado novamente. Em seguida, outra instrução deve avisar a controladora que a interrupção já foi atendida (limpar a flag). Depois, a rotina é executada normalmente e, ao terminar, as interrupções devem ser reabilitadas de maneira geral. Esse comando de reabilitar as interrupções faz com que a controladora retome o fluxo normal de execução e volte a aguardar outra interrupção.

b. Arquitetura

A controladora de interrupções utilizada consiste num núcleo (interrupt_ctl) fornecido pelo professor com alguns decodificadores e uma máquina de estados ligados a ele. Veja a arquitetura na figura 24.

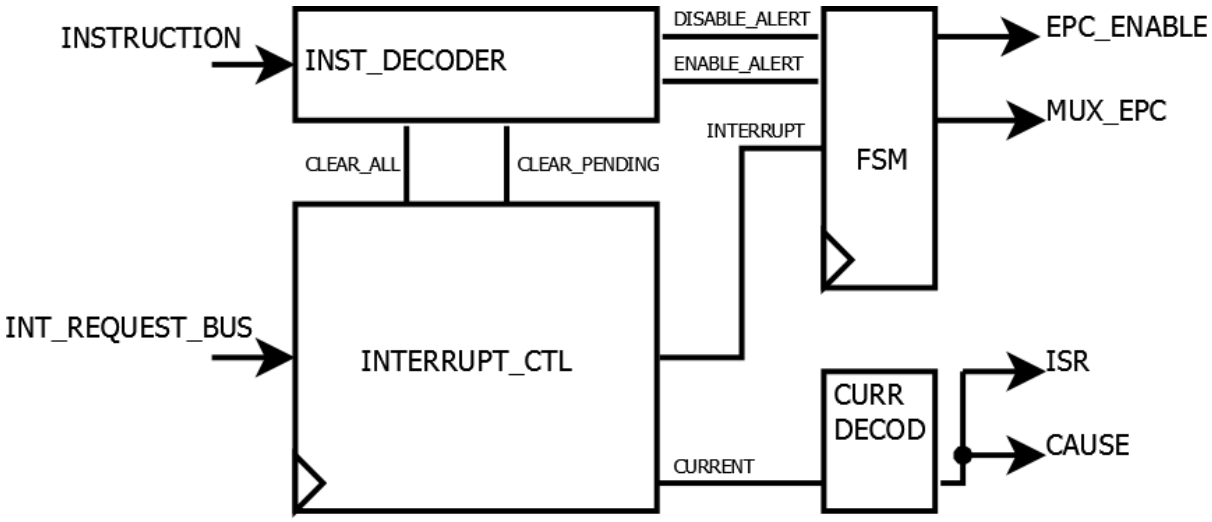

Figura 24: Arquitetura da controladora de interrupções

O decodificador de instruções transforma a instrução do processador em sinais de controle para o núcleo da controladora e para a máquina de estados.
O núcleo gerencia as flags de interrupção, ligando e desligando-as de acordo com os comandos e ainda avisa qual interrupção está sendo tratada no momento (sinal current).
O decodificador da interrupção corrente (Current Decod) recebe a instrução corrente e devolve o endereço da ISR correspondente.
A máquina de estados recebe os sinais tanto da instrução quanto do núcleo para controlar os sinais do caminho de dados.
A máquina de estados possui o seguinte diagrama de estados, mostrado na figura 25.

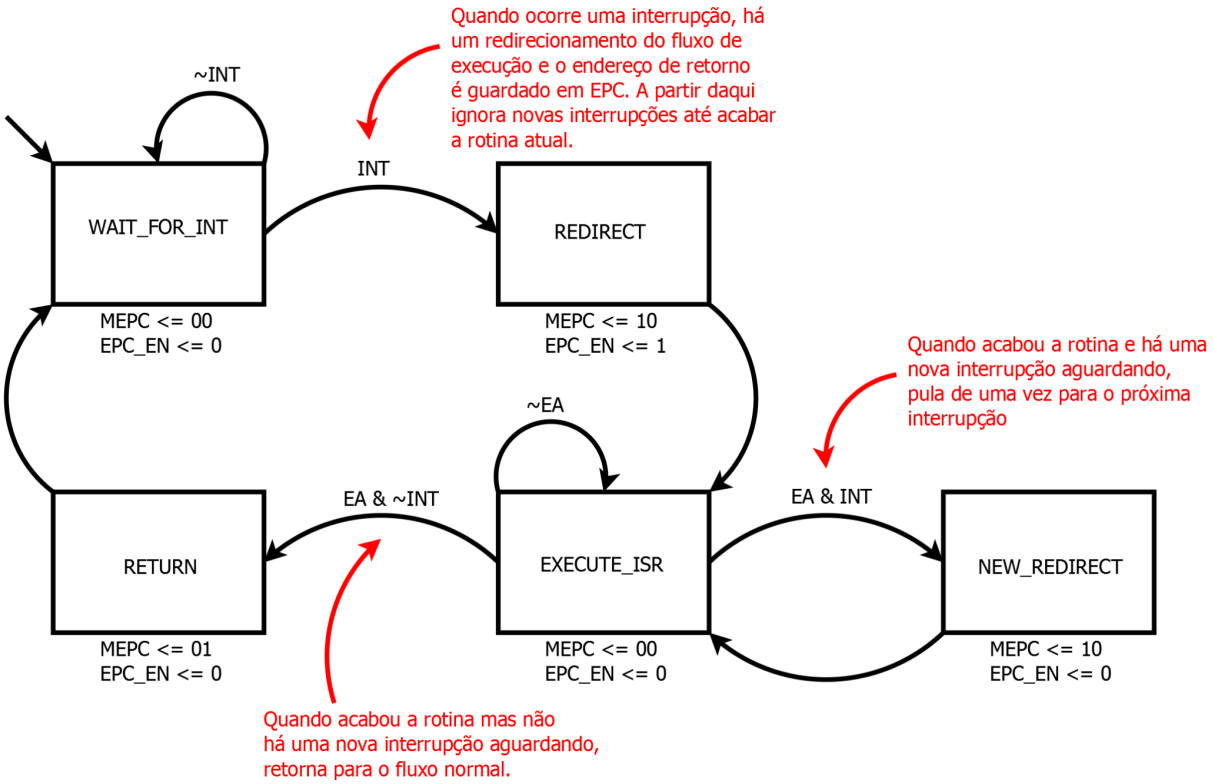

Figura 25: Diagrama de estados da FSM na controladora de interrupções

Como se pode ver na figura 25, a máquina de estados desvia o fluxo de execução quando recebe o sinal INTERRUPT do núcleo, e fica aguardando o fim da rotina de interrupção. Quando a rotina termina (Enable Alert = ‘1’), a máquina confere se há uma nova interrupção. Se houver uma nova interrupção, ela é atendida, caso contrário, o fluxo volta ao normal.

c. Implementação no Caminho de Dados

No caminho de dados foram adicionados a controladora de interrupções, um multiplexador 4x1 e um registrador EPC, que guarda o endereço de retorno. A controladora de interrupções é responsável pelo multiplexador e por habilitar o registrador EPC. Veja na figura 26.

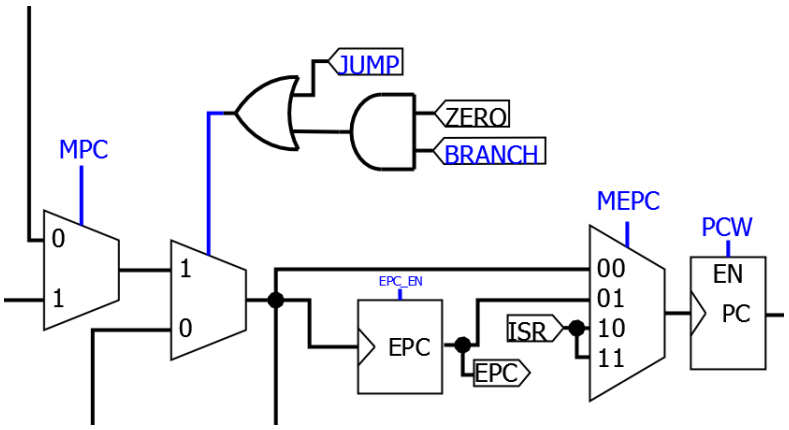

Figura 26: Modificação no caminho de dados para a controladora de interrupções

Além desses componentes, um dos multiplexadores na entrada do banco de registradores foi alterado de 2x1 para 4x1 com o objetivo de guardar EPC ou a causa de interrupção no banco. Veja na figura 27.

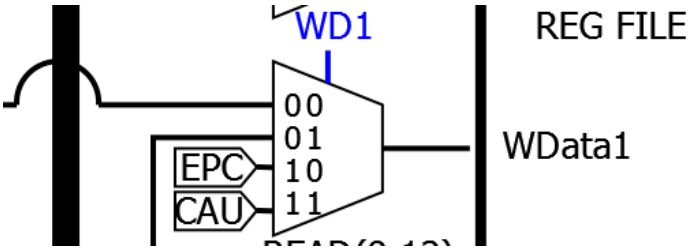

Figura 27: Multiplexador alterado no banco de dados

Também foi adicionado um sinal de overflow na ULA, que funciona somente para adição. Ele foi criado para gerar uma exceção a ser tratada. Veja na figura 28.

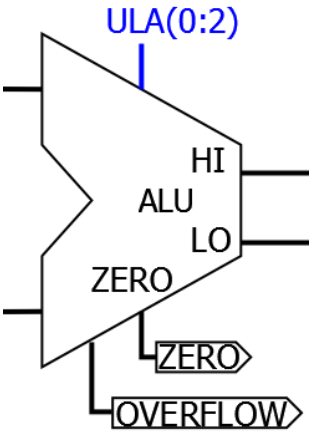

Figura 28: Sinal de overflow criado na ULA para teste de exceção

d. Simulação Overflow e Observação

O endereço da rotina de interrupção é 20, e como o PC guarda o dobro do endereço, o valor visto na simulação é 40. Após terminar a rotina, o

processador volta ao fluxo normal de execução. Veja o conteúdo da memória de instrução.

```
0       => X"43FF", -- LI1 1023
1       => X"4820", -- LI2 32
2       => X"7080", -- MUL $1
3       => X"6088", -- LOP12 $1 $1
4       => X"0900", -- ADD $2, deveria dar overflow
5       => X"43FF", -- LI1 1023
6       => X"4820", -- LI2 32
7       => X"7080", -- MUL $1
8       => X"6088", -- LOP12 $1 $1
9       => X"0900", -- ADD $2, deveria dar overflow

-- rotina de interrupção Overflow
20      => X"4821", -- LI2 33
21      => X"0001", -- SYSCALL para limpar a flag de interrupção
22      => X"0004", -- SYSCALL para dizer que acabou a rotina
23      => X"7800",  -- NOP porque a controladora leva um ciclo de clock para retornar
```

Veja o resultado da simulação nas figuras 29 e 30.

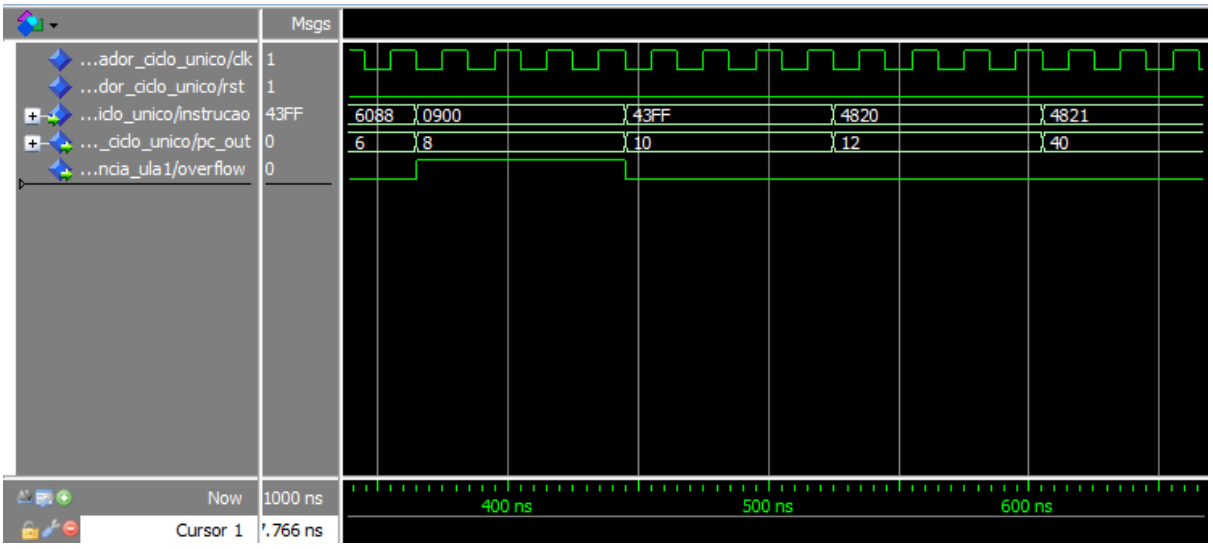

Figura 29: Desvio do fluxo de execução pela controladora de interrupções após ocorrer um overflow

A figura 29 mostra o desvio (PC = 12 pula para PC = 40) da controladora de interrupções alguns momentos após a ocorrência do overflow. Esse atraso se deve ao tempo que a controladora leva para guardar a interrupção e ao tempo que a FSM leva para chegar no estado em que o desvio ocorre efetivamente.

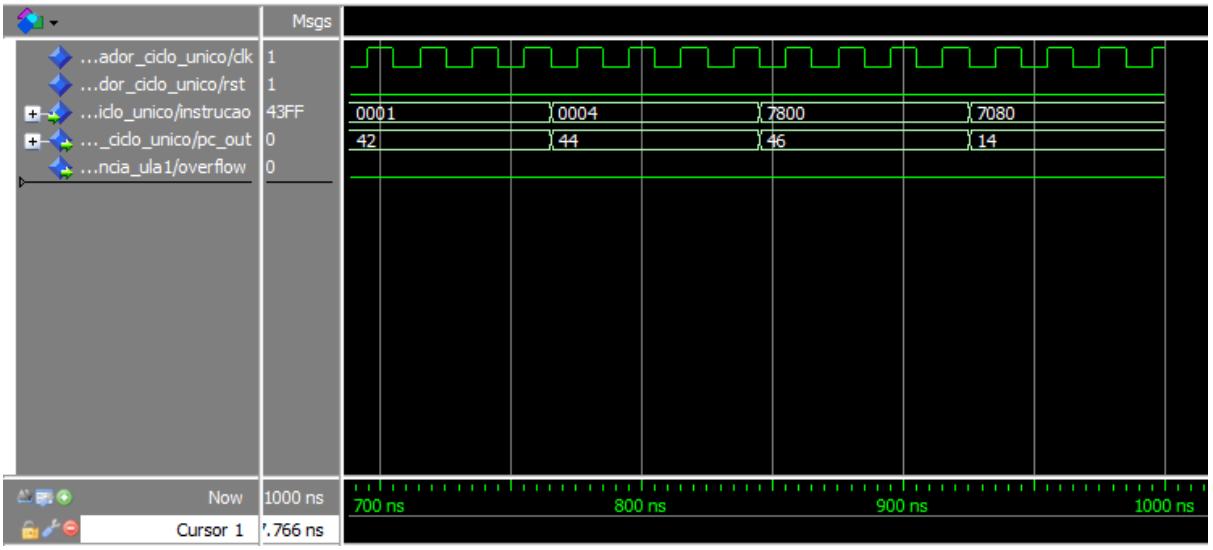

Figura 30: Retorno ao fluxo normal de execução pela controladora de interrupções

A figura 30 mostra o processador voltando ao fluxo normal de execução (PC = 46 pula para PC = 14) da maneira correta. Apesar de o processador ter voltado para o ponto certo, há um problema. A última instrução da rotina de interrupção era a Enable Alert (x"0004"), mas uma outra instrução (NOP = x"7800") é executada antes de retornar ao fluxo normal.

O problema encontrado se deve ao fato de a máquina de estados demorar um ciclo de relógio para realizar o desvio de fluxo, e pode ser contornado colocando uma instrução NOP ao fim de cada rotina de interrupção.

6. General Purpose Input Output (GPIO)

Uma vez implementada a controladora de interrupções, foi iniciada a montagem dos periféricos. O periférico adicionado ao processador foi o General Purpose Input Output (GPIO). Veja o caminho de dados com o GPIO na figura 31.

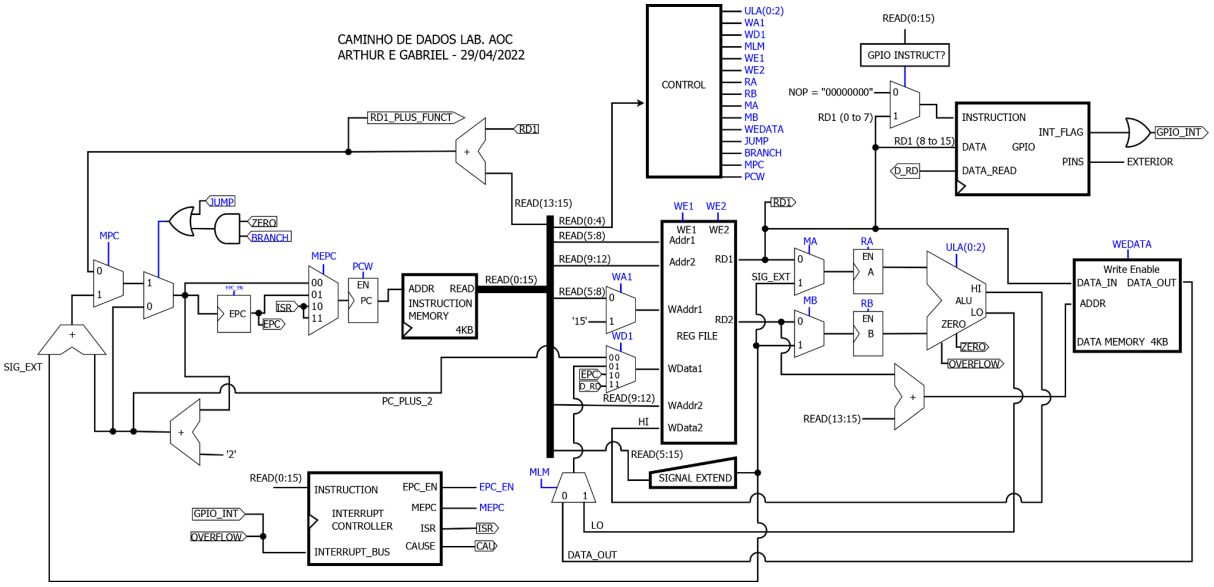

Figura 31: Caminho de dados com GPIO

a. Funcionamento de Alto Nível

O GPIO possui uma entrada de instrução, por onde ele recebe o comando a ser realizado e uma entrada de dados para carregar os registradores internos. Ele possui duas saídas: o barramento de interrupções e os dados lidos. Os pinos que comunicam com o mundo externo podem servir tanto de entrada quanto de saída. Veja a controladora GPIO na figura 32.

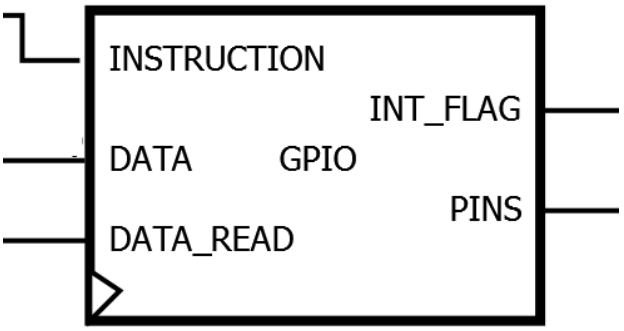

Figura 32: Bloco da controladora GPIO

As instruções desta controladora estão no arquivo “Discriminacao_SYSCALL”, sendo elas de configuração do periférico ou de leitura e escrita de dados nos pinos. Basta colocar a instrução (do GPIO) na porta INSTRUCTION e executar a instrução(do processador) “SYSCALL GPIO Command” que no próximo pulso de relógio ela será executada.
Tanto a instrução(do GPIO) quanto os dados vêm de um registrador específico (r9) no qual devem ser armazenados os bits. Veja na figura 33 o formato dos bits no registrador.

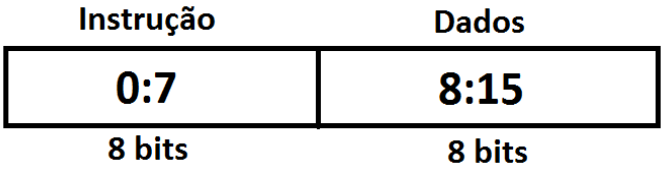

Figura 33: Formato dos bits para realizar comandos no GPIO

b. Arquitetura

A controladora de GPIO possui dois blocos: o núcleo, que interage com os pinos e configura o funcionamento, e o decodificador, que recebe o comando do GPIO e gerencia os sinais que habilitam os registradores. Veja um desenho na figura 34.

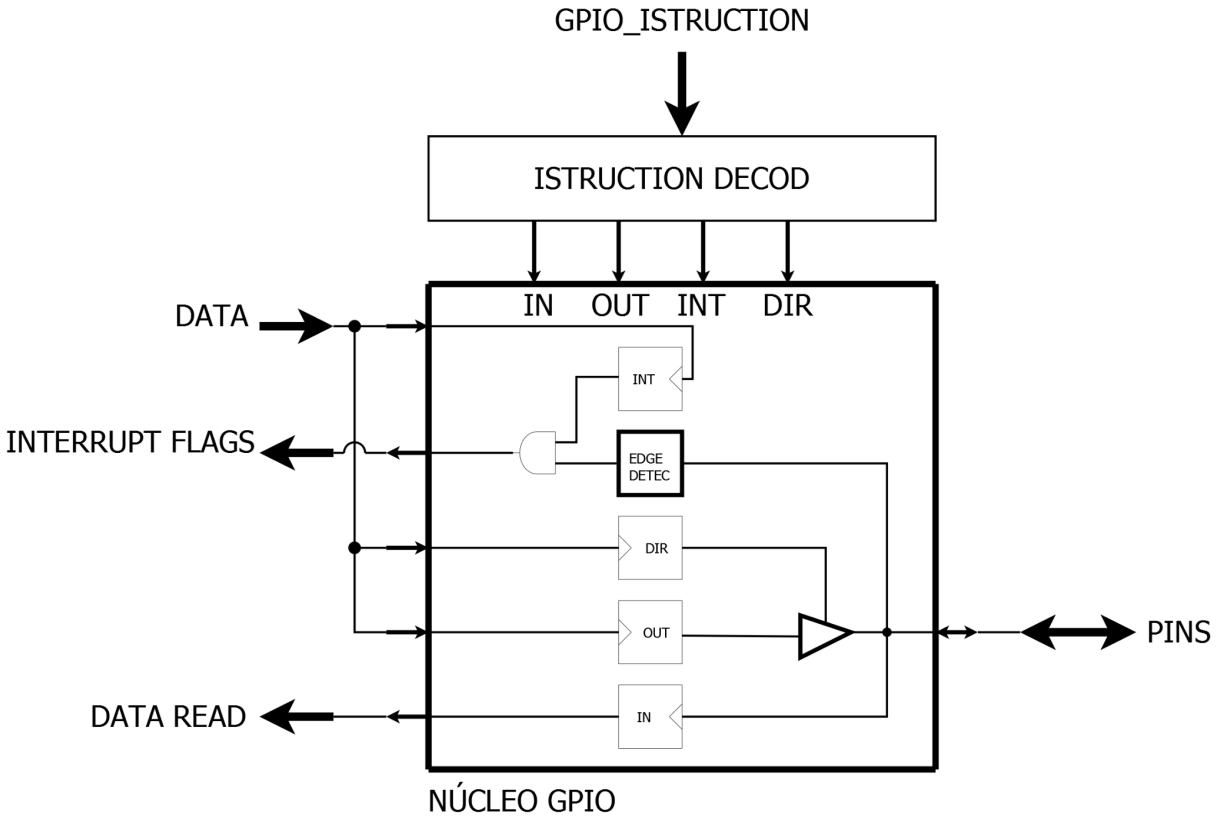

Figura 34: Arquitetura interna da controladora de GPIO

O principal objetivo de incluir um codificador e criar comandos especiais para o GPIO foi o de simplificar seu uso. Os 10 sinais que o núcleo possui se tornam apenas 5 após o encapsulamento.

c. Implementação no Caminho de Dados

O posicionamento do GPIO no caminho de dados é bem simples. A entrada de comando e de dados vem do registrador rd1 do banco de registradores. Um multiplexador, no entanto, pode impedir o GPIO de operar forçando um comando NOP em sua entrada. Esse MUX só permite o comando chegar ao GPIO quando o processador estiver executando a instrução “SYSCALL GPIO Command”, que é conferida pelo bloco “GPIO INSTRUCT?”. Veja a figura 35.

O barramento de interrupção passa por uma porta OR, de maneira que uma interrupção em qualquer um dos pinos ativará o sinal GPIO_INT. O sinal DATA_READ vai até o banco de registradores para ser armazenado. Os pinos vão para o exterior do sistema.

O módulo GPIO instanciado foi chamado de PORT_A.

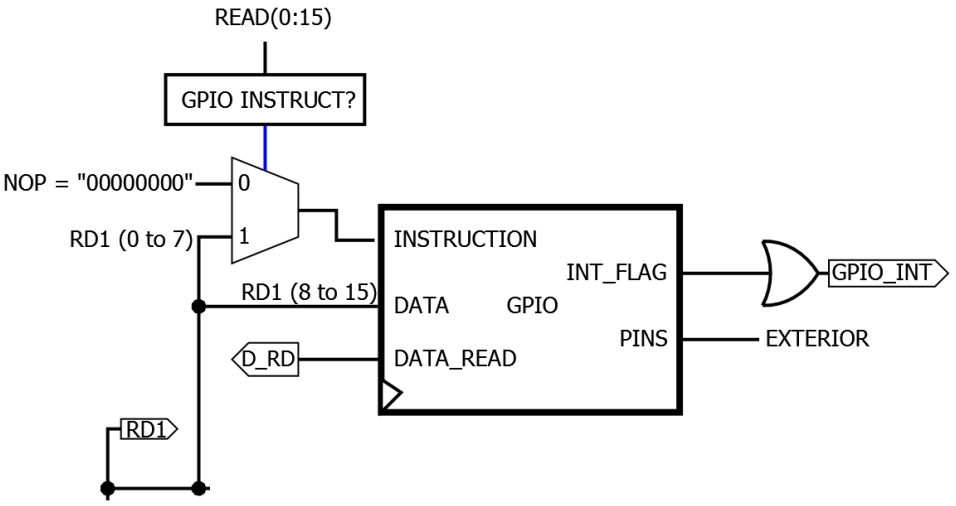

Figura 35: Implementação da controladora de GPIO no caminho de dados.

d. Simulação de Interrupção

Foi feita uma simulação na qual o GPIO é configurado para receber interrupções e, logo em seguida, uma interrupção é disparada. A seguir o código principal e a rotina de interrupção na memória de instruções.

```
0       => X"4003", -- LI1 3 (instrucao GPIO Interrupt Enable Configure)
1       => X"4900", -- LI2 256 (para << 8)
2       => X"7480", -- MUL $9
3       => X"40FF", -- LI1 3 (8 bits 1 seguindos)
4       => X"5848", -- LOP2 $9
5       => X"8C80", -- OR $9 (guarda a instrução GPIO completa em $9)
6       => X"0485", -- GPIO Command (habilita interrupcao de todo mundo)

-- rotina de interrupção Overflow
20      => X"4821", -- LI2 33
21      => X"0001", -- SYSCALL para limpar a flag de interrupção
22      => X"0004", -- SYSCALL para dizer que acabou a rotina
23      => X"7800",  -- NOP porque a controladora leva um ciclo de clock para retornar

-- rotina de interrupção do GPIO
32      => X"4822", -- LI2 34
33      => X"0001", -- SYSCALL para limpar a flag de interrupção
34      => X"0004", -- SYSCALL para dizer que acabou a rotina
35      => X"7800",  -- NOP porque a controladora leva um ciclo de clock para retornar
```

Nesse código, o processador configura o GPIO por meio das instruções de 0 a 6 e fica ocioso até ocorrer uma interrupção. Veja o registrador do GPIO sendo carregado com “1111 1111” na figura 36.

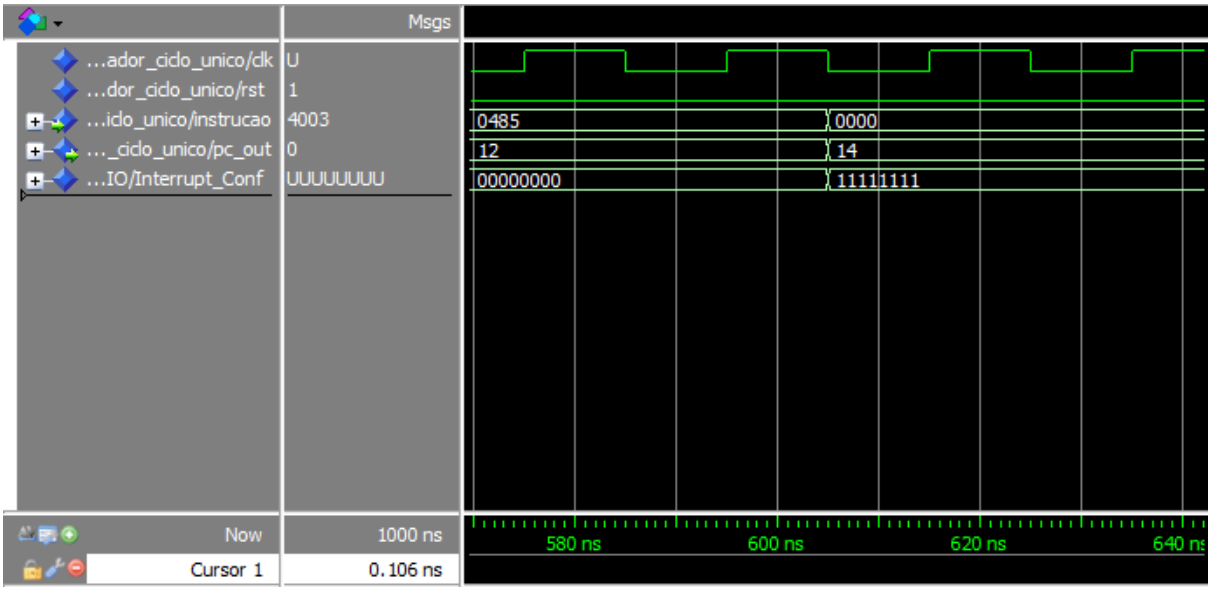

Figura 36: Registrador de configuração do GPIO sendo carregado

O registrador Interrupt_Conf totalmente habilitado ativa a interrupção para todos os pinos do GPIO.
Em seguida, é feita uma alteração nos pinos do PORT_A para que ocorra uma interrupção. Veja na figura 37.

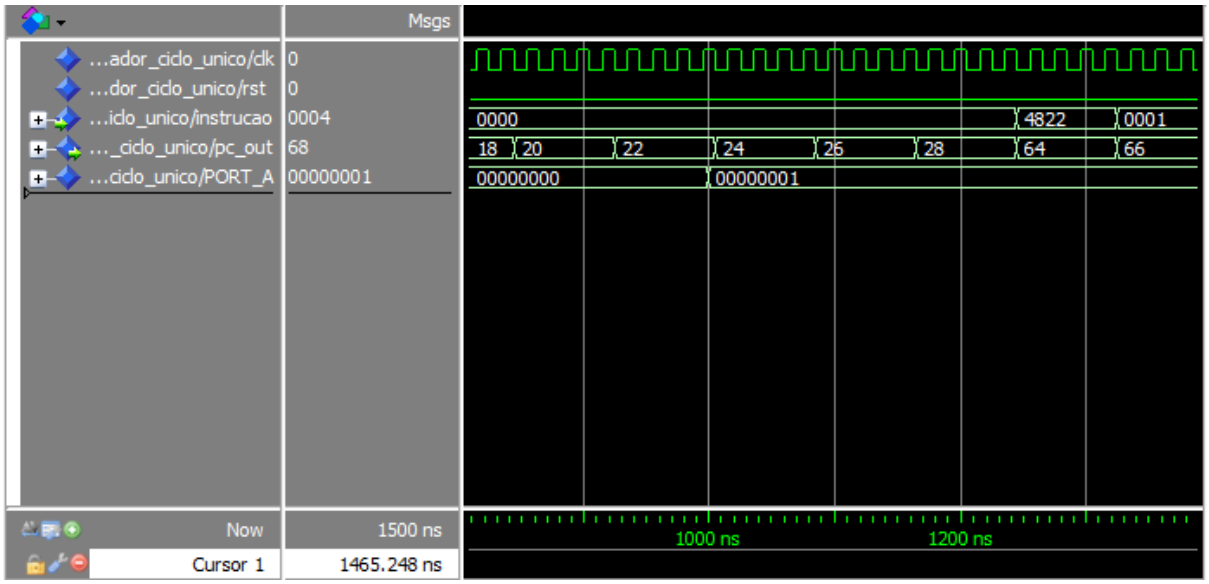

Figura 37: Alteração no fluxo de execução devido a uma interrupção gerada pelo GPIO

Note que PC salta de 28 para 64 algumas instruções após a alteração no PORT_A. PC = 64 é justamente a posição que aponta para a rotina de interrupção do GPIO.

7. Conclusão

O trabalho aqui desenvolvido teve como produto um processador ciclo único e outro pipeline funcionais, sendo que o processador ciclo único recebeu alguns periféricos. Apesar das simulações feitas apontarem para o funcionamento correto, não foi

possível testar todos os casos de operação e, por isso, há a possibilidade de haver erros no circuito desenvolvido.

Algumas características não esperadas do circuito surgiram durante o desenvolvimento do trabalho, como por exemplo o fato de a rotina de interrupção executar uma instrução a mais antes de retornar ao fluxo normal.

Os processadores (sem periféricos) ciclo único e pipeline foram testados em FPGA com sucesso, e foram utilizadas ferramentas do Quartus para facilitar a depuração.

A atividade exigiu esforço criativo para projetar cada parte, levando o grupo a rever princípios de arquitetura de sistemas digitais e de computadores. Uma atividade neste formato simula a função de um engenheiro projetista sendo, portanto, enriquecedora para o curso de graduação.