Relatório do trabalho ULA e Banco de Registradores

Alyson Gonçalves Jorge[1], Natanael Tagliaferro Galafassi[2], Vinicius Kurek[3]

Universidade Tecnológica Federal do Paraná – UTFPR

COCIC – Coordenação do Curso de Bacharelado em Ciência da Computação

Campo Mourão, Paraná, Brasil

[1]alysonjorge@alunos.utfpr.edu.br

[2]natanaelgalafassi@alunos.utfpr.edu.br

[3]viniciuskurek@alunos.utfpr.edu.br

Resumo

Este trabalho tem como objetivo criar uma arquitetura de 32 bits baseada na arquitetura *MIPS,* a qual será construída pela junção dos componentes que virão a ser desenvolvidos, como a Unidade Lógica e Aritmética, um Banco de Registradores, Decodificador, Extensor de Bits, Memória de Instruções e Memória de Dados por meio de um simulador lógico utilizado pela disciplina de Arquitetura e Organização de Computadores, o Logisim.

- **1.0 Introdução**

A arquitetura consiste no conjunto do funcionamento de diversos componentes, este relatório visa explicar o funcionamento e construção de cada um dos componentes desenvolvidos para a formulação do DataPath de nossa arquitetura. Aqui será possível ver detalhes sobre a **ULA**, o **Banco de Registradores**, **Memória de Instruções, Decodificador, Extensor de sinal** a **Memória de Dados**, **Unidade de Controle, Unidade de Controle da ULA, e adicionalmente a Unidade de Controle de Exceções.**

- **2.0 Unidade lógica e aritmética (ULA)**

A ULA é a peça fundamental da Unidade Central de Processamento (UCP) nela podemos executar as principais operações lógicas e aritméticas de um computador, nossa ULA Possue 4 entradas e 3 saídas que no caso das entradas são: A que seria o primeiro registrador de 32 bits; B que seria o segundo registrador de 32 bits; Operação ALU de 4 bits que em conjunto com um multiplexador dita a operação a ser realizada, o shamt de 5 bits que dita a quantidade de bits a serem deslocadas em uma operação de shift. e nas saídas temos: a saída result que mostra o resultado da operação realizada, a saída ZERO que apenas será 1 em caso de toda a expressão de result for 0 e o overflow que apenas será 1 se houver overflow. Nossa ULA é capaz de realizar as operações de AND, OR, NOR, XOR, SLT, ADD, SUB, SLL e SRL

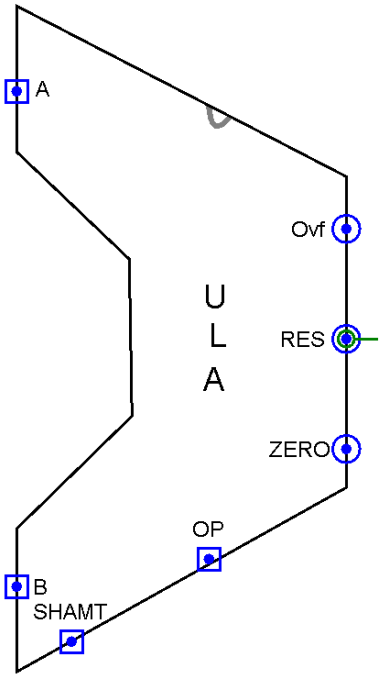


Imagem 2.0.1: Imagem da ULA em nosso circuito no Logisim.

Operação aritmética ou lógica	Número da operação em ALU	Tipo da operação:
AND	0000	Lógica
OR	0001	Lógica

ADD	0010	Aritmética
XOR	0011	Lógica
SLL	0100	Lógica
SLR	0101	Lógica
SUB	0110	Aritmética
SLT	0111	Lógica
NOR	1100	Lógica

Na nossa ULA também fizemos alterações para fazer o tratamento de exceção em caso de overflow, para isso foi feito um circuito a parte que verifica 4 tipos de overflow, 2 em adição e 2 em subtração, na adição será disparado exceção caso dois negativos se tornem um positivo, e caso dois positivos se tornem um negativo, já na subtração ocorrerá overflow quando A for positivo e B for negativo e o resultado gerar um negativo, e quando A for negativo e B for positivo e o resultado for um positivo, para realizar essas verificações utilizamos o verificador pronto do Simulador Logisim.

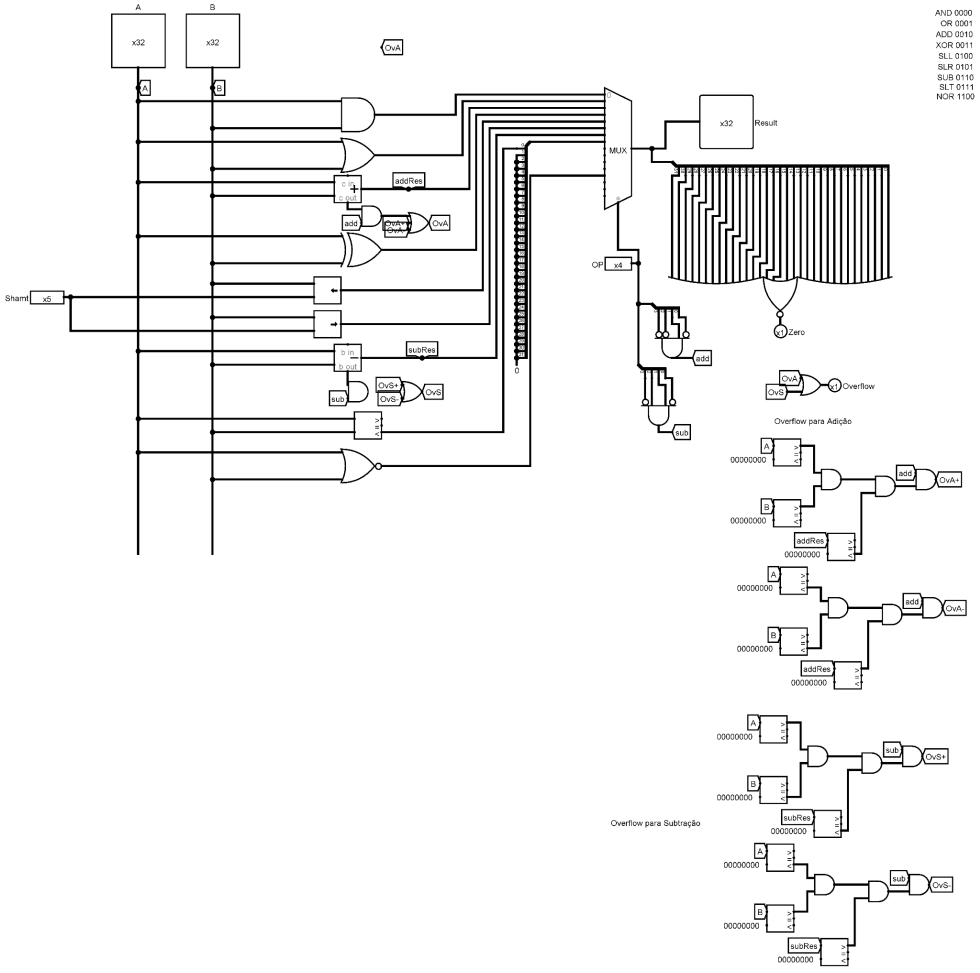


Imagem 2.0.2: Imagem do circuito da nossa ULA

2.1 Operação AND

Utilizamos a operação AND fornecida pelo simulador Logisim, na qual a operação verifica de bit a bit na expressão e emite 1 na saída Result na posição onde as entradas A e B são simultaneamente 1.

2.2 Operação OR

Foi utilizada a operação OR fornecida pelo simulador Logisim, na qual a operação verifica de bit a bit na expressão e emite 1 na saída Result na posição onde qualquer uma das entradas A ou B forem 1.

2.3 Operação NOR

Foi utilizada a operação NOR fornecida pelo simulador Logisim, na qual a operação realiza a verificação bit a bit e realiza a ação inversa da porta OR, emitindo 1 na saída Result apenas se ambas as portas A e B forem 0.

2.4 Operação SLT

A operação SLT foi feita com o Comparador fornecido pelo simulador Logisim, na qual possui duas entradas e três saídas, como SLT é set if less than conectamos a saída que emite 1 no bit menos significativo caso A seja menor que B

2.5 Operação ADD

A operação ADD foi realizada com o somador fornecido pelo simulador Logisim, onde possui três entradas e duas saídas, as duas primeiras entradas são dos números que queremos somar e a terceira entrada é o carry in que nada mais é que o carry out de uma expressão passada, como não há expressão passada não há nada conectado nesta entrada, para detectarmos o carry out da expressão sem haver problemas com a operação SUB realizamos um decodificador em conjunto com uma porta AND que apenas emitirá o overflow caso a expressão que deu carry out seja a expressão chamada pela ALU.

2.6 Operação SUB

Muito similar com a Operação ADD, utilizamos o subtrator fornecido pelo simulador Logisim, onde conectamos A e B nas entradas que queremos que sejam subtraídos, sem Borrow in, e utilizamos a mesma lógica de um decodificador em conjunto com uma porta AND para emitir o Underflow caso a expressão chamada pela ALU seja a de SUB.

2.7 Operação SLL

A operação Shift Left Logical foi realizada com o Deslocador com seu tipo de deslocamento sendo lógico a esquerda fornecida pelo simulador Logisim, onde conectamos o Pino A em conjunto com o pino Shamt de 5 bits para deslocar para esquerda o valor de A pelo mesmo número de vezes carregado em Shamt, o que posiciona 0 no bit menos significativo a cada salto realizado.

2.8 Operação SRL

Seguindo o raciocínio da operação SLL porém colocando seu tipo de deslocamento para Lógico à direita, na operação Shift Right Logical colocamos a entrada A em conjunto com o Entrada Shamt para realizar os saltos para a direita colocando 0 no bit mais significativo a cada salto.

● 3.0 Banco de registradores

O banco de registradores é uma área no computador onde estão os registradores de uso geral, utilizamos 32 registradores de 32 bits do simulador Logisim para criação desse banco de registradores, os registradores possuem 3 entradas e uma única saída, onde podemos adicionar um valor que está armazenado na entrada WD utilizando a entrada WR que define o registrador a ser inserido em conjunto com a entrada Regwrite com um demultiplexador que emite o sinal que queremos realmente sobrescrever o conteúdo deste registrador com o conteúdo de WD, a ação de escrita só será realizada a partir de um pulso de clock enviado no registrador, o Único registrador que não pode ser sobrescrito é o registrador de posição 0 que sempre terá o número 0 armazenado . Para fazer a leitura do valor contido no registrador utilizamos as entradas LR1 e LR2 de 5 bits que definem quais serão os registradores a serem lidos, utilizamos 1 multiplexador em LR1 e 1 multiplexador em LR2, ambos de 32 bits, conectados em cada registrador do banco de registradores, o valor do registrador deve ser inserido nas saídas LD1 e LD2 de 32 bits.

Número do Registrador Na Saída WR	Nome do Registrador	Pode ser escrito?
00000	$0	Não

00001	$at	Sim
00010	$v0	Sim
00011	$v1	Sim
00100	$a0	Sim
00101	$a1	Sim
00110	$a2	Sim
00111	$a3	Sim
01000	$t0	Sim
01001	$t1	Sim
01010	$t2	Sim
01011	$t3	Sim
01100	$t4	Sim
01101	$t5	Sim
01110	$t6	Sim
01111	$t7	Sim
10000	$s0	Sim
10001	$s1	Sim
10010	$s2	Sim
10011	$s3	Sim
10100	$s4	Sim
10101	$s5	Sim
10110	$s6	Sim
10111	$s7	Sim
11000	$t8	Sim
11001	$t9	Sim
11010	$k0	Sim
11011	$k1	Sim
11100	$gp	Sim
11101	$sp	Sim
11110	$fp	Sim

11111	$ra	Sim

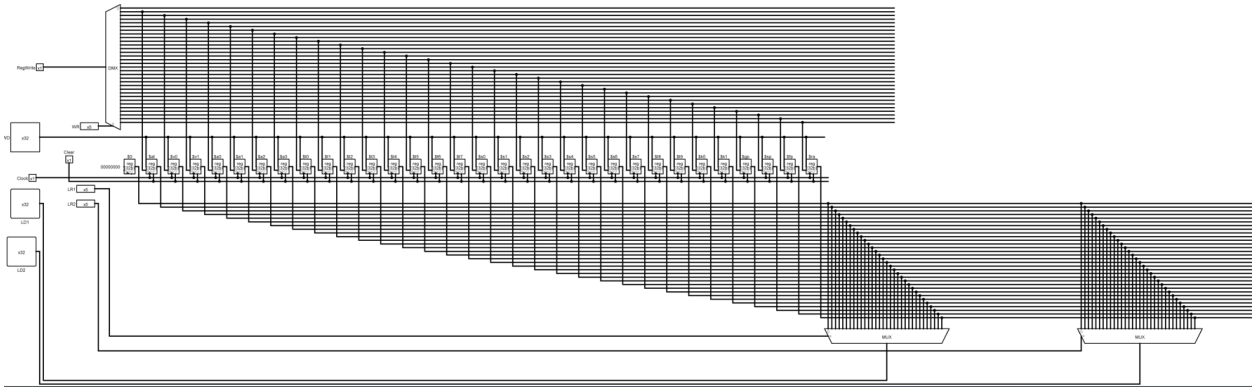


Imagem 3.1: uma imagem que representa circuito do banco de registradores

- **4.0 Memória de Instruções**

Utilizamos uma Memória ROM (*Read Only Memory)* para fazer a memória de instruções que é utilizada para armazenar o código fonte do programa a ser executado, ela recebe um endereço como entrada, e em sua saída dá a instrução que se encontra no endereço especificado pela entrada. Apesar de ser um componente que não precisamos construir no Logisim, ainda há a necessidade de fazer alguns ajustes devido às limitações do próprio software.

A princípio o software de simulação não nos permite utilizar uma memória de 2^32 bits para termos todos os endereços possíveis da nossa arquitetura como sendo armazenados, pois o limite de tamanho da ROM do Logisim é de 2^24 bits, sendo assim tivemos de utilizar uma segunda memória de 2^6 bits. Há de se notar que isso totaliza apenas 30 bits, por qual motivo apenas 30 ao invés dos 32? O motivo é simples, demos uma um shift lógico físico para esquerda nos nossos distribuidores a fim de fazer os endereços serem lidos de 4 em 4, pois estamos considerando o tamanho de uma instrução como sendo uma palavra do MIPS ou 4 bytes.

Outra coisa que a limitação do Logisim do tamanho da primeira ROM poder ser de até 2^24 bits de endereço é que temos que bloquear a escrita nela quando estamos na verdade querendo acessar os bits da segunda memória de 2^6 bits de endereço, para isso temos apenas uma porta lógica OR que verifica quando estamos querendo acessar a segunda memória e que bloqueia a escrita da primeira enquanto isso.

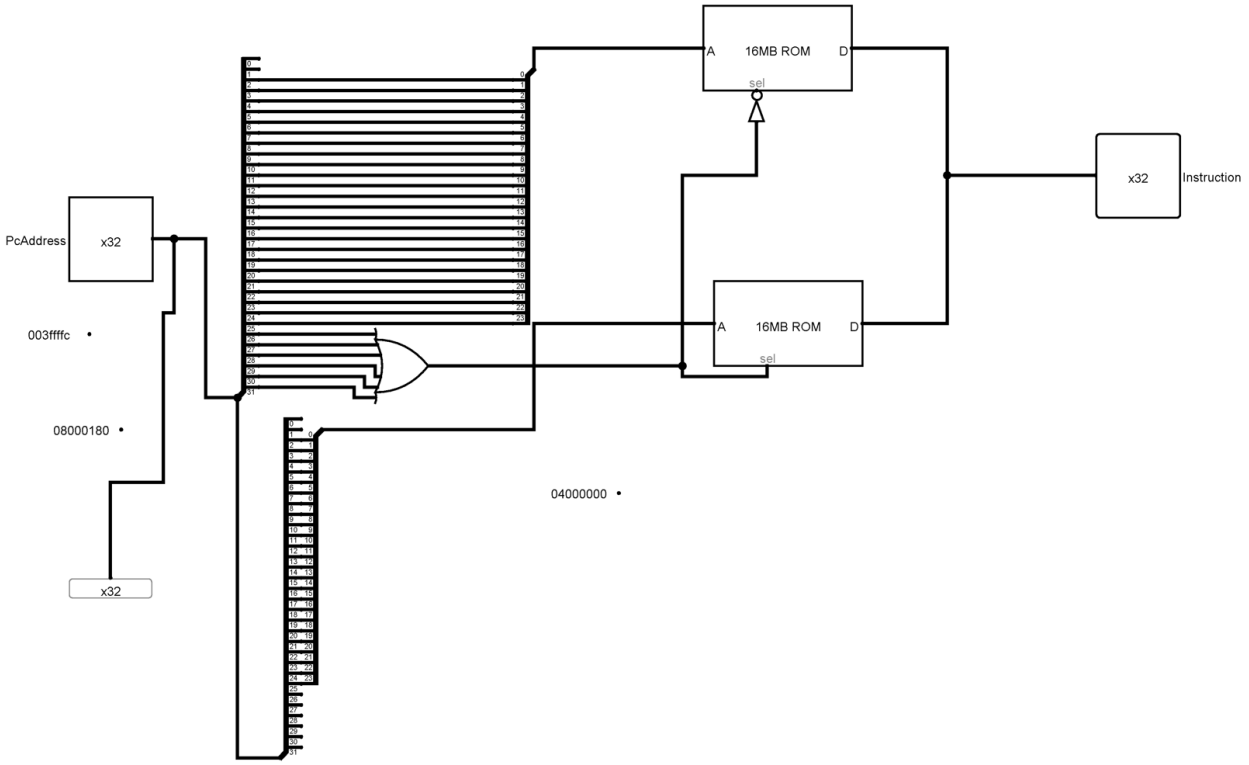


Imagem 4.1: Representação do circuito da memória de instruções

- **5.0 Decodificador**

Após o endereço ser lido na memória de instruções o caminho o qual ele deve seguir é de vir para este componente, o decodificador. O propósito do decodificador é nada mais que decodificar a instrução lida na memória de instruções para poder enviar de forma correta as informações ao hardware de acordo com o tipo de instrução lida, sendo ela do tipo R, I ou J.

Antes de começar a explicar a linha de raciocínio por trás do decodificador quero mostrar como as instruções R, I e J são divididas de acordo com os bits reservadas para elas:

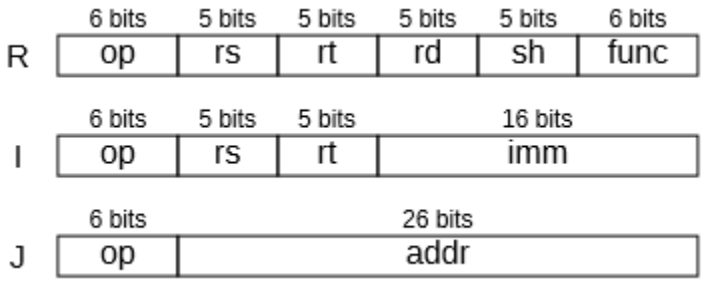


Imagem 5.1: Esquema de separação das instruções R, I e J de acordo com seus bits

Nota-se que todas os tipos de instruções possuem algo em comum entre elas, o código de operação de 6 bits, a partir disso conseguimos verificar o tipo da instrução. A primeira instrução a qual fazemos a verificação na nossa simulação é a do tipo R já que todas as instruções desse tipo possuem o código de operação 0 fica fácil decodificar ela, precisamos apenas de uma AND de 6 bits com todos eles sendo negados, o que faz com que essa porta retorne 1 quando os bits forem 0, verificando assim a operação R.

Após a verificação da operação R temos duas opções, verificar pelas operações I ou pelas operações J, optamos por uma verificação das instruções J por um motivo muito simples, existem apenas 2 instruções J as quais acabam por gerar apenas 2 códigos de operações diferentes, enquanto as da operação I possuem 20 códigos diferentes para as suas instruções, os quais não necessariamente iremos implementar mas nos servem como base a partir do MIPS ref card. Sabendo disso conseguimos verificar se a instrução é J apenas com mais duas ANDs uma que verifica uma instrução jump e outra um jal, com os respectivos códigos 2 e 3, sabendo esses códigos fazemos as ANDs para o jump e para o jal separadamente e caso a instrução não é do tipo R ou J ela automaticamente é do tipo I.

Daqui em diante é só separar de acordo com o que foi mostrado na imagem 5.1 e enviar às suas respectivas saídas do decodificador. Há de se notar que quando, por exemplo, uma instrução do tipo R é verificada, apenas os "campos" que essa instrução usa, possuem sinais de saída, ou seja, o imediato e o endereço são X ou não possuem valores nesse caso, pois não fazem diferença para nós.

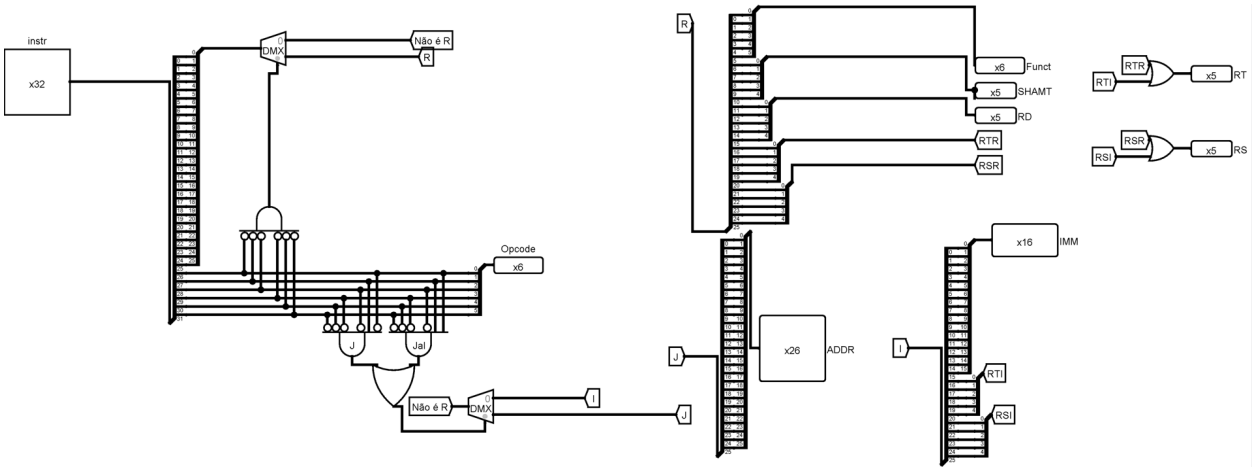


Imagem 5.2: Representação do Decodificador

- **6.0 Extensor de Bits**

O Extensor de Bits é utilizado na saída do imediato do decodificador para estender os bits de 16 bits para 32, onde os 16 que entram ficam nos menos significativos dos 32 e o último bit, o mais significativo, desses 16 é repetido para o restante dos 32.

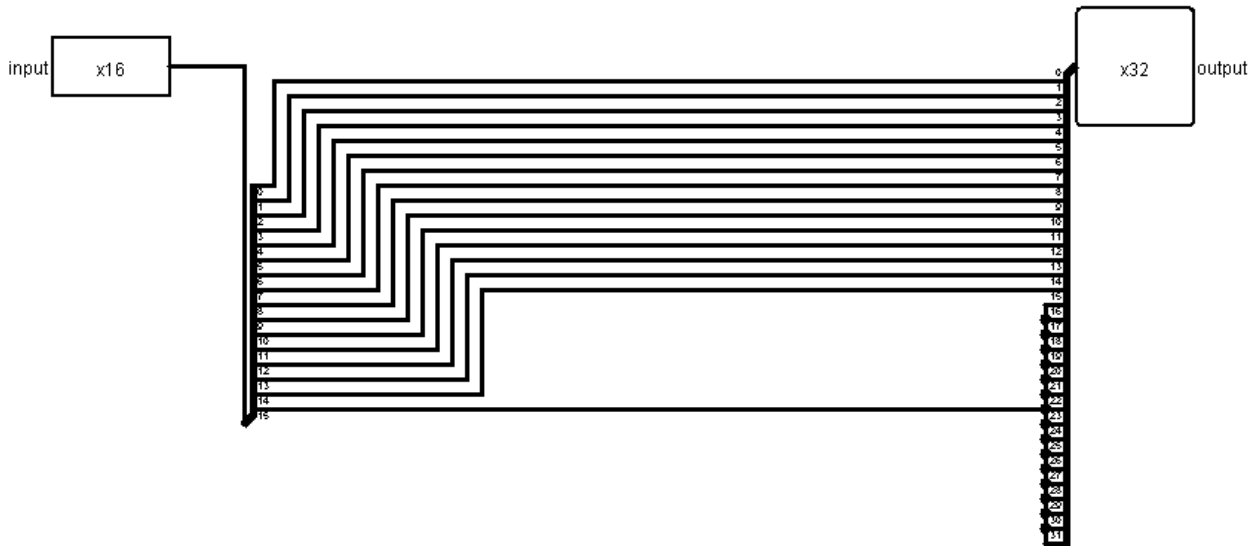


Imagem 6.1: Representação do extensor de bits

- **7.0 Memória de Dados**

A memória de dados encontra o mesmo problema de limitação pelo Logisim onde sua extensão de bits de endereço é limitado a 24 bits, então concatenamos duas memórias por meio de um distribuidor, ficando os 24 bits menos significativos para a primeira memória e os 6 mais significativos para a segunda. A diferença da memória de dados para a de instruções é que utilizamos uma memória RAM (Random Access Memory), se trata de uma memória volátil, no qual, ao ter sua energia cessada, perde-se os dados armazenados nela. A memória de dados possui 5 entradas, sendo uma para *clock*. Temos o **EL**, que é o seletor de endereço da memória, o **WD** que é o dado de 32 bits que será inserido na memória, **MemWrite** que é uma flag que permite a escrita de dados na memória, e **MemRead** que permite a saída dos dados da memória em um endereço específico para a saída **LD,** ou seja, permitindo a leitura do dado em um determinado endereço. Para evitar qualquer conflito com os dados das memórias concatenadas, foi colocado uma porta OR nos 6 bits mais significativos de **EL** (os quais também selecionam os endereços da segunda memória), sua saída foi dividida sendo inserida na entrada de seleção da segunda memória, e inserida com uma NOT na entrada de seleção da primeira. O shift também

foi realizado na memória de dados, pulando os 2 primeiros bits menos significativos do distribuidor.

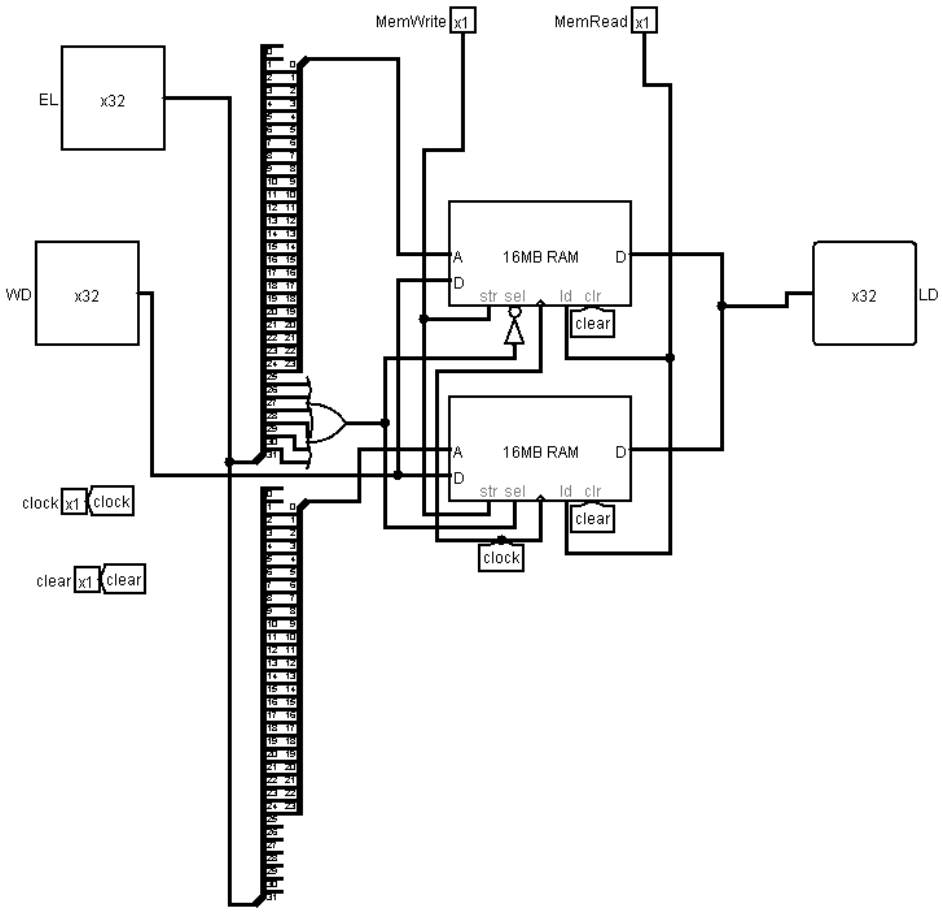


Imagem 7.1: Representação da memória de dados

- **8.0 Unidade de Controle**

Dado que a unidade de controle possui apenas uma entrada, o OpCode, é muito simples lidar com as suas devidas saídas. Por exemplo, utilizamos uma and com todas as portas negadas para que quando o OPCODE for 000000 (Ou seja, uma instrução de tipo R) ele possa emitir 1 nas saídas RegDst (permite RD seja ligado a entrada WR do banco de registradores), RegWrite (libera a escrita de um registrador), ExcOut (faz o valor de PC voltará para a execução do programa principal) e WriteReg (indicando que a operação é de escrita no registrador da unidade de exceção). Há de se notar que isso é apenas lidando com instruções para o tipo R as quais

recebem todas o mesmo OpCode, já para instruções I e J temos que verificar exatamente qual instrução para assim podermos dar a saída necessária fazendo o mesmo processo com ANDs para decodificar o código que estamos recebendo.

Instruções tipo I:

RFE: irá apenas enviar sinal 1 para ExcOut e WriteReg, OpCode 100000;

MTC0: irá apenas enviar 1 para MTC0, OpCode 100001;

MFC0: enviará 1 para MFC0 e WriteReg, OpCode 100010;

addi: enviará 1 para AluSrc, WriteReg e 01 para AluOp, OpCode 001000;

lw: enviará 1 para AluSrc, WriteReg, MemToReg, MemRead e 01 para AluOp, OpCode 100011;

sw: enviará 1 para AluSrc, MemWrite e 01 para AluOp, OpCode 101011;

beq: apenas branch será 1 e AluOp receberá 10, OpCode 000100;

bne: enviará 1 para branch e bne, AluOp receberá 10, OpCode 000101;

Intruções tipo J:

J: apenas jump receberá 1, OpCode 000010;

jal: receberão 1, link, jump e WriteReg, OpCode 000011;

Utilizamos um MUX para dar vários resultados a AluOp em para cada instrução, nas instruções do tipo R elas serão 00, já nas instruções do tipo I Addi, Lw ou Sw enviarão no AluOp 01 e nas instruções do tipo I beq ou bne enviará 10 em AluOp. AluOP 01 vai pedir uma soma na ULA e AluOp 10 vai pedir uma subtração.

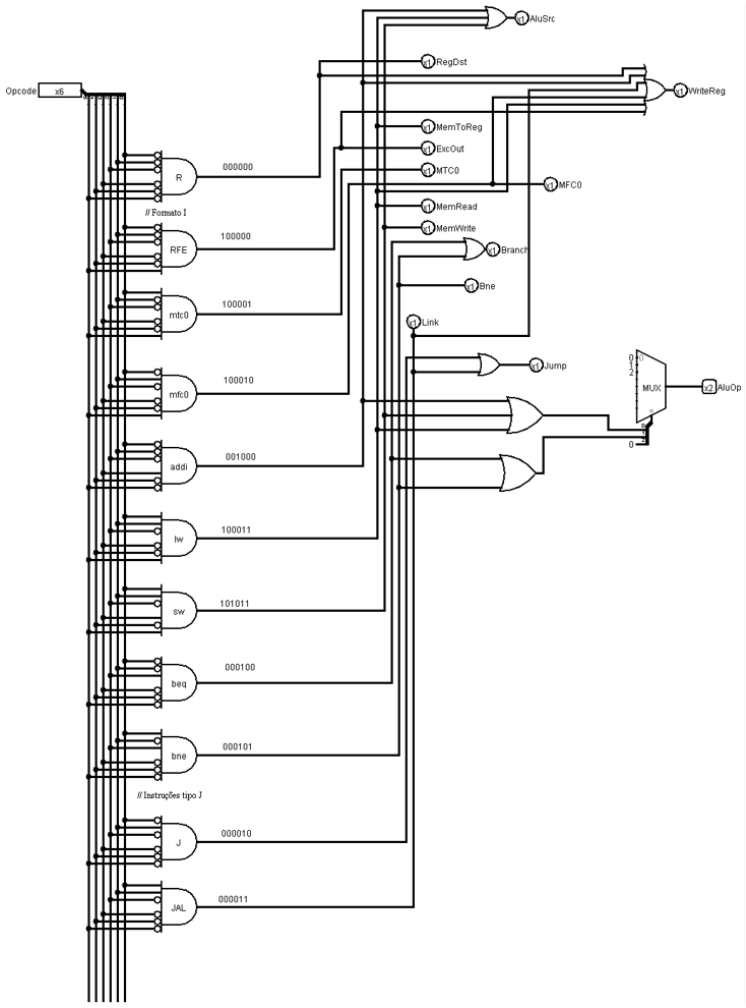


Imagem 8.1: Representação da unidade de controle

- **9.0 Unidade de controle da ULA**

Na unidade de controle da ULA utilizamos várias portas do tipo AND como forma de decodificar as instruções de funct que estão baseados no MIPS para o número da saída OP baseado na nossa ULA, para estender o sinal que saiu da AND utilizamos uma and com o valor do sinal que deverá ser encaminhado na saída OP, que está sendo passado em um MUX que utiliza o valor de ALUOP para selecionar a operação que deve ser feita.

Tipo de instrução	Número de funct	Número de OP
AND	100100	0000
OR	100101	0001
ADD	100000	0010
XOR	100110	0011

SLL	000000	0100
SRL	000010	0101
SUB	100010	0110
SLT	101010	0111
NOR	100111	1100
JR	001000	xxxx

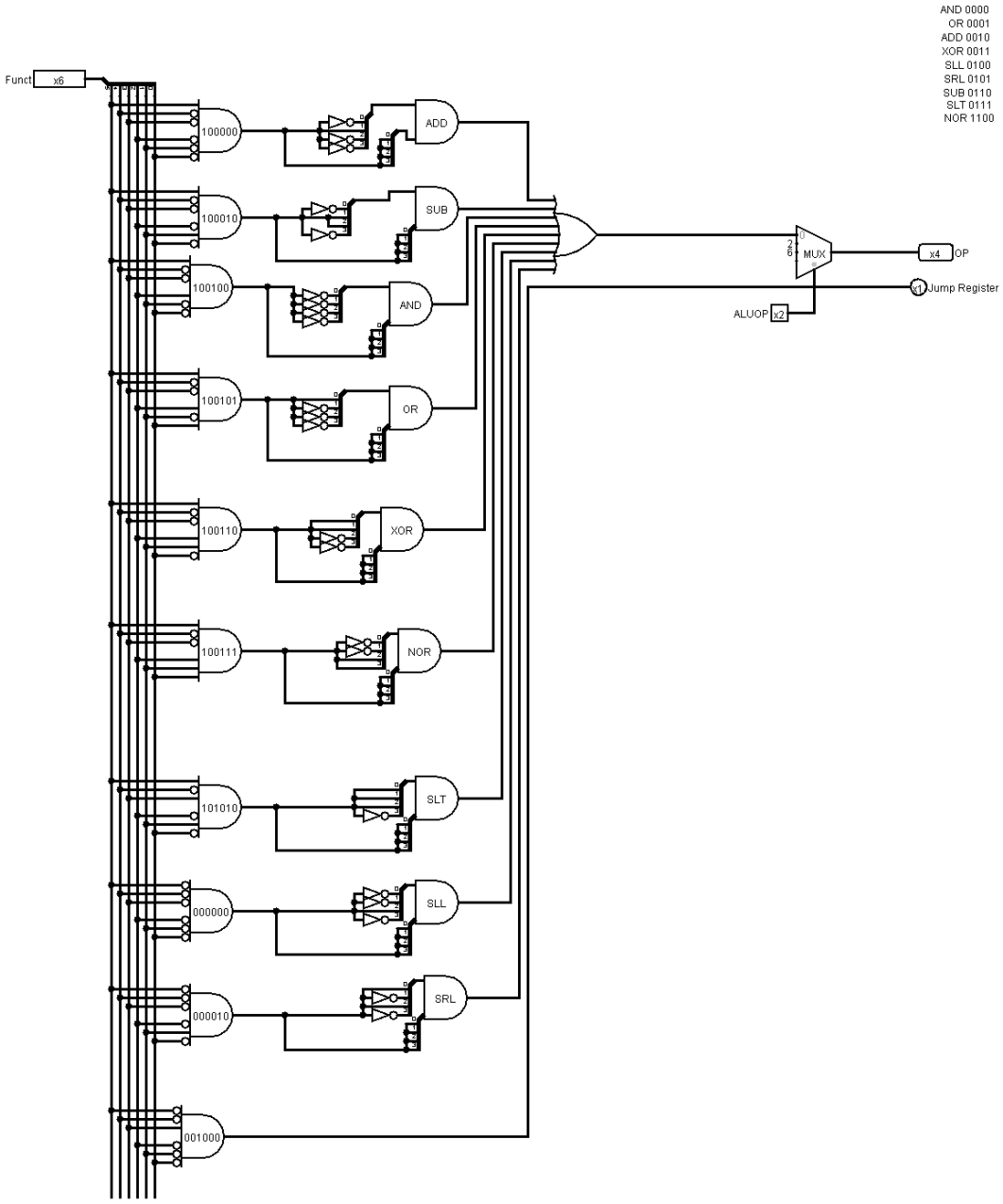


Imagem 9.1: Representação do circuito interno da unidade de controle da ULA

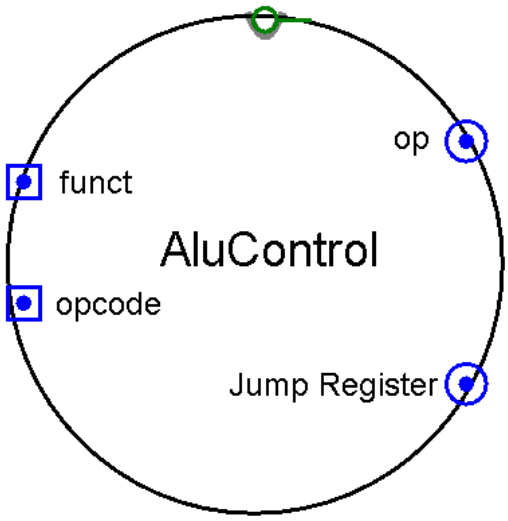


Imagem 9.2: Representação externa da unidade de controle da ULA

- **10.0 Unidade de Controle de Exceções**

A Unidade de Controle de Exceções (UCE) é um componente que lida com casos que fogem do fluxo de execução comum no Datapath. Geralmente este componente lida com diversos casos anômalos na arquitetura MIPS, contudo para fins de praticidade, neste trabalho ela recebeu modificações, lidando apenas com o caso de *Overflow*, e as instruções de exceção ao invés de serem tratadas no formato que são originalmente tratadas no MIPS, no caso formato R, sua versão modificada será tratada como sendo do formato I, sendo definidas na Unidade de Controle. Seu funcionamento, como dito anteriormente, se dará pela ocorrência de um *Overflow* na ULA, o qual ativará uma *flag* denominada "Exception", que servirá como gatilho de ativação para a UCE.

A UCE consiste basicamente de 3 registradores:

- EPC: Registrador que receberá a instrução seguinte à instrução que ocasionou a exceção.
- WR: Registrador que vai conter o identificador do registrador que iria receber a instrução incorreta.
- Res: Registrador que vai ou receber o conteúdo incorreto da instrução que causou a exceção, ou receber o conteúdo de mtc0.

Sabendo disso, o funcionamento interno se da pela ativação do except, que permitirá a escrita nos registradores EPC e WR. Externamente no Datapath, ao ocorrer a exceção, PC receberá

um endereço específico para tratamento de exceções, o qual se encontrará instruções que lidarão com o caso que originou a exceção, que no caso podem ser MFC0, MTC0 e RFE, as duas primeiras instruções irão interagir com o registrador Res, e a instrução RFE ativará o sinal ExcOut, que levará para o endereço que está dentro de EPC (ou seja, redirecionando o fluxo de execução para a instrução seguinte a instrução que ocasionou a exceção).

- **11.0 Datapath**

O Datapath se trata de toda a arquitetura que o grupo montou, seu conjunto é capaz de realizar instruções que executam operações lógicas e/ou aritméticas, deslocar por endereços, saltar para endereços, salvar dados em registradores, salvar dados na memória de dados e lidar com exceções.

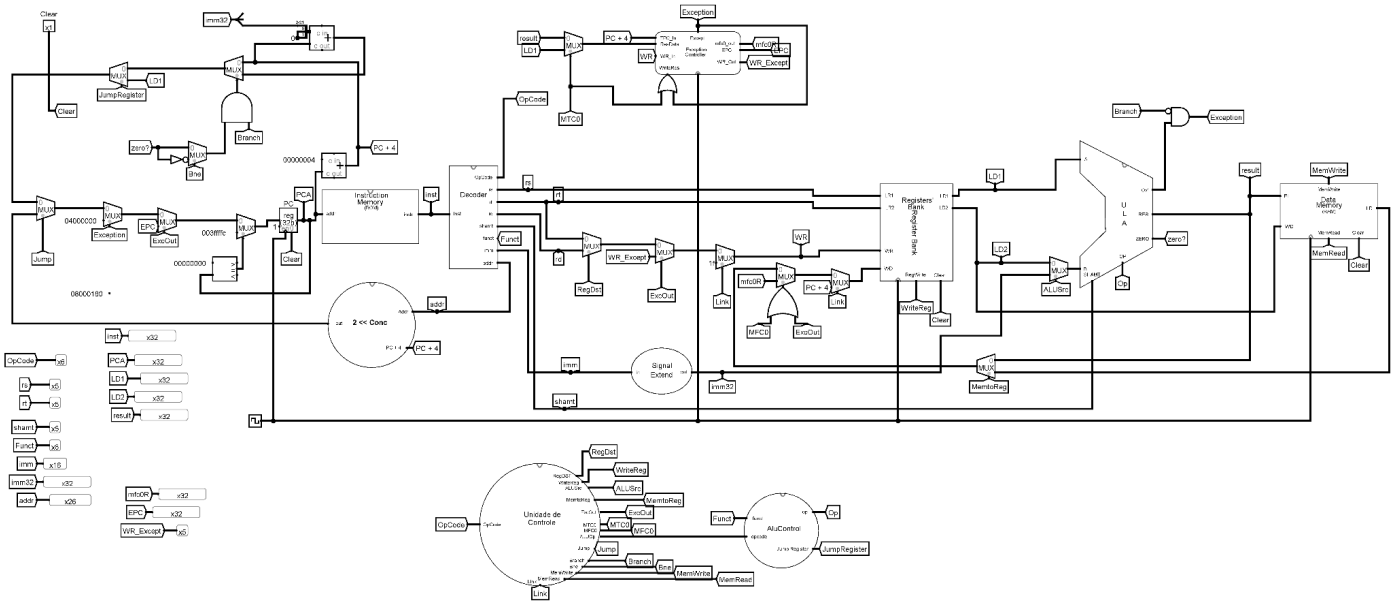


Figura 10.1 - Datapath Completo.

A princípio, nós temos um comparador que recebe a saída de PC, no qual ele vai verificar se é igual a 0, se for ele inicia PC com 0x00400000-4 por meio de um MUX na entrada de PC. Feito isso a cada subida, ele irá verificar uma instrução que estará presente no endereço em PC, você deve carregar ou digitar diretamente as instruções na memória de instruções, tendo em conta que ele vai começar a executar as instruções no endereço 0x00400000, você pode usar diretamente instruções geradas pelo MARS, a princípio é para funcionar corretamente, desde que não sejam usadas instruções que não estejam presentes em nossa arquitetura, logo deve-se tomar cuidado com instruções que geram extensões, como por exemplo passar um imediato para addi

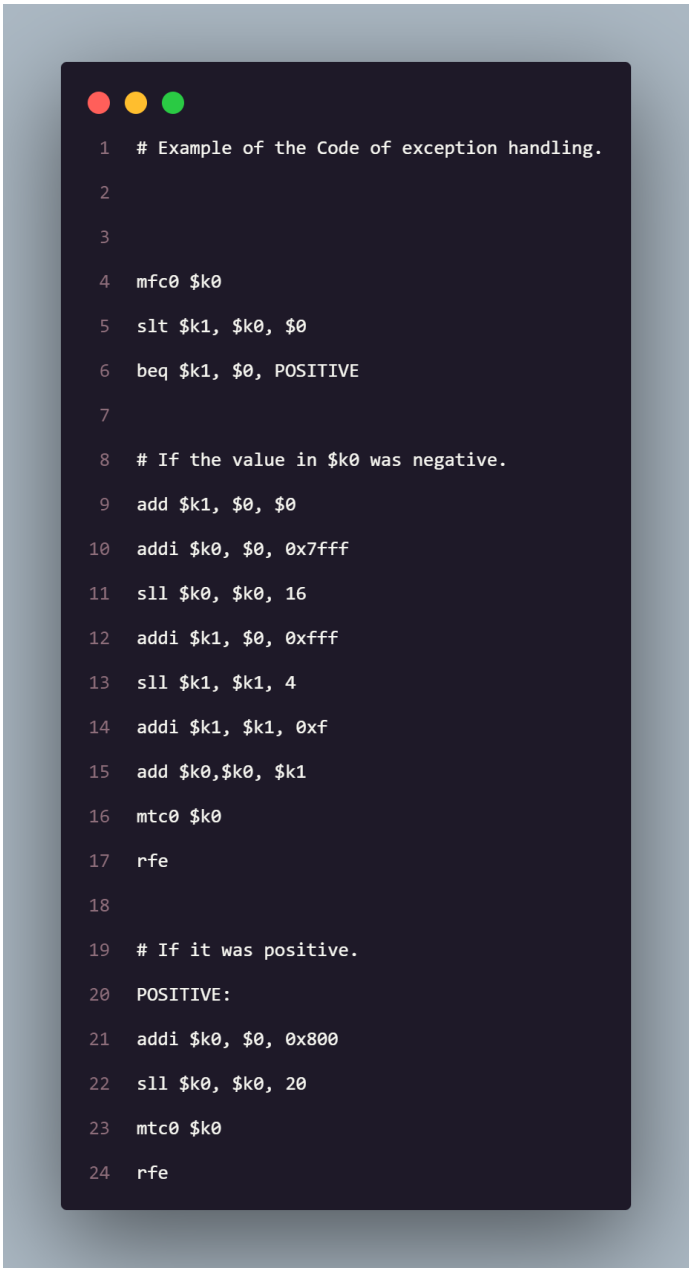
que estoure o limite de 16 bits, o MARS pode chamar instruções como li ou lui, que não estão presentes em nossa arquitetura e iriam gerar conflito. Se você for escrever um código que não seja pelo MARS, para instruções de salto, é importante ter em mente que você deve passar o endereço de destino com shift de 2 para direita, pois ao sair de addr, ele passará pelo componente 2<< conc, que fará um shift de 2 para esquerda, o que elevaria a distância do endereço. Outra coisa para se ter em mente, é que para carregar instruções na memória de instruções, ela deve estar no seguinte formato:

v2.0 raw

1048576*0 2010000a 20110005 00119820 00004020 00004820 0111582a 1160000a 0130602a 20160001 15960005 02739820 200d0050 126d0004 21290001 08100007 22100001 08100005 00132020 00102820 0c10001d 0002a020 20157fff 0015ac00 20160fff 0016b100 22d6000f 02b6a820 22b50001 0810002a 23bdfff8 afb00004 afa80000 00a52820 0085402a 11000002 00a41022 08100026 00851022 8fb00004 8fa80000 23bd0008 03e00008 00004020

sendo “1048567*0” um indicador que as instruções devem ser inseridas em 0x00400000.

a cada ciclo de clock, as instruções serão decodificadas em saídas saídas específicas para o tipo da instrução lida, ela vai seguir para o banco de registradores, depois para a ULA, e pode ou não passar para a memória de dados, em caso de exceção, ele vai ativar a flag em um MUX que está conectado ao PC e vai levar ao endereço 0x3ffffc na memória de instruções, a partir deste endereço, deve-se ter o seguinte conjunto de instruções em assembly:



```
# Example of the Code of exception handling.



mfc0 $k0
slt $k1, $k0, $0
beq $k1, $0, POSITIVE

# If the value in $k0 was negative.
add $k1, $0, $0
addi $k0, $0, 0x7fff
sll $k0, $k0, 16
addi $k1, $0, 0xfff
sll $k1, $k1, 4
addi $k1, $k1, 0xf
add $k0,$k0, $k1
mtc0 $k0
rfe

# If it was positive.
POSITIVE:
addi $k0, $0, 0x800
sll $k0, $k0, 20
mtc0 $k0
rfe
```

Figura 10.2 - Código em Assembly que trata Exceção

As instruções em hexadecimal estão na pasta Instructions com o arquivo nomeado “Exception Handling”.

Algumas coisas que são importantes de referenciar são, temos um MUX na entrada B da que vai permitir que um imediato passe por lá quando AluSrc é ativado, um MUX que conecta rt e rd que saem do decoder, e é rd que sai do MUX quando RegDst é ativado, um outro MUX que direciona o que sai de WR_Out quando ExcOut é ativado, e um outro MUX que direciona para 1f quando Link é ativado, abaixo temos um MUX com uma OR de que vai passar o resultado

que sai da UCE quando ocorre MFC0 ou RFE, que leva a uma mux que passa o valor de PC+4 quando Link é ativado, também temos o circuito a seguir, que trabalha com instruções como beq, bne, J, JAL e casos de exceção:

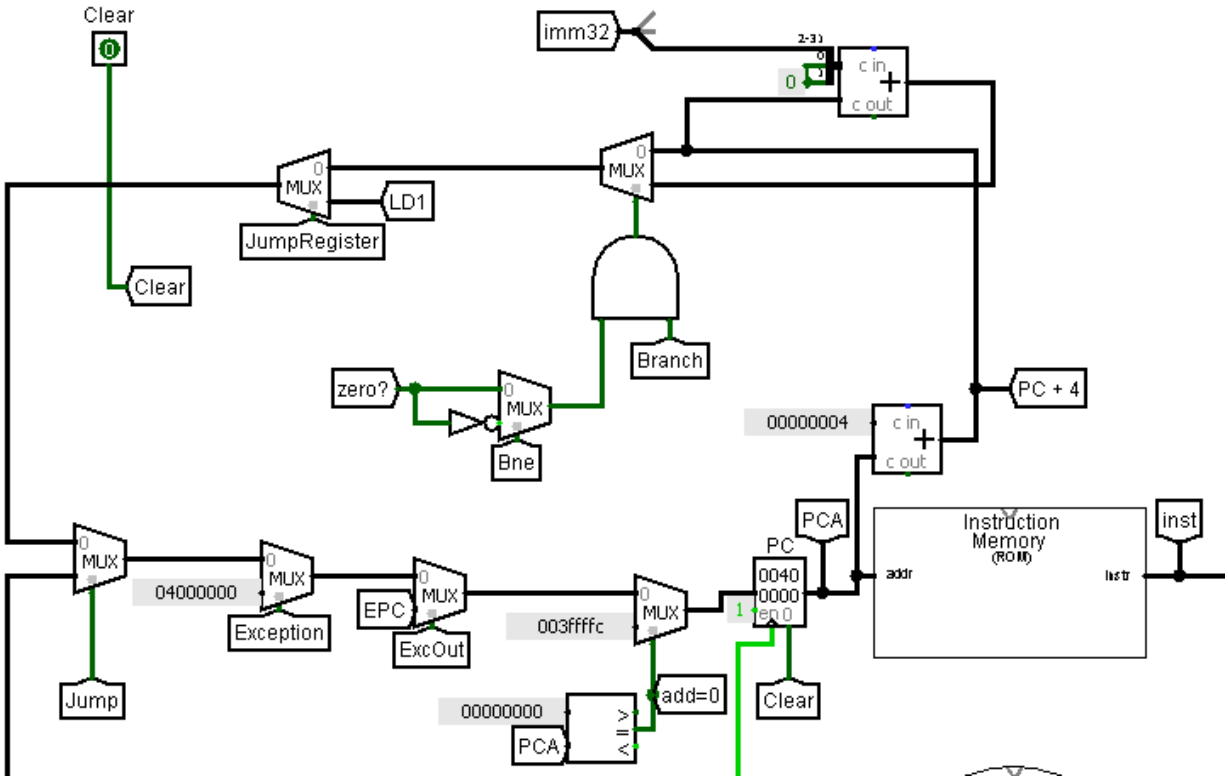


Figura 10.3 - Processos que ocorrem com o PC.

Como código de teste para a entrega deste projeto temos o seguinte código de alto nível:

```cpp
# include <iostream>
# include <limits>

int ASubDobroB(int a, int b){

    b += b;
    if(a > b)
        return a-b;
    else
        return b-a;
}


int main(int* argc, char** argv){

    int a = 10, b = 5;
    int c = a;
    for(int i = 0; i < b; i++){
        for(int j = 0; j < a; j++){
            c += c;
            if(c == 80)
                break;
        }
        if(c == 80)
            break;
    }
    int d = ASubDobroB(c, a);

    int e = std::numeric_limits<int>::max(); // não tratar isso como função, só é um jeito prático de inserir o maior numero inteiro em "e".
    e = e + 1; // Supostamente causa exceção, mas o C++ deve lidar com isso.

    return 0;
}
```

Figura 10.4 - Código em alto nível, pode ser encontrado na pasta High-level Languages com o nome de “Submission code.cpp”

Código este que foi traduzido para o seguinte código em assembly:

MAIN:

addi $s0, $0, 10 # **a=10**

addi $s1, $0, 5 # **b=5**

add $s3, $0, $s1 # **c=a**

add $t0, $zero, $zero **#i=0**

add $t1, $0, $0 **#j=0**

LOOPA:

```
        slt $t3, $t0, $s1
        # se i >= b, pula pra ENDA
        beq $t3, $0, ENDA  #diferente do Jump, beq recebe um imediato no lugar da label
        LOOPB:
                # if a < j, t4 = 1
                slt $t4, $t1, $s0
                addi $s6, $0, 1
                bne $t4, $s6, ENDB
                        add $s3, $s3, $s3
                        addi $t5, $0, 80
                        beq $s3, $t5, ENDA

                addi $t1, $t1, 1
                j LOOPB
                ENDB:
                addi $s0, $s0, 1
        j LOOPA

ENDA:
add $a0, $0, $s3 # c => a
add $a1, $0, $s0 # a => b

JAL ASubDobroB

add $s4, $0, $v0 # d = ASubDobroB(c, a)
addi $s5, $0, 0x7fff
sll $s5, $s5, 16
addi $s6, $0, 0xfff
sll $s6, $s6, 4
addi $s6, $s6, 0xf
add $s5, $s5, $s6
```

```
addi $s5, $s5, 1

j GEXIT



ASubDobroB:
        addi $sp, $sp, -8
        sw $s0, 4($sp)
        sw $t0, 0($sp)
        add $a1, $a1, $a1
        #if a < b
        slt $t0, $a0, $a1
        # if a > b
        beq $t0, $0, ELSE
                sub $v0, $a1, $a0
                j EXIT

        ELSE:
                sub $v0, $a0, $a1
        EXIT:
        lw $s0, 4($sp)
        lw $t0, 0($sp)
        addi, $sp, $sp, 8
        jr $ra

GEXIT:
add $t0, $0, $0
```

Pode ser encontrado na pasta Instructions/ASM/submission code.asm, e a instrução em hexadecimal pode ser encontrado na pasta Instructions/Submission Code Instruction.

***Importante: Para instruções que usam o $sp, é preciso iniciá-lo manualmente no banco de registradores com o valor: 0x7fffffc. Uma alternativa para não ter que fazer isso, seria começar o valor em PC bem antes de 0x00400000, e passar um conjunto de instruções que inicializariam $sp no endereço especificado, onde no final ele teria que fazer um jump para 0x00400000.**

Por exemplo, poderíamos passar o seguinte código para iniciar $sp:

```
addi $t0, $0, 0x07fff
sll $t0, $t0, 12
addi $t0, $t0, 0xffc
add $sp, $0, $t0
j 0x00400000
```

Desta forma inicializaremos $sp com o valor **0x7fffffc**

- **Conclusões**

Aqui já podemos ter uma ideia do Datapath em sua totalidade, podendo observar a entrada de um endereço na memória de instruções, a qual retorna uma instrução específica para o decodificador, que irá verificar pelo OpCode se a instrução segue os formatos R, I ou J, retornando as saídas específicas. Do decodificador, essas informações passam pelo Banco de Registradores um componente fundamental contendo os 32 registradores padrões operáveis da arquitetura, ele fornece duas saídas como entrada para outro componente fundamental que é a ULA, que fará as operações necessárias para a execução das instruções da arquitetura, a ULA fornecerá uma saída que passa pela Memória de dados, seguindo para a entrada de endereço, onde permite que a memória de dados forneça uma saída que se direciona para um *MUX* entre ele e o resultado da ULA que leva para a entrada **WD** no banco de registradores, temos a Unidade de controle que lidará com questões de acesso no DataPath baseado no tipo de instrução e temos também a Unidade Controladora da ULA, que ficará responsável por definir o código de operação que será passado para ULA por meio do FUNC ou do OpCode de 2 bits para instruções

que não são do tipo R. Além disso tudo, graças a Unidade De Controle de Exceções, poderemos lidar com instruções que venham a ocasionar *Overflow* assim tratando ele, e permitindo que se retorne à normalidade no fluxo de execução do Datapath.