
**UNIVERSIDADE FEDERAL DO CARIRI**

**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**PEDRO HENRIQUE BEZERRA SIMEÃO**

**RAISSA KAROLINY DA SILVA RODRIGUES**

**FRANCISCO ANDERSON MACIEL CRUZ**

# IMPLEMENTAÇÃO PROCESSADOR 16 BITS COM PIPELINE

**Arquitetura e organização de computadores**

**JUAZEIRO DO NORTE - CE**

**2023**

**PEDRO HENRIQUE BEZERRA SIMEÃO**
**RAISSA KAROLINY DA SILVA RODRIGUES**
**FRANCISCO ANDERSON MACIEL CRUZ**

# IMPLEMENTAÇÃO PROCESSADOR 16 BITS COM PIPELINE

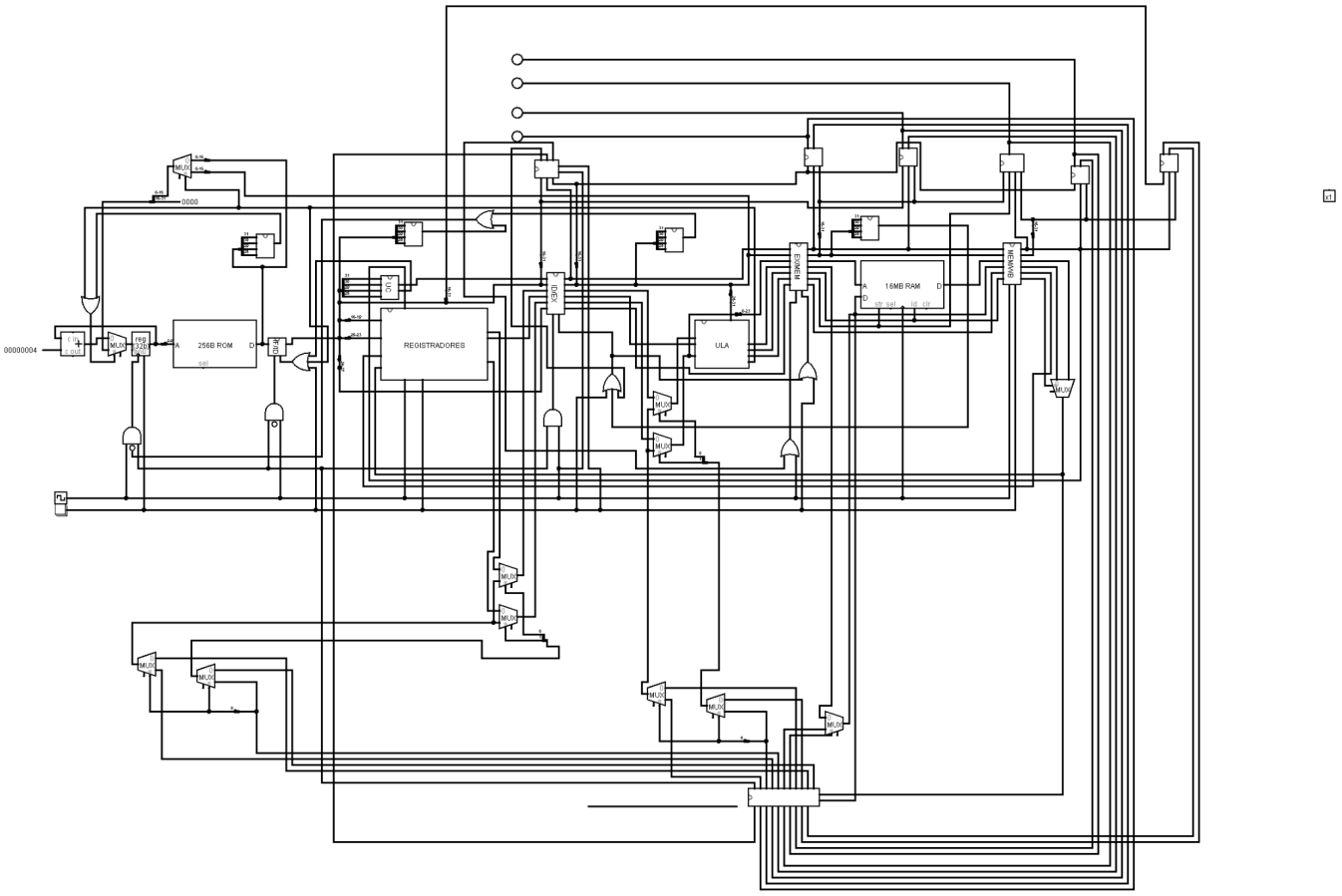
**Arquitetura e organização de computadores**

**Este documento tem a função de ensinar como o nosso processador de 16 bits funciona.**

**JUAZEIRO DO NORTE - CE**
**2024**

## INTRODUÇÃO

Neste relatório, apresentaremos o desenvolvimento de um processador de 16 bits utilizando a técnica de pipeline no software Logisim. O pipeline é uma abordagem crucial para otimizar o desempenho dos processadores, dividindo a execução de instruções em estágios sequenciais. Abordaremos a arquitetura do processador, os desafios enfrentados durante o desenvolvimento e as técnicas empregadas para resolução dos problemas relacionados ao pipeline.

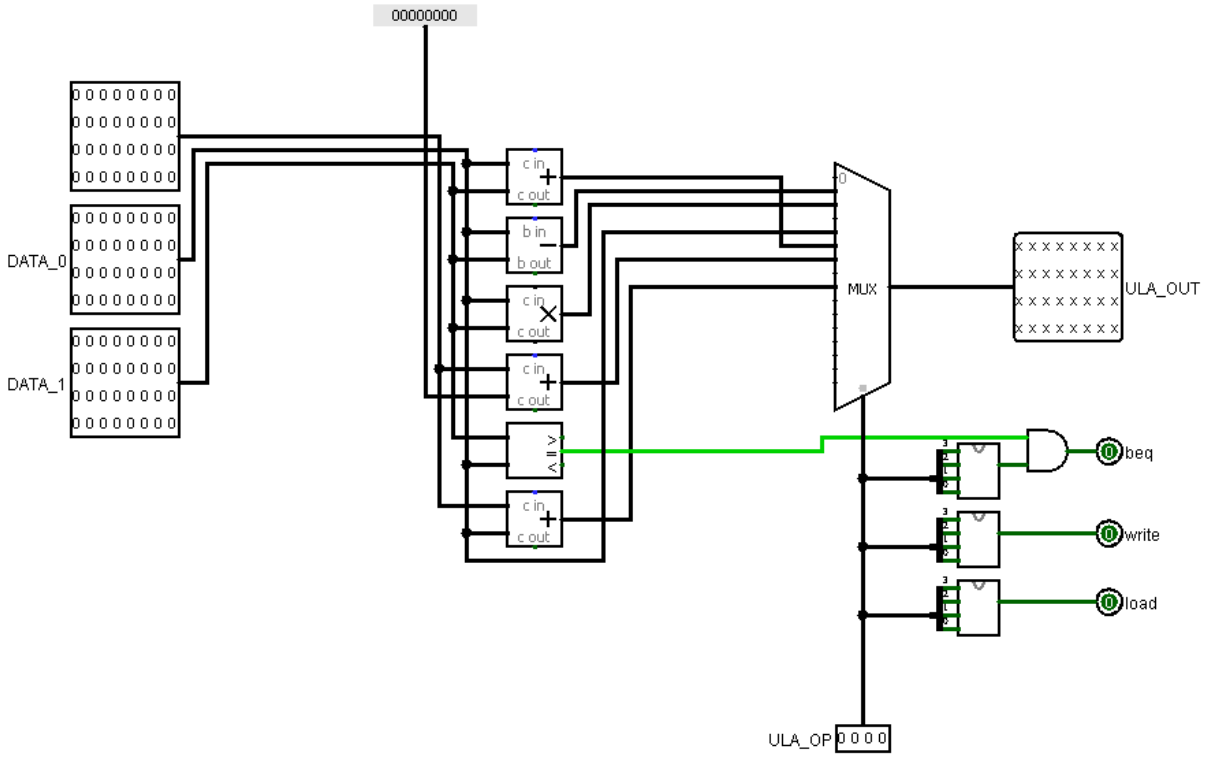



**Processador**

**ULA (unidade lógica e aritmética)**

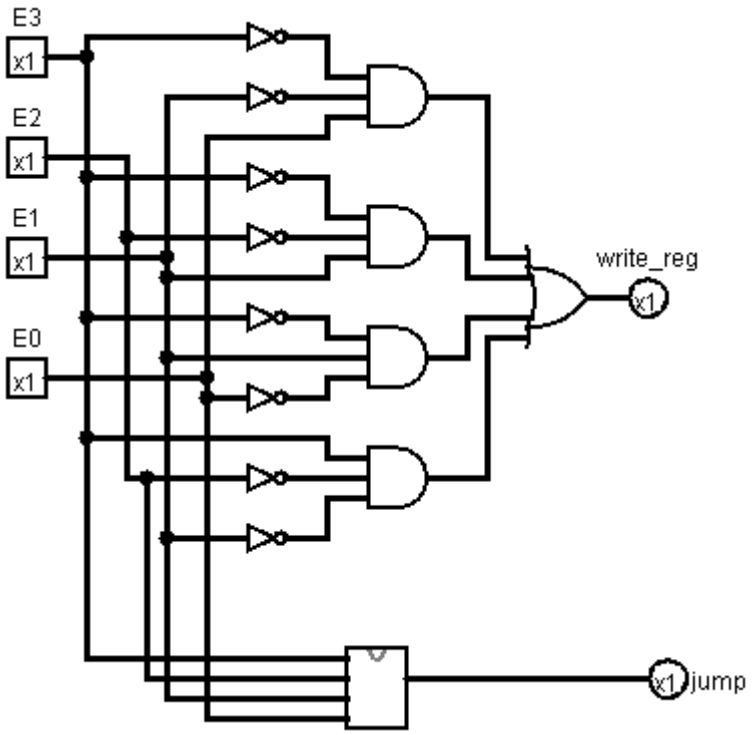
A Unidade Lógica e Aritmética (ULA) é um componente crucial em uma arquitetura de computadores. Sua função principal é realizar operações aritméticas (como adição, subtração, multiplicação) e operações lógicas .

No processador em questão os valores dos registradores recebidos são entradas de 32 bits que passam por operações aritméticas ou lógicas resultando numa saída ULA_OUT e ou sinais resultantes da operação de comparação no caso da instrução beq. A decisão da operação é recebida na ULA_OP que indica qual tipo de operação a ULA deverá passar.



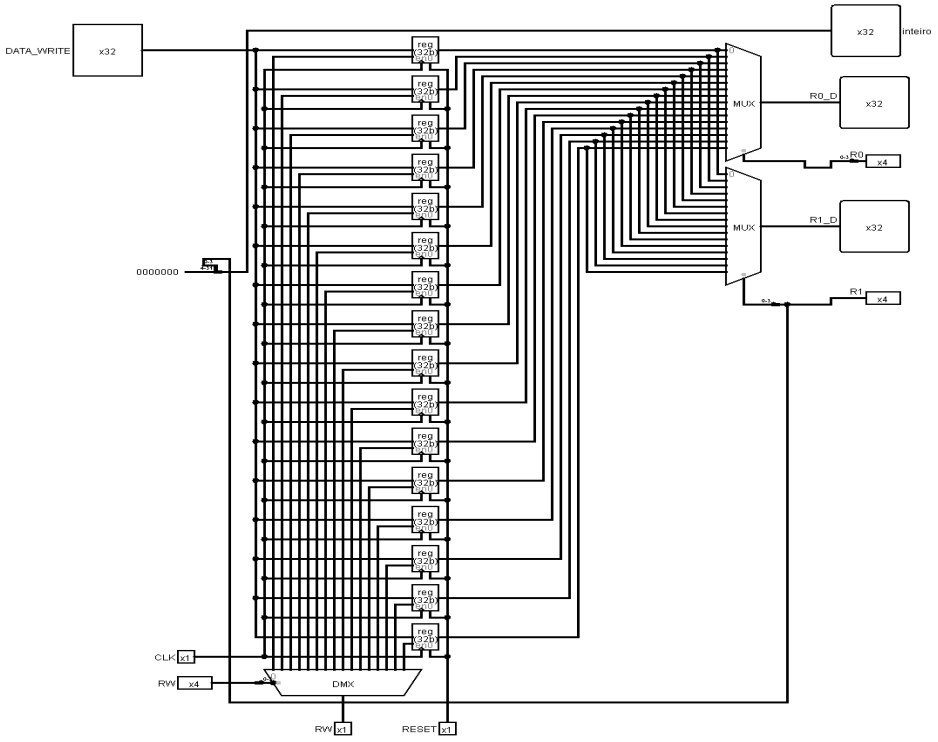

**UC (unidade de controle)**

unidade responsável pela decisão do tipo de operação, recebe os bits 28-31 e retorna se aquela instrução irá escrever um valor ou se é uma instrução do tipo salto.



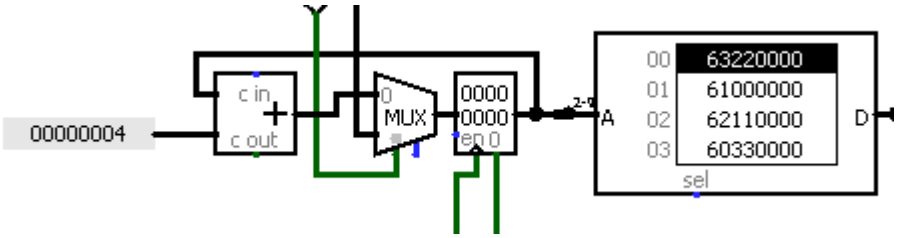

**BANCO DE REGISTRADORES**

Ele consiste em um conjunto de registradores que armazenam temporariamente dados que estão sendo processados ou que serão utilizados em operações futuras. Além da saída de dois registradores, o banco de registradores pode enviar um ‘inteiro’ passado na instrução e possui o input de reset para resetar o banco de registradores.



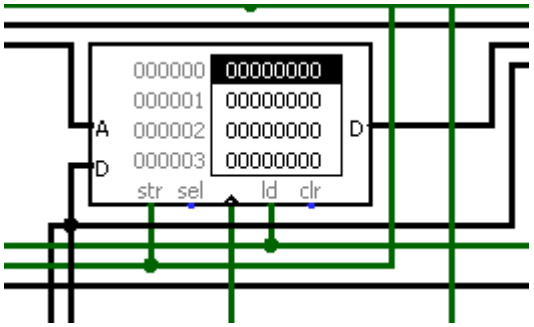
## MEMÓRIA DE INSTRUÇÃO

A memória de instrução é responsável por carregar e selecionar a instrução a ser executada.No fluxo normal as instruções são passadas sequencialmente, ou no caso de uma instrução de salto o endereço da instrução é passado no multiplexador  ocasionando a seleção da instrução passada.




## MEMÓRIA DE DADOS (RAM)

A memória de dados é responsável por armazenar os dados que estão em uso ou que serão processados, ela possui entradas que indicam se algo irá ser escrito nela, como é o caso de algumas instruções usadas no nosso processador, ela também pode retornar valores armazenados para uso.




## IMPLEMENTAÇÃO DO PIPELINE

Os componentes citados anteriormente poderiam ser utilizados num processador monociclo, com a implementação do pipeline foi necessário a criação de estruturas adicionais para manter a integridade e operacionalidade do processador. Com a partição do processador em estágios, é possível realizar de maneira mais eficiente o processamento das instruções. Nosso processador possui os seguintes estágios:  IF/ID - ID/EX - EX/MEM - MEM/WB

IF (Instruction Fetch)Busca das Instruções

ID (Instruction Decode): Decodificação de Instruções e Leitura do banco de registradores

EX (Execution): Execução ou cálculo de endereço

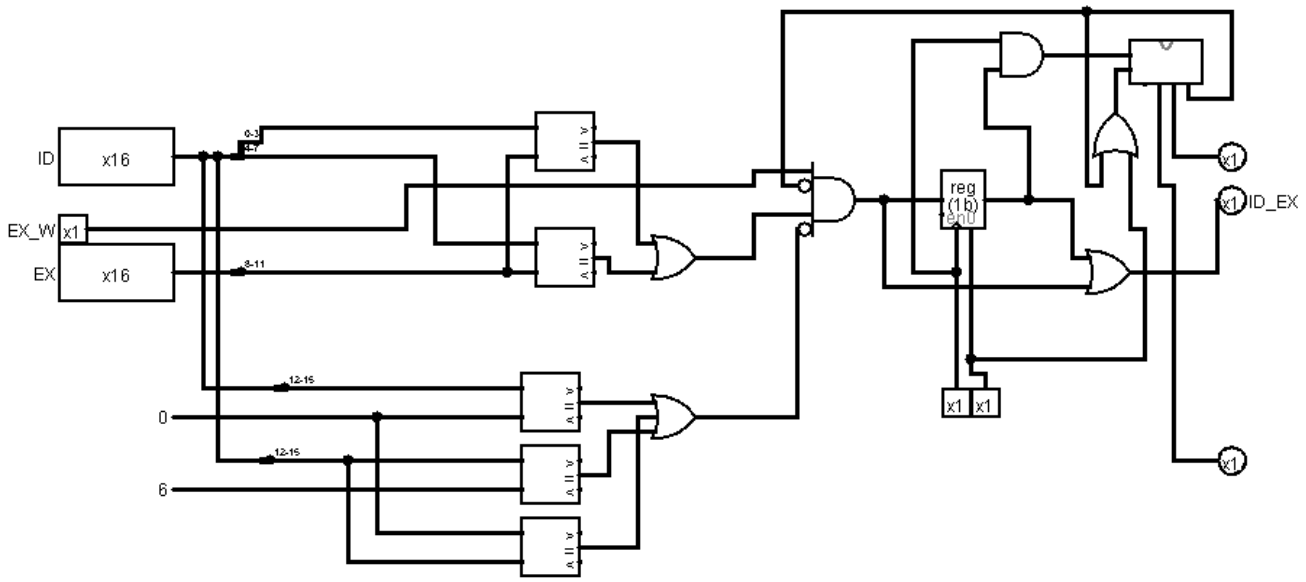
MEM(Memory): Acesso à memória de dados
WB(Write Back): Escrita do resultado

**HAZARDS:**

Para evitarmos problemas de falta de informação entre os setores do nosso processador criamos 6 detectores de hazards a maioria deles tem um funcionamento parecido, ele identifica se a instrução posterior é um 6(carregar inteiro) ou um 0 (instrução bolha) se isso acontece o nosso hazard não dispara, nos outros casos, ele dispara se a instrução da frente escrever em um registrador que a instrução anterior está usando, além disso passamos qual dos registradores está sendo alterado pela instrução que passou.

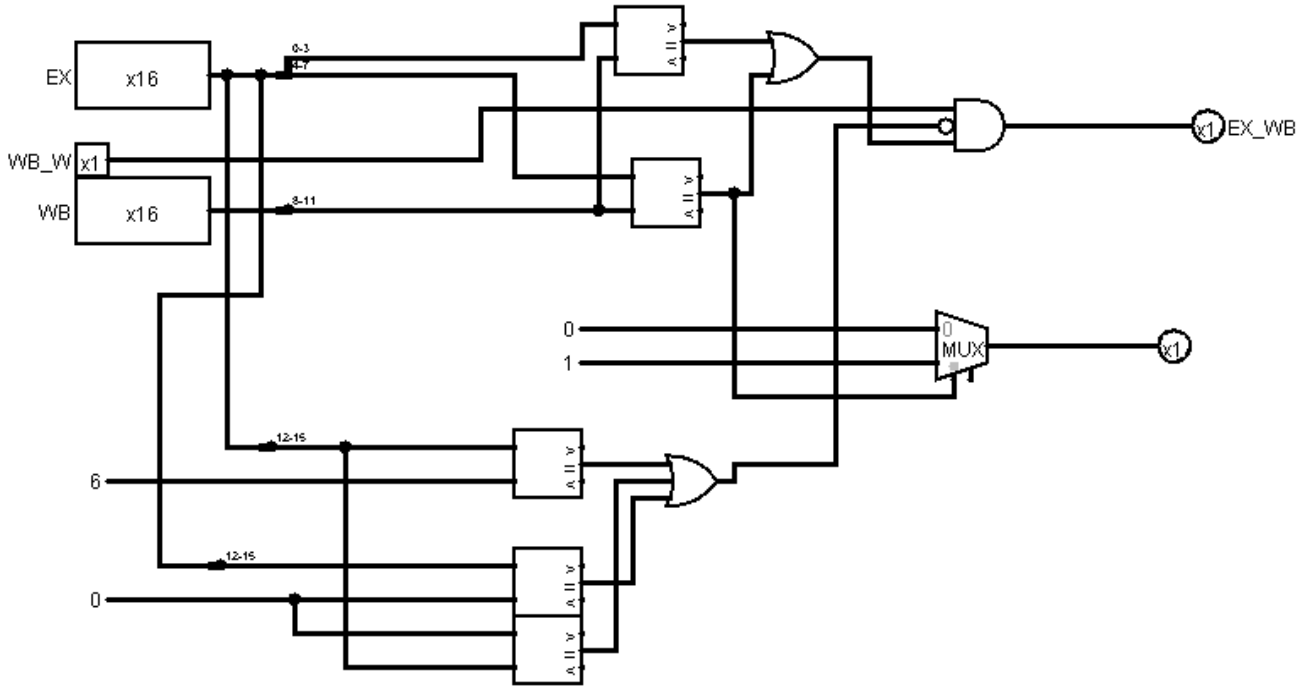
O único diferente é o Hazard_ID_EX, que tem um mecanismo diferente para a criação do bubble.

Hazard_ID_EX:

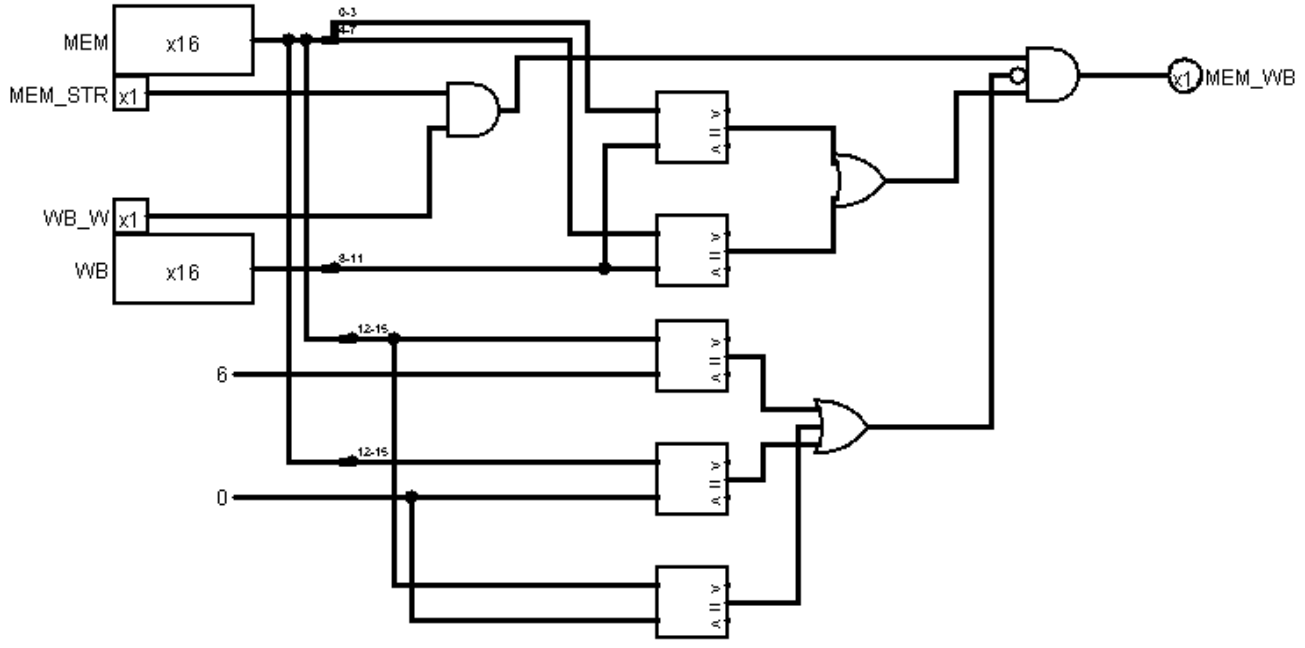


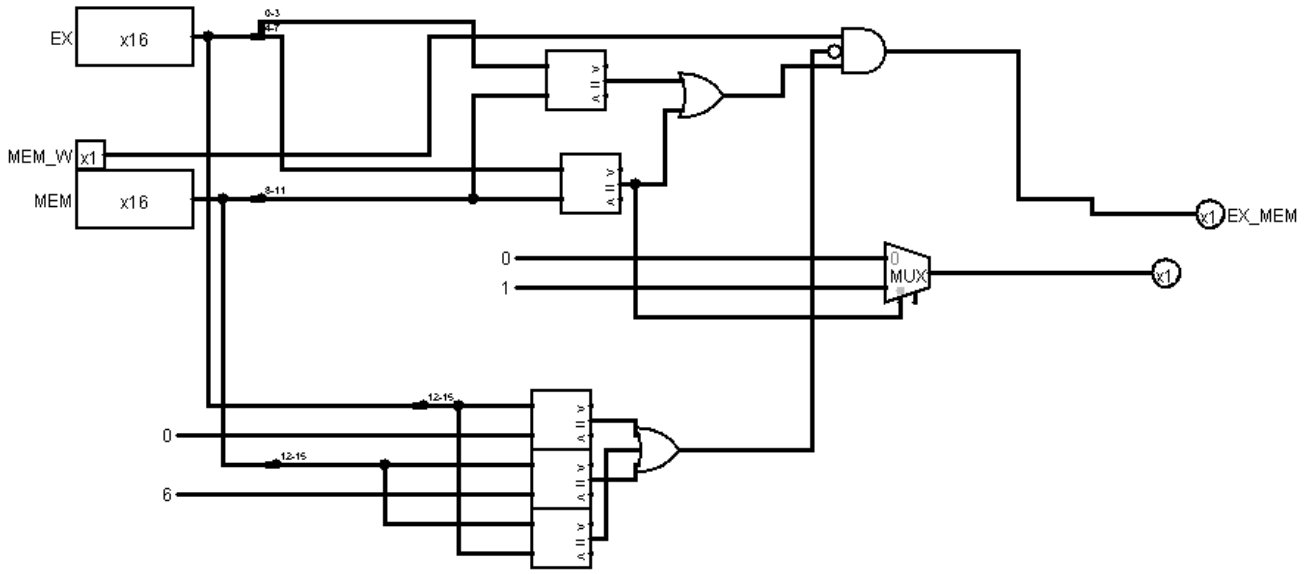

Apenas nesse caso implementamos uma instrução bolha, nas demais (hazard_ex_mem, hazard_ex_mem, hazard_id_mem, hazard_mem_wb, hazard_ex_wb, hazard_wb_id) implementamos o forwarding usando o operation_unit.

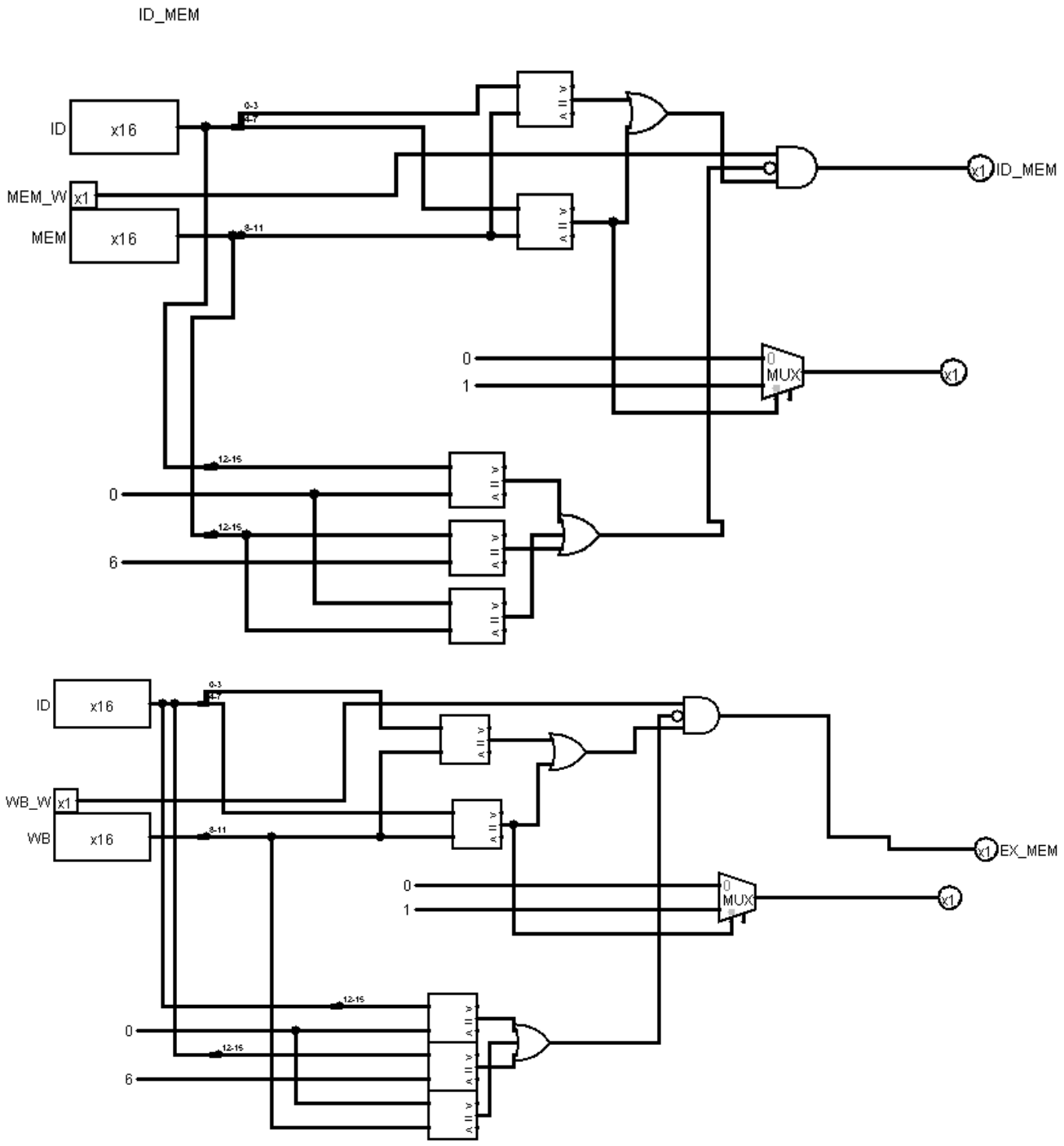
EX_WD


EX
x16
WB_W
x1
WB
x16
0-3
4-7
8-11
12-15
6
0
1
MUX
x1
EX_WB

EX-MEM

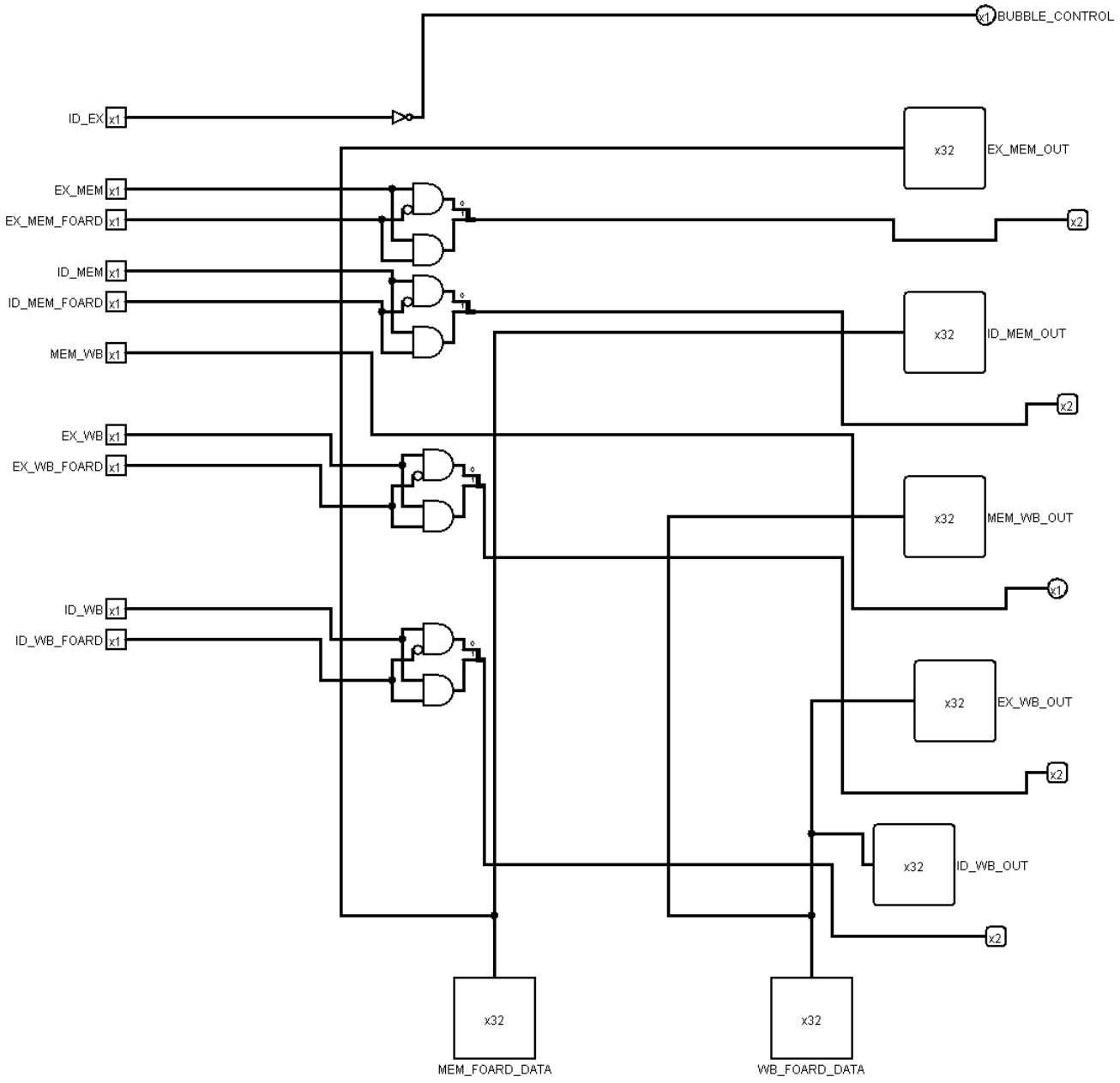

MEM
x16
MEM_STR
x1
WB_W
x1
WB
x16
0-3
4-7
8-11
12-15
6
0
x1
MEM_WB


EX
x16
0-3
MEM_W
x1
MEM
x16
8-11
0
1
0
MUX
1
x1
x1
EX_MEM
12-15
0
12-15
6




**OPERATION UNIT**

Para efetuarmos os forwardings usamos a operation_unit, nela recebemos os dados da seção mem e da seção wb já que os dados dos forwards só podem sair dessas duas áreas, além disso também recebemos se estar acontecendo algum hazard e se sim, ele também passa qual o registrador que recebe o dado do  forward.




Depois de passar pelo operation_unit o dado é passado por multiplexadores para ser encaminhado para o setor de destino.

**FUNCIONAMENTO DO ASSEMBLER**

Para facilitar nosso trabalho na hora de convertermos as instruções em códigos que o logisim entendesse e pudesse realizar o funcionamento.

A linguagem possui as seguintes operações: **soma, dimi, vezes, cg, gm, ci, igual, somaInt e salte.**

**OBS:** registradores possuem {}

**soma:** operação de soma

**Sintaxe: soma {2} {3} {4}**

O registrador {2} recebe o resultado de {3} + {4}.

**dimi:** operação de subtração

**Sintaxe: dimi {4} {3} {4}**

O registrador {4} recebe o resultado de {3} - {4}

**vezes:** operação de multiplicação

**Sintaxe: vezes {2} {3} {4}**

O registrador {2} recebe o resultado de {3} * {4}

**cm:** carregar da memória

**Sintaxe: cg {4} {3}**

O registrador {3} recebe o conteúdo de {4}

**gm:** guarda na memória

**Sintaxe: gm {2} {3}**

O conteúdo do registrador {2} é guardado da memória de {3}

**ci:** coloca um inteiro no registrador

**Sintaxe: ci {2} 4**

O registrador {2} recebe o valor 4.

**igual:** compara o conteúdo de dois registradores

**Sintaxe: igual {2} {3}** 33

Compara o conteúdo de {2} e {3} e em caso válido salta para o endereço 33.

**somaInt:** soma o conteúdo de um registrador com um inteiro

**Sintaxe: somaInt {2} 4 {3}**

O registrador {2} recebe o resultado de 4 + {3}

**salte:** salta para o endereço desejado

**Sintaxe: salte 8**

Salta para o endereço 8