INSTITUTO FEDERAL DO NORTE DE MINAS GERAIS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

BRENO VAMBÁSTER CARDOSO LIMA,
GUSTAVO HENRIQUE ALVES ROCHA,
NATÃ TEIXEIRA SANTOS DE OLIVEIRA

Relatório de Desenvolvimento – ACII: Trabalho I

Arquitetura de Computadores II

MONTES CLAROS – MG
2023

SUMÁRIO

1.INTRODUÇÃO....3
2.DECISÕES DE PROJETO....3
2.1.Processador....3
2.2.Tabelas de Separação das Instruções....4
2.3.Tamanho das *Caches*....6
2.4.Política de Escrita das *Caches*....7
2.5.Considerações Preliminares....7
3.MEMÓRIA PRINCIPAL....10
3.1.Cálculo do Endereço de uma Palavra....10
3.2.Gerenciamento dos Sinais de Escrita....12
3.2.Sinais Enviados Para a *Cache*....14
4.*CACHE* DIRETA....16
4.1.Linha da *Cache*....16
4.2.TAG....18
4.3.Bit de Validade....18
4.4.Falha de *Cache*....19
4.4.1.Verificando Se o Bloco Está na *Cache*....19
4.4.2.Verificando Se a Linha É Válida....20
4.4.3.Sinais *Cache* Miss e *Cache Miss Load*....21
4.4.4.Lidando com as Falhas de *Cache*....21
4.5.*Load* e o Padrão *Handler*....23
4.6.Buffers....24
4.7.Store....25
5.FIM DO PROGRAMA....28
6.VALIDANDO A *CACHE* DIRETA....29
7.*CACHE* ASSOCIATIVA POR CONJUNTO....29
7.1.Reaproveitando o Circuito....30
7.2.Criando os Conjuntos....30
7.3.*LRU* e a Falha de *Cache*....30
7.4. O Apogeu do Desenvolvimento....32
8.REFERÊNCIAS....34

1.INTRODUÇÃO

Conforme os avanços dos sistemas de computação, os processadores sofreram um grande aumento em sua velocidade de operação. Simultaneamente, as memórias também tiveram um aumento de suas velocidades, não sendo suficiente para acompanhar os processadores.

Nesse sentido, visando minimizar a disparidade de velocidade entre o processador e a memória principal, foram criadas memórias individuais para o processador, as quais receberam o nome de "*cache"*.

Este trabalho propõe-se a implementar duas memórias *cache*, cujas técnicas de mapeamento são a direta e a associativa por conjunto, respectivamente.

2.DECISÕES DE PROJETO

2.1.Processador

Iremos utilizar o processador construído na etapa final da disciplina de Arquitetura de Computadores I. No entanto, ele possuía algumas simplificações de escopo, sendo uma delas, por exemplo, o registrador de destino do resultado de alguma operação lógica ou aritmética sempre ser o registrador acumulador. Para possibilitar o seu uso, realizamos algumas alterações.

O banco de registradores passará a possuir 8 registradores e o registrador RB, totalizando 9 registradores. No conjunto de instruções inicial, era possível endereçar apenas 8 registradores. Para possibilitar endereçar os 9 registradores, retiramos as instruções "ori" e "mv" do conjunto, o que permitirá a retirada de 1 *bit* do opcode.

Nas instruções do tipo R e do tipo I, alocamos esse *bit* no campo do primeiro registrador fonte. Nas instruções do tipo J, esse *bit* foi alocado no campo de Endereço.

Foi criado um novo tipo de instrução, o tipo D. Ele foi criado pois o campo do imediato das instruções de *branch*, para possibilitar a execução do programa disponibilizado no trabalho, necessita de 7 *bits*. Portanto, o tamanho dos campos não é mais o mesmo do tipo I, necessitando assim da separação de tais instruções em um novo tipo.

Além disso, a segunda entrada da ULA agora pode ser: a segunda saída do banco de registradores, o imediato das instruções do tipo I ou o imediato das instruções do tipo D, para isso, foram adicionados multiplexadores.

No banco de registradores, na primeira entrada, em virtude da disposição dos campos serem diferentes nos tipos R, I e D, foram adicionados multiplexadores para seleção do campo correspondente na instrução atual. O mesmo ocorreu na entrada do registrador de escrita. Internamente, o acumulador foi substituído por um registrador comum.

Então, agora o registrador de destino não é mais sempre o acumulador, ele é informado na instrução e as instruções de *branch* utilizam um operando imediato para comparação em vez do acumulador.

2.2.Tabelas de Separação das Instruções

Tabela 1 – Instruções do Tipo R

Instrução	**Tipo**	**Opcode**		**Reg. Destino**	**Reg. Fonte 1**	**Reg. Fonte 2**	**Funct**
	R	3 *bits*		3 *bits*	4 *bits*	3 *bits*	3 *bits*
		Binário	**Decimal**				
add	R	000	0				000
sub	R	000	0				001
and	R	000	0				010
or	R	000	0				011
nand	R	000	0				100
nor	R	000	0				101
sll	R	000	0				110
slr	R	000	0				111

Tabela 2 – Instruções do Tipo I

Instrução	**Tipo**	**Opcode**		**Reg. Destino**	**Reg. Fonte 1**	**Constante**
	I	3 *bits*		3 *bits*	4 *bits*	6 *bits*
		Binário	**Decimal**			
addi	I	001	1			
andi	I	010	2			
lw	I	101	5			
sw	I	110	6			

Tabela 2 – Instruções do Tipo D

Instrução	**Tipo**	**Opcode**		**Reg. Fonte 1**	**Reg. Fonte 2**	**Imediato**
	I	3 *bits*		3 *bits*	3 *bits*	7 *bits*
		Binário	**Decimal**			
beq	I	011	3			
bne	I	100	4			

Tabela 3 – Instruções do Tipo J

Instrução	**Tipo**	**Opcode**		**Endereço**
	J	3 *bits*		13 *bits*
		Binário	**Decimal**	
j Label	J	111	7	

Tabela 4 – Registradores Visíveis ao Programador

Código	Registrador
000	$r1
001	$r2
010	$r3
011	$r4
100	$r5
101	$r6
110	$r7
111	$r8
1000	$rb
1001	$r0

2.3.Tamanho das *Caches*

Para a *cache* que utiliza o mapeamento direto, optamos por utilizar 8 linhas, cada uma composta de 4 palavras de 16 *bits* cada.

Já para a *cache* que utiliza o mapeamento associativo por conjunto, optamos por uma implementação de 4 conjuntos, os quais possuindo 2 linhas, cada uma composta por 4 palavras de 16 *bits*.

Logo, o tamanho de ambas as *caches* é de 512 *bits* ou 64 bytes.

2.4.Política de Escrita das *Caches*

A política de escrita adotada para ambas as *caches* foi a de escrita direta (*write-through*). Nesse sentido, as operações de escrita são realizadas na memória principal e na *cache*, de modo que ambas sempre estão atualizadas.

2.5.Considerações Preliminares

Ao longo do desenvolvimento da *cache*, nos deparamos com alguns obstáculos oriundos do Logisim. O primeiro deles foi o fato de o Logisim iniciar com o *clock* na baixa. Nossos componentes, no entanto, realizam a leitura na alta e a escrita na baixa. Assim, a primeira instrução iniciaria já na baixa, não sendo realizada a sua etapa de leitura. Logo, não haveria nada no barramento para ser escrito.

Há várias formas de solucionar esse problema, sendo elas o uso de uma instrução *dummy* como primeira instrução, mas isso teria de ser feito sempre ao adicionar um novo programa nas submemórias; negar o *clock*, o que, de certo modo, pode tornar a execução do circuito pouco intuitiva; pulsar o *clock* uma vez para que ele fique na alta e, após isso, resetar todas as memórias e afins.

Todas essas abordagens potencialmente podem funcionar. Entretanto, optamos por automatizar essa parte, ou seja, fazer com que o próprio circuito lide com essa conjuntura inicial do Logisim.

Tal automação foi feita simplesmente verificando se o circuito está na execução da primeira instrução. Caso sim, como o *clock* já inicia na baixa, não podemos executar a escrita, pois a leitura ainda não ocorreu. Então, invertemos o *clock* no primeiro ciclo, fazendo com que o circuito inicie devidamente na leitura. Ao ocorrer a transição para o segundo ciclo, a escrita irá acontecer e deixamos de inverter o clock. Basicamente, empurramos apenas a primeira instrução meio ciclo para frente. O “fluxo” do *clock* se normaliza a partir da segunda instrução. Assim, em vários pontos do circuito há componentes relacionados ao *clock* cujo objetivo é possibilitar essa automação.

Imagem 1 - Gerenciador Clock Inicial (Parte Interna)

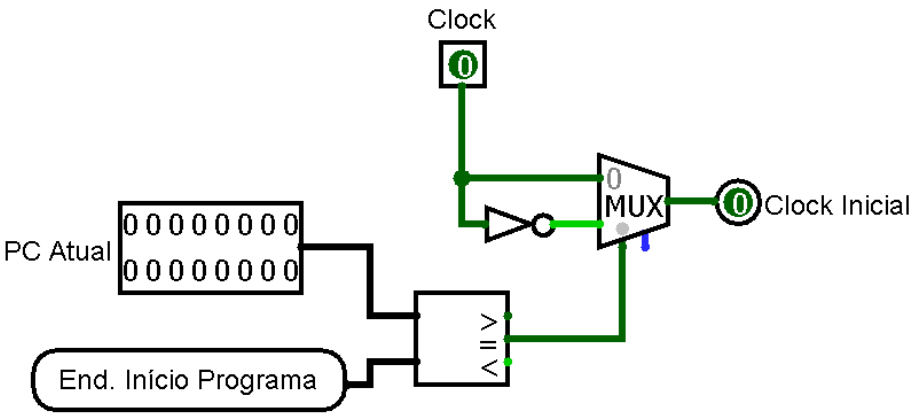


Imagem 2 - Gerenciador Clock Inicial (Parte Externa)

Além disso, enfrentamos dois comportamentos anômalos por parte do Logisim conforme o circuito atingia um tamanho moderado. Notamos que alguns registradores estavam salvando valores inconsistentes no respectivo momento, apesar do dado presente no barramento ser o dado correto.

Após vários dias de investigação e leitura da documentação, descobrimos que o problema era proveniente do Logisim. Um registrador salva um dado quando a sua entrada de *clock* muda de baixo para alto. No entanto, o *clock* estava chegando mais rápido ao registrador que o novo dado no barramento de entrada do registrador. Acreditamos que isso acontecia devido ao fato de todo componente do Logisim ter um *delay* associado. Ao contrário do *clock*, o dado passa por vários componentes até chegar ao registrador, assim, ele estava chegando depois do *clock*.

Então, tornou-se necessária uma forma de atrasar a chegada do *clock* aos registradores que apresentavam essa inconsistência. No entanto, não existe um componente nativo no Logisim que realiza essa ação, tivemos então que criá-lo, o chamamos de "*Delayer*".

Se utilizarmos duas portas NOT, o sinal que entrou na primeira será o mesmo na saída da segunda. Estamos nos apoiando no fato de que cada componente possui um *delay* associado. Em nosso circuito, o *delay* de duas portas NOT não foi suficiente. Após vários testes, o número mínimo de portas NOT que soluciona o problema foi de 12 portas. Com essa estratégia, foi possível sincronizar o *clock* com a chegada do dado no barramento dos registradores.

Imagem 3 - Delayer (Parte Interna)

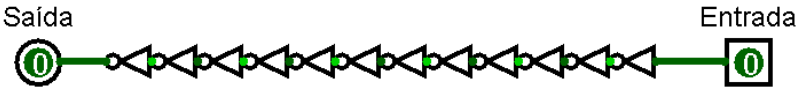


Imagem 3 - Delayer (Parte Externa)

O segundo problema foi o fato de que, em certos momentos durante a transição entre os ciclos de *clock,* ruídos de sinal estavam passando pelos barramentos, ativando sinais em momentos indevidos, resultando na ocorrência de operações indesejadas. Para solucionar esse problema, adicionamos o *delayer* juntamente com portas lógicas para que, ainda que ruídos acontecessem, não fosse realizada nenhuma operação indesejada.

Portanto, tais adversidades foram as motivações da criação de certos *buffers* e componentes não relacionados a um projeto de *cache* comum. O que acabou tornando o circuito ligeiramente mais complexo.

3.MEMÓRIA PRINCIPAL

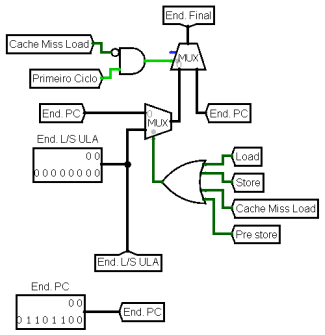
Após a adaptação do processador, partimos para a memória principal. Tendo em vista que o componente “Memória RAM” do Logisim possui apenas um barramento de saída, e que a comunicação entre a memória principal e a *cache* é por blocos de quatro palavras, a criação de uma memória principal que possibilitasse tal comunicação foi necessária.

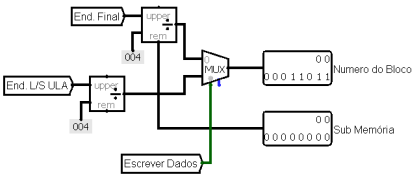
À luz das características supracitadas, combinamos quatro componentes “Memória RAM” do Logisim visando permitir a comunicação via bloco entre memória principal e *cache.*

3.1.Cálculo do Endereço de uma Palavra

Tal configuração fez necessário um mecanismo para calcular a submemória e em qual de suas posições uma palavra ou dado se encontra. Tal mecanismo é feito utilizando o resto e o chão do quociente resultantes da divisão entre o endereço desejado e o número 4. O resto da divisão nos dá a submemória. O chão (⌊ ⌋) do quociente nos dá a posição da palavra ou dado naquela submemória. Exemplo: o endereço atual é 35, o resto da divisão de 35 por 4 é 3; o chão do quociente da divisão entre 35 e 4 é 8. Portanto, a palavra cujo endereço é 35 está na submemória 3 e em seu endereço 8.

Imagem 4 - Calcular Endereço (Parte Interna)





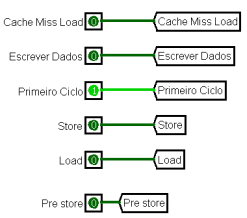
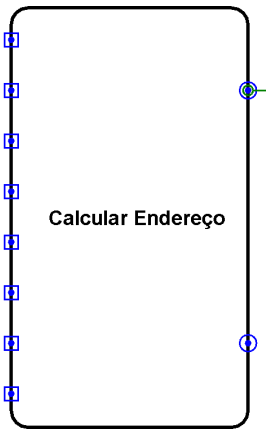
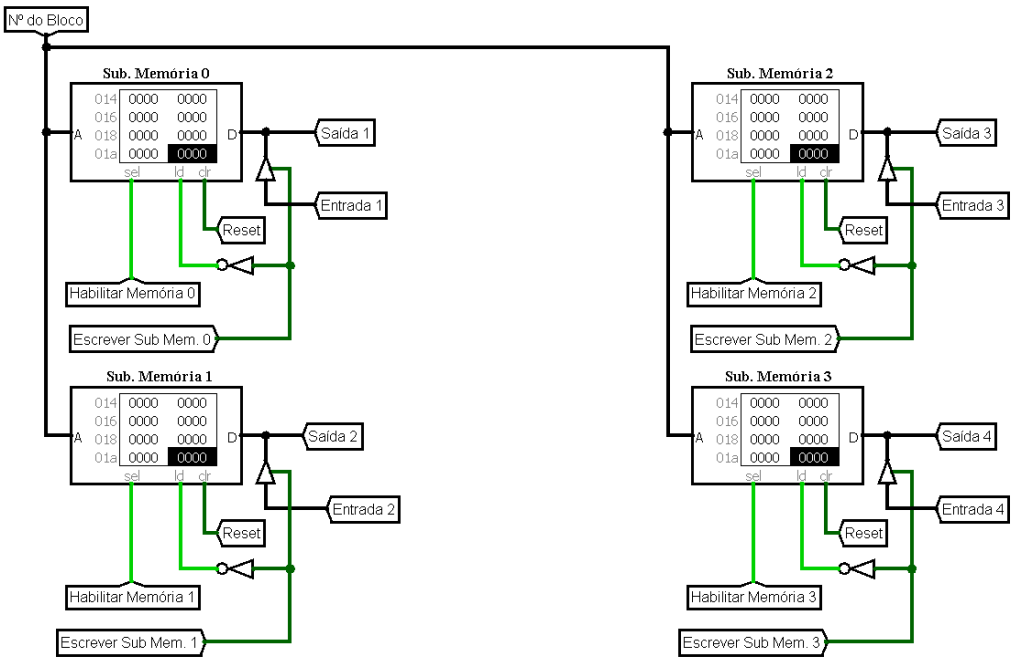


Imagem 4 - Calcular Endereço (Parte Externa)



Como consequência de tal mecanismo, cada palavra de um bloco está dividida entre as submemórias. Por exemplo, as 4 palavras do primeiro bloco estão distribuídas nas posições zero das submemórias. Dessa forma, como o barramento de endereço é compartilhado entre as quatro submemórias, quando o endereço de uma palavra em uma das submemórias é acessado, o mesmo endereço é selecionado nas outras três submemórias, pois as palavras de um bloco estão distribuídas nas mesmas posições entre as submemórias. Possibilitando assim a leitura de 4 palavras simultaneamente, o que configura um bloco.

Imagem 5 - Submemórias com o Barramento de Endereço Compartilhado



3.2.Gerenciamento dos Sinais de Escrita

A escrita na memória principal acontece em apenas uma ocasião, no *store*. Antes de descrever como o *store* acontece, precisamos pontuar alguns problemas que tivemos durante a implementação.

Durante a escrita, apesar do mecanismo que encontra exatamente em qual submemória e em qual de suas posições o endereço desejado está, alguma outra submemória também era ativada. Outro caso era que a escrita acontecia em mais de um endereço na mesma submemória, sobrescrevendo posições indevidas.

Durante a criação da memória principal, ainda não havíamos encontrado os problemas mencionados na seção "**2.5. Considerações Preliminares**". Nesse sentido, o problema na escrita que estávamos tendo era semelhante aos citados em tal seção, era relacionado com o sinal de escrita de cada submemória.

Por alguma razão, ruídos estavam passando nos barramentos das outras memórias, em vez de apenas na memória alvo. A solução para contornar ruídos é forçar um *delay* no sinal problemático. Como ainda não havíamos criado o *delayer*, a solução encontrada foi a criação de um *buffer* dos sinais de escrita, cujo objetivo, de certa forma, é isolar cada sinal em registradores, visando garantir que o sinal de apenas uma das submemórias seja ativado.

Imagem 6 - *Buffer* de Sinais de Escrita (Parte Interna)

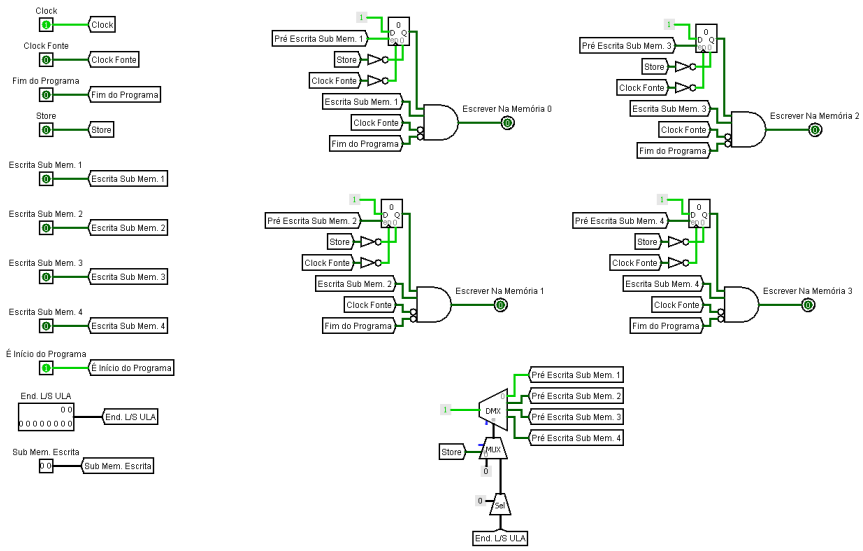
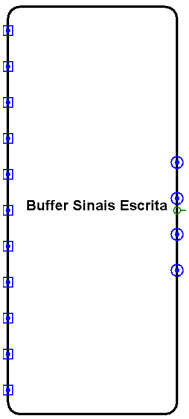


Imagem 7 - *Buffer* de Sinais de Escrita (Parte Externa)



Além disso, outros componentes foram criados visando individualizar os sinais de ativação de cada uma das submemórias. A soma dessas duas soluções mitigou tal adversidade.

Imagem 8 - Direcionar Dados Entrada Memória Principal (Parte Interna)

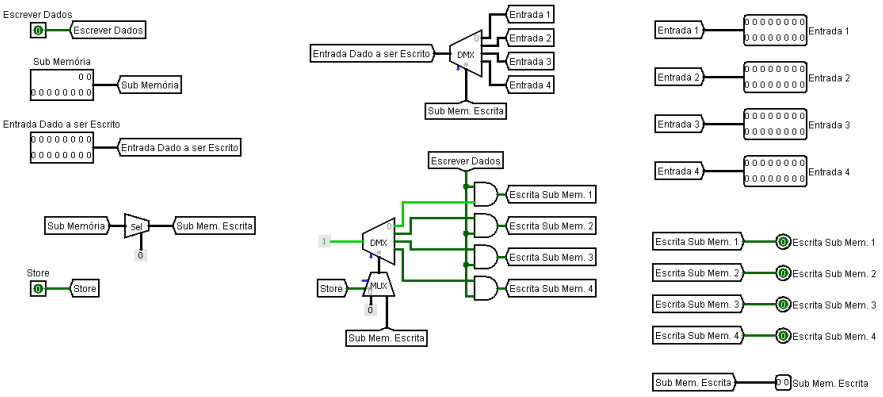


Imagem 9 - Direcionar Dados Entrada Memória Principal (Parte Externa)

Imagem 10 - Habilitar Submemória (Parte Interna)

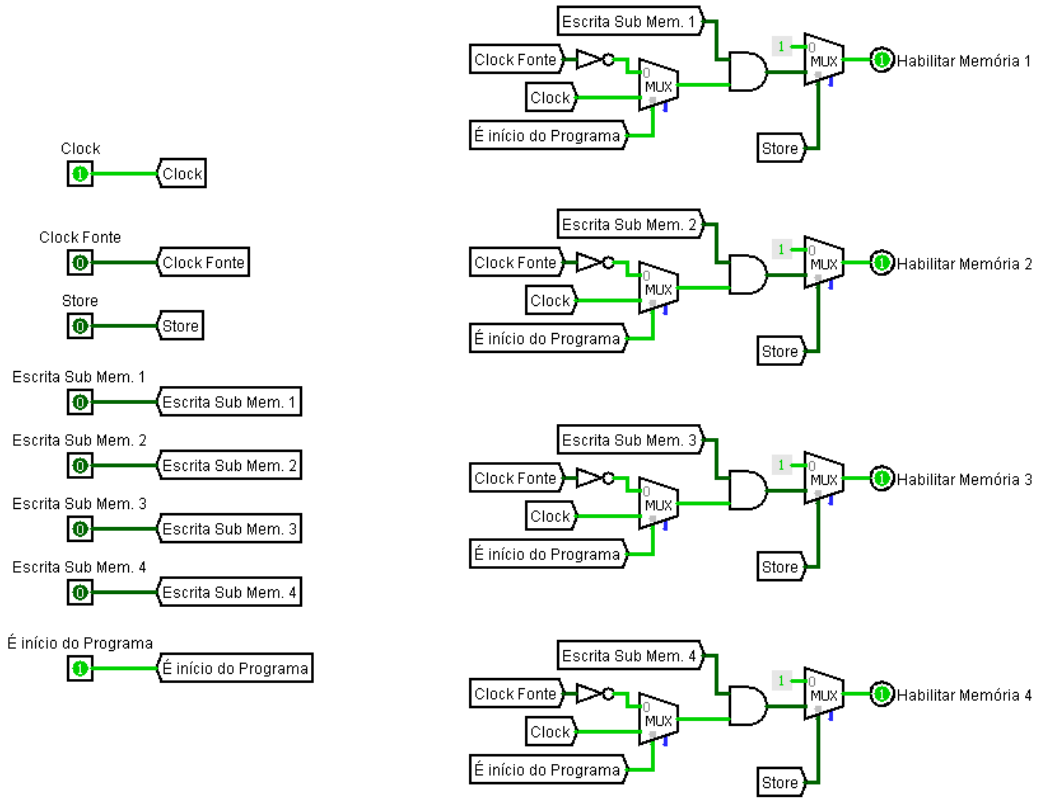
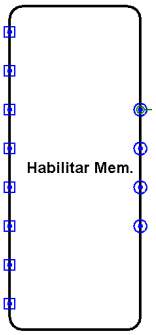


Imagem 11 - Habilitar Submemória (Parte Interna)



O *store* será detalhado na seção sobre a memória *cache*.

3.2.Sinais Enviados Para a *Cache*

A memória principal envia dois sinais para a *cache*: “Escrever na *Cache*” e “Escrever Bloco *Load*”.

O sinal “Escrever na *Cache*”, naturalmente, sinaliza que deverá ocorrer a escrita na *cache*. A escrita ocorre quando é um *store*, quando ocorre *cache miss* e quando estamos no primeiro ciclo, pois o programa inicia com a *cache* vazia. Nesse sentido, nem mesmo a primeira instrução está na *cache*.

O sinal “Escrever Bloco *Load*” é ativado apenas caso a instrução atual seja um *load,* e o dado do *load* não esteja na *cache*.

Ambos os sinais informam a *cache* que a escrita do bloco que está chegando no barramento vindo da memória principal deve ser realizada.

Imagem 12 - Sinal Escrever na *Cache*

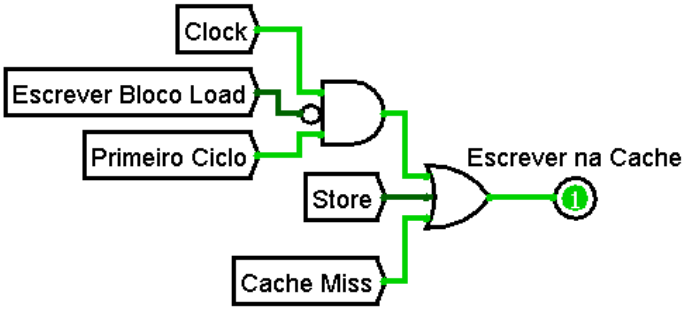


Imagem 13 - Sinal Escrever Bloco *Load*

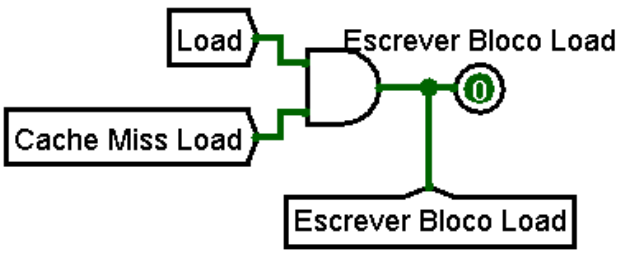


Imagem 14 - Memória Principal (Parte Interna)

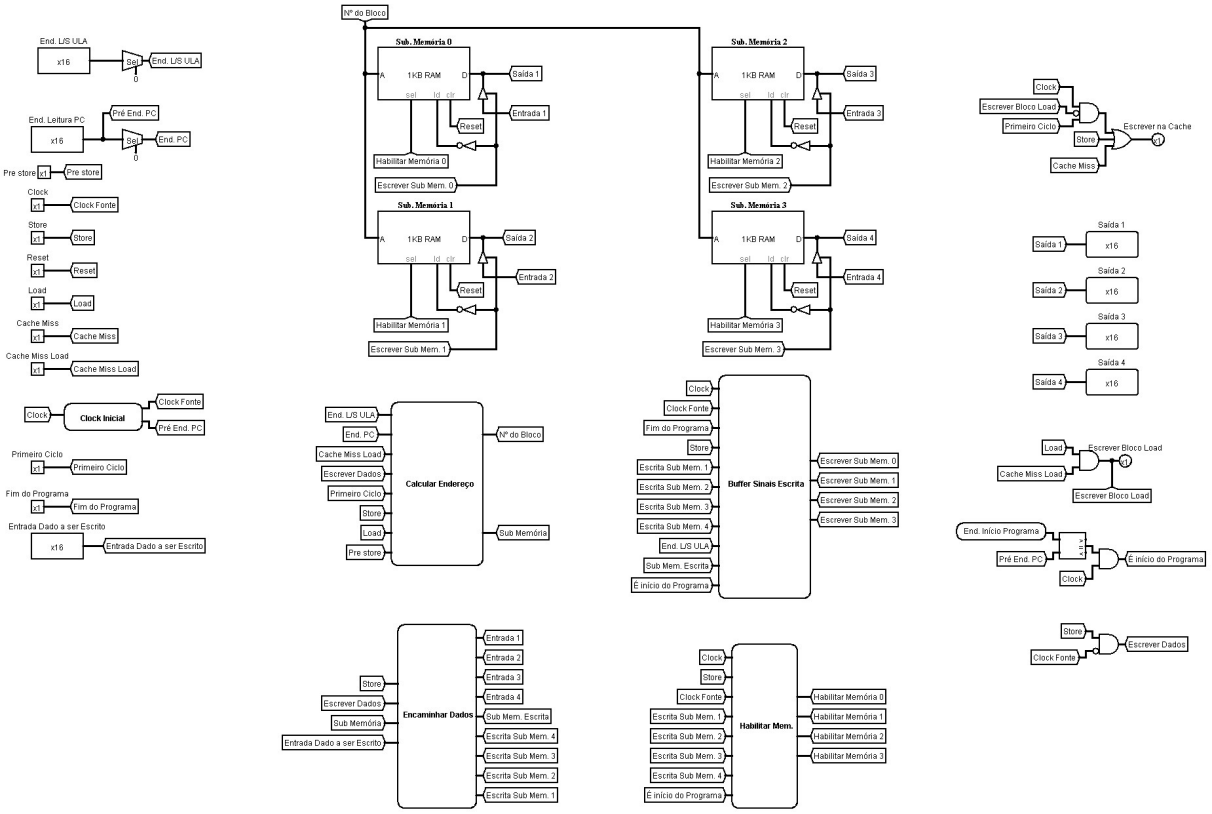
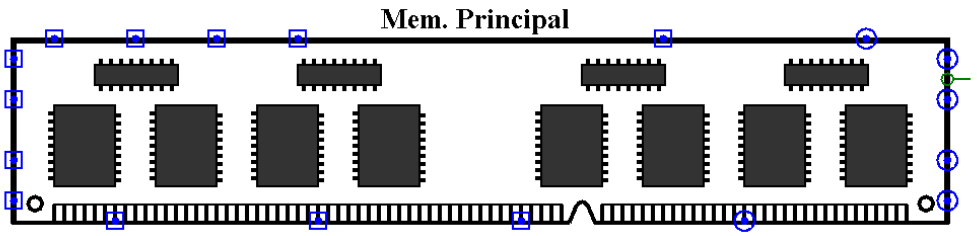


Imagem 15 - Memória Principal (Parte Externa)



4.*CACHE* DIRETA

4.1.Linha da *Cache*

O primeiro passo da implementação da *cache* foi a criação da linha. A linha possui quatro barramentos de entrada, um para cada palavra do bloco. Para armazená-las, utilizamos quatro registradores. A escrita na linha ocorre na baixa do *clock*.

Tendo em vista que a *cache* é unificada, a possibilidade de um bloco, contendo instruções e dados, ser escrito na linha e que o processador obtém a instrução atual da *cache*, caso a instrução atual seja um *load*, não seria possível a obtenção do dado, pois o barramento de saída já estaria ocupado com a instrução atual.

Por conseguinte, foi necessária a adição de um novo barramento de saída na linha, para possibilitar a execução de uma instrução *load.* Tal duplicação de recursos não aconteceria se a *cache* fosse separada.

Imagem 16 - Linha da *Cache* (Parte Interna)

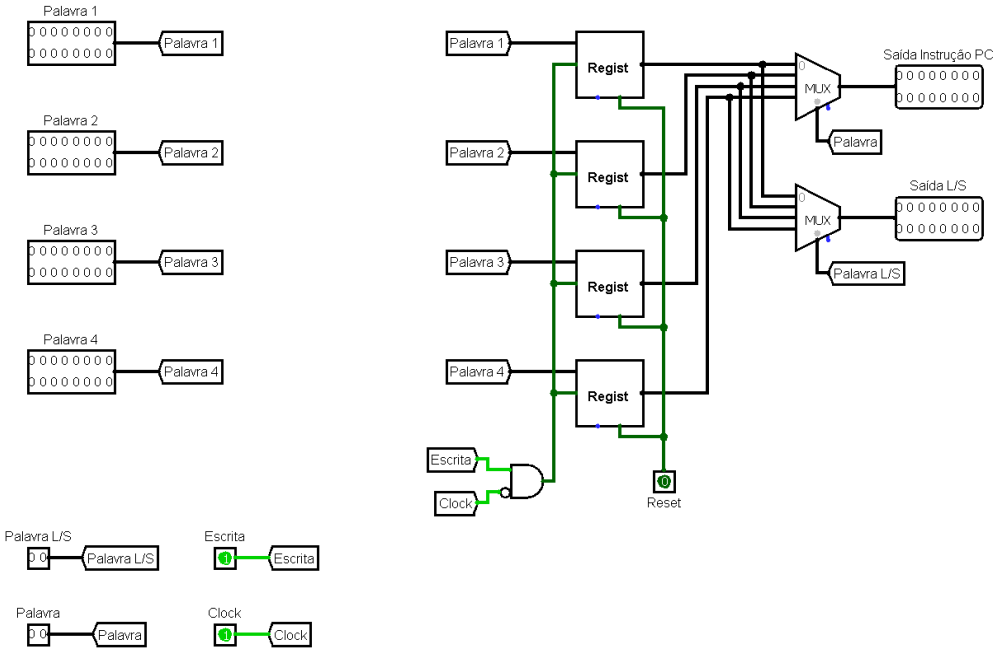
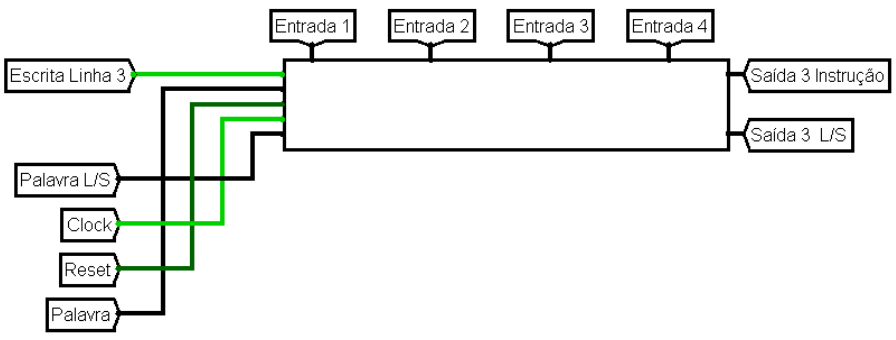


Imagem 17 - Linha da *Cache* (Parte Interna)

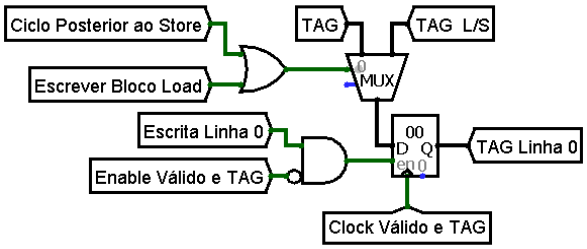


4.2.TAG

Para armazenar a TAG do bloco que está na linha, utilizamos um registrador de 5 *bits* de dados. Antes de explicarmos como a escrita da TAG ocorre, é importante salientar que os pontos citados na seção de "**Considerações Preliminares**" foram descobertos próximos do fim da implementação da *cache*, ou seja, não tínhamos ciência de que o Logisim poderia gerar tais comportamentos anômalos. Dito isso, enfrentamos alguns problemas de sincronização enquanto realizávamos a implementação da escrita da TAG.

Como ainda não sabíamos que o problema era originário do Logisim, o que nos restou foi testar as combinações possíveis com os sinais de *enable* e *clock* nas entradas dos registradores. A combinação que funcionou opera da seguinte forma: o sinal de *enable*, que é o que habilita a entrada de *clock* do registrador, é ativado apenas na baixa do *clock* quando ocorre falha de *cache*. A entrada de *clock* do registrador é ativada na alta do próximo ciclo e a escrita da TAG é realizada. Consequentemente, a TAG faltante agora está na *cache*.

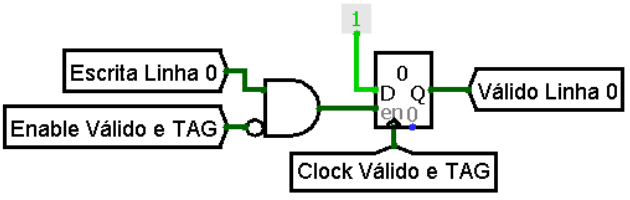
Imagem 18 - Registrador Que Armazena uma das TAGs



4.3.*Bit* de Validade

Para a atualização do *bit* de validade da linha, a configuração dos sinais é a mesma da TAG.

Imagem 19 - Registrador Que Armazena um *Bit* de Validade



4.4.Falha de *Cache*

4.4.1.Verificando Se o Bloco Está na *Cache*

No que se refere à falha de *cache*, a sinalização ocorre da seguinte maneira: o endereço atual é dividido em TAG, linha e palavra. A TAG do endereço atual é comparada com a TAG da linha informada pela instrução. Caso a TAG seja a mesma, o bloco está na *cache* e não ocorreu falha de *cache*. Caso seja diferente, isso caracteriza uma falha de *cache*.

Imagem 20 - Verificar Se o Bloco Está na *Cache* (Parte Interna)

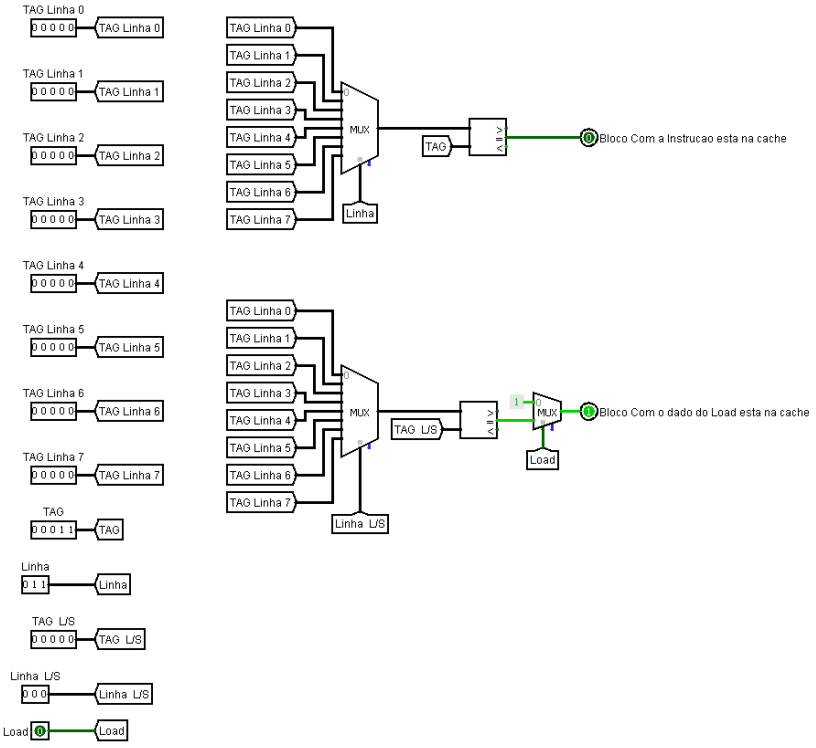
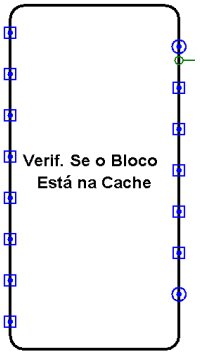


Imagem 21 - Verificar Se o Bloco Está na *Cache* (Parte Externa)



4.4.2.Verificando Se a Linha É Válida

Outro modo de ocorrer uma falha de *cache* é se o *bit* de validade da linha for zero. Se pelo menos um dos casos citados ocorrer, uma falha de *cache* aconteceu.

Ocorrendo uma falha de *cache*, na alta do *clock*, a *cache* envia para a memória principal o sinal de "*cache miss"* e a memória devolve um sinal informando a *cache* que ela deve realizar a escrita do bloco que chegará no barramento. Utilizando o campo linha do endereço, a *cache* identifica qual linha sofrerá a escrita. Na baixa do *clock*, o bloco é escrito na linha.

Imagem 22 - Verificar Se a Linha É Válida (Parte Interna)

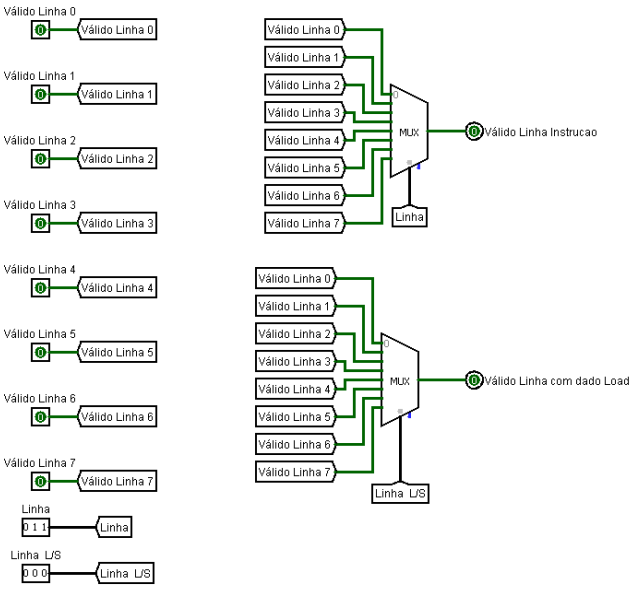


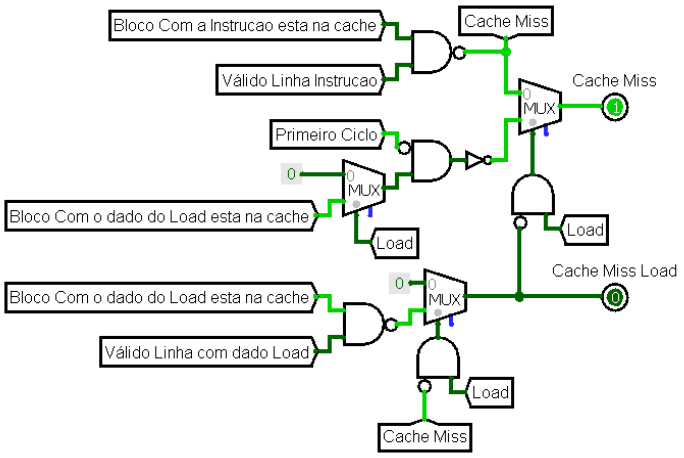
Imagem 23 - Verificar Se a Linha É Válida (Parte Externa)

4.4.3.Sinais *Cache Miss* e *Cache Miss Load*

As operações de *load* e *store* tornaram-se um pouco mais delicadas com a inclusão da *cache*.

No *load*, há a possibilidade de ocorrer duas falhas de *cache* seguidas, a primeira da instrução em si e a segunda do dado a ser lido. Por simplicidade, optamos separar as falhas de *cache*. A falha de cache comum (de instrução) nomeamos "*cache miss"* e a do dado do *load* nomeamos "*cache miss load"*. Fizemos isso pois a falha de *cache* do dado do *load* envolve mais nuances que uma falha comum. Quando um "*cache miss load"* ocorre, a memória principal utiliza o endereço proveniente da ULA, pois é o endereço do dado do *load*. O bloco contendo o dado é enviado à *cache* e a escrita do dado é realizada.

Imagem 24 - Sinais *Cache Miss* e *Cache Miss Load*



4.4.4.Lidando com as Falhas de *Cache*

Quando uma falha de *cache* acontece, independente de sua natureza, é preciso realizar um *stall*, para que no próximo ciclo o PC ainda aponte para a mesma instrução. Isso é feito utilizando um multiplexador. Na entrada 0, está o endereço da próxima instrução. Na entrada 1, está o endereço atual. A entrada de seleção é ativada, ou seja, o PC continuará com a mesma instrução, caso algum dos seguintes cenários aconteça: ocorreu um *cache miss*,

ocorreu um *cache miss load* ou é o ciclo posterior ao *store* (tal conjuntura será explicada posteriormente).

Alguns sinais adicionais são utilizados juntamente com os sinais citados, a criação deles foi necessária pois, ao longo do fluxo de execução, alguns sinais terminavam, o que é esperado pois, no caso do *cache miss*, por exemplo, a falha de *cache* só é uma falha de *cache* enquanto a instrução ou dado não está na *cache*. No ciclo seguinte, a instrução ou dado estará na *cache*, não caracterizando mais uma falha de *cache*. Entretanto, para o *stall* funcionar, ainda era necessário saber que uma falha de *cache* aconteceu, caso contrário, o PC iria ser incrementado pois os sinais de falha não estão mais ligados. Então, tendo em vista as motivações citadas, foram criados sinais que informam: se uma falha de *cache* ocorreu e se o *load* foi finalizado.

Imagem 25 - Registrador PC (Parte Interna)

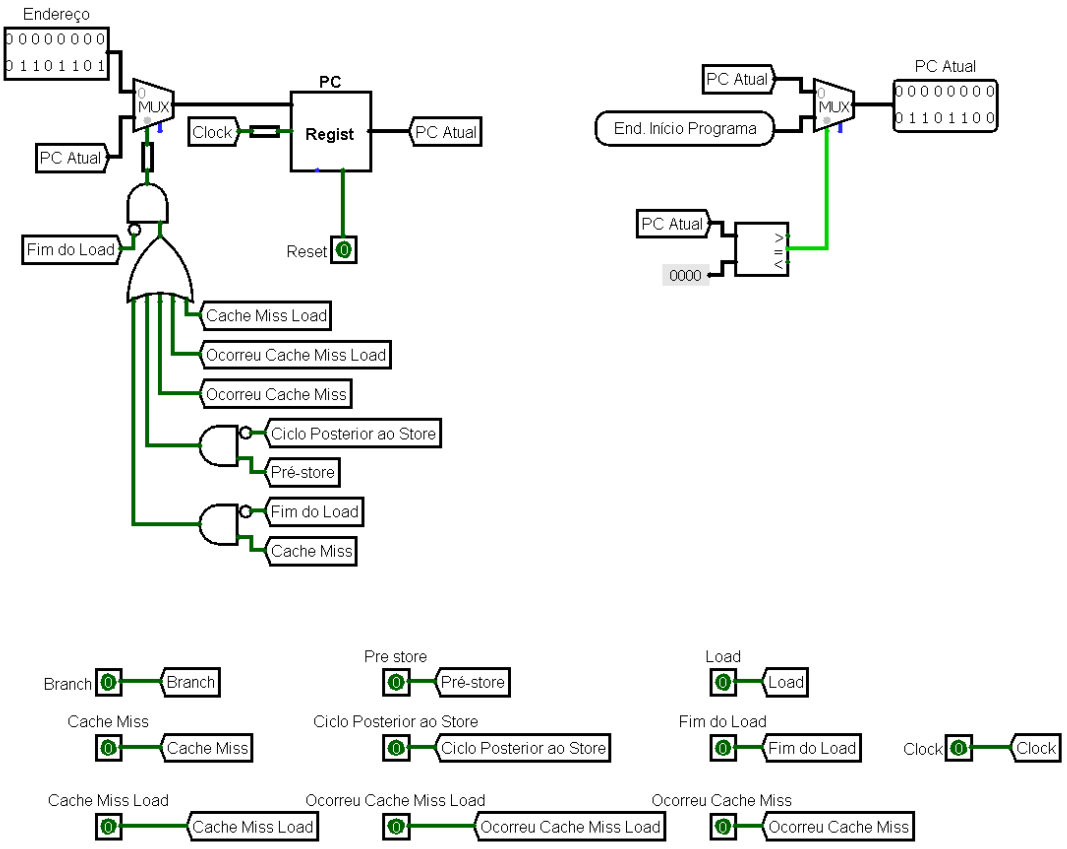
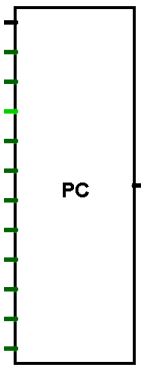


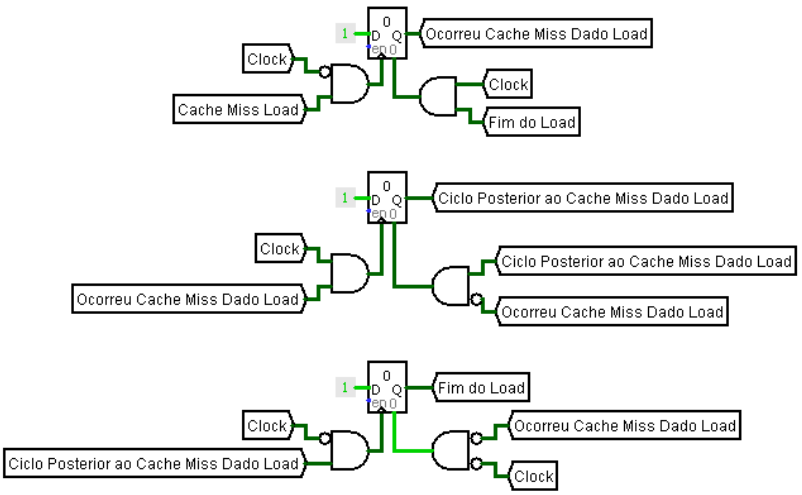
Imagem 26 - Registrador PC (Parte Externa)



4.5.*Load* e o Padrão *Handler*

Os sinais que informam se uma falha de *cache* ocorreu e se o *load* foi finalizado não são gerados pela UC. Foram criados componentes "*handlers"* que são responsáveis por gerá-los. Enquanto pensávamos em como iríamos solucionar tais necessidades, concebemos um padrão bem simples de circuito que foi utilizado em outros pontos do projeto. Vamos utilizar o sinal de "Ocorreu *Cache Miss* Dado *Load"*, para exemplificá-lo.

Imagem 27 - Registradores Sinal Ocorreu *Cache Miss* Dado *Load*



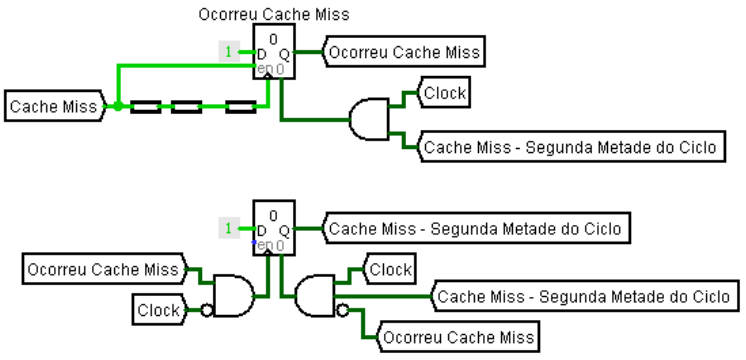
Quando ocorre um *cache miss load,* um registrador guarda esse sinal, essa informação é armazenada, ela significa que ocorreu uma falha de *cache* na tentativa de obter o dado do *load*. Se o *clock* está na alta e ocorreu uma falha de *cache* na tentativa de obter o dado do

load, então essa informação é armazenada em um segundo registrador, ela sinaliza que estamos no ciclo posterior ao ciclo que ocorreu a falha de *cache* na tentativa de obter o dado do *load*. Se o *clock* está na baixa e estamos no ciclo posterior ao ciclo que ocorreu a falha de *cache* na tentativa de obter o dado do *load*, então essa informação é armazenada em um terceiro registrador, ela sinaliza que estamos no fim do *load.* Se o sinal de "fim do *load"* está ligado e o *clock* está na alta, então já finalizamos todas as operações que ocorrem no *load.*

Consequentemente, os três registradores são limpos. A implementação dessa limpeza é simples, bastou vincular a porta CLEAR de um registrador ao sinal de saída do registrador seguinte. Ainda no exemplo do sinal de "Ocorreu *Cache Miss* Dado *Load*", se é o fim do *load* e o *clock* está na alta, então ativamos a entrada CLEAR do primeiro registrador. Se o sinal do primeiro está desligado e o sinal do segundo está ligado, então ativamos a entrada CLEAR do segundo registrador. Se o sinal do primeiro está desligado e estamos na baixa do ciclo do fim do *load*, então ativamos a entrada CLEAR do terceiro registrador. Desse modo, temos um circuito que informa em qual momento do *load* estamos, o que possibilita a utilização do *stall*.

Um mecanismo semelhante ao citado também foi necessário para sinalizar em qual momento da falha de *cache* comum estamos.

Imagem 28 - Registradores Sinal Ocorreu *Cache Miss*

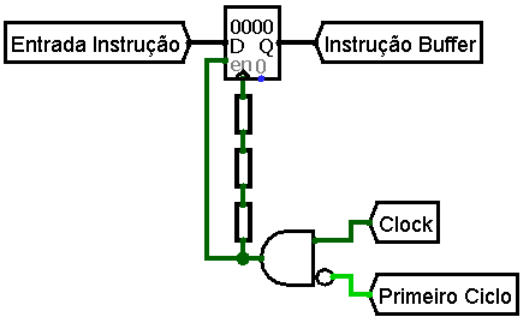


4.6.*Buffers*

Em virtude da utilização do *stall,* foram necessários dois *buffers*: o *buffer* de instrução (MAR) e o *buffer* do *store* que se assemelha ao MBR (ele será explicado na seção do *store)*.

O *buffer* de instrução foi necessário em virtude do caso onde ocorreu a falha de *cache* do dado do *load* e o bloco do dado é mapeado para a mesma linha que está a instrução atual. Logo, perdemos a instrução e, como ocorre o *stall*, precisaremos da instrução no ciclo seguinte. Então, sempre que ocorre uma falha de *cache* do dado do *load*, no ciclo seguinte, o processador obtém a instrução do *buffer* de instrução em vez da *cache*.

Imagem 29 - *Buffer* de Instrução



4.7.*Store*

Alguns entraves surgiram durante a implementação do *store*. O primeiro deles é o fato da necessidade de isolar o sinal de ativação e escrita das submemórias, pois, conforme já explicado, ruídos de sinal estavam passando e acarretando na escrita do dado, ou em mais de uma submemória ou em mais de uma posição da mesma submemória.

Em virtude de tal isolamento, quando está ocorrendo a escrita em uma das submemórias as outras são desativadas, evitando assim tais conflitos. No entanto, como a política de escrita utilizada é a escrita direta, a *cache* deve ser atualizada com o novo bloco. Logo, não é possível ler das quatro submemórias enquanto a escrita estiver ocorrendo, não sendo possível assim enviar o bloco para a *cache*. A solução é realizar um *stall* e ter um *buffer* para o *store*.

O *store* foi implementado da seguinte forma: no primeiro ciclo, um *stall* é feito para salvar o bloco que contém a posição da memória que receberá a escrita no *buffer* do *store*. O salvamento ocorre obtendo o bloco vindo da memória e, em vez de salvar a palavra que será substituída pelo novo dado, salvamos o bloco já com o novo dado no *buffer*, o ciclo termina e então estaremos no ciclo seguinte ao que ocorreu o *stall* para salvamento do bloco.

Imagem 30 - *Buffer* do *Store* (Parte Interna)

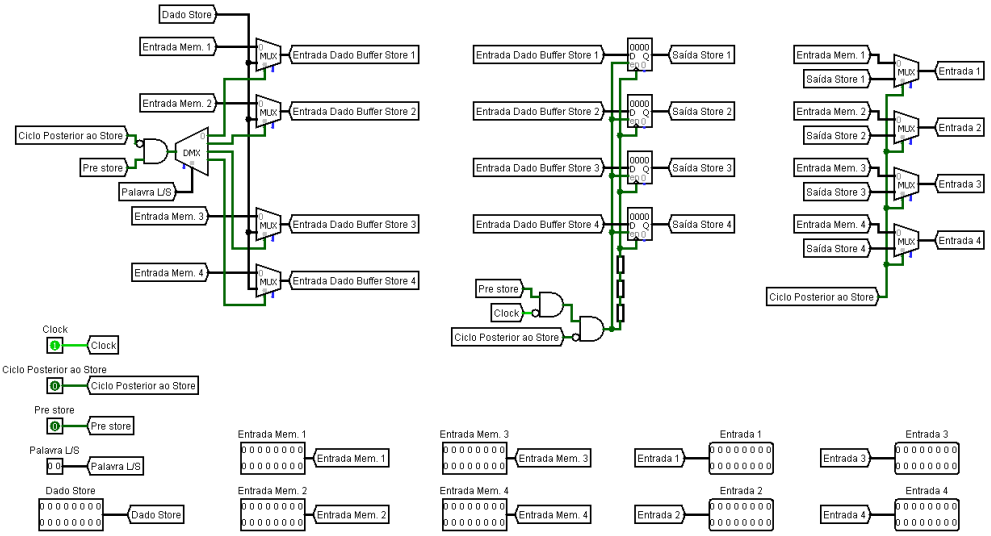
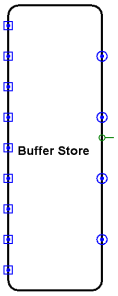


Imagem 31 - *Buffer* do *Store* (Parte Externa)



Nesse ciclo seguinte, realizamos a escrita do dado na respectiva submemória e realizamos a escrita do bloco que está no *buffer* do *store* na linha correspondente à linha do endereço que sofreu o *store*. A identificação de se é um *store* e em qual dos ciclos estamos é realizada por um circuito que utiliza o padrão explicado anteriormente (o utilizado na falha de *cache* do dado do *load*). Desse modo, foi possível implementar o *store* utilizando a escrita direta.

Imagem 32 - *Store Handler* (Parte Interna)

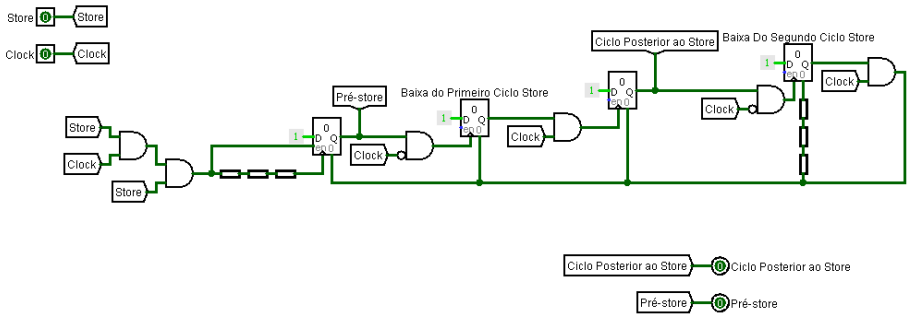


Imagem 33 - *Store Handler* (Parte Externa)

Imagem 34 - *Cache* Direta (Parte Interna)

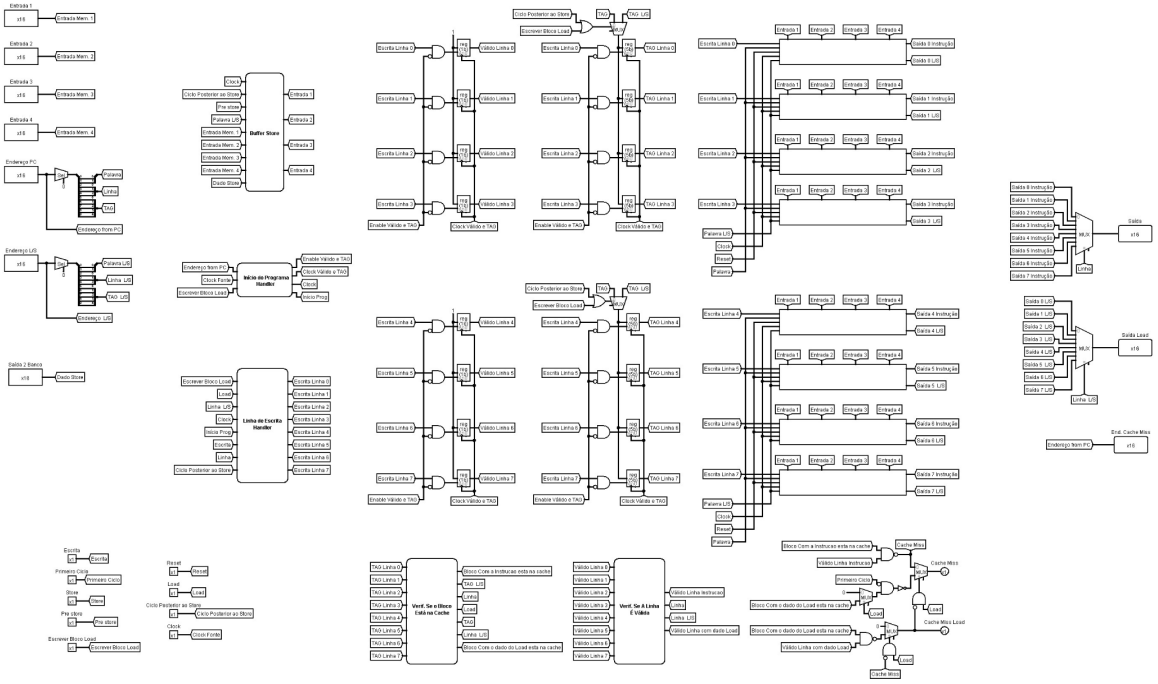
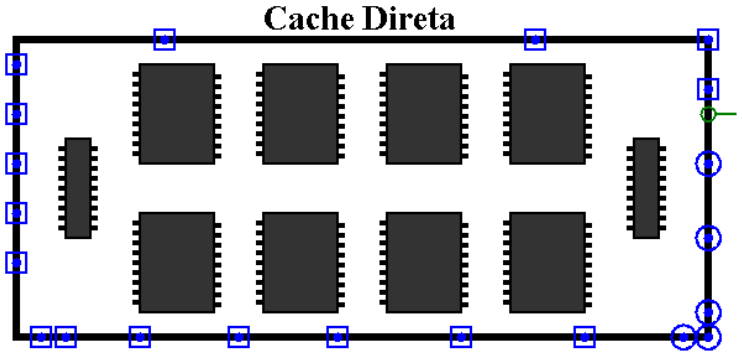


Imagem 35 - *Cache* Direta (Parte Externa)



5.FIM DO PROGRAMA

A identificação do fim do programa ocorre quando a saída da *cache* (instrução atual) é zero. Se a instrução atual é zero, então estamos no fim do programa e um *led* é ativado para ajudar na identificação. Como não é possível parar a simulação automaticamente via circuito, quando o fim do programa acontece, travamos o *clock*. Dessa maneira, mesmo que a simulação continue, o *clock* não surtirá mais efeito e os dados do banco de registradores e da memória não serão sobrescritos com valores indevidos.

Imagem 36 - Detectar Fim do Programa

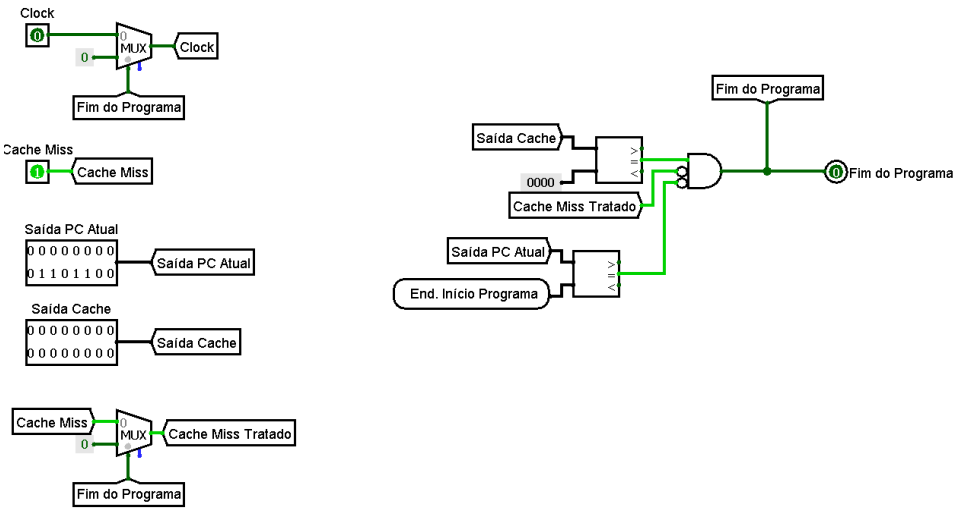
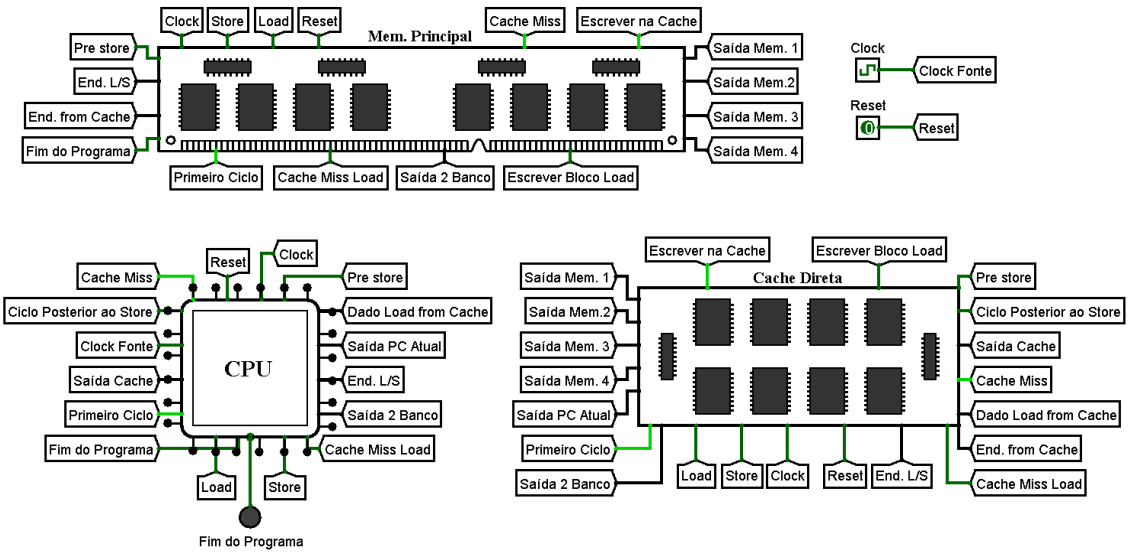


Imagem 37 - Circuito Principal



6.VALIDANDO A *CACHE* DIRETA

Para validarmos a *cache*, utilizamos o programa disponibilizado no documento do trabalho. Entretanto, como o programa possui uma quantidade de iterações razoável, resolvê-lo manualmente seria contraproducente.

Uma alternativa é a transposição do programa para uma linguagem de alto nível. Por simplicidade, escolhemos a linguagem *Python*.

Após a transposição e execução do programa, descobrimos que o valor final das posições da memória referentes às variáveis de soma (soma1 à soma5) eram, respectivamente, 155, 155, 310, 75 e 516. Valores os quais estavam presentes nas respectivas posições em cada uma das nossas submemórias. Dessa forma, foi possível validar a corretude da *cache* direta.

7.*CACHE* ASSOCIATIVA POR CONJUNTO

Após a finalização da *cache* direta, iniciamos a implementação da associativa por conjunto. No entanto, não conseguimos finalizá-la, pois chegamos a um ponto que, mesmo aplicando todos os padrões de soluções para as anomalias oriundas do Logisim que

aprendemos durante a implementação da direta, não conseguimos resolvê-los. Assim, iremos relatar a implementação até o momento mencionado.

7.1.Reaproveitando o Circuito

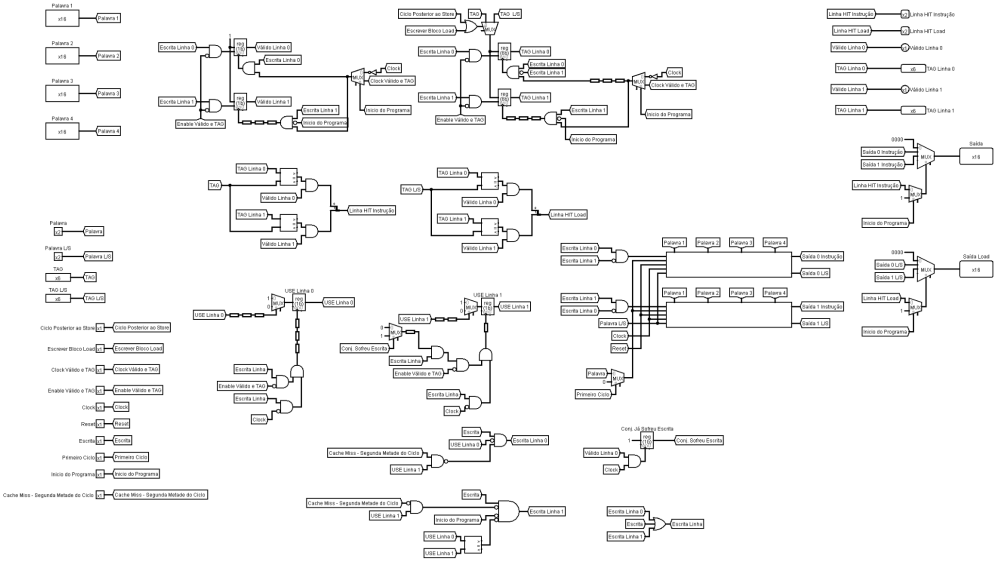
Tendo em vista que a Memória Principal foi criada e o processador se tornou robusto durante a implementação da *cache* direta, foi conveniente reaproveitarmos ambos para a implementação da associativa por conjunto. Além disso, reaproveitamos também o circuito da *cache* direta, pois a maioria dos componentes não mudarão.

7.2.Criando os Conjuntos

O primeiro passo foi transformar a *cache* direta em associativa por conjunto. Para isso, retiramos as linhas da *cache* e criamos os conjuntos.

Em cada conjunto, temos duas linhas. Além disso, temos seus respectivos *bits* de validade, *tags* e os *bits USE* de cada linha usados no *LRU*.

Imagem 38 - Conjunto



7.3.*LRU* e a Falha de *Cache*

Para implementarmos o *LRU*, usamos um *bit USE* para cada linha no conjunto.

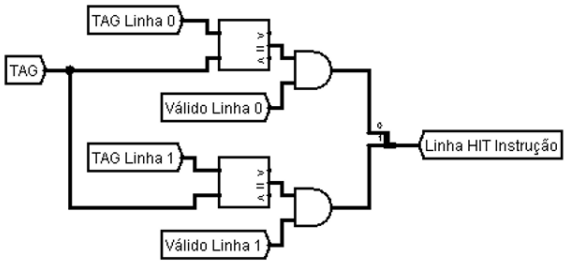
No *LRU*, o bloco é escrito na linha cujo *bit USE* é 0, utilizamos registradores de 1 *bit* para armazená-los. No entanto, o Logisim inicia com os valores de ambos os registradores sendo 0. Logo, foi necessário tratar tal cenário.

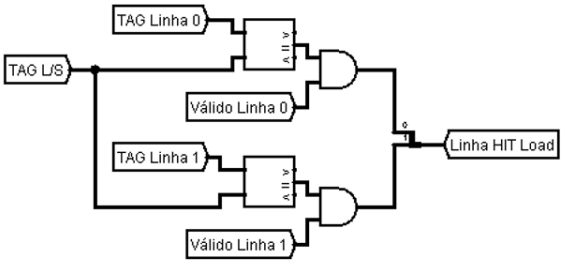
No início do programa, já iniciamos com uma falha de *cache.* Então, é feita a verificação de se estamos no início do programa, caso sim, o sinal de escrita da linha 1 não é ligado. Nas próximas falhas de *cache,* em que os conjuntos ainda não tiveram blocos mapeados para eles, verificamos se ambos os de ambos os *bits USE* são 0. Caso sim, o sinal de escrita da linha 1 também não é ligado.

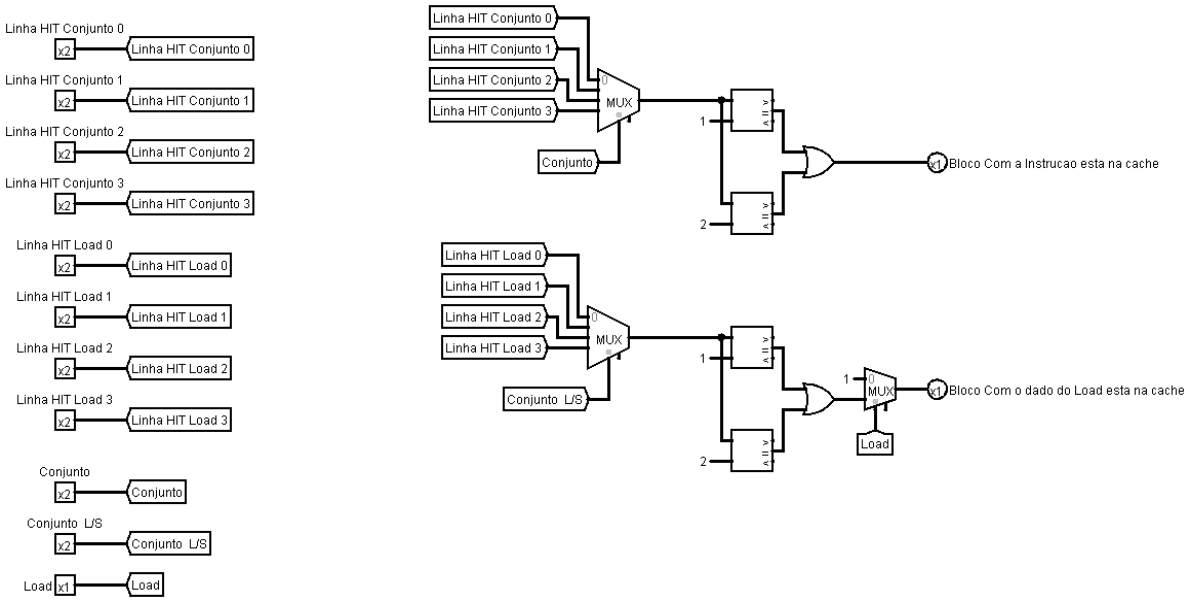
Assim, ainda que ambos os *bits USE* iniciem sendo 0, conseguimos realizar a escrita de forma correta. Após a escrita, o *bit USE* da linha que sofreu a escrita será 1 e o da outra linha será 0.

Na falha de *cache*, verificamos se alguma das *tags* é igual à *tag* do endereço e, quando é um *load*, se é igual a do endereço do *load* oriundo da ULA. Visando deixar o circuito conciso, juntamos os sinais de *cache hit* em um só. Um distribuidor recebe ambos os sinais, fazendo com que a saída seja um número de 2 *bits*. Assim, se após a comparação de *tags* nenhum dos *bits* for 1, o que seria os números 01 ou 10 (1 ou 2), então o bloco não está em nenhuma das linhas.

Imagens 39 e 40 - Detectando Uma Falha de *Cache*







7.4. O Apogeu do Desenvolvimento

Apesar do sucesso na implementação da *cache* direta, infelizmente, não conseguimos tal feito no desenvolvimento da associativa por conjunto.

Na segunda falha de *cache* do programa, o bloco seria escrito na segunda linha de um dos conjuntos, pois o bloco de um dos *loads* foi mapeado para ele. No entanto, anomalias oriundas do Logisim aconteceram.

Apesar de existirem portas lógicas que realizam o filtro para que a escrita ocorra apenas na linha cujo sinal de escrita esteja ligado, fazendo com que o sinal de escrita da outra linha permaneça desligado até o fim da escrita, por alguma razão, a qual desconhecemos, algo como um ruído de sinal passa pelos filtros e faz com que o bloco seja gravado em ambas as linhas.

Simultaneamente, nos registradores que armazenam os *bits USE*, apesar da entrada de *clock* ser ativada durante a escrita, por algum motivo, não surte efeito nos registradores, fazendo com que os *bits USE* permaneçam sem alterações.

Mediante tais circunstâncias, tentamos todas as combinações que conseguimos pensar dos padrões que sempre resolviam as anomalias do Logisim, no entanto, sem sucesso. Em virtude disso, não conseguimos dar continuidade ao desenvolvimento da *cache* associativa por conjunto, o que, ao final, resultou em apenas o projeto dessa.

Imagem 41 - *Cache* Associativa Por Conjunto

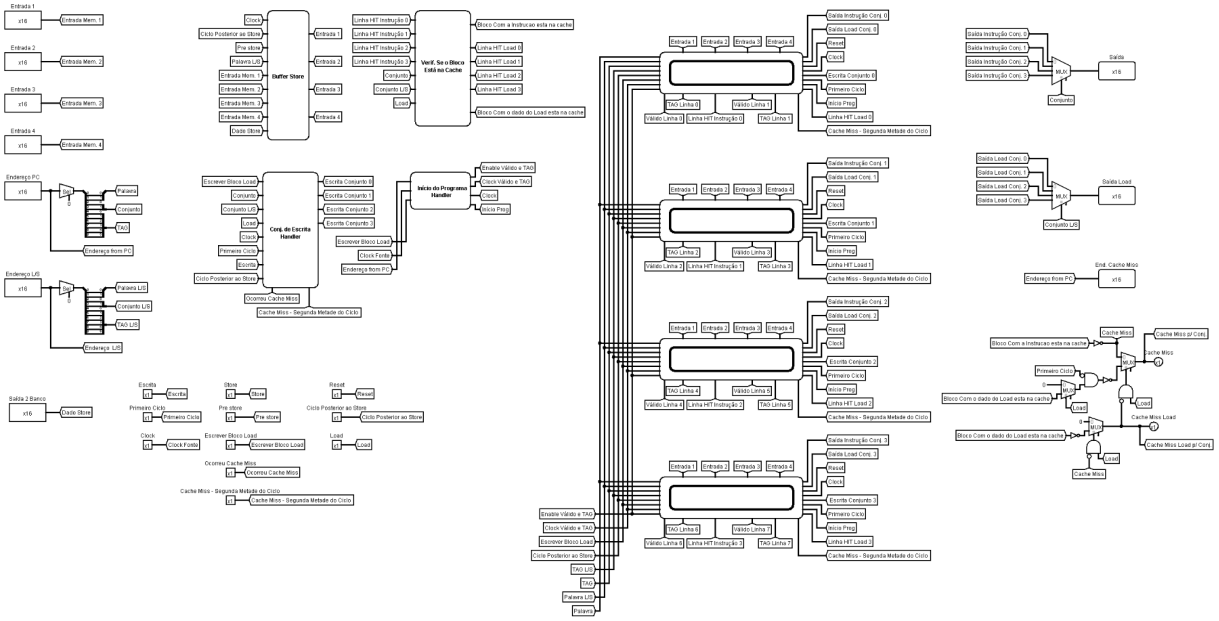
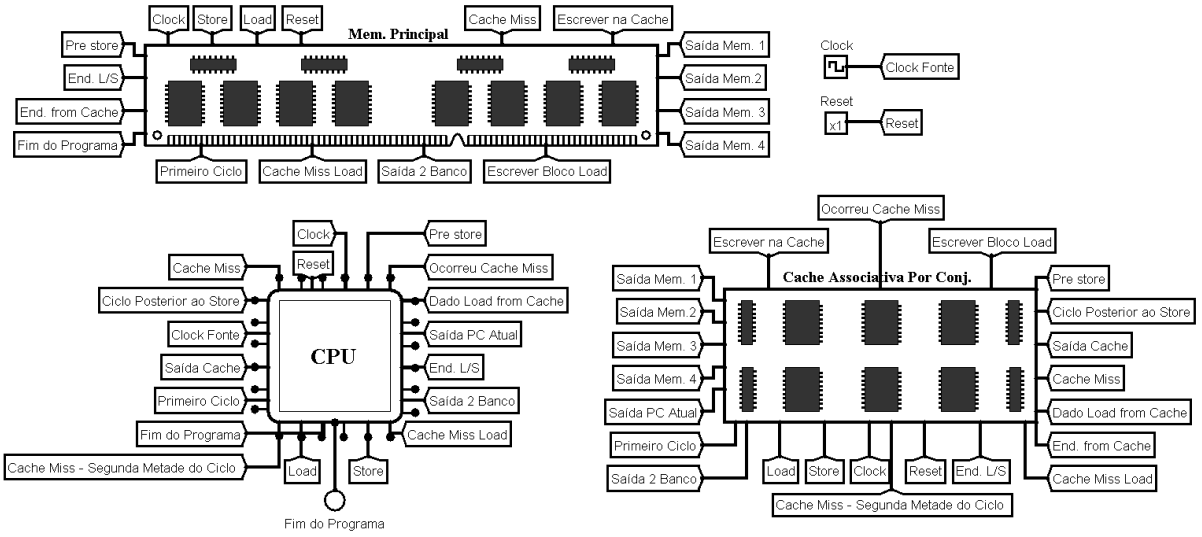


Imagem 42 - Circuito Principal



8.REFERÊNCIAS

PATTERSON, David, HENNESSY, John .Organização e Projeto de Computadores: Interface Hardware/Software. 4° Edição. Rio de Janeiro, Editora Elsevier, 2014.

STALLINGS, William. Arquitetura e Organização de Computadores. 10ª Edição. São Paulo: Pearson Pratice Hall, 2010.

Documentação oficial da versão 2.7.x, 2023. Disponível em: <http://www.cburch.com/logisim/pt/index.html>. Acesso em: 03 de outubro de 2023.