Uma Implementação do RISC I Utilizando o Simulador Logisim Evolution

Diogo Valadares Reis dos Santos*

2024 v-0.01

Resumo

Este relatório técnico detalha a implementação da arquitetura RISC I utilizando o software Logisim Evolution. A arquitetura RISC I foi escolhida por sua simplicidade e importância histórica no desenvolvimento de processadores RISC, com informações obtidas de publicações acadêmicas e outros relatórios técnicos. Várias adaptações foram necessárias para garantir o funcionamento adequado no ambiente de simulação, incluindo ajustes de temporização e na unidade de controle para acomodar as limitações do software e a falta de certas informações dos trabalhos originais. Diferenças notáveis entre o modelo implementado e a arquitetura original do RISC I incluem a simplificação de certos circuitos e o uso de componentes pré-existentes no Logisim Evolution, o que facilitou a construção e a simulação. Para programar e testar o modelo implementado, foi desenvolvido um *assembler* em C# para converter o código *assembly* em instruções de máquina compatíveis com a memória do simulador, envolvendo a listagem do conjunto de instruções do RISC I e a implementação de um parser para traduzir o código *assembly*.

Palavras-chaves: RISC, RISC I, RISC II, Arquitetura e Organização de Computadores, Simuladores, Logisim, Assembly, C#.

Abstract

This technical report details the implementation of the RISC I architecture using Logisim Evolution software. The RISC I architecture was chosen for its simplicity and historical significance in the development of RISC processors, with information obtained from academic publications and other technical reports. Various adaptations were necessary to ensure proper functioning in the simulation environment, including timing adjustments and modifications to the control unit to accommodate software limitations and the lack of certain information from the original works. Notable differences between the implemented model and the original RISC I architecture include the simplification of certain circuits and the use of pre-existing components in Logisim Evolution, which facilitated construction and simulation. To program and test the implemented model, an assembler was developed in C# to convert assembly code into machine instructions compatible with the simulator's memory, involving the listing of the RISC I instruction set and the implementation of a parser to translate the assembly code.

Key-words: RISC, RISC I, RISC II, Computer Organization and Architecture, Simulator, Logisim, Assembly, C#.

*diogo-valadares@hotmail.com

1 Introdução

A arquitetura RISC I teve uma grande importância, sendo a definição do conceito de arquiteturas RISC. Ela foi desenvolvida no início dos anos 80, liderada pelos professores David Patterson e Carlos Séquin, e executada pelos seus estudantes (Fitzpatrick, Foderaro, Peek, Peshkess e Van Dyke), com melhorias feitas por outro grupo (Katevenis e Sherburne), resultando no RISC II (Daniel T. Fitzpatrick et al., 1982).

Este relatório técnico é resultado de uma monografia com o objetivo de criar uma arquitetura didática baseada em uma arquitetura já existente. O RISC I foi escolhido por ser uma excelente arquitetura, que além de ser relativamente simples em comparação a arquiteturas atuais, permitindo modificações, também apresenta recursos bem relevantes e que ainda estão presentes em arquiteturas recentes.

Para a replicação do RISC I, foram utilizados dois principais trabalhos realizados pelos estudantes que criaram o RISC I e o RISC II. O primeiro foi feito por Peek (1983), onde ele detalha os diversos componentes da arquitetura, porém o trabalho deixa alguns detalhes de fora, como o funcionamento mais detalhado de interrupções, códigos condicionais, controle da janela de registradores, e leitura/escrita de diferentes tamanhos de dados. Para complementar os detalhes não encontrados no trabalho de Peek, foi utilizado o trabalho de Katevenis (1985), que detalhou diversos aspectos do RISC II, que apesar de conter algumas modificações em relação ao RISC I, ainda possui diversas similaridades que são citadas pelo trabalho.

Neste trabalho, a recriação do RISC I foi realizada como um modelo de simulação dentro do software Logisim Evolution, que é um *fork* do Logisim, descontinuado pelo seu criador. O Logisim Evolution foi escolhido por oferecer visualmente o funcionamento da lógica do circuito de forma interativa, permitindo rápidos testes e entendimento do funcionamento de certas partes, sendo excelente para a criação e prototipagem de circuitos sem conhecimento técnico de linguagens de descrição de hardware.

O modelo de simulação criado e todo código do assembler estão disponíveis via GitHub em https://github.com/Diogo-Valadares/Didactic-RISC-I.

2 A Arquitetura RISC I

2.1 Formato de Instrução

Há basicamente dois formatos de instrução no RISC I, o com imediato longo e o com dois registradores, ou um registrador e imediato. Estes formatos são demonstrados tanto no trabalho de Katevenis quanto no de Stallings (1988), demonstrado na figura 1.

O formato com o imediato longo é utilizado em instruções relativas, onde o imediato longo é adicionado ao contador de programa para obter um endereço relativo ao atual. Ele também é utilizado na instrução *Load High Immediate* (LDHI), onde os 19 bits são carregados para os bits mais significativos, permitindo o carregamento de constantes que ocupem os 32 bits dos registradores em menos ciclos do que com instruções de carregamento normais. Já o formato com imediato curto é utilizado para quaisquer outras instruções.

Para ambos os formatos, os 3 primeiros parâmetros são iguais. Os primeiros 7 bits representam a instrução que deve ser executada, seguidos de uma *flag* (um bit) para definir códigos de condição, indicando se a instrução deve gravar as *flags* de resultado da ULA na PSW. A última parte possui 5 bits, que podem representar um endereço de destino do resultado ou uma condição. Caso seja uma condição, apenas os 4 bits menos significativos são usados, com o quinto sendo ignorado.

Para o formato com imediato curto, os primeiros 5 bits após o destino são sempre um registrador fonte. Este registrador fonte é então seguido de uma segunda fonte, que pode ser tanto um registrador quanto um imediato de 13 bits. Para que o imediato seja escolhido, o primeiro bit após a primeira fonte deverá ser 1, seguido do valor desejado para o imediato. Caso o bit após a primeira fonte seja 0,

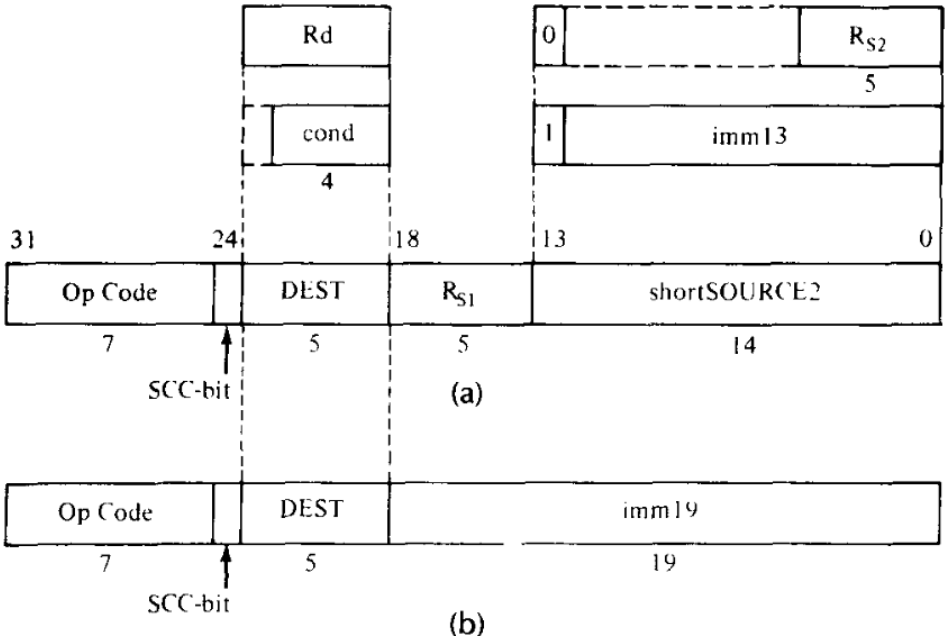

Figura 1 – Formato de instrução do RISC I(STALLINGS, 1988).

os 5 bits menos significativos representam o registrador que será utilizado como segunda fonte.

2.2 Conjunto de Instruções

+	-	0 —-000	1 —-001	2 —-010	3 —-011	4 —-100	5 —-101	6 —-110	7 —-111
0	0000—	CALLI	SLL		STS	AND	SUB	RET	LDHI
1	0001—				STRS				
2	0010—		GETPSW				SUBC		
3	0011—								
4	0100—			LDBU	STB	XOR	SUBR	RETI	
5	0101—			LDRBU	STRB				
6	0110—			LDBS			SUBCR		
7	0111—			LDRBS					
8	1000—	CALL	SRL	LDW	STW	OR	ADD	GETLPC	
9	1001—			LDRW	STRW				
A	1010—	JMP	PUTPSW						
B	1011—								
C	1100—	CALLR	SRA	LDSU			ADDC		
D	1101—			LDRSU					
E	1110—	JMPR		LDSS					
F	1111—			LDRSS					

Tabela 1 – Conjunto de instruções do RISC I(adaptação de Peek (1983))

A tabela 1 mostra todas as instruções existentes no RISC I. Algumas das instruções tiveram os seus nomes modificados para melhorar a consistência e legibilidade, mas todas mantém as suas funções originais. Das alterações que foram feitas temos a remoção do caractere "X"em instruções como "JMPX", "LDXW", "STXW"ou "CALLX", pois este "X"representa que a instrução utiliza um registrador ao invés de um operante relativo, o que se torna desnecessário já que as instruções que usam relativo já são marcadas com um "R", além de que o "X"pode ser confuso para entender apenas de olhar a instrução. As instruções de subtração foram alteradas para não usar a letra "I", no lugar elas usam "R", um padrão usado por Stallings (1988) que ajuda a não confundir com outras instruções que envolvem interrupções e que também terminam em "I". "GTLPC"foi alterado para "GETLPC"para ficar consistente com a instrução "GETPSW". Por fim, para instruções de *Load* e *Store* foram adotadas as letras "B"para as que utilizam apenas *bytes*, e "S"para as que utilizam *shorts*(*2 bytes*).

Mnemonico	Nome completo	HEX	BIN	Operação
CALLI	**CALL** WITH **I**NTERRUPT	00	000 0XXX	CWP– Rd <= Last pc
CALL	**CALL**	08	000 100X	CWP– Rd <= pc Next pc <= Rx + S2
JMP	CONDITIONAL **JUMP**	0A	000 101X	pc <= Rx+S2
CALLR	**CALL R**ELATIVE	0C	000 110X	CWP– Rd <= pc Next pc <= pc + Y
JMPR	CONDITIONAL **JUMP R**ELATIVE	0E	000 111X	pc <= pc + Y
SLL	**S**HIFT **L**EFT **L**OGICAL	10	001 0X0X	Rd <= Rs « S2
GETPSW	**GET P**ROCESSOR **S**TATUS **W**ORD	12	001 0X1X	Rd <= PSW
SRL	**S**HIFT **R**IGHT **L**OGICAL	18	001 100X	Rd <= Rs » S2
PUTPSW	**PUT P**ROCESSOR **S**TATUS **W**ORD	1A	001 101X	PSW <= Rm
SRA	**S**HIFT **R**IGHT **A**RITHMETIC	1C	001 11XX	Rd <= Rs »> S2
LDBU	**L**OAD **B**YTE **U**NSIGNED	24	010 0100	Rd <= M[Rx+S2]
LDRBU	**L**OAD **R**ELATIVE **B**YTE **U**NSIGNED	25	010 0101	Rd <= M[pc+Y]
LDBS	**L**OAD **B**YTE **S**IGNED	26	010 0110	Rd <= M[Rx+S2]
LDRBS	**L**OAD **R**ELATIVE **B**YTE **S**IGNED	27	010 0111	Rd <= M[pc+Y]
LDW	**L**OAD **W**ORD	28	010 10X0	Rd <= M[Rx+S2]
LDRW	**L**OAD **R**ELATIVE **W**ORD	29	010 10X1	Rd <= M[pc+Y]
LDSU	**L**OAD **S**HORT **U**NSIGNED	2C	010 1100	Rd <= M[Rx+S2]
LDRSU	**L**OAD **R**ELATIVE **S**HORT **U**NSIGNED	2D	010 1101	Rd <= M[pc+Y]
LDSS	**L**OAD **S**HORT **S**IGNED	2E	010 1110	Rd <= M[Rx+S2]
LDRSS	**L**OAD **R**ELATIVE **S**HORT **S**IGNED	2F	010 1111	Rd <= M[pc+Y]
STS	**ST**ORE **S**HORT	30	011 00X0	M[Rx+S2] <= Rm
STRS	**ST**ORE **R**ELATIVE **S**HORT	31	011 00X1	M[pc+Y] <= Rm
STB	**ST**ORE **B**YTE	34	011 01X0	M[Rx+S2] <= Rm
STRB	**ST**ORE **R**ELATIVE **B**YTE	35	011 01X1	M[pc+Y] <= Rm
STW	**ST**ORE **W**ORD	38	011 1XX0	M[Rx+S2] <= Rm
STRW	**ST**ORE **R**ELATIVE **W**ORD	39	011 1XX1	M[pc+Y] <= Rm
AND	**AND**	40	100 00XX	Rd <= Rs & S2
XOR	E**X**CLUSIVE **OR**	44	100 01XX	Rd <= Rs Ŝ2
OR	**OR**	48	100 1XXX	Rd <= Rs | S2
SUB	**SUB**TRACT	50	101 000X	Rd <= Rs - S2
SUBC	**SUB**TRACT WITH **C**ARRY	52	101 001X	Rd <= Rs - S2 - C
SUBR	**SUB**TRACT **R**EVERSED	54	101 010X	Rd <= S2 - Rs
SUBCR	**SUB**TRACT WITH **C**ARRY **R**EVERSED	56	101 011X	Rd <= S2 - Rs - C
ADD	**ADD**	58	101 10XX	Rd <= Rs + S2
ADDC	**ADD** WITH **C**ARRY	5C	101 11XX	Rd <= Rs + S2 + C
RET	**RET**URN	60	110 00XX	pc <=Rx+S2 Next CWP++
RETI	**RET**URN WITH **I**NTERRUPT	64	110 01XX	pc <=Rx+S2 Next CWP++
GETLPC	**GET L**AST **P**ROGRAM **C**OUNTER	68	110 1XXX	Rd <= Last pc
LDHI	**L**OAD **H**IGH **I**MMEDIATE	70	111 XXXX	Rd <= (Y[31:13],13'0)

Tabela 2 – Instruções do RISC I

2.3 *Datapath*

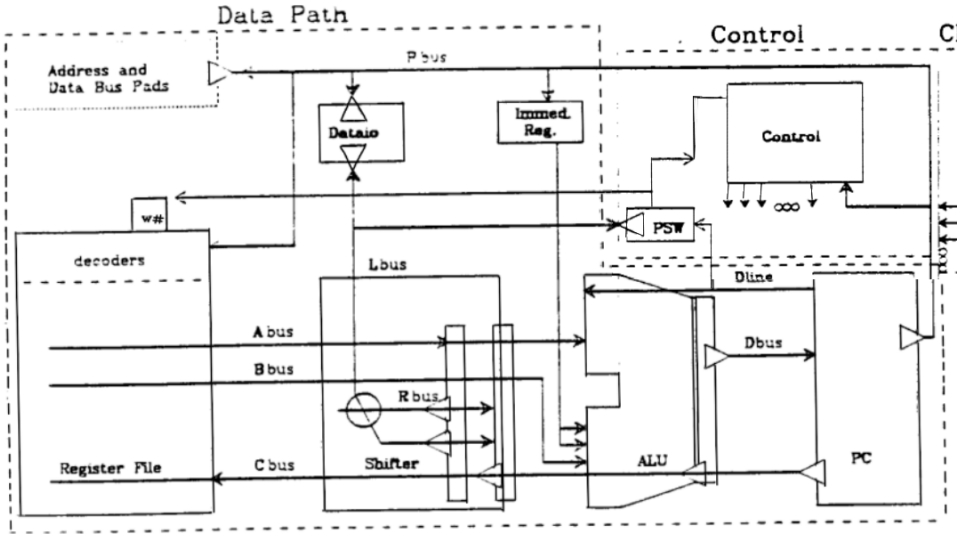

Figura 2 – *Datapath* do RISC I(PEEK, 1983)

A figura 2 mostra uma visão um pouco mais detalhada do RISC em comparação a outros trabalhos relacionados. Ela mostra componentes que geralmente são omitidos como o *DataIO*, que serve como *buffer* de entrada e saída de informações, o registrador imediato, o número de janela e o registrador de estado do processador(PSW). uma boa parte dos barramentos é mostrado de forma atravessada sobre certos componentes pois é assim que estes são colocados no chip como descrito por Peek. Nesse trabalho, como tentativa de deixar as informações mais visíveis, os barramentos não passam por cima dos componentes, no caso os barramentos A, B e C passam abaixo de alguns dos componentes enquanto a linha D passa por cima, como demonstrado na figura 3.

Vale notar que o trabalho disponível do Peek possui erros de impressão nos seus diagramas, onde linhas verticais ficam falhadas. Para tentar diminuir erros, foi executada uma restauração nos diagramas, porém alguns ainda podem conter falta de informações. A conexão entre o PSW, o número da janela estava apagada no trabalho original o que dificulta o entendimento das conexões. Isso veio junto com uma inconsistência, onde há um diagrama que demonstra que o controlador do número de janela se conecta a PSW através do barramento L. A solução para a implementação no simulador foi utilizar a conexão pelo barramento D como demonstrado na figura 3.

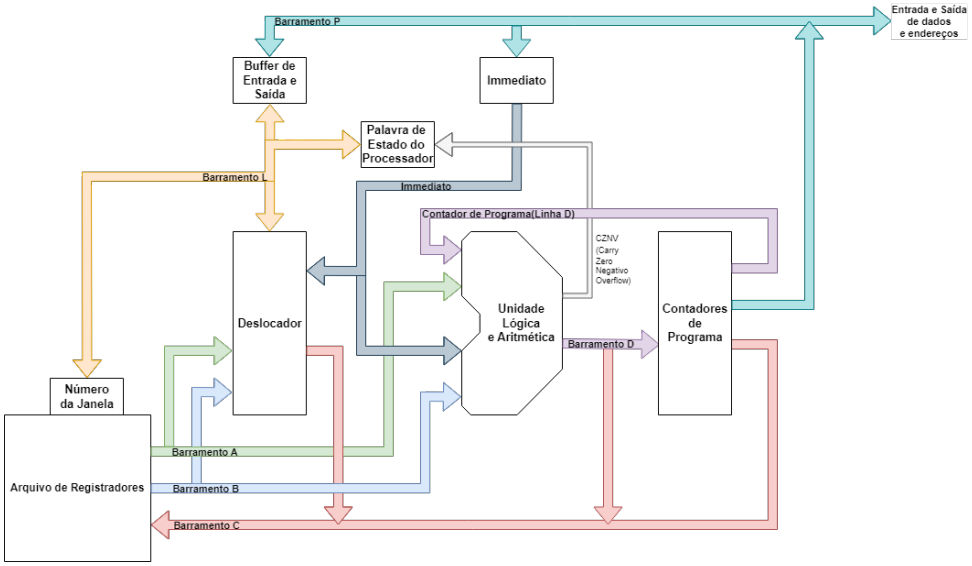

Figura 3 – Arquitetura do RISC I(autoria própria).(Note que componentes de controle e *clock* foram omitidos)

Uma alteração foi realizada entre a unidade lógica e aritmética(ULA) e o contador de programa(PC). No trabalho original, o resultado da ULA se separa internamente entre duas saídas, uma para o barramento C, e outro negada para o barramento D. A negação para o barramento D se deve a implementação física, onde os barramentos A, B, C e D são pré-carregados em alta voltagem, e são lidos descarregando as conexões onde há zeros, resultando em uma inversão da informação. A saída para o barramento C não precisa ser invertida, porque ela é armazenada em registradores que são lidos apenas através dos barramentos A ou B, que invertem a informação de volta ao que era originalmente. O barramento D por outro lado, se conecta diretamente ao contador de programa, que precisa que a informação não esteja negada, pois essa informação são os endereços que são passados para o barramento P, que não é pré-carregado.

Como a implementação no simulador não necessita de pré carregamento dos barramentos, a ULA nesta implementação possui apenas uma saída que conecta ao barramento D. O barramento D ainda conecta a ULA ao PC, porém foi adicionado uma conexão controlada(através de *buffers tri-state*) ao barramento C. Em uma visão geral, esta mudança não altera o funcionamento dos recursos da arquitetura sendo equivalente ao que foi implementado originalmente.

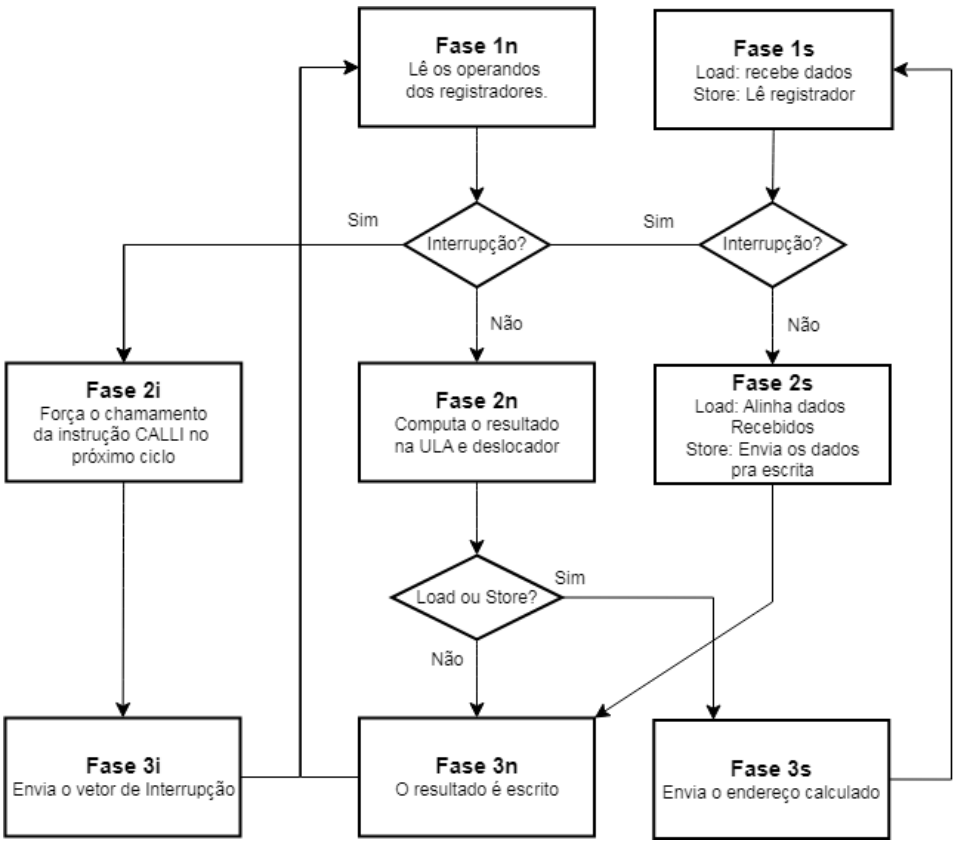

Figura 4 – Temporização do RISC I (adaptado de Peek (1983)

2.4 Temporização

A temporização do RISC I funciona a partir de 3 fases por instrução, o que geralmente envolve o carregamento de alguma informação durante a fase 1, uma execução durante a fase 2 e finalmente o armazenamento do resultado durante a fase 3.

As instruções de *store* e *load* possuem um ciclo secundário (denominado com um **s**), que ocorre entre as fases 2 e 3 do ciclo normal de instruções(denominado com um **n**). Este ciclo secundário ocorre pois para a execução dessas instruções primeiro é necessário calcular o endereço de memória que será utilizado para depois este ser enviado a memória para que a informação seja retornada, o que exige o ciclo adicional.

Além do ciclo secundário para instruções de *load* e *store* existe o ciclo de interrupção. Este ciclo ocorre quando algum problema ou erro ocorre durante a instrução atual. No RISC I isso envolve o *overflow* ou *underflow* do número de janela de registradores. Para que a interrupção ocorra, também é necessário que a *flag* de interrupções esteja ativa(será discutido na seção 3.3). A interrupção é detectada durante o fase 1, e caso ocorra, as fases 2i e 3i são executadas, forçando a próxima instrução a ser um "CALLI"e enviando o vetor de de interrupção para a memória, que seria o endereço que possui a rotina de tratamento da interrupção.

2.5 *Pipeline*

A *pipeline* de instruções é uma técnica que melhora o desempenho do processador ao dividir a execução de uma instrução em várias etapas, como busca, decodificação, cálculo de operandos, busca de operandos e execução. Semelhante a uma linha de montagem, onde diferentes estágios de produção ocorrem simultaneamente, a *pipeline* permite que várias instruções sejam processadas ao mesmo tempo em diferentes estágios. No entanto, a eficiência da *pipeline* pode ser afetada por fatores como o tempo de execução ser maior que o tempo de busca e instruções de desvio condicional que tornam o endereço da próxima instrução desconhecido. Para mitigar esses problemas, técnicas como a

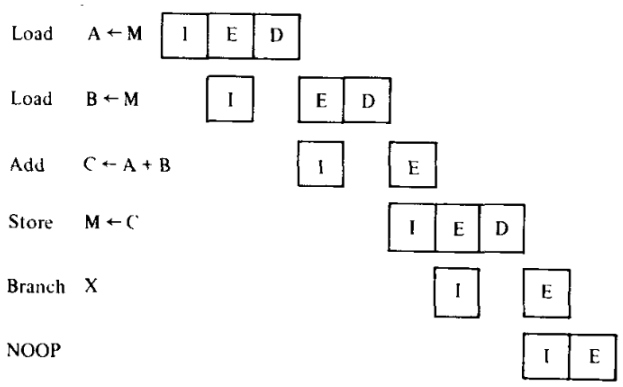

Figura 5 – Demonstração da *pipeline* do RISC I.(STALLINGS, 1988)

predição de desvios são utilizadas, permitindo um aumento significativo na velocidade de execução das instruções. (STALLINGS, 1988, p. 47-51)

A *pipeline* do RISC I(Figura 5) é bem simples, ela se divide em duas etapas: A busca da próxima instrução seguida da execução. O fato da pipeline ser tão simples pode ajudar a simplificar o entendimento de como uma *pipeline* funciona. Na busca da próxima instrução é possível ver que o processador esta acessando um certo endereço que esta sendo lido e separado entre varias partes pelo circuito. Enquanto isso, é possível também ver os barramentos A,B e C trocando dados como instruído no endereço que estava sendo lido anteriormente. Além disso, diversos registradores mostram no simulador qual é a próxima coisa que será utilizada e qual esta sendo utilizada no momento.

3 Implementação

3.1 Limitações da simulação e principais diferenças da implementação física

Os simuladores de circuitos lógicos são ferramentas poderosas para o desenvolvimento e teste de sistemas digitais. No entanto, esses simuladores possuem limitações que devem ser consideradas. Primeiramente, os simuladores de circuitos lógicos, como o Logisim, são incapazes de replicar com precisão todos os aspectos físicos dos circuitos reais, como atrasos de propagação e efeitos parasitas. Além disso, certos comportamentos complexos e interações entre componentes podem não ser completamente modelados, resultando em discrepâncias entre a simulação e o desempenho real do circuito.

Para poder executar essa implementação, algumas medidas foram tomadas para que o modelo funcione corretamente. Uma destas medidas já foi citada na seção 2.3, onde foi removida a negação do barramento D pelo diferente funcionamento dentro do simulador, onde os barramentos não precisam ser pré carregados.

Outra modificação que foi realizada foi a alteração de todos os *latches* para *flip flops*. Na implementação original, *latches* foram utilizados em diversos dos componentes, porém certos problemas de sincronização surgiram durante a replicação do circuito e foi decidido a substituição para *flip flops*. Com essa modificação ao invés dos registradores armazenarem os valores durante o nível alto, eles apenas gravam durante a subida(ou descida)de *clock*, ajudando na sincronia entre componentes e evitando erros.

Como será discutido no capítulo 3.12, o circuito de controle foi uma das maiores diferenças sobre o circuito original. Isso se deve tanto a falta de informações nos documentos originais quanto a diferenças de implementação nos componentes. Apesar disso o resultado final da execução do conjunto de instruções é igual em ambas implementações.

Uma limitação menos significativa do Logisim é a falta de suporte para conectores que são simultaneamente de entrada e saída(conhecidas como *inout*). Este tipo de conexão tem uma certa importância para poder conectar barramentos a certas partes do circuito, porém pode ser simulada

utilizando uma entrada e uma saída, uma a frente da outra no pacote de componente como mostrado no capítulo 3.6.

3.2 Componentes e Simbologia do Logisim Evolution

O Logisim Evolution oferece componentes prontos para uso em circuitos lógicos. Eles podem servir desde funções lógicas básicas até operações mais complexas, como armazenamento de dados ou decodificação.

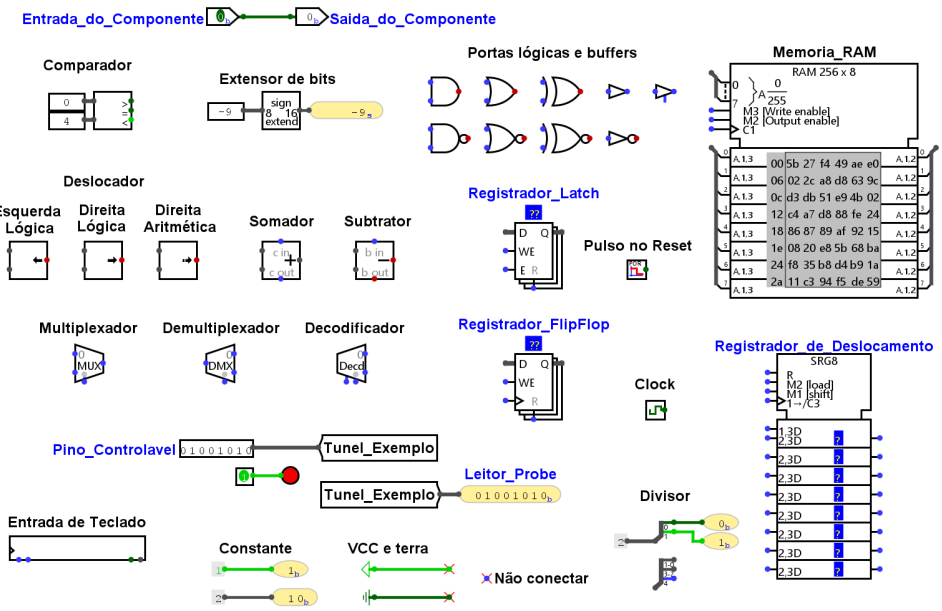

Figura 6 – Componentes do Logisim utilizados no trabalho

A Figura 6 mostra todos os componentes nativos utilizados para a criação do modelo de simulação. Muitos desses componentes podem ter suas propriedades modificadas, o que também pode alterar sua aparência, quantidade de saídas e entradas. No simulador, todas as operações lógicas são suportadas, além de *buffers* que apenas permitem a passagem de informações de um lado para outro, e *buffers tri-state*, que permitem a passagem de dados apenas quando ativos, ficando em alta impedância caso contrário.

Operações de deslocamento podem ser realizadas pelo mesmo componente, desde que suas propriedades sejam modificadas. Operações aritméticas também são suportadas; entretanto, neste trabalho, foi utilizado apenas o subtrator e o somador. O simulador também oferece multiplexadores e de-multiplexadores, que permitem a seleção de uma entre várias conexões para a entrada ou saída de dados. Há também o decodificador, que permite a decodificação de um número binário para um número decimal.

Além das operações comuns, o simulador suporta componentes que armazenam um certo valor até que a simulação reinicie, como os pinos, ou componentes que permanecem com o mesmo valor, como as constantes. Para conectar dois fios sem que eles estejam visivelmente conectados no simulador, podemos usar túneis; todos os túneis com o mesmo nome se comportam como se estivessem conectados ao mesmo fio. Para ler os valores, utilizamos um leitor, ou *probe*, que mostra qual é o valor do fio ou barramento ao qual está conectado. Esse valor pode ser verificado posteriormente em um histórico de valores.

Todas as conexões podem ter quatro estados diferentes, que são diferenciados pelas cores: verde claro (1), verde escuro (0), azul (?,Z ou alta impedância), e vermelho (X ou erro/curto). Todas essas cores podem ser identificadas em fios que possuem apenas 1 bit de dado sendo transmitido; no entanto, há a possibilidade de criar barramentos que utilizam múltiplos bits. No tema original, a cor para esses

barramentos é branca, mas neste trabalho foi alterada para cinza devido à troca da cor de fundo. O simulador possui componentes que podem dividir ou combinar de diversas formas um barramento entre vários outros.

Também é possível criar nossos próprios componentes, que podem ter sua aparência modificada. Neste trabalho, a aparência será referenciada como "pacote". O pacote pode conter entradas e saídas, que são definidas quando componentes específicos de entrada e saída são colocados dentro do circuito do componente. O pacote dos componentes pode conter caixas de texto que indicam o valor dos registradores dentro desses componentes.

3.3 Circuito de *clock*

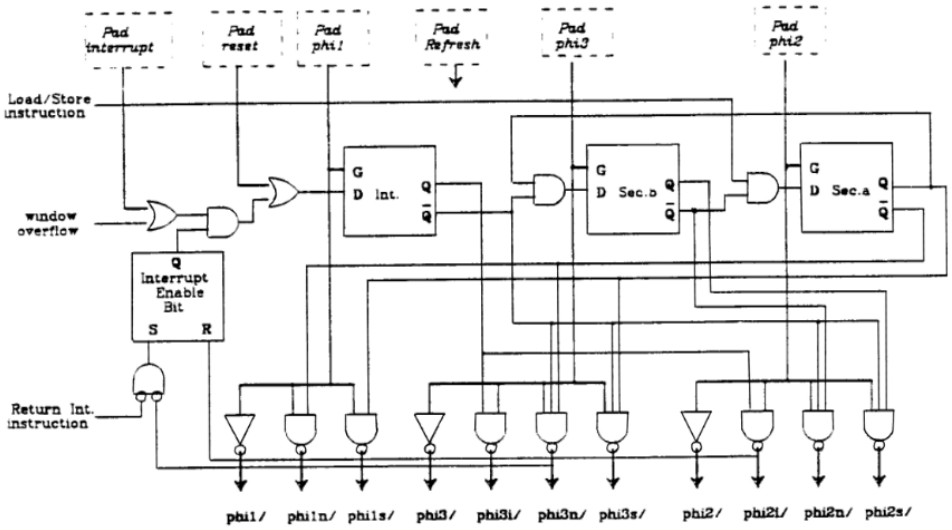

Figura 7 – Circuito de *clock* do RISC I (PEEK, 1983)

O circuito de *clock* do RISC I é responsável por controlar o funcionamento dos ciclos e fases da arquitetura descritos na seção 2.4. Ele possui uma entrada para cada fase e diversas saídas contendo a fase junto com o tipo de ciclo, este sendo normal(n), secundário(s) e de interrupção(i).

Para o controle de qual tipo de ciclo esta sendo executado, este circuito possui *flip flops* para armazenar se uma interrupção esta ocorrendo e se uma instrução de save/load esta ocorrendo. Para a interrupção ocorrer, há um *flip flop* que deve estar ativo, o *Interrupt_Enable*, que é ligado durante a instrução "RETI".

Na implementação realizada no Logisim, as saídas das fases não foram invertidas, e o mesmo ocorre na entrada das portas logicas conectadas ao *Interrupt_Enable*. Funcionalmente isso não fez diferença para a arquitetura, e os possíveis efeitos disso foram tratados no controlador e outros componentes que fazem uso direto dos sinais de fase.

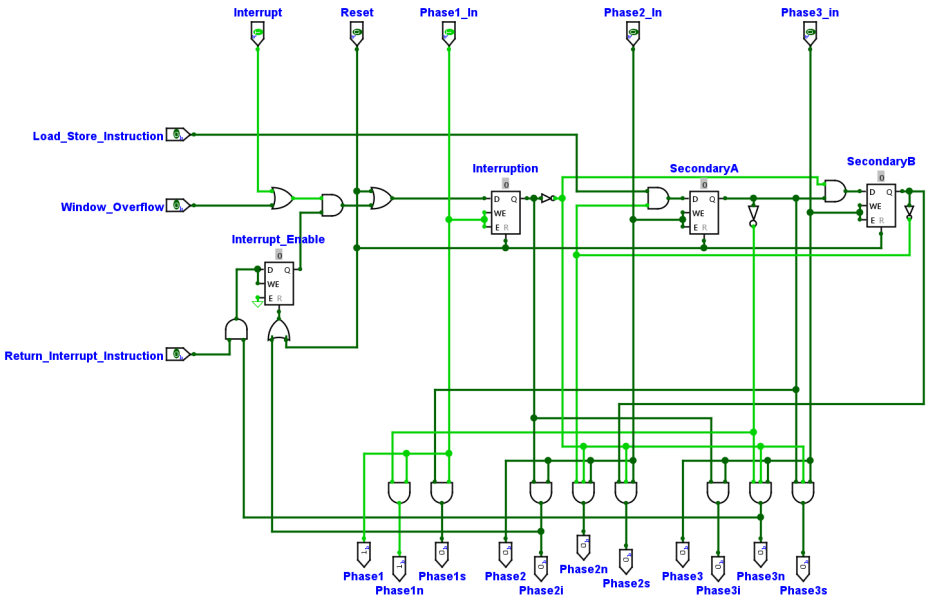

Figura 8 – Circuito de *clock* no Logisim

O pacote do componente visto na figura 9 mostra as entradas de fases por cima e as saídas à direita. À esquerda foram colocados os sinais de controle. Foram adicionados textos para cada um dos *flip flops*, ao lado da entrada ou saída que os representam.

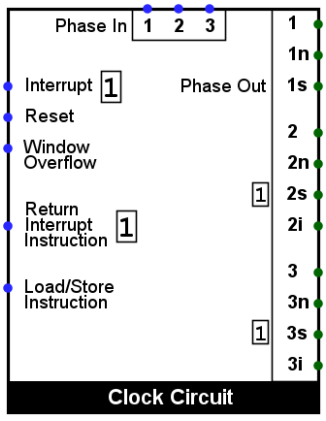

Figura 9 – Pacote do circuito de *clock* no Logisim

A entrada das fases do circuito de *clock* são geradas por um circuito a parte, fora do chip principal. Este gerador foi implementado com 3 flip flops D em cadeia que quando recebem o sinal de *reset* ficam com um ativo, que vai pro próximo em toda borda de descida do *clock*.

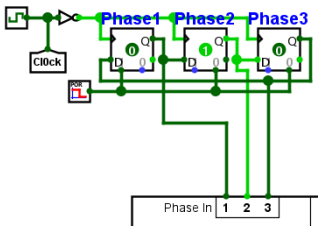

Figura 10 – Gerador de *clock* e fases no Logisim

3.4 Contador de Programa

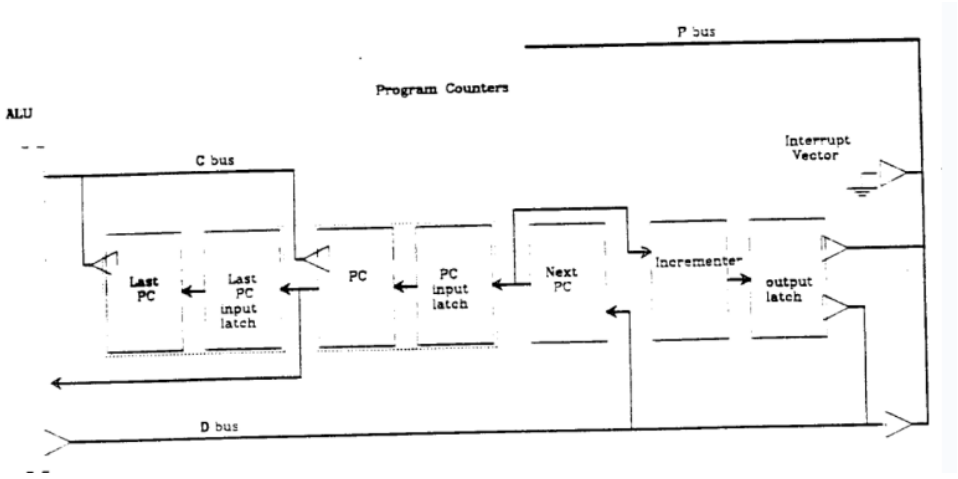

Figura 11 – Contadores de programa(PEEK, 1983)

O RISC I possui um contador de programa dividido em 3 partes: O próximo, o atual e o anterior. No circuito original foram usados *latches master-slave*(ou *flip flops*) tanto para o contador atual quanto para o anterior. O circuito original também faz uso de um *latch* para a saída do próximo contador de programa, que é usado no primeiro ciclo da *pipeline* para dizer qual é o próximo endereço a ser lido.

Para a implementação no Logisim, algumas modificações foram feitas. Os *flip flops* foram colocados diretamente para cada um dos contadores no lugar de *latches master-slave*. O *latch* de saída foi removido, o que causaria oscilação no próximo contador, porém com a troca de *latch* para *flip flop*, este problema foi resolvido.

Outro modificação que foi feita sobre o contador de programa foi a temporização dos sinais do controle. No trabalho de Peek é dito que:

> O último *latch* PC é carregado de seu *latch* de entrada a cada Phi1n. Durante cada Phi2n, o próximo PC é passado para o *latch* de entrada PC. O próximo PC é carregado do barramento D durante cada Phi3n e Phi3i. Os dados no barramento D vêm do incrementador para o fluxo normal de instruções; e da ALU para saltos. O incrementador opera de Phi3 para Phi1 e é válido por Phi2.

Porém ao seguir essa temporização, houveram problemas nas instruções "CALLI"e "GetLPC", além de que a atualização dessincronizada entre os contadores não faz sentido considerando seu contexto. Para corrigir isso, os 3 contadores se atualizam durante as 3n e 3i. Com essa alteração, também foi necessário trocar a saída do PC atual para o próximo PC, caso contrário, ao retornar de chamadas de "CALLI", o processador voltaria para a instrução de "CALL"ao invés da instrução seguinte.

Parte do controle de *byte* foi transferido para dentro do contador de programa. Este é o registrador *Byte_Offset_Reg*, que possui os 2 bits menos significativos de um endereço para instruções de *store*. Estes 2 bits dizem em qual byte do endereço a informação esta ou será armazenada, por exemplo, se estes bits forem 10, a informação estará armazenada a partir do terceiro byte.

O vetor de interrupção foi modificado para retornar o número hexadecimal 80000000. Essa modificação se deve parcialmente a forma com que este vetor funciona no RISC II, onde os endereços para tratamento de interrupções estão em torno deste valor(KATEVENIS, 1985, p. 198). Outro motivo é que ao reiniciar a simulação, o contador de programa utiliza o endereço 0, o que causaria problemas caso este estivesse ocupado com uma rotina de tratamento de interrupção. Esse vetor, diferentemente da implementação original, é passado para o próximo PC, fazendo com que a próxima instrução a ser decodificada e executada seja a da interrupção.

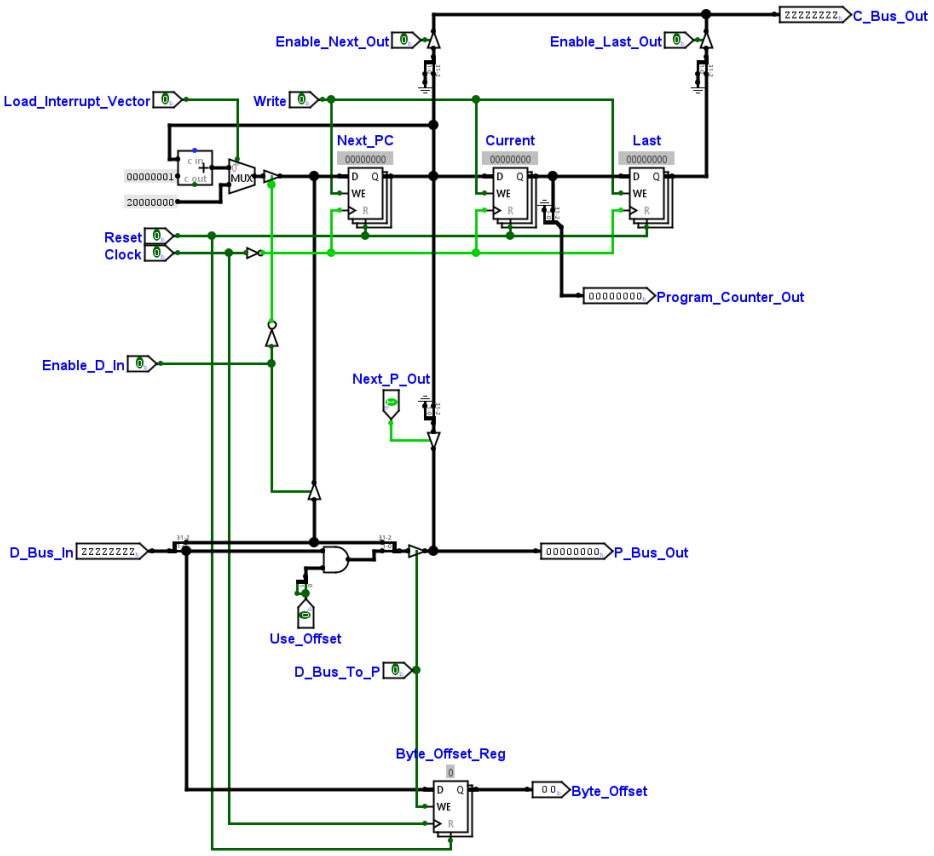

Figura 12 – Contadores de programa no Logisim

O componente mostra o valor de cada um dos contadores em tamanho grande, um seguido pra outro, com a conexão indicado por uma seta. O número de seleção do *byte* também foi adicionado ao lado de sua saída. Todas as entradas ficam à esquerda enquanto as saídas ficam à direita. em cima foram adicionados os sinais de *clock* e *reset*

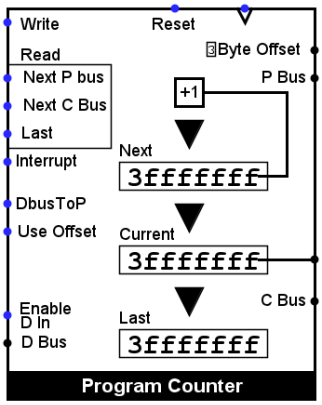

Figura 13 – Pacote dos contadores de programa no Logisim

3.5 Registrador Imediato

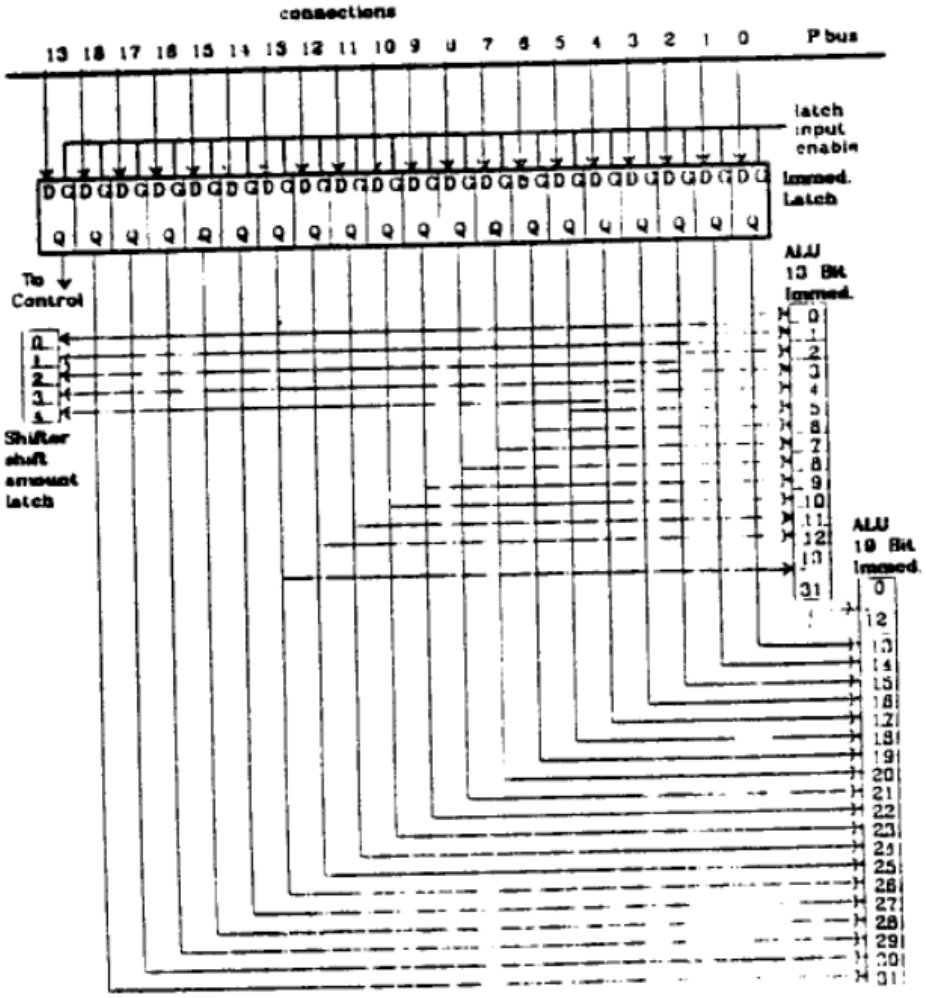

Figura 14 – Registrador imediato(PEEK, 1983)

Para a implementação do imediato foi utilizado um *flip flop* de 19 bits, similar a proposta do original. O diagrama original deixa a ideia de que a saída de 19 bits serve apenas para carregar os 19 bits mais significantes para a ula, porém como indicado por Katevenis (1985, p. 40), as instruções relativas utilizam os 19 bits e não necessariamente carregados nos bits mais significantes. Para a seleção do *Load High* foi adicionada uma entrada no circuito como mostrado na figura 15.

Katevenis também diz que a saída de 13 bits possui sinal estendido, que foi implementado no Logisim com um componente de extensor de bits, que também foi utilizado para estender os bits da saída de 19 bits.

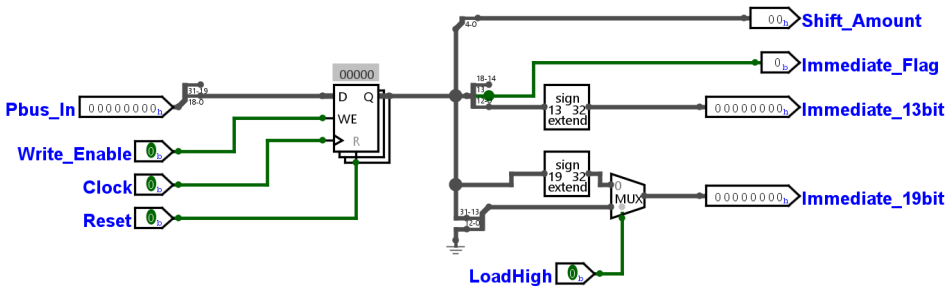

Figura 15 – Registrador imediato no Logisim

O imediato possui todas suas saídas à direita e sinais de controle acima e à esquerda. Adicionalmente foi adicionado um texto com o valor atual do registrador.

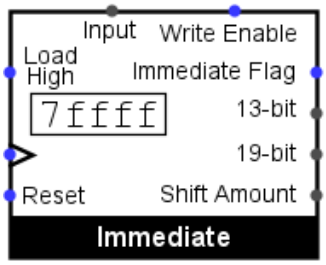

Figura 16 – Pacote do registrador imediato no Logisim

3.6 *Buffer* de entrada e saída

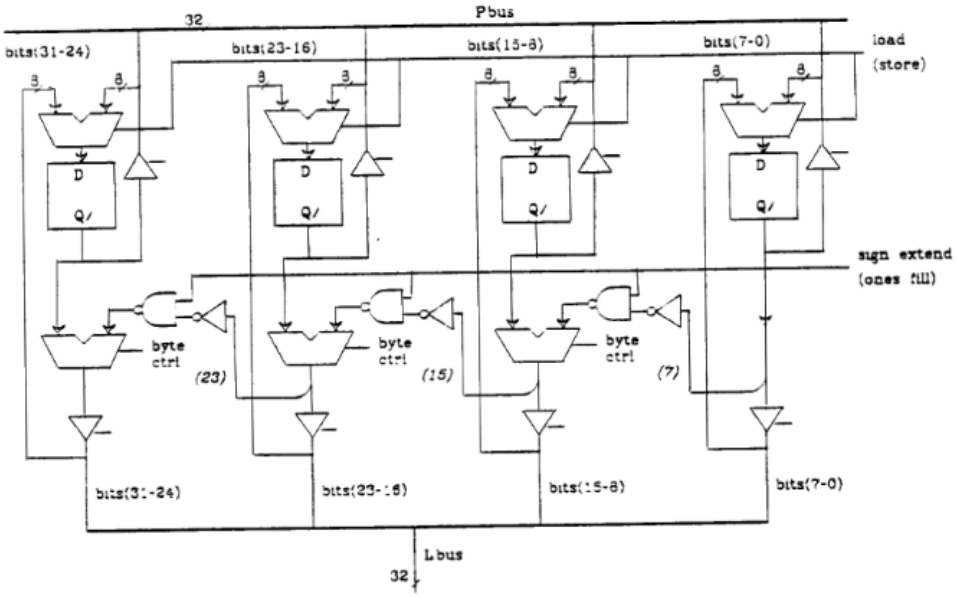

Figura 17 – *Buffer* de entrada e saída(PEEK, 1983)

Para poder haver a troca de informações entre o processador e dispositivos de entrada e saída precisamos de um componente que consiga tratar esta troca de informações. Este componente é o *buffer* de entrada e saída(*dataIO*), que é responsável por receber e enviar dados para a memória do sistema através do barramento P, e estender o sinal caso a informação recebida não ocupe os 32 bits.

No outro lado do *dataIO* temos o barramento L, que é uma conexão com o deslocador, que também é utilizado ao carregar dados da memória. O *dataIO* consegue através de *buffers tri-state* alterar qual barramento esta recebendo informações e qual esta enviando.

Na implementação do Logisim(figura 18), parte do controle de byte(um componente de controle separado na implementação original) foi adicionado ao circuito. Esta parte é o decodificador de byte, que recebe qual é o tamanho da informação(8, 16 e 32 bits) e qual é o seu deslocamento(8, 16 ou 24 bits), como por exemplo, um byte pode estar tanto nos 8 bits mais significativos quanto nos 8 menos significativos. A informação decodificada então controla os *muxes* que selecionam quais *bytes* vão receber a informação e quais serão estendidos. Dos que são estendidos, estes podem estendidos por sinal ou com zeros, o que é definido pela entrada de controle de dado com sinal(*Signed_Data*)

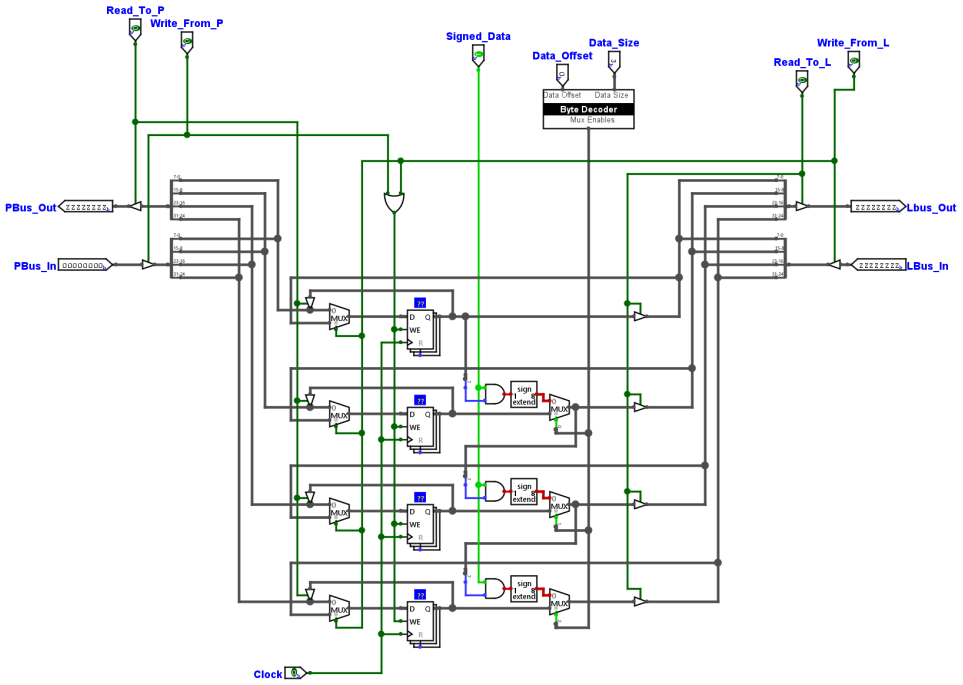

Figura 18 – Circuito do *buffer* de entrada e saída no Logisim

O componente do dataIO possui uma certa simetria, onde tanto em baixo quanto em cima há a entrada e saída dos barramento, um controle para leitura e outro para escrita. Como dito na seção 3.1, as conexões que são tanto entrada quanto saída precisam ser recriadas usando uma entrada e uma saída. Como o Logisim não permite que a entrada e saída fiquem no mesmo lugar, uma das duas conexões é colocada à frente da outra, permitindo que um fio passe por elas e as conecte. Os sinais de controle de *byte* ficaram à direita do componente enquanto o clock ficou à esquerda.

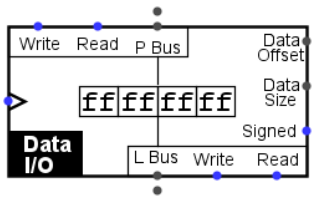

Figura 19 – Pacote do *buffer* de entrada e saída no Logisim

3.7 Unidade Lógica e Aritmética

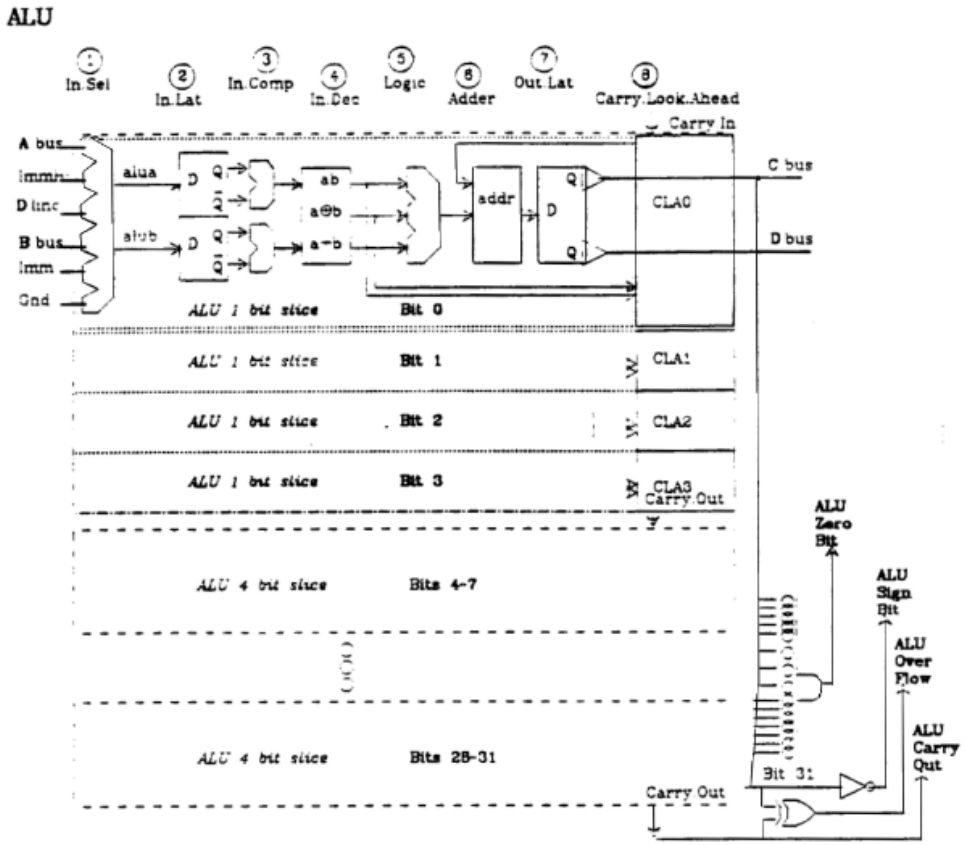

Figura 20 – Unidade lógica e aritmética(PEEK, 1983)

A Implementação original da Unidade Lógica e Aritmética(ULA) possui 7 entradas de valores(Barramentos A e B, imediato de 13 e 19 bits, contador de programa, conexão terra e *carry* in) e 6 saídas(barramentos C e D, e as *flags* zero, sinal, *overflow* e *carry out*). Ela pode ser separada entre 8 partes, porém para a implementação no simulador, houve uma simplificação nos circuitos, resultando em 6 partes:

1. Seleção de entrada: Na implementação do Logisim(figura 21), esta foi feita utilizando 2 *muxes*, um para a entrada do primeiro parâmetro, que escolhe entre o barramento A, o Contador de Programa e a conexão terra, enquanto o outro seleciona entre os imediatos e o barramento B.

2. *Latch* de entrada: Dois *latches*(*flip flops* no Logisim) recebem as entradas que foram selecionadas no passo anterior durante a descida de *clock* com o pino de escrita da ULA ligado, geralmente na fase 1.

3. Troca de operantes: Na implementação original esta parte seria um complementador, porém o trabalho do Peek não explica direito como o complemento dos operadores é utilizado, sem contar que não há nenhuma operação puramente de complementação documentada. No Logisim, esta parte é utilizada para trocar os operantes A e B, para assim realizar as operações de subtração onde o B deve vir primeiro. Para alterar a ordem dos operantes, foi designado o bit 4 da entrada de operação como controle dos *muxes*.

4. Cálculo de operações: Os operantes então passam por diversos circuitos que calculam as operações da ULA. Diferentemente da implementação original, a implementação desse trabalho inclui um bloco de adição e subtração diretamente nessa seção para simplificar o circuito mantendo a mesma funcionalidade. Os circuitos aritméticos possuem saídas de *carry out* que são selecionadas a partir de um *mux* controlado pelo segundo bit da entrada da operação.

5. Seleção do resultado: O resultado é então escolhido por um *mux* que recebe a entrada do operador como controle.

6. *Latch* de saída: Um *latch*, que também foi implementado com um *flip flop* no Logisim, salva o resultado da operação durante a fase 2 para poder ser finalmente lida na fase 3. As *flags* de resultado, comumente chamadas de CNZV, representam respectivamente *carry*, negativo, zero e *overflow*, e são apenas válidas após a escrita do *flip flop* de saída.

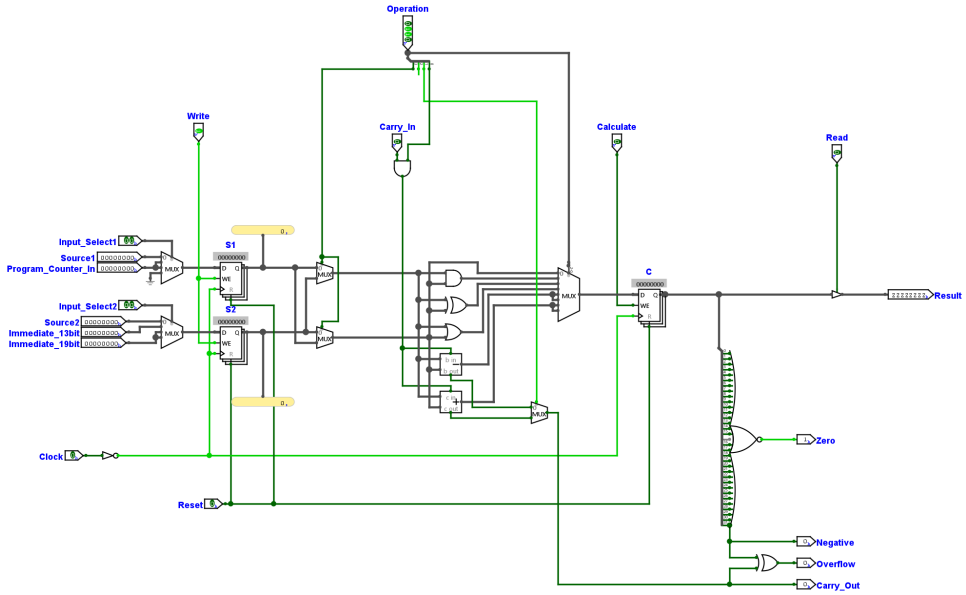

Figura 21 – Implementação da ULA no Logisim

Para a representação visual do componente, adotou-se o formato comumente utilizado em diagramas de arquiteturas, conforme ilustrado na Figura 22. As entradas da Fonte 1, o seletor correspondente e o *carry in* estão posicionados à esquerda, na parte superior da ULA. Na parte inferior esquerda, encontram-se a seleção da Fonte 2 e suas respectivas entradas. A seleção de operação, o *clock* e o *reset* estão centralizados no lado esquerdo. Os sinais de leitura, execução e escrita estão localizados acima do componente. Por fim, a saída das *flags* e do resultado final está à direita do componente.

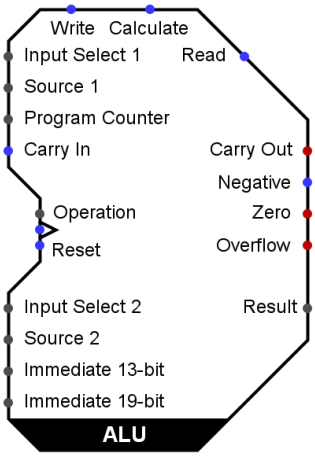

Figura 22 – Pacote da ULA no Logisim

Apesar de ser considerado, não foi adicionado nenhum texto que mostre o valor dos registradores da ULA em seu pacote. isso se deve a falta de espaço utilizável, que torna difícil uma inserção desses

textos sem aumentar o componente de tamanho. Futuramente isto pode ser reconsiderado, pois a exibição desses valores pode ser uma grande ajuda em passos de *debugging*

3.8 Deslocador

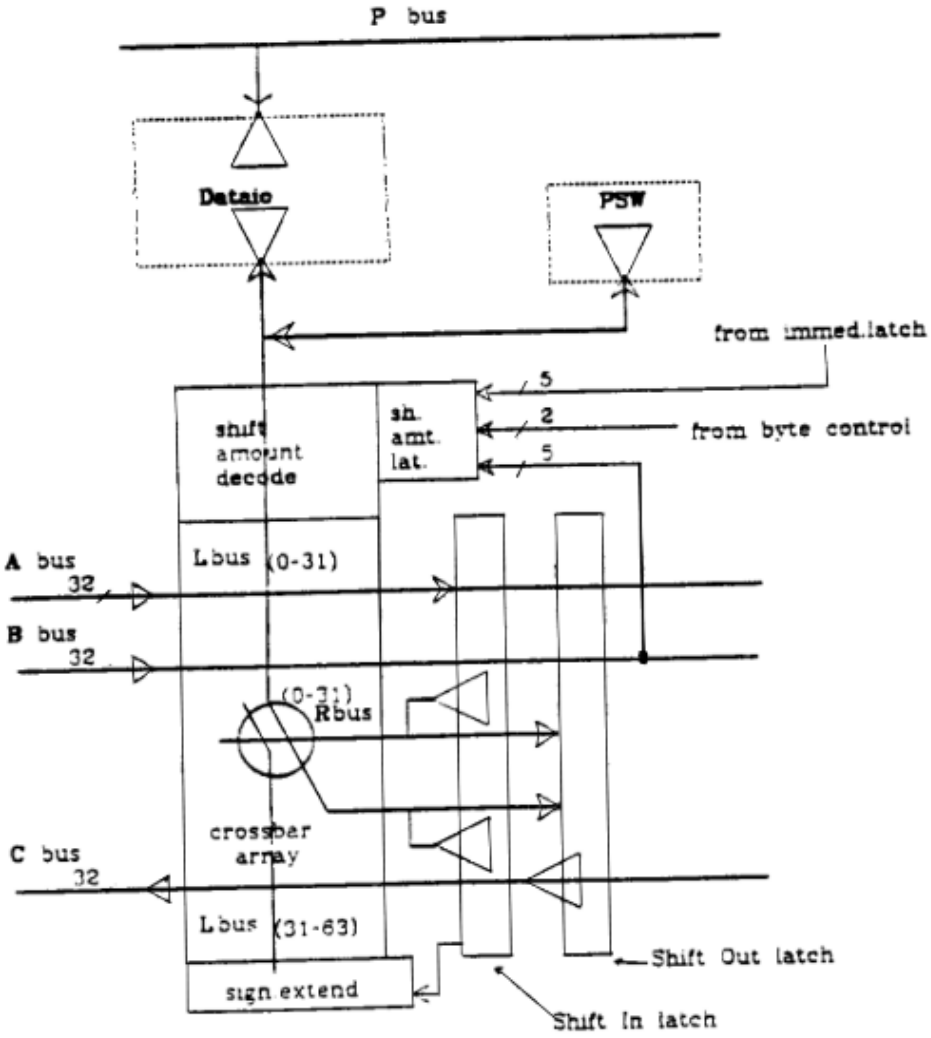

Figura 23 – Deslocador(PEEK, 1983)

O deslocador no RISC I possui uma importância não só para a possibilidade de realizar cálculos de deslocamento, mas também porque ele serve como parte da interface para exterior do processador, juntamente com o *dataIO*. Na implementação original ele possui dois barramentos internos, o barramento R(direita) e L(esquerda), que se conectam utilizando um matriz de conexões transversais, ou *crossbar array*.

Ao invés de recriar o *crossbar array*, foram utilizados diretamente os componentes de deslocamento de bits já prontos do *Logisim*. Isso permite uma visualização mais simplificada do circuito final. O barramento R foi omitido enquanto o L foi mantido explicitamente para conexões externas como a do *dataIO*.

Em instruções de carregamento, assim que uma informação é recebida, esta informação deve ser alinhada pelo deslocador caso ela não comece no primeiro byte. Para isso é realizado um deslocamento aritmético para a direita que alinha a informação ao primeiro byte, enquanto o sinal do valor é mantido.

Assim como a ULA, para haver sincronia de dados, os *clocks* dos registradores de entrada foram invertidos, porém não houve necessidade dessa inversão no registrador de saída.

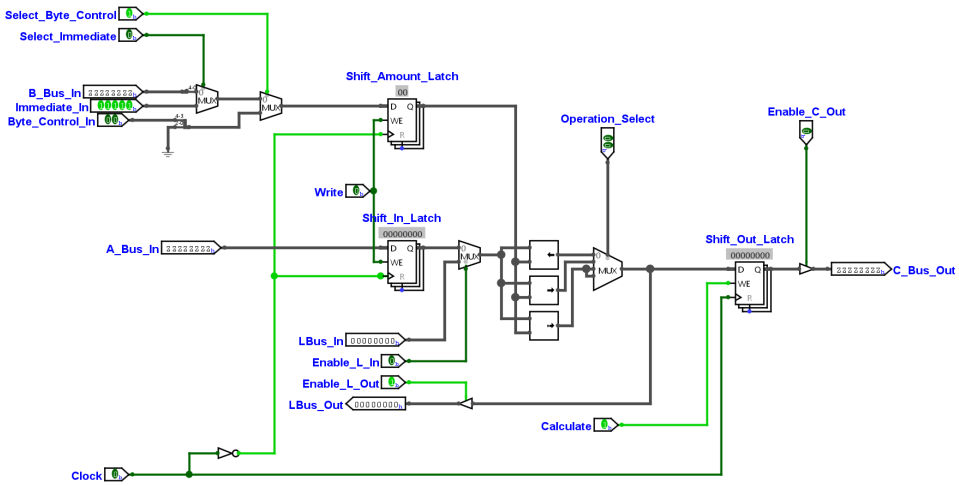

Figura 24 – Deslocador no Logisim

Para o pacote do deslocador foram adicionados textos com os valores dos registradores de entrada e saída, além da quantidade de deslocamento. linhas foram adicionadas para indicar algumas das conexões internas de entrada e saída. À direita foram colocados sinais de controles gerais com a saída para o barramento C na parte mais inferior. À esquerda há a seleção da operação e da entrada da quantidade de deslocamento, com a entrada A na parte mais inferior. No Topo do deslocador ficam o *clock* e a conexão de entrada e saída com o barramento L.

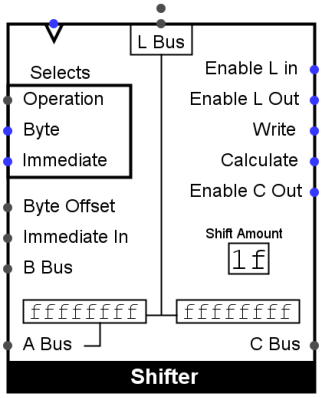

Figura 25 – Pacote do deslocador no Logisim

3.9 Arquivo de Registradores

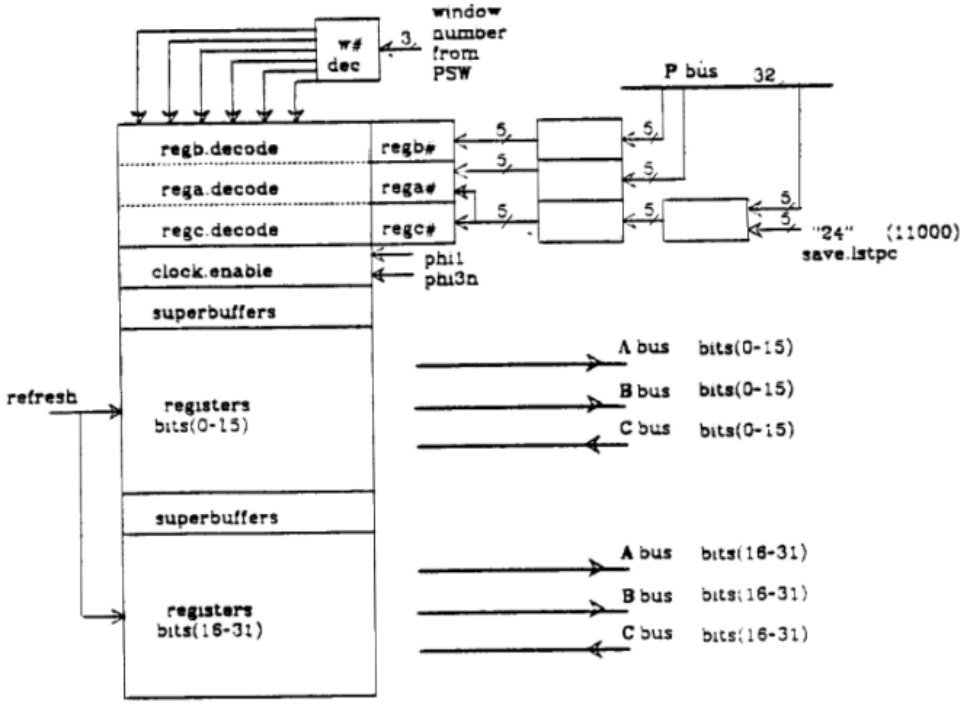

Figura 26 – Arquivo de registradores(PEEK, 1983)

O arquivo de registradores é uma das peças mais importantes da arquitetura RISC I, permitindo a simplicidade de seu funcionamento. Nos chips RISC I e RISC II, ele ocupa a maior parte do chip e armazena informações de forma similar a uma pilha (*stack*).

Os programas têm acesso a 32 registradores. O primeiro registrador é conectado ao terra, retornando 0 quando chamado e não sendo modificado quando utilizado como destino. 17 registradores são globais, e as últimas 14 conexões são dinâmicas, ligadas a diferentes grupos de registradores locais, chamados janelas. Essas janelas mudam a cada instrução de chamada e retorno, similar ao funcionamento de uma pilha. Os quatro primeiros registradores locais correspondem aos quatro últimos da janela anterior, permitindo o compartilhamento de parâmetros entre janelas.

O RISC I original implementa apenas 6 janelas, mas com pequenas modificações no circuito, é possível adicionar mais 2, como no RISC II. Na implementação realizada, foi utilizado o padrão de tamanho de janelas do RISC II, com apenas nove registradores globais, mas 16 locais, dos quais seis são compartilhados entre janelas. Isso permite que mais parâmetros sejam passados pelos registradores em chamadas de funções e retornos, reduzindo a necessidade de armazenamento em memória com instruções de *load* e *store*, acelerando o processamento.

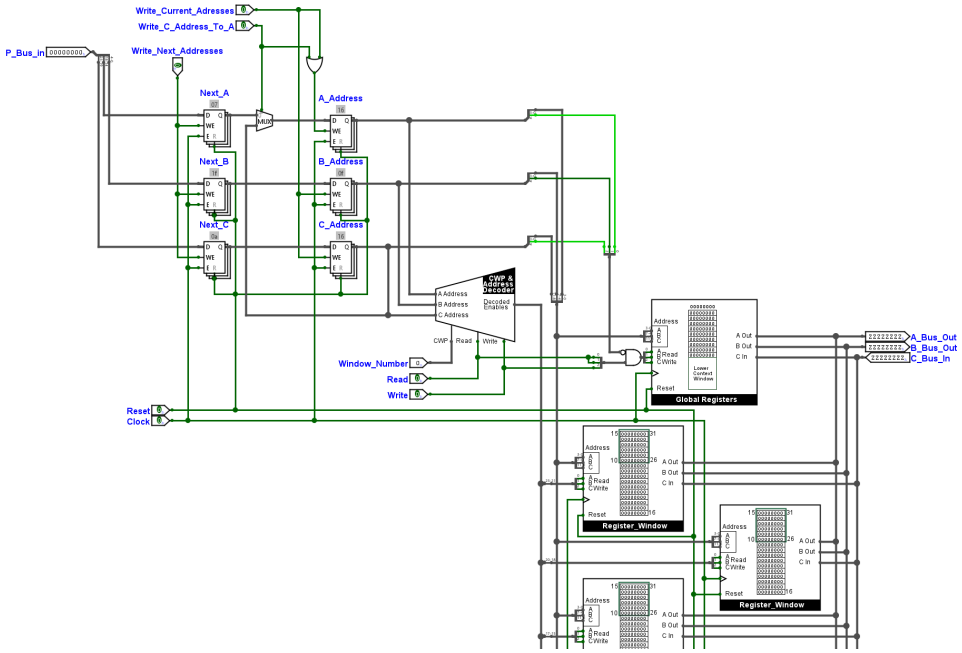

Figura 27 – Arquivo de registradores no Logisim

A figura 27 mostra uma parte do circuito to arquivo de registradores. A parte que não é mostrada é apenas a repetição dos 8 registradores locais. Algumas partes do circuito foram separadas em componentes próprios para simplificar a construção do modelo e melhorar o entendimento visual.

No RISC original, há 7 *latches* para armazenar os endereços durante as diversas fases da *pipeline*. No entanto, foi possível reproduzir a mesma funcionalidade utilizando apenas 6 *flip-flops*. Destes, 3 armazenam a instrução que está sendo lida da memória, enquanto os outros 3 armazenam os endereços que estão sendo executados.

Em instruções de *store*, o endereço C precisa ser passado para o endereço A, pois ele contém o registrador com a informação que deve ser gravada. Esta informação, por sua vez, deve ser passada como parâmetro para o deslocador. Para a passagem do endereço C para o endereço A, a saída do endereço C que está em execução é conectada à entrada do endereço A em execução através de um multiplexador.

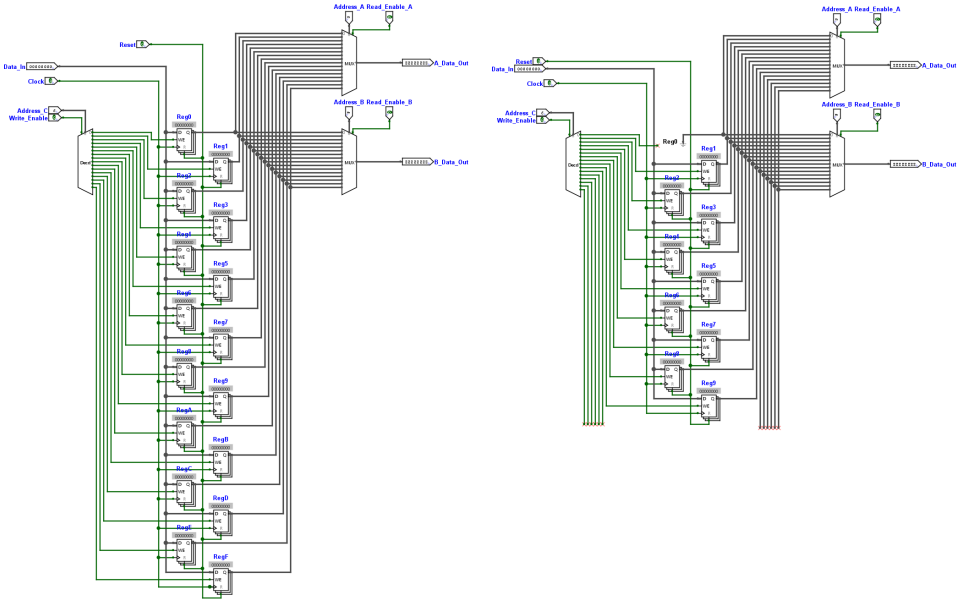

Figura 28 – À direita, uma célula de registradores locais; À esquerda, a célula contendo os registradores globais

A figura 28 mostra os circuitos de uma janela de registradores e os registradores globais. O circuito da janela recebe os 4 bits menos significativos dos endereços, que são utilizados em decodificadores e multiplexadores para a seleção dos registradores que serão lidos e escritos.

Para a seleção de qual registrador está sendo escrito, foi utilizado um decodificador que, ao receber o endereço e o sinal de escrita, ativa o pino de leitura de um dos 16 registradores da janela. Para a leitura de dados, foram utilizados 2 multiplexadores, um para cada barramento de leitura. Cada entrada dos multiplexadores recebe um registrador diferente e, ao receber o sinal de leitura com o endereço, a informação do registrador selecionado é retornada pelo componente. Caso não haja sinal de leitura, a conexão fica em alta impedância, similar a um *buffer tri-state*, o que permite conectar a saída de várias janelas nos barramentos A e B sem criar curtos.

Para a criação dos registradores globais foi reutilizado o layout do circuito da janela, porém os 6 últimos registradores foram removidos por serem os registradores locais da janela abaixo, enquanto o primeiro registrador foi substituído por uma conexão terra na saída, retornando sempre 0. Será mostrado depois que os sinais pros registradores da janela anterior ainda são passados para esse componente, porém, por não haver nenhuma conexão interna, nenhum dado é gravado ou retornado neste componente quando isso acontece.

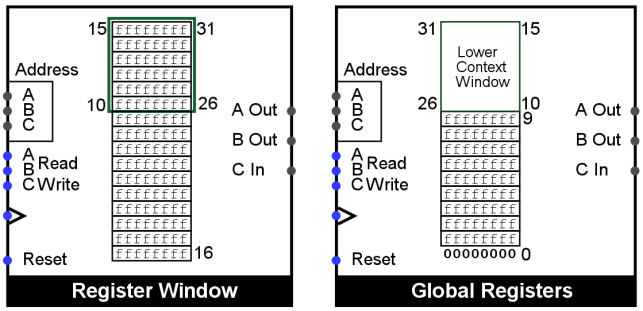

Figura 29 – Pacotes das células locais e globais das janelas de registradores.

Os pacotes dos componentes ficaram similares. À esquerda, o componente recebe os três endereços dos registradores sendo usados pela instrução atual. Abaixo dos endereços, há três sinais de leitura e escrita, que só se ativam na janela em uso. À direita, estão as conexões de troca de dados: as

saídas nos barramentos A e B, e a entrada no barramento C. No centro, é possível ver o valor de todos os registradores, com alguns endereços à direita (se for a janela atual) e à esquerda (se for a janela inferior), para ajudar na localização dos registradores compartilhados entre janelas.

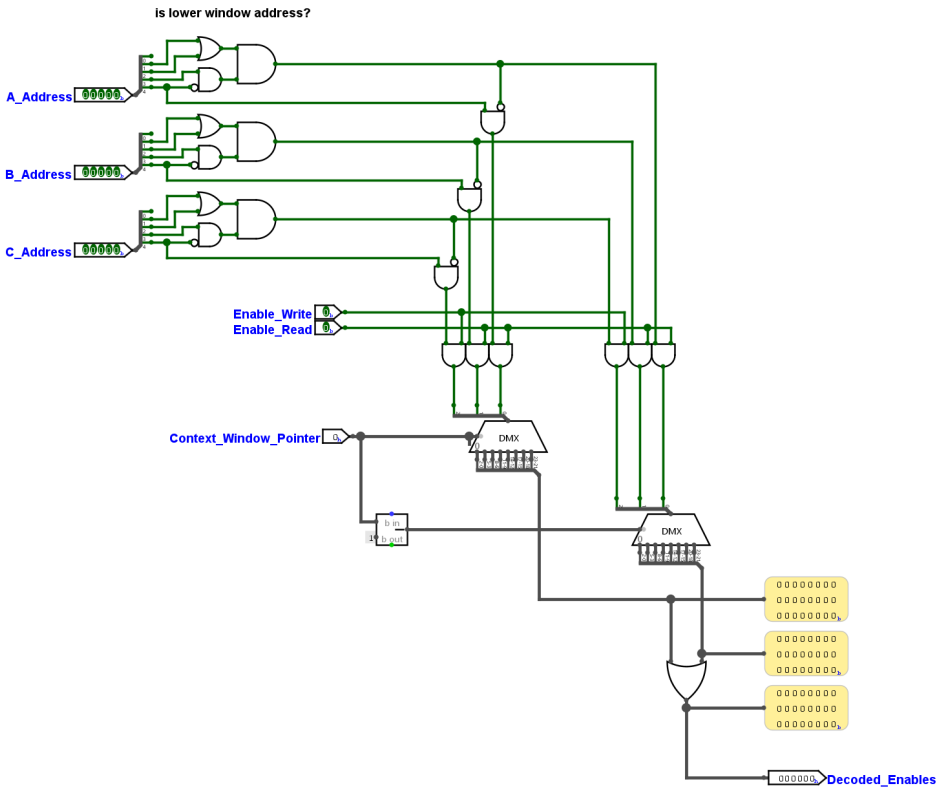

Figura 30 – Decodificador de endereço e do ponteiro da janela atual no Logisim

A figura 30 mostra o circuito do decodificador de endereços. Este componente recebe cada um dos endereços na instrução em execução e o número de janela, e retorna sinais controle para a janela que esta em atividade. Caso o endereço esteja abaixo de 10, é um endereço global e não ativa nenhuma das janelas. Caso o endereço esteja entre 10 e 15, o sinal de controle vai para a janela que esta abaixo da janela atual. Caso seja entre 16 e 31 então o sinal de controle vai para a janela atual.

Note que na figura 27, os sinais de controle para os registradores globais não passam pelo decodificador. Para ativar os registradores globais, é necessário apenas verificar se o bit mais significativo é 0. Caso seja 0, o registrador sendo manipulado é ou o local da janela anterior ou um dos globais. Como no componente de registradores globais as posições dos registradores locais não estão conectadas a nada, podemos enviar sinais de controle a este componente sem nos preocupar com a escrita simultânea em alguma janela local.

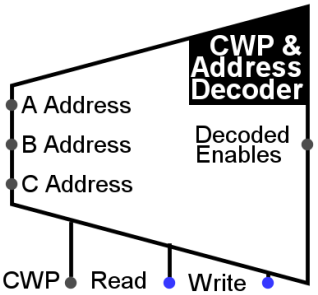

Figura 31 – Pacote do decodificador de endereço e do ponteiro da janela atual

O Pacote do circuito possui um formato similar ao de um de-multiplexador. A entrada dos endereços fica à esquerda enquanto a saída dos sinais de controle para cada janela fica à direita. Em baixo do componente temos a entrada do número de janela atual(CWP) e dos sinais de leitura e escrita.

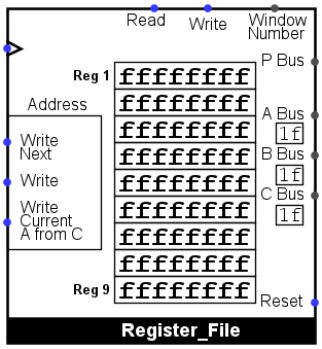

Figura 32 – Pacote do arquivo de registradores no Logisim

Por fim, temos o pacote que engloba o arquivo de registradores, similar ao componente de janelas individuais, com os registradores globais no centro. Nas entradas e saídas dos barramentos A, B e C à direita, foram adicionados textos exibindo os endereços dos registradores utilizados. Há uma entrada do barramento P acima dos outros barramentos, usada para obter os endereços dos registradores da instrução lida. Acima do componente, temos a entrada do número de janela e os sinais de controle para leitura e escrita dos dados armazenados. À esquerda, encontram-se os sinais de controle para a escrita de endereços: os próximos endereços, os endereços da instrução em execução e a escrita do endereço C no A.

3.9.1 Controlador de Janela

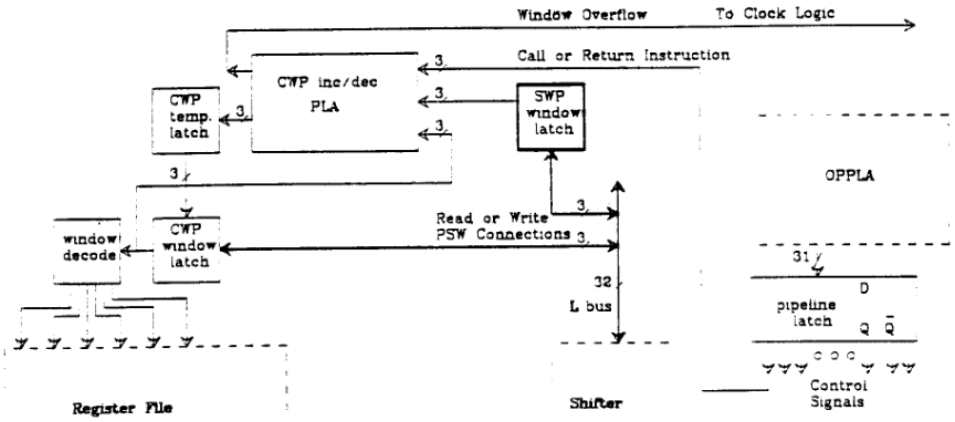

Figure 27: RISC I Window Controller Block Diagram

Figura 33 – Controlador de janela(PEEK, 1983).

O controlador de janela original (figura 33) serve para guardar não só o número da janela atual, mas também o menor número de janela que foi salvo para poder ser usado no tratamento de *overflows* de janela. Uma diferença na implementação realizada no Logisim foi que o número de janela aumenta em instruções de chamada e diminui em instruções de retorno. Isso foi feito para ficar mais alinhado com a ideia de uma pilha, além de fazer um pouco mais de sentido começar no 0 e a cada chamada aumentar o número. Esta modificação não gerou nenhuma diferença na funcionalidade da arquitetura, mas caso seja um problema futuro, pode ser facilmente revertido com poucas alterações.

Os valores dos registradores do número atual e do salvo da janela são passados para o barramento L durante toda fase 3n para poderem ser lidos pela palavra do processador. Ao executar uma instrução "PUTPSW", os valores dos números de janela são atualizados. Esta é a única forma de atualizar o número de janela salvo e essa instrução é executada durante rotinas de tratamento de *overflow* de janela.

No controlador de janela original, há um decodificador embutido que transforma o número binário em um fio para cada janela, porém como mostrado na seção 3.9, essa decodificação ocorre dentro do arquivo de registradores, então foi removido da implementação no simulador. Para o cálculo de acréscimo e decremento de janela foi utilizados componentes subtratores e somadores conectados a um multiplexador para a seleção da operação. Finalmente foi utilizado um componente de comparação para a detecção de *overflow* de janelas.

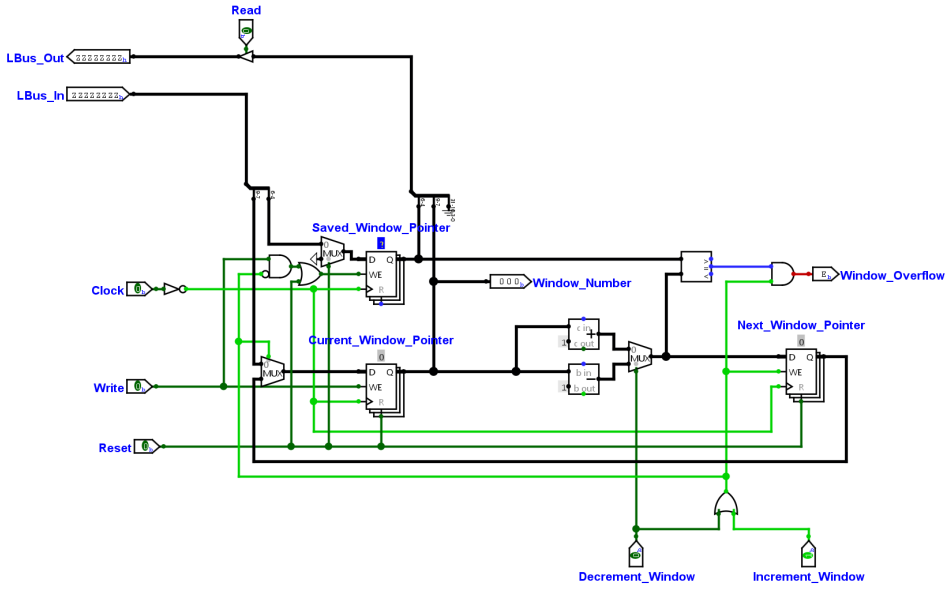

Figura 34 – Controlador de janela no Logisim

O pacote do contador possui os sinais de controle à direita junto com a conexão para o barramento L. A saída do número da janela fica a baixo e os sinais de *reset* e *clock* ficam à esquerda.

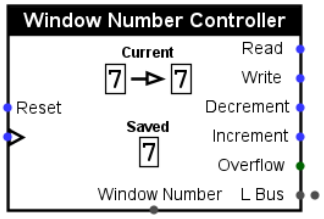

Figura 35 – Pacote do controlador de janela no Logisim

3.10 Palavra de Estado do Processador

A palavra de estado do processador precisa indicar duas coisas, os códigos de condição, que são retornados pela ULA, e os números de janela atual e salva. Na especificação original é utilizada uma palavra inteira(32 bits), mesmo que a informação salva ocupe apenas 10 desses bits, logo os outros 22 bits podem ser definidos pelo usuário ao utilizar a instrução "PUTPSW".

O trabalho de Peek não detalha a implementação deste componente, então foi realizada a implementação a partir das especificações necessárias para seu funcionamento. Um registrador de 32

bits se conecta a entrada e saída para barramento L, assim pode ser utilizado nas instruções que modificam ou leiam seu valor, além de poder se comunicar com o registrador de número de janela.

Quando a ULA realiza uma operação e a *flag* para mudar códigos de condição(SCC) esta ativa, o código CNZV é gravado no registrador. O funcionamento da *flag* SCC será explicada em mais detalhe no capítulo 3.12, mas podemos dizer que ela é gravada durante a busca de instrução e transmitida para o PSW durante a fase 3n. Nessa implementação, o número de janela é apenas gravado durante a ativação da *flag* SCC. Caso a instrução em execução seja "PUTPSW", esta grava o seu valor durante a fase 2n.

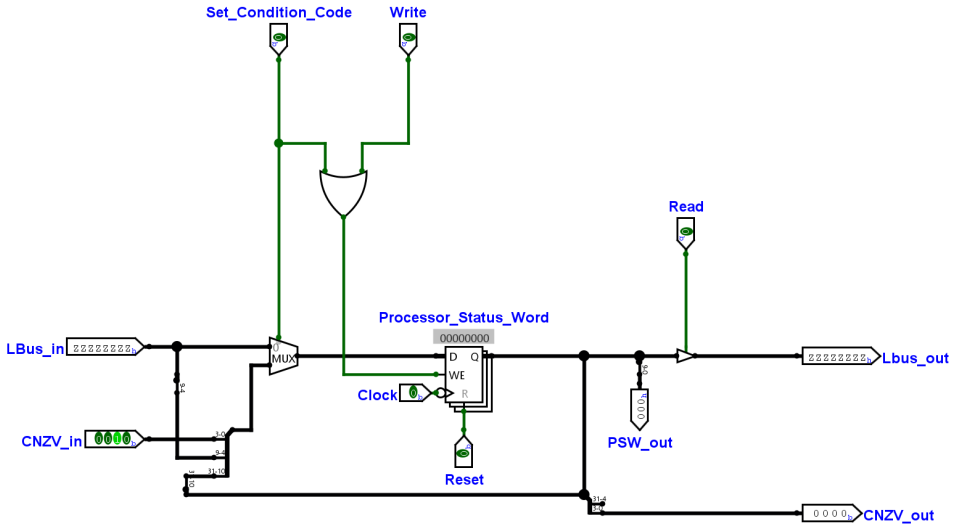

Figura 36 – Registrador de palavra de estado do processador no Logisim

O pacote do PSW possui sua conexão com o barramento L à esquerda, e o CNZV à direita. sinais de controle foram deixados em ambos os lados para manter o componente mais compacto. Adicionalmente, pela importância de seu valor, foi adicionado um texto grande que fica logo acima de seu título.

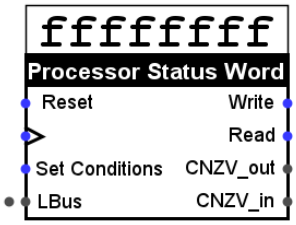

Figura 37 – Pacote do registrador de palavra de estado do processador no Logisim

3.11 Entrada e Saída

O RISC I pode se comunicar externamente com outros dispositivos através do barramento P, que se conecta diretamente a memória e dispositivos de entrada e saída. Além do barramento P, Alguns sinais de controle também devem ser utilizados para indicar quando uma leitura ou escrita devem ser realizadas, além de sinais para indicar qual é o tamanho do dado que esta sendo escrito. Finalmente, alguns sinais de temporização são utilizados para indicar quando o registrador de endereço da memória deve ser escrito, que ocorre durante as fases 1n e 3s.

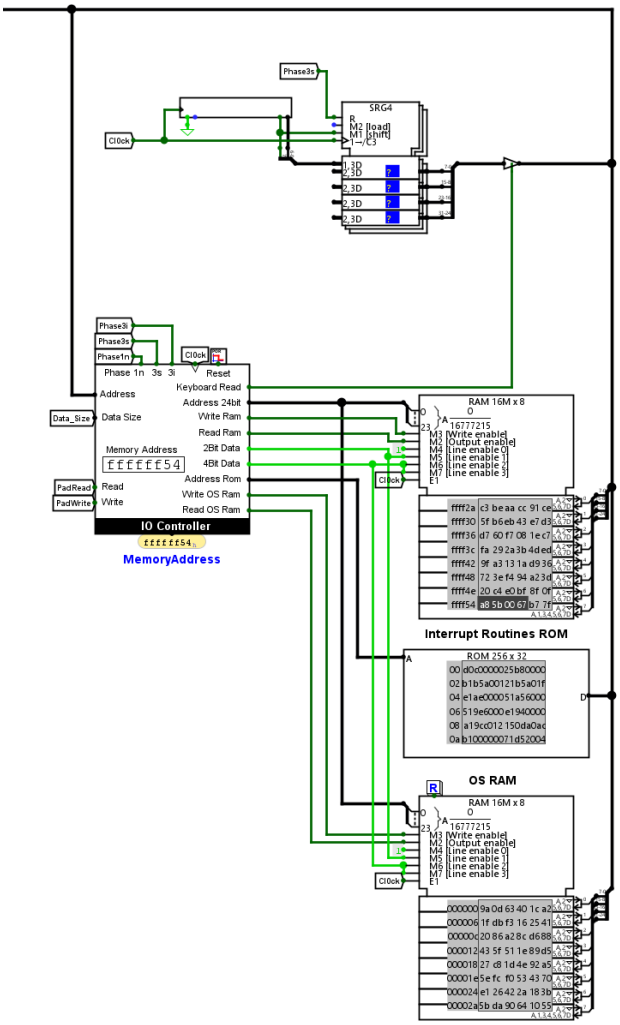

Figura 38 – De cima para baixo: Dispositivo de entrada, memória RAM, ROM de rotinas de interrupção, RAM para o sistema operacional

A Figura 38 mostra a memória RAM, um dispositivo rudimentar para leitura de dados pelo usuário, uma ROM que armazena rotinas para tratamento de interrupções e uma memória RAM que é utilizada para armazenar resultados dessas rotinas. Um controlador foi criado para lidar com estes dispositivos. Este controlador armazena o endereço atual e o decodifica para selecionar qual dispositivo esta sendo utilizado. O registrador de endereço contido no controlador é atualizado durante a fase 1n no ciclo de busca de instrução e durante a fase 3s no ciclo de *load* e *store* e durante a fase 3i em interrupções. O circuito do controlador pode ser visto na Figura 39.

A memória RAM foi criada a partir de um componente pronto do Logisim, permitindo que programas compilados sejam carregados diretamente nela. Cada endereço possui 1 byte de informação e, a cada leitura, 4 endereços são lidos simultaneamente. Estes podem ser truncados e alinhados pelo *dataIO* e pelo deslocador, caso a informação buscada seja menor que 32 bits. Durante a escrita, o endereço enviado para o registrador não precisa estar alinhado aos endereços das palavras completas (de 4 em 4), diferentemente da leitura. Além disso, um sinal do tamanho do dado sendo escrito deve ser enviado para a memória, podendo limitar a quantidade de endereços escritos a 1 ou 2 bytes.

A ROM é um componente simples, que recebe um endereço e devolve um valor. Caso o endereço esteja em alta impedância, então a saída também fica em alta impedância. Como só podem ser executadas instruções de *load*, cada célula da ROM possui todos 32 bits, contendo a informação completa de cada instrução. Nessa versão do modelo, só há uma rotina para tratar problemas de *overflow* e *underflow* de janela, porém futuramente novas rotinas podem ser adicionas. Junto com a ROM uma segunda memória RAM foi adicionada, porém esta é de uso exclusivo para as rotinas. Assim

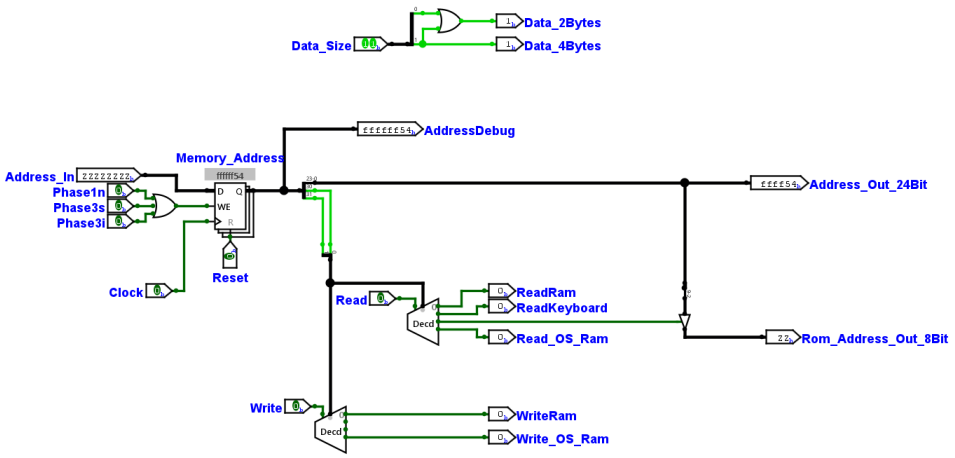

Figura 39 – Controlador dos dispositivos de entrada e saída.

que ocorre um *overflow* de janela, a janela seguinte deve ser salva nesta RAM, podendo ser restaurada após a rotina de *underflow*. Para funcionar corretamente é necessário que a o valor inicial da primeira célula desta memória seja 0xC0000000, o que indica que as janelas serão salvas logo após esta primeira célula.

O dispositivo de entrada foi criado a partir de dois componentes do Logisim: um para coletar entrada do teclado em código ASCII e um registrador de deslocamento para armazenar os 4 últimos caracteres digitados (*buffer* de entrada). A implementação deste dispositivo permite que dados escritos pelo usuário sejam lidos pelo processador sem a necessidade de alterar diretamente a memória RAM do sistema, oferecendo uma interface interativa. No entanto, a implementação possui várias limitações, como a falta de uma limpeza do *buffer* após a leitura, a ausência de mecanismos para detectar quando não há novas informações para serem lidas e o tamanho limitado do *buffer*. Este dispositivo pode ser aprimorado em versões futuras deste trabalho. No momento da criação deste relatório, ele serve mais como uma prova de conceito.

3.12 Controle

O circuito de controle (figura 40) foi uma das partes mais diferentes da implementação original pela falta de informação no trabalho original. Apesar disso, foi possível replicar a mesma funcionalidade do original utilizando alguns dos conceitos que foram apresentados de forma geral.

Foi utilizado um registrador que armazena a próxima instrução a ser executada na pipeline, que se conecta a 3 registradores referentes a instrução em execução, estes sendo o registrador do código da instrução(*opcode*), um que armazena qual código de condição que esta sendo usado caso seja uma instrução "JMP", e o bit que diz se a ULA deve gravar no PSW. Além desses registradores, também há um registrador CNZV que copia os códigos de condição da PSW.

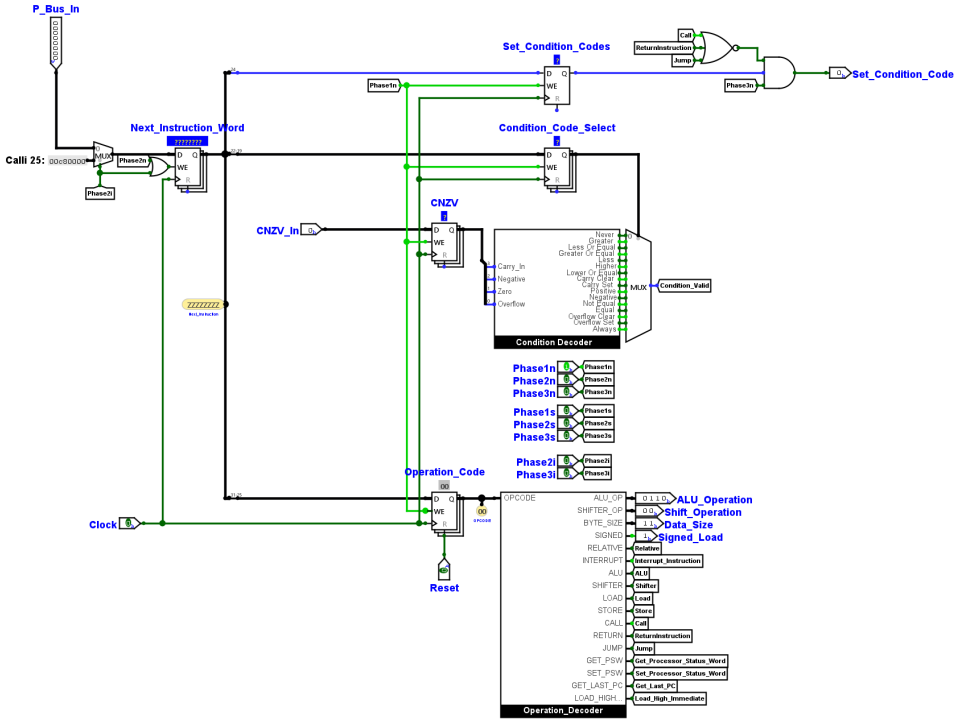

Figura 40 – Registradores e decodificadores do controlador.

A informação armazenada pelos registradores passa então por uma decodificação em 2 partes. A primeira separa o *opcode* em sinais que representam informações sobre qual tipo de instrução esta em execução. Esses sinais podem ser vistos no componente *Operation Decoder* na figura 40, e as suas ativações podem ser vistas na tabela 3. Além da decodificação do *opcode*, também é realizada a decodificação do código de condição, resultando em 16 sinais diferentes que são escolhidos por um multiplexador controlado pelo registrador de seleção do código de condição.

Após a primeira parte da decodificação, vem a segunda parte, que combina os sinais gerados com os sinais das fases, gerando sinais compatíveis com os que cada componente individual recebe. Estes sinais e suas lógicas podem ser vistas na figura 41.

	Entrada	Parâmetros							Instruções									
Mnemonico	OPCODE[6..0]	ALU_OP[3..0]	SHIFTER_OP[1..0]	BYTE_SIZE[1..0]	SIGNED	RELATIVE	INTERRUPT	ALU	SHIFTER	LOAD	STORE	CALL	RETURN	JUMP	GET_PSW	SET_PSW	GET_LAST_PC	LOAD_HIGH_IMMEDIATE
CALLI	0000XXX	0110	00	11	1	0	1	0	0	0	0	1	0	0	0	0	0	0
CALL	000100X	0110	00	11	1	0	0	0	0	0	0	1	0	0	0	0	0	0
JMP	000101X	0110	00	11	1	0	0	0	0	0	0	0	0	1	0	0	0	0
CALLR	000110X	0110	00	11	1	1	0	0	0	0	0	1	0	0	0	0	0	0
JMPR	000111X	0110	00	11	1	1	0	0	0	0	0	0	0	1	0	0	0	0
SLL	0010X0X	0110	00	11	1	0	0	0	1	0	0	0	0	0	0	0	0	0
GETPSW	0010X1X	0110	00	11	1	0	0	0	0	0	0	0	0	0	1	0	0	0
SRL	001100X	0110	01	11	1	0	0	0	1	0	0	0	0	0	0	0	0	0
PUTPSW	001101X	0110	00	11	1	0	0	0	0	0	0	0	0	0	0	1	0	0
SRA	00111XX	0110	10	11	1	0	0	0	1	0	0	0	0	0	0	0	0	0
ILLEGAL	01000XX	0110	00	11	1	0	0	0	0	0	0	0	0	0	0	0	0	0
LDBU	0100100	0110	00	00	0	0	0	0	0	1	0	0	0	0	0	0	0	0
LDRBU	0100101	0110	00	00	0	1	0	0	0	1	0	0	0	0	0	0	0	0
LDBS	0100110	0110	00	00	1	0	0	0	0	1	0	0	0	0	0	0	0	0
LDRBS	0100111	0110	00	00	1	1	0	0	0	1	0	0	0	0	0	0	0	0
LDW	01010X0	0110	00	11	1	0	0	0	0	1	0	0	0	0	0	0	0	0
LDRW	01010X1	0110	00	11	1	1	0	0	0	1	0	0	0	0	0	0	0	0
LDSU	0101100	0110	00	01	0	0	0	0	0	1	0	0	0	0	0	0	0	0
LDRSU	0101101	0110	00	01	0	1	0	0	0	1	0	0	0	0	0	0	0	0
LDSS	0101110	0110	00	01	1	0	0	0	0	1	0	0	0	0	0	0	0	0
LDRSS	0101111	0110	00	01	1	1	0	0	0	1	0	0	0	0	0	0	0	0
STS	01100X0	0110	00	01	1	0	0	0	0	0	1	0	0	0	0	0	0	0
STRS	01100X1	0110	00	01	1	1	0	0	0	0	1	0	0	0	0	0	0	0
STB	01101X0	0110	00	00	1	0	0	0	0	0	1	0	0	0	0	0	0	0
STRB	01101X1	0110	00	00	1	1	0	0	0	0	1	0	0	0	0	0	0	0
STW	0111XX0	0110	00	11	1	0	0	0	0	0	1	0	0	0	0	0	0	0
STRW	0111XX1	0110	00	11	1	1	0	0	0	0	1	0	0	0	0	0	0	0
AND	10000XX	0001	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
XOR	10001XX	0010	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
OR	1001XXX	0011	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
SUB	101000X	0100	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
SUBC	101001X	0100	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
SUBR	101010X	0101	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
SUBCR	101011X	1100	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
ADD	10110XX	0110	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
ADDC	10111XX	0111	00	11	1	0	0	1	0	0	0	0	0	0	0	0	0	0
RET	11000XX	0110	00	11	1	0	0	0	0	0	0	0	1	0	0	0	0	0
RETI	11001XX	0110	00	11	1	0	1	0	0	0	0	0	1	0	0	0	0	0
GETLPC	1101XXX	0110	00	11	1	0	0	0	0	0	0	0	0	0	0	0	1	0
LDHI	111XXXX	0000	00	11	1	0	0	0	0	0	0	0	0	0	0	0	0	1

Tabela 3 – Tabela de decodificação de *opcode*

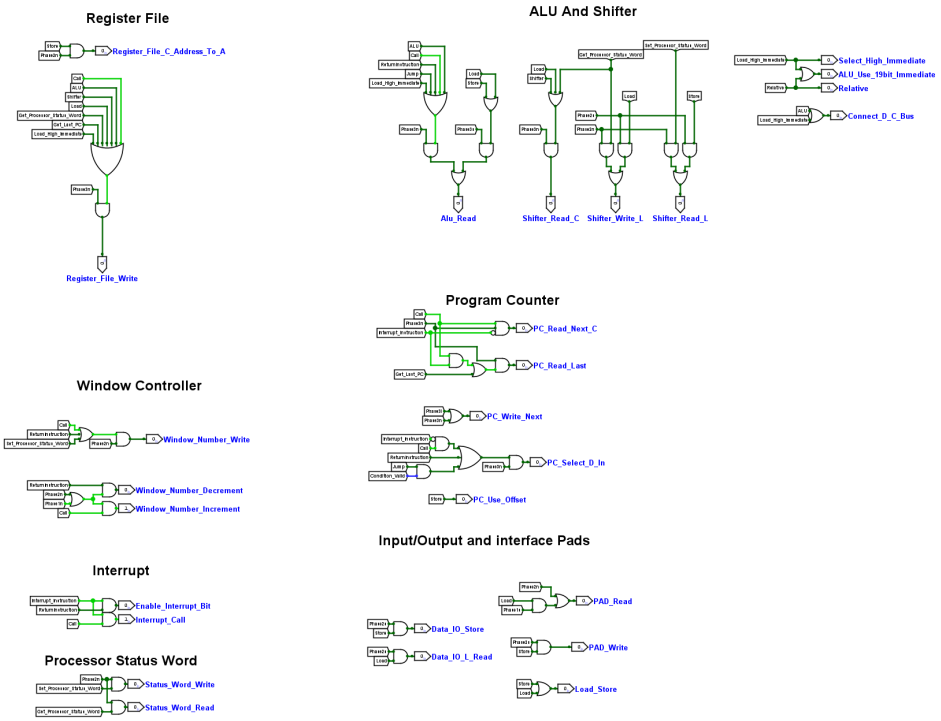

Figura 41 – Geração dos sinais de controle finais.

No pacote do componente é possível ver a entrada de todas fases na parte superior, junto com o *clock* e a entrada do barramento P. todas as saídas de controle ficaram nas laterais, junto com a entrada do CNZV na parte inferior esquerda, enquanto o *reset* fica em baixo. Para ajudar na organização os sinais foram agrupados em casos onde eles controlam o mesmo componente. Os registradores que guardam informações da instrução em execução tem seu valor exposto.

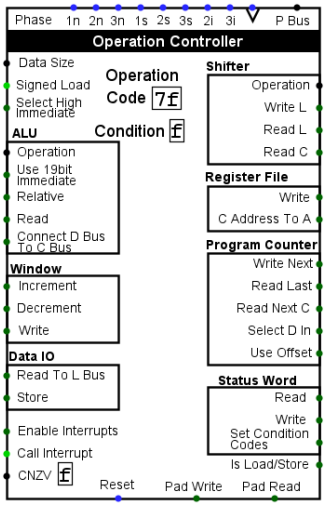

Figura 42 – Pacote do controlador principal no Logisim

3.13 Integração final

Para a conexão de todos os componentes algumas regras foram seguidas. Todos sinais do controlador são passados para os componentes utilizando componentes de portal para manter a visualização do circuito mais limpa, e a mesma regra vale para todos os sinais de tempo que se conectam diretamente aos componentes. Conexões de barramento devem ficar explicitas, e sinais de controles gerado por componentes que não são o controle também ficam explicitas, como por exemplo o *overflow* de janela. Um espaçamento mínimo foi dado para que os componentes não fiquem muito

próximos, e qualquer conexão, quando possível, não pode passar sobre nenhuma outra. Componentes que tem seus valores gravados antes de serem lidos não precisam de sinal de *reset*, os que precisam tem sua entrada de *reset* conectada a um componente que gera um sinal positivo por x tempo toda vez que a simulação é reiniciada.

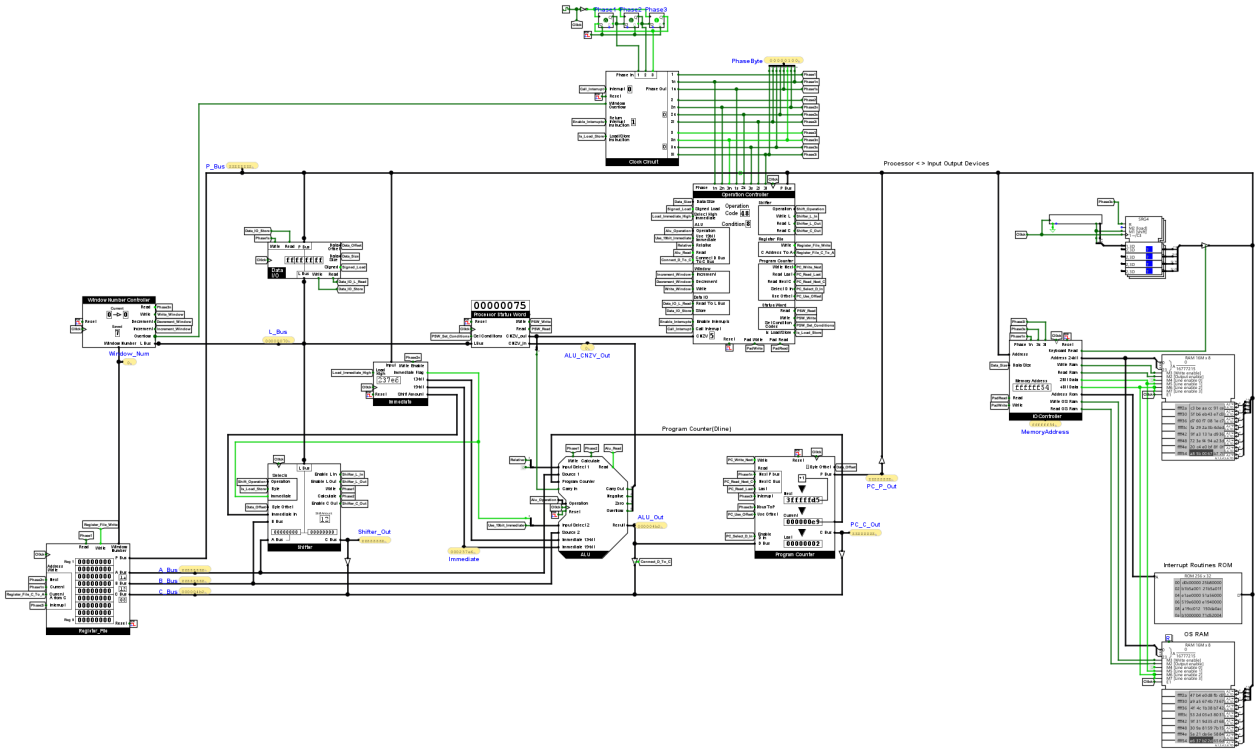

Figura 43 – Circuito do RISC I completo

4 Assembler

4.1 Funcionamento Geral

Um assembler foi criado nesse trabalho para facilitar o trabalho de criação de programas para o modelo de simulação. A estrutura do código do assembly é bem simples, e seu funcionamento bem rudimentar, porém funcional. Ele funciona a partir de 6 fases: Leitura e divisão do código, remoção de espaços brancos e comentários, leitura e gravação de rótulos(*labels*), leitura e gravação de palavras, tradução das instruções e geração da saída.

4.1.1 Leitura e Divisão do Arquivo

A primeira parte é bem simples. Primeiro, o arquivo é lido do sistema usando as classes `System.IO.File` e `System.IO.StreamReader`, guardando a informação lida em uma *string*. Após essa leitura, o método `Assemble()` da classe do nosso *assembler* é chamado, passando a string que foi lida e a largura de endereço e dados que definirá a formatação de saída.

```
1  var fileContents = File.Open(targetFile, FileMode.Open);
2  var reader = new StreamReader(fileContents);
3  var input = reader.ReadToEnd();
4  string output = "";
5  try
6  {
7      output = RiscAssembler.Assembler.Assemble(input,
           int.Parse(configSettings["bitWidth"]),
           int.Parse(configSettings["addressWidth"]));
```

```
}
```

Dentro do nosso método `Assemble()`, a *string* da entrada é então dividida em uma lista de strings onde cada uma representa uma linha. Adicionalmente, é criado um vetor de *uint* que representa a porção da memória que será enviada para o simulador, contendo todo o código compilado do nosso programa.

```
private static readonly string[] separator = ["\r\n", "\r", "\n"];
[...]
public static string Assemble(string input, int bitWidth, int addressWidth)
{
    var lines = input.Replace("\t", "").Split(separator, StringSplitOptions.None);
    var memory = new uint[2 << addressWidth];
    var linesList = lines.ToList();
```

4.1.2 Remoção de Espaços Brancos e comentários

Após a divisão das linhas, estas são passadas para um método de limpeza que itera sobre cada linha removendo quaisquer comentários e linhas vazias.

```
private static void RemoveCommentsAndBlankLines(List<string> lines)
{
    for (int currentLine = lines.Count - 1; currentLine >= 0 ; currentLine--)
    {
        var commentIndex = lines[currentLine].IndexOf("//");
        if (commentIndex >= 0) lines[currentLine] =
            lines[currentLine][0..(commentIndex)];
        if (lines[currentLine] == "")
        {
            lines.RemoveAt(currentLine);
        }
    }
}
```

4.1.3 Leitura de Rótulos

Os rótulos, ou *labels*, são importantes para facilitar o trabalho de referenciar linhas especificas de código em outras. Ao declarar um rótulo, este fica vinculado a linha seguinte, que pode ser referênciada em outras linhas apenas chamando pelo nome do rótulo. Para declarar e chamar um rótulo, basta que ele seja colocado seguido de dois pontos(:). No assembler todos os rótulos são salvos em um dicionário onde a chave é o nome do rótulo o valor é o endereço onde a próxima linha de código será gravada.

```
Dictionary<string, uint> labels = ExtractLabels(linesList);
[...]
private static Dictionary<string, uint> ExtractLabels(List<string> lines)
{
    Dictionary<string, uint> labels = [];
    for (int currentLine = 0; currentLine < lines.Count; currentLine++)
    {
        if (!lines[currentLine].StartsWith(':')) continue;
        labels.Add(lines[currentLine], (uint)currentLine << 2);
        lines.RemoveAt(currentLine);
        currentLine--;
    }
    return labels;
}
```

4.1.4 Leitura de Palavras

Para a leitura de palavras, que correspondem às variáveis da nossa linguagem assembly, é realizado um processo semelhante ao da leitura de rótulos. A diferença é que não queremos apagar as

linhas da nossa lista, pois estas serão gravadas na memória. É importante notar que o RISC I suporta não apenas palavras (32 bits), mas também *short* ou *half* (16 bits) e byte (8 bits). No entanto, nesta versão do *assembler*, não houve a necessidade de implementar esses tipos. Futuras versões poderão incluir a adição desse recurso, mas, por enquanto, o código contém apenas uma declaração desses tipos para futura implementação.

```
Dictionary<string, uint> words = ExtractVariables(linesList);
[...]
private static Dictionary<string, uint> ExtractVariables(List<string> lines)
{
    Dictionary<string, uint> words = [];
    for (int currentLine = 0; currentLine < lines.Count; currentLine++)
    {
        var parts = lines[currentLine].Split(' ');
        switch (parts[0])
        {
            case ".word":
                words.Add(parts[1], (uint)currentLine << 2);
                break;
            case ".short":
            case ".half":
            case ".byte":
                throw new NotImplementedException();
        }
    }
    return words;
}
```

4.1.5 Tradução das Instruções

Esta é a parte mais longa e complexa do *assembler*, porém segue um padrão básico para a tradução de cada parâmetro. Primeiro a linha é dividida em várias partes utilizando os caracteres em branco entre os parâmetros, e após isso cada parâmetro é decodificado e escrito na "memória"utilizando operações *bitwise*.

```
for (uint currentLine = 0, address = 0; currentLine < lines.Length; currentLine++,
    address++)
{
    var parts = lines[currentLine].Split(' ');
```

A tradução se inicia com um *loop for* que itera sobre cada linha, dividindo a instrução em seus diversos parâmetros.

```
if (parts[0].StartsWith('.'))
{
    if (parts[2].StartsWith('#'))
    {
        memory[address] = uint.Parse(parts[2][1..^0],hexStyle);
    }
    else if (parts[2].StartsWith(':'))
    {
        memory[address] = labels[parts[2]];
    }
    else
    {
        memory[address] = uint.Parse(parts[2],intStyle);
    }
    continue;
}
```

Após a separação, verifica-se se a linha começa com ".", significando que é uma declaração de uma variável. Caso seja, a terceira parte da linha pode ser um valor hexadecimal (começando com "#"), um endereço de um rótulo (começando com ":") ou um inteiro. Este valor é então gravado no endereço de memória, e o resto do *loop* é pulado para a leitura da próxima linha.

```
if (parts[0][^1] == '*')
{
    parts[0] = parts[0][0..^1];
}
else
{
    memory[address] |= 1 << 24;
}
```

As instruções do RISC I contém uma *flag* no bit 24, que diz se a instrução deve gravar os códigos de condição. No código assembly criado, esta *flag* é ativada por padrão, e caso seja necessário desativa-la, um asterisco(*) pode ser colocado após o mnemônico da instrução. Caso tenha um asterisco, este deve ser removido para o passo de decodificação do mnemônico.

```
if (!Enum.TryParse(parts[0].ToUpper(), out Instructions mnemonic))
{
    throw new Exception($"Unknown instruction \"{parts[0]}\".");
}
memory[address] |= (uint)mnemonic << 25;
```

O mnemônico é então decifrado e escrito para os bits 25 a 31. Para definir quais são as instruções e qual é o número relacionado a cada uma, um enumerável foi criado(mostrado na seção 4.2), contendo todos mnemônicos e seus respectivos valores. Esse enumerável é convertido de texto para número através do `Enum.TryParse()`. Caso o mnemônico esteja errado, um erro de compilação é transmitido.

```
if (TryGetRegisterAddress(parts[1], out uint destination))
{
    memory[address] |= destination << 19;
}
else if (Enum.TryParse(parts[1].ToUpper(), out Condition condition))
{
    memory[address] |= (uint)condition << 19;
}
else
{
    throw new Exception($"Invalid Register or Unknown condition \"{parts[1]}\".");
}
```

O primeiro argumento da instrução pode ser tanto um registrador quanto uma condição. Para um registrador, o texto deve começar com "R"e ser seguido por um número, como por exemplo "R25". Um método simples que verifica esta formatação é usado na estrutura *if*. sJá para a condição, um enumerável é utilizado para realizar o deciframento, no mesmo modelo de deciframento do mnemônico. A descrição deste enumerável também se encontra na seção 4.2. Caso o código não detecte nem um registrador nem uma condição, um erro é transmitido.

```
switch (parts[2][0])
{
    case 'r':
    case 'R':
        if (TryGetRegisterAddress(parts[2], out uint source1))
        {
            memory[address] |= source1 << 14;
            break;
        }
        throw new Exception("Invalid Register: " + parts[2]);
    case '.':
        if (words.TryGetValue(parts[2][1..^0], out uint value))
        {
            memory[address] |= value;
            continue;
        }
        throw new Exception("Variable \"" + parts[2][1..^0] + "\" Not found");
    case ':':
```

```
        if (labels.TryGetValue(parts[2], out value))
        {
            memory[address] |= value;
            continue;
        }
        throw new Exception("Label \"" + parts[2][1..^0] + "\" Not found");
    case '#':
        uint immediate = uint.Parse(parts[2][1..^0], hexStyle);
        memory[address] |= immediate & 0x7FFFF;
        continue;
    default:
        immediate = uint.Parse(parts[2], intStyle);
        memory[address] |= immediate & 0x7FFFF;
        continue;
}
```

O segundo e o terceiro argumentos são bastante similares, podendo representar um registrador ou um valor imediato. No caso de valor imediato, estes podem ser definidos por endereços de rótulos, endereços de palavras, valores hexadecimais ou valores inteiros. Caso o segundo argumento seja um valor imediato, ele ocupará os primeiros 19 bits, seguindo para a próxima instrução em seguida. Caso seja um registrador, então o terceiro argumento é utilizado, podendo ser outro registrador ou um valor imediato de 13 bits.

4.1.6 Geração da Saída

Por fim, para que o código possa ser lido pelo simulador, ele precisa estar em um formato específico. Neste formato, cada linha representa os valores de 16 endereços, sendo iniciada pelo número de seu primeiro endereço e seguida por cada valor separado por espaço. Este assembler permite definir a largura de dados no arquivo de configuração, caso o modelo de simulação tenha essa largura modificada futuramente. A largura de dados determinará quantos bits cada endereço representa. Se a largura de dados for menor que os 32 bits necessários para cada instrução, esta deve ser dividida entre vários endereços.

Para realizar essa divisão, é utilizada uma máscara numérica, operações de *bitshift* e *bitwise*, para gravar apenas os bits corretos em cada endereço. Todos os dados são escritos em uma *string* que, ao final, é retornada e salva em disco pela classe principal.

```
var output = "v3.0 hex words addressed";
uint mask = uint.MaxValue >> (32 - bitWidth);
for (int i = 0, address = 0; i < memory.Length; i++, address += 32 / bitWidth) {
    if (i % (bitWidth / 2) == 0) output += $"\n{address:x8}:";
    for (int j = 0; j <= 32 - bitWidth; j += bitWidth) {
        var portion = (memory[i] >> j) & mask;
        output += $" {portion.ToString($"x{bitWidth / 4}")}";
    }
}
return output;
```

4.2 Sintaxe

A sintaxe do assembler é bem simples, possuindo 2 formatos de instrução principais, podendo ser definida pela seguinte notação BNF:

```
<instrução> ::= <formato1> | <formato2> | <rotulo> | <declaracaoPalavra>

<formato1> ::= <mnemônico> <registradorCondicao> <registrador> <registradorValor>

<formato2> ::= <mnemônico> <registradorCondicao> <valor>

<rotulo> ::= ":" <enderecoDeRotulo>

<declaracaoPalavra> ::= ".word" [a-zA-Z][a-zA-Z0-9]* <valor>
```

```
10
11 <mnemônico> ::= "CALLI" | "CALL" | "JMP" | "CALLR" | "JMPR" | "SLL" | "GETPSW" |
       "SRL" | "PUTPSW" | "SRA" | "LDBU" | "LDRBU" | "LDBS" | "LDRBS" | "LDW" | "LDRW"
       | "LDSU" | "LDRSU" | "LDSS" | "LDRSS" | "STS" | "STRS" | "STB" | "STRB" | "STW"
       | "STRW" | "AND" | "XOR" | "OR" | "SUB" | "SUBC" | "SUBR" | "SUBCR" | "ADD" |
       "ADDC" | "RET" | "RETI" | "GETLPC" | "LDHI"
12
13 <registradorCondicao> ::= <registrador> | <condição>
14
15 <registrador> ::= "R" <número> | "r" <número>
16
17 <registradorValor> ::= <registrador> | <valor>
18
19 <valor> ::= <número> | "#" <númeroHexadecimal> | ":" <enderecoDeRotulo> | "."
       <enderecoDePalavra>
20
21 <condição> ::= "NEVER" | "GREATER" | "LESS_EQUAL" | "GREATER_EQUAL" | "LESS" |
       "HIGHER" | "LOWER_SAME" | "CARRY_CLEAR" | "LOWER" | "CARRY_SET" |
       "HIGHER_OR_SAME" | "POSITIVE" | "NEGATIVE" | "NOT_EQUAL" | "EQUAL" |
       "OVERFLOW_CLEAR" | "OVERFLOW_SET" | "ALWAYS" | "NEV" | "GT" | "LE" | "GE" | "LT"
       | "HI" | "LOS" | "NC" | "LO" | "C" | "HIS" | "PL" | "MI" | "NE" | "EQ" | "NV" |
       "V" | "ALW"
22
23 <número> ::= [0-9]+
24
25 <númeroHexadecimal> ::= [0-9a-fA-F]+
26
27 <enderecoDeRotulo> ::=[a-zA-Z][a-zA-Z0-9]*
28
29 <enderecoDePalavra> ::= [a-zA-Z][a-zA-Z0-9]*
```

A definição valor dos mnemônicos de cada instrução fica em um enumerável para todas instruções. Considerando que uma mesma instrução pode ser representada por vários valores no RISC I, *assembler* sempre usa o menor valor hexadecimal que representa o mnemônico.

```
1  public enum Instructions
2  {
3      CALLI = 0x00,
4      CALL = 0x08,
5      JMP = 0x0A,
6      CALLR = 0x0C,
7      JMPR = 0x0E,
8      SLL = 0x10,
9      GETPSW = 0x12,
10     SRL = 0x18,
11     PUTPSW = 0x1A,
12     SRA = 0x1C,
13     LDBU = 0x24,
14     LDRBU = 0x25,
15     LDBS = 0x26,
16     LDRBS = 0x27,
17     LDW = 0x28,
18     LDRW = 0x29,
19     LDSU = 0x2C,
20     LDRSU = 0x2D,
21     LDSS = 0x2E,
22     LDRSS = 0x2F,
23     STS = 0x30,
24     STRS = 0x31,
25     STB = 0x34,
26     STRB = 0x35,
27     STW = 0x38,
28     STRW = 0x39,
29     AND = 0x40,
30     XOR = 0x44,
31     OR = 0x48,
```

```
    SUB = 0x50,
    SUBC = 0x52,
    SUBR = 0x54,
    SUBCR = 0x56,
    ADD = 0x58,
    ADDC = 0x5C,
    RET = 0x60,
    RETI = 0x64,
    GETLPC = 0x68,
    LDHI = 0x70,
}
```

Para as condições, foram definidas duas formas de escrever cada uma. A primeira, criada para este trabalho, utiliza o nome completo de cada condição para facilitar a legibilidade do *assembly*. A segunda forma foi definida por Katevenis, onde cada condição tem seu nome abreviado. Além disso, foi adicionada uma condição "nunca"para completar os 16 valores possíveis.

```
#pragma warning disable CA1069
public enum Condition
{
    NEVER = 0x0,
    GREATER = 0x1,
    LESS_EQUAL = 0x2,
    GREATER_EQUAL = 0x3,
    LESS = 0x4,
    HIGHER = 0x5,
    LOWER_SAME = 0x6,
    CARRY_CLEAR = 0x7,
    LOWER = 0x7,
    CARRY_SET = 0x8,
    HIGHER_OR_SAME = 0x8,
    POSITIVE = 0x9,
    NEGATIVE = 0xA,
    NOT_EQUAL = 0xB,
    EQUAL = 0xC,
    OVERFLOW_CLEAR = 0xD,
    OVERFLOW_SET = 0xE,
    ALWAYS = 0xF,
    //Condições encurtadas baseado no trabalho de katevenis
    NEV = 0x0,  //never // não existe na definição de katevenis
    GT = 0x4,   //Greater than
    LE = 0x7,   //less or equal
    GE = 0x5,   //Greater or equal
    LT = 0x6,   //less than
    HI = 0xA,   //higher than
    LOS = 0xB,  //lower or same
    NC = 0xE,   //no carry
    LO = 0xE,   //lower than
    C = 0xF,    //carry
    HIS = 0xF,   //higher or same
    PL = 0x8,   //plus
    MI = 0x9,   //minus
    NE = 0x2,   //not equal
    EQ = 0x3,   //equals
    NV = 0xC,   //no overflow
    V = 0xD,    //overflow
    ALW = 0x0,  //always
}
#pragma warning restore CA1069
```

4.3 Programa exemplo

O seguinte programa executa um algoritmo de *quicksort*, traduzido manualmente de um código C. O RISC I é uma arquitetura altamente dependente de um compilador para otimizações de código.

No entanto, devido a limitações de tempo, foi possível criar apenas o *assembler* para este trabalho. Por isso, este programa não possui nenhum tipo de otimização e foi escrito apenas como um teste geral para o funcionamento do *assembler* e do modelo de simulação.

Além disso, para funcionar corretamente na versão atual do modelo de simulação, a quantidade de variáveis a serem ordenadas deve ser pequena. Isso ocorre porque o algoritmo de *quicksort* funciona recursivamente e requer uma quantidade de janelas de registradores maior do que o modelo possui, resultando em um *overflow* do número de janelas. Para resolver isso, uma rotina de tratamento de interrupções de *overflow* e *underflow* de número de janelas deve estar disponível. No entanto, devido a restrições de tempo, não foi possível finalizar o código dessas rotinas. Ainda assim, este programa serve como um bom exemplo para testes.

```
ADD R0 R0 0
LDW R1 R0 .basePointer  //the array start and end will be global variables
LDW R2 R0 .arrayEnd

ADD R26 R1 0            //R16 is the low index passed to the quickSort function
ADD R27 R2 0            //R17 is the high index passed to the quickSort function
CALL R31 R0 :quickSort
ADD R0 R0 0

JMPR ALWAYS 0
ADD R0 R0 0

//*****************************************************************************//

:quickSort
SUB R0 R10 R11          //if low >= high, return
JMP GREATER_EQUAL R0 :endQuickSort
ADD R0 R0 0

ADD R26 R10 0           //passing low and high to the partition function
ADD R27 R11 0
CALL R31 R0 :partition  //the partition will return the pivot at register R28
ADD R0 R0 0
ADD R16 R28 0           //p = partition();

ADD R5 R0 3 //3 indicates its searching the left
//Quicksort on the left
ADD R26 R10 0           //low
SUB R27 R16 4           //pivot - 1
CALL R31 R0 :quickSort
ADD R0 R0 0

ADD R5 R0 5 //5 indicates its searching the right
//Quicksort on the right
ADD R26 R16 4           //pivot + 1
ADD R27 R11 0           //high
CALL R31 R0 :quickSort
ADD R0 R0 0

:endQuickSort
RET R0 R0 R31
ADD R0 R0 0

//*****************************************************************************//

:partition
LDW R18 R11 0           //Initialize pivot to be the highest element: pivot =
    array[high]

SUB R16 R10 4           // i = low - 1 //pointer for the greater element
SUB R17 R10 0           // j = low

JMP ALWAYS R0 :partitionForLoopComparison
```

```
ADD R0 R0 0

:partitionForLoopStart

LDW R19 R17 0           //array[j]
SUB R20 R19 R18         //array[j] <= pivot
JMP GREATER R0 :partitionIfEnd //if array[j] <= pivot swap with i
ADD R0 R0 0

ADD R16 R16 4           //i++

ADD R26 R16 0           //passing address of i to swap
ADD R27 R17 0           //passing address of j to swap
CALL R31 R0 :swap       //swap(array[i], array[j])
ADD R0 R0 0
:partitionIfEnd

ADD R17 R17 4           //j++
:partitionForLoopComparison
SUB R0 R17 R11          //while j < high
JMP LESS R0 :partitionForLoopStart
ADD R0 R0 0

ADD R26 R16 4           //passing address of i to swap
ADD R27 R11 0           //passing address of high(pivot) to swap
CALL R31 R0 :swap       //Swapping pivot with the greater element at i
ADD R0 R0 0

ADD R12 R16 4           //return i+1 as the position of the pivot
RET R0 R0 R31
ADD R0 R0 0
//*************************************************************************//

:swap
LDW R20 R10 0           //R20 = array[i]
LDW R21 R11 0           //R21 = array[j]
STW R20 R11 0           //array[i] = R21
STW R21 R10 0           //array[j] = R20
RET R0 R0 R31
ADD R0 R0 0

.word basePointer :arrayStart
.word arrayEnd :arrayEnd

.word padding0  #00000000
.word padding1  #00000000
.word padding2  #00000000

:arrayStart
.word element1  #6
.word element2  #4
.word element3  #5
.word element4  #2
.word element5  #3
.word element6  #1
.word element7  #9
.word element8  #F
.word element9  #A
.word element10 #3
.word element11 #C
.word element12 #6
.word element13 #3
.word element14 #0
.word element15 #8
:arrayEnd
.word element30 #2
```

Após a a montagem, todo o código pode ser carregado diretamente na memória RAM do simulador. O código a seguir mostra o resultado da montagem.

```
v3.0 hex words addressed
00000000: 00 20 00 b1 f0 20 08 51 f4 20 10 51 00 60 d0 b1
00000010: 00 a0 d8 b1 24 20 f8 11 00 20 00 b1 00 00 78 1d
00000020: 00 20 00 b1 0b 80 02 a1 6c 20 18 15 00 20 00 b1
00000030: 00 a0 d2 b1 00 e0 da b1 74 20 f8 11 00 20 00 b1
00000040: 00 20 87 b1 03 20 28 b1 00 a0 d2 b1 04 20 dc a1
00000050: 24 20 f8 11 00 20 00 b1 05 20 28 b1 04 20 d4 b1
00000060: 00 e0 da b1 24 20 f8 11 00 20 00 b1 1f 00 00 c1
00000070: 00 20 00 b1 00 e0 92 51 04 a0 82 a1 00 a0 8a a1
00000080: b0 20 78 15 00 20 00 b1 00 60 9c 51 12 c0 a4 a1
00000090: ac 20 08 15 00 20 00 b1 04 20 84 b1 00 20 d4 b1
000000a0: 00 60 dc b1 d8 20 f8 11 00 20 00 b1 04 60 8c b1
000000b0: 0b 40 04 a1 88 20 20 15 00 20 00 b1 04 20 d4 b1
000000c0: 00 e0 da b1 d8 20 f8 11 00 20 00 b1 04 20 64 b1
000000d0: 1f 00 00 c1 00 20 00 b1 00 a0 a2 51 00 e0 aa 51
000000e0: 00 e0 a2 71 00 a0 aa 71 1f 00 00 c1 00 20 00 b1
000000f0: 04 01 00 00 40 01 00 00 00 00 00 00 00 00 00 00
00000100: 00 00 00 00 06 00 00 00 04 00 00 00 05 00 00 00
00000110: 02 00 00 00 03 00 00 00 01 00 00 00 09 00 00 00
00000120: 0f 00 00 00 0a 00 00 00 03 00 00 00 0c 00 00 00
00000130: 06 00 00 00 03 00 00 00 00 00 00 00 08 00 00 00
00000140: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Considerações finais

Apesar do resultado final não seguir completamente a implementação original, pode-se afirmar que o trabalho realizado está bem próximo, sendo totalmente funcional e possuindo todas as instruções do RISC I.

Para trabalhos futuros, alguns aspectos podem ser melhorados. Caso a ideia seja manter a estrutura original do RISC I, o pacote de componentes, o posicionamento e o *layout* dos circuitos podem ser grandes fatores para facilitar o entendimento do funcionamento desta arquitetura.

No caso da monografia para a qual esse trabalho está sendo realizado, além dessas melhorias visuais, melhorias e simplificações na arquitetura podem ser executadas. A alteração de certos componentes pode ajudar tanto na eficiência da arquitetura quanto no seu aspecto didático. Inspirações de outras arquiteturas, incluindo o RISC II, vão ajudar a melhorar diversos aspectos da arquitetura, com a adição de componentes e recursos que vão simplificar a execução de operações específicas.

No *assembly*, a linguagem pode ser polida, oferecendo uma linguagem mais simples de ser entendida por humanos. Apesar do RISC I possuir um compilador que simplificava boa parte do processo, esta não é uma opção viável para este trabalho devido às limitações de tempo. Para trabalhos futuros, seria interessante a implementação de um compilador de alguma linguagem de alto nível para a linguagem *assembly* criada. Porém, por ora, o melhor que pode ser feito é a melhoria da sintaxe para algumas instruções, removendo a necessidade de inserção de parâmetros que a instrução usa ou criando formas mais fáceis de inserir dados.

Referências

Daniel T. Fitzpatrick et al. A RISCy approach to VLSI. *ACM Sigarch Computer Architecture News*, v. 10, n. 1, p. 28–32, jan. 1982. MAG ID: 2048226925. Citado na página 2.

KATEVENIS, M. Reduced instruction set computer architectures for VLSI. jan. 1985. MAG ID: 2171342458 S2ID: f2e99c1a499176ffe665c9b523080b1ac65665a0. Citado 3 vezes nas páginas 2, 11 e 13.

PEEK, J. *The VLSI Circuitry of RISC I*. Berkeley, 1983. 59 p. Disponível em: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/1983/6347.html>. Citado 12 vezes nas páginas 2, 3, 4, 6, 9, 11, 13, 14, 16, 18, 20 e 24.

STALLINGS, W. Reduced instruction set computer architecture. *Proceedings of the IEEE*, v. 76, n. 1, p. 38–55, jan. 1988. ISSN 1558-2256. Disponível em: <https://ieeexplore.ieee.org/document/3287>. Citado 3 vezes nas páginas 2, 3 e 7.

ANEXO A – Rotina de *Overflow e Underflow* de janela

```
//this code contains the routine for register window saving and restoring
//whenever there is an window number overflow or underflow.
//It starts at a certain address, right now it is set to #80000000.

//calli* R25 R0 R0 //saves the return address in R25 -This is a hardwired instruction, no need
getlpc* R24 R0 R0 //saves last instruction address in R24
getpsw R23 R0 R0 //saves the processor status word in R23

add R22 R22 1 //Every address starts at #80000000, so we create an offset
sll R22 R22 31 //removing the need to use relative instructions

ldhi R21 #60000     //sets the address to the lowest saved window address
ldw R20 R21 0       //loads lowest saved window address

//if the last instruction that was executed was an return execute the underflowRoutine
ldw R19 R25 0
ldhi R18 #40000
sub R19 R19 R18
jmp GREATER R22 :UnderflowRoutine
add R0 R0 R0

:OverflowRoutine
stw R26 R20 #04  //storing the input registers of the next window
stw R27 R20 #08
stw R28 R20 #0C
stw R30 R20 #10
stw R29 R20 #14
stw R31 R20 #18
add R27 R20 0 //passes the address offset as an argument
callr R10 R0 4 //goes to the window that is being saved
stw R16 R11 #1C
stw R17 R11 #20
stw R18 R11 #24
stw R19 R11 #28
stw R20 R11 #2C
stw R21 R11 #30
stw R22 R11 #34
stw R23 R11 #38
stw R24 R11 #3C
```

```
stw R25 R11 #40
add R10 R10 #34     //makes the return go to the next line instead of the callr instruction
ret R0 R0 R10 //returns to the routine window

add R20 R20 #40 //Incrementing the lowest saved window address
stw R20 R21 0

add R20 R23 #10     //Incrementing the Saved Window, but also incrementing the current window
and R20 R20 #70     //so that when we put the PSW, the window does not change
add R21 R23 #80
and R21 R21 #380
and R23 R23 #FFFFFC0F
or R23 R23 R20
or R23 R23 R21

putpsw R0 R23 R0 //restore the PSW
jmp* ALWAYS R25 R0
reti* ALWAYS R24 R0


:UnderflowRoutine

add R31 R15 0 //save the returning window output registers
add R30 R14 0
add R29 R13 0
add R28 R12 0
add R27 R11 0
add R26 R10 0

add R10 R20 0       //passes the lowestSavedWindowAddress to the lower window

getlpc R16 0        //calculates the address of the instruction after the next return
add R16 R16 #10
ret R0 R0 R16     //returns to the window with the return that triggered the underflow

add R31 R15 0 //save the parameters that were being passed to the lower window.
add R30 R14 0
add R29 R13 0
add R28 R12 0
add R27 R11 0
add R11 R26 0       //passes the lowestSavedWindowAddress to the lower window
add R26 R10 0

getlpc R15 0        //calculates the address of the instruction after the next return
add R15 R15 #10
ret R0 R0 R15     //returns to the window that needs to have its info restored

ldw R25 R27 #0 //now we can restore the saved window registers
ldw R24 R27 #1FFC
ldw R23 R27 #1FF8
ldw R22 R27 #1FF4
ldw R21 R27 #1FF0
ldw R20 R27 #1FEC
ldw R19 R27 #1FE8
```

```
ldw R18 R27 #1FE4
ldw R17 R27 #1FE0
ldw R16 R27 #1FDC
ldw R15 R27 #1FD8
ldw R14 R27 #1FD4
ldw R13 R27 #1FD0
ldw R12 R27 #1FCC
ldw R11 R27 #1FC8
ldw R10 R27 #1FC4

callr R0 R0 4 //calls the return underflow window

add R15 R31 0 //restore the arguments that were being passed to the lower window
add R14 R30 0
add R13 R29 0
add R12 R28 0
add R11 R27 0
add R10 R26 0

callr R0 R0 4 //calls the routine initial window

add R15 R31 0 //restore the arguments that were being passed to the lower window
add R14 R30 0
add R13 R29 0
add R12 R28 0
add R11 R27 0
add R10 R26 0

sub R20 R20 #40 //decrements the lowest saved window address
stw R20 R21 0

sub R20 R23 #10     //Decrementing the Saved Window, but also incrementing the current window
and R20 R20 #70     //so that when we put the PSW, the window does not change
add R21 R23 #80
and R21 R21 #380
and R23 R23 #FFFFFC0F
or R23 R23 R20
or R23 R23 R21

putpsw R0 R23 R0
jmp* ALWAYS R25 R0
reti* ALWAYS R24 R0


//.word lowestSavedWindowAddress #40000000
```