ENGENHARIA DE COMPUTAÇÃO

Gerson Miguel Beckenkamp

ARQUITETURA DE MÁQUINA DE PILHA, IMPLEMENTAÇÃO E AVALIAÇÃO DE PROCESSADOR EM UMA DESCRIÇÃO DE ALTO NÍVEL

Santa Cruz do Sul, dezembro de 2015

ENGENHARIA DE COMPUTAÇÃO

Gerson Miguel Beckenkamp

ARQUITETURA DE MÁQUINA DE PILHA, IMPLEMENTAÇÃO E AVALIAÇÃO DE PROCESSADOR EM UMA DESCRIÇÃO DE ALTO NÍVEL

Prof. Eduardo Weber Wächter
Orientador

Santa Cruz do Sul, dezembro de 2015

ENGENHARIA DE COMPUTAÇÃO

Gerson Miguel Beckenkamp

ARQUITETURA DE MÁQUINA DE PILHA, IMPLEMENTAÇÃO E AVALIAÇÃO DE PROCESSADOR EM UMA DESCRIÇÃO DE ALTO NÍVEL

Prof. Leandro Sehnem Heck
Avaliador

ENGENHARIA DE COMPUTAÇÃO

Gerson Miguel Beckenkamp

ARQUITETURA DE MÁQUINA DE PILHA, IMPLEMENTAÇÃO E AVALIAÇÃO DE PROCESSADOR EM UMA DESCRIÇÃO DE ALTO NÍVEL

Prof. Guilherme Castilhos
Avaliador

Gerson Miguel Beckenkamp

ARQUITETURA DE MÁQUINA DE PILHA, IMPLEMENTAÇÃO E AVALIAÇÃO DE PROCESSADOR EM UMA DESCRIÇÃO DE ALTO NÍVEL

Trabalho de Conclusão II apresentado ao Curso de Engenharia da Computação da Universidade de Santa Cruz do Sul, como requisito parcial para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Prof. Eduardo Weber Wächter

Santa Cruz do Sul, dezembro de 2015

RESUMO

Máquinas com arquitetura orientada a pilha são uma alternativa as arquiteturas de registradores encontradas nos computadores atuais. Em uma máquina de pilha, as instruções não precisam endereçar operandos pois o comportamento da pilha limita o acesso a uma posição referente ao topo da mesma, embora algumas abordagens permitam o endereçamento de até dois operandos. Esta abordagem pode apresentar algumas vantagens em relação a máquinas de registradores, visto que esta máquina trabalha naturalmente com pilhas para qualquer operação aritmética, lógica ou de chamada de função. Enquanto que em máquinas de registradores, estruturas de pilha precisam ser montadas e acessadas em memória, o que normalmente tem menor desempenho em relação ao uso de registradores. Por outro lado, a máquina de registradores apresenta maior flexibilidade em relação a máquinas de pilha sem operandos, visto que o endereçamento explícito de operandos não requer nenhum tipo de movimentação na estrutura de memória para que os operandos sejam alocados no topo de uma pilha. Entretanto, a programação de máquinas de pilha se diferencia muito da programação normalmente utilizada em máquinas de registradores, tornando-as incompatíveis entre si. Com isto, propõem-se um estudo sobre as máquinas de pilha de forma a analisar quais mudanças arquiteturais podem vir a apresentar maior impacto no desempenho do sistema. Ainda, espera-se obter uma comparação de desempenho com uma arquitetura baseada em registradores.

Palavras-chave: Máquina de pilha, máquina de registradores, arquitetura de computadores.

ABSTRACT

Machines with stack oriented architecture are an alternative to register architectures found in today's computers. In a stack machine, the instructions do not need address operands because the stack behavior limits the access to a position on the top of it, although some approaches allow addressing of up to two operands. This approach may have some advantages over registers machines, since this machine works naturally with stack for any arithmetic, logic or function call. While in registers machines, stack structures need to be mounted and accessed in memory, which typically has lower performance compared to using registers. On the other hand, the register machine has greater flexibility compared to no operand stack machines, as the explicit addressing operands does not require any drive in the memory structure so that the operands are allocated on top of a stack. However, the stack machine programming is substantially different from the typically programming used in registers machine, making them incompatible. With this, we propose a study of the stack machines in order to analyze architectural changes which are likely to have greater impact on system performance. Still, it is expected to obtain a performance comparison with a register-based architecture.

Keywords: Stack machine, register machine, architecture.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ASIC	Application Specific Integrated Circuit
CPU	Central Process Unit
DS	Data stack
DSP	Digital Signal Processor
LIFO	Last In First Out
MIPS	Millions of Instructions Per Second
NTos	Next to the top of the stack
PC	Program Counter
RAM	Random Acess Memory
RISC	Reduced Instruction Set Computers
RS	Return stack
TLM	Transaction-Level Modeling
Tors	Top of the Return Stack
Tos	Top of the stack
ULA	Unidade Lógica e Aritmética
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VM	Virtual Machine

SUMÁRIO

1 INTRODUÇÃO

Arquiteturas de máquinas utilizando pilhas são uma alternativa as arquiteturas de computadores utilizadas atualmente, que utilizam registradores como forma de trabalhar com os operandos. Apesar de algumas implementações em ASIC terem sido fabricadas e comercializadas no passado, o presente desta tecnologia encontra-se esquecido tanto na indústria quanto na academia.

Máquinas de pilha apresentam características que podem vir a trazer maior desempenho em certos cenários, quando comparadas as máquinas de registradores, pois estas fazem uso de estruturas de pilhas em um nível de memória mais alto do que a usada em máquinas de pilha. Contudo, a programação orientada a pilha sendo mais complexa do que a programação orientada a registrador, pois boa parte das máquinas de pilha não permitem o endereçamento explicito de qualquer posição da pilha, mas somente o topo da pilha.

O desenvolvimento deste trabalho utiliza a linguagem de programação e descrição de *hardware SystemC*. Esta permite a descrição de *hardware* em um maior nível de abstração se comparado com VHDL e *Verilog*, por exemplo, o que possibilita um tempo de desenvolvimento reduzido. Para aumentar o grau de abstração, será utilizado Transaction Level Modeling(TLM), que lida com barramentos entre módulos de forma mais genérica e simples, atingindo assim um melhor aproveitamento do espaço de projeto.

Este volume é dividido da seguinte forma. O Capítulo 2 apresenta a teoria básica sobre os temas apresentados neste trabalho, como máquinas de pilha e registradores. O Capítulo 3 apresenta uma análise de algumas implementações já existentes de máquinas de pilha, bem como as otimizações encontradas nas máquinas analisadas. O Capítulo 4 apresenta uma analise detalhada

sobre as principais características de uma arquitetura de máquina de pilha, propondo otimizações e as vantagens e desvantagens destas.

O Capítulo 5 apresenta como as arquiteturas de máquina de pilha implementadas para este projeto foram desenvolvidas e como estas funcionam. No Capítulo 6 é explicado como foram implementados os algoritmos para as máquinas propostas, bem como as ferramentas desenvolvidas. O Capítulo 7 apresenta os resultados obtidos com as máquinas desenvolvidas, comparando-as entre si e com uma máquina de registradores. Por fim, o Capítulo 8 apresenta a conclusão sobre as máquinas implementadas e os resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados tópicos relacionados aos presente trabalho que se fazem necessários para o entendimento deste Trabalho de Conclusão.

2.1 Máquina de registradores

Uma máquina de registradores consiste em uma arquitetura onde um conjunto de registradores formam a memória local da CPU, que é acessada de forma explicita pelos operandos na instrução. Com isto, a instrução deve especificar ambos os endereços de origem dos operandos, quanto o endereço de destino em relação ao banco de registradores. Este comportamento faz com que o processador tenha de buscar todos os operandos no banco de registradores a cada início de instrução e, ao término da execução, necessita que a posição de destino seja procurada novamente para a alocação do resultado.

2.1.1 MIPS

MIPS é uma máquina de 32 *bits* desenvolvida na universidade de Stanford em 1981 (DAVID A. PATTERSON, 2005). Tendo como abordagem manter o conjunto de instruções reduzido (RISC), contendo apenas seis tipos de instrução possíveis:

- Aritmética: responsável pelas operações matemáticas básicas;
- Transferência de dados: responsável por leitura e escrita na memória principal;
- Lógico: responsável por operações lógicas básicas;

- Deslocamento: responsável por deslocamentos dos *bits* de um operando;
- Desvio condicional: responsável por saltos a partir de igualdade ou não de dois operandos;
- Salto incondicional: responsável por saltos simples e saltos com retorno;

A maior parte do banco de registradores MIPS é utilizado para operações lógicas e aritméticas. Entretanto, alguns registradores são reservados para darem suporte a controle de estruturas de dados em memória principal e à passagem de parâmetros entre chamadas.

Em casos de chamadas de função, os registradores auxiliares entre *$a0* e *$a3* são utilizados para a passagem de parâmetros, os registradores *$v0* e *$v1* recebem o valor de retorno da função, e o registrador *$ra* guarda o endereço de retorno do contador de programa (PC) de retorno (DAVID A. PATTERSON, 2005). Assim, estes registradores são utilizados exclusivamente para as tarefas citadas anteriormente, visto que caso sejam modificados de forma incorreta a execução da função não será correta e o resultado do processamento estará errado.

Entretanto, quando existe uma chamada de função durante a execução de uma função, é necessário o fazer uma cópia do valor de alguns registradores para a troca de contexto. Na arquitetura MIPS, por convenção, antes de uma chamada de função, o contexto é copiado para os registradores *s0* a *s7* (DAVID A. PATTERSON, 2005).

Para isto, os registradores de *frame pointer* (*$fp*) e *stack pointer* (*$sp*) são utilizados para gerenciar uma pilha na memória principal. A cada chamada de função, são passados à esta pilha os argumentos e retornos da chamada anterior que estavam alocados no banco de registradores. Desta forma, a nova chamada pode alocar seus argumentos, retorno e endereço de retorno diretamente no banco de registradores sem afetar o comportamento da chamada anterior. Quando a chamada corrente na CPU for terminada, a chamada anterior pode ser carregada de volta da memória principal.

Outro requisito necessário ao MIPS é dar suporte a variáveis estáticas, que são mantidas independente do início e término de funções e chamadas. Para isto o MIPS designa um registrador chamado *global poiter $gp*, que é responsável por indexar a posição de memória em relação ao frame pointer onde as variáveis estáticas estão guardadas.

2.2 Máquina de pilha

Uma pilha é uma estrutura de dados do tipo *Last In First Out* (LIFO) cuja a única interface de acesso é sua posição mais ao topo. Sua implementação é feita de forma que o menor endereço de memória fique sempre como o último valor alocado, fazendo com que a pilha cresça na direção do maior endereço de memória. Um ponteiro chamado *Top of stack* (Tos) é usado para guardar a posição de memória mais ao topo da pilha, permitindo assim que a posição de memória do topo da pilha não precise ser fisicamente apagada em caso de retirada de valor da pilha, visto que logicamente este valor não faz mais parte da mesma (PHILIP J. KOOPMAN, 1989).

Os comandos mais básicos de uma pilha são os de empilhamento (*push*) e desempilhamento (*pop*). Ao executar um comando de *push*, o valor a ser adicionado a pilha é carregado no endereço de memória acima de Tos e após a escrita do valor, Tos é atualizado com a nova posição de topo. Ao chamar um *pop*, o valor contido no endereço de Tos será devolvido a quem fez a chamada e Tos será atualizado com o próximo endereço abaixo do topo da pilha anterior.

Desta forma, toda e qualquer operação lógica ou aritmética que envolva mais de um valor contido na pilha implica no consumo destes valores. Isto acontece pois a pilha pode apenas inserir ou expelir valores de seu topo, o que leva à retirada de um valor da estrutura de memória para que este possa ser computado.

2.2.1 Taxonomia de *hardware* em uma máquina de pilha

É descrito por Philip Koopman como: "Uma taxonomia de *hardware* separa em grupos diferentes ramificações de um determinado tipo de máquina baseando-se em aspectos importantes na implementação da mesma. Uma boa taxonomia permite a observação de questões de design sem aprofundar-se nos detalhes de implementação de uma determinada máquina".

O espaço de planejamento de uma arquitetura em máquina de pilha é moldado em três aspectos principais, sendo elas o tamanho da pilha, quantidade de pilhas e o número de operandos na instrução. Cada um destes aspectos tem importância vital na forma como a máquina se comporta e em seu resultado final, permitindo-se assim implementar uma máquina mais adequada à uma determinada aplicação.

2.2.1.1 Tamanho da pilha

O tamanho da pilha tem grande importância sobre o desempenho de uma máquina de pilha pois afeta diretamente a quantidade de iterações necessárias com a memória RAM. Segundo *Koopman*, uma pilha com mais de 16 posições de memória é considerada grande, embora esta noção de tamanho esteja muito mais ligada ao escopo do projeto do que a um tamanho pré-definido.

Por um lado, pilhas maiores podem comportar um maior número de valores em sua estrutura, o que acaba dando maior autonomia em relação a acessos a memória RAM para carregamento de mais elementos. Contudo, esta abordagem pode diminuir o desempenho em casos de execução multi tarefa, já que a cada troca de contexto toda a pilha tem que ser guardada em memória.

Em pilhas de tamanho pequeno, as principais vantagens são a simplicidade de implementação em *hardware* e a manipulação mais rápida de elementos perto da base da pilha. Entretanto o acesso à memória principal é mais frequente, visto que o tamanho reduzido implica em um maior número de leituras e escritas.

2.2.1.2 Quantidade de pilhas

Normalmente, máquinas que contém apenas uma pilha tem sua implementação em *hardware* mais simples comparado com máquinas com múltiplas pilhas, pois a CPU só precisa controlar uma unidade de memória interna. Porém, todo dado tratado pela máquina, incluindo passagem de parâmetros e endereços de sub-rotina, é guardado em uma única pilha. Como diferentes tipos de dados ficam guardados na mesma estrutura de memória, a programação da máquina se torna mais complexa e um maior número de operações de ajuste para trazer os valores desejados ao topo da pilha é requerido em chamadas de sub-rotinas.

Já em máquinas com múltiplas pilhas, o fluxo de controle para dados pode ser dividido em várias pilhas separadas, tornando o sistema mais organizado. Dentre as implementações de máquinas com múltiplas pilhas, a abordagem mais adotada é uma pilha de dados (DS) em conjunto com uma pilha de retorno (RS). Entretanto outras pilhas podem ser adicionadas, como

por exemplo, pilha para variáveis locais e pilha de parâmetros.

Uma importante vantagem de se implementar uma máquina com múltiplas pilhas é a de poder acessar múltiplos valores em um único ciclo de relógio. Possibilitando assim, que mais de uma operação seja feita simultaneamente.

2.2.1.3 Número de operandos na instrução

Em relação aos operandos, uma arquitetura de máquina de pilha pode apresentar de 2 a nenhum operando. A mudança do número de operandos na instrução de uma máquina de pilha dita como será feito o acesso à pilha de dados, variando o comportamento da CPU como consequência. Com isto, o sistema tende a aumentar ou diminuir a complexidade de *hardware* conforme determinada abordagem é utilizada.

Uma arquitetura com nenhum operando é considerada uma máquina pura. Sua instrução suporta comandos ou apenas um carregamento de valor literal por vez, tendendo a ter menor tamanho de instrução e a mais simples das abordagens. As operações aritméticas e lógicas normalmente são feitas através do acesso indireto ao topo da pilha de dados, acesso este concedido por um ou dois *buffers* que estão sempre sincronizados com o topo da pilha. Desta forma, os operandos estão sempre prontos para serem usados, permitindo assim que o processo de busca de instrução (*fetch*) seja executado em paralelo com a operação da ULA, o que acaba por eliminar a necessidade do uso de *pipeline* nos estagios de *fetch* e armazenamento de operandos (*store*) (PHILIP J. KOOPMAN, 1989). Outra vantagem desta abordagem é a de que como não há operandos na instrução, as instruções podem ter até 8 *bits* contendo apenas o comando da computação a ser realizada (BAILEY, 1996).

Em máquinas que consideram um operando na instrução, permite-se que qualquer posição da pilha seja acessada para efetuar uma computação, enquanto o outro operando a ser carregado continua sendo o do topo da pilha. Esta abordagem garante uma maior flexibilidade em relação à máquina pura, pois permite o acesso de uma posição da pilha de forma explícita. Além disto, as máquinas de um operando podem suportar instruções da máquina de zero operandos. Por exemplo, se a máquina tiver uma palavra de 16 *bits* para instruções com um operando e 8 *bits* para zero operandos, pode-se atribuir uma *flag* na instrução indicando o número de operandos e

por consequência o tamanho da mesma. Desta forma, a máquina de um operando pode emular operações de zero operandos.

Por fim, a máquina com instruções de dois operandos é a abordagem mais explícita de todas, pois permite que duas posições da pilha sejam acessadas pela instrução. É a abordagem mais flexível pois não requer um grande controle sobre o fluxo de dados da pilha, já que qualquer posição pode ser acessada. Entretanto a abordagem é mais custosa em relação a *hardware*, visto que ambos os operandos devem ser buscados a pilha.

2.2.2 A pilha de dados

A pilha de dados é utilizada principalmente para armazenar valores lógicos e aritméticos, servindo tanto de fonte como de destino de valores para a ULA. Outra finalidade atribuída a ela é a passagem de parâmetros em chamadas de sub-rotina. Sua alocação pode ser feita tanto em um banco de registradores da CPU quanto em memória RAM, dependendo dos requisitos do projeto. Apesar de ter a mesma forma de um banco de registradores comum, sua utilização como pilha muda o comportamento do resto da CPU devido ao seu modo de acesso.

Sua implementação básica é feita com um *buffer* apontando para o topo da pilha, cuja a funcionalidade principal é facilitar o acesso da pilha, mantendo o valor do topo da pilha carregado para o uso da CPU, em especial para uma das entradas da ULA. Em máquinas que implementam apenas Tos, operação que requerem mais de um operando realizam um *pop*, onde o valor retirado da pilha é guardado em um barramento de dados. Como Tos está agora apontando para o segundo operador, a ULA carrega os valores contidos em Tos e no barramento de dados, atualizando o topo da pilha com o valor resultante (PHILIP J. KOOPMAN, 1989).

Algumas implementações podem apresentar o *buffer* próximo ao topo da pilha (Ntos). Nestes casos, NTos é conectado a segunda entrada da ULA e ao barramento de dados, facilitando o acesso do mesmo. Em máquinas com ambos Tos e Ntos, operações com dois operandos permitem que a ULA acesse simultaneamente os operandos. Após a leitura dos valores da pilha, esta recebe um *pop*, restando apenas à ULA sobrescrever Tos com o valor calculado.

2.2.3 A pilha de retorno

A pilha de retorno é um conjunto de memória complementar à pilha de dados que como ela, pode ser implementada tanto em um banco de registradores da CPU quanto em memória RAM. Sua finalidade principal na máquina é guardar endereços de retorno para as chamadas de função, dando assim suporte de *hardware* a sub-rotinas. Sendo que sua implementação é idêntica à de uma pilha de dados que contém apenas um *buffer* apontando para seu topo, que neste casso recebe o nome de Tors (*Top of return stack*).

2.2.4 Chamadas de função

Antes de uma chamada de função, a função que invoca a chamada (*caller*) se responsabiliza por colocar os parâmetros mais ao topo da pilha de dados como visto na transição de *a* para *b* na Figura 1. Por convenção, a função invocada (*callee*) se responsabiliza por manipular a pilha somente a partir dos parâmetros, mantendo os dados do *caller* intactos como visto em *c*. Assim ao terminar sua execução, o *callee* deixa na pilha somente os valores de retorno e elimina da pilha todo o resto que tenha sobrado da computação *d*. Por fim, a função *caller* recebe a pilha de dados com sua partição de dados intacta e com os valores retornados da chamada mais ao topo.

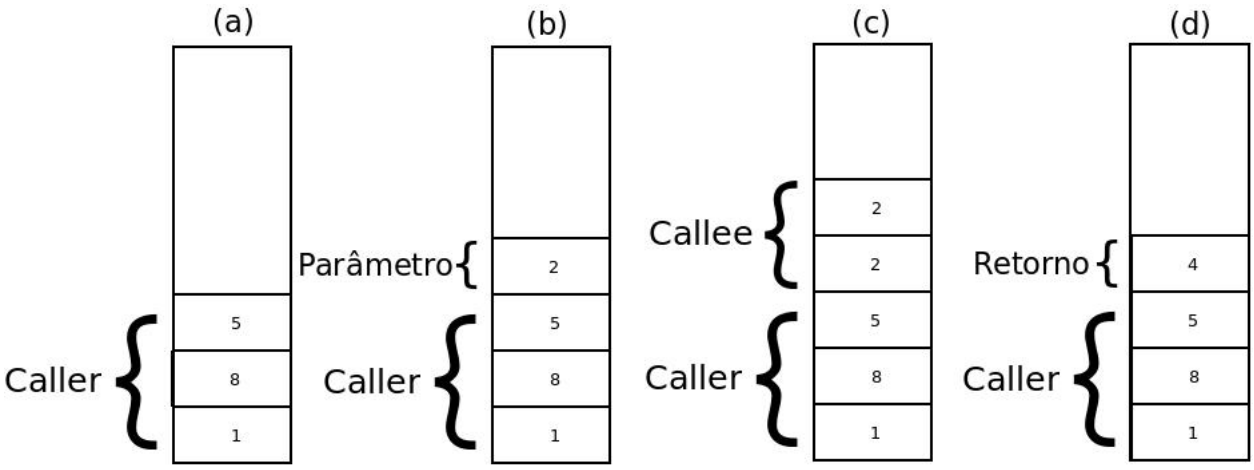

Figura 1: Exemplo do uso da pilha de dados em uma chamada de função.

Como visto na Figura 2, ao executar uma chamada de função o PC é carregado na pilha de retorno *a*. Na sequência, o endereço onde o início do código da função está localizado é carregado no registrador de PC *b*. Após a execução da função, há uma chamada de retorno onde o último valor carregado na pilha re retorno é passado ao registrador de PC *c*, que na sequência

sofre um acréscimo *d* para que a execução continue.

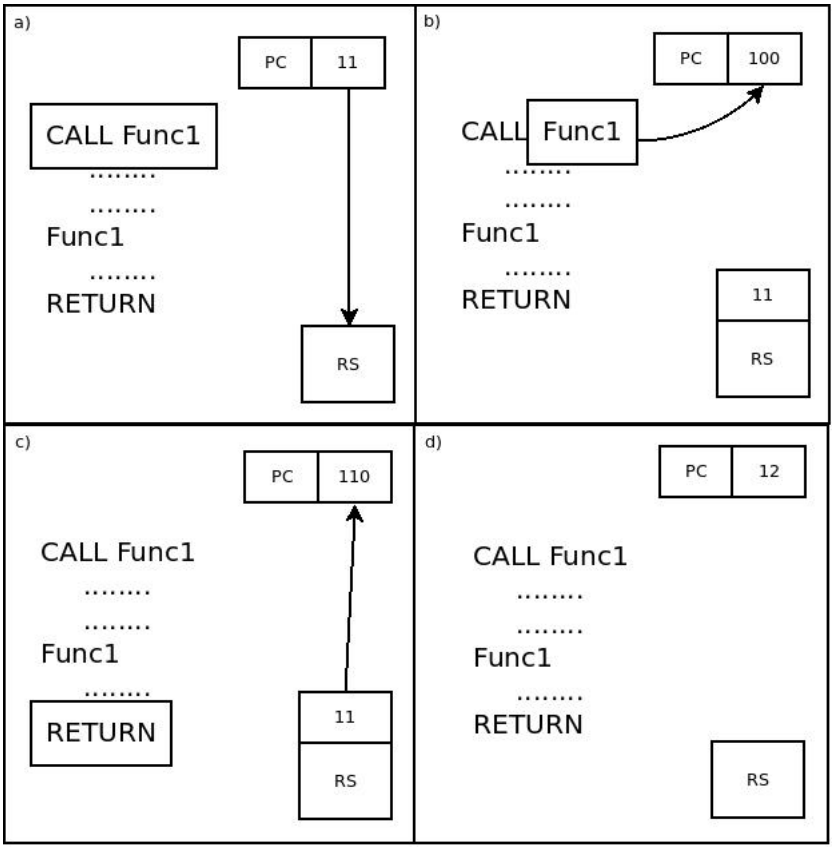

Figura 2: Exemplo do uso da pilha de retorno em uma chamada de função.

2.3 A linguagem *Forth*

Forth é uma linguagem de programação desenvolvida por Chuck Moore em meados dos anos 60, cuja sua primeira versão foi desenvolvida em *ALGOL*. De início, *Forth* era apenas um interpretador de textos que lê os comandos oriundos do teclado e executa funções baseadas neles (PELC, 2005). Posteriormente, a orientação à pilha foi adicionada na linguagem, devido a projetos de Moore em *hardware* orientado a pilha.

A linguagem *Forth* pode ser descrita como iterativa e extensiva. Iterativa porque pode funcionar em tempo real com entradas do teclado. Extensiva pois para *Forth* não há diferenciação entre funções nativas do sistema em relação as escritas pelo usuário. Aquilo que para outras linguagens de programação são chamadas funções, sub-rotinas e procedimentos, em *Forth* são chamadas de palavras. Todo e qualquer tipo de função, sub-rotina procedimento inserido pelo usuário é considerado da mesma forma pelo compilador, que após compilar, considera a palavra como nativa do sistema.

Para isto, *Forth* implementa um dicionário de palavras conhecidas pelo sistema. O dicionário vem carregado por padrão com todas as funções básicas da linguagem, como controle de laços, operações lógico e aritméticas, controle sobre memória, operações sobre pilha, etc. Conforme o usuário necessitar, mais palavras podem ser adicionadas a partir das primitivas básicas. Assim, o programador pode tanto programar chamadas em baixo nível diretamente com primitivas, quanto programar funções de alto nível com palavras preexistentes.

O pacote de execução em tempo real do *Forth* traz consigo um conjunto compacto de compilador, interpretador e ferramentas. Assim, o código *Forth* pode ser escrito tanto em terminal a partir do teclado quento carregado do disco, sendo que o código escrito em terminal pode ser salvo em disco. Isto permite que programas salvos anteriormente possam ser carregados por terminal ou acessados por outros programas.

Outra de suas características é a versatilidade de código entre alto e baixo nível de abstração. Uma aplicação pode ser escrita totalmente em alto nível de abstração sem precisar se preocupar com o código de máquina gerado. Após a validação da aplicação, palavras cuja a forma de funcionamento tenham que ser alteradas podem ser modificadas de forma individual em baixo nível de abstração (BRODIE, 2004).

Existem duas formas básicas de executar um programa em *Forth*, que são a execução em uma máquina de pilha física e o uso de máquinas virtuais. Considerando máquinas físicas, normalmente microcontroladores, o código é compilado e carregado diretamente na memória utilizada pela CPU. Com isto, podem ser tanto carregados códigos para alguma função específica quanto o compilador, o interpretador e as ferramentas *Forth*, possibilitando assim a utilização de um pequeno sistema operacional no microcontrolador.

Considerando a execução em máquina virtual(VM), o sistema executa sobre o sistema operacional corrente. Nesta VM o sistema operacional *Forth* sempre é carregado com a adição de funções especiais para possibilitar a comunicação de recursos com o sistema operacional hospedeiro (*host*), especialmente o acesso a arquivos.

A VM *Forth* contém basicamente a CPU, a DS, a RS e a memória principal como visto na Figura 3. A CPU se conecta em paralelo com cada uma das memórias citadas, sendo as memórias para pilha de dados e retorno não endereçáveis e separadas da memória principal

(BRODIE, 2004).

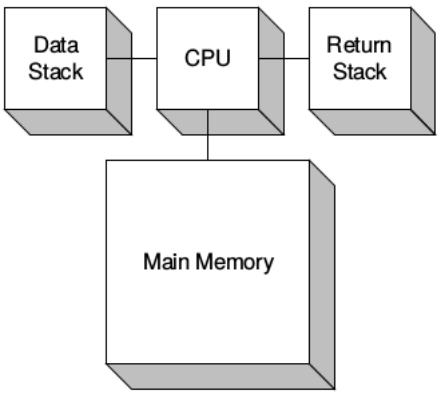

Figura 3: Esquema básico da VM *Forth*.
Fonte: (PELC, 2005).

Uma das peculiaridades da linguagem *Forth* é o uso de notação polonesa reversa (RPN). Sua adoção é dada pois esta notação tem como seu comportamento natural as operações em relação a pilha. Nesta notação, os valores a serem usados como operandos são carregados antes da chamada do operador, o que modulariza naturalmente o formato da expressão e elimina a necessidade de parenteses.

O Código 1 mostra um exemplo simples de definição de palavra em *Forth*. O compilador ira detectar uma palavra a partir do caractere ':' que define o início da palavra, e o caractere ';' que indica o fim da palavra. Após o início da palavra, o primeiro conjunto de caracteres nomeara a palavra, sendo este o nome a ser passado em código toda vez que a função for chamada. Na sequência, o '*' indica a multiplicação dos dois componentes mais ao topo da pilha de dados, seguidos de um '.' que realiza desempilhamento do topo da pilha de dados para imprimir este valor no terminal.

Código 1: Trecho de código usado para definir uma palavra em *Forth*.

```
1 : TIMES * . ;
2 2 4 TIMES
```

A segunda linha do Código 1 utiliza a nova palavra. Devido a noção RPN, os dois operandos são passados por primeiro ao compilador, que os interpreta como um empilhamento de valores na pilha de dados. Na sequência, o compilador chama a função referente à palavra especificada em código, que efetuara uma multiplicação entre os dois operandos ao topo da pilha

de dados. Com isto, o valor oito resultante da operação de multiplicação da linha anterior estará presente na pilha, que por fim é desempilhado para ser impresso no terminal.

2.4 Formas de codificação

A maior parte dos processadores requer uma unidade de controle. Esta unidade de controle é responsável por decodificar as instruções e gerar os sinais de controle para um ou mais módulos da CPU. Na instrução, um ou mais grupos de *bits* são utilizados exclusivamente para informar à unidade de controle qual computação deve ser realizada. Com isto, esta instrução pode ser codificada basicamente de maneira *hardwired* e *microcoded*.

2.4.1 *Microcoded*

É a maneira de codificação mais adotada nos processadores atuais. Consiste em uma memória rápida que se encontra na unidade de controle, e que pode ser programada com uma ou mais microinstruções de máquina. Com isto, o programador tem a liberdade de adicionar novas instruções informando ao processador o conjunto de microinstruções necessárias para a execução daquela função.

Este método garante maior flexibilidade na implementação da máquina pois as instruções aceitas pela máquina podem ser modificadas, permitindo assim que o sistema possa aceitar mais instruções ou que instruções antigas sejam alteradas. O método requer menor esforço do projetista visto que a instrução opera apenas na unidade de controle, sem ter efeito direto em nenhuma outra parte da CPU.

2.4.2 *Hardwired*

Esta técnica de codificação foi a primeira a ser empregada em circuitos de controle. Ela permite que a instrução tenha acesso direto a qualquer módulo da CPU. Com isto, a unidade de controle necessita apenas selecionar, a partir da instrução, qual o sinal é endereçado a qual módulo.

Uma das principais vantagens desta abordagem é que a unidade de controle é bastante simples, necessitando de poucas portas lógicas para atribuir as funções de cada módulo da CPU, desta forma, acaba sendo mais rápido em relação a abordagem anterior (KOOPMAN, 1987). Por outro lado, esta abordagem requer que todas unidades funcionais da CPU já tenham seu comportamento definido para que a unidade de controle seja desenvolvida. Com isto, qualquer modificação na arquitetura requer que todo o controle da CPU seja redesenhado e alterado.

2.5 *SystemC*

A linguagem *SystemC* tem como objetivo modelar circuitos de pequena a grande complexidade em nível de sistema. Garantindo maior flexibilidade no desenvolvimento de *hardware* e *software* em múltiplos níveis de abstração (SYSTEMC® REFERENCE MANUAL, 2012).

Trata-se de uma biblioteca para *C++* com uma série de recursos para abstração de *hardware*, tipos de dados orientados a hardware, sincronização por meio de eventos e sensitividade, etc. Tornando assim possível que se descreva o comportamento de um circuito a partir de uma linguagem de programação de alto nível, se comparado a outras linguagens de descrição de *hardware*(HDL).

Recursos do *C++* podem ser usadas no desenvolvimento da aplicação em conjunto com os recursos do *SystemC*. Porém, todos os recursos usados devem seguir as regras e restrições contidas no padrão *SystemC* (SYSTEMC® REFERENCE MANUAL, 2012).

Por permitir uma modelagem mais abstrata comparada a outras linguagens de descrição de *hardware*, o *SystemC* permite que sistemas de *hardware* de grande complexidade sejam avaliados com rapidez. Sendo assim, a validação de um sistema, bem como suas modificações, se tornam mais rápidas e fáceis, reduzindo o tempo de projeto.

2.5.1 Modelagem em nível de transação (TLM)

Transaction-level modeling (TLM) consiste no uso de pacotes para abstrair a comunicação de barramentos entre módulos. Com isto, são utilizadas interfaces de transporte bloqueantes e não bloqueantes para transmitir pacotes de dados genéricos entre o módulo de origem e o módulo

de destino (SYSTEMC® REFERENCE MANUAL, 2012).

Para maximizar a interoperabilidade entre os *sockets*, o protocolo utilizado em cada comunicação pode ser personalizado. Isto junto da escolha de primitivas bloqueantes, conferem ao programador maior liberdade na elaboração da comunição entre módulos.

A utilização de TLM 2.0 em sistemas desenvolvidos em *SystemC* permite um grau mais elevado de abstração. Isto porque a criação de um barramento, bem como a implementação de seu protocolo, não é necessário. Ao invés disto, o TLM garante um canal de comunicação abstrato, configurável e com protocolo de recebimento de dados pré-estabelecido, facilitando a implementação e diminuindo o tempo de projeto.

3 REFERENCIAL BIBLIOGRÁFICO

Neste capítulo são analisadas as máquinas utilizadas como referência no desenvolvimento do trabalho. Somente aspectos que cabem ao Trabalho de Conclusão serão detalhados, como formatos de instrução e tratamento de pilhas. Os demais tópicos como tratamento de interrupção e uso de *pipeline* não terão grande atenção porque este Trabalho de Conclusão não visa o uso destes recursos. Contudo, em uma futura continuidade do trabalho, estes tópicos podem se tornar interessantes.

Outro ponto a ser abordado em relação a análise dos conceitos referentes a este trabalho é a definição de uma máquina de pilha, uma máquina de registrador e uma máquina híbrida. Será considerado **máquina de pilha**, qualquer máquina cujo o banco de registradores seja tratado como pilha pelo sistema. Caso hajam *buffers* internos apontando à memória RAM, é considerado uma máquina de pilha somente se os blocos de memória apontados sejam tratados como pilha pelo processador, especialmente nos blocos utilizados para operações lógicas e aritméticas.

Uma **máquina de registradores** é aquela onde a memória local da máquina é acessada de forma totalmente explícita. Vale ressaltar que muitas arquiteturas deste tipo se utilizam de artifícios para garantir suporte em *hardware* a determinados recursos da RAM, especialmente para chamadas de função. Nestes casos é comum o uso de *buffers* em máquina para apontar estruturas de pilha em RAM. Mesmo assim, esta máquina continua a ser de registradores pois sua instruções de operação lógico e aritmética continuam sendo completamente explícita.

Por fim, uma **arquitetura híbrida** é aquela que fica entre os casos citados anteriormente. Tanto máquinas com uma pilha de dados e banco de registradores endereçáveis em CPU, quanto máquinas com pilhas auxiliares em *hardware* juntamente de um banco de registradores para

operações lógico aritméticas entram nesta categoria.

3.1 O microcontrolador *J1*

A máquina *J1* foi criada com o intuito de ser um microcontrolador extremamente simples e eficiente para a transmissão de vídeo não compactado via *ethernet*. Para isso, a máquina implementa a maioria das microinstruções *Forth*, deixando de lado desvios relativos, interrupções, exceções, multiplicação e divisão.

A *J1* se apresenta como uma CPU de 16 *bits*, contendo uma pilha de dados de 33 posições (32 mais Tos), uma pilha de retorno de 32 posições e um contador de programa de 13 *bits*, sem nenhum outro tipo de memória interna para a alocação de estados. Além disto, *buffers* para as posições do topo da pilha (*T*) e próximo ao topo (*N*) são usados para acessar a pilha de dados, e outro *buffer* (*R*) é utilizado para acessar o topo da pilha de retorno. Como toda estrutura de memória manipulada pela máquina é somente de 16 *bits*, o uso de estruturas de dados que necessitam mais de 16 *bits* são implementadas em *software*.

Uma das otimizações encontradas na *J1* é o uso de instruções *hardwired*, isto é, a instrução ao invés de ser codificada é organizada em séries de *bits* que são atribuídos a determinada unidade funcional da CPU, que no caso da *J1* são *fetch* e ULA. Com isso, a etapa de decodificação não é necessária já que todas as unidades funcionais já recebem suas tarefas diretamente da instrução. Como visto na Figura 4 estas instruções divididas entre literal, salto, salto condicional, chamada e ULA, como na maioria das demais máquinas de pilha.

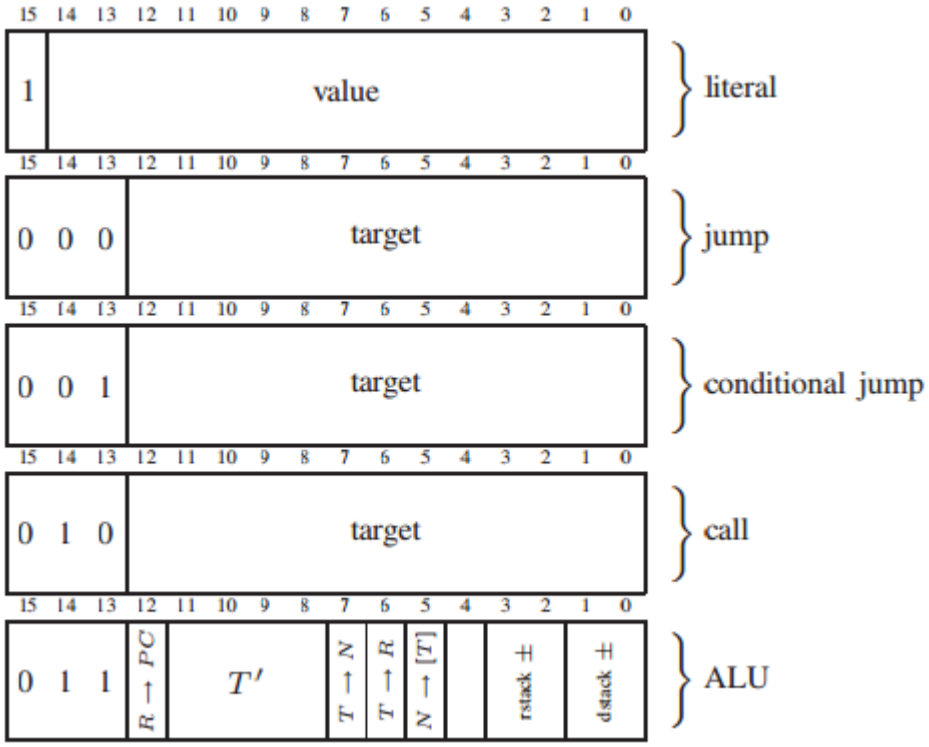

Figura 4: Conjunto de instruções para J1.
Fonte: (BOWMAN, 2010).

Em *Forth*, a instrução do tipo literal normalmente indica que a instrução a seguir é um valor a ser carregado e não uma instrução a ser executada, fazendo que a máquina gaste dois ciclos de relógio para carregar um valor na pilha. No caso da *J1*, o primeiro *bit* da instrução indica se a instrução é literal ou não, sendo o valor carregado dos demais 15 *bits* da instrução. No caso de carregamento de um valor que necessite do *bit* mais significativo, o compilador carrega os 15 *bits* do imediato com o valor invertido, e após executa uma instrução de inversão no topo da pilha. Desta forma, a máquina passa a carregar valores de até 15 *bits* em um ciclo de relógio.

Todos os endereços de chamada e salto são de 13 *bits*, limitando o tamanho do código para 16 KB. As principais otimizações encontradas neste tipo de instrução estão na ausência de salto relativo, o que impede a ocorrência de saltos para fora do limite de código pelo compilador. Outra otimização é encontrada na instrução nativa do *Forth* chamada *branch0*, onde em um único ciclo a máquina executa um *pop* no topo da pilha de dados e caso este seja igual a zero, passa o valor contido nos 13 *bits* para o PC.

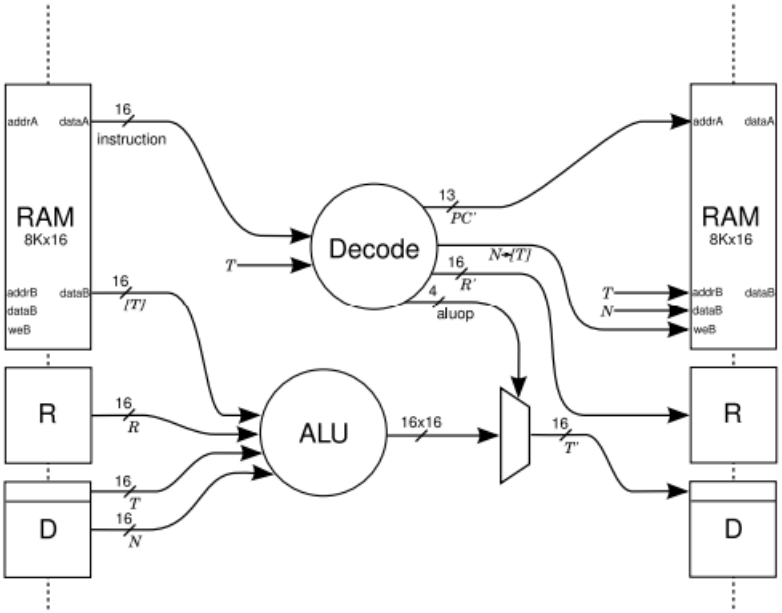

Figura 5: Diagrama de blocos da J1.
Fonte: (BOWMAN, 2010).

O tipo de instrução ULA é feita de forma que a execução da ULA não dependa de nenhum outro campo da instrução, tornando possível que esta unidade execute em paralelo com as demais unidades de *fetch* e decodificação, demonstrado no diagrama de blocos da Figura 5. Nesta instrução, várias unidades funcionais da ULA podem ser atribuídas a uma única chamada, especialmente em casos de retorno de função, onde o valor contido no topo da pilha de retorno pode ser passado diretamente ao PC ao final da execução desta instrução e tornando assim o retorno de função gratuito, ou seja, não requer mais um ciclo só para a chamada de retorno.

code	operation
0	T
1	N
2	$T + N$
3	$T \texttt{and} N$
4	$T \texttt{or} N$
5	$T \texttt{xor} N$
6	$\sim T$
7	$N = T$
8	$N < T$
9	$N \texttt{rshift} T$
10	$T - 1$
11	R
12	$[T]$
13	$N \texttt{lshift} T$
14	depth
15	$N \texttt{u<} T$

Figura 6: Conjunto de instruções específicos da ULA.
Fonte: (BOWMAN, 2010).

Com a junção dos campos da ULA mostrados na Figura 4 e do microcódigo apresentado na figura 6, é possível realizar um vasto número de instruções distintas de aritmética e de dados entre pilhas. Na terminologia mostrada em ambas *T*, *N* e *R* representam Tos, NTos e topo da pilha de retorno. Ao computar um comando da ULA, os campos *dstack* e *rstack*, vistos na instrução de ULA da Figura 4, são tomados como referência para o consumo (*push* ou *pop*) de determinada pilha. Desta forma, o comando referente aos campos de consumo serão aplicados antes do valor computado ser alocado no Tos, o que torna desnecessário o uso de *buffers* para a manipulação de valores na pilha de dados.

Na verdade, todos os campos da ULA mostrados na Figura 4 são efetuados antes da alocação do valor computado, caso haja algo a ser computado. A chamada do *opcode* da ULA pode também ser apenas um carregamento de um dos topos das pilhas, pois este é o único campo desta instrução que permite acessar este recurso para ser usado pelos demais campos, como um carregamento em memória por exemplo. Entretanto, uma atenção especial é necessária ao carregar o código 12 ou *[T]*, vistos na Figura 6, pois este é exclusivamente referido à Tos e é interpretado como endereço de memória. Sendo assim, ao ser carregado, a máquina usa a variável carregada de Tos para buscar o valor desejado na RAM e aloca o valor encontrado em Tos, reescrevendo-a e realizando assim uma leitura em memória. A operação oposta pode ser obtida ativando o campo de *bit* 5 da instrução, sabendo-se que o valor a ser alocado deve estar em NTos e o endereço em Tos.

Além dos códigos de ULA citados anteriormente, existe um código que normalmente não é encontrado em aritméticas de outros processadores, seja de pilha ou de registrador, sendo este o de código 10, visto na Figura 6. Este, na verdade, é uma otimização natural para uma máquina de pilha pois todos os cálculos de endereços ou *index* são feitos na pilha de dados, Sendo assim, em casos onde vetores estão sendo tratados em RAM, o endereço referente ao vetor normalmente sofre adição ou subtração de uma posição por vez. O que requer que a máquina carregue um literal que após a soma será consumido e não poderá ser utilizado novamente em outro cálculo do mesmo tipo. Assim, adição desta instrução na ULA a máquina poupa 1 ciclo de relógio em cada cálculo de endereço, além de reduzir a quantidade de instruções de imediato no código, acarretando em melhor densidade de código.

Ao contrário de boa parte das CPUs que executam *Forth* apresentadas pela comunidade,

a *J1* não é somente uma máquina conceitual para os conceitos herdados de uma determinada máquina de pilha ou com algum tipo de otimização criada pelo autor. A *J1* vai além disto, sendo pensada para uma aplicação real e em consequência é possível quantificar seu desempenho, tornando possível a comparação com máquinas equivalentes existentes no mercado e a avaliação de *benchmarks*.

Quando comparado à versão anterior do sistema, onde mesma função era exercida por um microcontrolador *MicroBlaze* executando *C*, a *J1* otimizou em 62% o tamanho de código, permitindo que sejam adicionadas mais funcionalidades à aplicação. Apesar de não implementar *Forth* por completo, a *J1* consegue alcançar uma boa performance entregando 100 MIPS a 80 MHz, rodando em uma *Spartan 3E FPGA* (BOWMAN, 2010).

Na tabela 1 é mostrada a diferença de tamanho de código em relação a implementação anterior na *Microblaze*, na atual *J1* e na tentativa de construir o mesmo sistema em *Forth* na *Microcore*. Apesar de não ser uma máquina de propósito geral, a *J1* consegue executar a aplicação para qual foi desenvolvida 3X a performance da plataforma baseada em *C* (BOWMAN, 2010).

Tabela 1: Diferença de tamanho entre os códigos das duas versões da aplicação.

Component	Code size (bytes)		
	MicroBlaze	J1	MicroCore
CI^2	948	132	113
SPI	180	104	105
flash	948	316	370
ARP responder	500	112	-
entire program	16380	6349	

Fonte: (BOWMAN, 2010).

3.2 A máquina *Microcore*

Microcore é uma máquina de duas pilhas, com arquitetura *harvard* primariamente desenvolvida para executar as primitivas *Forth* com instruções de 8 *bits*. Ambas pilha de dados e pilha de retorno são alocadas na RAM e podem ser acessadas separadamente, contando ainda com uma memória de programa ROM. Como mostrado na Figura 7, a arquitetura básica da máquina contém os módulos de pilha de dados, ULA, pilha de retorno, caminho de dados central (*uBus*) e Tos, sendo que mais funções podem ser adicionadas ao sistema conforme a necessidade do projeto.

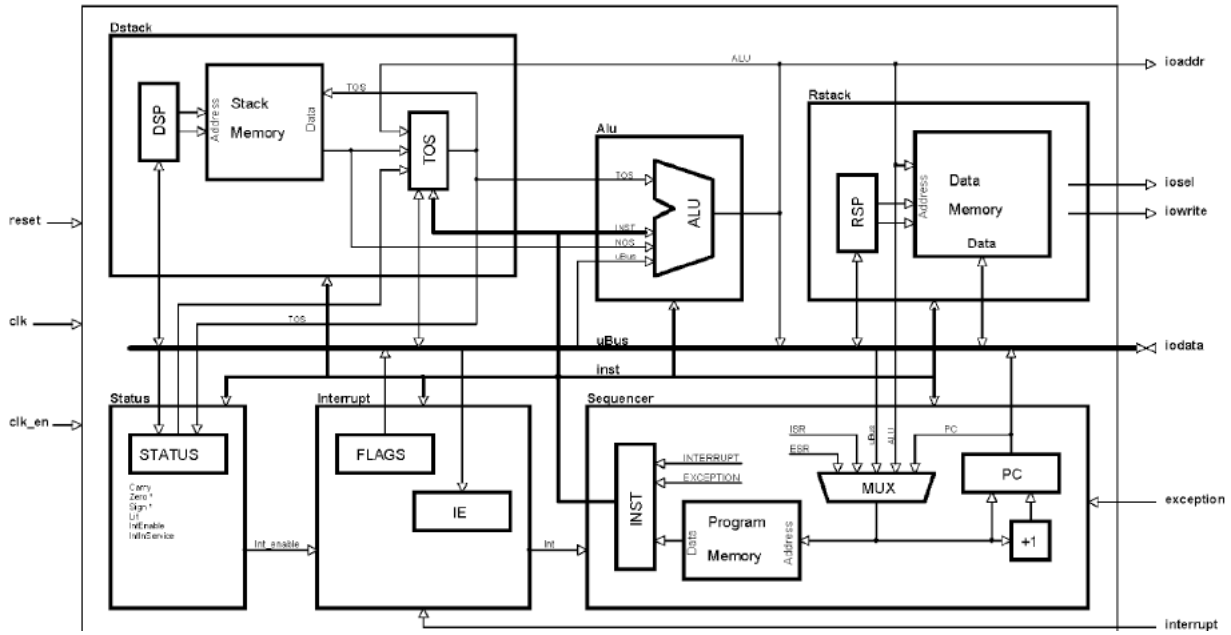

Figura 7: Diagrama de bloco básico para o *MicroCore*.
Fonte: (SCHLEISIEK, 2005).

Como visto na Figura 8, a chamada de literal é dada com o *bit* mais significativo da instrução em 1, sendo os demais *bits* o valor a ser atribuído ao literal. Para variáveis maiores do que 7 *bits*, o compilador utiliza uma série de literais. Desta forma, a CPU reconhece a sequência de literais e começa a montar o topo da pilha de dados lendo cada literal e efetuando um deslocamento à esquerda para que a concatenação possa ser efetivada (SCHLEISIEK, 2004). Caso o primeiro *bit* da instrução não seja 1, a instrução passa a ser *opcode*, contendo na sequência 2 *bits* de tipo, 2 de pilha e 3 de grupo como mostrado na figura.

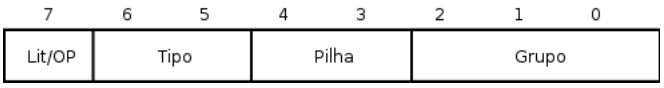

Figura 8: Formato da instrução de 8 *bits*.
Fonte: Modificada pelo Autor (SCHLEISIEK, 2004).

No caso de um desvio, a CPU deve desempilhar o topo da pilha de dados para alocar o endereço de PC. Ao executar uma chamada de função, o PC é carregado para a pilha de retorno enquanto o novo endereço estiver sendo carregado e, após a execução da chamada, a instrução de retorno é responsável por carregar o topo da pilha de retorno para o PC. Como visto na Tabela 2, instruções do tipo ULA realizam operações aritméticas utilizando a pilha de dados como fonte e destino, MEM da acesso à RAM e USR, microcódigo cuja a funcionalidade pode ser programada pelo usuário, pode ser implementada conforme a aplicação necessitar.

Tabela 2: *Opcodes* para o campo de tipo.

Código	Nome	Ação
00	BRA	Desvio, chamada de função e retorno
01	ALU	Operações unárias ou binárias
10	MEM	Acesso a memória e registrador
11	USR	Não usado pelo *MicroCore*, podendo ser implementado pelo usuário

Fonte: (SCHLEISIEK, 2004).

Na Tabela 3 são mostrados os *opcodes* do campo de pilha, onde estão contidos os comandos básicos de *pop* e *push*, sendo os demais campos dependentes do campo tipo mostrados na Tabela 2 e de sinais externos, tornando o *opcode* não ortogonal e complexo. Os demais níveis de complexidade são apresentados pelo campo de grupo apresentado na Figura 8, que quando combinados com os demais campos do *opcode* podem gerar as primitivas *Forth* como mostrado na tabela 4. Como visto, o comportamento de uma determinada instrução depende dos três campos, o que permite que três diferentes módulos diferentes da máquina sejam chamados ao mesmo tempo no momento da execução.

Tabela 3: *Opcodes* para o campo de pilha.

Código	Nome	Ação
00	NONE	Não depende de tipo
01	POP	pilhas $\rightarrow NTos \rightarrow Tos$
10	PUSH	Tos $\rightarrow NTos \rightarrow pilhas$
11	BOTH	Dependente de tipo

Fonte: (SCHLEISIEK, 2004).

Tabela 4: *Opcodes* para o campo grupo e as instruções geradas a partir dos campos tipo e pilha.

Code	Binary-Ops ALU	Unary-Ops ALU BOTH	Complex-Math ALU NONE	Conditions BRA	Branches BRA PUSH	Registers MEM
000	ADD	NOT	MULTS	NEVER	DUP	STATUS
001	ADC	SL	0DIVS	ALWAYS	EXC	TOR
010	SUB	ASR	UDIVS	ZERO	QDUP	RSTACK
011	SSUB	LSR	not used	NZERO	QOVL	LOCAL
100	AND	ROR	LDIVS	SIGN	INT	RSP
101	OR	ROL	not used	NSIGN	IRET	DSP
110	XOR	ZEQU	not used	NOVL	THREAD	TASK
111	NOS	CC	SWAPS	NCARRY	TOKEN	IP

Fonte: (SCHLEISIEK, 2004).

Um dos principais aspectos do *MicroCore* é se aproveitar da implementação em HDL para

gerar uma máquina customizada com as exigências da aplicação que será executada na FPGA, podendo gerar desde uma máquina simplista com poucas funcionalidades até uma máquina de pilha com suporte completo a *Forth*. Apesar disto, não foi encontrado nenhum tipo de aplicação real que se utilize desta máquina. É comentado em (SCHLEISIEK, 2005) sobre um protótipo de compilador que permitiria executar código *C* no *MicroCore*, porém este não encontra-se disponível ou tem qualquer tipo de demonstrações ou resultados apresentados.

3.3 O microporcessador *Novix NC4016*

O *NC4016* foi o primeiro microprocessador *Forth* 16 *bits* a ser fabricado, rodando a 8 MHz e usando cerca de 4000 portas lógicas em tecnologia $3\,\mu\mathrm{m}$ HCMOS. Tinha como objetivo executar aplicações de tempo em tempo real e alta velocidade na execução de *Forth* para programação de propósito geral (PHILIP J. KOOPMAN, 1989). Seu projeto original foi desenvolvido como um protótipo para validar a utilização de *Forh* em uma máquina de pilha física, porém acabou sendo comercializado tendo como foco o mercado de controle embarcado.

Como a tecnologia da época não permitia portar memórias locais ao *chip*, o *NC4016* utiliza dois canais de memória 16 *bits* para acessar as pilhas de forma externa e dedicada. Além dos canais utilizados para comunicar a pilha de dados e pilha de retorno, um canal é utilizado para acessar a RAM. Com isto, dois caminhos de dados são utilizados dentro do microprocessador como visto na figura 9, sendo um deles a interface com as duas pilhas e o outro a interface com a RAM. Este arranjo permite com que as pilhas sejam acessadas paralelamente com a RAM e assim facilita a execução monociclo para determinadas instruções.

Ambas as memórias externas para as pilhas têm 256 posições, sendo referenciadas pelos *buffers* Tos e NTos no caso da pilha de dados e Tors no caso da pilha de retorno presentes dentro do *chip*. Assim os *buffers* só precisam ser atualizados quando há alguma alteração na pilha à qual eles referenciam, o que otimiza o tempo de acesso ao recurso se comparado a um carregamento direto da memória externa. Porém, a utilização de memórias externas faz com que a CPU não tenha controle sobre sobrecarga de dados das pilhas.

Para aplicações com múltiplos processos, uma memória maior que 256 registradores pode ser utilizada. Com isto, cada processo pode mapear através das portas entrada e saída (E/S)

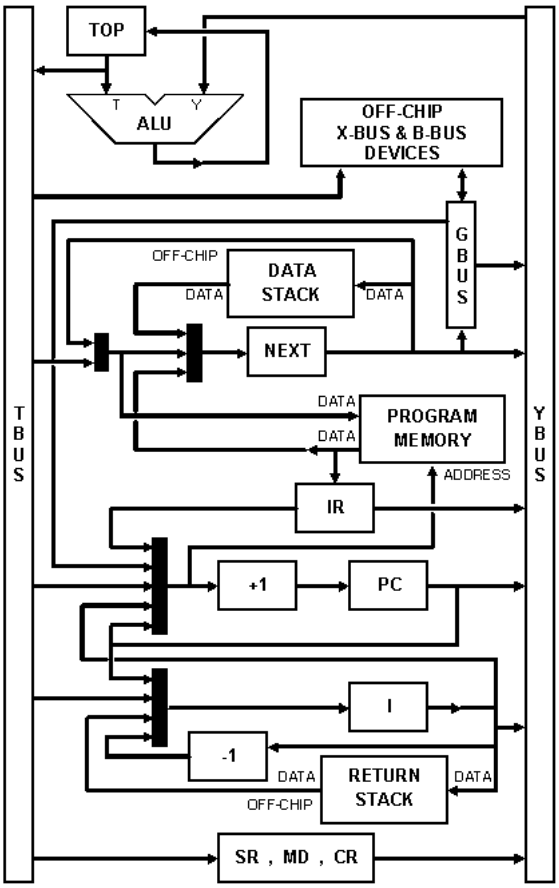

Figura 9: Diagrama de bloco do *Novix NC4016*.
Fonte: (PHILIP J. KOOPMAN, 1989).

uma determinada área da memória externa para usar como pilha, limitando os *buffers* internos somente a estas áreas. Assim, as pilhas estão sempre carregadas e a troca de contexto só precisa fazer os *buffers* apontarem para a pilha referente ao próximo processo a ser executado, reduzindo o *overhead*.

O *NC4016* tem 2 duas portas de *E/S*, sendo uma porta B com 16 *bits* de barramento, e outra X de 5 *bits*. Estas portas permitem que dispositivos externos tenham acesso ao microprocessador sem interromper outros barramentos de memória, e também podem ser usadas para estender o endereço de memória como citado anteriormente. Por padrão a máquina opera com tamanhos fixos de memória de 16 *bits*, fazendo assim com que o uso de estruturas de dados com tamanho diferente de 16 *bits* tenham que ser implementados em *software*.

Quanto ao conjunto de instruções, o *NC4016* é o pioneiro no uso de instruções não codificadas, característica esta que foi herdada pela máquina *J1*, vista na Seção 3.1, e por muitas outras máquinas de pilha sucessoras a *NC4016*. Como visto na Figura 10 há cinco tipos de instrução,

onde por padrão o primeiro *bit* mais significativo indica uma chamada de função e os demais *bits* são interpretados como endereço caso seja chamado. No caso de um desvio, o campo *cs* é responsável por indicar a máquina se a chamada incondicional (*cs*=10), um laço referente ao registrador de índice (*cs*=11) ou ainda se o salto é condicional ao Tos ser igual a zero (*cs*=01).

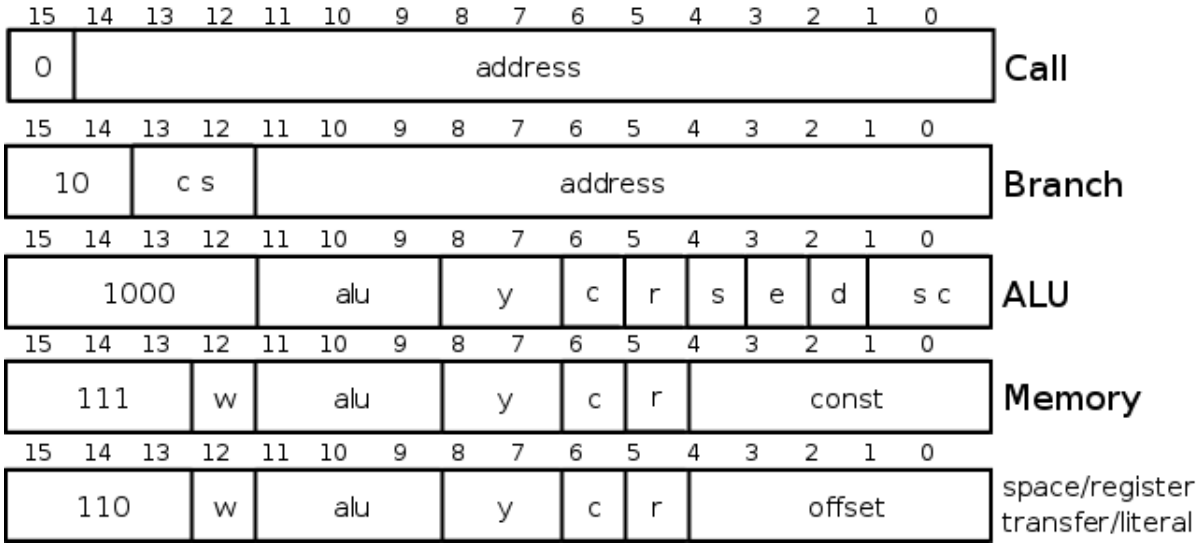

Figura 10: Conjunto de instruções do *Novix NC4016*.

Nas demais instruções, não descritas na Figura 10, uma série de *bits* de controle se repete:

- *w*, controla a leitura ou escrita de memória;
- ULA, controla a ação da ULA caso haja uma operação unária ou binária, ou se responsabiliza por carregar o valor certo para ser usado pelos demais campos da instrução, descritos na Tabela 5.

Tabela 5: *Opcodes* de controle da ULA

000	carrega Tos
001	Tos *and y*
010	Tos - *y*
011	Tos *or y*
100	Tos + *y*
101	Tos *xor y*
110	*y* - Tos
111	*y*

Fonte: (PHILIP J. KOOPMAN, 1989).

- *y*, controla a segunda entrada da ULA conforme visto na Tabela 6.

Tabela 6: Entradas secundarias para ULA

00	NTos
01	NTos com *carry flag*
10	registrador MD
11	registrador SR

Fonte: (PHILIP J. KOOPMAN, 1989).

- *c*, copia o valor em Tos para NTos;
- *r*, retorno de subrotina (Tors $\rightarrow PC$);
- *s*, atua juntamente com o campo *c* para executar *push* ou *pop* na pilha de dados;
- *e*, habilita o deslocamento de 32 *bits*;
- *sc*, controla o comportamento do deslocamento executado pela ULA, visto na Tabela 7:

Tabela 7: *Opcodes* para deslocamento

00	não efetua o deslocamento
01	Deslocamento lógico à direita
10	Deslocamento a esquerda
11	Deslocamento aritmético à direita

Fonte: (PHILIP J. KOOPMAN, 1989).

Instruções de memória demoram dois ciclos de relógio, um ciclo para a busca da instrução e outro para a operação (PHILIP J. KOOPMAN, 1989). O campo ULA neste caso pode realizar cálculos diretamente no endereço a ser utilizado a partir do campo de constante, visto que o endereço nestes casos sempre é pego em Tos. Este comportamento permite que cálculos de endereços não necessitem ser executados na pilha de dados, o que otimiza sua utilização e economiza alguns ciclos de relógio.

A última instrução mostrada na Figura 10 pode ser considerada como a mais flexível, pois permite o carregamento de literais de 5 *bits* em um ciclo de relógio ou literais de 16 *bits* em dois ciclo de relógio. Outra utilidade interessante desta instrução está montagem de valores de 32 *bits* no espaço reservado ao usuário, que se encontra nas primeiras posições da ROM, evitando que várias operações de aritmética, movimentação e memória sejam necessárias para realizar a mesma função.

O *Novix NC4016* além de pioneiro em seu nixo de arquitetura de computadores, traz con-

sigo otimizações que são adotadas em boa parte das máquinas de pilha subsequentes. Teve relevante atuação no mercado como microcontrolador utilizado em impressoras, *switches* e aparelhos de telecomunicação em geral (PHILIP J. KOOPMAN, 1989).

3.4 A CPU *Mill*

A CPU *Mill* é uma arquitetura de computadores de proposito geral onde a banco de registradores é organizada de forma circular e tamanho ajustável, utilizando-se de *Very Large Instruction Word* (VLIW) como paradigma de instrução. Sob o desenvolvimento de Ivan Godard e sua empresa chamada *Out Of The Box*, a máquina encontra-se em desenvolvimento no período em que este trabalho é escrito, porém, os desenvolvedores disponibilizam uma grande quantidade de informações a respeito do funcionamento da máquina (GODARD, 2015).

Como dito anteriormente, a máquina apresenta seu banco de registradores na forma circular. Com isto, cada processo a ser executado informa à máquina qual o tamanho da janela de registradores (*belt*) que necessita. Este *belt* se comporta como pilha circular, onde a última posição é sobrescrita pelo novo valor caso todas as posições do *belt* já esteja preenchida. Por se tratar de uma máquina de dois operandos, qualquer posição do *belt* pode ser acessada via instrução. Como o *belt* é circular o seu endereçamento é feito com base na última posição ao topo do *belt* que sofreu alteração, o índice de endereçamento dos registradores muda a cada inserção no *belt*, trazendo assim um paradigma de registrador temporal.

Este modelo de uso do banco de registradores encaixa-se no paradigma de pilha com dois operandos. Por um lado, não há a necessidade de renomeação pois os registradores são temporais, além de não haver dependência explicita entre registradores. Por outro, não há a necessidade de manipulações em pilhas para que os valores desejados se encontrem no topo, pois todas as posições são acessíveis.

Ao alocar um *belt* a máquina atribui a ele um identificador denominado *frame*, cuja a função é identificar o *belt* caso este seja guardado em memória e precise ser carregado novamente no futuro. Em outras máquinas de pilha mais complexas é comum encontrar o uso de *frames* para referenciar uma pilha a determinado processo, possibilitando assim uma troca de contexto organizada.

Quando há uma chamada de função, a instrução especifica quais valores serão usados como entrada para esta chamada. Estes valores são copiados na ordem em que se encontram no *belt* que executou a chamada (*caller*), para um novo *belt* que será da função que foi chamada (*callee*). Após isto, o *frame* contendo o *belt* do *caller* é guardado em memória e o *frame* do *callee* toma seu lugar. Após a computação, o *callee* chama o *frame* de seu *caller* e o carrega no *belt* da CPU, deixando carregados os valores de retorno como mostrado na Figura 11.

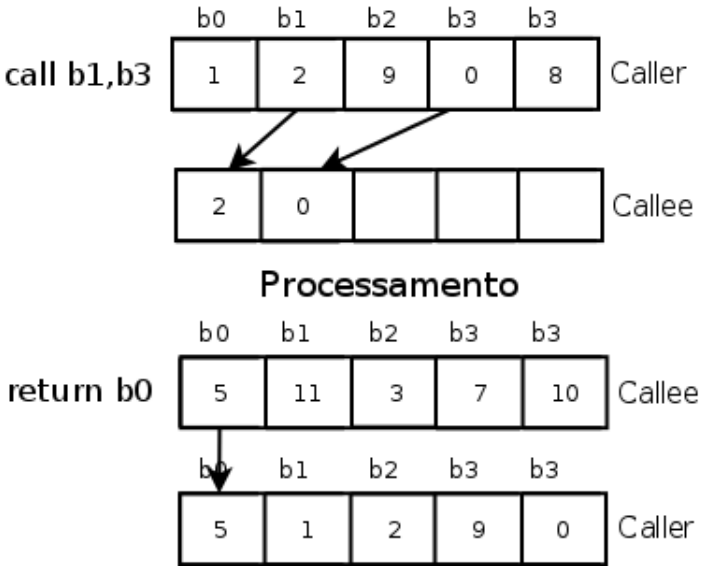

Figura 11: Manipulação dos *belts* ao chamar uma função.

Na alocação de um novo *belt*, além de especificar o tamanho que o *belt* terá, é atribuído a ele uma memória local de tamanho fixo chamada de *scratchpad*. *Scratchpad* é uma memória especial que se localiza na CPU, se assemelhando ao *belt*, porém é acessada como uma se fosse uma memória externa e é limpa assim que um *belt* é alocado. Seu comportamento é o de uma pilha, porém são necessários pelo menos três ciclos de relógio entre escrita e a leitura do mesmo valor no *scratchpad*.

O comportamento do *belt* é bem peculiar se comparado as demais máquinas de pilha analisadas neste Capítulo. O comportamento desta lembra um pouco uma máquina de pilha sem pilha de retorno, pois o *scratchpad* não se comporta como uma pilha de retorno mas sim como uma pilha de variáveis. Porém há uma alocação de um novo *belt* cada vez que uma função é chamada, diferindo-a bastante das demais implementações.

Por um lado as chamadas ficam mais organizadas pois somente os parâmetros são passados para o *belt* e só o retorno é adicionado ao *belt* do *caller*, se comparado a uma máquina de pilha

normal onde ambos *caller* e *callee* dividem a mesma pilha. Por outro lado, o uso extensivo de chamadas pode gerar uma grande quantidade de *frames* a serem guardados em memória, o que pode ser considerado sobrecarga. O tempo necessário para manipular todas as montagens e retornos de *frames* e seus *belts* pode ser muito custoso para o sistema em casos extremos de chamadas de função.

Embora a maioria das máquinas de pilha, especialmente as que rodam *Forth*, faça uso extensivo de chamadas de função, a CPU *Mill* foi pensada para a execução código *C*. Com isto, é seguro presumir que as chamadas de função não serão tão frequentes em um código *C*, como são em código *Forth*. A chamada de novos *belts* a cada *call* pode fazer sentido e vir a ser efetivo na execução de *C*.

É difícil fazer uma avaliação mais profunda da CPU *Mill* tendo em vista que o projeto esteja em desenvolvimento e nenhum tipo de *benchmark* ou demonstração esteja disponível. Os desenvolvedores afirmam que seu produto terá desempenho *2,3X* maior e menos consumo que uma máquina de registradores *superscalar* por quase a metade do preço. Tal *superscalar* não é especificada e tais afirmações não podem ser mais que estimativas, visto que o produto final não esta concluído. Resta esperar para que a primeira versão seja comercializada e avaliações de desempenho possam ser feitas, validando as afirmações dos desenvolvedores e a efetividade das otimizações.

3.5 Súmula

Após a análise dos principais aspectos de cada máquina abordadas neste capitulo, resta fazer uma comparação direta delas. A Tabela 8 mostra um apanhado das principais características de cada uma das máquinas analisadas, gerando uma comparação entre as abordagens adotadas em cada máquina.

Dentre as máquinas analisadas, todas as que implementam *Forth* apresentam uma CPU monociclo e uma pilha de dados, pilha esta que é um dos pré-requisitos para a linguagem. A CPU *Mill*, apesar de não ser explicitamente denominada de máquina de pilha pelos autores, tem em seu funcionamento básico o uso de estruturas de pilha em todas as suas estruturas de memória em CPU, classificando-a assim como uma máquina de pilha.

Tabela 8: Tabela comparativa entre as máquinas analisadas nesta capitulo.

	J1	Microcore	Novix NC4016	Mill CPU
Bits da arquitetura	16	8	16	64
Tamanho da pilha de dados	33	Variável	Variável	Variável
Apresenta pilha de retorno	Sim	Sim	Sim	Não
Alocação das pilhas	Local(CPU)	RAM	Memória externa dedicada	Local(CPU)
Linguagem	Forth	Forth	Forth	C
Número de operandos	0	0	0	>1
Tipo de implementação	VHDL	VHDL	ASIC	ASIC
Aceita modificações	Não	Módulos podem ser adicionados e há possibilidade de se programar um	Bloco de memórias externas dedicadas podem ser trocados	Não
Ciclos de execução	Monociclo	Monociclo	Monociclo	Multiciclo

Um ponto importante na escolha destas máquinas para a analise teórica é a alocação das pilhas usadas pela CPU. Na máquina *Microcore*, ambas as pilhas são alocadas em memória, o que traz flexibilidade à CPU pois as pilhas podem ser alocadas com tamanhos variáveis, além da vantagem das pilhas já se encontrarem escritas em memória em trocas de contexto.

Por outro lado tem-se a máquina *J1*, que aloca sua pilha em um banco de registradores. Com isto, esta máquina perde em flexibilidade comparada à máquina *Microcore*, pois suas pilhas tem um tamanho máximo fixo. Contudo, a forma de acesso às pilhas é mais rápido se comparado ao acesso à memória feito pela máquina *Microcore*.

Por fim a máquina *NOVIX NC4016* apresenta-se como uma máquina hibrida se comparada as maquinas citadas anteriormente, visto que esta usa de uma memória externa dedicada para a alocação de suas pilhas. Com isto, esta máquina consegue alcançar o mesmo grau de flexibilidade encontrado na alocação de pilhas em memória, contando ainda com um barramento otimizado para o acesso da memória externa pela CPU, diminuindo o tempo de acesso ao recurso.

4 ANÁLISE TEÓRICA SOBRE AS MÁQUINAS DE PILHA

Neste capítulo, serão analisados os conceitos básicos sobre a forma de acesso e uso das pilhas, além da quantidade na CPU. A partir desta análise, serão propostas melhorias para cada uma destas análises, levantando quais as possíveis vantagens e desvantagens seriam encontradas nestas implementações.

4.1 Quantidade de pilhas

Conforme apresentado no Capítulo 2, a maior parte das máquinas de pilha são desenvolvidas para executar *Forth*. Este pode ser um dos fatores que contribuem para o reduzido interesse de pesquisa acadêmica e industrial sobre as arquiteturas do tipo pilha, citadas por ambos (BAILEY, 1996) e (PHILIP J. KOOPMAN, 1989). Além disto, a curva de aprendizado para a programação destas CPUs, especialmente no nível de programação encontrado em *Forth* que é muito similar à linguagem do *assembly* utilizado em máquinas de registradores.

4.1.1 Máquina de duas pilhas

Apesar da forma de programação, a máquina base utilizada pela linguagem e pela máquina virtual *Forth* apresenta um conjunto de características interessantes. Para começar, o uso de duas pilhas, como visto no capitulo 2, facilita o controle do fluxo de dados e os separa do fluxo de endereços que é detido pela pilha de retorno. Para a implementação de máquinas *Forth*, a CPU enxerga e trata as pilhas como se fossem entidades de memória distintas uma da outra, mesmo estas sendo muitas vezes um mesmo bloco de registradores físico.

Esta característica é reforçada por instruções primitivas encontradas na linguagem *Forth* que explicitamente fazem transferência de dados entre as duas pilhas, como `R@` (copia o topo da pilha de retorno para o topo da pilha de dados). Desta forma, a ULA terá de interagir quase que exclusivamente com a pilha de dados e somente em casos transferência entre pilhas a pilha de retorno será requisitada pela ULA.

Outro fator importante desta abordagem é a possibilidade de independência da pilha de retorno em relação à ULA quanto a operações de chamada e retorno de função. Isto porque nestes casos, a única interação necessária é efetuada entre a pilha de retorno e o registrador de PC, como visto na Figura 12. Assim, o *hardware* gerado para detectar e efetuar as trocas de dados entre o PC e a pilha de retorno, nos casos de retorno, e a chamada de função será reduzido se comparado à máquinas que utilizam somente a pilha de dados para manipular endereços de retorno.

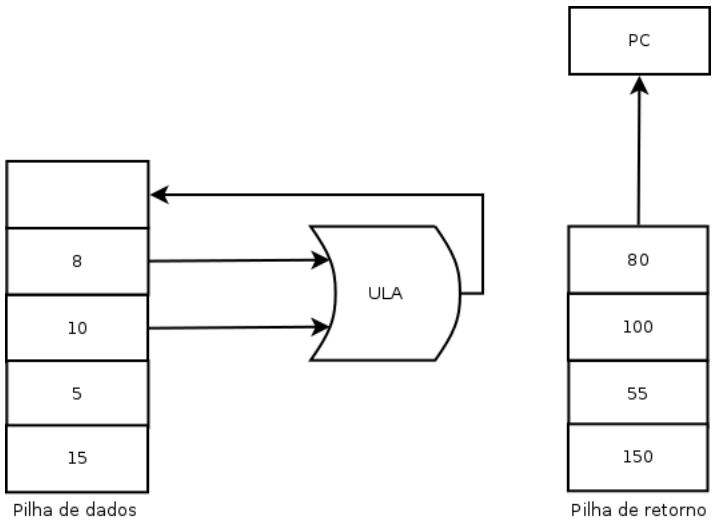

Figura 12: Visualização de uma iteração entre as pilha de dados e retorno com a ULA e o registrador de PC.

4.1.2 Máquina de duas pilhas *vs* máquina de uma pilha

Embora a composição com duas pilhas seja a mais vista em máquinas de pilha do passado, outras configurações de pilhas são possíveis. A configuração mais simples possível é a utilização de uma única pilha para lidar com todo o fluxo de dados da CPU. Como visto no capítulo 2, esta configuração apresenta uma desvantagem importante que é a desorganização entre dados aritméticos/lógicos, endereços e parâmetros em única pilha.

Por um lado, o *hardware* é consideravelmente simplificado pois todo o fluxo de dados é endereçado a uma única pilha. Como visto na Figura 13, todo e qualquer dado ou endereço que deva ser manipulado pela CPU é obrigatoriamente alocado na pilha de dados e tem a ULA como interface de saída para a memória principal e para o registrador de PC.

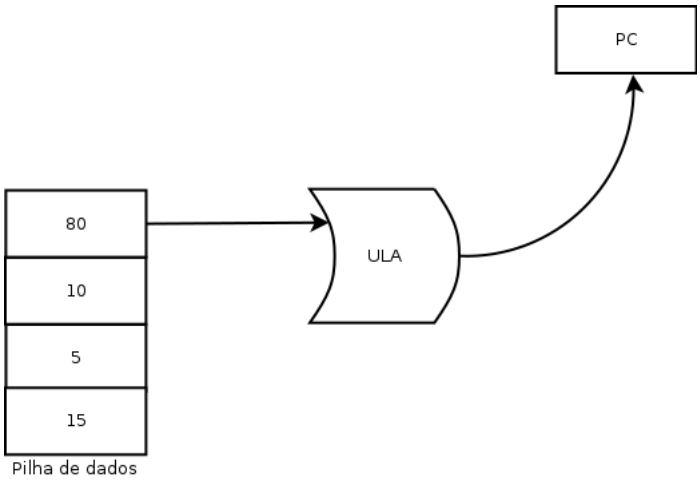

Figura 13: Visualização de uma escrita de endereço no registrador de PC.

Por outro lado, chamadas de função não são otimizadas como em máquinas com mais pilhas, pois não há a possibilidade de realizar retornos de função em paralelo com outra instrução visto que a pilha já esta sendo usada. Outro problema significativo é que uma máquina com apenas uma pilha tende a necessitar de uma pilha maior que a pilha de dados de uma máquina com mais de uma pilha executando o mesmo algoritmo, pois o fluxo de dados para duas ou mais pilhas é convergido à apenas uma pilha.

4.1.3 Máquinas com múltiplas pilhas

Já o uso de mais de duas pilhas em um máquina permite que alguns fluxos de dados sejam separados para obter-se assim um melhor controle sobre determinados tipos de dados pelo programador. Estas pilhas extras podem ser utilizadas para guardar parâmetros entre *caller* e *callee* em chamadas de função, contador de laço, endereços de entrada e saída, podendo ser usada até como uma pilha global.

Apesar de poder melhorar consideravelmente o manejo e organização de determinados tipos de dados, esta abordagem apresenta algumas desvantagens expressivas sobre as demais

abordagens citadas anteriormente. Para começar, cada pilha introduzida à arquitetura da CPU requer um maior número de registradores, um caminho de dados maior e uma lógica de controle sobre as pilhas do sistema mais complexa. Assim, a área final cresce linearmente conforme mais pilhas são adicionadas à CPU, bem como um aumento no consumo energético.

Além disto, a existência de três ou mais pilhas não significa necessariamente que estas pilhas serão usadas em paralelo. Isto é bastante evidente para pilhas de parâmetros e outras pilhas que sejam usadas para dados, pois normalmente a ULA só tem acesso direto à pilha de dados, e para que qualquer operação seja feita sobre os dados de uma das pilhas citadas anteriormente, o dado tem que ser transferido de sua pilha original para a pilha de dados. Isto acaba por gerar uma considerável quantidade de sobrecarga de transferência de dados no sistema, pois o dado que é alocado fora da pilha de dados foi copiado originalmente da pilha de dados e, quando este tem de ser usado novamente, precisa ser copiado de volta para a pilha de dados novamente.

Este problema pode ser parcialmente contornado utilizando-se de múltiplos caminho de dados para acessar cada uma das pilhas, contando ainda com um multiplexador que possibilitasse o acesso direto da ULA aos topos de todas pilhas. Esta abordagem não é efetiva pois amplifica ainda mais o problema de área e consumo no circuito projetado, além do fato que nem sempre estes recursos serão extremamente explorados pelo programador, tornando ocioso um recurso consideravelmente caro ao projeto da CPU.

Outra desvantagem do uso de três ou mais pilhas é o tempo de troca de contexto, visto que a cada ocorrência deste evento todas as pilhas da CPU devem ser copiadas em memória, e vice versa ao chamar novamente o processo. Isto acaba por aumentar em muito o tempo de troca de contexto, sem contar que os dados já poderiam estar alocados em pilha na RAM pelo programador como citado anteriormente.

Contudo, (PHILIP J. KOOPMAN, 1989) comenta em suas previsões para as próximas gerações de máquinas de pilha sobre o uso de uma terceira pilha para a alocação de variáveis locais. Apesar deste recurso ser trivial em uma máquina de registradores devido ao endereçamento explicito de registradores, a atomicidade dos dados na manipulação da pilha não permite um controle explicito sobre um determinado dado.

Como visto anteriormente, a adição desta nova pilha pode ser driblada com abordagens na

programação da máquina. (PHILIP J. KOOPMAN, 1989) propõem uma técnica de otimização para código nativo à pilha chamada de *intra-block-scheduling*. Em súmula, esta técnica tenta eliminar qualquer operação de escrita em memória desnecessária, e manter cópias de variáveis em pilha o maior tempo possível. Desta forma, o programador procura manter suas variáveis dentro da pilha conforme a sua necessidade, quando a pilha estiver quase cheia e mais dados precisarem ser alocados na mesma, as variáveis que são utilizadas com menor frequência ou que não serão usadas por algum tempo devem ser passadas para a memória . A técnica de Koopman é analisada e testada mais profundamente por (BAILEY, 1996), confirmando a efetividade do uso da técnica e derrubando a necessidade de uma terceira pilha para este propósito.

Por fim, a Tabela 9 mostra uma breve súmula das principais vantagens e desvantagens das abordagens analisadas nesta seção. Concluindo-se que no geral o uso de duas pilhas é a abordagem mais efetiva na implementação de uma máquina de pilha.

Tabela 9: Comparação entre número de pilhas.

Quantidade de pilhas	**Vantagens**	**Desvantagens**
1	*Hardware* simplificado	Desorganização entre dados e endereços
		Requer uma pilha maior
2	Fluxo de dados separado do fluxo de endereços de retorno	Controle de duas pilhas distintas
	Operações lógico/aritméticas realizadas em paralelo a chamadas e retornos de função	
3 ou mais	Pilhas exclusivas para determinado tipo de dado	Controle de três ou mais pilhas distintas
		Overhead
	Melhor organização	Aumento expressivo de hardware

4.2 Quantidade de operandos

Por padrão, a máquina virtual *Forth* apresenta zero operandos, também chamado de máquina pura. Nesta configuração a máquina se contém basicamente ao acesso do topo de ambas as pilhas de dados e retorno. Sendo assim, qualquer operação primitiva de *Forth* que efetue trocas de posições na pilha, como *swap* por exemplo, ou que opere com alguma posição da pilha que não esteja no topo, como *pick*, requer que parte da mesma seja desempilhada para o acesso da

posição desejada.

Esta situação pode ser facilmente contornada pela máquina virtual *Forth* e por implementações de CPUs que alocam suas pilhas diretamente na memória RAM, pois as operações podem ser efetuadas diretamente sobre os endereços de memória referentes a estas posições da pilha. Contudo, em implementações de máquinas onde o acesso da ULA à pilha seja dado somente por *buffers* de topo e próximo ao topo da pilha, o acesso endereçado à pilha não é possível.

Para evitar o desempilhamento em implementações deste tipo, pode-se fazer o uso de mais de dois *buffers* para o acesso da ULA. Para determinar a quantidade de *buffers* a serem usados, pode-se analisar as instruções primitivas *Forth* que necessitem de posições da pilha além das de topo, vistos na Tabela 10.

Tabela 10: Relação de instruções primitivas *Forth* que acessam elementos profundos na pilha.

Primitiva Forth	Profundidade	Representação	Funcionalidade
OVER	2	N1 N2 - N1 N2 N1	Copia ao topo a segunda posição ao topo da pilha
PICK	n	... N1 - ...N2	Copia ao topo o N1-ésimo elemento da pilha
ROLL	n	... N1 - ...N2	Traz ao topo o N1-ésimo elemento da pilha
ROT	3	N1 N2 N3 - N2 N3 N1	Traz ao topo o terceiro elemento da pilha
SWAP	2	N1 N2 - N2 N1	Troca de lugar os dois elementos mais ao topo da pilha

Fonte: (PHILIP J. KOOPMAN, 1989).

Na Tabela 10, são listados todos os instruções primitivas *Forth* que utilizam elementos da pilha que não se encontram no topo dela. Nesta tabela, o campo *Representação* apresenta o estado do topo da pilha antes e depois da execução da instrução, sendo o caractere - um separador de antes e depois, e os elementos da pilha são representados por *N1,2,3,...*, sendo que os mais à direita estão mais ao topo da pilha.

As instruções *PICK* e *ROLL* são consideravelmente críticas em relação ao acesso da pilha pois permitem ao programador o acesso a qualquer posição da mesma, tornando necessário o desempilhamento de todo o topo da pilha até a posição desejada. Permitindo assim um acesso de custo linear em relação à ciclos de execução e quantidade de dados desempilhados conforme,

a profundidade do elemento desejado. Problema este que não é encontrado em implementações de máquina de pilha onde as pilhas se encontram em memória.

Nas demais instruções primitivas *Forth* mostradas na Tabela 10, é amostrado um máximo de três elementos em relação ao topo da pilha sendo utilizados. Com isto, é possível assumir que três *buffers* sejam suficientes para diminuir a quantidade de desempilhamentos oriundas destas instruções listadas. Quanto as instruções que acessam *n* elementos da pilha, o melhor a se fazer é evitar o acesso de posições maiores que três em relação ao topo da pilha.

Utilizando otimizações de acesso à pilha por meio de *buffers* não só otimiza a execução de certas operações sobre a pilha como também alça a máquina de zero operandos ao patamar de máquinas de um operando. Apesar de não apresentar uma forma tão efetiva de endereçar qualquer posição da pilha, este arranjo de *buffers* garante uma boa flexibilidade ao programador sem precisar endereçar explicitamente posições da pilha em instruções, tão pouco a adição de um hardware especifico para a busca do endereço desejado.

Koopman comenta que a abordagem de zero operandos pode vir a diminuir o tamanho de programa devido a maior flexibilidade, porém, é necessário um ciclo de relógio maior ou um estágio de pipeline especifico para a busca do operando na pilha. Com isto, pode-se presumir que ao melhorar o desempenho do acesso à posições que não sejam somente a de topo da pilha de dados em uma máquina pura, não há uma relação de custo e beneficio que favoreça a implementação de uma máquina de um operando em relação à uma de zero.

Por fim, Koopman comenta sobre o conceito de máquina de dois operandos como sendo a mais flexível de todas as máquinas de pilha, assemelhando-se à máquinas de registradores e necessitando de um *hardware* mais complexo que as demais máquinas de pilha. Isso se dá pois assim como em uma máquina de um operando, é necessário que os operandos sejam localizados e carregados na pilha para que qualquer computação possa ser feita.

Este tipo de máquina não teve tanta atenção por parte dos desenvolvedores no passado, visto que em nenhuma das fontes utilizadas nesta pesquisa há uma descrição ou análise detalhada sobre o funcionamento da mesma, tão pouco investigações em busca de otimização. Este fato é ressaltado pelo fato de que *Forth* implementa apenas uma máquina pura e pode emular instruções de um operando (PHILIP J. KOOPMAN, 1989), não dando suporte nativo à implementações de

dois operandos.

Em sua base, uma arquitetura de dois operandos necessita acessar duas posições da pilha para poder suprir a ULA em uma instrução aritmética. Ao término da computação da ULA, o resultado pode ser alocado tanto no topo da pilha, quanto pode sobrescrever uma das posições que foram usadas como operandos. Em uma implementação onde há somente a sobrescrita do resultado em uma das posições dos operandos, é necessário que um *hardware* extra seja utilizado na saída da ULA, de forma a permitir a escrita do resultado na posição desejada.

Com isto, é mantido um comportamento similar à máquina de um operando, com a diferença de que o segundo operando não é necessariamente o topo da pilha e de que o resultado não sobrescreve sempre a mesma posição. Por outro lado, há um acréscimo significativo no *hardware* se comparado a uma máquina pura, fazendo com que ou o ciclo de relógio aumente ou que a escrita do resultado seja executada em uma etapa de *pipeline*. Isto mais a etapa de *pipeline* inerente ao carregamento dos dois operandos citado anteriormente torna o *pipeline* da máquina descrita muito semelhante a um *pipeline* genérico para uma máquina de registradores, que normalmente contém uma etapa de busca de instrução, decodificação de instrução, execução, acesso à memória e escrita em registrador (DAVID A. PATTERSON, 2005).

Nesta comparação, as etapas de busca de instrução, acesso à memória e execução tem basicamente o mesmo funcionamento em ambas as arquiteturas de pilha e registrador. A etapa de decodificação em uma máquina de registradores costuma ser responsável tanto pela decodificação da instrução, quanto pela leitura dos operandos no banco de registradores. Já em máquinas de pilha, boa parte das implementações físicas utilizam codificação *hardwired*, excluindo assim a necessidade de decodificação. Quanto ao carregamento dos operandos, ambas as máquinas com um ou dois operandos necessitam realizar o acesso a pilha, enquanto que na máquina pura não é necessária uma etapa de decodificação pois o topo da pilha está sempre carregado em *buffers* e a pilha não precisa ser acessada diretamente.

Já a etapa de escrita em registrador pode vir a ser utilizada somente em máquinas de dois operandos onde um dos operandos é sobrescrito, como citado anteriormente, pois a posição específica do operando deve ser localizado na pilha e então sobrescrita, similar ao que ocorre em uma máquina de registradores. Já em implementações de qualquer tipo de máquina de pilha onde o resultado seja sempre escrito no topo da pilha, o acesso ao recurso é feito através de *buffer* e a

etapa de escrita em registrador não se faz necessária. Assim, toda a operação que resulte em um valor sendo alocado na pilha de dados necessariamente efetuará um *push* e escreverá o resultado nesta nova posição alocada.

Este comportamento acaba por quebrar o fluxo natural das operações de pilha vistas nas máquinas puras, pois em máquinas de zero operandos os dois elementos mais ao topo da pilha são removidos e somente um valor é alocado de volta, acarretando em um desempilhamento da mesma. No caso desta máquina de dois operandos descrita anteriormente, todas as operações que não sejam necessariamente operações de desempilhamento com um *pop*, por exemplo, acabam por empilhar valores na pilha de dados.

Assim, o tamanho da pilha de dados tende a ser maior em uma máquina deste tipo se comparada à uma máquina pura. Além disto, o programador tem que controlar manualmente o desempilhamento da máquina, correndo o risco de se obter programas maiores e gastando uma quantidade desnecessária de ciclos de execução efetuando o desempilhamento se comparado à utilização de uma máquina pura.

Este comportamento pode ser evitado utilizando uma pilha circular, semelhante ao *belt* utilizado por (GODARD, 2015). Desta forma, a pilha pode crescer infinitamente, sobrescrevendo a sua própria base, desvencilhando-se de qualquer operação de desempilhamento e excluindo a necessidade de controle sobre estouro de pilha.

4.3 Tamanho de pilha

Tomando como base o que foi apresentado no Capítulo 2, não há um tamanho ideal de pilha que possa ser adotado como base para qualquer implementação de máquina de pilha. Este tamanho é dado conforme a área de aplicação da CPU e as especificações da arquitetura implementada, variando de implementação para implementação.

5 IMPLEMENTAÇÃO E FUNCIONAMENTO DAS MÁQUINAS

Neste capítulo, são descritas todas as implementações referentes ao Trabalho de Conclusão de Curso. Também serão analisadas tomadas de decisão para aspectos importantes das arquiteturas idealizadas, explicando as otimizações e vantagens pretendidas.

5.1 Visão geral

Para este trabalho foi utilizado a biblioteca *SystemC* de versão 2.1.3 incluindo TLM. São implementadas duas máquinas de pilha, sendo que ambas implementam duas pilha, uma pilha de dados e outra de retorno. Quanto ao número de operandos, uma máquina é implementada com nenhum operando enquanto a outra é implementada com dois operandos, seguindo a análise descrita no capítulo anterior.

Ambas as máquinas implementam instruções de 32 *bits* e duas pilhas de até 16 posições que juntas totalizam um total máximo de 32 registradores em CPU. As memórias das máquinas são dispostas conforme a arquitetura de *Harvard*, contendo uma memória RAM para manipulação de dados e uma memória ROM para programa, que é carregada na inicialização da simulação. Quanto à execução, ambas as máquinas são monociclo.

Todas as implementações tem suporte nativo a chamadas e retorno de funções, operações lógico/aritméticas, acesso a memória principal e carregamento de valores literais. Contudo, nenhuma das implementações apresenta suporte a ponto flutuante e interrupções. Isto porque em todas as máquinas analisadas para este trabalho na Capítulo 3, nenhuma fazia referência a algum tipo de otimização em *hardware* para dados com ponto flutuante e cada uma das abordagens

implementa e trata de formas diferentes a interrupção em *hardware*.

Como não há nenhum compilador específico para as duas arquiteturas que foram desenvolvidas neste trabalho, não há uma forma efetiva de testar o real desempenho da implementação da interrupção em hardware. Quanto ao suporte a ponto flutuante, é possível tratar esse tipo de operações por *software*.

Vale ressaltar que a falta de compilador poderia ser contornada configurando alguma versão de *Forth* para executar na máquina virtual desenvolvida. Contudo, esta abordagem só resolveria metade do problema, pois a linguagem *Forth* só tem suporte à máquina de zero operandos, deixando a máquina de dois operandos sem nenhum tipo de compilador. Outro fator que dificulta a integração com *Forth* é que todas as funções primitivas da linguagem devem ser mapeadas para as instruções específicas da máquina, além de configurações específicas do funcionamento da máquina e dos testes necessários para garantir o funcionamento do conjunto, garantindo que a integração teve sucesso. Todo este processo por si só já compreende um novo trabalho de pesquisa, e por este motivo a integração da máquina com a linguagem *Forth* foi deixada de lado.

Para contornar esta situação, é implementado um *assembler* básico para que seja possível a programação básica de ambas as máquinas. Desta forma, os *bytes* da instrução são abstraídos em conjuntos de palavras (*macros*) definidas pelo programador.

Apesar da ausência dos itens citados anteriormente, as principais funcionalidades da CPU estão presentes em ambas as máquinas. Assim, é possível fazer uma comparação da execução dos algoritmos entre uma máquina MIPS e as duas máquinas desenvolvidas, tanto em relação a tamanho de programa, quanto a quantidade de acessos a RAM e ciclos de execução.

Vale ressaltar que nenhuma das máquinas descritas aqui foram desenvolvidas com o intuito de serem referencia para a implementação de uma máquina real. Estas foram desenvolvidas com a única finalidade de testar e comparar o comportamento e o desempenho de execução simples em relação à uma máquina de registradores, validando ou não as teorias de otimização propostas por neste trabalho.

5.2 Máquina de zero operandos

A máquina de zero operandos desenvolvida para este trabalho é inspirada nas máquinas *Forth* descritas na Capítulo 3, juntamente com as otimizações discutidas na Capítulo 4. De codinome *0op*, a máquina é dividida em seis blocos operacionais denominados ROM, *fetch*, ULA, pilha de dados, pilha de retorno e RAM. A organização destes blocos pode ser vista na Figura 14, onde cada uma das ligações entre eles representa um barramento TLM.

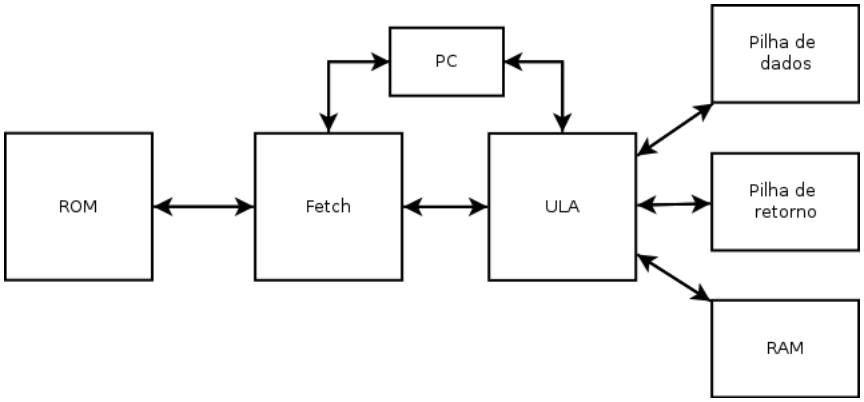

Figura 14: Diagrama de blocos da máquina de zero operandos.

O PC é implementado como um ponteiro para um valor inteiro que é dividido entre ambos os módulos de *fetch* e ULA. Um módulo principal nomeado *top* é utilizado para a instanciação e interconexão de todos os módulos, mostrados na Figura 14, e o PC, sendo este responsável por inicializar a execução da máquina virtual *0op*.

Os módulos *fetch* e ULA são os blocos responsáveis pela execução ativa da máquina, enquanto os demais atuam passivamente com o controle de suas memórias. Desta forma, o módulo *fetch* tem como função carregar da memória ROM a instrução referente ao PC atual e repassá-la ao médulo ULA. A ULA por sua vez, interpreta a instrução e realiza a operação requerida pela mesma, podendo ler ou escrever em qualquer um dos módulos de RAM, pilha de dados ou retorno.

Todos os módulos de memória são implementados de forma semelhante, contendo um único vetor de inteiros com 32 *bits*, o que faz com que o acesso em instrução à posições de memória RAM seja feito por índice de vetor e não por índice de *byte* como normalmente é feito em máquinas reais. Como a máquina não implementa nenhum tipo de operação em *byte* ou *bit*,

esta forma de indexação não afeta a leitura ou a escrita dos resultados em RAM.

O tamanho dos vetores de cada módulo de memória é pré-definido na compilação da máquina virtual, como pode ser visto no Código 2. Por padrão, ambas as máquinas são definidas com ambas as pilhas de dados e retorno com 8 posições, e ambas as memórias RAM e ROM são definidas com 128 posições.

Código 2: Instanciação dos tamanhos das memórias e pilhas.

```
1 #define RAM_DEPTH    128
2 #define ROM_DEPTH    128
3 #define STACK_SIZE   16
4 #define DS_SIZE      8
5 #define RS_SIZE      8
```

Quanto ao acesso à memória, cada módulo implementa seu próprio protocolo de acesso conforme o endereço oriundo do pacote TLM. O módulo de RAM e ROM permitem a leitura de qualquer posição de seus vetores, mas só a RAM permite a escrita. Já o módulo de pilha de retorno só permite a leitura e escrita ao topo de sua pilha, além das instruções de *pop* e *push*.

Os *buffers* de topo de pilha para dados e retorno são definidos como ponteiros inteiros. Em ambos os casos, são encontrados quatro ponteiros para o acesso de seu topo. Este total de *buffers*, visto na Figura 15, para a pilha de dados segue o que foi descrito na Capítulo 4, utilizando de um *buffer* a mais do que o comentado para obter maior flexibilidade.

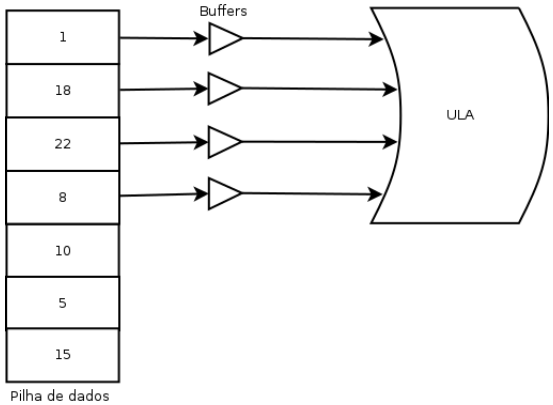

Figura 15: Ilustração do acesso ao topo da pilha de dados por meio de *buffers*.

Assim, a ULA não faz a requisição de leitura do topo das pilhas diretamente ao módulos

que as mantem. Ao invez disto, a cada modificação feita no final do ciclo de execução pela ULA na pilha, os *buffers* são sincronizados com a nova configuração de topo da pilha. Desta forma, em todo inicio do ciclo de execução os *bufers* estão coerentes em relação ao topo de sua respectiva pilha.

Esta implementação faz uso desta técnica também na pilha de retono. O principal motivo desta decisão é adicionar flexibilidade à programação da máquina, visto que em cenários sem uso extensivo de chamadas de função a pilha de retorno pode ser usada como receptáculo de variáveis locais pelo programador. Sendo assim, o usuário pode acessar facilmente suas variáveis locais sem a necessidade de desempilhamento da pilha de retorno caso decida usa-la para este propósito.

5.2.1 Conjunto de instruções

Seguindo o padrão visto nas máquinas *Forth*, são implementadas quatro tipos de instrução. Por padrão, os quatro primeiros *bits* da instrução começam divididos em um *bit* de imediato e caso a instrução não seja de imediato, três *bits* de controle. Os demais *bits* variam conforme o tipo de instrução, que são explicados em mais detalhes a seguir.

5.2.1.1 Imediato

O primeiro *bit* com valor 1 é interpretado como um carregamento de valor imediato, como é mostrado na Figura 16. Todos os 31 *bits* da instrução são carregados como um único valor no topo da pilha de dados.

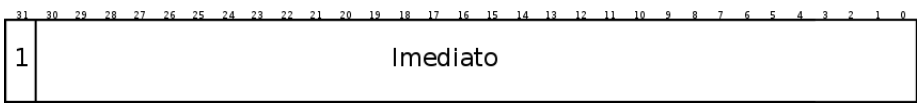

Figura 16: Formato de instrução para instanciação de imediatos.

5.2.1.2 Desvio

A instrução de desvio mostrada na Figura 17 é responsável por lidar tanto com saltos condicionais e incondicionais, quanto chamadas de função. Para isto, o campo *CTRL* da instrução

pode conter os valores ’001’, ’010’ e ’011’, indicando chamada de função, salto e salto condicional, respectivamente.

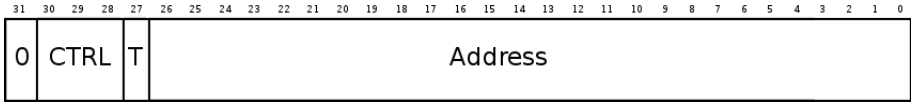

Figura 17: Formato de instrução para operações de desvio.

Ambas instruções de salto e chamada de função são executados de forma incondicional, sem a necessidade de nenhum valor oriundo da pilha de dados. Já no caso de um salto condicional, o topo da pilha de dados é carregado pela ULA e comparado em relação a zero, efetivando o salto caso haja igualdade.

O campo de único *bit* da instrução chamado de *T* é responsável por indicar a fonte de endereço a ser usada. Caso o valor do *bit* seja 1, o endereço a ser usado como base para o salto é lido do topo da pilha de dados, e no caso contrário o campo de endereço encontrado no *offset* da instrução é utilizado. Esta instrução é implementada desta forma para que seja possível o salto em endereços maiores que 27 *bits*.

5.2.1.3 Memória

Com a função principal de possibilitar a leitura e escrita da memória principal, a instrução de memória vista na Figura 18 apresenta dois tipos de controle possíveis. Caso o campo *CTRL* contenha o valor ’101’, uma leitura será efetuada, enquanto que o valor ’110’ indica uma escrita em memória.

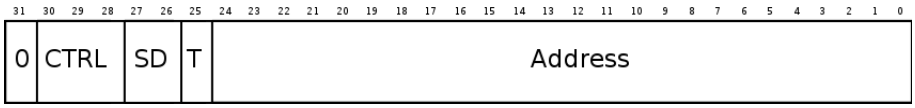

Figura 18: Formato de instrução para acesso à memória RAM.

Quanto ao campo *SD*, é especificado qual a origem do valor de escrita ou o destino da leitura efetuada em memória. Os valores encontrados neste campo são ’00’, ’01’, e ’10’, indicando respectivamente o Tos, NTos e PC. Com esta configuração a troca de contexto pode ser é otimizada devido a possibilidade de transferência direta entre PC e a memória.

Como a máquina não apresenta um mecanismo para troca de contexto, é discutível a troca do acesso do PC pelo campo *SD* e sua substituição pelos outros dois *buffers* da pilha de dados que não foram listados. Contudo, em uma implementação real o acesso direto ao PC em relação à memória pode ser mais vantajoso, visto que uma manipulação na pilha é o suficiente para trazer ao topo o valor contido no terceiro e quarto *buffer* da pilha.

Por fim, o campo *T* funciona da mesma forma que na instrução de desvio. A única diferença aqui é que o campo de endereço no *offset* da instrução tem 25 *bits* ao inves de 27 *bits*.

5.2.1.4 ULA

A instrução utilizada para acessar a ULA, vista na Figura 19, é a mais complexa do conjunto de instruções. Por padrão, o campo *CTRL* de valor '111' indica à máquina que a instrução corrente é endereçada a ULA, subdividindo o restante da instrução nos campos origem (*SRC*), destino (*DST*), modulação (*MOD*), comando (*CMD*), retorno (*R*) e um *offset* de 10 *bits*.

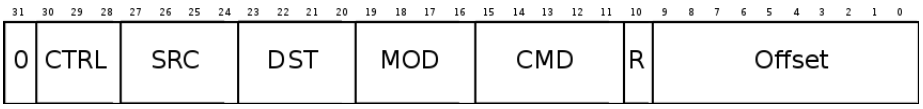

Figura 19: Formato de instrução para operações de ULA.

Assim como no conjunto de instruções da máquina *J1* (BOWMAN, 2010), é possível fazer modulações em pilha e endereçar qual o *buffer* que receberá o resultado, além do próprio *opcode* de operação da ULA. Contudo, este formato de instrução se diferencia do usado na *J1* na acesso à memória, visto que a máquina *0op* apresenta uma instrução própria para leitura e escrita em memória RAM.

Ambos os campos origem e destino utilizam da mesma codificação, vista na Tabela 11. A campo origem é utilizado principalmente em operações unárias, ou seja, que dependa de apenas um valor vindo da CPU, como uma adição com o *offset* da instrução ou uma movimentação de valores. Para casos de instruções lógico/aritméticas ou qualquer outra que não seja unária, o campo deve endereçar somente o topo da pilha de dados ou retorno. Caso contrário a instrução será ignorada pela máquina.

Tabela 11: Operações encontradas nos campos *SRC* e *DST*.

Valor	Nome	Descrição
0000	*None*	Não utiliza
0001	Tos	Topo da pilha de dados
0010	Ntos	Próximo ao topo da pilha de dados
0011	*3os*	Terceiro ao topo da pilha de dados
0100	*4os*	Quarto ao topo da pilha de dados
0101	Tors	Topo da pilha de retorno
0110	Ntors	Próximo ao topo da pilha de retorno
0111	*3ors*	Terceiro ao topo da pilha de retorno
1000	*4ors*	Quarto ao topo da pilha de retorno
1001	PC	Contador de programa
1010	Zero	Valor zero absoluto
1011	*DS Size*	Tamanho atual da pilha de dados
1100	*RS Size*	Tamanho atual da pilha de retorno

Quanto ao campo de destino, qualquer uma das opções apresentadas na Tabela 11 que enderece alguma posição de uma das pilhas ou o PC pode ser usada. Ao final da execução da instrução pela ULA, o resultado obtido será alocado no local especificado pelo campo destino.

Contudo, antes da escrita do resultado obtido o campo de modulação é levado em consideração. Analisado em detalhes na Tabela 12, o campo de modulação é responsável por alocar ou desalocar as posições ao topo de ambas as pilhas antes que a escrita por parte da ULA seja realizada. Assim, uma maior flexibilidade é atingida em relação à escrita nas pilhas pois três resultados diferentes podem ser obtidos com a mesma operação logico/aritmética.

Tabela 12: Operações encontradas no campo *MOD*.

Valor	Ação
0000	Nada
0001	*Push* na pilha de dados
0010	*Pop* na pilha de dados
0011	*Push* na pilha de retorno
0100	*Pop* na pilha de retorno
0101	*Push* na pilha de dados e retorno
0110	*Pop* na pilha de dados e retorno
0111	*Push* na pilha de dados e *pop* na pilha de retorno
1000	*Pop* na pilha de dados e *push* na pilha de dados

Em uma máquina de pilha comum vista na Capítulo 2, é necessária a retirada dos dois

valores mais ao topo na pilha *A* de dados para sua leitura por parte da CPU. Após a computação, o resultado é escrito de volta na pilha de dados, comportando-se sempre como uma operação de *pop* seguida de uma sobrescrita do topo da pilha.

Contudo, com a presença de um campo de modulação de pilha, o comportamento de uma operação aritmética pode ter diferentes comportamentos sobre a pilha de dados. Como por exemplo a Figura 20, onde uma instrução de adição é feita em paralelo com um *pop* (caso *B*), nada (caso *C*) e *push* (caso *D*). Como o resultado é escrito sobre o topo da pilha, este pode vir a sobrescrever um dos valores usados para a operação, ou até mesmo ser alocado sem nenhuma sobrescrita.

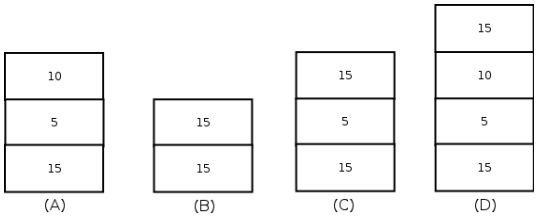

Figura 20: Exemplo de diferentes modulações de pilha.

No caso de um zero ser especificado, o primeiro valor a entrar na ULA será sempre o topo da pilha de dados, seguido de um valor zero absoluto. Desta forma, toda a operação lógica ou comparativa é tomada em relação ao topo da pilha de dados, sem a possibilidade de comparação com as demais origens especificadas pelo campo de origem da instrução.

Já na implementação da máquina *0op*, o topo da pilha já está carregado nos *buffers* da pilha de dados. Caso o comportamento desejado pelo programador seja o descrito no paragrafo anterior *B*, a operação de adição pode ser feita com uma modulação de *pop* de pilha de dados, visto que o *pop* será efetuado antes da sobrescrita do topo da pilha. Caso o programador deseje que o segundo valor *C* ou todos os valores atuais *D* no topo da pilha sejam mantidos, a adição pode ser realizada sem modulação na pilha de dados, ou com uma modulação de *push*, respectivamente.

Na Tabela 13 são vistos em detalhes todas as operações possíveis para a computação da ULA. Nestas, todas as operações aritméticas dependem única e exclusivamente do topo da especificada no campo origem. Já as operações contendo operações lógicas ou de comparação, tanto o topo de uma das pilhas pode ser especificado quanto o operador zero pode ser especificado no campo de origem.

Tabela 13: Operações encontradas no campo *CMD*.

Valor	Operação	Descrição
00000	*Nothing*	A ULA não faz nada
00001	*Add*	Soma os dois valores de entrada
00010	*Sub*	Subtrai os dois valores de entrada
00011	*Mult*	Multiplica os dois valores de entrada
00100	*Div*	Divide os dois valores de entrada, gerando um resultado inteiro
00101	*And*	Operação lógica *and* entre os dois valores de entrada
00110	*Or*	Operação lógica *or* entre os dois valores de entrada
00111	*Xor*	Operação lógica *xor* entre os dois valores de entrada
01000	*Xnor*	Operação lógica *xnor* entre os dois valores de entrada
01001	*Equal*	Compara a igualdade das entradas, resultando em um valor lógico
01010	*Different*	Compara a desigualdade das entradas, resultando em um valor lógico
01011	*More or equal*	Testa se o valor mais ao topo da pilha é maior ou igual ao segundo ao topo, resultando em um valor lógico
01100	*Less or equal*	Testa se o valor mais ao topo da pilha é menor ou igual ao segundo ao topo, resultando em um valor lógico
01101	*More*	Testa se o valor mais ao topo da pilha é maior que o segundo ao topo, resultando em um valor lógico
01110	*Less*	Testa se o valor mais ao topo da pilha é menor que o segundo ao topo, resultando em um valor lógico
01111	*Not*	Inverte todos os *bits* da entrada
10000	*Shift left*	Move os *bits* de entrada à esquerda
10001	*Shift right*	Move os *bits* de entrada à direita
10010	*Pluss offset*	Soma o valor de entrada ao offset
10011	*Less offset*	Subtrai o valor de entrada ao offset
10100	*Mult offset*	Multiplica o valor de entrada ao offset
10101	*Move*	Move o valor de entrada ao destino especificado

Em instruções de deslocamento e *offset*, todos as origens especificadas pelo campo de origem da instrução podem ser usados. Na execução destes tipos de instrução, a ULA terá uma única entrada de dados proveniente da origem especificada. Este valor terá sua operação efetuada com base nos dez *bits* de *offset* da instrução, guardando o o resultado no destino especificado.

A instrução *move* tem a principal serventia de transferir dados entre entre diferentes localidades da CPU. Ao executar uma instrução deste tipo, a ULA unicamente acessa a origem e escreve o valor lido no destino especificado, sem nenhum tipo de computação extra. Esta instrução se torna útil principalmente para o acesso de recursos normalmente não acessíveis as pilhas, como o conteúdo da pilha adjacente, o tamanho de ambas as pilhas e o PC atual, no caso da pilha de dados.

Por fim, o campo de retorno tem a única finalidade de permitir o retorno gratuito de chamada de função. Desta forma, qualquer instrução de ULA pode em paralelo à sua execução, realizar uma troca entre o topo da pilha de retorno com o registrador de PC. Caso não seja possível sincronizar uma código uma operação lógico/aritmética com um retorno de função, o programador deve usar uma instrução com os campos origem, destino e comando zerados, deixando somente o campo de retorno habilitado.

5.3 Máquina de dois operandos

De codinome *2op*, a máquina de pilha com dois operandos implementada para este trabalho teve o funcionamento de sua pilha de dados inspirado no *belt* encontrado na CPU *Mill*, vista na Capítulo 3.4. Seguindo as análises da Capítulo 4, a máquina é implementada com uma pilha de dados circular e sem desempilhamento, bem como uma pilha tradicional para endereços de retorno.

Como os operandos são precisam ser carregados da pilha de dados antes que a ULA possa acessa-los, um módulo responsável por controlar o acesso à pilha de dados é adicionado à CPU. Como o visto na Figura 21, o controlador recebe a instrução do módulo de *fetch* e se responsabiliza por buscar os operandos necessários na pilha de dados.

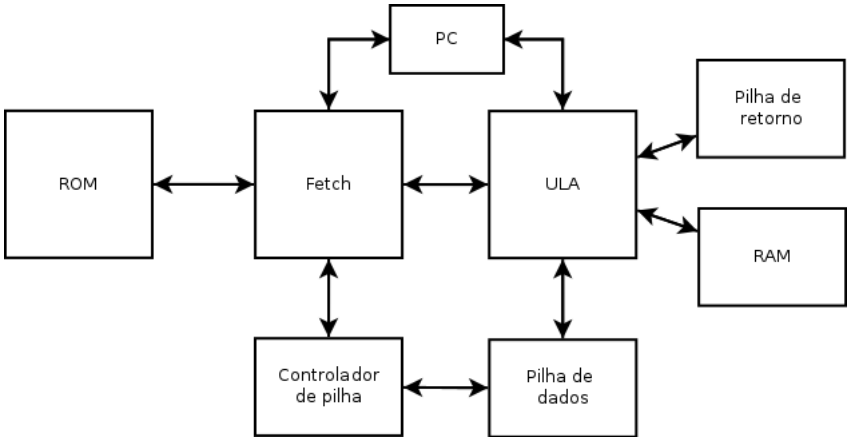

Figura 21: Diagrama de blocos da máquina de dois operandos.

Ao carregar os operandos, o módulo responsável por manter a pilha de dados se responsabiliza por enviar à entrada da ULA os operandos requeridos pelo controlador, na sequencia especificada na instrução. Assim, os operandos são passados diretamente da pilha para a ULA,

sem precisar de uma conexão direta entre o controlador da pilha e ULA.

Em relação a escrita dos resultados calculados, o módulo da ULA sempre irá efetuar um *push* seguido de uma escrita no topo da pilha de dados. Apesar desta abordagem não apresentar a mesma flexibilidade dos múltiplos destinos dentro da CPU encontrado na máquina *0op*, esta forma de implementação sendo necessária para a escrita rápida na pilha de dados, visto que não existem *buffers* de entrada na ULA.

O módulo de ULA apresenta dois registradores como interface com o módulo de pilha de dados, sendo estes registradores utilizados como fonte para os valores de entrada pela própria ULA. Como o visto no trecho de Código 3 da controladora de pilha, a instrução é testada pela controladora e um requerimento (*make_Transaction*) de leitura é enviado à pilha de dados. Junto ao requerimento, um endereço informa à pilha qual registrador de entrada da ULA receberá o valor lido do registrador temporal.

Código 3: Trecho de código usado para tratar a instrução recebida pelo módulo ULA.

```
1  if ((int) instruction[28] == 1) {
2
3          if ((int) instruction.range(30, 29) == 3) {                    //ALU - '11'
4
5                  if ((int) instruction[11] == 0) {
6
7                          make_Transaction(instruction.range(7, 4), 0);   //OP1
8
9                  }
10         } else if ((int) instruction.range(30, 29) == 1) {             //branch - '01'
11
12         } else if ((int) instruction.range(30, 29) == 2) {             //store - '10'
13                 // Se usa um operando como endereco
14                 if (instruction[27] == 1) {
15
16                         make_Transaction(instruction.range(7, 4), 0);   //OP1
17                         make_Transaction(instruction.range(3, 0), 1);   //OP2
18
19                 } else {
20
21                         make_Transaction(instruction.range(7, 4), 0);   //OP1
22
23                 }
24         }
25
26 }
```

Os demais aspectos referentes aos módulos da máquina são implementados de forma idêntica a máquina *0op*. Com a diferença de que o módulo de *fetch* não se comunica apenas com a ULA mas também com o controlador da pilha de dados. Além disto, as implementações de ambas as pilhas não apresentam nenhum *buffer*, pois a pilha de dados é lida pela controladora enquanto que o módulo da pilha de retorno só aceita leituras e escritas a seu topo.

5.3.1 Conjunto de instruções

Assim como na máquina *0op*, é utilizada mesma quantidade de tipos de instrução. Apesar disto, o funcionamento da maioria das instruções muda devido a presença de dois operandos. O formato padrão para todas as instruções começa com um *bit* para o carregamento de imediato, seguido de dois *bits* de controle que especificam o tipo de instrução.

Outro padrão encontrado nas instruções da máquina *2op* é o de que o *bit* 28 da instrução quando ativo, informa ao controlador da pilha irá carregar apenas um um operando, como pode ser visto no trecho de código da Capítulo 5.3. Apesar disto, alguns casos encontrados em instruções que serão vistas mais a frente são considerados como exceções carregando dois ou nenhum operando com o *bit* 28 habilitado.

5.3.1.1 Imediato

Como pode ser visto da Figura 22, a instrução de imediato é idêntica a encontrada na máquina *0op*. Da mesma forma, ao executar esta instrução o valor inteiro contido nos 31 *bits* mais ao fim da instrução são escritos na pilha de dados após um *push* da pilha.

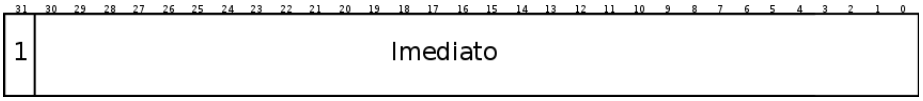

Figura 22: Formato de instrução para instanciação de imediatos.

5.3.1.2 Desvio

Mostrado em detalhes na Figura 23, a instrução de desvio detém os campos de condição (*N*), endereço (*address*), *opcode* (*OP*), chamada (*C*) e dois campos de endereçamento de operandos. Por padrão, a sequencia de *bits* '01' no campo de tipo da instrução.

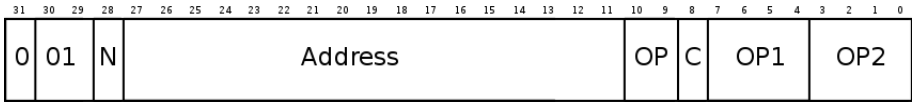

Figura 23: Formato de instrução para operações de desvio.

Ambos os operandos detém quatro *bits* pois o tamanho máximo de pilha de dados encontrada nesta implementação é de 16 posições. Neste e nos demais tipos de instrução que façam uso dos operandos, os 8 últimos *bits* da instrução sempre são reservados para os dois campos de operando. Além disto, qualquer operação lógica/aritmética é sempre efetuada utilizando os operandos em ordem crescente, como por exemplo, uma divisão sempre efetuará *OP1* dividido por *OP2*.

Os campos de chamada e condição atuam de forma conjunta para a especificação do comportamento da instrução na CPU. Caso o *bit* de condição esteja ativo, ambos os operandos são carregados da pilha de dados para comparação e, caso contrário, nenhum valor será carregado da pilha de dados. Já o *bit* de chamada quando ativo informa à CPU que além do salto no final da operação, o PC atual deve também ser guardado na pilha de retorno.

Desta forma, consegue-se efetuar tanto saltos quanto chamadas de função de forma condicional ou incondicional. Isto é possível graças ao acesso dos valores de dois operandos, que não é possível na implementação com zero operandos onde a única forma de comparação utilizada é o topo da pilha em relação à zero.

Por fim, o campo de *opcode* é responsável por sinalizar qual a operação lógica a ser utilizada ao executar saltos ou chamadas condicionais. Visto em detalhes na Figura 14, o campo implementa quatro operações lógicas para a comparação dos operandos carregados, efetivando a operação caso o resultado da comparação lógica seja verdadeira.

Tabela 14: Relação de operações do campo *OP* da instrução de desvio.

OP	**Operação**	**Efeito**
00	Igual	*OP1* == *OP2*
01	Diferente	*OP1* != *OP2*
10	Maior	*OP1* >*OP2*
11	Maior igual	*OP1* >= *OP2*

5.3.1.3 Memória

Por padrão o valor '10' nos *bits* do campo de tipo de instrução, visto na Figura 24, indica uma operação de leitura ou escrita em memória. Nesta instrução são encontrados os campos escrita (*S*), origem (*O*), endereço (*Address*) e operandos (*OP1* e *OP2*).

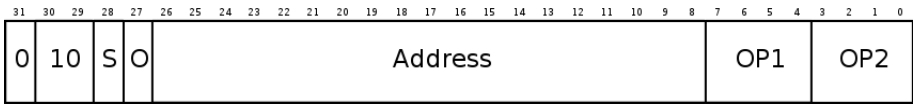

Figura 24: Formato de instrução para acesso à memória RAM.

Cada operando contém quatro *bits*, podendo endereçar assim um total máximo de 16 registradores temporais da pilha de dados. Por padrão, um operando zero aponta para o topo da pilha, podendo assumir o tamanho máximo da pilha de dados menos um, sendo este a base atual da pilha de dados. Desta forma quanto maior o operando, mais profundo na pilha o registrador temporal acessado se encontra.

Caso o *bit* do campo de escrita esteja ativo uma escrita será realizada e, caso contrário, uma leitura de memória será efetuada. Em ambos os casos, o campo de origem ativo faz com que o valor contido no registrador *OP2* seja utilizado como endereço e caso contrário o valor contido no campo de endereço é utilizado para a operação.

Assim como na máquina *0op*, a instrução de memória permite o uso do campo de endereço ou um endereço alocado em pilha como base para a manipulação da memória. Desta forma , é possível usar endereços estáticos de até 19 *bits* sem a necessidade de aloca-los em pilha, trazendo mais flexibilidade ao conjunto.

Ao executar esta instrução, o registrador temporal que irá receber os dados da memória

ou que servirá de fonte de dados para a escrita em memória deve ser especificado sempre no primeiro operando. Ao receber uma instrução deste tipo, o controlador da pilha detecta se a operação é uma leitura ou escrita. Caso a instrução seja uma escrita, um comando de leitura do registrador temporal é emitido ao módulo da pilha de dados, que por sua vez providencia este valor à entrada da ULA. Tal comportamento pede ser visto em detalhes no fluxograma da Figura 25.

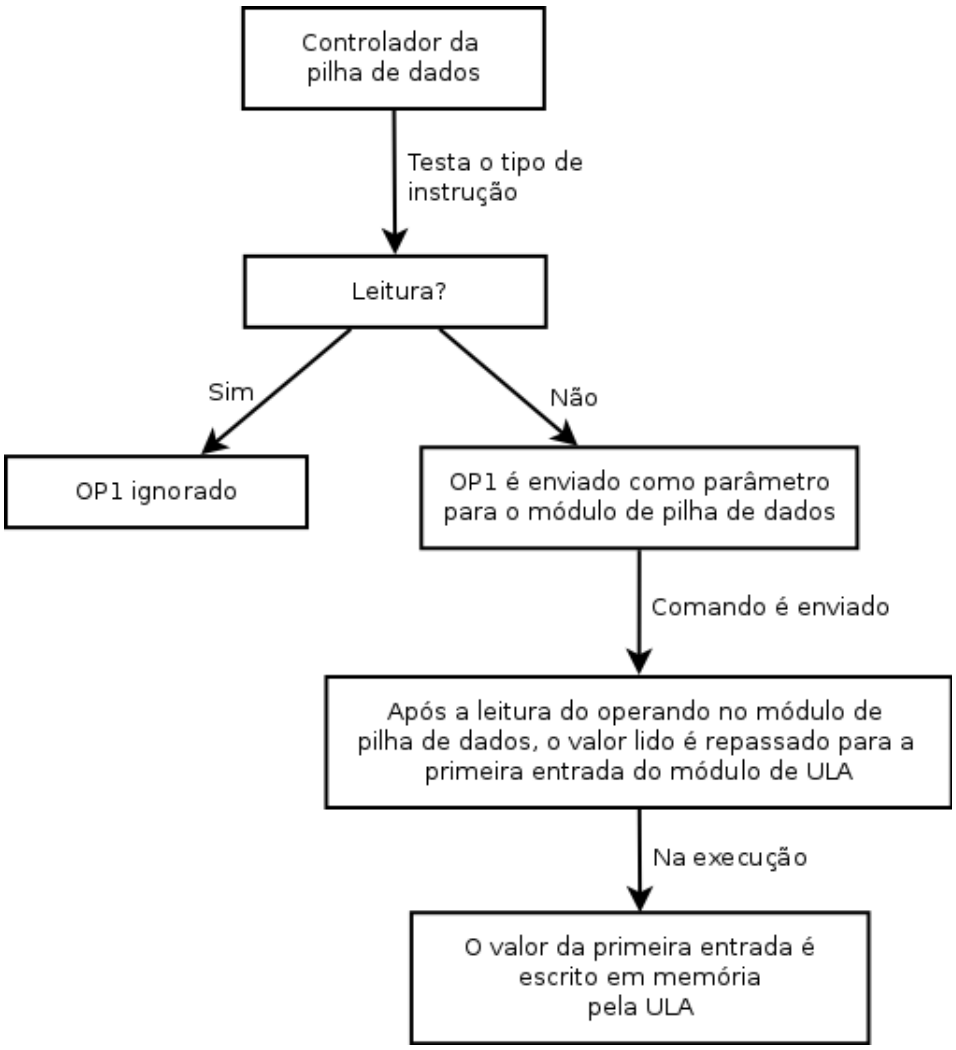

Figura 25: Fluxograma da tomada de decisões no módulo controlador de pilha referente ao primeiro operando.

Vale ressaltar que o controlador de pilha não tem acesso direto à ULA. Com isto, seus comandos são enviados diretamente para o módulo responsável por manter o vetor onde está contida a pilha de dados, que por sua vez executa a ação requerida pelo comando e envia o resultado à ULA. No caso do comando ser um aviso de término de busca, o módulo da pilha passa o comando adiante, sem executar nenhuma ação.

Ainda no controlador, o campo de origem da instrução é analisado para efetuar, ou não, o carregamento do valor a ser usado como endereço. Assim como no fluxograma da Figura 26, caso o campo origem seja, o registrador temporal apontado pelo segundo operando é carregado na segunda entrada do módulo de ULA. Caso contrário, a segunda entrada da ULA é ignorada e a própria ULA reconhece que o campo de origem é zero, usando como consequência disto o campo de endereço como base para a operação de escrita ou leitura.

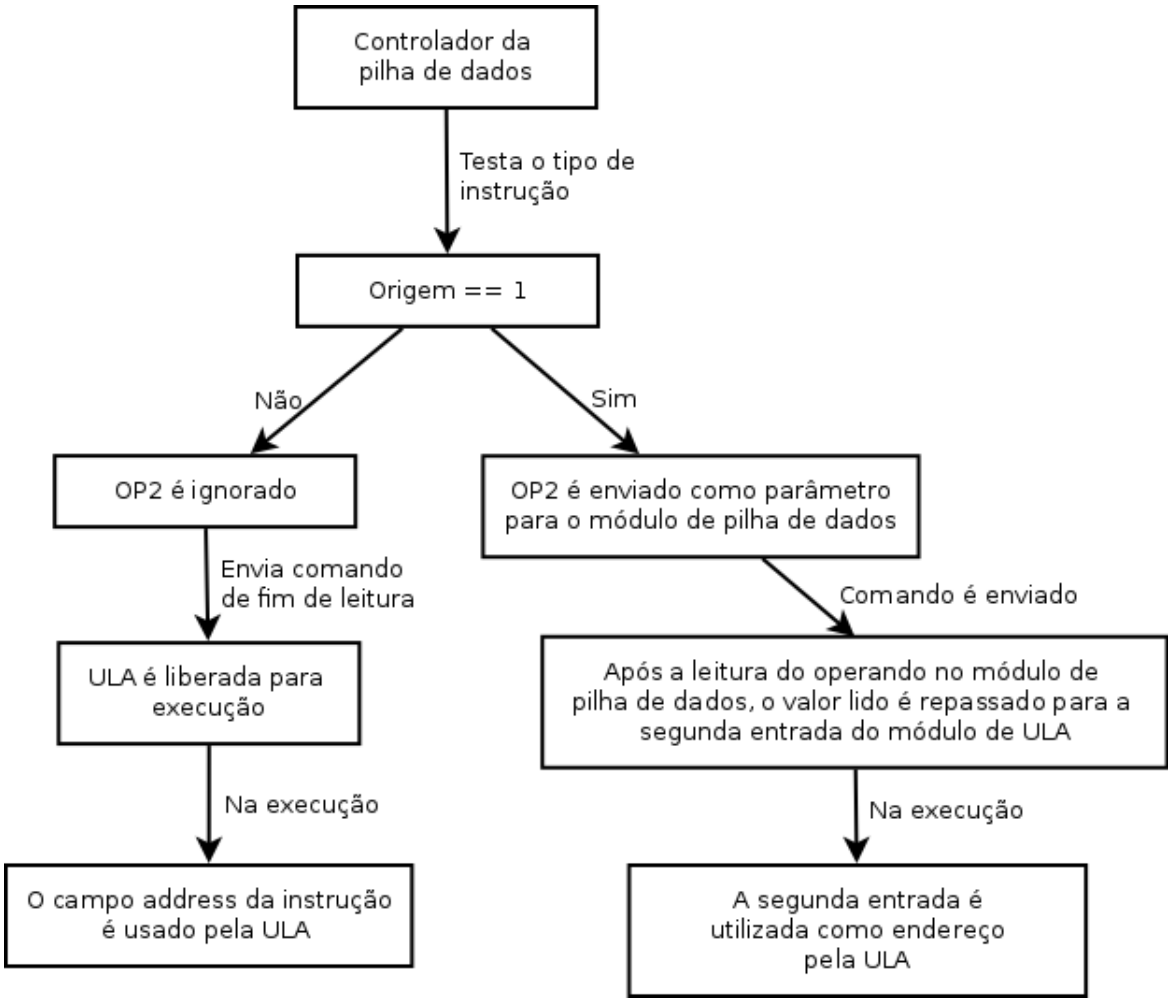

Figura 26: Fluxograma da tomada de decisões no módulo controlador de pilha referente ao segundo operando.

Ambos os testes citados para o primeiro e segundo operando são feitos paralelo. Desta forma, independentemente da composição da instrução, o valor a ser escrito estará sempre na primeira entrada da ULA, enquanto que o valor a ser usado como endereço sempre estará na segunda entrada da ULA.

5.3.1.4 ULA

A instrução de ULA é identificada pelos *bits* '11' no campo de tipo de instrução, observado na Figura 27. Em seus campos, são encontrados os campos de carregamento único (*L*), *offset* ou base para operação de deslocamento (*Shamt*), movimento de pilha (*STK*), operação de ULA (*OPT*) e operandos (*OP1* e *OP2*);

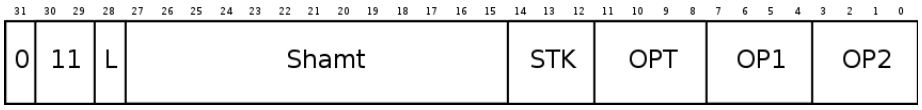

Figura 27: Formato de instrução para operações de ULA.

O campo *L* por se encontrar no *bit* 28 da instrução, é responsável por indicar o carregamento de um ou dois registradores temporais da pilha de dados. Com isto, este campo atua em conjunto do campo *OPT* para designar uma operação especifica para a execução da ULA. Vistos na Tabela 15, um mesmo *opcode* para o campo *OPT* pode ter um comportamento diferente conforme *L*.

Tabela 15: Composição de operações encontradas no campo *OPT*.

L	OPT	Função	Descrição
1	0000	*Nothing*	Não faz nada
1	0001	*Not*	Inverte todos os *bits* de *OP1*
1	0010	*Shift Left*	Move os *bits* de *OP1* à esquerda
1	0011	*Shift Right*	Move os *bits* de *OP1* à direita
0	0000	*ADD*	Soma os dois valores de *OP1* e *OP2*
0	0001	*SUB*	Subtrai os dois valores de *OP1* e *OP2*
0	0010	*AND*	Operação lógica *and* entre *OP1* e *OP2*
0	0011	*OR*	Operação lógica *or* entre *OP1* e *OP2*
0	0100	*XOR*	Operação lógica *xor* entre *OP1* e *OP2*
0	0101	*XNOR*	Operação lógica *xnor* entre *OP1* e *OP2*
1	0100	*Add Offset*	Soma o valor de *OP1* ao *offset*
1	0101	*Sub Offset*	Subtrai o valor de *OP1* ao *offset*
0	1101	*	Multiplica os dois valores de *OP1* e *OP2*
0	0110	/	Divide os dois valores de *OP1* e *OP2*, gerando um resultado inteiro
0	0111	>	Testa se o valor de *OP1* é maior que o *OP2*, resultando em um valor lógico
0	1000	=	Compara a igualdade entre *OP1* e *OP2*, resultando em um valor lógico
0	1001	!=	Compara a desigualdade entre *OP1* e *OP2*, resultando em um valor lógico
0	1010	>=	Testa se o valor de *OP1* é maior ou igual ao *OP2*, resultando em um valor lógico
0	1011	<=	Testa se o valor de *OP1* é menor ou igual ao *OP2*, resultando em um valor lógico
0	1100	<	Testa se o valor de *OP1* é menor que o *OP2*, resultando em um valor lógico
1	1101	*RS Size*	Retorna o tamanho da pilha de retorno

Seguindo o que foi comentado a respeito de exceções ao comportamento de carregamento único do *bit* 28 citado na Capítulo 5.3, o uso da função *RS size* não requer que nenhum operando seja carregado da pilha visto que somente o índice da pilha de retorno deve ser lido. No mais, instruções que envolvam deslocamento, operações sobre *offset* ou operações lógicas fazem o uso normal do *bit* 28, carregando somente o primeiro operando da instrução.

O campo de *Shampt* é utilizado como segundo operando tanto por instruções de deslocamento quanto por instruções de aritmética com *offset*. Apesar do exagero dos 13 *bits* no campo de *Shampt* para instruções de deslocamento, esta abordagem de compartilhamento do campo não requer que duas instruções de ULA implementando um *offset* e um campo de deslocamento com tamanho apropriado sejam implementadas.

Em paralelo à execução da ULA, há a possibilidade de se fazer trocas de dados entre ambas as pilhas e o PC via o campo *STK*. Visto em detalhes na Tabela 16, o procedimento utiliza sempre o registrador de PC, o topo da pilha de retorno e a primeiro valor carregado na entrada da ULA, além de realizar a escrita do valor após a escrita efetuada pela ULA.

Tabela 16: Composição das operações do campo *STK*.

STK	**Função**	**Descrição**
000	Nada	Não faz nada
100	PC $\rightarrow RS$	Transfere o PC atual para o topo da pilha de retorno
101	RS $\rightarrow PC$	Transfere o topo da pilha de retorno para o PC
001	RS $\rightarrow DS$	Transfere o topo da pilha de retorno para a pilha de dados
010	DS $\rightarrow RS$	Transfere o *OP1* carregado na entrada da ULA para a pilha de retorno

Esta funcionalidade permite que o retorno de função seja realizado em paralelo com uma operação de ULA. Além da possibilidade de transferir para ambas as pilhas, sem sobrescrever o resultado calculado pela ULA mesmo que estes dois processos concorrentes escrevam sobre a pilha de dados.

A Figura 28 mostra um exemplo de codificação de uma instrução de adição com e sem uso de *offset*. Na instrução *a*, o *bit* 28 indica que ambos os operandos serão carregados da pilha de dados, que neste caso são os registradores temporais 1 e 2. Já na instrução *b* o *bit* 28 indica o carregamento de apenas um operando, fazendo com que o valor contido no campo *Shamt* seja utilizado como segundo operando. Nos demais campos, o *STK* com valor zero indica que nenhuma troca de valores entre pilhas será feita, e o campo *OPT* indica, juntamente com o *bit* 28, uma adição.

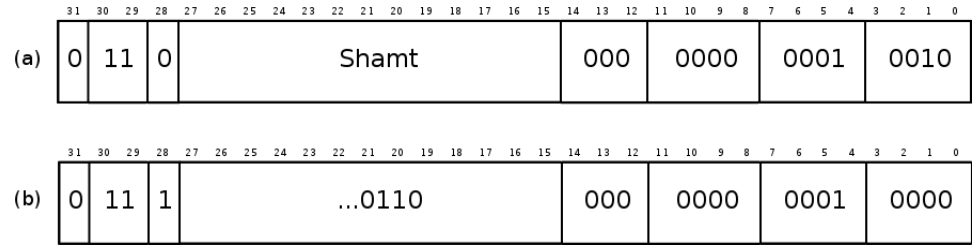

Figura 28: Exemplo de instrução para operações de adição de operados e adição com *offset*.

5.4 Súmula

Com a análise de toda a implementação realizada em ambas as máquinas deste Trabalho de Conclusão, é possível realizar uma comparação das principais características destas máquinas em relação à máquina de registradores usada como base de comparação neste trabalho. Na Tabela 17 são apresentados aspectos gerais das máquinas *0op*, *2op* e MIPS, permitindo um rápido *overview* de todas as implementações.

Tabela 17: Comparação das características das máquinas desenvolvidas e o MIPS.

Propriedade	***0op***	***2op***	***MIPS***
Operandos	0	2	3
Tamanho de palavra	*32 bits*	*32 bits*	*32 bits*
Pipeline	Monociclo	Monociclo	Monociclo
Tipos de instrução	4	4	3
Registradores	16*	16*	32
Banco de registradores	2	2	1
Registradores endereçaveis	4**	9	32
Chamada condicional	Não	Sim	Não
Unidade de controle	*Hardwired*	*Hardwired*	*Microcoded*
Arquitetura de memória	*Harvard*	*Harvard*	*Von Neumann*

* - Cada pilha pode usar até 16 registradores, alcançando um total de 32.

** - Quatro *buffers* ao topo da pilha, porém a pilha não é explicitamente endereçável.

6 VALIDAÇÃO E CASOS DE TESTE

Neste capítulo, são analisados os métodos e ferramentas utilizados para testar e avaliar o funcionamento das máquinas descritas no capítulo anterior. O capítulo apresenta a forma de uso das máquinas *0op* e *2op*, bem como os testes efetuados.

6.1 *Assembler*

A partir das máquinas desenvolvidas, implementou-se um programa *assembler* para que seja possível definir mnemônicos para as instruções desejadas. Desta forma, a programação não é feita sobre os *bits* da instrução e sim sobre uma instrução de texto pré-definido pelo programador.

Foram desenvolvidos dois *assemblers*, um para cada máquina, sendo ambos implementados em *C++*. Para ambas as máquinas, a entrada é um arquivo contendo o código em forma de texto (mnemônicos), semelhante ao código *assembly* para MIPS. Após a interpretação do código de entrada, o *assembler* gera um arquivo contendo as instruções decodificadas em binário, sendo este utilizado como entrada de execução da máquina de pilha.

Ao analisar os códigos de entrada, são usadas todas as instruções definidas no *assembler* para tradução da palavra binária utilizada. Caso a palavra utilizada não esteja especificada, o *assembler* gera um erro. O *assembler* considera cada linha do arquivo de texto como sendo uma instrução diferente, separando o mnemônico e os operandos com o caractere de espaço.

O endereço de cada instrução é considerado pelo seu índice de linha, ou seja, a primeira linha do arquivo irá gerar a instrução de endereço zero ROM da máquina. Desta forma, saltos e chamadas em código referenciam o índice da primeira instrução do bloco de código chamado, e

não uma palavra abstrata para o trecho de código como é usado normalmente em um *assembler* MIPS.

Ainda, existem dois caracteres especias que são interpretados pelo *assembler* independentemente das instruções implementadas. O primeiro deles é o conjunto de caracteres '##', que são utilizados como especificador de comentários, e o caractere ';' que é utilizado para retornos de função. Vale ressaltar que os caracteres de comentários devem ser precedidos de espaço para não serem interpretados como operando ou instrução, além de que o caractere de retorno deve ser usado única e exclusivamente com instruções que utilizem a ULA, sendo necessário implementar uma instrução especifica de retorno para o caso contrário.

Foram implementados dois conjuntos de instruções, sendo um para cada máquina. Para a máquina *0op*, foram desenvolvidos as instruções descritas na Tabela 18 e para a máquina *2op*, as instruções descritas na Tabela 19. Sendo que para os dois conjuntos de instruções, a variável *n* é interpretada como um valor de *offset* ou endereço, dependendo do tipo de instrução que o usar.

Tabela 18: Operações básicas implementadas para o *assembler* da máquina *0op*.

Mnemônico	**Semântica**	**Descrição**
add	*add* ou *add n*	Soma os dois valores ao topo ou o topo da pilha de dados e o enésimo valor
sub	*sub* ou *sub n*	Subtrai os dois valores ao topo ou o topo da pilha de dados e o enésimo valor
mlt	*mlt* ou *mlt n*	Multiplica os dois valores ao topo ou o topo da pilha de dados e o enésimo valor
div	*div* ou *div n*	Divide os dois valores ao topo ou o topo da pilha de dados e o enésimo valor
not	*not*	Todos os *bits* do topo da pilha são negados
and	*and*	Tos *and* NTos
or	*or*	Tos *or* NTos
xor	*xor*	Tos *xor* NTos
xnor	*xnor*	Tos *xnor* NTos
equal ou ==	*equal* ou ==	Retorna o valor booleano de Tos igual a NTos
dif ou *!=*	*dif* ou *!=*	Retorna o valor booleano de Tos diferente de NTos
me ou >=	*me* ou >=	Retorna o valor booleano de Tos maior ou igual a NTos
le ou <=	*l*e ou <=	Retorna o valor booleano de Tos menor ou igual a NTos
more ou >	*more* ou >	Retorna o valor booleano de Tos naior que NTos
less ou <	*less* ou <	Retorna o valor booleano de Tos menor que NTos
sr ou >>	*sr n* ou >> *n*	Efetua um deslocamento a direita *n* posições
sl ou <<	*sl n* ou << *n*	Efetua um deslocamento a esquerda *n* posições
load	*load* ou *load n*	Lê do endereço do topo ou da enésima posição da pilha
store	*store* ou *store n*	Escreve o topo da pilha no endereço Ntos ou em *n*
call	*call* ou *call n*	Chama o endereço em Ntos ou na enésima posição
branch	*branch* ou *branch n*	Salta para o endereço *n* ou Ntos
branch0	*branch0* ou *branch0 n*	Caso o topo da pilha seja 0, salta para o endereço *n* ou Ntos
pop	*pop*	Remove o topo da pilha
nop	*nop*	Encerra a execução
dup	*dup*	Duplica o topo da pilha
over	*over*	Traz ao topo uma cópia da segunda posição da pilha
pick	*pick n*	Traz ao topo uma cópia da enésima posição da pilha
pick'	*pick' n*	Escreve o topo da pilha na enésima posição da pilha
return	*return*	Retorno de função

Tabela 19: Operações básicas implementadas para o *assembler* da máquina *2op*.

Mnemônico	***Semântica***	***Descrição***
add	*add op1 op2*	*op1 + op2*
sub	*sub op1 op2*	*op1 - op2*
mlt	*mlt op1 op2*	*op1 * op2*
div	*div op1 op2*	*op1 / op2*
not	*not op1*	Todos os *bits* de *op1* são negados
and	*and op1 op2*	*op1 and op2*
or	*or op1 op2*	*op1 or op2*
xor	*xor op1 op2*	*op1 xor op2*
xnor	*xnor op1 op2*	*op1 xnor op2*
equal ou ==	*equal op1 op2 ou == op1 op2*	*op1 == op2*
dif ou *!=*	*dif op1 op2* ou *!= op1 op2*	*op1 != op2*
me ou >=	*me op1 op2* ou >= *op1 op2*	*op1 >= op2*
le ou <=	*le op1 op2* ou <= *op1 op2*	*op1 <= op2*
more ou >	*more op1 op2* ou >*op1 op2*	*op1 >op2*
less ou <	*less op1 op2* ou <*op1 op2*	*op1 <op2*
sr ou >>	*sr op1 op2* ou »*op1 op2*	Efetua um deslocamento a direita *op2* posições em *op1*
sl ou <<	*sl op1 op2* ou « *op1 op2*	Efetua um deslocamento a esquerda *op2* posições em *op1*
load	*load op1*	Lê do endereço *op1*
store	*store op1 op2*	Escreve o *op1* no endereço *op2*
call	*call n*	Chama o endereço *n*
callm	*callm op1 op2 n*	Se *op1 >op2*, chama para o endereço *n*
callme	*callme op1 op2 n*	Se *op1 >= op2*, chama para o endereço *n*
calldif	*calldif op1 op2 n*	Se *op1 != op2*, chama para o endereço *n*
calleq	*calleq op1 op2 n*	Se *op1 == op2*, chama para o endereço *n*
branch	*branch n*	Salta para o endereço *n*
bm	*bm op1 op2 n*	Se *op1 >op2*, salta para o endereço *n*
bme	*bme op1 op2 n*	Se *op1 >= op2*, salta para o endereço *n*
beq	*beq op1 op2 n*	Se *op1 != op2*, salta para o endereço *n*
bdif	*bdif op1 op2 n*	Se *op1 == op2*, salta para o endereço *n*
++	*++ op1 n*	*op1 + n*, sendo *n* um *offset*
- -	*- - op1 n*	*op1 - n*, sendo *n* um *offset*
stro	*stro op1 n*	Escreve na posição *n* da memória o valor de *op1*

6.2 Programação

Para a comparação das máquinas desenvolvidas neste trabalho e a máquina MIPS, foram escolhidos os algoritmos de fatorial, *fibonacci*, *bubble sort* e *insertion sort*. Para que a comparação seja justa, as implementações seguem o mesmo algoritmo base, sem nenhum tipo de alteração no comportamento que privilegie uma determinada máquina.

Apesar de todas as máquinas implementarem o mesmo algoritmo, seu comportamento não é o mesmo. Tanto a forma como as técnicas de programação utilizadas em máquinas de registradores são diferem das utilizadas nas máquinas de pilha, tornando difícil a comparação de igualdade na forma de execução e consumo de recursos em ambos.

Com isto, além do uso do mesmo algoritmo base para a implementação dos programas específicos para cada máquina, são utilizadas somente instruções de chamadas de função sequenciais, ou seja, sem recursividade. Isto porque a pilha de retorno da uma vantagem à maquina de pilha em relação à máquina de registradores, que precisa alocar uma pilha em memória para efetuar o mesmo comportamento.

6.2.1 Analise de algoritmo

Para a melhor compreenção da programação das máquinas desenvolvidas neste trabalho, bem como o comportamento das pilhas diante do algoritmo utilizado, será analisado em detalhes o código utilizado para um dos testes da máquinas. O código analisado é o *bubble sort* para ambas as máquinas *0op* e *2op*, devido a sua complexidade e a quantidade de instruções utilizadas.

Inicialmente, o vetor de dados a ser ordenado é carregado em memória. Mostrado no trecho de Código 4, uma série de oito literais são passados à máquina *2op*, preenchendo assim a pilha de dados com o valor 3 como base e o valor 12 como topo. Na sequencia, oito instruções de escrita em memória são efetuados. Cada instrução de escrita referencia explicitamente um registrador temporal seguido do endereço de memória a ser escrito, sendo o valor 12 escrito na posição 0 e assim sucessivamente até que o valor 3 seja escrito na posição 7 da RAM.

Código 4: Processo de escrita em memória.

```
1  3              ## DS[Tos] = 3
2  8              ## DS[Tos] = 8
3  7              ## DS[Tos] = 7
4  9              ## DS[Tos] = 9
5  1              ## DS[Tos] = 1
6  2              ## DS[Tos] = 2
7  4              ## DS[Tos] = 4
8  12             ## DS[Tos] = 12
9  stro 0 0       ## Memoria[0] = DS[0]
10 stro 1 1       ## Memoria[1] = DS[1]
11 stro 2 2       ## Memoria[2] = DS[2]
12 stro 3 3       ## Memoria[3] = DS[3]
13 stro 4 4       ## Memoria[4] = DS[4]
14 stro 5 5       ## Memoria[5] = DS[5]
15 stro 6 6       ## Memoria[6] = DS[6]
16 stro 7 7       ## Memoria[7] = DS[7]
```

Na máquina *0op* o mesmo processo é realizado. Contudo, é usado uma mnemônica *store* especificando somente a posição de memória a ser utilizada. Apesar disto, o comportamento é o mesmo que no trecho de Código 4, visto que o *store* implicitamente utiliza o topo da pilha como fonte do valor a ser escrito. A principal diferença entre as duas máquinas o fato de que a *0op* desempilha o valor escrito enquanto que a *2op* somente lê o valor.

Com o vetor alocado em RAM, são inicializadas as variáveis locais para o controle dos índices analisados em memória utilizando a técnica de *intra-block-scheduling* proposta (PHILIP J. KOOPMAN, 1989). Para isto, as variáveis são alocadas na pilha e manejadas de uma forma que sempre permaneçam nela.

No trecho de Código 5 para a máquina *2op*, são carregados os valores 7 e 0 na pilha, sendo estes respectivamente a última e a primeira posição do vetor em memória. Na sequencia, a primeira posição do vetor que está mais ao topo da pilha é somado à um, sendo este resultado usado como endereço para carregar as primeiras posições do vetor em memória. O comportamento

deste trecho de código dobre as pilhas pode ser visto na Figura 29.

Código 5: Processo de leitura de memória e comparação para realizar a troca entre as posições em memória, na máquina *2op*.

```
1 7                 ## DS[Tos] = 7
2 0                 ## DS[Tos] = 0
3 ++ 0 1            ## DS[0] += 1
4 load 1            ## DS[Tos] = Memoria[DS[1]]
5 load 1            ## DS[Tos] = Memoria[DS[1]]
6 callm 1 0 26      ## DS[1] > DS[0] ? PC = 26 : PC++
```

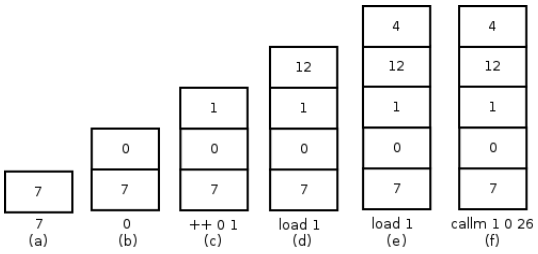

Figura 29: Comportamento da pilha de dados da *2op* com a comparação das primeiras posições do vetor em memória.

Na sequencia uma chamada condicional é realizada, verificando se o registrador temporal de valor 12 é maior que o registrador temporal 0 de valor 4. Caso a operação lógica seja verdadeira, a máquina efetuará uma chamada de função para o endereço 26 e caso contrario, o PC avançara.

Para a máquina *0op*, mostrado no trecho de Código 6, uma abordagem diferente é feita sobre o carregamento dos valores do vetor. Inicialmente, o valor zero é duplicado e então usado como endereço para ler o vetor em memória. Isto é feito desta forma pois a máquina irá desempilhar o topo da pilha, que será utilizado como endereço para poder efetuar a leitura. Desta forma, se a variável local não fosse duplicada, a mesma seria perdida.

Código 6: Processo de leitura de memória e comparação para realizar a troca entre as posições em memória, na máquina *0op*.

```
1 7             ## DS[Tos] = 7
2 0             ## DS[Tos] = 0
3 dup           ## DS[Tos] = DS[Tos]
```

```
4 load          ## DS[Tos] = Memoria[DS[Tos]]
5 pick 2        ## DS[Tos] = DS[2]
6 add 1         ## DS[Tos] = DS[Tos] + 1
7 load          ## DS[Tos] = Memoria[DS[Tos]]
8 <             ## DS[Tos] > DS[NTos]
```

Este comportamento pode ser visto na Figura 30, onde o valor de 0 duplicado *a* é sobrescrito pelo valor carregado da memória *b*. Na sequencia o valor de endereço da primeira posição é trazido ao topo da pilha por meio da mnemônica *pick*. Vale ressaltar que o *pick* considera o topo da pilha como a primeira posição, por isto o endereço usado para buscar zero é 2 e não 1, como seria o caso do mesmo endereçamento na máquina *2op*.

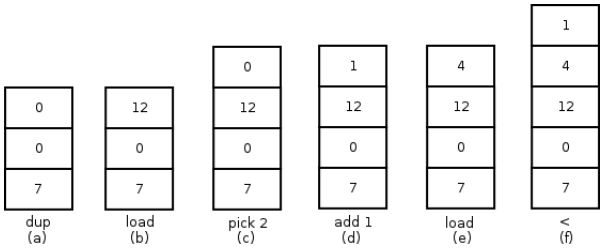

Figura 30: Comportamento da pilha de dados da *0op* com a comparação das primeiras posições do vetor em memória.

Com o endereço trazido ao topo da pilha, o mesmo é somado a um e usado para carregar o segundo valor do vetor em memória. Com os dois valores oriundos da memória carregados no topo da pilha, a instrução *f* é chamada para comparar se o valor mais ao topo da pilha é menor que o segundo ao topo da pilha. Esta operação sempre alocara no topo da pilha um valor lógico informando se a comparação foi verdadeira ou falsa.

Na sequencia, o Código 7 é utilizado na máquina *0op* para testar se é necessário ou não a troca de posições em memória dos valores carregados. De inicio, a mnemônica *branch0* é utilizada para efetuar um salto condicional baseado no valor do topo da pilha de dados ser igual a zero, além de desempilhar o valor utilizado para a comparação.

Código 7: Processo de comparação para realizar a a chamada de função ou seguir para a próxima iteração, na máquina *0op*.

```
1 branch0 27        ## DS[Tos] == 0 ? PC = 27 : PC++
2 call 40           ## RS[Tos] = PC, PC = 40
```

Vale ressaltar que o uso de *branch0* é amplamente usado em máquinas de pilha e em máquinas *Forth*. Esta mnemônica apresenta uma peculiaridade referente a sua comparação à zero, o que acaba levando o programador a sempre considerar somente uma lógica referente à resposta falsa. Isto é feito desta forma pois em *Forth*, o valor lógico de falso é um zero absoluto , enquanto que o valor verdadeiro é interpretado pela máquina como um valor absoluto um, ou seja, todos os *bits* de um valor verdadeiro apresentam o valor um (PELC, 2005).

A mnemônica *branch0*, vista também na Figura 31, faz com que a instrução de chamada do trecho de código responsável pela troca de posições no vetor seja chamado quando a comparação seja verdadeira e o desvio falhe. Caso o desvio seja efetivo, o endereço de código saltado se encontra logo após a chamada de função, fazendo assim com que independentemente da chamada de função, o fluxo de execução de código sempre convergirá para o trecho após a chamada.

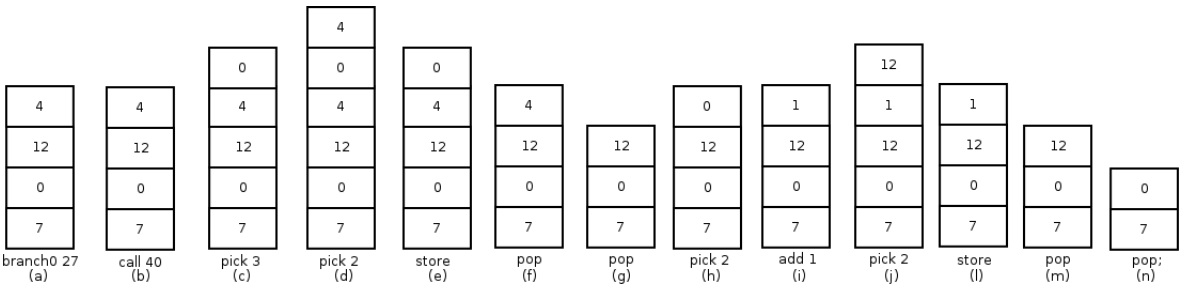

Figura 31: Comportamento da pilha de dados na troca de valores em memória.

Ao executar o salto, o processo mostrado na Figura 32 é realizado. De inicio, o PC atual é carregado para na pilha de retorno *a*, seguido da reescrita do registrador PC com o endereço da chamada. Ao terminar o processo de troca dos valore na RAM, o topo da pilha de retorno é desempilhado e este valor acrescido de um é escrito no registrador de PC, dando continuidade à execução do programa.

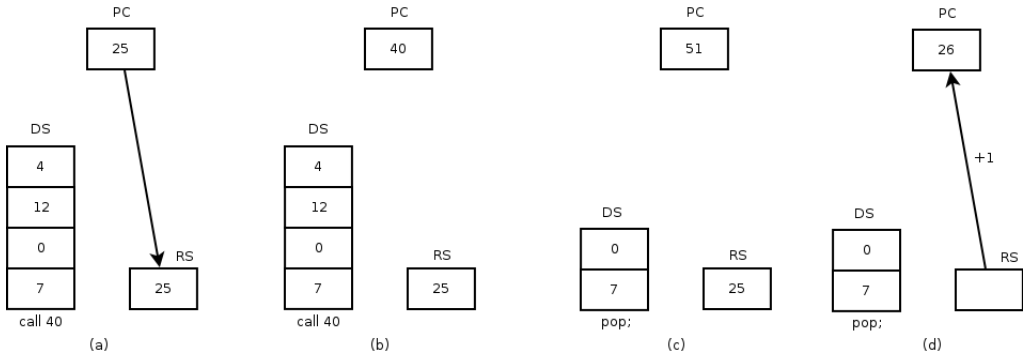

Figura 32: Efeito da chamada e retorno de função sobre a máquina.

Caso o a chamada se efetive, o endereço 40 é chamado e o tratamento para a troca de posição entre os valores analisados em memória é inicializado, como visto no Código 8 da máquina *0op* e na Figura 31. Como a operação de *store* utiliza o topo de dados como valor a ser escrito e Ntos como endereço, o primeiro *pick* traz ao topo o menor endereço *c*, seguido do valor carregado da posição endereço mais um do vetor *d*. Com o topo da pilha pronto a escrita do valor é feita sobre sua nova posição *e*, desempilhando o valor escrito.

Código 8: Processo de troca entre as posições em memória, na máquina *0op*.

```
1  pick 3        ## DS[Tos] = DS[3]
2  pick 2        ## DS[Tos] = DS[2]
3  store         ## Memoria[DS[NTos]] = DS[Tos]
4  pop           ## Tos = Tos -1
5  pop           ## Tos = Tos -1
6  pick 2        ## DS[Tos] = DS[2]
7  add 1         ## DS[Tos]++
8  pick 2        ## DS[Tos] = DS[2]
9  store         ## Memoria[DS[NTos]] = DS[Tos]
10 pop           ## Tos = Tos -1
11 pop;          ## Tos = Tos -1, PC = RS[Tos]
```

Em seguida, tanto o endereço da última escrita quanto o valor que já foi reescrito são desempilhados *f* e *g* com a mnemônica *pop*. Como o feito anteriormente, o endereço de base é trazido ao topo novamente *h*, só que desta vez acrescido de um *i*, visto que este irá ser usado para o valor carregado do endereço sem acréscimo do vetor *j*. Por fim a segunda escrita é realizada *l*,

terminando assim a troca de posições, restando apenas desempilhar as sobras da operação *m* e *n*, além do retorno da função, deixando somente as variáveis locais na pilha.

Já na execução do mesmo processo de troca pela máquina *2op*, visto na trecho de Código 9, a flexibilidade do endereçamento explicito facilita o uso da mnemônica *store*. Desta forma, ao invés de ter que posicionar de forma correta o endereço e o valor no topo da pilha, apenas duas operações de *store* endereçando quais registradores temporais são endereços e quais são valores.

Código 9: Processo de troca entre as posições em memória, na máquina *2op*.

```
1 store 1 2          ## Memoria[DS[2]] = DS[1]
2 store 0 3          ## Memoria[DS[3]] = DS[0]
3 return             ## PC = RS[Tos]
```

Na Figura 33 é visto em detalhes quais os valores utilizados para tal operação. Como os endereços foram usados na mesma ordem em que se encontram na base da pilha para o carregamento da memória, um simples cruzamento entre valores e endereços já é o suficiente para a escrita cruzada no vetor em memória. Assim, resta apenas realizar um retorno de função, visto que a última instrução do bloco é de memória e não de ULA.

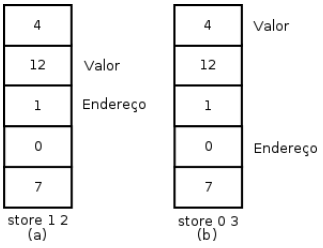

Figura 33: Endereçamento dos registradores temporais.

Com a efetivação, ou não, da troca em memória a execução converge para o trecho de Código 10 da máquina *2op*. Caso o endereço último endereço acrescido de um seja maior ou igual à variável local que indica a última posição do vetor a ser comparada, haverá um salto para o trecho de código onde a variável de última posição é decrescida e a variável de endereço atual é zerada.

Código 10: Processo de atribuição das variáveis locais ao topo da pilha, na máquina *2op*.

```
1 bme 2 4 29        ## DS[2] >= DS[4] ? PC = 29 : PC++
2 ++ 4 0            ## DS[Tos] = DS[4]
3 ++ 4 1            ## DS[Tos] = DS[1] + 1
4 branch 18         ## PC = 18
```

Caso o salto do Código 10 falhe, as duas variáveis locais, sendo a variável de endereço atual acrescida de um, são copiados no topo da pilha e a execução recomeça. Vale ressaltar que na máquina *0op* a organização da pilha é feita de tal forma que a cada execução do algoritmo, as variáveis são passadas ao topo e os demais valores da pilha são desconsiderados, já que a pilha é circular. Desta forma, para cada nova execução do algoritmo, a ocorrência da primeira variável local é tida como base da pilha de dados.

Um último teste é feito no trecho de Código 11, caso o salto do código anterior seja efetivo. De inicio, a variável local de última posição é decrescida de um, seguida da escrita do valor zero na variável de endereço atual. Na sequencia a nova variável de última posição é testada com 1 e, caso a comparação seja verdadeira, o fim do programa é chamado. Caso contrário, o as variáveis são trazidas novamente ao topo e a execução da função principal recomeça.

Código 11: Processo de atribuição das variáveis locais ao topo da pilha, na máquina *2op*.

```
1 -- 4 1            ## DS[Tos] = DS[4] - 1
2 0                 ## DS[Tos] = 0
3 1                 ## DS[Tos] = 1
4 beq 2 0 36        ## DS[2] == DS[0] ? PC = 36 : PC++
5 ++ 2 0            ## DS[Tos] = DS[2]
6 ++ 2 0            ## DS[Tos] = DS[2]
7 branch 18         ## PC = 18
```

Logo após a chamada no código da máquina *0op*, ocorre a convergência dos dois fluxos de execução, vistos no trecho de Código 12. Caso a chamada seja efetiva, a sequência da execução segue para o *branch 29*, que fica logo após a sequência de *pop*. Caso não ocorra a chamada, o *branch0* leva a execução diretamente para a sequência de *pop*, desempilhando os valores carregados da memória.

Código 12: Ponto de convergência para os fluxos de execução com troca ou não, na máquina *0op*.

```
1 branch0 27        ## DS[Tos] == 0 ? PC = 27 : PC++
2 call 40           ## RS[Tos] = PC, PC = 40
3 branch 29         ## PC = 29
4 pop               ## Tos = Tos -1
5 pop               ## Tos = Tos -1
```

Neste ponto, visto na Figura 34, a pilha de dados da máquina *0op* se encontra idêntica para ambos os fluxos de execução com troca em memória ou não. Aqui, são executados uma série de testes para definir se as variáveis locais devem ser trocadas ou se o fim da execução foi alcançado.

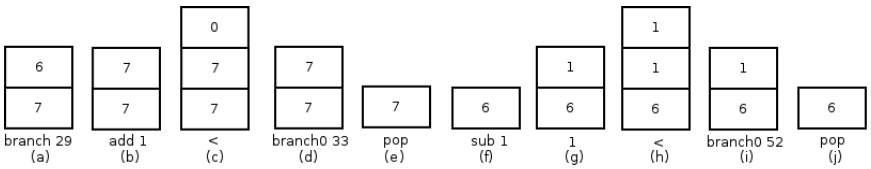

Figura 34: Testes realizados sobre a pilha de dados *0op*.

De inicio a variável de endereço atual é acrescida de um *b*, seguida de um teste para determinar se o novo valor da variável é menor que a variável de última posição. Caso não seja menor *d*, é necessário que a variável de última posição seja decrescida *f* e comparado com um *h* e caso seja verdadeiro, o fim do programa é chamada. Caso a variável de última posição seja maior um, somente a variável testada é deixada na pilha e o inicio do programa é chamado.

7 RESULTADOS

Para o levantamento dos resultados deste trabalho, foram utilizados os algoritmos de *insertion sort*, *bubble sort*, *Fibonacci* e fatorial, como visto no inicio da seção anterior. Cada um destes algoritmos foi implementado de forma similar para as máquinas *0op*, *2op* e MIPS, permitindo a comparação justa das arquiteturas abordadas.

Além disto, um algoritmo que faz uso de recursividade é implementado. Tendo este o intuito de reforçar as vantagens encontradas na utilização de pilhas, além das otimizações realizadas sobre elas.

Os testes da máquina MIPS realizados neste trabalho foram efetuados sobre o simulador *Mars* de versão 4.5 (MISSOURI STATE UNIVERSITY, 2014). Nele, além da possibilidade de simular a execução da máquina, é possível coletar informações como quantidade de ciclos de execução e quantidade de acesso a recursos como ULA e memória RAM.

Para realizar as medições, foram adicionados contadores aos módulos de *fetch* e RAM de ambas as máquinas de pilha. Desta forma, qualquer instrução executada ou qualquer leitura ou escrita sobre a memória deve fazer com que um contador seja acrescido, possibilitando determinar a utilização destes dois recursos.

Para os algoritmos de *bubble* e *insertion sort* foram usados os mesmos vetores de teste, variando entre os tamanhos de 8, 16 e 32 posições. Já para os algoritmos de *Fibonacci* e fatorial, são executadas 7, 14 e 28 iterações do algoritmo.

Ambas as máquinas de pilha implementadas estão configuradas com pilhas de dados e

retorno com 8 posições. Totalizando assim uma memória de 16 posições em ambas as implementações, contra as 32 posições da máquina MIPS.

Na Figura 35 são visualizadas as medições do número total de ciclos de execução para os algoritmos de *insertion sort*, *bubble sort*, *Fibonacci* e fatorial, respectivamente. No geral a máquina *2op* ganhou todos os *benchmarks*, seguida da máquina MIPS e por fim a máquina *0op*.

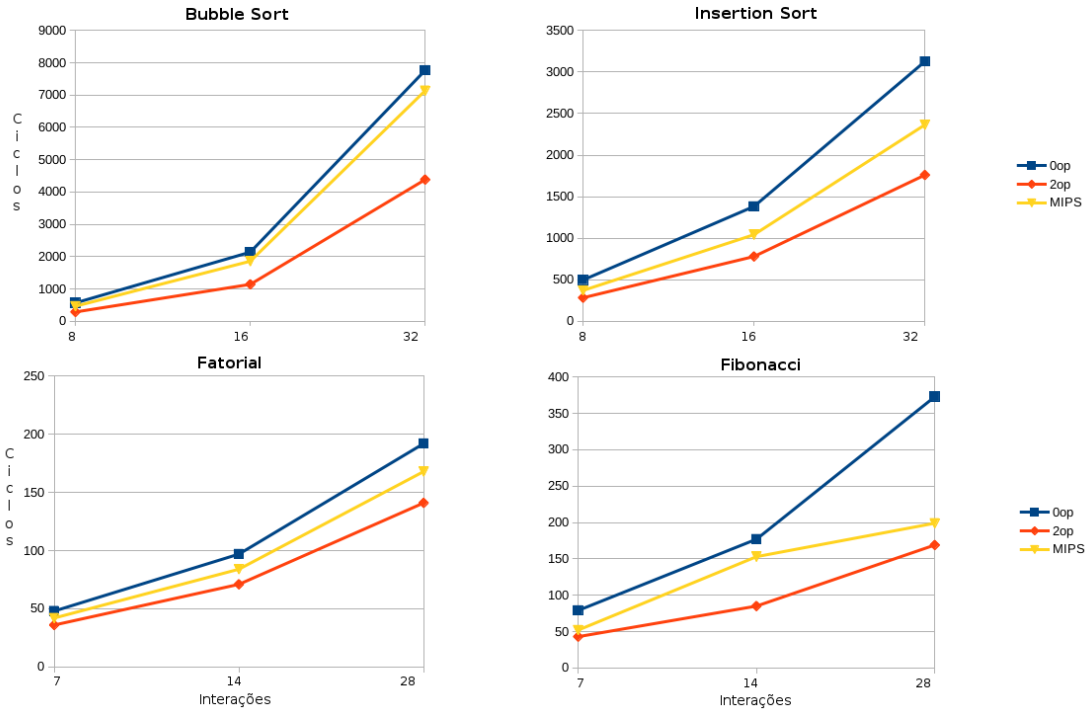

Figura 35: Resultados do total de ciclos de execução para todos os algoritmos.

A quantidade de ciclos de execução tem o comportamento semelhante em todos os algoritmos testados, tendo apenas uma margem de diferença no total de ciclos em cada teste. Contudo, o algoritmo de *fibonacci* apresenta um comportamento diferente dos demais, pois, enquanto as duas máquinas de pilha seguem o mesmo padrão de aumento, o total de ciclos de execução para a máquina MIPS decai o seu padrão de aumento conforme o número de interações.

Tal comportamento de diminuição da taxa de aumento dos ciclos de execução pode ser resultado da maior flexibilidade sobre a manipulação dos dados dentro do banco de registradores da máquina MIPS. Outra teoria sobre este comportamento é a técnica de programação empregada em ambas as máquinas de pilha, visto que a cada iteração o programa faz com que as variáveis locais sejam trazidas ao topo da pilha de dados, gerando um fluxo estático de realocações por iteração.

Já na Figura 36 são vistas as medição referentes aos totais de acessos à memória RAM para cada algoritmo. Ambos os algoritmos de ordenação tiveram o mesmo número de acessos em todas as máquinas, pois implementam o mesmo pseudo-algoritmo, enquanto que os algoritmos de fatorial e *fibonacci* não fazem uso de memória. Vale ressaltar que apesar das diferenças entre o estilo de programação e comportamento das máquinas, o algoritmo base é o mesmo e por isto o total de acessos para cada algoritmo é o mesmo em todas as máquinas.

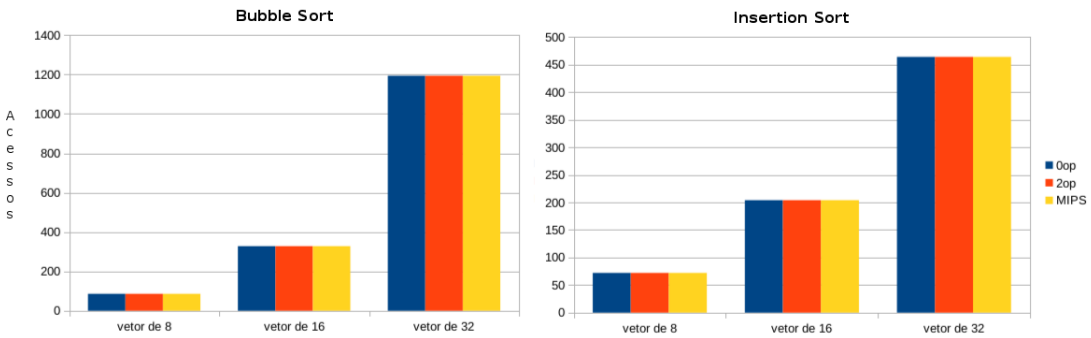

Figura 36: Resultados do total de acessos à memória para todos os algoritmos.

Por fim, é realizado um teste com o algoritmo fatorial recursivo para todas as máquinas, visto na Tabela 37. Apesar deste algoritmo quebrar o cenário de simulação estipulado neste trabalho, o funcionamento deste ajuda a visualizar as principais vantagens das máquinas desenvolvidas em relação a abordagem de máquina de registradores.

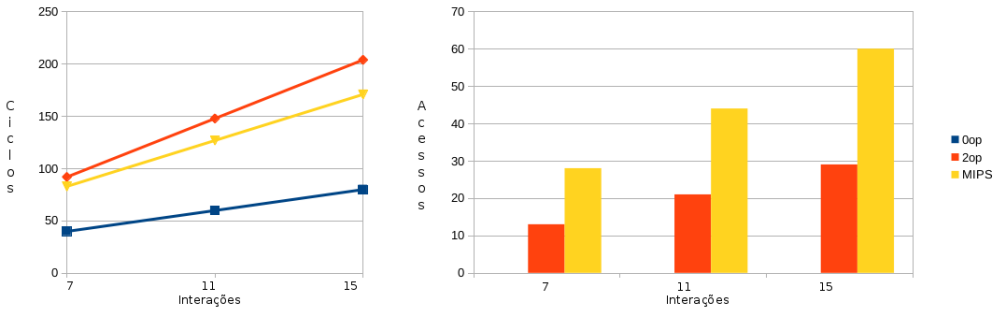

Figura 37: Resultados obtidos com o algoritmo recursivo. O gráfico da esquerda informa o total de ciclos de execução e o gráfico da direita informa o total de acessos à memória.

No algoritmo recursivo executado na máquina *0op*, as pilhas de dados e retorno foram capazes de conter todos os valores recursivos e endereços de retorno na execução de 7 iterações. Porém, para 11 e 15 iterações, foi necessário configurar ambas as pilhas da máquina para o seu tamanho máximo, de forma a suportar todos os dados e endereços recursivos.

Já no algoritmo recursivo executado na máquina *2op*, somente a pilha de retorno teve de ser alterada para o tamanho máximo. Nesta implementação, os endereços de retorno são contidos na pilha de retorno, enquanto que o os valores recursivos são alocados em pilha na memória. Caso o número de iterações seja baixo, é possível implementar um algoritmo que consiga manter os valores recursivos dentro da pilha de dados, mas como os testes foram realizados com até 15 iterações, esta implementação não foi possível.

Por fim o algoritmo implementado na máquina MIPS faz uso de uma pilha em memória tanto para endereços de retorno quanto para valores recursivos. Esta prática é comum em máquina de registradores pois o endereçamento explicito dos registradores dificulta o arranjo de muitos valores dentro do banco de registradores.

Como pode ser observado na Figura 37, a máquina *0op* consegue ter um desempenho muito superior ao visto nas demais máquinas. Além disto, a máquina *2op* pula pela primeira vez de primeira colocada nos *benchmarks* para o último lugar, ficando um pouco atraz da máquina MIPS.

Como a máquina *2op* utiliza-se da pilha de retorno para a alocação de endereços, a mesma deveria apresentar uma melhor desempenho em relação ao MIPS. Desta forma, tal perda na comparação do *benchmark* pode ser atribuida ao algoritmo utilizado na máquina *2op*, visto que o número de acessos à memória é menor.

A Figura 38 mostra o tamanho de código obtido para cada um dos programas implementados para este teste, sem contar as instruções usadas para carregar os dados no vetor em memória. No geral, a máquina MIPS obteve os menores tamanhos de código, seguido da máquina *2op* e por último a máquina *0op*.

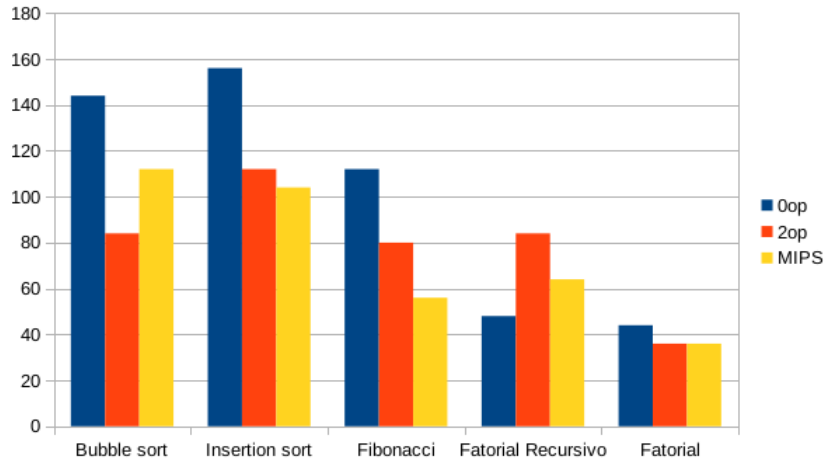

Figura 38: Tamanho de programa para cada um dos algoritmos em *bytes*.

Os resultados obtidos mostraram que a implementação da máquina *2op* foi a mais efetiva de todas na execução dos algoritmos sem recursividade. Este fato pode ser justificado pelo aproveitamento eficiente da localidade dos registradores temporais, visto que o fluxo de execução acaba deixando as variáveis mais usadas sempre em posições mais ao topo da pilha em relação as menos usadas.

Além disto, as variáveis locais do algoritmo são sempre trazidas ao topo da pilha sem a necessidade de desempilhar nenhuma posição que esteja no caminho, como o que é feito em máquinas de pilha puras. Com isto, a mesma praticidade de reescrever um registrador usado para determinada variável em uma máquina de registradores pode ser alcançada na máquina *2op*.

Outra utilidade importante da máquina de dois operandos é a possibilidade de realizar chamadas de função condicionais. Esta facilidade não é encontrada em nenhuma outra das máquinas testadas e acaba economizando algumas instruções do programa, visto que as camadas não precisam ser precedidas por saltos condicionais, podendo também vir a facilitar a lógica de programação.

Neste cenário de execução sem uso de recursividade ou mais de uma chamada sucessiva, a máquina *0op* teve os piores resultados em todos os testes. Contudo, estes resultados não significam necessariamente que uma máquina pura tem desempenho inferior à máquina de registradores.

Na verdade uma máquina pura executando longos trechos de código sem uso de chamadas de função tende sempre a ter menor desempenho se comparado à máquina MIPS. Isto porque trechos de código longos tendem a fazer maior uso do banco de registradores ou pilha, fazendo com que a flexibilidade do endereçamento explicito obtenha uma vantagem sobre o acesso dos valores comparado à máquina de pilha.

Esta vantagem se da pois em cenários onde a pilha de dados detenha muitos valores, o acesso à valores empilhados mais perto da base é dificultado. Além disto, algumas operações requerem que os valores estejam alinhados de uma forma especifica no topo da pilha, tornando necessário que, em algumas ocasiões, o topo da pilha tenha que ser ajeitados para que a instrução desejada seja efetuada.

O contraponto a este cenário é o resultado obtido na execução recursiva, onde a máquina *0op* ganha com menos da metade dos ciclos de execução da máquina MIPS e sem nenhum acesso à memória. Neste caso, a máquina de registradores necessita montar uma pilha em memória com os dados e endereços de retorno de cada iteração, enquanto que a máquina de pilha só necessita alocar estes valores em suas devidas pilhas a cada iteração.

Em casos similares, onde o código utiliza uma ou mais chamadas consecutivas de função, a máquina de pilha pura tende a obter melhores resultados que uma máquina de registradores devido ao comportamento das pilhas citado anteriormente. Além disto, chamadas consecutivas em algoritmos otimizados para máquinas de pilha tendem a ter poucos dados empilhados por iteração. Desta forma, os valores que são retornados ao *calle* tendem a estar mais ao topo da pilha, facilitando o seu acesso.

No caso da execução do algoritmo recursivo na máquina *2op*, o comportamento temporal da pilha de dados faz com que somente os endereços de retorno sejam mantidos em CPU, enquanto que os dados precisam ser alocados em uma pilha em memória. Com isto os acessos à memória RAM não são tão numerosos quanto os acessos da máquina MIPS desempenhando a mesma função. Contudo, o desempenho da máquina *2op* em relação ao ciclos de execução ficou um pouco atraz dos resultados amostrados na máquina MIPS.

8 CONCLUSÃO

O Capítulo 2 apresentou os conceitos básicos sobre máquinas de pilha, permitindo assim entender o funcionamento destas máquinas. Com o conhecimento básico sobre estas máquinas, consegue-se analisar e entender as tomadas de decisões na implementação das máquinas de pilha existentes, vistas no Capítulo 3. Neste último capítulo, são abordados com mais afinco o formato de instrução *hardwired* utilizado, a forma de acesso às pilhas e otimizações propostas.

No Capítulo 4, é analisado todo o comportamento básico possível à uma máquina de pilha. Neste, são exploradas as combinações de tamanho de pilha, quantidade de pilhas e número de operandos, analisando nestas as principais vantagens e desvantagens de sua implementação.

Com a analise feita, foram idealizadas duas máquinas de pilha no Capítulo 5. Ambas as máquinas apresentam duas pilhas de dados e implementam 2 ou nenhum operandos. Estas foram concebidas para avaliar se existe alguma vantagem entre as máquinas em si ou em relação à uma máquina de registradores.

O Capítulo 6 da uma visão geral de como o *assembler* para ambas as máquinas foi implementado, e quais instruções se fazem presentes. Além disto, é explicado em detalhes como o algoritmo de *bubble sort* funciona em ambas as máquinas de zero e 2 operandos, ilustrando tanto o comportamento das máquinas quanto as técnicas de programação otimizadas para pilha.

Por fim, o Capítulo 7 mostra os resultados das execuções dos algoritmos especificados em ambas as máquinas desenvolvidas e em uma máquina MIPS, levando em conta o total de ciclos de execução, acessos à memória e tamanho de programa. Além disto, é feita uma análise sobre os resultados obtidos, analisando as possíveis causas de ganhos e perdas de desempenho das

máquinas para cada cenário de execução.

Este trabalho mostrou em detalhes a fundamentação teórica sobre o funcionamento e otimização de uma máquina de pilha. Além de comparações diretas entre estas máquinas e a de registradores.

Ao contrario dos trabalhos usados como referência, onde o desempenho das máquinas é uma comparação direta do uso de *Forth* em uma máquina de pilha (simulada ou física) em relação a uma implementação similar na linguagem C em uma máquina de registradores, os testes são feitos em nível de instrução de máquina. Desta forma, consegue-se observar o desempenho da CPU em si, ao invés do desempenho do compilador e do conjunto de ferramentas.

Com isto, tem-se uma métrica para máquina de pilha que é completamente independente da linguagem *Forth*, indo contra a noção passada pela bibliografia de que as arquiteturas de máquina de pilha são somente uma extensão física para a linguagem *Forth*. Tal independência permite o teste de uma máquina de pilha com dois operandos, que devido a organização não é suportada por *Forth* e por isto ignorada na maior parte da bibliografia.

8.1 Considerações finais

Ambas as abordagens de máquinas de pilha estudadas e implementadas neste trabalho tiveram pouca pesquisa no âmbito acadêmico ao longo do tempo. Das pesquisas encontradas, a maior parte toma como foco a linguagem *Forth*, dando a impressão que este tipo de máquina é uma consequência direta da implementação física da máquina virtual *Forth*.

Como a máquina *0op* é inspirada em outros modelos de máquina *Forth*, a mesma pode ser integrada com a linguagem. Desta forma, a implementação física em FPGA ou circuito impresso desta máquina tem como vantagem a existência de um compilador já implementado.

A máquina *2op* pode ser considerada como um meio termo entre arquiteturas de máquinas de puras e máquinas de registradores, aproveitando-se tanto da flexibilidade do endereçamento explicito quanto do uso de pilhas na CPU. Apesar do melhor desempenho em parte dos testes, não há exemplos significativos na bibliografia analisada sobre máquinas de dois operandos, além

da inexistência de um compilador especifico para este tipo de máquina.

Ainda, este trabalho falha ao fazer uma análise do desempenho das máquinas para algoritmos mais complexos devido a simplicidade do *assembler* desenvolvido, além da não integração da linguagem *Forth* com a máquina *0op* e a inexistência de um compilador para a máquina *2op*. Contudo, o comportamento básico das funcionalidades fundamentais de uma CPU como operações lógico/aritmética e memória puderam ser testadas, permitindo assim uma análise fundamental do comportamento e desempenho básico das máquinas.

8.2 Trabalhos futuros

Apesar de que ambas as máquinas implementadas para este trabalho apresentem um tamanho ajustável a até 16 palavras, a pilha de dados pode vir a não suportar a quantidade de dados sendo alocados em algoritmos mais complexos ou gulosos. Neste cenário, o programador ou o compilador é obrigado a desempilhar dados para a memória, obtendo assim uma perda de desempenho significativa.

Este problema pode ser contornado na máquina virtual *Forth* ou em máquinas que aloquem suas pilhas diretamente em memória, pois nestes casos a pilha pode ser alocada usando todo o espaço de memória. Apesar da possibilidade de alocar pilhas de tamanhos grandes, uma implementação de CPU em *hardware* utilizando somente memória RAM acaba por ter seu desempenho comprometido visto que o tempo de acesso à memória é mais demorado que o acesso à um registrador em CPU.

Com isto, uma arquitetura de máquina que permitisse alocar pilhas de dados ou retorno em memória e que, ao mesmo tempo, utilizasse de uma pilha de registradores local à CPU para o acesso dos dados seria mais flexível e eficiente do que uma máquina de pilha simples. Isto porque esta configuração permite alocar uma pilha de tamanho grande em memória e acessar o topo da mesma via uma pequena pilha de *buffers* localizados dentro da CPU, semelhante ao mostrado na Figura 39.

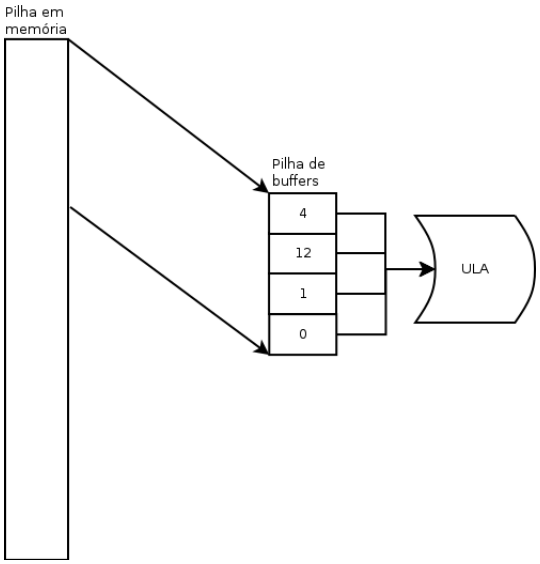

Figura 39: Pilha de *buffers* sincronizada com a pilha alocada em memória.

Para tal configuração é necessário implementar também um algoritmo de *cache* para manter sincronizada a pilha em CPU com o topo da pilha em memória. Seria vantajoso a este método que as escritas ou leituras de sincronização fossem realizadas em paralelo a execução da CPU, quando o recurso de memória não estivesse sendo usado e gerando interrupções nos piores casos onde o dado requerido ainda não tenha sido sincronizado com a memória.

Tal otimização teria como principais vantagens a maior flexibilidade referente ao tamanho da pilha alocada, menor quantidade de registradores ou *buffers* alocados em CPU e uma melhora no tempo de troca de contexto, visto que os dados da CPU já estariam sincronizados em memória. Contudo esta otimização teria maior efeito em máquinas de zero operandos, visto que em máquinas de dois operandos todas as posições da pilha podem ser acessadas, fazendo necessário que somente o tamanho da pilha em CPU seja alocada e sincronizada em memória, ou o dado requerido não esteja contido na pilha em CPU e um ciclo seja gasto trazendo tal valor da memória RAM.

REFERÊNCIAS

BAILEY, Christopher. *Optimisation Techniques for Stack Based Architectures*. 1996. 185p. Tese (Doutorado em Ciência da Computação) — University of Teesside.

BOWMAN, James. J1: a small forth cpu core for fpgas. In: EUROFORTH, 2010, Menlo Park, CA. *Proceedings...* [S.l.: s.n.], 2010. p.4. Disponivel em: < http://www.excamera.com/sphinx/fpga-j1.html >. Acesso em: Mar. 2015.

BRODIE, Leo. *Thinking Forth*. [S.l.]: Punchy Publising, 2004. 295–313p.

DAVID A. PATTERSON, John L. Hennessy. *Computer organization and design*. [S.l.]: Morgan Kaufmann, 2005. 689p. v.3.

GODARD, Ivan. Drinking from the Firehouse. In: OF THE , 2015. *Anais...* [S.l.: s.n.], 2015. Disponivel em: < http://millcomputing.com/ >. Acesso em: May. 2015.

KOOPMAN, Phil. Microcoded versus Hard-wired Control. , [S.l.], p.8, 1987. Disponivel em: < http://users.ece.cmu.edu/ koopman/misc/byte87a.pdf >. Acesso em: Jun. 2015.

MISSOURI STATE UNIVERSITY, Missouri. *MARS (MIPS Assembler and Runtime Simulator)*. Disponivel em: < http://http://courses.missouristate.edu/kenvollmar/mars/ >. Acesso em: Nov. 2015.

PELC, Stephen. *Programming Forth*. 133 Hill Lane, Southampton, SO15 5AF, UK: MicroProcessor Engineering Limited, 2005. 182–192p.

PHILIP J. KOOPMAN, Jr. *Stack Computers, the new wave*. [S.l.]: Moutain View Press, 1989. v.1.

SCHLEISIEK, Klaus. MicroCore, an Open-Source, Scalable, Dual-Stack, Harvard Processor Synthesisable VHDL for FPGAs. , [S.l.], p.23, 2004. Disponivel em: < http://microcore.org >. Acesso em: Mar. 2015.

SCHLEISIEK, Klaus. Disponivel em: < http://microcore.org/index.html >. Acesso em: Mar. 2015.

SYSTEMC® Reference Manual. 3 Park Avenue ,New York, NY 10016-5997, USA: IEEE Computer Society, 2012. IEEE Standard for Standard SystemC® Language Reference Manual.