ID: .....

Desenvolvimento e Descrição de uma Arquitetura de Pilha

São José dos Campos - Brasil

Abril de 2019

ID: .....

Desenvolvimento e Descrição de uma Arquitetura de Pilha

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Docente: Prof. Dr. Tiago de Oliveira

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Abril de 2019

Resumo

O objeto de estudo desse relatório é o projeto e a implementação de um sistema digital em lógica programável composto por processador, memória e interface de comunicação. Podendo ser uma arquitetura de *Pilha*, *Acumulador*, *Registrador-Memória* ou *Registrador-Registrador*, o objetivo é desenvolver todo o projeto necessário para o bom funcionamento da arquitetura escolhida, desde a definição do conjunto de instruções, do caminho de dados (*datapath*) e dos modos de endereçamento, até a implementação de cada uma das unidades em linguagem de descrição de *hardware*, *Verilog*. Segundo a literatura, a arquitetura de pilha, como aqui é abordada, é uma arquitetura bastante importante, dado seu alto nível de desempenho com uma baixa complexidade de implementação, sendo muito interessante tanto o seu estudo quanto a sua escolha para a realização do projeto. Após a revisão bibliográfica sobre arquitetura e organização de computadores, com ênfase nas arquiteturas de pilha em (1), desenvolveu-se o conjunto de instruções da máquina, o caminho de dados e os modos de endereçamento requeridos. Com o estudo sistemático e aprofundado desse tipo de arquitetura, tivemos a oportunidade de aprimorar o entendimento sobre o assunto bem como embasar de forma robusta os próximos passos de implementação do projeto a serem desenvolvidos.

Palavras-chaves: Arquitetura e Organização de Computadores. *Hardware*. Circuitos Digitais. *Stack Architecture*. Arquiteturas de Pilha. Processador.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

De microprocessadores em um único *chip*, que custam alguns poucos dólares, até supercomputadores, que custam dezenas de milhões de dólares, muitos produtos podem justificadamente reivindicar o nome de *computador*. Segundo (2), essa variedade não aparece só no custo, mas também no tamanho, no desempenho e na aplicação. Se considerarmos ainda as mudanças que afetam a tecnologia computacional, seja em circuitos integrados, utilizados para construir os componentes dos computadores, até preocupações de realização paralela de tarefas para melhora de desempenho, o estudo de Arquitetura e Organização de Computadores se mostra uma área bastante fértil.

Segundo (2), um sistema de computador, como qualquer outro sistema, consiste em um conjunto inter-relacionado de componentes. O sistema, porém, se faz mais bem caracterizado em termos de **estrutura** (a forma como os componentes estão ligados) e de **função** (a operação que cada um dos componentes realiza). Dessa forma, podemos organizar hierarquicamente um computador, em uma abordagem *top-down*: **sistema computacional**, que tem por principais componentes o processador, a memória e os periféricos de entrada e saída; **processador**, que os principais componentes são a unidade de controle, os registradores, a **ULA** - Unidade Lógico-Aritmética - e a unidade de execução de instruções; **unidade de controle**, que fornece os sinais de controle para operação e coordenação de todos os componentes do processador.

Dessa hierarquia, quando analisamos a forma como é organizada, em termos de como realizamos as interconexões, isto é, como é a estrutura do computador, estaremos falando de **organização de computadores**. O entendimento de **arquitetura de computadores**, por outro lado, se refere aos atributos de um sistema visíveis a um programador, ou seja, aos atributos que possuem um impacto direto na lógica de execução de um programa, isto é, recai sobre o estudo da função dos componentes.

2 Objetivos

2.1 Geral

A unidade curricular de *Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores* faz parte das unidades curriculares do curso que visam possibilitar ao alunos que desenvolva um sistema computacional completo, integrando *hardware* e *software*. Como pode ser visto em (3), o projeto geral dessas unidades curriculares é o de desenvolver um processador, o de definir uma linguagem de programação, o de desenvolver um compilador, o de definir um sistema operacional e finalmente o de estabelecer um processo de comunicação entre dois ou mais sistemas.

Nesse ínterim, o presente projeto visa o desenvolvimento de um sistema digital composto por processador, memória e interfaces de comunicação. Em outros termos, se objetiva descrever a arquitetura de um processador utilizando linguagem de descrição de *hardware*, *Verilog*, utilizando lógica programável, realizar simulações e testes que corroborem com a funcionalidade do projeto, desenvolver um sistema em lógica programável de um sistema de memória e de um sistema de comunicação.

2.2 Específico

Os objetivos específicos dessa primeira etapa do projeto consistem no desenvolvimento da arquitetura. Mais especificamente, temos:

- Definição de um conjunto de instruções (*Instruction Set Architecture - ISA*):
  - Definição do tipo de máquina, dentre as opções de pilha, acumulador, registrador-memória, registrador-registrador;
  - definição dos modos de endereçamento.
- Criação das instruções:
  - Definição do número de operandos e de operações por instrução;
  - Definição dos tipos de instrução;
  - Definição do significado dos *bits* nas instruções.
- Desenvolvimento do *datapath* e do diagrama da arquitetura.

3 Fundamentação Teórica

Como mencionado nos objetivos específicos desse trabalho, grande parte do estudo consistiu em revisão bibliográfica para possibilitar as definições de como a arquitetura elaborada seria feita. Dessa maneira, alguns pontos essenciais da literatura devem ser destacados, com ênfase às obras de *Stallings* (2) para os conceitos gerais e de *Koopman* (1) para os conceitos relacionados à pilhas.

3.1 Arquitetura e Organização de Computadores

Segundo (2), **arquitetura de computador** refere-se aos atributos de um sistema visíveis a um programador ou, em outras palavras, aqueles atributos que possuem um impacto direto sobre a execução lógica de um programa. Exemplos de atributos de arquitetura são o conjunto de instruções, o número de *bits* utilizado para cada tipo de variável, como números e caracteres, os mecanismos de entrada e saída (E/S) e as técnicas de endereçamento de memória.

Por outro lado, **organização de computador** refere-se às unidades operacionais e suas interconexões que realizam e viabilizam as especificações da arquitetura. Exemplos de atributos de organização são os detalhes de *hardware* transparentes ao programador, como sinais de controle, interfaces entre computador e periféricos e a tecnologia de memória utilizada.

3.2 Arquitetura do Conjunto de Instruções

Ainda segundo (2), **arquitetura do conjunto de instruções** (***Instruction Set Architecture - ISA***) é um termo muitas vezes utilizado no lugar de "arquitetura de computadores". Se considerarmos o fato de que o *ISA* define não só os formatos das instruções, os códigos de operação (*opcodes*), os registradores utilizados, a memória de dados e de instrução, como também define o efeito das instruções executadas nos registradores e na memória e ainda define o controle da execução das instruções, é natural que sua determinação seja suficiente para determinar todos os pontos necessários à arquitetura do computador, justificando a equivalência dos termos.

Determinar o *ISA* de um computador é de extrema importância, pois o *ISA* corresponde aos níveis de linguagem de montagem e de máquina, e é o conjunto de instruções que o processador é capaz de executar. Como ele utiliza mnemônicos (*opcodes*) para referenciar as instruções binárias, o acesso aos componentes da máquina se vê

facilitado, servindo assim de interface com programador, ainda que em baixo nível.

Dentre as definições do *ISA*, destacamos a necessidade de definir o formato das instruções. Quando trabalhamos sobre o formato de uma instrução, devemos fazer com que ela possua alguns itens básicos: operação a ser executada, operandos requeridos, local de destino do resultado e qual a próxima instrução a ser executada. Dentre esses requisitos, o item de operandos requeridos requer uma atenção especial. Isso se deve ao fato de que, dependendo de como definimos nossas instruções, esse operando deve ser buscado na memória ou diretamente de lido de uma forma específica. A busca desses operandos consiste nos modos de endereçamento das instruções.

3.2.1 Modos de Endereçamento

De acordo com (2), temos vários tipos básicos de endereçamento. Utilizamos a notação do livro (2):

- A = Conteúdo de um campo de endereço dentro da instrução
- R = Conteúdos de um campo de endereço dentro da instrução se refere a um registrador
- EA = Endereço real (efetivo) do local que contém o operando referenciado
- (X) = Conteúdos do local de memória X ou do registrador X

Com esta nomenclatura, os modos de endereçamento são:

Modo	**Algoritmo**
Imediato	Operando = A
Direto	$EA = A$
Indireto	$EA = (A)$
Por registrador	$EA = R$
Indireto por registrador	$EA = (R)$
Por deslocamento	$EA = A + (R)$
De pilha	EA = topo da pilha

Tabela 1 – Modos básicos de Endereçamento. Fonte: (2)

A discussão acerca de modos e endereçamento é bastante extensa e, dada a simplicidade da arquitetura que será realizada, discutiremos de forma mais aprofundada apenas os modos que serão utilizados. Em ordem decrescente da frequência com que ocorrerão.

3.2.1.1 Endereçamento por pilha

Como estamos trabalhando com uma arquitetura de pilha, o endereçamento de operandos por pilha será o mais comum. Como veremos quando aprofundarmos no conceito

de pilha, a pilha será uma porção de posições sequenciais na memória, respeitando a regra de **LIFO**, ou *Last In, First Out*, que significa que o último elemento a entrar é o primeiro a sair. Usualmente, temos um registrador que guarda o topo dessa pilha, que seria o último elemento adicionado nela. Dessa maneira, é um modo de endereçamento que não precisa referenciar o operando que está sendo utilizado, pois este sempre será o topo da pilha, isto é, é um endereçamento implícito. A desvantagem é que esse modo tem aplicabilidade limitada e pouca flexibilidade na escolha dos operandos.

3.2.1.2 Endereçamento por Imediato

O endereçamento por imediato consiste na forma mais simples de endereçamento. A ideia dele é que na própria instrução já está presente o valor a ser armazenado. É um modo que pode ser utilizado para definir e utilizar constantes ou definir valores iniciais de variáveis. Sua vantagem é que não requer nenhuma referência à memória, o que economiza um ciclo da instrução. Por outro lado, o valor do imediato passado é limitado ao tamanho da instrução, o que muitas vezes é bem inferior ao tamanho da palavra necessária.

3.2.1.3 Endereçamento por Deslocamento

O endereçamento por deslocamento requer dois campos de endereço, dois quais só é necessário explicitar um. O primeiro campo, que pode ser implícito, se refere a um endereço utilizado de referência, que será utilizado diretamente. O segundo campo, que é explícito, refere-se a um registrador cujo conteúdo será somado ao endereço do primeiro campo.

Existem três tipos mais comuns de endereçamento por deslocamento. São eles:

- **endereçamento relativo:** Também conhecido como *PC*-relativo, o registrador implicitamente referenciado é o próprio contador do programa (PC). O campo de endereço do registrador que deveria guardar o número a ser somado é normalmente tratado como o próprio número que deve ser somado. Dessa forma, o endereço efetivo é o deslocamento relativo ao endereço da instrução.

- **endereçamento por registrador base:** Nesse tipo de endereçamento, o registrador base contém um endereço da memória principal e o campo de endereço contém um deslocamento (usualmente, um inteiro sem sinal).

- **endereçamento por indexação:** Ao contrário do endereçamento por registrador base, no endereçamento por indexação o endereço da memória principal é guardada no campo de endereço e no registrador base contém o deslocamento. É um modo de endereçamento bom para rotinas iterativas.

3.2.2 RISC - *Reduced Instruction Set Computer*

Tendo definido de acordo com (2) os conceitos da arquitetura do conjunto de instruções e de seus modos de endereçamento, podemos mais facilmente diferenciar os tipos de conjuntos de instruções.

Quando falamos de um conjunto de instruções do tipo **RISC** (*Reduced Instruction Set Computer*), estamos falando de uma arquitetura de conjunto de instruções reduzida. Nesse tipo, o processador é simples e executa apenas as instruções mais utilizadas, mantendo todas as instruções de mesma duração. Como visa a simplicidade, poucos modos de endereçamento são utilizados. Além disso, como todas as instruções têm mesma duração, o uso de *pipeline* é bastante intenso nesse tipo de arquitetura.

Exemplos de máquinas que implementam RISC são a MIPS, o PowerPC e o Spark, da Apache.

3.2.3 CISC - *Complex Instruction Set Computer*

Um conjunto de instruções do tipo **CISC** (*Complex Instruction Set Computer*) é um conjunto de instruções bem mais complexo do que o RISC. Enquanto o RISC tem algumas dezenas de instruções, o CISC trabalha com centenas de instruções. Como tem muito mais instruções do que o RISC, é bem mais fácil para o programador que o está utilizando realizar tarefas com poucas instruções, em um código mais enxuto. Porém, há um custo alto de complexidade na execução dessas instruções para decodificação dessas instruções.

3.2.4 Conjunto de Instruções de Pilha

A discussão RISC *versus* CISC dá muitas vezes a entender que uma arquitetura deve ser ou uma ou outra. Porém, como é destacado em (2), é muito difícil de fato uma máquina que possa ser descrita como apenas uma delas. Além disso, com o aumento da densidade dos *chips* e da velocidade do *hardware*, conjuntos de instruções RISC pode se tornar mais complexo, e com a busca por um maior desempenho, conjuntos de instruções CISC começaram a se voltar para questões antes da arquitetura RISC, como aumentar o número de registradores. Dessa forma, ambas tecnologias se encontram em convergência gradual e em mistura.

Nesse contexto, é natural que quando falamos de arquiteturas de conjuntos de instrução de pilha não consigamos classificá-la em apenas uma dessas arquiteturas. Em (1) é explicitado que, como uma arquitetura RISC, as arquiteturas de pilha têm primitivas simples que executam em um único ciclo de *clock* e, como uma arquitetura CISC, estas arquiteturas fazem uso intensivo de micro-rotinas de forma barata.

3.3 Tipos de Arquiteturas

De acordo com como lidamos com o acesso a memória e onde estão os operandos, temos quatro possibilidades de arquitetura: **pilha**, **acumulador**, **registrador-memória** e **registrador-registrador**.

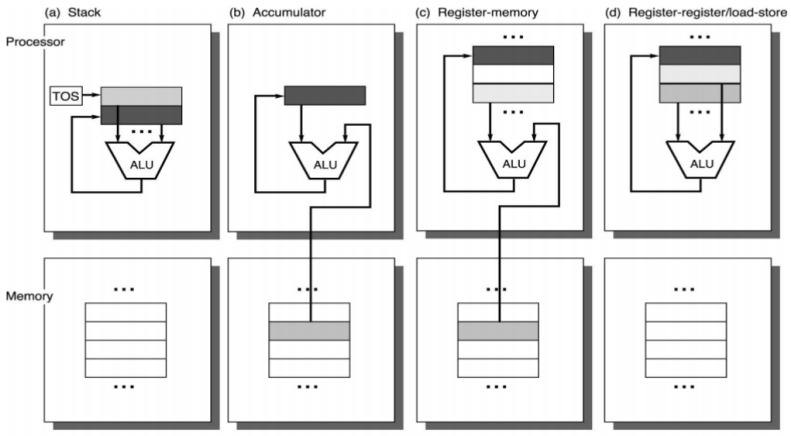


Figura 1 – Os quatro tipos principais de arquitetura. Da esquerda para a direita: pilha, acumulador, registrador-memória, registrador-registrador.

3.3.1 Pilha

Como podemos observar na figura 1, na primeira coluna, uma arquitetura de pilha é aquela que sempre executa suas operações sobre os termos da pilha, com referência explícita para o topo desta. É uma arquitetura de instruções curtas e bastante rápida, porém com pouca flexibilidade de modos de endereçamento.

3.3.2 Acumulador

A arquitetura de acumulador, que é representada na segunda coluna da figura 1, tem apenas um registrador que funciona como acumulador, e sempre é uma das entradas e é a saída da Unidade Lógico-Aritmética (ULA). O outro parâmetro vêm sempre da memória. É uma arquitetura que diminui o número de estados em relação à de pilha, mas perde por ter acesso constante à memória.

3.3.3 Registrador-Memória

A arquitetura de registrador-memória, representada na terceira coluna da figura 1, é parecida com a de acumulador, por ter um operando sempre vindo da memória, mas ganha em flexibilidade por ter vários registradores para lidar com a entrada e a saída da

ULA. Suas instruções são mais longas por ter de passar um operando da memória e outro dos registradores, mas é bastante flexível.

3.3.4 Registrador-Registrador

A arquitetura de registrador-registrador, que é representada na última coluna da figura 1, é a mais usual atualmente. É mais rápida em relação à registrador-memória por não ter que ficar acessando a memória sempre, e é mais flexível por lidar só com registradores. Porém, suas instruções são bastante longas e é mais difícil o controle e a decodificação destas.

3.4 Arquitetura de Pilha

Das quatro possibilidades de arquitetura citadas na seção 3.3, a que será implementada nesse trabalho será a de pilha. Desta forma, vamos aprofundar nos conceitos relacionados à arquiteturas de pilha, para não só embasar como também justificar o trabalho realizado.

3.4.1 Pilha - Estrutura de Dados

A ideia de **pilha** recai sobre estruturas de dados que respeitam a regra **LIFO** (*Last-In,First-Out*), que significa que o último elemento a entrar na estrutura será o primeiro a sair. Como podemos notar no exemplo a seguir, que é encontrado em (1), quando vamos inserir elementos (*push*) precisamos explicitar quem está sendo inserido, mas quando vamos retirar um elemento, não precisamos passar nenhum parâmetro, pois a regra garante que o elemento que deve ser retirado é o último inserido.

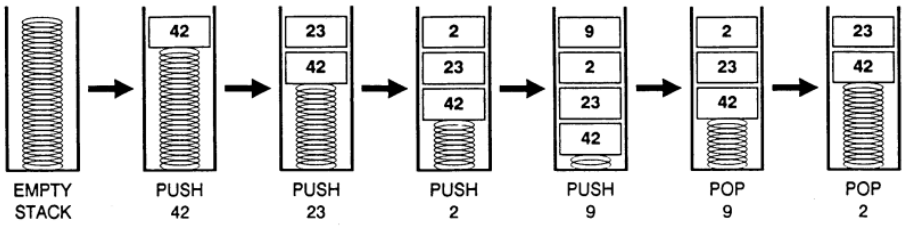


Figura 2 – Exemplo de operações sobre uma pilha. Fonte: (1)

Segundo (1), pilhas são, conceitualmente, o jeito mais fácil de salvar informação em armazenamento temporário. Além disso, muitas expressões matemáticas podem ser calculadas de forma simples com chamadas recursivas, de forma que a estrutura de pilha é bastante útil.

3.4.2 Implementações de Pilha em *Software* e *Hardware*

Existem várias maneiras de implementar pilhas. A mais comum é alocar um *array* na memória e manter uma variável que controla o índice relativo ao último elemento inserido. Como já comentamos, duas serão as funções executadas sobre a pilha:

- ***Pushing*:** Consiste em alocar uma nova palavra na pilha e colocar o dado nela.
- ***Popping*:** Consiste em remover o último elemento inserido, isto é, o topo da pilha retornando o dado removido para a função que o requisitou.

Usualmente, os elementos da pilha crescem dos endereços de memória mais altos e caminham para os endereços mais baixos, permitindo maior flexibilidade do uso da memória em uma abordagem *von Newmann*.

A ideia de alocar a pilha na memória é uma solução de *software*. É uma abordagem boa, mas não é a única. Implementações de *hardware* também são possíveis, e possuem a vantagem de que são mais rápidas do que as implementadas em *software*. Em máquinas que o uso de pilha é bastante requerido, esse tipo de implementação é vital para o bom funcionamento do sistema.

A implementação em *hardware* também usa um vetor de elementos da memória para ser armazenado, porém o topo da pilha tem um ponteiro apontando para ele e esse ponteiro é um registrador, que é incrementado e decrementado conforme operações são executadas. É possível também alocar a pilha toda em registradores, usando um cadeia de registradores de deslocamento, com apenas o registrador do topo sendo acessível.

3.4.3 A Importância das Máquinas de Pilha

Em (1), alguns pontos são destacados para expressar a essencialidade da máquinas de pilha. O primeiro ponto é que qualquer máquina com suporte em *hardware* pra estruturas de pilha irá provavelmente executar aplicações que requerem pilhas de forma mais eficiente do que outras máquinas. Além disso, é mais fácil escrever compiladores para máquinas de pilha, visto que elas têm poucos casos de exceção para complicar o compilador. Por fim, máquinas de pilha são muito mais eficientes para rodar alguns programas do que máquinas baseadas em registrador, principalmente programas bem modularizados.

3.4.4 Utilização de Pilhas em Computadores

Pilhas tanto de *hardware* quanto de *software* conseguem executar 4 das principais requisições computacionais: execução de operações lógico-matemáticas, armazenamento de endereços de retorno de sub-rotina, alocação dinâmica de variáveis e passagem de parâmetros para sub-rotinas. Com isso, alguns tipos de pilha podem ser implementados.

- **Pilha de operações lógico-matemáticas:** Quando o compilador interpreta uma operação aritmética, ele precisa armazenar de alguma forma resultados intermediários da operação, e ele faz isso na pilha em questão. O compilador mantém controle sobre as operações durante suas instruções, e o *hardware* usa uma única pilha para armazenar resultados intermediários.

- **Pilha de retorno:** Quando uma sub-rotina é chamada, o endereço para o qual o *PC* do programa deve retornar precisa ser armazenado, e isso é feito na pilha de retorno. Isso garante que todas as instruções serão concluídas em ordem reversa das chamadas, como esperado.

- **Pilha de variáveis locais:** Essa pilha é utilizada quando uma sub-rotina é chamada e serve para armazenar as variáveis da sub-rotina que será executada. A cada chamada de sub-rotina, um bloco da memória é alocado. Mesmo que todos os registradores estejam sendo usados para armazenar valores temporários, a pilha de variáveis locais é necessária para economizar o uso de registradores e para conseguir retornar os valores necessários de cada sub-rotina antes que estes sejam destruídos. essa pilha permite recursão, reentrada de programas e ainda consegue economizar espaço de memória.

- **Pilha de parâmetros:** A pilha de parâmetros é facilmente implementada simplesmente copiando os parâmetros da sub-rotina para ela antes de efetivamente fazer a chamada da sub-rotina. Essa pilha permite recursão e reentrada de programas.

- **Pilhas combinadas:** Nas máquinas reais, dificilmente você encontra essas pilhas de forma pura. É bastante comum em arquiteturas baseadas em registrador ter uma pilha que é de variáveis, de parâmetros e de retorno.

Os resultados construídos com o uso dessas pilhas são bastante significativos, pois possibilitam alto desempenho sem *pipeline*, lógica de processador bastante simples, baixa complexidade de sistema, programas pequenos, rápida execução dos algoritmos em qualquer escala de tempo e ainda viabilizam um custo baixo para trocas de contexto.

3.4.5 Taxonomia das Arquiteturas de Pilha

Com a definição de (4), **taxonomia** é o estudo científico responsável por determinar a classificação sistemática de diferentes coisas em categorias. Dessa maneira, quando falamos em taxonomia das arquiteturas de pilha estamos tentando estabelecer parâmetros bons de arquitetura para classificar os vários tipos de arquiteturas de pilha. Uma boa taxonomia deve ser capaz de permitir uma análise global da arquitetura sem requerer um estudo aprofundado sobre sua implementação. Uma boa taxonomia permite que seja possível

localizar um arquitetura com relação a outros *designs* já implementados. A taxonomia na figura 3 é sugerida em (1).

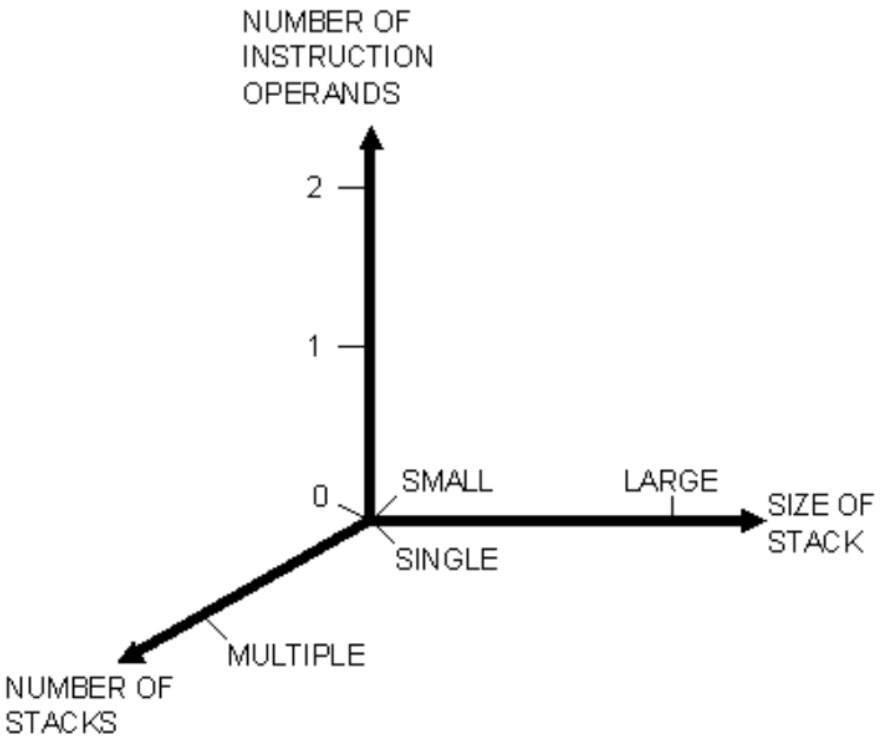


Figura 3 – Espaço de três eixos do *design* de pilhas. Fonte: (1)

3.4.5.1 Quantidade de Pilhas

Computadores podem ter uma ou várias pilhas. Computadores de apenas uma pilha são aqueles que possem exatamente uma pilha suportada pelo conjunto de instruções. Essa pilha é utilizada para guardar retornos da função, estado de sub-rotinas e até para realizar operações matemáticas. É simples em termos de *hardware*, além de facilitar o controle do sistema operacional. Por outro lado, como dados e endereços de retorno ficam aninhados, o custo operacional de técnicas de *design* de *software* modular faz com que elementos de listas de parâmetros sejam propagados por várias camadas da pilha, sendo copiado cada vez que chama a sub-rotina.

Computadores de múltiplas pilhas possuem pelo menos duas pilhas suportadas pelos conjunto de instruções. Uma normalmente é utilizada para armazenar endereços de retorno de funções, enquanto a outra é para operações matemáticas e passagem de parâmetros. Esse tipo de máquina de pilha autoriza controle separado das informações nas pilhas. Computadores de múltiplas pilhas têm a vantagem da velocidade, pois conseguem acessar vários valores em um único ciclo de *clock*. Porém, o controle sobre essas pilhas é mais difícil do que o de pilha única.

3.4.5.2 Quantidade de Elementos no *Buffer*

O tamanho da pilha em *buffer* é também um ponto muito importante a ser discutido. Quando temos um *buffer* pequeno de pilha, a pilha é vista como um pedaço reservado da memória de uso geral de programa. A pilha usa o mesmo subsistema de memória que as instruções e que as variáveis, permitindo que instruções de acesso à memória possam acessar elementos da pilha se desejado. Esse tipo de abordagem permite uma rápida troca entre pilhas que estejam executando tarefas distintas, desde que os elementos estejam na memória principal a maior parte do tempo.

Buffer de pilha maior consiste em uma arquitetura que tem um *buffer* extenso o suficiente para que não precise acessar a memória para buscar elementos da pilha. Ele pode ser feito como um conjunto de registradores usando um controle como o do processador RISC I, pode usar uma memória separada e isolada da memória de programa como também pode ser uma cache no processador. Tem a vantagem de que ciclos de programa são economizados enquanto acessa elementos e endereços de retorno de sub-rotinas. Isso aumenta bastante a velocidade dos programas, principalmente em ambientes de uso intensivo de sub-rotinas. Por outro lado, essas pilhas podem não ser suficientemente extensas para todas as aplicações, e tratar esse problema de *overflow* não é trivial.

3.4.5.3 Operandos de Zero Argumentos

O número de operandos de uma instrução é relacionado aos possíveis modos de endereçamento que podemos utilizar para construir os programas. Quando trabalhamos com instruções de zero operandos, estamos trabalhando com instruções com as quais nenhum operando pode ser associado ao *opcode* utilizado. Isto é, os elementos utilizados sempre estão no topo da pilha, de forma que os operandos são passados de forma implícita, como explicado na seção 3.2.1 sobre modos de endereçamento. Esse tipo de endereçamento é chamado de endereçamento de pilha puro. Vale destacar que instruções de leitura e escrita na memória, bem como instruções de inclusão de imediatos na pilha necessitam de pelo menos um operando sendo passado, e estas instruções são usualmente feitas de maneira simples.

Instruções de Zero operandos têm a vantagem de que apenas os elementos do topo da pilha e o seguinte podem ser referenciados, o que simplifica a construção da memória da pilha permitindo o acesso de uma pequena parte da memória, que tenha o primeiro e o segundo elementos da pilha. Há ganho de velocidade, pois não há a necessidade de um ciclo para busca dos registradores onde estão os operandos, pois de antemão já o sabemos, o que elimina a necessidade de *pipeline* para buscar e armazenar operandos na memória. Além disso, as instruções podem ser bastante compactas, com operandos de 8-*bits* sendo suficientes em termos de quantidade de *opcodes*.

Por outro lado, instruções de zero operandos dificultam a realização de códigos que precisam de modos complexos de endereçamento, levando várias instruções para realizar o que outras máquinas poderiam fazer em um único ciclo. Ainda, é muito custoso acessar elementos muito abaixo na pilha, sendo necessário tirar todos os elementos que estão acima deste (que entraram depois na pilha) para acessar o elemento desejado.

3.4.6 Exceções em Máquinas de Pilha

Na seção 3.4.3, comentamos que seria mais fácil construir compiladores para máquinas de pilha, visto que poucos são os casos de exceção em uma máquina desse tipo. De fato, duas serão as principais exceções com as quais teremos que lidar: **estouro positivo e negativo da pilha** (*stack overflow* e *stack underflow*, respectivamente) e **Interrupções de serviços de entrada e saída (E/S)**.

A exceção de estouro positivo da pilha ocorre quando a capacidade de espaço de *hardware* da memória é excedido pela aplicação executada, isto é, quando tentamos colocar mais elementos do que a pilha suporta. A exceção de estouro negativo, por sua vez, ocorre quando o limite inferior da pilha é ultrapassado, isto é, quando se tenta tirar mais elementos do que tem na pilha. Temos algumas formas de lidarmos com essas exceções, e vamos citá-las abaixo, deixando uma discussão mais aprofundada sobre o assunto quanto coerente com o tratamento de exceções no decorrer do trabalho.

- Podemos ignorar o estouro e deixar o *software* colapsar.
- Podemos parar o programa e enviar um erro de execução fatal.
- Podemos copiar uma parte da pilha para a memória de programa e permitir que o programa continue a ser executado.

Quanto às exceções de entrada e saída, as máquinas de pilha as tratam como chamadas de sub-rotinas. Isso é bastante interessante, pois as interrupções passam a ser muito menos custosas do que em outras máquinas mais convencionais, pois os registradores não precisam ser salvos, já que a pilha aloca memória automaticamente; não existem sinais (*flags*) condicionais a serem salvas, pois estas só ocorrem em operações de desvio e são salvas na pilha; muitas arquiteturas de pilha ou possuem uma pequena ou nem possuem *pipeline*, o que implica que não há penalidade quando a interrupção é executada.

3.4.7 Linguagem de Programação *Forth*

Forth é uma linguagem de programação orientada à pilha que é bastante eficiente e flexível em termos de interação *software*/*hardware*. Essa linguagem é bastante poderosa para lidar com uma grande variedade de tarefas.

A principal característica do *Forth* é que, diferentemente de outras linguagens, ele trabalha com um uso intenso de chamadas de sub-rotinas. Além disso, como uma linguagem de baixo nível e orientada a pilhas, suas primitivas são bastante interessantes para elaborar um conjunto de instruções de máquinas de pilha, tanto é que podemos citar algumas máquinas históricas que tinham em seu ISA instruções semelhantes ao *Forth*, como a *QFORTH* e a *The Forth Engine*, ambas de múltiplas pilhas, *buffer* grande e de zero operandos. Dissemos anteriormente que para instruções de zero operandos 8-*bits* seriam suficientes para suprir a necessidade de *opcodes* da arquitetura. Isso é fato, mas não podemos deixar de destacar que 8-*bits* não são suficientes para lidar com programas em geral, pois sua capacidade de endereçamento é extremamente baixa.

Dessa forma, os *designs* de 16-*bits* começaram a ganhar espaço principalmente entre máquinas *Forth*, o que fez com que fosse tradicional arquitetar máquinas desse tipo utilizando esse número de *bits*. Com 16-*bits* conseguimos endereçar $64k$ da memória, o que é suficientemente bom para uma memória de programa. Além disso, a precisão de inteiros consegue ir de -32768 até $+32767$, o que também é grande o suficiente para a maioria das aplicações mas, se não for, é possível usar precisão dupla (32-*bits*), mudando o intervalo para ir de -2147483648 até $+2147483647$.

Nesse ponto pode surgir o questionamento do porquê não então uma arquitetura de 32-*bits*. Em (1), o autor afirma que, apesar de serem interessantes, arquiteturas dessa magnitude são um exagero para a maioria das aplicações de arquiteturas de pilha, como por exemplo em sistemas embarcados.

3.5 Arquitetura do *NOVIX NC4016*

Das arquiteturas de 16-*bits* discutidas em (1), a **NOVIX NC4016** chamou a atenção como uma interessante inspiração para o trabalho em questão. Dessa forma, vamos discutir suas caraterísticas, seu diagrama de blocos e seus tipos e forma de instrução. O fluxo seguido acompanha o capítulo 4, seção 4.4 *Architecture of the NOVIX NC4016* em (1).

3.5.1 Características Principais

Formalmente, o *Novix NC4016* é conhecido por *NC4000*. É uma arquitetura de 16-*bits* que executa as primitivas da linguagem *Forth*. Essa foi historicamente a primeira máquina *Forth* de *chip* único a ser construída, e inspirou vários outros *designs* (como este que será realizado). Ela é voltada para aplicações de tempo real e com alta velocidade de execução das primitivas *Forth* para programação de propósito geral. É uma arquitetura que possui *off-chips* dedicados para a Pilha de Dados e para a Pilha de Retorno. Como

três grupos separados de *pins* conectam as duas pilhas e o barramento de dados *RAM*, a arquitetura é capaz de executar a maioria das instruções em um único ciclo de *clock*.

3.5.2 Diagrama de Blocos - Caminho de Dados

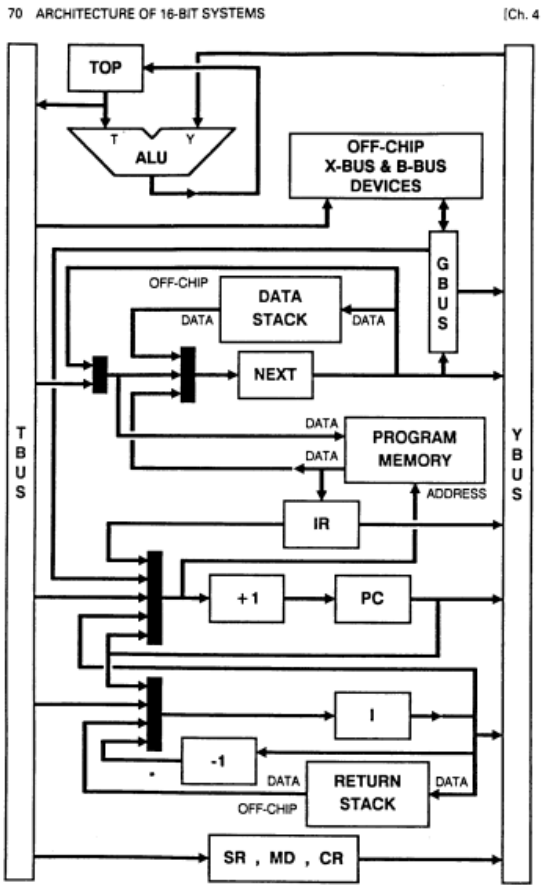


Figura 4 – Diagrama de blocos NC4016. Fonte: (1)

A pilha de dados armazena até 256 elementos, com o topo sendo um endereço dentro do *chip*, que aponta para um endereço *off-chip*. Como tem um barramento de dados de 16-*bits* separado, a pilha de dados pode ser escrita e lida em paralelo. Além disso, o topo (**T**) e o segundo elemento da pilha (**N**) estão em *buffers*, caracterizando esta arquitetura com de *buffer* pequeno.

A pilha de retorno é bem parecida com a pilha de dados, com exceção ao fato de que apenas o topo (**INDEX**) está em um *buffer*.

O contador do programa (**PC**) aponta para a localização da próxima instrução a ser executada na memória de programa externa. Como esperado, instruções de desvio (*jump*), laço (*loop*) e de chamadas de sub-rotinas (*call*) podem alterar o valor de PC.

O *NC4016* tem dois barramentos de E/S. A porta **B** é um barramento de 16-*bits* e

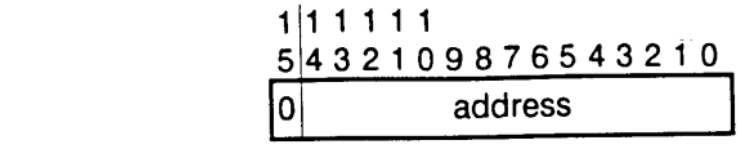
a porta **X** é de 5-*bits*. Estas portas permitem acesso direto os dispositivos de entrada e saída para controlar a aplicação sem roubar banda do barramento de memória.

A arquitetura possui 4 barramentos de 16-*bits* separados para lidar com transferências de dados a todo ciclo de *clock*, o que gera uma alta performance. Os barramentos são: barramento de memória de programa, barramento da pilha de dados, barramento da pilha de retorno e barramento de E/S.

3.5.3 Tipos e Formatos de Instruções

A arquitetura do *NC4016* possui 5 tipos de instruções: instruções de chamada de sub-rotinas, instruções de desvio, instruções lógico-aritméticas, instruções de acesso à memória, outras instruções.

3.5.3.1 Instruções de Chamada de Sub-rotinas

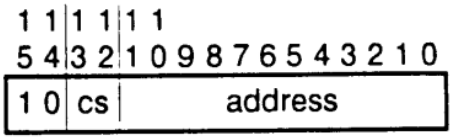


Bits	Function
15	Constant value of 0
0-14	Subroutine address

Figura 5 – Instruções de chamada de sub-rotinas

Estas instruções são identificadas pelo *bit* 0 na primeira posição com o resto da instrução indicando o endereço absoluto para onde a sub-rotina deve ser desviada.

3.5.3.2 Instruções de Desvio

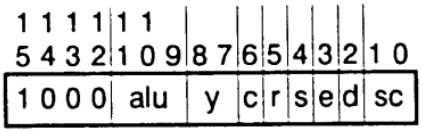


Bits	Function
14-15	Constant 10 specifies conditional instruction
12-13	Condition select 01 Branch if T = 0 10 Unconditional branch 11 Loop on INDEX register
0-11	Low 12 bits of branch address

Figura 6 – Instruções de desvio

Estas instruções são identificadas pela constante 10 em seu início. os dois *bits* seguintes indicam: **01** para desvio quando o topo da pilha for igual a zero, **10** faz um desvio incondicional, **11** faz um laço com o registrador INDEX. Os *bits* seguintes indicam o valor que deve ser somado ao PC para fazer o desvio. É um modo de endereçamento relativo ao PC.

3.5.3.3 Instruções Lógico-Aritméticas

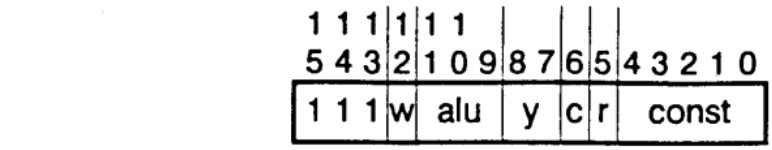


Bits	Function	
12-15	Constant 1000 specifies ALU operation	
9-11	ALU function select	
	000 T	100 T + Y
	001 T and Y	101 T xor Y
	010 T - Y	110 Y - T
	011 T or Y	111 Y
7-8	Y input control for ALU	
	00 N	10 MD reg
	01 N with carry bit	11 SR reg
6	Copy T to N	
5	Subroutine return	
4	Stack active	
3	32 Bit shift enable	
2	Divide step enable	
0-1	Shift control	
	00 No shift	10 shift left
	01 logical shift right	11 arithmetic shift right

Figura 7 – Instruções de desvio

Estas instruções são identificadas pela constante 1000 em seu começo. Os três *bits* seguintes indicam qual operação da ULA será executada. Os dois seguintes indicam qual é o segundo parâmetro da operação que deve ser tomado, se é o próximo elemento da pilha, se é o próximo elemento da pilha com *bit-carry*, se é o registrador de multiplicação-divisão (**MD**) ou se é o registrador de raiz quadrada (**SR**). Em seguida define-se se o valor do topo da pilha deve ser copiado para o registrador do segundo elemento da pilha. Define-se então se é um retorno de sub-rotina, qual a pilha utilizada, se é para tratar o dado com precisão dupla, se é uma etapa de divisão e, finalmente, se algum deslocamento deve ser realizado no dado.

3.5.3.4 Instruções de Acesso à memória

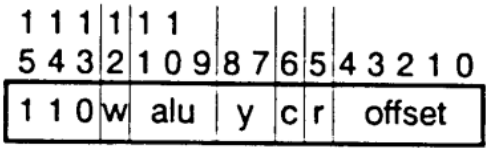


Bits	Function	
13-15	Constant 111 specifies memory reference	
12	Read/write control	
	0 Read	1 Write
9-11	ALU function select	
	000 T	100 T + Y
	001 T and Y	101 T xor Y
	010 T - Y	110 Y - T
	011 T or Y	111 Y
7-8	Y input control for ALU	
	00 N	10 MD reg
	01 N with carry bit	11 SR reg
6	Copy T to N	
5	Subroutine return	
0-4	Auto-increment/decrement constant	

Figura 8 – Instruções de acesso à memória

Estas instruções são identificadas pela constante 111. O *bit* seguinte indica se é para ler (0) ou escrever (1) na memória. Os três *bits* a seguir indicam uma instrução da ULA, seguindo bastante o tipo de instrução anterior, com exceção ao fato de que não executa passos de divisão ou precisão dupla ou mesmo de deslocamento, substituindo isso para uma constante de auto-incremento/decremento do endereço, o que é interessante para manipular a memória.

3.5.3.5 Outras Instruções



Bits	Function	
13-15	Constant 110 specifies instruction format	
12	Read/write control	
	0 Read	1 Write
9-11	ALU function select	
	000 T	100 T + Y
	001 T and Y	101 T xor Y
	010 T - Y	110 Y - T
	011 T or Y	111 Y
7-8	Y input control for ALU	
	00 N	10 MD reg
	01 N with carry bit	11 SR reg
6	Copy T to N	
5	Subroutine return	
0-4	Offset within user space/short literal	

Figura 9 – Outras Instruções

Estas instruções são identificadas pela constante 110. Podem ser usadas para ler um espaço do usuário de 32 palavras que estão nas primeiras 32 posições da memória de programa, o que economiza tempo de colocar o endereço de memória na pilha antes de executar o desvio. Pode ainda ser usado para entrada e saída de dados.

4 Desenvolvimento

4.1 Definição da Arquitetura Base

A arquitetura desenvolvida consiste em uma arquitetura de pilha, que será de **zero operandos**, com **múltiplas pilhas**, de ***buffer* pequeno**, **16-*bits*** e será uma **Máquina *Forth*** multiciclo.

Como se pode acompanhar na fundamentação teórica desse relatório, cada uma dessas decisões merece comentário. Arquiteturas de pilha são arquiteturas bastante eficientes para se trabalhar com contextos que podem ser resolvidos com pilha, como contextos aritméticos. São ainda mais eficientes para resolver problemas bem modularizados. Além disso, o compilador é bem mais simples em relação a máquinas de registradores, pois têm poucas exceções.

A decisão de ser uma arquitetura de **zero operandos** é inerente ao fato de termos optado por uma arquitetura de pilha. Instruções de zero operandos têm por operandos implícitos o topo e o próximo elemento da pilha, sendo esta uma implementação de pilha pura. Como veremos na determinação do formato das instruções na seção conseguinte, que é baseada na arquitetura do *NOVIX NC4016*, a arquitetura não será estritamente pura, podendo trabalhar com alguns registradores como segundo parâmetro que não o segundo elemento da pilha, mas isso será melhor explicado na seção adequada.

A decisão de se trabalhar com uma arquitetura de **múltiplas pilhas** se deve ao fato de se ter priorizado uma arquitetura que seja capaz de controlar separadamente - e porque não paralelamente - informações em pilhas distintas, sem aninhar informações, o que dificultaria o acesso a elas. Dessa maneira, teremos duas pilhas: **Pilha de Dados** e **Pilha de Retorno**.

Trabalhar com um ***buffer* pequeno** é uma decisão que priorizou rápida troca de contexto e minimização do uso de registradores frente um controle dificultado de um *buffer* maior. Na arquitetura desenvolvida, teremos três registradores de elementos da pilha: **TOP**, **NEXT** e **INDEX**. NEXT é, *a priori*, o segundo elemento da pilha de dados e INDEX é o primeiro elemento da pilha de retorno. TOP é o elemento topo da pilha que está sendo utilizada, isto é, se a pilha for a de dados, TOP é o primeiro elemento da pilha de dados e NEXT é de fato o segundo elemento. Caso contrário, TOP passa a ser o topo da pilha de retorno, INDEX passa a ser o segundo elemento da pilha de retorno e NEXT passa a ser o topo da pilha de dados.

Como destacamos na seção 3.4.7, um ***design* de 16-*bits*** é clássica dentre máquinas de pilha *Forth*, além de ter boa precisão para a maioria dos propósitos e de ter um bom

espaço de acesso à memória.

A ideia de utilizar uma **Máquina *Forth*** recai no fato de sua linguagem ser extremamente versátil, simples e próxima às primitivas necessárias ao conjunto de instruções. Ela é bastante boa para realizar uma grande variedade de tarefas, além de ser bastante flexível em termos de interação *software/hardware*. Como já dissemos, é uma linguagem voltada a execução rápida de problemas com várias chamadas de sub-rotinas, sendo muito bom para trabalhar com pilhas.

Como é uma máquina *Forth*, a maioria das instruções será capaz de ser executada em apenas um ciclo de *clock*. Porém, instruções como as de leitura e escrita na memória, bem como instruções de multiplicação, divisão e raiz quadrada precisarão de mais de um ciclo para executar.

4.1.1 Modos de endereçamento

Como estamos trabalhando com uma arquitetura de pilha, é natural que o endereçamento por pilha seja utilizado. Além dele, teremos endereçamento por deslocamento, mais especificamente endereçamento relativo ao PC para a maioria das instruções de desvio (chamadas de sub-rotinas, desvios condicionais e incondicionais), endereçamento por indexação para as instruções de laço e endereçamento por registrador-base para chamadas de sub-rotinas mais distantes.

Comparando-se a arquitetura do *NOVIX NC4016*, se poderá notar que na arquitetura desenvolvida a instrução de chamada de sub-rotinas com 15 *bits* para a passagem de endereço foi substituída por uma passagem de imediato. Para contornar esse fato, as instruções de chamada de sub-rotina passaram a ser possíveis por meio das instruções de desvio, adicionando um *bit* que possibilita endereçamento por registrador, permitindo acesso de endereços mais distantes em relação ao PC do que $2k$. Vale destacar que essa medida foi tomada como prevenção a futuras necessidades de endereçamento em projetos futuros como o desenvolvimento do Sistema Operacional. Porém, considerando fatos como o Princípio da Localidade, isto é, que se uma instrução é executada, a probabilidade de suas instruções vizinhas serem também executadas é bem grande, a chance de precisarmos desse tipo de endereçamento, pelo menos nesse primeiro projeto é bem pequena.

4.2 Conjunto de Instruções

A determinação do conjunto de instruções é fortemente inspirada na arquitetura do *NOVIX NC4016*. De mesma forma, teremos instruções de tamanho fixo de 16 bits, contando com 5 tipos de instruções. As instruções serão ainda micro-programadas como nas arquiteturas *CISC*, de modo que uma só instrução é capaz de passar várias informações de controle e de executar várias tarefas simultaneamente. Nas duas próximas subseções,

explicitaremos primeiro os formatos de cada instrução, e depois as instruções criadas com estes formatos.

4.2.1 Formato das Instruções

4.2.1.1 Instruções de Passagem de Imediato

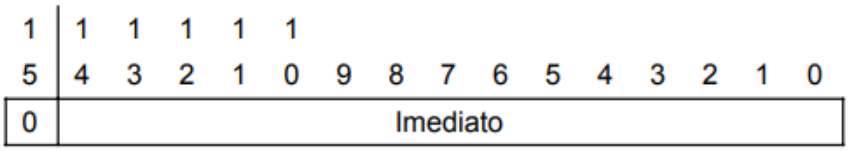


Figura 10 – Instruções de passagem de imediato

[15] Valor constante 0 indica instrução de passagem de imediato.

[0:14] Valor de 15 *bits* do imediato.

Este formato de instrução para passagem de imediatos permite a passagem de valores em complemento de 2, sendo então o *bit* 14 um *bit sinal* e ou outros *bits* relativos ao número. Dessa forma, podemos passar imediatos de -16384 até $+16383$, o que é uma boa quantidade de valores para a maioria dos procedimentos.

4.2.1.2 Instruções de Desvio

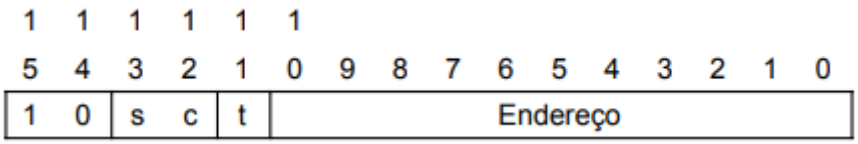


Figura 11 – Instruções de desvio

[14:15] Valor constante 10 especifica uma instrução condicional.

[13:14] Seletor condicional:

01 Desvia se $T = 0$.

10 Desvio incondicional.

11 *Loop* no registrador INDEX.

[11] Tipo de endereçamento:

0 Endereçamento por desvio relativo ao PC.

1 Endereçamento relativo a registrador.

[0:11] Últimos 11 *bits* do endereço de desvio.

Como havíamos destacado nos modos de endereçamento, as instruções de chamadas de sub-rotinas poderão naturalmente ser executadas aqui com a inserção do *bit* de tipo de endereçamento. Conseguimos fazer instruções de desvio condicional e incondicional aqui também.

4.2.1.3 Instruções da Unidade Lógico-Aritmética

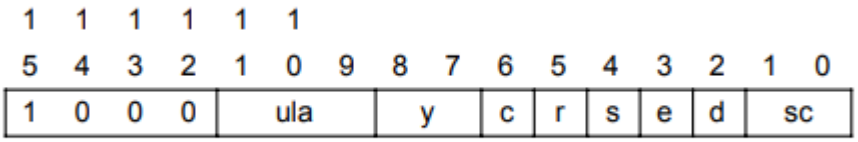


Figura 12 – Instruções Lógico-Aritméticas

[12:15] Valor constante 1000 especifica uma instrução lógico-aritmética.

[9:11] Seletor da função da ULA:

000 T.

001 $TandY$.

010 $T - Y$.

011 $TorY$.

100 $T + Y$.

101 $TxorY$.

110 $Y - T$.

111 Y.

[7:8] Seletor do Y:

00 N.

01 N com *bit carry*.

10 registrador MD.

01 registrador SR.

[6] Copiar T para N.

[5] Retorno de sub-rotina.

[4] Pilha ativa.

[3] Ativador para dados de 32 *bits*.

[2] Ativador de divisão.

[0:1] Seletor de *shift*:

00 Sem *shift*.

01 *Shift* lógico para a direita.

10 *Shift* para a esquerda.

01 *Shift* aritmético para a direita.

É possível decodificar 8 operações lógico-aritméticas diferentes na ULA, como podemos ver na descrição dos *bits* de 9 a 11. As operações possíveis são as mais comuns, sendo suficiente trabalhar só com elas.

Quando discutimos no início desse capítulo do relatório sobre o fato de esta não ser uma arquitetura puramente de pilha, estávamos nos referindo principalmente aos *bits* 7 e 8 desse formato de instrução. Quando mantemos em 00, o comportamento que ocorre é o natural para pilhas, executando tarefas sobre o topo e o elemento seguinte da pilha. Porém, o *NC4016* trás a ideia de manter 3 registradores que viabilizar multiplicação, divisão, raiz quadrada e somas com *carry*, trabalhando com esses registradores seguindo a ideia de acumuladores. Essa abordagem permite maior flexibilidade em vários sentidos, mas o principal deles é que conseguimos deixar essas operações multiciclo, não precisando aumentar o tempo do ciclo para instruções mais simples, justificando o fato de essa abordagem ter sido mantida aqui.

O *bit* 6 tem um comportamento bastante peculiar, porém muito versátil. Quando ativo, isto é, quando seu valor é 1, ao executar a instrução, o valor que estava armazenado em T é copiado para o registrador apontado por Y. Isto permite que mantenhamos um valor de referência, por exemplo, sem ter que executar operações de $SWAP$ (troca) repetidamente, economizando vários ciclos de *clock*.

O *bit* 5 indica se a operação é um retorno de sub-rotina. É um *bit* de controle que atua como *flag*, economizando também ciclos de *clock* para fazer retornos de função, pois podemos fazer isso em paralelo com instruções aritméticas.

O *bit* 4 indica qual é a pilha com a qual estamos trabalhando. Quando seu valor é 0, estamos na pilha de dados; caso contrário, estamos na pilha de retorno. Isso permite que, por exemplo, façamos uma conta e a coloquemos e T, troquemos de pilha e salvemos o resultado na pilha de retorno, usando o *bit* 6 ativo.

O *bit* 3 indica se o dado armazenado têm precisão dupla. Quando ativo, a conta deve considerar os próximos dos elementos da pilha como apenas um operando.

O *bit* 2 indica se está executando uma etapa da divisão. Ele serve de controle para a operação em questão.

Os *bits* 0 e 1 indicam qual deslocamento o dado deve sofrer, podendo não sofrer nenhum, ou deslocamentos lógicos ou aritméticos.

4.2.1.4 Instruções de Referência à Memória

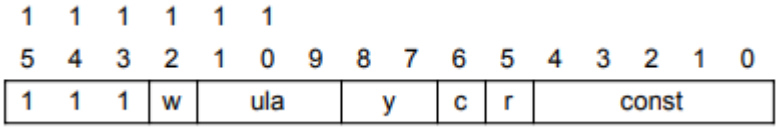


Figura 13 – Instruções de referência à memória

[13:15] Valor constante 111 especifica uma instrução de acesso à memória.

[12] Seletor de leitura-escrita:

0 Leitura.

1 Escrita.

[9:11] Seletor da função da ULA:

000 T.

001 $TandY$.

010 $T - Y$.

011 $TorY$.

100 $T + Y$.

101 $TxorY$.

110 $Y - T$.

111 Y.

[7:8] Seletor do Y:

00 N.

01 N com *bit carry*.

10 registrador MD.

01 registrador SR.

[6] Copiar T para N.

[5] Retorno de sub-rotina.

[0:4] Constante de incremento/decremento.

Podemos observar que estas instruções são muito próximas em formato às instruções lógico-aritméticas. A ideia é que possamos trabalhar com endereços da mesma forma que trabalhamos com dados, sempre visando a flexibilidade da arquitetura.

Estas instruções levam 2 ciclos de *clock*, um para decodificação, outro para a operação em si, com o endereço sempre sendo tomado do topo da pilha.

4.2.1.5 Instruções Gerais

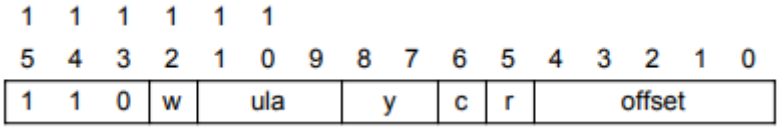


Figura 14 – Instruções gerais

[13:15] Valor constante 110 especifica uma instrução do tipo geral.

[12] Seletor de leitura-escrita:

0 Leitura.

1 Escrita.

[9:11] Seletor da função da ULA:

000 T.

001 $TandY$.

010 $T - Y$.

011 $TorY$.

100 $T + Y$.

101 $TxorY$.

110 $Y - T$.

111 Y.

[7:8] Seletor do Y:

00 N.

01 N com *bit carry*.

10 registrador MD.

01 registrador SR.

[**6**] Copiar T para N.

[**5**] Retorno de sub-rotina.

[**0:4**] *Offset* para instruções.

São instruções que não se encaixam nos tipos anteriores. Nesse primeiro projeto, só a utilizaremos para entrada e saída de dados, pelo menos por enquanto.

4.2.2 Conjunto de Instruções

Com os tipos de instruções estabelecidos, apresentamos abaixo o conjunto de instruções criadas, divididas pelo o que executam, e não pelo tipo de instrução necessariamente.

4.2.2.1 Instruções de Passagem de Imediato

Nome	Formato	Descrição
Instruções com Imediato		
PUSH	**0**XXX_XXXX_XXXX_XXXX	Coloca um imediato no topo da pilha

Figura 15 – Instruções com Imediato

Esta instrução é do tipo de passagem de imediato. Como esperado, com a constante 0 no início da instrução, o resto da instrução é interpretada como imediato a ser colocado direto na pilha de dados.

4.2.2.2 Instruções de Desvio

4.2.2.3 Instruções da Unidade Lógico-Aritmética

Instruções da Unidade Lógico-Aritmética		
AND	**1000_001**Y_YCRS_EDSS	Faz *bit-a-bit* a operação lógica AND
OR	**1000_011**Y_YCRS_EDSS	Faz *bit-a-bit* a operação lógica OR
XOR	**1000_101**Y_YCRS_EDSS	Faz *bit-a-bit* a operação lógica XOR
+	**1000_100**Y_YCRS_EDSS	Faz a operação de soma dos operandos
-	**1000_010**Y_YCRS_EDSS	Faz a operação T - Y
--	**1000_110**Y_YCRS_EDSS	Faz a operação Y - T
SHIFT LR	**1000**_UUUY_YCRS_ED**01**	Faz a operação de *shift* lógico para direita
SHIFT AL	**1000**_UUUY_YCRS_ED**10**	Faz a operação de *shift* aritimético para esquerda
SHIFT AR	**1000**_UUUY_YCRS_ED**11**	Faz a operação de shift aritmético para direita
/	**1000_0001_0**CRS_E**1**SS	Faz a operação de divisão usando T e o reg MD
*	**1000_0001_0**CRS_E**0**SS	Faz a operação de multiplicação usando T e o reg MD
SQRT	**1000_0001_1**CRS_EDSS	Faz a operação de raiz quadrada inteira (piso) de T com o reg SR

Figura 16 – Instruções da unidade lógico-aritmética

As 6 primeiras instruções descritas alteram apenas a função escolhida da ULA. As outras duas possibilidades de instrução da ULA são aquelas que retornam ou T ou N, que não consideramos como instrução de forma isolada.

As instruções de *shift* são decodificadas alterando os últimos *bits* de instruções do tipo lógico-aritméticas.

As instruções de multiplicação e divisão são escritas utilizando o registrador MD como Y, e se é multiplicação o *bit* de *division step* é 0, e se é divisão o *bit* em questão é 1.

De maneira análoga, quando o Y referenciado for o de raiz quadrada, a operação será executada.

4.2.2.4 Instruções de Referência à Memória

4.2.2.5 Instruções Gerais

4.3 Caminho de Dados - *Datapath*

5 Considerações Finais

O presente projeto teve por proposta desenvolver um circuito lógico programável de um contador crescente e decrescente no *software Quartus Prime* que no *display* de 7 segmentos do *kit* de FPGA apresente a contagem, utilizando a linguagem de descrição de *hardware Verilog* para isso. De acordo com a sequência sorteada e o tempo de *clock*, deveria-se elaborar uma máquina de estados apropriada, e todos os módulos mais que fossem necessários.

Como descrito nos objetivos específicos desse relatório, o primeiro passo foi determinar os estados da máquina e elaborar o diagrama de estados. Como mostrado na figura **??**, precisamos de 10 estados para a máquina elabora, 9 estados para os números da sequência e 1 para o estado em que a tela deveria ficar limpa (*blank*). Os estados foram rotulados com números binários de 4 bits, em ordem crescente, iniciando-se do 0000, indo até 1000, e rotulando o estado de *blank* com 1111.

Com o diagrama pronto, a tabela de próximo estado só precisava ser digitada, pois todo o trabalho de entender as mudanças já fora realizado no diagrama de estados.

O próximo passo do projeto foi o de começar a codificar a máquina de estados desejada em *Verilog*. Para isso, aproveitou-se a tabela de próximo estado elaborada para fazer o módulo de mudanças de estado. Pela escolha de rotular os estados com números binários crescentes, a implementação dessa parte do projeto deveria se fazer bastante simples, pois bastaria um bloco **case** com o parâmetro das chaves passadas *up,down* para determinar o próximo estado, pois bastaria incrementar/decrementar o valor do estado, atentando para se era o último/primeiro estado. Deveria ser simples, mas encontrou-se bastante dificuldade para conseguir compilar essa parte do projeto, visto que se estava tentando alterar no bloco **always** o valor dos parâmetros. Até se perceber da necessidade das variáveis auxiliares para fazer com que o módulo **CCME** funcionasse, muito tempo fora gasto. Por fim, conseguiu-se compilar o módulo e gerar sua *waveform* que, como discutido nos resultados, mostrou a exatidão do módulo implementado.

A elaboração do circuito sequencial de memória (**memoria**), que seguiu o módulo anterior, apresentou também os problemas de compilação, que foram resolvidos com o uso de variáveis auxiliares. Além disso, vale destacar o cuidado necessário quanto a usar atribuição "não-bloqueante"para guardar os valores nos *flip-flops* e do uso de um *reset* síncrono, como esperado em uma implementação de *Moore*. Apesar das dificuldades encontradas, o módulo foi corrigido e funcionou corretamente, como mostrou a discussão de sua *waveform*.

Para a elaboração do circuito combinacional de saída, se fez necessário escrever

uma tabela de saída, relacionando o estado à sua respectiva saída. Com isso, o código do módulo **CCS** ficou bastante simples, sendo só um bloco **case** com parâmetro o estado atual. Com os problemas encontrados nos módulos anterior e suas respectivas soluções, esse módulo foi implementado sem nenhum problema, compilando logo na primeira tentativa e apresentando um resultado correto, como mostrado na discussão de sua *waveform*.

Apesar de no projeto poder ser mais instrutivo apresentar o divisor de frequência nessa parte e, por conseguinte, tê-lo elaborado após o módulo **CCS**, não foi isso que ocorreu. Esse foi o último bloco implementado. Foram encontradas várias dificuldades no entendimento de como realizar o divisor de frequência, e esse só foi implementado após a explicação em sala de como fazê-lo, e a implementação inclusive segue exatamente o código passado pelo professor. Novamente ressaltamos que não foi possível simular seu resultado dado o tempo máximo de simulação.

O módulo do decodificador de *BCD (Binary Coded Decimal)* para *display* de 7 segmentos também não apresentou problemas em sua implementação. Esse módulo também fora bastante trabalhado em sala de aula e inclusive compôs um projeto anterior do curso, de maneira que bastou copiar esse módulo desse projeto anterior para o trabalho. Não podemos deixar de destacar que, como mostrado na *waveform*, o módulo funciona corretamente.

O último módulo que objetivou-se elaborar fora o de integração do projeto. Esse foi um módulo bastante interessante de se elaborar. Trabalhando apenas com a instanciação dos módulos, a noção de encapsulamento é muito bonita, pois os estados da máquina em si passam transparentes para quem utiliza a máquina, mas são de extrema importância para a implementação interna da máquina. Além disso, termos optado por fazer esse módulo de integração ajuda a mostrar o lado de módulos de integração e não só funcionais que o *Verilog* tem capacidade de codificar.

Como primeiro projeto um pouco mais robusto não só em *Verilog* mas em qualquer *HDL*, com certeza esse projeto deve contar com ineficiências de implementação que não foram destacadas afinal não foram percebidas, visto que tais implementações são as primeiras de uma longa trajetória. Não podemos deixar de destacar a grande relevância desse projeto para a formação de Engenheiro de Computação. Muito foi aprendido sobre detalhes de *Verilog* e o conhecimento sobre máquinas de estado foi ainda mais aprofundado. Espera-se, neste ínterim, que o desempenho no laboratório de Arquitetura e Organização de Computadores seja elevado, como resultado do bom aproveitamento desse curso como um todo.

Referências

1 KOOPMAN, P. *Stack Computers the new wave*. 1th edition. ed. Pittsburgh, EUA: Mountain View Press, 1989. Citado 8 vezes nas páginas 1, 5, 8, 10, 11, 13, 16 e 17.

2 STALLINGS, W. *arquitetura e Organização de Computadores*. 10th edition. ed. São Paulo, Brasil: Pearson, 2017. Citado 5 vezes nas páginas 4, 1, 5, 6 e 8.

3 EMENTA da Unidade Curricular de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores. 2019. Disponível em: <https://www.unifesp.br/campus/sjc/images/sjc/Secretaria_de_Gradua%C3%A7%C3%A3o/UCs_Vigentes/L/Laborat%C3%B3rio_de_Sistemas_Computacionais_-_Arquitetura_e_Organiza%C3%A7%C3%A3o_de_Computadores.pdf>. Citado na página 3.

4 TAXONOMIA. 2019. Disponível em: <https://www.significados.com.br/taxonomia/>. Citado na página 12.