Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

João Henrique Luciano

Avaliação de desempenho de algoritmos implementados em FPGA via síntese de alto nível

São Paulo
Novembro de 2018

Avaliação de desempenho de algoritmos implementados em FPGA via síntese de alto nível

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman

São Paulo
Novembro de 2018

Agradecimentos

Se eu realmente fosse agradecer todas as pessoas que eu sinto que tiveram um impacto positivo real na confecção deste trabalho, esse provavelmente seria o maior capítulo dele.

Obviamente, o primeiro agradecimento é para meus pais, que aguentaram todos os bons e maus momentos do trabalho, da graduação e da vida, ouviram minhas ideias para resolver os problemas insurgentes mesmo sem entender o que estava falando, sempre me apoiaram nas minhas decisões benignas e me advertiram sobre os impactos das más decisões, também.

O segundo agradecimento vai para a pessoa que me ajudou a me motivar no meu pior momento da graduação, meu amigo germano-brasileiro Mauro Zanella. Se ele não tivesse me chamado para sair da faculdade por 1 semestre para estagiar com ele em Friedrichshafen, era capaz de que eu não conseguisse continuar a graduação e me formar.

Não menos importante, agradeço a todos os meus amigos e pessoas que passaram pela minha vida para me trazer ensinamentos importantes sobre ela, além de terem me apoiado quando mais precisei, nos picos de estresse e ansiedade. Passando pela Santa Batcaverna, pelas Miçangueiras, até chegar no CC e no Simplão, todos tiveram um papel importante, mesmo os que me causaram mal em algum momento. Não citarei nomes porque só a ideia de esquecer alguém já me incomoda bastante, mas quem me ajudou sabe o que fez, o que também torna desnecessária as citações específicas.

Por fim, agradeço também ao meu orientador Alfredo Goldman, que entre brincadeiras e convívios sociais (também conhecido como *bullying* para os menores de 18 anos), me aconselhou e ajudou muito durante a maior parte da graduação. Devo boa parte do meu bom-senso profissional e acadêmico a ele.

Não fossem por todas essas pessoas, eu não seria quem eu sou, nem estaria onde estou, ou saberia o que sei. Agradeço a todos, ainda mais o que se mantiveram por perto nas piores fases.

Resumo

O presente trabalho visa o estudo e experimentação do uso de ferramentas de síntese de alto nível para o projeto de circuitos eletrônicos. O intuito é destacar tecnologias emergentes de hardware, como os FPGAs, assim como aproximar engenheiros de software do desenvolvimento de hardware. Para tanto, foi usado o LegUp, um arcabouço de código aberto para síntese de alto nível em FPGAs. Logo, o estudo do funcionamento interno e uso dessa ferramenta foi primordial para a elaboração do projeto. Com ele, dois algoritmos foram desenvolvidos na linguagem de programação C e, dessa forma, efetuar-se a síntese de alto nível do algoritmo para um circuito descrito em Verilog e implementado em FPGA. Os resultados das experiências mostraram que os algoritmos em *hardware* foram mais eficientes em termos de quantidade de ciclos de *clock*, e potencialmente iguais no que diz respeito ao tempo total de execução dos programas.

Palavras-chave: FPGA, síntese de alto nível, hardware, algoritmos, C, Verilog.

Abstract

This work aims to study and use high-level synthesis tools for electronic circuits design. Alongside, there is emphasis in newer technologies, such as FPGAs, as well as bringing together software engineers and hardware development. To do that, LegUp, a open-source high-level synthesis framework, is used along with a system-on-a-chip FPGA. Furthermore, the internal structure and usage of the framework was essential for the project development. Using LegUp, two algorithms were implemented using C programming language to feed its high-level synthesis process and create a Verilog HDL, which was programmed into a FPGA. Further experiments with the software and hardware algorithms were able to show that the FPGA-implemented algorithms were more efficient than the software-implemented, and potentially equal considering the execution time of the programs.

Keywords: FPGA, high-level synthesis, hardware, algorithms, C, Verilog.

Sumário

Capítulo 1

Introdução

A melhoria do desempenho dos computadores sempre foi uma constante na história da Computação. Das válvulas a vácuo até os nanotransístores, por muito tempo o desenvolvimento de *hardware* foi regido pela Lei de Moore, dizendo que a cada 18 meses a quantidade de transístores em um *chip* de silício dobraria e, por consequência, seu desempenho. Atualmente, tal lei chega aos limites da Física Moderna, onde um nanotransístor de 1 nanômetro de comprimento já foi inventado[1]. Novas técnicas estão sendo estudadas e aplicadas na melhoria da fabricação de *chip*s, como a manipulação de novos tipos de partículas físicas[2] e de propriedades quânticas de partículas já existentes, como o *spin*[3].

Para impulsionar o desempenho computacional sem envolver diretamente a quantidade de transístores em um *chip* ou as propriedades das partículas envolvidas na fabricação, novas metodologias computacionais foram adotadas. Por exemplo, a paralelização do processamento de dados a partir do trabalho conjunto entre *hardwares* dedicados e de propósito geral tem sido amplamente empregada em aplicações que demandam baixa latência de resposta, como processamento gráfico[4] e gerenciamento de memória[5]. Um bom exemplo de *hardware* dedicado é a *GPU* (*Graphics Processing Unit*), criada com o intuito de paralelizar o processamento gráfico de computadores de uso geral a fim de acelerar a visualização das interfaces gráficas destes.

Apesar do desempenho alcançado com o uso de *hardwares* dedicados, é necessário um grande esforço para inventar ou otimizar dispositivos. O ciclo de desenvolvimento de um novo *chip* vai desde o planejamento do circuito até a encomenda de fabricação de amostras para testar seu funcionamento. O custo e tempo envolvidos podem ser proibitivos, na maioria das aplicações, envolvendo meses de pesquisas e milhares de reais ou dólares por ciclo. Uma das soluções encontradas recentemente para acelerar e baratear esse processo foi o investimento na evolução de *hardwares* reprogramáveis, como as PALs (*Programmable Array Logic*), até a criação dos FPGAs.

Os FPGAs são *chips* de silício reprogramáveis que podem, dentro de seus limites, recriar qualquer circuito lógico. Essa característica o faz interessante no ciclo de desenvolvimento de *hardware* devido à facilidade em se ter um *hardware* com um algoritmo personalizado, programado e de alteração relativamente fácil.

Apesar das facilidades trazidas com a evolução dos FPGAs, há uma demanda no mercado por pesquisadores e desenvolvedores de *hardware*. Devido às dificuldades citadas com a Lei de Moore, seria interessante ter uma maior quantidade de pesquisas em novas tecnologias e

[1] http://science.sciencemag.org/content/354/6308/99
[2] https://www.nature.com/articles/nnano.2017.178
[3] https://arxiv.org/abs/1212.3362
[4] https://link.springer.com/chapter/10.1007/3-540-63508-4_107
[5] https://patents.google.com/patent/US9652230

arquiteturas de circuitos para manter o crescimento de desempenho dos computadores. Por exemplo, a proporção entre a quantidade de profissionais das áreas de engenharia de *software* e *hardware* é de 10 para 1, como citado pelo Bureau de Estatística Trabalhista dos EUA (2010).

Uma solução para a situação seria adotar formas de aproximar essas duas áreas de engenharia, integrando ambos os profissionais de *software* e *hardware* em uma mesma tarefa. Essa possibilidade é dada pelo uso de ferramentas de síntese de alto nível, onde um engenheiro de *software* pode programar um algoritmo arbitrariamente complexo e traduzí-lo em uma especificação de *hardware*, que pode ser otimizada pelo engenheiro de *hardware*. A síntese de alto nível é, portanto, a criação de algoritmos em *hardware* através das descrições destes por linguagens de alto nível, tais como C.

A fim de pesquisar sobre essa solução, o presente trabalho explica sobre o que é e como funcionam FPGAs e síntese de alto nível nos capítulos 2 e 3. Foi escolhida uma ferramenta de síntese de alto nível chamada *LegUp High-Level Synthesis*, que é descrita no capítulo 4. Para testar o processo de síntese da ferramenta, foram estudados e programados dois algoritmos, cujos contextos são dados no capítulo 5. Depois de desenvolvidos os algoritmos e *hardwares* equivalentes, suas qualidades de processamento foram avaliadas a partir da contagem de ciclos de *clock* utilizados para concluir suas execuções. Os resultados das avaliações são dados no capítulo 6.

Capítulo 2

FPGA

Este capítulo descreve o que é um FPGA e do que são essencialmente compostos. As descrições aqui feitas foram sintetizadas a partir das obras de Moore e Wilson (2017) e Farooq *et al.* (2012).

2.1 Introdução

*FPGA*s (do inglês *Field Programmable Gate Array*) são dispositivos de silício que podem ser programados após sua fabricação, permitindo que quase qualquer *design* de circuito digital possa ser implementado nele. Essa maleabilidade torna-os atrativos para tarefas que envolvam a produção e alteração de *designs* de circuitos lógicos, pois o custo e tempo de execução dessas atividades são muito reduzidos em comparação ao tradicional ciclo de desenvolvimento com o uso de dispositivos *ASIC*s (do inglês *Application-Specific Integrated Circuit*).

2.2 Reprogramabilidade

A flexibilidade do *FPGA* se deve à sua capacidade de ser reprogramada após sua fabricação. Para tanto, interruptores programáveis são posicionados em suas rotas de interconexão. Como esses interruptores (também conhecidos como *switch*s) são componentes eletrônicos que redirecionam a corrente elétrica, é necessário que o interruptor seja configurado para que cada corrente de entrada seja desviada para a rota certa. Isso é feito a partir de memórias, que são programadas juntas do FPGA ao subir um programa no *chip* ou na placa que o contém.

Além das rotas de interconexão, as portas lógicas do FPGA também devem ser arranjadas de tal forma que o processamento dos dados seja feito de acordo com o especificado pelo usuário. Ao invés de criar um possível esquema de tradução da linguagem de especificação de hardware para funções booleanas, são empregadas as chamadas *LUTs* (*lookup tables*).

Para definir qual o circuito implementado no *chip*, certas tecnologias de programação de circuitos são usadas, memórias para guardar os estados dos *switchs* e de outros componentes. Idealmente, essas memórias são não-voláteis, infinitamente reprogramáveis, baratas e que consumam pouca energia. Atualmente, a tecnologia mais utilizada é a *SRAM*, que não atinge todos os critérios ideais mas suas desvantagens podem ser facilmente contornadas.

A *SRAM* (*'Static Random Access Memory'*, ou 'memória estática de acesso aleatório') é um tipo de memória volátil que utiliza uma combinação de 6 transístores para guardar um *bit* de informação. O termo 'estático' refere-se à falta de necessidade de atualizações de

memória (*memory refreshes*) em seu circuito. Note que isso não significa que, na ausência de corrente elétrica, o estado se manterá: caso haja uma queda de energia, a memória perderá o dado nela contido.

Devido à sua característica estática, as *SRAM*s são mais rápidas e consomem menos energia que as *DRAM*s. Entretanto, como as *SRAM*s necessitam de 6 transistores, ao contrário das *DRAM*s que utilizam apenas 1 transistor e 1 capacitor, aquelas são muito mais caras e demandam mais espaço no circuito para serem implementadas, além de ambas serem voláteis. Morawiec e Coussy (2008)

2.3 Componentes

A composição de um *FPGA* pode ser resumido em três componentes:

Blocos lógicos - blocos de *hardware* responsáveis pelo processamento lógico dos dados

Rotas de interconexão - fios e barramentos que transportam os sinais digitais pelo circuito

Blocos de entrada e saída - unidades responsáveis pela comunicação entre o *chip* e componentes externos

Os blocos lógicos são configuráveis e implementam funções lógicas e armazenamento de dados (i.e. memória). Os blocos de E/S recebem e enviam dados para fora do *chip*. Por fim, as rotas de interconexão conectam os blocos lógicos entre si e entre os blocos de E/S. Uma forma de visualizar esses componentes é através de uma matriz, onde os blocos lógicos estão dispostos bidimensionalmente, numa grade, e conectados pelas rotas de interconexão. Nas bordas dessa matriz se encontram os blocos de E/S, integrados à matriz pelas rotas de interconexão, servindo para a comunicação do *FPGA* com dispositivos exteriores a ele.

2.3.1 Blocos lógicos

Os blocos lógicos configuráveis (BLCs) são as unidades que provêm capacidade lógica e de armazenamento para o FPGA. BLCs podem ser implementados de diversas maneiras, desde simples transistores até processadores inteiros, e essa implementação define sua granularidade. BLCs com granularidade muito pequena, como transistores, ocupam muito espaço e os torna ineficientes em termos de área. Por outro lado, os de granularidade muito grande, como processadores, podem representar um desperdício de recurso quando tratamos de funções mais simples. Entre esses máximos, temos um espectro de implementações de BLCs.

Os BLCs são compostos por blocos lógicos básicos (BLBs), que podem ser usados em conjunto ou de forma isolada, ou seja, um BLC pode ser composto por um único BLB ou por um conjunto deles. As componentes usadas nos BLBs podem variar, mas a fabricante da placa usada no presente trabalho, a Altera, utiliza *lookup tables* e *flip-flops* para armazenamento. As LUTs são usadas como tabela de valor para representar qualquer função booleana com determinado número de *bits* de entrada e de saída. Assim, uma LUT que recebe k *bits* é chamada de *k-LUT* e representa qualquer função booleana f tal que

$$f : \{0,1\}^k \to \{0,1\}^k$$

A vantagem do uso de LUTs e *flip-flops* reside em não ter uma granularidade nem muito pequena, nem muito grande, permitindo o uso em conjunto para implementações mais complexas.

Apesar do uso de LUTs ou outros métodos para implementar BLBs, como NANDs, esses métodos são mais usados para criar a parte programável do FPGA. Uma parte dele pode vir já programada com blocos lógicos especializados, como processadores de sinais digitais (conhecidos como DSPs, *digital signals processor*), multiplicadores, somadores, ALUs inteiras, todos criados de forma otimizada para suas tarefas. Estes são chamados de blocos rígidos, pois não podem ser reprogramados, apenas usados como estão no FPGA. Isso implica em um possível desperdício de espaço e recursos no caso desses blocos não serem utilizados pelo circuito, mas também traz a vantagem de se usar blocos dedicados a determinadas tarefas.

2.3.2 Rede de interconexão

Como já mencionado, a flexibilidade de um FPGA vem da capacidade de ter sua rede de interconexão reprogramada. Essa rede precisa ser flexível não só em termos de configuração de rotas, mas também de tipos de fios presentes no dispositivo para poder implementar uma grande variedade de circuitos. Apesar da maior parte das componentes de um circuito apresentar localidade (isto é, se localizarem perto umas das outras), há conexões que podem necessitar de fios mais longos. Cerca de 85% da área de um FPGA consiste da rede de interconexão entre os blocos lógicos. Visando otimizar a comunicação de acordo com a finalidade do circuito, essa rede pode ser construída usando arquiteturas diferentes. A arquitetura utilizada neste trabalho é a baseada em malha.

As redes baseadas em malha (do inglês '*mesh-based*', também conhecida como '*island-style*') são organizadas em formato matricial, com blocos lógicos cercados de fios de conexão - por isso o termo 'estilo de ilha', onde os BLs parecem estar "ilhados"em um "mar de fios". Nas extremidades se encontram os blocos de entrada e saída. Na rede de conexão estão localizadas concentrações de *switches* que estabelecem a rota dos sinais entre os blocos lógicos. Por último, existem as conexões entre os blocos lógicos e a rede de comunicação, que são chamadas de caixas de comunicação, também configuráveis. A figura 2.1 mostra essa arquitetura. Nela, temos os BLCs em rosa, responsáveis pela implementação do processamento de dados do circuito; as caixas de *switchs* programáveis em azul, responsáveis pelo roteamento dos sinais através do circuito; e as caixas de conexão em azul, também responsáveis pelo roteamento dos sinais digitais, mas principalmente pela entrada e saída entre os blocos lógicos e o exterior do *chip*.

Um exemplo de FPGA que segue o modelo de rede baseada em malha é a Stratix IV, da Altera, com o modelo EP4SGX230 ilustrado na figura 2.2. A Stratix é uma das famílias de FPGAs industriais da Altera usadas principalmente para aplicações que demandam alto desempenho. O modelo ilustrado, por exemplo, possui 228.000 elementos lógicos (BLBs) compostos de 4-LUTs e *flip-flops* tipo D, 182.400 registradores, 91.200 BLCs, 22 blocos de memória M144K (que guardam até 144 *kilobits* endereçáveis), dentre outras características que o tornam uma boa escolha para usuários que necessitam de um bom desempenho.

2.4 Desvantagens

A maior vantagem de *FPGAs* - sua flexibilidade - também é a causa de sua maior desvantagem. Essa característica baseia-se na reprogramação das rotas de interconexão, dos blocos lógicos e dos blocos de entrada e saída. Entretanto, a área usada por tais rotas

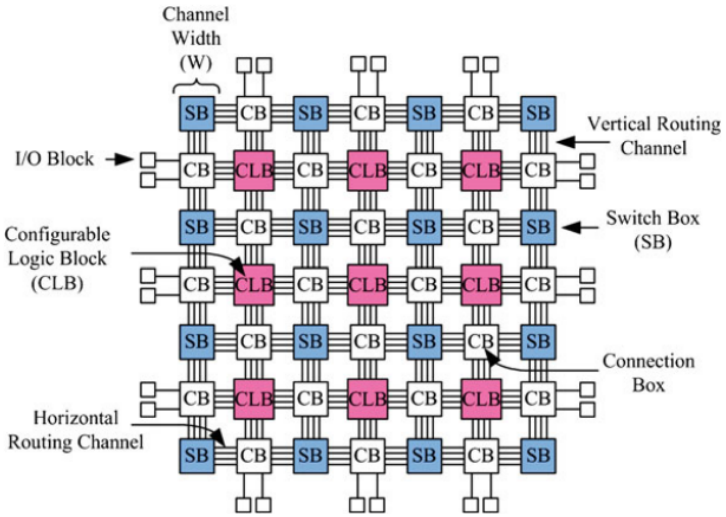

Figura 2.1: *Exemplo de um FPGA com rede de interconexão em malha. Fonte: Farooq et al. (2012)*

Figura 2.2: *SoC FPGA usando um FPGA Stratix IV EP4SGX 230, da Altera.*

ocupa a maior parte do dispositivo, chengando a quase 90% da área útil do dispositivo, para permitir a sua reprogramação. Não obstante, os *switches* e os componentes necessários para implementar as LUTs geram resistência elétrica à propagação dos pulsos de *clock* do sistema, o que obriga os fabricantes a diminuir a frequência máxima de *clock* do *chip* e da placa. Por consequência, os FPGAs são mais lentos e consomem mais energia do que os *ASIC*s.

Outra desvantagem de FPGAs é a programação. Apesar da popularidade de linguagens de descrição de *hardware* no desenvolvimento de ASICs, o uso delas na programação de FPGAs por parte de pessoas leigas da área pode ser um empecilho inicial considerável, dada a necessidade de conhecimentos básicos de circuitos eletrônicos, como sinais de comunicação, protocolos de processamento de dados, máquinas de estados etc. A utilização de linguagens de alto nível, tais como C, para especificar o comportamento do circuito pode ajudar no entendimento desses conceitos básicos de desenvolvimento de *hardware*, sabendo como os elementos e comandos da linguagem são mapeados a componentes do circuito.

Capítulo 3

Conceitos fundamentais

Este capítulo descreve alguns dos conceitos fundamentais sobre dois tópicos essenciais para o desenvolvimento e entendimento do presente trabalho: síntese de alto nível e o projeto LLVM.

3.1 Síntese de alto nível

Síntese de alto nível ("*High level synthesis*") é o processo de transformação de linguagens de programação de alto nível para sintetizar arquiteturas RTL ("*Register-transfer level*"), ou seja, sintetizar arquiteturas de circuitos digitais síncronos a partir de descrições comportamentais, algorítmicas, do *hardware*. As saídas geradas são, em sua maioria, arquivos em linguagens HDL ("*Hardware description language*") usados para configurar o *hardware* através de ferramentas próprias de compilação. As ferramentas de síntese de alto nível realizam o mesmo fluxo geral na sintetização dos circuitos, desde a compilação dos programas de entrada até a geração de arquiteturas RTL.

3.1.1 Fluxo de síntese

Segundo Morawiec e Coussy (2008) e Meredith e Takach (2009), o fluxo de síntese das ferramentas de HLS é composto pelas seguintes etapas:

Compilação - transformação de um algoritmo descrito em uma linguagem de alto nível para algum outro tipo de representação

Alocação - definição das unidades funcionais disponíveis no *hardware*

Escalonamento - mapeamento das instruções a cada ciclo de *clock* do *hardware*

Emparelhamento - ligar cada instrução à unidade funcional que a executará

Geração - criação do arquivo de saída a partir da arquitetura produzida nos passos anteriores

3.1.2 Compilação

A entrada de uma ferramenta de síntese de alto nível consiste em um programa que descreve o comportamento desejado a ser feito via *hardware*. Algumas dessas ferramentas,

tais como o OpenCL[1] e o LegUp[2], usam linguagens com sintaxes baseadas em C devido a uma proximidade maior delas com o *hardware* de um computador sem diminuir demais o nível de abstração.

Algoritmos podem ser descritos como um procedimento comportamental ou atemporal em relação a um *hardware*. Idealmente, eles recebem todos os dados de entrada simultaneamente, realizam seus processamentos de forma instantânea, e devolvem todos os dados de saída de uma vez. Esse comportamento não é realista, divergindo muito da forma como um sistema de *hardware* ou *software* funciona. Um sistema de *software*, ainda que seja executado através de instruções após compilado ou interpretado, também é desenvolvido de forma atemporal em relação ao *hardware*, pois não há uma preocupação com quando (isto é, em qual ciclo de *clock* específico) cada instrução será executada. Dessa forma, há a necessidade de transformar o programa em um procedimento temporal, onde cada instrução é executada em ciclos bem definidos.

A modelagem atemporal deve ser traduzida em outra, temporal, onde os ciclos de *clock* do circuito são levados em consideração na execução das operações descritas. Para tanto, um modelo formal do comportamento do circuito é criado para visualizar melhor as dependências de dados e de controle de fluxo do algoritmo. O modelo é representado por um grafo direcionado chamado DFG ('*Data flow graph*' ou grafo de fluxo de dados), onde os arcos são valores constantes ou variáveis e os vértices são operações que usam os valores. Essa forma de representação explicita o paralelismo intrínseco ao algoritmo descrito, facilitando as fases seguintes da síntese.

Como os DFGs representam apenas fluxos de dados, há dificuldades em utilizá-los para representar laços limitados por variáveis ao invés de constantes (e.g. `for (int i = 0; i < n; i++)`) ou trechos condicionais (e.g. `if-else`). Para tanto, seriam necessárias transformações no grafo que, dependendo da complexidade da implementação, poderia gastar mais memória para armazenamento e mais processamento.

Pensando nisso, uma versão estendida do DFG foi criada, chamada CDFG ("*Control and Data Flow Graph*", ou grafo de fluxo de controle e dados), onde os arcos são controles de fluxo (como `if-else`) e os nós são *blocos básicos*. Blocos básicos são sequências de instruções com apenas um ponto de entrada e um ponto de saída. Os CDFGs são mais expressivos por conseguirem representar tanto o fluxo de dados quanto o de controle; no entanto, faz-se necessária uma análise mais profunda para explicitar as dependências de dados e memória entre as operações dentro dos blocos básicos e expor o paralelismo entre eles. Um exemplo simples de um CDFG é dado pela figura 3.1.

Apesar da possibilidade de traduzir uma larga gama de algoritmos para uma descrição temporal, é importante ressaltar que nem todos podem ser descritos diretamente em *hardware*. Um bom exemplo são algoritmos recursivos, que não são convertidos para formas iterativas de maneira automatizada caso não sejam casos de recursão de cauda.

3.1.3 Alocação

A compilação do modelo comportamental explicita as operações feitas no algoritmo e em qual ordem devem ser feitas. Após essa etapa, é preciso transformar essas representações abstratas no modelo lógico/físico do circuito.

Na fase de alocação, ocorre a identificação dos recursos de *hardware* necessários para implementar o circuito desejado. Dentre esses recursos, podemos citar as unidades funcionais, unidades de memória, barramentos de comunicação, dentre outros. A alocação destes

[1] https://www.khronos.org/opencl/

[2] http://legup.eecg.utoronto.ca/

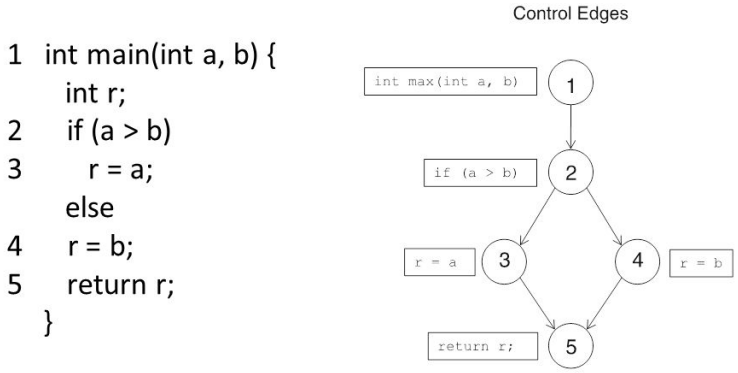

Figura 3.1: *Exemplo de um programa com seu grafo de fluxo de controle e dados.*

componentes é feita usando a biblioteca RTL das ferramenta de HLS. Ela contém os recursos disponíveis para cada modelo de *hardware* coberto pela ferramenta, bem como dados sobre esses recursos (e.g. área necessária, consumo de energia, latência) necessários para outras fases da síntese.

Certos componentes a serem alocadas, principalmente as de comunicação como os barramentos, podem ser deixadas para uma alocação tardia a fim de otimizar sua utilização. Ela pode ser realizada depois da fase de emparelhamento, para otimizar as comunicações entre as unidades funcionais, ou da fase de escalonamento, para não introduzir restrições de paralelismo entre as operações das unidades funcionais.

3.1.4 Escalonamento

A fase de escalonamento é responsável por mapear as operações feitas pelo circuito a cada ciclo de *clock* do *hardware*, levando em consideração as dependências de dados, fluxo e memória entre elas, as restrições desejadas do modelo (como área ou consumo de energia máximos) e os componentes alocados.

Nela, a representação do modelo em um CDFG é de extrema valia. Ao usá-lo, o escalonador reconhece o possível paralelismo entre blocos básicos, que é aplicado para otimizar o processamento dentro das restrições estabelecidas. Aproveitam-se possíveis faltas de dependência de dados para realizar múltiplas operações por ciclo de *clock*, sob a restrição de haver unidades funcionais suficientes para tal. Neste caso, é notável como aumentar a área implementada de circuito, o número de recursos alocados e a energia consumida, pode diminuir o consumo de tempo e aumentar a taxa de processamento de dados.

Na análise interna dos blocos básicos, a latência e a dependência de dados das operações contidas neles são usadas para determinar onde cada uma delas deve começar e terminar em relação às demais operações. Dependendo do algoritmo de escalonamento utilizado, é possível aplicar otimizações como o encadeamento de operações (ou *operation chaining*), onde uma operação é colocada no mesmo ciclo de *clock* que outra operação da qual ela depende. Dessa forma, a latência da execução geral do algoritmo é diminuída.

É também durante essa fase que pode ocorrer a comunicação entre a alocação e o emparelhamento para otimizar aspectos do *layout* do circuito digital. Essas três fases estão intimamente ligadas por lidarem diretamente com a síntese do circuito, diferente da compilação, que lida com o comportamento de forma ainda abstrata, e da geração da arquitetura RTL, que usa os dados gerados pela síntese para construir o circuito.

Um exemplo de escalonamento de instruções pode ser visto na figura 3.2. Nela, uma série de instruções dadas na LLVM IR (vide seção 3.2) são mapeadas para ciclos de *clock* que

respeitem as dependências de dados e fluxo entre elas.

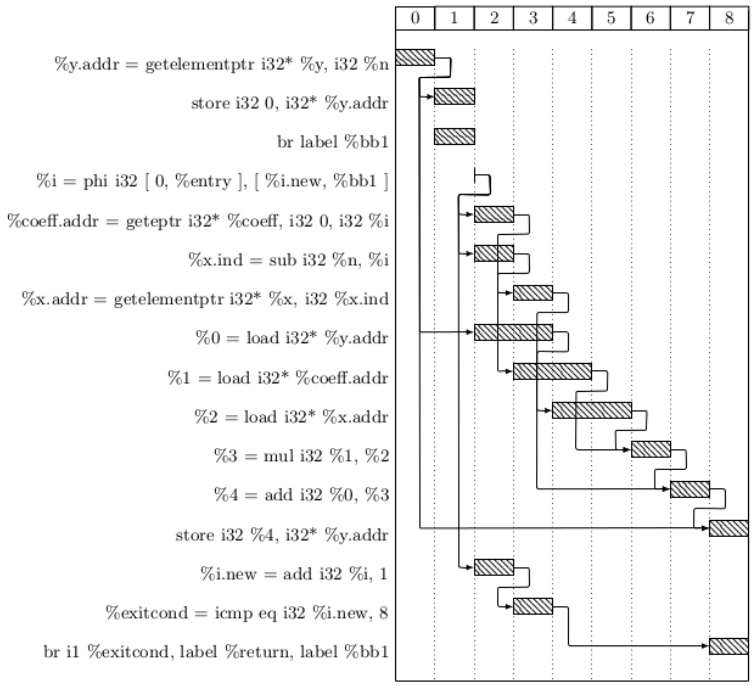

Figura 3.2: *Exemplo de escalonamento de uma sequência de instruções.*

3.1.5 Emparelhamento

Para cada operação que um algoritmo executa, é preciso não só alocar os recursos necessários para efetuá-la, como também definir a unidade funcional, de memória ou de comunicação na qual ela será feita. A fase de emparelhamento é a responsável por essa tarefa, utilizando-se dos resultados das outras fases para fazer tais ligações. Nela, podem ocorrer mais otimizações, usufruindo da comunicação com as fases de escalonamento e alocação, para diminuir a área utilizada.

Por exemplo: se duas operações são feitas em ciclos diferentes pelo mesmo tipo de unidade funcional, pode-se reutilizar a unidade funcional designada para elas, apresentando economia de recursos do *hardware*. Da mesma forma, unidades de memória podem guardar valores de variáveis que possuem tempos de vida diferentes, possivelmente detectados pela análise de vida de variáveis (*live variable analysis*) feita na fase de compilação.

3.1.6 Geração

Após a ferramenta de síntese de alto nível ter realizado todas as suas fases, é gerada uma arquitetura RTL representando, em *hardware*, o comportamento descrito pelo modelo. O arquivo de saída pode ser de diversos formatos, tais como SystemC, Verilog e VHDL. Cada arcabouço trabalha com um número limitado de modelos de placa FPGA, uma vez que o uso de FPGAs em placas integradas (*SoC FPGAs*, do inglês "*System-on-a-Chip FPGA*") está crescendo e, por consequência, a variedade de FPGAs está aumentando.

3.1.7 Considerações especiais

As fases de alocação, escalonamento e emparelhamento estão intimamente ligadas, como já observado ao longo da seção anterior. A compilação do programa e a geração do RTL

transformam, respectivamente, linguagens de alto nível, seja de programação ou de descrição de *hardware*, em uma representação intermediária e vice-versa. Por sua vez, essas três fases manipulam a representação intermediária com o objetivo de dizer, de forma concreta, de quais recursos do *chip* e quando o algoritmo em execução precisará deles. Essas etapas podem ocorrer de forma concorrente ou sequencial, dependendo da arquitetura da ferramenta, e a ordem de execução delas pode alterar a construção do circuito. Por exemplo:

- A alocação pode ocorrer primeiro quando há restrição de recursos. Dessa forma, a ferramenta otimiza a latência e o *throughput* (isto é, a quantidade de dados processados por unidade de tempo) do circuito a partir da quantidade de recursos disponível. É mais usado ao programar *chips* com poucas LUTs.

- Em contrapartida, o escalonamento pode tomar lugar antes da alocação quando há restrição de tempo. Assim, o algoritmo de síntese tenta otimizar a quantidade de recursos alocados e área utilizada dado o tempo máximo de cada operação. Essa estratégia pode se fazer mais útil em aplicações críticas, como FPGAs automotivos aplicados a carros inteligentes, onde as decisões feitas pelo algoritmo têm que ocorrer sempre em um espaço de tempo previsível e limitado (e.g. RTOS).

- A execução das três fases podem ocorrer de forma concorrente e intercomunicativa, de forma que os três processos se otimizem mutuamente. Apesar desse ser o modelo ideal, ele cria um modelo complexo demais, que acaba não sendo possível de se aplicar no processo de síntese de alto nível em exemplos realistas.

Em geral, aplicações com restrições diferentes exigem ordens de execução diferentes. Aplicações com restrições de recurso severas (e.g. área de implementação, quantidade de unidades funcionais) rodam primeiro a alocação, para estabelecer o máximo de recursos e área que o circuito poderá utilizar e, a partir disso, otimizar sua geração nos outros passos. Por outro lado, restrições de tempo exigem o uso prévio do escalonador para estabelecer a latência máxima de todo o processamento dos dados e, em seguida, ocorrem as otimizações de recursos usando esses resultados.

3.2 Projeto LLVM

LLVM (antigo acrônimo para *"Low-level virtual machine"*) é um projeto de código aberto que dispobiliza ferramentas de compilação e otimização para diversas linguagens. Tais ferramentas conseguem compilar códigos de diferentes linguagens e otimizá-los em tempo de compilação, provido de um *front-end* e um *back-end* do usuário. Por *front-end* entende-se um *parser* e um *lexer* da linguagem de programação a qual se deseja compilar, enquanto que por *back-end* entende-se uma lógica de transformação do código próprio da LLVM em código de máquina. Um exemplo de uma ferramenta famosa pertencente ao projeto LLVM é o Clang[3], um compilador de C/C++/Objective-C alternativo ao GCC, que pode apresentar perfomances superiores a este[4].

Nesta seção, serão apontadas características do projeto de forma direcionada ao entendimento do LegUp, descrito no capítulo 4.

Figura 3.3: *Estrutura básica de um compilador.*

3.2.1 Estrutura

A arquitetura mais utilizada na construção de um compilador é a chamada *arquitetura trifásica*, apresentando um *front-end*, um otimizador de código, e um *back-end*, como mostra a figura acima. O *front-end* é responsável pela transformação do arquivo de entrada em algum tipo de representação que permita sua leitura e otimização como, por exemplo, os *bytecodes* da linguagem Java. O otimizador recebe uma representação de um programa e realiza otimizações no código, que podem diminuir seu tempo de execução e/ou reduzir a quantidade de memória utilizada em sua execução. Por fim, o *back-end* converte o código otimizado na representação final desejada (também chamada de "*target*"ou "alvo"), que pode consistir em diversas representações, tais como um arquivo de texto simples que descreve o programa, ou um arquivo binário compatível com processadores da arquitetura x86. A LLVM também adota esse tipo de arquitetura, como visto na figura 3.4.

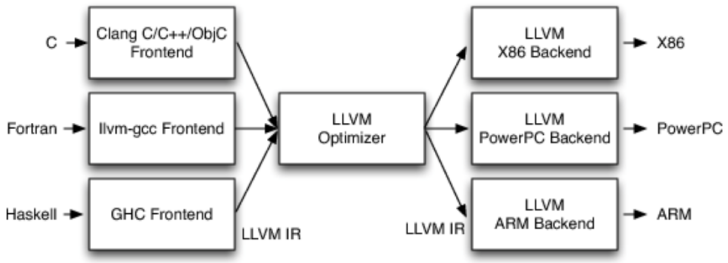

Figura 3.4: *Abstração da implementação do Projeto LLVM.*

A principal vantagem de se adotar esse tipo de estrutura é a modularização do sistema, resultando na possibilidade de se reutilizar partes do sistema para novas aplicações. Por exemplo: se existir uma aplicação cujo *front-end* recebe um código em Python, com um otimizador do código gerado pelo *front-end*, e um *back-end* que gera o código equivalente em Java, e houvesse a necessidade de mudar o alvo de Java para Haskell, não seria necessário reescrever todo o sistema apenas para mudar o *back-end*: bastaria mudar apenas a geração do código em Haskell, sem precisar repensar o resto do código.

A LLVM, além de adotar essa arquitetura, também apresenta uma forte modularização em seu código, através da orientação a objetos da linguagem C++. Isso porque aplicações como o GCC, ainda que sigam a arquitetura trifásica, possuem módulos altamente acoplados, tal que o desenvolvimento do *back-end* necessita do conhecimento do *front-end* e vice-versa. Esses tipos de aplicações são chamadas de *monolíticas*, ou seja, aplicações que possui um código altamente acoplado, com dependências difíceis de serem desfeitas sem alterar partes críticas e variadas do sistema.

[3] http://clang.llvm.org/
[4] http://clang.llvm.org/features.html#performance

3.2.2 Representação intermediária

As implementações e detalhes de ambos *front-end* e *back-end* dependem muito da aplicação para qual a LLVM está sendo usada. O *front-end* pode consistir de um *parser* e *lexer* de uma linguagem totalmente nova, cuja sintaxe siga um padrão bem diferente das linguagens já existentes, ou até um novo paradigma. O *back-end*, por sua vez, pode transformar o código em instruções ou outros códigos de outras linguagens, como Scratch[5], destinadas a robôs feitos de peças Lego, ou até um texto simples que contém o número de instruções do programa compilado em cada uma das arquiteturas de hardware existentes. Como as possibilidades são muitas, o projeto adotou um tipo de representação de código utilizado em sua arquitetura, a chamada *representação intermediária da LLVM*, mais conhecida como *LLVM IR* ("*LLVM intermediate representation*"). Esta é enviada do *front-end* ao otimizador, onde é modificada de acordo com as regras descritas pelos desenvolvedores da aplicação e, depois, encaminhada para o *back-end* construir a saída apropriada para o alvo da aplicação. Um exemplo da LLVM IR pode ser visto abaixo.

```
define i32 @add1(i32 %a, i32 %b) {
entry:
%tmp1 = add i32 %a, %b
ret i32 %tmp1
}
```

O código acima é a *representação textual* da LLVM IR, uma vez que ela também pode ser serializada em *bitcode*, isto é, ter uma representação binária. O código define uma função chamada "'add1"', que recebe dois inteiros `a` e `b` e devolve a soma deles. Como é possível perceber para quem já estudou ou viu códigos de alguma linguagem de montagem, a LLVM IR se assemelha a esse tipo de linguagem, de uma arquitetura RISC. O equivalente da função, em C, seria:

```
unsigned int add1(unsigned int a, unsigned int b) {
unsigned int tmp1 = a + b;
return tmp1;
}
```

O uso dessa representação intermediária facilita o desenvolvimento de uma aplicação ao padronizar a saída do *front-end* e a entrada do *back-end*, bem como partes do otimizador. Assim, ao criar um novo *front-end* para a LLVM, por exemplo, um programador deve saber apenas as características da entrada e da LLVM IR. Como o otimizador e o *back-end* utilizam a LLVM IR de forma independente, não é necessário saber sobre eles para a execução de seu trabalho.

3.2.3 LLVM Pass Framework

No meio do processo de compilação, e considerando a arquitetura trifásica, encontra-se o otimizador do código. Ele é responsável por realizar modificações que melhorem, por exemplo, o tempo de execução do programa e o uso de espaço de memória do computador. No caso da LLVM, o otimizador recebe um código descrito pela LLVM IR e altera as instruções ao reconhecer determinados padrões. Por exemplo, se houver uma instrução onde há a subtração de um número inteiro por ele mesmo é atribuída a uma variável:

[5] https://scratch.mit.edu/about

```
...
%tmp1 = sub i32 %a, %a
...
```

É possível, ao invés disso, atribuir 0 à variável:

```
%tmp1 = i32  0
```

Ou seja, reconhecendo um padrão na instrução (e.g. subtração de um inteiro por ele mesmo), substitui-se a instrução por outra mais eficiente (e.g. atribuir 0 à variável).

O mecanismo empregado na LLVM para realizar essas otimizações são os chamados *passes*, do arcabouço *LLVM Pass Framework*, pertencente ao projeto. Em termos práticos, os passes são etapas, possivelmente independentes entre si, pelas quais o código (ou parte dele) passa por uma análise, onde há a busca por padrões desejados em suas instruções e há possíveis alterações feitas nelas; em termos técnicos, os passes são classes derivadas da superclasse `Pass` direta ou indiretamente, que indicam o escopo mínimo pelo qual o passe é responsável (e.g. escopo global, de função, de bloco básico, de *loop*) e que implementam interfaces usados pelo arcabouço para realizar as otimizações. Cada passe é, assim, responsável por identificar padrões de instrução dentro do seu escopo e otimizar o padrão observado. A alteração retratada acima, onde temos a subtração de um inteiro por ele mesmo trocada pela atribuição da variável pelo valor 0, poderia ser colocada dentro de um passe junto de outras otimizações com respeito à aritmética de inteiros, como transformar $x - 0$ em x.

Capítulo 4

LegUp High-Level Synthesis

LegUp é um arcabouço *open-source* de síntese de alto nível desenvolvido na Universidade de Toronto (Canadá). Sua síntese converte códigos em C para Verilog e usa algumas ferramentas comerciais como o Quartus Prime II, da Intel, e o Tiger MIPS Processor, da Universidade de Cambridge (Reino Unido). Seguindo o modelo trifásico de compiladores (como explicado no capítulo capítulo 3), ele implementa o *back-end* de um compilador, usando o CLang como *front-end* e o LLVM Pass Framework como otimizador.

Atualmente em sua versão 4.0[1], apresenta uma arquitetura que permite alterações em seus algoritmos de forma relativamente simples, devido à sua modularização. Como sua arquitetura usa uma compilação em escopo de funções para implementação em *hardware* - isto é, ele usa funções como unidade básica para síntese de *hardware* -, é possível, por exemplo, especificar funções específicas para aceleração em *hardware* enquanto o resto do programa é executado em *software*; tal técnica é chamada de "fluxo híbrido"pelos criadores da ferramenta e é explicada melhor na seção 4.1.1.

Os conhecimentos aqui expostos foram sintetizados a partir do uso da ferramenta, bem como da leitura da documentação e dos artigos escritos por **?** e Canis *et al.* (2012).

4.1 Fluxo de execução

A figura 4.1 representa o fluxo geral do arcabouço. A entrada da ferramenta é um programa desenvolvido em C puro, que é compilado, otimizado e transformado em sua representação intermediária (IR) pela LLVM (seção 3.2). Em seguida, na fase de alocação, o LegUp usa os dados sobre o *hardware* no qual queremos implementar o algoritmo para alocar os recursos disponíveis no *chip*, tais como blocos de memória e unidades lógicas. Na etapa de escalonamento, as instruções da IR são mapeadas do grafo de controle e fluxo de dados para uma máquina de estados finita, onde cada estado é designado para um ciclo de *clock* específico. Depois desse mapeamento, o algoritmo de síntese atribui, a cada estado da máquina de estados, os recursos do *chip* necessários para a execução de suas instruções. Com as operações e recursos definidos, o arcabouço gera o RTL equivalente ao algoritmo e, por fim, o usa para criar um arquivo de descrição de *hardware*, escrito em Verilog.

Dada a forma como o arcabouço foi construído, isto é, na linguagem C++ e utilizando o paradigma de programação orientada a objetos para modularizar o código, as etapas da síntese de alto nível feitas sobre o código compilado são implementadas em classes separadas, uma para cada etapa.

[1] A partir da versão 5.0, o LegUp tornou-se comercial.

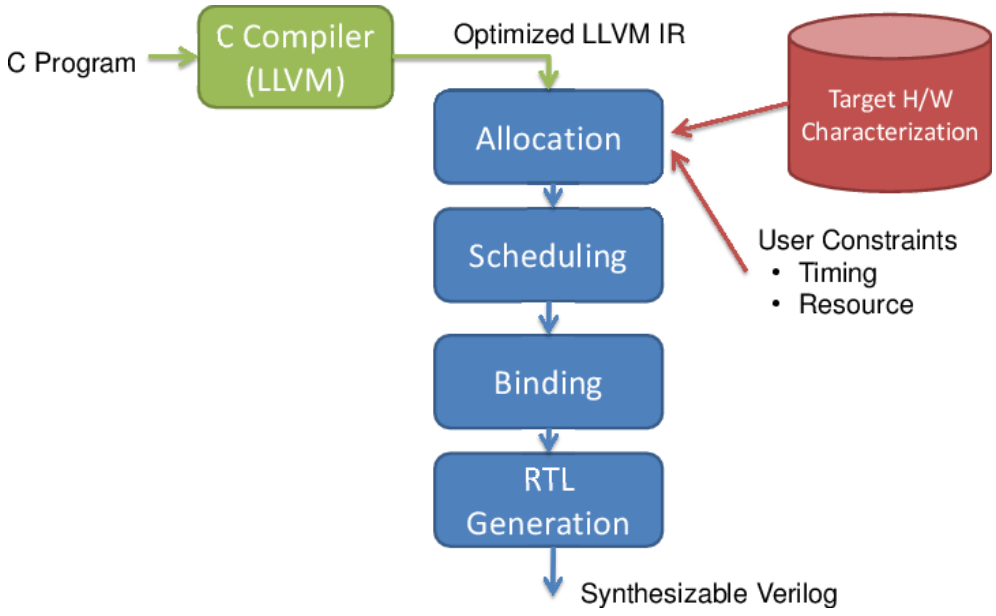

Figura 4.1: *Fluxo de execução do LegUp.*

4.1.1 Fluxos de transformação

Apesar da existência do fluxo geral de funcionamento do LegUp, a ferramenta define três fluxos distintos chamados aqui de *fluxos de transformação*. Cada um deles transforma o programa de entrada em um tipo de circuito diferente, cada qual apresentando suas vantagens e desvantagens. Os fluxos implementados são o de puro *hardware*, puro *software*, e híbrido. O fluxo utilizado neste trabalho foi o de puro *hardware*, com o intuito de aproximar cientistas da computação à compreensão do processo de pesquisa e desenvolvimento em *hardware*.

Puro hardware

Neste fluxo, todo o programa de entrada do LegUp é transformado em *hardware*. Cada função do código é mapeada em um módulo Verilog que, ao ser compilado para o *chip* FPGA, funciona de forma paralela. Devido à paralelização inerente aos componentes de *hardware*, o controle do algoritmo é feito em um módulo Verilog chamado `main`, que descreve e controla a execução da máquina de estados finita que modela o algoritmo.

A maior vantagem desse fluxo é a velocidade de execução do algoritmo, que chega a ser 8 vezes maior, como mostrado via experiências por Canis *et al.* (2012). Porém, ele não permite a implementação de técnicas importantes como recursão ou alocação dinâmica de memória.

Puro software

Neste fluxo, todo o programa de entrada do LegUp é transformado em *software*. Um processador *soft* (*softprocessor*) é instanciado pelo arcabouço, junto dos dados da aplicação, como instruções a serem executadas. Após a compilação da descrição de *hardware* na placa FPGA, o processador é executado no tecido FPGA como um processador comum. O processador usado pelo LegUp é descrito na subseção 4.1.2.

Utilizar esse fluxo dá a oportunidade de uso de técnicas importantes de programação, como recursão e alocação dinâmica de memória, ambas inviáveis via *hardware* puro. Além disso, ao executar um processador de forma isolada, o único processo existente para utilizá-lo é o da aplicação da FPGA, resultando em um menor *overhead* de troca de processos por parte de um sistema operacional. Entretanto, devido à frequência de *clock* de um *chip*

FPGA ser da ordem de 10 vezes menor que o de um processador médio de um computador pessoal atual (**colocar referencia**), mesmo com a exclusividade de acesso do processo ao processador, a velocidade de execução pode ser muito inferior a um sistema embarcado com processador de uso geral implementado em um ASIC.

Fluxo híbrido

No fluxo híbrido, o programa de entrada é compilado de forma semelhante à feita no fluxo de puro *software*. A diferença principal é o fato de que o usuário pode definir marcações no código para dizer quais funções devem ser aceleradas por *hardware*, gerando um acelerador a ser usado na chamada da função especificada. Assim, chamadas dela no código de entrada são substituídas por *funções embrulhadas* (i.e. *wrapper functions*), que enviam um sinal para o acelerador executar o processamento de dados representados pela função. Nesse cenário, o processador tem duas opções quanto a seu funcionamento durante tal chamada: continuar executando o código da aplicação enquanto continuamente verifica se o acelerador terminou sua execução, ou esperar o acelerador terminar seu processamento e então, retomar a execução da aplicação. No LegUp, a segunda opção foi adotada na implementação da ferramenta.

O fluxo híbrido permite a aceleração de funções computacionalmente pesadas enquanto ainda dá abertura para o uso das técnicas de programação proibidas no fluxo de *hardware*. Porém, sua velocidade de processamento ainda é consideravelmente inferior ao do algoritmo totalmente implementado em *hardware*.

4.1.2 Compilação

O código usado como entrada do arcabouço deve ser escrito em C e possui limitações para certos fluxos. A versão gratuita mais recente da ferramenta não cobre implementações de recursão ou alocação dinâmica de memória; apesar disso, o LegUp consegue sintetizar estruturas, controles de fluxo, aritmética de inteiros, manipulação de ponteiros (inclusive ponteiros de funções), dentre outras características da linguagem.

A compilação do código é feita no *front-end* da LLVM usando o Clang 3.5, um compilador da linguagem C pertencente ao projeto LLVM, e cria um arquivo de *bytecode* contendo a LLVM IR correspondente ao programa de entrada. Algumas funções nativas da linguagem que lidam com o manejo da memória (como `memset` e `memcopy`, da biblioteca `string.h`) são compiladas pelo Clang em funções já implementadas pela LLVM, chamadas *funções intrínsecas*. Para contornar a situação, passes do otimizador são executados no código para substituir as funções intrínsecas para funções implementadas manualmente pelo arcabouço, gerando um *bytecode* composto da LLVM IR pura, sem funções intrínsecas.

Processador

O processador *soft* utilizado pelo arcabouço é o Tiger MIPS Soft Processor[2], um processador que pode ser implementado usando síntese lógica, isto é, seu comportamento pode ser descrito por uma linguagem de descrição de *hardware* (e.g. Verilog HDL) e então convertido em um *design* de *hardware*. Possuindo um tamanho de palavra (*word size*) de 32 bits, ele é usado na elaboração do circuito apenas nos fluxos híbrido e puro *software*, onde há a necessidade de um módulo central que controle o funcionamento do circuito. A vantagem de se

[2] https://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html

usar o Tiger é seu código aberto e sua arquitetura RISC, que permitem a adição de novas instruções no processador, e de maneira menos complexa que a arquitetura CISC.

A possibilidade de modificação do processador do circuito é a característica chave do processo de autoavaliação que o LegUp realiza em seu fluxo de execução. Ao adicionar instrumentações para observar a execução do programa, é possível dizer quais instruções são mais utilizadas e por quais funções elas são mais chamadas. Isso permite uma análise extremamente precisa, uma vez que ela é feita a nível de instrução. Isso dá oportunidade ao usuário de verificar as instruções resultantes da compilação do código em C e otimizá-las manualmente, de acordo com suas necessidades.

Apesar da utilização do Tiger, que é um *softprocessor*, a versão mais recente do LegUp open-source também dá suporte aos processadores *hard* da Altera e da Xilinx. Um processador *hard* não pode ser descrito por uma HDL e, por isso, seu design é construído de forma rígida no *chip*, como propriedade intelectual. O motivo dessa impossibilidade em descrevê-lo por uma HDL é pelo fato de que um processador *hard* tem sua construção especificada a nível de transístor, resultando em uma arquitetura muito específica para ser precisamente descrita por uma descrição de *hardware*. Apesar de afetar a flexibilidade de personalização do processador, o uso desses tipos de processador aumenta a eficiência do FPGA em termos de energia, latência e área.

4.1.3 Alocação de recursos

Essa etapa é feita pela classe `Allocation` da ferramenta, e usa *scripts* TCL para efetuar a alocação dos recursos presentes no *chip* FPGA. Um desses *scripts* contém a especificação do dispositivo, opções de síntese de alto nível e restrições de tempo; outro contém as restrições de área e latência de operações. Todas essas informações são armazenadas em uma instância da classe para que os estágios seguintes da síntese possam usá-las.

4.1.4 Escalonamento

Cada função do código de entrada é transformado em uma função na LLVM IR durante a compilação. O escalonador do LegUp transforma cada uma dessas funções em um objeto da classe `FiniteStateMachine`, que representa a máquina de estados finita daquela função. Cada objeto desses contém objetos da classe `State` que guardam cada estado da máquina de estados; este, por sua vez, contém instâncias da classe *InstructionNodes* que guardam informações sobre as instruções a serem executadas no estado correspondente, tais como suas latências. Ao final do processo, o escalonador devolve um objeto *FiniteStateMachine* para cada função compilada, que serão usados na etapa de emparelhamento.

As instâncias de `InstructionNodes` são criadas por uma classe chamada `SchedulerDAG`, responsável por ler cada instrução do programa e calcular as dependências de memória e de dados entre elas e, depois, inserir nas instâncias tais cálculos. Depois do cálculo de dependências, o escalonador mapeia cada `InstructionNodes` para seus respectivos estados através da classe `SchedulerMapping`.

A estratégia adotada pelo escalonador é baseada na formulação matemática das dependências como um problema de otimização linear, chamado *sistema de restrições de diferenças*, como descrito por Cong e Zhang (2006). Nessa formulação, o programa linear contém restrições da forma

$$x_1 - x_2 \; REL \; y \tag{4.1}$$

onde

$$REL \in \{\leq, \geq, =\} \tag{4.2}$$

O termo "restrições de diferenças" dá-se pelas restrições serem compostas por diferenças de valores. Os termos x_1 e x_2 em 4.1 representam os ciclos aos quais duas operações, op_1 e op_2, devem ser mapeadas, onde op_1 é dependente de op_2. O termo à direita da inequação é uma constante que pode surgir dada a natureza da operação. Por exemplo, uma das operações pode ser uma leitura de memória e, por isso, necessitar de pelo menos y ciclos de *clock* para ser concluída.

No processo de criação do sistema linear, o arcabouço mapeia as operações para serem feitas o mais cedo possível, dado que suas dependências são satisfeitas. Tal estratégia, chamada *as-soon-as-possible scheduling* ou *ASAP scheduling*, pode ser trocada para outra, oposta, chamada *as-late-as-possible scheduling* ou *ALAP scheduling*. Finalmente, após a modelagem do programa linear com as operações e suas dependências, o sistema é resolvido utilizando-se a biblioteca *open source* `lpsolve` (http://lpsolve.sourceforge.net/). Ao resolver o programa linear, o ciclo ao qual cada operação op_n pertence será armazenado na variável x_n.

Período de *clock* do *chip* utilizado, estratégia de escalonamento (*ALAP* ou *ASAP*), dentre outras informações importantes para o processo de escalonamento são encontradas em arquivos TCL pelos diretórios da ferramenta, que podem ser modificados para customizar o processo de síntese de alto nível de acordo com as necessidades do usuário.

4.1.5 Emparelhamento

Depois de calcular quais operações devem ser feitas em quais ciclos de *clock*, o LegUp precisa atribuir cada uma dessas operações às unidades funcionais correspondentes. Como exposto no capítulo 2, essas unidades são compostas de *lookup tables* e registradores, e seus tipos e respectivas quantidades disponíveis no *chip* são determinados na fase de alocação de acordo com o dispositivo almejado.

Cada ciclo de *clock* contém um conjunto de operações a serem feitas. Estas, por sua vez, podem ser executadas por um conjunto de unidades funcionais disponíveis no *chip*. Uma única unidade funcional consegue ser usada para fazer mais de uma operação ao usar-se multiplexadores na entrada, e mapeando operações em ciclos diferentes. Este padrão de implementação de circuitos é chamado de *compartilhamento de recursos* e, em termos de recursos do *chip*, pode ser custoso. Deve-se ter em mente três pontos principais sobre o compartilhamento de recursos:

- É preferível que, caso haja necessidade de compartilhar unidades funcionais, isso seja feito da forma mais uniforme possível. Assim, evita-se a sobrecarga de uma única unidade funcional com muitas entradas e a criação de multiplexadores com muitas entradas.

- Além disso, ter um multiplexador com muitas entradas diminui a latência do circuito, devido à quantidade de lógica necessária para implementá-lo. Assim, operações que compartilham da mesma entrada no mesmo ciclo podem ser atribuídas à mesma unidade funcional sem precisar de um multiplexador.

- Uma unidade funcional pode realizar operações pertencentes a ciclos distintos. Dessa forma, ela usará apenas um registrador de saída e, por consequência, não precisará de um multiplexador de saída.

Tendo em vista estes pontos, os desenvolvedores da ferramenta criaram uma função para calcular o custo de emparelhamento entre uma operação *op* e uma unidade funcional uf, dada pela equação

$$\begin{aligned} custo(op, uf) &= \omega * numeroInputsDeMuxExistentes(fu) \\ &\quad +\beta * novasEntradasParaMux(op, fu) \\ &\quad -\theta * registradorDeSaidaCompartilhavel(op, fu) \end{aligned} \tag{4.3}$$

onde $\omega = 0.1$, $\beta = 1$ e $\theta = 0.5$. Os pesos são atribuídos a cada item considerado no compartilhamento de recursos de forma a priorizar a economia de criação de novas entradas nos multiplexadores (β), depois a economia de registradores de saída (θ) e, por fim, balancear as entradas nos multiplexadores existentes (ω).

Calculados os custos, a ferramenta modela o problema do emparelhamento da síntese de alto nível como um problema de emparelhamento de um grafo bipartido com pesos. Dois conjuntos, O e U, representam as operações e as unidades funcionais, e cada arco entre $op \in O$ e $uf \in U$ tem peso $custo(op, uf)$, como representado na equação 4.3. O problema pode ser resolvido usando-se o Método Húngaro em tempo polinomial, como descrito por Kuhn (1955). A cada ciclo de *clock*, a ferramenta faz a formulação e resolução do problema, mapeando as operações às unidades funcionais mais adequadas para executá-las.

4.1.6 Geração do RTL

A geração do RTL correspondente ao programa de entrada é feito pela classe `GenerateRTL` e, posteriormente, escrito em um arquivo Verilog pela classe `VerilogWriter`.

A classe `GenerateRTL` recebe os dados das etapas de escalonamento e emparelhamento para gerar o circuito do algoritmo usando cinco outras classes que, quando aninhadas entre si, geram a arquitetura desejada. As classes são:

- `RTLModule` - um módulo de *hardware*.
- `RTLSignal` - um registrador ou sinal no circuito. O sinal pode ser gerenciado por outros `RTLSignal`, também gerenciados por um `RTLSignal`, a fim de se criar um multiplexador.
- `RTLConst` - um valor constante.
- `RTLOp` - uma unidade funcional que representa uma operação com um, dois ou três operandos.
- `RTLWidth` - o tamanho, em bits, de um `RTLSignal`.

No RTL gerado há algumas otimizações feitas pela ferramenta a fim de melhorar o desempenho do circuito, principalmente no que diz respeito à implementação da memória dos módulos. O LegUp define a arquitetura de memória em quatro tipos: memória local, global, cache e *off-chip*. As duas últimas, que correspondem à memória cache do processador e ao gerenciador de memória externa ao *chip* FPGA, não são pertinentes ao fluxo de puro *hardware* uma vez que não existe um processador para gerenciar seus funcionamentos.

As duas hierarquias comuns a todos os fluxos, a local e a global, são usadas de acordo com a localidade das variáveis e estruturas de dados empregadas no programa, e são gerenciadas por um controlador de memória. Ao fazer uma análise sobre as referências de memória

feitas durante a execução do programa (*points-to analysis*), o LegUp verifica quais regiões de memória (e.g. vetores) são usadas por quais módulos. Se uma região é usada apenas por um módulo, uma memória local é instanciada para lidar com ela. Por outro lado, se uma região for usada por mais de um módulo, ou se a análise de ponteiros não chegar a uma conclusão definitiva sobre os ponteiros, uma memória global é instanciada.

A memória global é composta por blocos de memória *RAM* e possui um controlador de memória usado como interface entre a memória em si e os módulos que desejam acessá-la. Para cada bloco de memória, existe uma etiqueta ou *tag* que o identifica de forma única. Um endereço de memória global é composto de 32 bits, dos quais os 8 bits mais significativos são os bits de etiqueta (ou *tag bits*) e os outros 24 bits são o endereço de memória que se deseja acessar. Considerando que as etiquetas `0x0` e `0x1` são reservadas para ponteiros nulos e endereços do processador, respectivamente, é possível, então, endereçar 254 blocos de 16 *megabytes*, totalizando 4080 *megabytes* de memória. Essa quantidade é especialmente útil em placas que possuem uma memória *off-chip* grande; no entanto, no fluxo de puro *hardware*, torna-se desnecessária dada a pouca quantidade de memória que pode ser alocada pelos recursos do *chip* FPGA.

A memória local, por sua vez, é também uma instância de um bloco de memória *RAM*, mas utilizada apenas pelo módulo que a instanciou. Com isso, a latência de acesso é menor, uma vez que não há necessidade um controlador de memória. Além disso, como cada módulo tem sua memória local, há a paralelização de acesso dos módulos a elas, propriedade inexistente na memória global por conta de sua natureza compartilhável.

Na implementação dos algoritmos, é interessante utilizar uma função a cada passo complexo deles, para que dessa forma seja criado um único módulo em Verilog que contenha toda a lógica da função descrita. Por exemplo, no caso do algoritmo de aproximação do problema do caixeiro viajante, criar uma função que contenha todos os passos do cálculo de uma árvore geradora mínima de um grafo facilita a depuração e otimização do código. Essa aproximação de desenvolvimento foi utilizada nos programas dos algoritmos estudados neste trabalho, descritos no próximo capítulo.

Capítulo 5

Algoritmos

Este capítulo descreve o processo de escolha e desenvolvimento dos algoritmos usados na elaboração deste trabalho. Ambos foram desenvolvidos em linguagem C, sem o uso de bibliotecas externas, e sob as restrições impostas pelo arcabouço LegUp referente às técnicas e recursos da linguagem que poderiam ser utilizadas no fluxo de puro *hardware*.

Tal fluxo foi utilizado devido à mudança radical entre um algoritmo programado para um processador comum, e o mesmo algoritmo rodando puramente em *hardware*. Usar os fluxos híbrido ou de puro *software* trariam muitas semelhanças a sistemas já existentes e, possivelmente, mais eficientes, como sistemas embarcados com uso de microprocessadores (e.g. placas Arduino) ou mesmo um computador pessoal de propósito geral.

Vale ressaltar que o intuito deste trabalho não é se aprofundar nas provas matemáticas envolvidas na construção dos algoritmos, mas sim em seus respectivos conceitos, contextualizações e implementações. Os códigos desenvolvidos estão disponíveis na página deste trabalho[1].

5.1 Algoritmo de Huffman

Nos tempos atuais, uma quantidade massiva de dados é produzida diariamente. Por exemplo, estima-se que a rede social Twitter, no segundo quadrimestre de 2018, possuiu uma média de 335 milhões de usuários ativos mensais[2]. Se cada usuário publicar um texto de 140 caracteres ASCII, que possuem 1 *byte* cada, serão gerados $46,9$ *gigabytes* em um único instante. Apesar de parecer uma quantia baixa, a hipótese é de que cada usuário publique apenas uma vez no mês, o que é irrealista. Dessa forma, podemos supor que essa rede social, sozinha, produz mensalmente uma quantidade de dados várias ordens de grandeza maiores que isso. Na verdade, estima-se que os servidores do Twitter armazenem cerca de 250 milhões de publicações por dia[3].

Essa quantidade de dados pode ser utilizada para aplicações modernas, como análise de sentimentos ou aprendizado de máquina. Ainda assim, é necessário uma forma eficiente de armazená-la e transportá-la. Nesse contexto, surgem os algoritmos de compressão de dados, muito utilizados por *softwares* de compressão de arquivos e por bancos de dados. Um deles é relativamente simples e eficiente para grandes sequências de dados: o algoritmo de Huffman.

O algoritmo (ou codificação) de Huffman é um algoritmo que constrói uma codificação para comprimir uma sequência de caracteres com base na frequência de cada um deles no

[1] https://linux.ime.usp.br/~joaoluciano/mac0499/

[2] https://investor.twitterinc.com/static-files/4bfbf376-fefd-43cc-901e-aedd6a7f1daf

[3] https://www.quora.com/How-much-data-does-Twitter-store-daily

arquivo. A ideia do algoritmo é a de que caracteres (ou sequências de caracteres) mais frequentes sejam codificados em um código menor, diminuindo a quantidade de *bits* necessários para representá-los. Tal algoritmo é utilizado em compactadores de arquivos famosos, como o *gzip*[4].

5.1.1 Implementação

A implementação do algoritmo de Huffman envolve, em termos de estruturas de dados, o uso de *heaps* mínimos para construir uma *trie* que representa a codificação. A entrada deve conter caracteres de um conjunto fechado e previamente fornecido para o algoritmo como, por exemplo, os caracteres ASCII ou UTF-8. Tal conjunto é denominado *alfabeto* do algoritmo. A codificação é descrita pelo pseudocódigo em 1.

Algoritmo 1 Algoritmo de Huffman

Entrada: A = alfabeto do algoritmo
Entrada: S = sequência de caracteres s tal que $\forall s \in S,\ s \in A$
  $M \leftarrow contaFrequenciaCaracteres(S, A)$
  $Heap \leftarrow constroiMinHeap(M)$
  while tamanhoDoHeap > 1 **do**
    $novoNo \leftarrow criaNovoNo()$
    $filho1 \leftarrow pegaMinimoHeap(Heap)$
    $filho2 \leftarrow pegaMinimoHeap(Heap)$
    $novoNo.frequencia \leftarrow filho1.frequencia + filho2.frequencia$
    $novoNo.filhos \leftarrow filho1, filho2$
    $insereNoHeap(novoNo, Heap)$
  end while
  $trie \leftarrow pegaMinimoHeap(Heap)$
  Devolve trie

A função `contaFrequenciaCaracteres` conta a frequência de cada caractere do alfabeto A na sequência S recebida pelo algoritmo. Ela devolve um conjunto M de pares chave-valor do tipo (c, f) tal que c é um caractere do alfabeto e f é o seu número de ocorrências na entrada. O conjunto é, depois, usado para construir o *heap* mínimo usando a função `constroiMinHeap`, criando-se, para cada caractere com frequência positiva não-nula, uma *trie* de um nó contendo o caractere correspondente a ele e sua frequência. A partir disso, começa o processo de construir a *trie* de codificação para o arquivo: a cada iteração do laço, retiram-se as duas *tries* com menor frequência e cria-se um novo nó, inserindo as *tries* retiradas como filhas dele, e atribuindo à sua frequência a soma das frequências das *tries* mínimas. Perecebe-se que ao retirar 2 elementos e adicionar o novo nó no *heap*, há a diminuição de 1 em seu número de *tries* a cada iteração do laço. Ao fim do laço há um único elemento no *heap* contendo a chamada *trie de Huffman*, que representa a codificação de cada caractere. O código é gerado ao percorrê-la em uma busca em profundidade, onde nós-filhos à direita de um nó representam um `1` e nós-filhos à esquerda, `0`, finalizando ao alcançar uma folha da *trie*.

Um exemplo do resultado da execução algoritmo, criado por Sedgewick e Wayne (2011), pode ser visto na figura 5.1. A entrada utilizada foi a sequência de caracteres `ABRACADABRA!`, cujo alfabeto é o código ASCII.

[4] http://www.gzip.org/

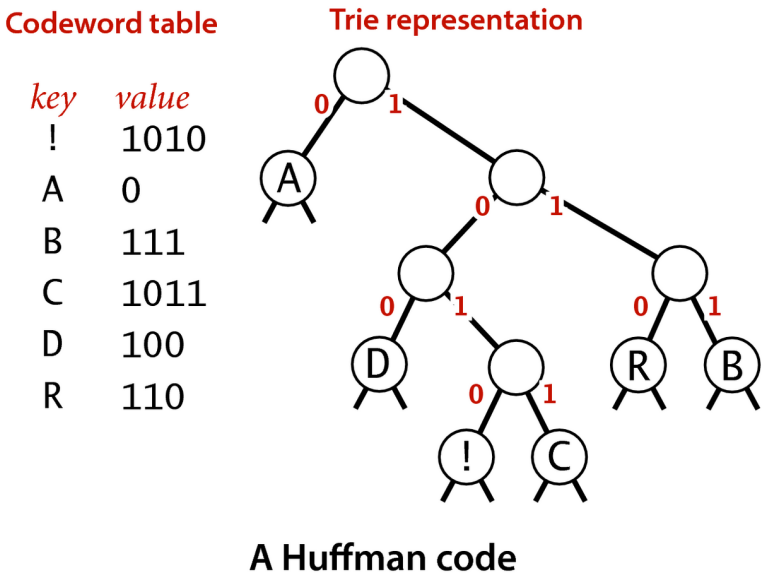

Figura 5.1: *Trie de Huffman para a frase* ABRACADABRA!

No código C, o nó é representado pela estrutura `Node`, como representado no código 5.1. Os campos `ch`, `code` e `freq` armazenam, respectivamente, o caractere do alfabeto, sua codificação final, e sua frequência na entrada. Os ponteiros `left` e `right` são usados dentro do laço de 1 para atribuir as *tries* mínimas como filhos de um novo nó, e também na geração do código de cada caractere. Por fim, `parent` e `done` são usados na codificação do alfabeto a partir da *trie* de Huffman, simulando uma busca em profundidade que percorre a *trie* e gera os códigos. Nota-se que a recursão é apenas simulada, pois ela não é permitida pelo LegUp para ser sintetizada em *hardware*.

```
typedef struct node Node;
struct node {
    unsigned long int freq;
    char ch;
    char code[50];
    short int done;
    Node *parent;
    Node *left;
    Node *right;
};
```

Listing 5.1: *Estrutura Node usada na implementação do algoritmo de Huffman*

Considerando o uso de *heap* mínimo em um vetor desordenado, o algoritmo de Huffman tem o tempo de execução de ordem $O(n \cdot log_2 n)$, onde n é o tamanho do alfabeto. No entanto, essa análise é estritamente válida para sua execução de forma atemporal, isto é, considerando que a entrada é recebida em sua totalidade de forma instantânea. No caso da implementação feita para este trabalho, o alfabeto utilizado foi o código ASCII, e o arquivo comprimido usado como entrada possuía tamanho da ordem de 2 *gigabytes* contendo apenas caracteres ASCII. Dessa forma, a leitura do arquivo e a contagem de frequência de caracteres foram os gargalos principais da experimentação feita.

Devido a esse gargalo, o foco das experiências feitas com a síntese de alto nível, na placa FPGA, foi no uso do algoritmo de aproximação para o problema do caixeiro viajante. No entanto, algumas métricas foram realizadas em termos de ciclos de *clock*, que são exibidas no capítulo 6.

5.2 Aproximação do problema do caixeiro viajante

O problema do caixeiro viajante, ou TSP (do inglês *Travelling Salesman Problem*), é um dos problemas de otimização combinatória mais famosos do mundo. Trata-se de um problema NP-Difícil, e ainda não foi encontrado um algoritmo que produza uma solução ótima em tempo polinomial. A formulação abstrata do problema é dada a seguir.

Problema do Caixeiro Viajante. *Dado um conjunto de cidades, e a distância entre cada par de cidades, qual o menor caminho que deve ser percorrido para que cada cidade seja visitada exatamente uma vez?*

O TSP é frequentemente modelado usando grafos adirecionados. As cidades são consideradas como vértices de um grafo, e as distâncias entre duas cidades são os pesos das arestas que as conectam. Existem casos específicos do problema, tal como o TSP métrico. Sua definição é

TSP métrico. *Um TSP métrico é um caso particular do problema do caixeiro viajante, tal que o grafo $G = (V, E)$ que o representa possui as seguintes propriedades:*

- *G é completo, ou seja, $\forall i, j \in V$, $\exists \bar{ij} \in E$*
- *os pesos das arestas de G respeitam a desigualdade triangular, ou seja, $\forall i, j, k \in V$, $p(\bar{ij}) \leq p(\bar{ik}) + p(\bar{kj})$, onde $p(\bar{ij})$ é o peso da aresta $\bar{ij}$.*

O caso métrico do TSP surge de forma natural pois, em exemplos reais como visitar todas as cidades de um estado brasileiro, sempre há uma rota entre duas cidades; além disso, percorrer a distância equivalente de uma rota que passa por uma cidade intermediária não deve ser maior do que a rota que vai direto para a cidade destino. Esse exemplo aponta uma das grandes utilidades da resolução do problema: a otimização de rotas em aplicações de localização, com uso de GPS, a fim de diminuir gastos com transporte.

O TSP métrico foi escolhido para implementação por ser condizente com situações reais, e apresentar um algoritmo de aproximação de tempo polinomial e implementação razoavelmente simples. O algoritmo em questão é uma 2-aproximação do TSP que calcula um caminho, no máximo, duas vezes mais comprido que o caminho ótimo do problema, como demonstrado por Rosenkrantz (1977).

5.2.1 Implementação

O algoritmo é descrito em pseudocódigo em 2.

Algoritmo 2 Algoritmo de Rosenkrantz-Stearn-Lewis para TSP métrico

Entrada: $G = (V, E)$
Entrada: $P = \{p(\bar{ij}),\ \forall i, j \in V\}$
$T \leftarrow ArvoreGeradoraMinima(G, P)$
$T' \leftarrow T + T$
$P \leftarrow CaminhoEuleriano(T')$
$C \leftarrow CaminhoHamiltoniano(P)$
Devolve C

A subrotina `ArvoreGeradoraMinima` calcula o subconjunto $T \subseteq E$ de arestas que compõem a árvore geradora mínima do grafo G, usando o algoritmo de Kruskal, como descrito por Cormen *et al.* (2009). Calculada a árvore geradora mínima T, dobram-se as arestas, isto é, cada aresta $i\bar{j} \in T$ é inserida duas vezes no conjunto T'. Com o conjunto T' é possível calcular um caminho euleriano da árvore duplicada, ou seja, um caminho que passe por todas as arestas do grafo uma única vez, utilizando-se o algoritmo de Fleury, descrito por Fleischner (1991). Por fim, calcula-se um caminho hamiltoniano a partir do caminho euleriano P usando o algoritmo 3, cuja suposição é a de que o grafo $G = (V, E)$ do qual se origina o caminho P é um grafo completo.

A ideia por trás do algoritmo 3 é usar a sequência de vértices do caminho euleriano P caso eles não tenham sido inseridos no caminho hamiltoniano C, uma vez que P foi calculado a partir da árvore geradora mínima; caso um vértice já tenha sido colocado em C, adiciona-se uma aresta que não está em P mas está em G, sob a garantia de que, por G ser completo, vão existir arestas para quaisquer pares de vértices de V.

Algoritmo 3 Algoritmo para achar um caminho hamiltoniano a partir de um caminho euleriano

Entrada: $G = (V, E)$
Entrada: $P =$ sequência de vértices $v_0, v_1, \ldots, v_n$
  $C \leftarrow v_0$
  for $v_i \in P$ **do**
    if $v_i \notin C$ **then**
      $C \leftarrow v_i$
    end if
  end for
  $C \leftarrow v_0$
  Devolve C

No algoritmo 3, P é um caminho euleriano em G. A sequência de vértices C representa as arestas de G que formam o caminho hamiltoniano, de tal forma que dois vértices consecutivos na sequência v_i e v_{i+1}, $i \in 0, \ldots, n-1$, implicam que a aresta $v_i\bar{v}_{i+1} \in E$ está contida no caminho. Note que o vértice v_0 é adicionado uma segunda vez ao final do algoritmo, para representar a aresta $v_n\bar{v}_0$ que fecha o caminho hamiltoniano.

Na implementação em C, o código usa uma estrutura denominada `Edge`, descrita melhor no código 5.2. Os campos `to` e `from` guardam os vértices de origem e destino da aresta, ainda que o grafo seja adirecionado. O campo `weight` guarda o peso da aresta, e `deleted` é usado nas subrotinas do algoritmo para simular a exclusão das arestas do grafo. Além de `Edge`, uma matriz de adjacência foi utilizada para guardar os pesos de todas as arestas do grafo, além de vetores quem contêm os caminhos euleriano e hamiltoniano, e a árvore geradora mínima. Todos os campos são do tipo `short int` na tentativa de diminuir o tamanho das entradas e, por consequência, a quantidade de memória utilizada no circuito gerado.

```
typedef struct edge {
    short int from;
    short int to;
    short int weight;
    short int deleted;
} Edge;
```

Listing 5.2: *Estrutura Edge da implementação do algoritmo de Rosenkrantz-Stearn-Lewis*

O grafo usado na implementação foi pensado de tal forma que o cálculo do caminho ótimo seja dado por uma expressão matemática fechada, a fim de comparar se o resultado devolvido pelo algoritmo 2 é, de fato, no máximo duas vezes maior que ele. A modelagem parte da árvore geradora mínima do grafo, como ilustrado pelo grafo da figura 5.2. Para a construção dele, foram usados números de vértices n tais que n é múltiplo de 3.

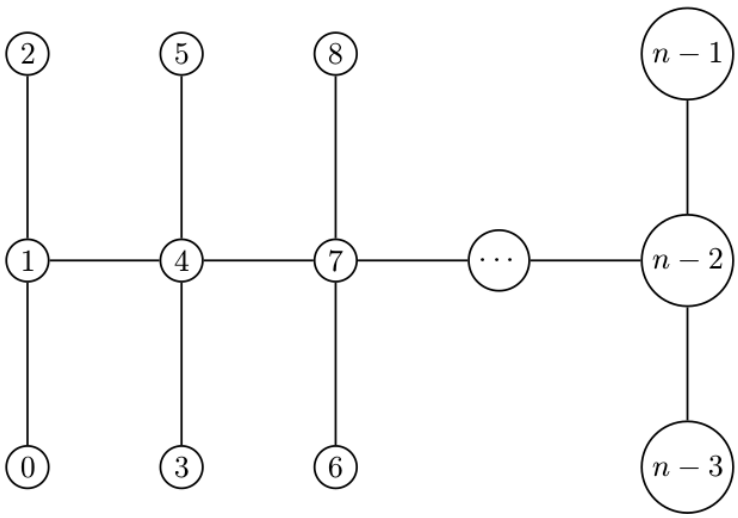

Figura 5.2: *Modelo de grafo usado na implementação da aproximação do TSP*

Na figura 5.2, as arestas pertencentes à árvore geradora mínima têm peso 5, enquanto todas as outras arestas do grafo G possuem peso 10. Os módulos dos pesos foram escolhidos de tal forma que o grafo obedeça à desigualdade triangular para qualquer subconjunto de arestas de G. O peso total de um caminho ótimo de tal grafo pode ser calculado pela expressão 5.1, e o caminho é ilustrado na figura 5.3.

$$pesoTotal = 2 \cdot 5 \cdot \frac{n}{3} + 10 \cdot \frac{n}{3} = 20 \cdot \frac{n}{3} \tag{5.1}$$

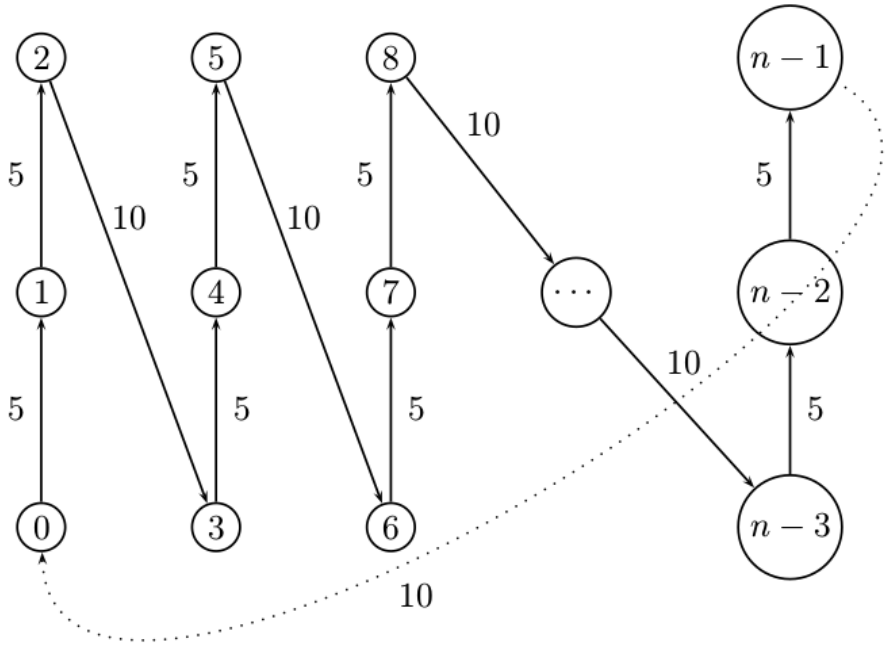

Figura 5.3: *Caminho ótimo do TSP aplicado ao grafo modelado*

O tempo de execução do algoritmo é de ordem $O(n^2 \cdot log_2 n)$. A principal vantagem do uso da implementação do TSP sobre o algoritmo de Huffman é a capacidade de aumentar o processamento realizado de forma mais expressiva: dobrar o número de vértices da entrada do TSP impacta mais o tempo de processamento do que dobrar a sequência de entrada do algoritmo de Huffman.

Além disso, para realizarem-se experimentações expressivas deste segundo algoritmo seriam necessárias, no circuito gerado, a implementação de uma interface de comunicação USB entre a placa FPGA e um computador, e a adaptação do programa sintetizado para captar a entrada por essa interface, o que fugiria muito do escopo do trabalho. No caso do TSP, menos recursos do *chip* são alocados para aumentar expressivamente a latência do processamento devido ao tempo de execução de maior ordem.

Capítulo 6

Experimentos

Este capítulo descreve os experimentos feitos com os códigos desenvolvidos, especificando ambientes, configurações e metodologia utilizados para avaliar o desempenho da implementação dos algoritmos em *hardware*, via síntese de alto nível, e *software*.

6.1 Metodologia

6.1.1 Configurações dos algoritmos

Para a execução dos experimentos, tamanhos de entrada específicos foram utilizados em cada um dos algoritmos. As entradas foram utilizadas igualmente em ambas as implementações em *software* e *hardware*.

O algoritmo de Huffman foi configurado com um texto de 682 caracteres, parte de um texto de teste chamado *Lorem ipsum*, gerado automaticamente por diversas ferramentas *online*. O alfabeto utilizado foi a tabela ASCII, contendo 128 caracteres (incluindo os não-imprimíveis). Para testar a ordem de tempo de execução do programa, foram usados textos *Lorem Ipsum* de tamanhos n tais que $n \in \{682, 1364, 2046, 2728, 3410, 4092, 4774, 5456, 6138, 6820, 7502, 8184, 8866, 9548, 10230, 10912\}$.

A 2-aproximação do TSP recebeu como entrada um número m representando o número de vértices do grafo, tal que $m \in \{12, 24, 36, 48, 60, 72, 84, 96, 108, 120, 132, 144, 156, 168, 180, 192\}$.

O tamanho e quantidade das instâncias de ambos os algoritmos foram escolhidas visando uma visualização melhor dos dados, junto de uma maior verossimilhança com a realidade. Ambos os algoritmos tiveram o tamanho de suas instâncias uniformemente espaçados em 682 caracteres no algoritmo de Huffman, e 12 vértices no algoritmo de Rosenkrantz-Stearn-Lewis. Devido ao ambiente usado na simulação (uma máquina virtual Java), as simulações demoraram substancialmente, a ponto de demorar 24 horas e 3 minutos para $m = 192$. O tamanho máximo do texto no algoritmo de Huffman foi limitado em 10912 caracteres devido ao esgotamento da memória interna do modelo de placa usado na simulação. Apesar disso, os testes foram suficientes para mostrar a curva de crescimento da quantidade de ciclos de *clock* em função do tamanho das entradas.

6.1.2 Implementação em software

A execução dos algoritmos implementados em *software* foi feita com o uso do *profiler* *OProfile*[1], disponível no repositório padrão da distribuição Ubuntu do sistema operacional

[1] http://oprofile.sourceforge.net/news/

Linux. O *OProfile* consegue medir uma contagem aproximada de eventos do processador que ocorrem durante a execução de um programa. No caso deste trabalho, ele foi utilizado para contar a quantidade total de ciclos de *clock* executados enquanto o processo especificado estava sendo executado, usando o comando `ocount` do OProfile. Por "quantidade total"entende-se a quantidade de ciclos que todos os processos usaram, não apenas os ciclos nos quais os programas a serem testados usavam o processador. Tal técnica foi adotada pois foi considerada mais verossímil com a realidade, onde circuitos de propósito geral frequentemente têm vários processos em execução.

Apesar do interesse em se medir também o número de acessos à memória, a dificuldade em se instrumentar essa medição nos algoritmos em *hardware* faria com que o resultado não fosse utilizado, devido à falta de dados da execução em *hardware* para comparação.

A compilação dos códigos foi feita usando o compilador da linguagem C `gcc` versão 5.4.0, sem nenhuma opção de otimização. A execução foi realizada 15 vezes via terminal em um computador com processador Intel Core *i*5, com dois núcleos de 1.8 GHz (totalizando frequência de 3.2 GHz em programas paralelizáveis), com aproximadamente 10 *gigabytes* de memória DDR3.

6.1.3 Implementação em hardware

Os códigos C criados no desenvolvimento dos algoritmos descritos foram utilizados como entrada do LegUp no fluxo de puro *hardware*. Não foram utilizadas opções de otimização, como as de compilação do compilador *clang*, ou as de emparelhamento e escalonamento, como *pipelining* de laços ou compartilhamento de unidades funcionais de adição.

Devido à dificuldade em lidar com as saídas serializadas da placa, foi executada 1 simulação por entrada e por algoritmo, utilizando a ferramenta *ModelSim*, dispobilizada pela Intel junto da ferramenta de programação de FPGAs *Quartus Prime*[2]. Inicialmente, mais simulações de *hardware* foram feitas por instância de teste, mas como as medidas iniciais apresentaram variância e desvio padrão iguais a 0, os testes subsequentes foram executados uma única vez. O *ModelSim*[3] foi executado através de uma opção de execução do *makefile* do LegUp, dada a disponibilidade do arquivo Verilog gerado pela síntese de alto nível.

O *bitcode* gerado pelo fluxo do LegUp foi programado na FPGA a fim de testar se o resultado da síntese era relevante para fins práticos. Após uma pequena modificação no código para acender um LED caso o resultado do algoritmo fosse correto, a placa foi programada e o resultado foi positivo, executando os algoritmos e acendendo o LED dado o resultado correto da execução. Vale notar que o tempo de compilação do RTL para *bitcode* e transferência deste para a placa FPGA através do Intel Quartus Prime foi consideravelmente alto, variando entre 6 e 8 minutos para sua execução completa.

A placa utilizada foi a Helio Board SoC FPGA, cujo *chip* FPGA é uma Intel Cyclone V, modelo 5CSXFC6C6U23C[4]. Dentre os recursos disponíveis, destaca-se a presença de 41.509 BLCs, 110.000 BLBs e 557 blocos M10K de memória RAM que totalizam 696.250 bytes de memória. Ambas as execuções via simulação e via programação da placa usaram uma frequência de *clock* de 70 MHz.

[2]https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html

[3]https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html

[4]Mais informações sobre a placa: https://rocketboards.org/foswiki/Documentation/MacnicaHelioSoCEvaluationKit

6.2 Resultados

Os resultados das experiências com respeito ao algoritmo de Huffman são mostrados nas tabelas 6.1 e 6.2. Os gráficos 6.1, construídos a partir dos dados dessas tabelas, ilustram o aspecto linear da ordem de tempo de execução do algoritmo para um texto suficientemente maior que o tamanho do alfabeto adotado no programa. As barras verticais ortogonais à curva do primeiro gráfico em 6.1 indicam o intervalo de confiança de 95% sobre o conjunto amostral das experiências. O intervalo não se aplica às execuções em *hardware* pois dada a natureza determinística da execução da simulação no ambiente usado, não houve variação no número de ciclos de *clock* para duas execuções da mesma instância dos algoritmos.

Número de caracteres	Média (ciclos de *clock*)	Desvio padrão
682	816.830	29.301,11
1.364	841.969	26.246,27
2.046	919.522	29.937,78
2.728	946.833	50.362,63
3.410	975.786	79.094,14
4.092	978.415	51.311,15
4.774	964.691	35.301,45
5.456	990.615	31.539,49
6.138	996.154	59.283,06
6.820	1.006.365	44.616,39
7.502	1.012.693	33.215,43
8.184	1.017.322	24.861,31
8.866	1.025.059	24.720,50
9.548	1.038.327	49.803,70
10.230	1.037.896	35.646,26
10.912	1.053.150	37.941,55

Tabela 6.1: *Dados das execuções do algoritmo de Huffman em software*

Número de caracteres	Ciclos de *clock*
682	12.066
1.364	16.158
2.046	20.250
2.728	24.342
3.410	28.434
4.092	32.526
4.774	36.618
5.456	40.710
6.138	44.802
6.820	48.894
7.502	52.986
8.184	57.078
8.866	61.170
9.548	65.262
10.230	69.354
10.912	74.235

Tabela 6.2: *Dados da simulação em hardware do algoritmo de Huffman*

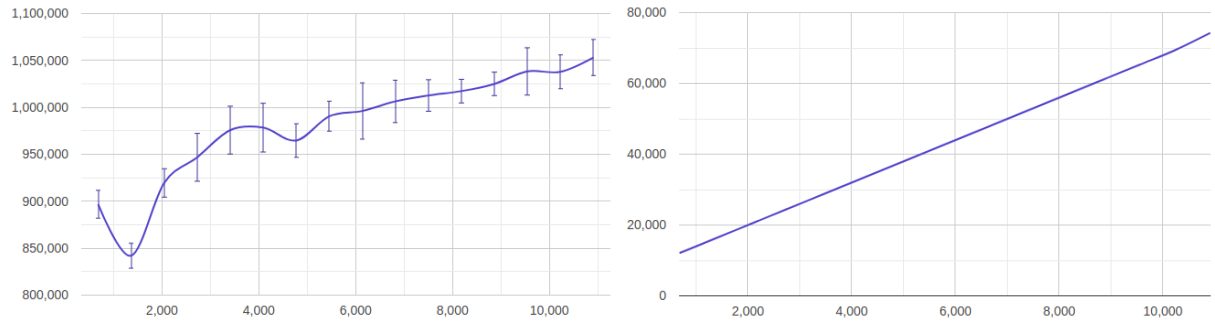

Figura 6.1: *Comparação de ciclos de clock do algoritmo de Huffman entre execuções em software (à esquerda) e hardware (à direita).*

Para os experimentos feitos com o algoritmo de Rosenkrantz-Stearn-Lewis, os dados gerados são representados nas tabelas 6.3 e 6.4, bem como nos gráficos presentes em 6.2. Nota-se o caráter quadrático da ordem do tempo de execução do algoritmo. Da mesma forma que o algoritmo de Huffman, foi adotado um intervalo de confiança de 95% com base no conjunto amostral de experiências.

Tamanho da entrada (vértices)	Média (ciclos de *clock*)	Desvio padrão
12	979.847	4.966,96
24	1.376.867	62.074,81
36	2.544.276	36.565,32
48	4.238.226	25.501,12
60	7.290.301	95.772,14
72	12.240.208	188.018,34
84	18.376.907	317.427,61
96	25.392.184	174.893,74
108	36.580.303	651.705,61
120	45.496.549	1.277.070,90
132	63.995.800	1.383.403,12
144	82.581.032	1.260.099,51
156	104.586.325	1.046.594,03
168	129.343.059	878.890,74
180	150.005.045	1.107.449,01
192	184.441.878	441.135,94

Tabela 6.3: *Dados das execuções do algoritmo de Rosenkrantz-Stearn-Lewis em software*

Tamanho da entrada (vértices)	Ciclos de *clock*
12	41.855
24	229.790
36	655.461
48	1.405.427
60	2.580.465
72	4.253.286
84	6.547.620
96	9.519.171
108	13.280.668
120	17.911.164
132	23.519.271
144	30.181.566
156	38.028.744
168	47.178.533
180	57.484.346
192	69.323.457

Tabela 6.4: *Dados da simulação em hardware do algoritmo de Rosenkrantz-Stearn-Lewis*

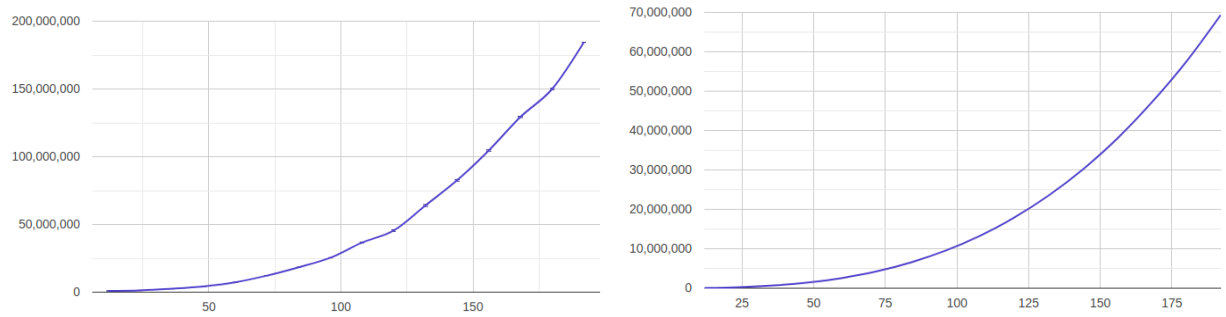

Figura 6.2: *Comparação de ciclos de clock do algoritmo de Rosenkrantz-Stearn-Lewis entre execuções em software (à esquerda) e hardware (à direita).*

Ambos os gráficos apontam que o crescimento do tempo de execução no ambiente de *software* é maior que no ambiente de *hardware*. A causa mais provável para esse fenômeno é o fato de que ao aumentar o tamanho da entrada do *software*, são necessários mais períodos de escalonamento dos processos por parte do sistema operacional, já que o tempo de execução dos programas em si aumenta. Em *hardware*, como todo o processamento é dedicado à execução dos algoritmos (uma vez que o circuito foi construído especificamente para a execução do algoritmo), não há *overhead* gerado pela execução da troca de contexto dos escalonadores, nem ociosidade do algoritmo enquanto outros processos são executados. Ou seja: em *hardware*, os algoritmos estão sendo executados 100% do tempo.

Mesmo com a exclusividade dos algoritmos em usar os recursos da placa, vale lembrar que a diferença entre as frequências de *clock* máxima de um *chip* ASIC (como um processador Intel) e de uma placa FPGA é de 1 ou 2 ordens de grandezas. Por exemplo, a frequência máxima da placa usada nos experimentos é de 100 MHz, contra 3.2 GHz da máquina utilizada para executar os *softwares*, apresentando uma proporção de 1 para 32 ciclos de *clock* entre os dispositivos. No caso das máquinas usadas neste trabalho, a placa FPGA e o computador executavam, respectivamente, 1 ciclo de *clock* a cada $1,4 \times 10^{-8}$ e $5,6 \times 10^{-10}$ segundos. Usando os dados do algoritmo de Huffman para um texto de 5.456 caracteres, o tempo de execução (em segundos) em *hardware* e *software* apresentados em 6.1 e 6.2.

$$1,4 \times 10^{-8} \times 40.710 = 5,6994 \times 10^{-4} s \tag{6.1}$$

$$5,6 \times 10^{-10} \times 990.615 = 5,5474 \times 10^{-4} s \tag{6.2}$$

Da mesma forma, para o algoritmo de Rosenkrantz-Stern-Lewis, para um grafo de 192 vértices, temos o tempo de execução, em segundos e respectivamente, para os experimentos em *hardware* e *software* apresentados em 6.3 e 6.4.

$$1,4 \times 10^{-8} \times 69.323.457 = 9,7053 \times 10^{-1} s \tag{6.3}$$

$$5,6 \times 10^{-10} \times 184.441.878 = 1,0328 \times 10^{-1} s \tag{6.4}$$

Há evidências, portanto, de que a utilização de um *hardware* dedicado, implementado em um *chip* FPGA, e uma máquina de propósito geral podem apresentar desempenhos muito similares no que diz respeito ao tempo de execução, mas altamente discrepantes em termos de ciclos de *clock*.

Capítulo 7

Conclusões

Os resultados dos experimentos mostram tempos de execução muito próximos, junto de um aumento gradativo na diferença entre o número de ciclos de *clock* necessários para executar os programas em *software* e em *hardware*, o primeiro sendo superior ao segundo. Tais dados indicam que a execução em *hardware* se torna mais vantajosa para grandes instâncias dos algoritmos implementados. No entanto, a grande discrepância entre a frequência de *clock* de dispositivos ASIC (como uma CPU) e *chips* FPGA pode se tornar um empecilho no uso exclusivo de FPGAs. Por isso, nota-se uma tendência ao uso híbrido de FPGAs com outros dispositivos, como GPUs e CPUs, enquanto procura-se aumentar a capacidade de processamento dos *chips* reprogramáveis.

Um caso notável sobre a integração desses dispositivos é o da Intel, que planeja integrar circuitos reprogramáveis em seus processadores[1], a fim de possibilitar o aumento de desempenho com a personalização do circuito de acordo com as necessidades do usuário, dispobilizando APIs e tutoriais de linguagens para síntese de alto nível (e.g OpenCL).

Quanto ao LegUp, há a desvantagem de usar em seu fluxo de execução, ferramentas proprietárias cujas versões gratuitas podem restringir seu funcionamento integral. Um exemplo disso são as mensagens de aviso do simulador de *hardware* do arcabouço, que usa o ModelSim. As mensagens avisam que um número muito grande de instruções no arquivo Verilog pode afetar de forma adversa o desempenho e qualidade da simulação, o que pode ser extremamente prejudicial para pesquisas futuras. Ainda assim, em termos de usabilidade e complexidade, o arcabouço se mostrou uma ótima ferramenta para fins acadêmicos.

O conhecimento obtido e apresentado sobre o processo de síntese de alto nível e possíveis ferramentas que a utilizem com placas-alvo de maior acessibilidade, como o LegUp em relações às placas Cyclone IV[2] da Altera, ajuda a difundir a cultura de desenvolvimento em FPGAs, atraindo jovens pesquisadores à área de pesquisa em *hardware*. Dada a disparidade entre o número de pesquisadores e desenvolvedores nas áreas de *software* e *hardware*, fazer uma ponte entre elas pode ajudar a acelerar a criação de novas tecnologias.

[1] https://www.networkworld.com/article/3230929/data-center/intel-unveils-hybrid-cpu-fpga-plans.html
[2] https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-iv.html

Referências Bibliográficas

Bureau de Estatística Trabalhista dos EUA(2010) Bureau de Estatística Trabalhista dos EUA. *Occupational Outlook Handbook 2010.* Skyhorse Publishing, 19340ª edição. Citado na pág. 2

Canis *et al.***(2012)** Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kommoona, Tomasz Czajkowski, Stephen D. Brown e Jason H. Anderson. Legup: An open source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems.* Citado na pág. 15, 16

Cong e Zhang(2006) J. Cong e Z. Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. *2006 Design Automation Conference, California.* Citado na pág. 18

Cormen *et al.***(2009)** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Algorithms.* MIT Press, 3ª edição. Citado na pág. 27

Farooq *et al.***(2012)** Umer Farooq, Zied Marrakchi e Habib Mehrez. *Tree-based Heterogeneous FPGA Architectures.* Springer-Verlag New York, 1ª edição. Citado na pág. 3, 6

Fleischner(1991) Herbert Fleischner. *Eulerian Paths and Related Topics.* North Holland, 1ª edição. Citado na pág. 27

Kuhn(1955) H. W. Kuhn. The hungarian method for the assignment problem. Citado na pág. 20

Meredith e Takach(2009) Michael Meredith e Andres Takach. *An Introduction to High-Level Synthesis.* IEEE. Citado na pág. 7

Moore e Wilson(2017) Andrew Moore e Ron Wilson. *FPGA for Dummies.* John Wiley Sons, Inc., 2ª edição. Citado na pág. 3

Morawiec e Coussy(2008) Adam Morawiec e Philippe Coussy. *High-Level Synthesis - from algorithm to digital circuit.* Springer, 1ª edição. Citado na pág. 4, 7

Rosenkrantz(1977) Daniel J. Rosenkrantz. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing.* Citado na pág. 26

Sedgewick e Wayne(2011) Robert Sedgewick e Kevin Wayne. *Algorithms.* Addison-Wesley, 4ª edição. Citado na pág. 24

Considerações pessoais

Apesar de ter sido extremamente estressante lidar com o desenvolvimento desse trabalho, a gratificação foi muito maior. O tema foi sugerido pelo professor Alfredo, apesar de, na época, eu já ter definido um tema, e não me arrependo de ter seguido com o tema desenvolvido aqui. Revi notas de aulas de aulas das quais gostei muito de ter na graduação, como Estrutura de Dados, com o professor Paulo Feofiloff, e Algoritmos em Grafos, com o professor Marcel Kenji. Essas revisões me lembraram do porquê eu gosto do que faço e porquê quero continuar nessa área e, possivelmente, continuar pesquisando sobre o uso de FPGAs por desenvolvedores de *software*.

O principal motivo para ter seguido a sugestão de tema do professor Algredo foi a memória do estágio que fiz no exterior, onde meus colegas lidaram com a elaboração dos circuitos de novos protótipos de placas e eu, sem poder ajudar, me frustrava bastante. Talvez disseminar mais este tema possa ajudar a desenvolvedores de *software* a não "sofrer"tanto quando quiserem aprender sobre o desenvolvimento de *hardware*.

O fato de estagiar e ainda cursar algumas matérias da graduação durante o período de desenvolvimento do trabalho também atrapalhou bastante a pesquisa, apesar dos efeitos benéficos dessas atividades em outros âmbitos pessoais.

O resultado final do trabalho foi satisfatório, considerando que os resultados das simulações foram obtidos praticamente às vésperas da data de entrega (devido a problemas nas simulações do *hardware*). Considerando o quanto tive que aprender e pesquisar desde os conceitos mais básicos dos assuntos fundamentais para a elaboração deste trabalho (como na estrutura de FPGAs e síntese de alto nível), eu posso dizer que foi bom ver o resultado das pesquisas e implementações feitas, além dos conhecimentos obtidos.