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

João Henrique Luciano

Implementação de algoritmos em FPGAs usando síntese de alto nível

São Paulo
Dezembro de 2018

Implementação de algoritmos em FPGAs usando síntese de alto nível

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman

São Paulo
Dezembro de 2018

Resumo

O presente trabalho visa o estudo e experimentação do uso de ferramentas de síntese de alto nível para o design 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. Como efeito secundário, foi criada uma referência em português para o estudo e uso da síntese de alto nível usando o sistema operacional Ubuntu, a placa SoC FPGA Cyclone V, da Altera, e o arcabouço de código aberto LegUp.

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

Abstract

This work aims the study and usage of high-level synthesis tools for electronic circuits design. Alongside that, 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 downloaded into a FPGA. As side effect, there is the creation of a brazilian reference for this area of study, which is quite scarse.

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

Sumário

Capítulo 1

Introdução

Uma monografia deve ter um capítulo inicial que é a Introdução e um capítulo final que é a Conclusão. Entre esses dois capítulos poderá ter uma sequência de capítulos que descrevem o trabalho em detalhes. Após o capítulo de conclusão, poderá ter apêndices e ao final deverá ter as referências bibliográficas.

Para a escrita de textos em Ciência da Computação, o livro de Justin Zobel, *Writing for Computer Science* (Zobel, 2004) é uma leitura obrigatória. O livro *Metodologia de Pesquisa para Ciência da Computação* de Wazlawick (2009) também merece uma boa lida.

O uso desnecessário de termos em lingua estrangeira deve ser evitado. No entanto, quando isso for necessário, os termos devem aparecer *em itálico*.

```
Modos de citação:
indesejável: [AF83] introduziu o algoritmo ótimo.
indesejável: (Andrew e Foster, 1983) introduziram o algoritmo ótimo.
certo : Andrew e Foster introduziram o algoritmo ótimo [AF83].
certo : Andrew e Foster introduziram o algoritmo ótimo (Andrew e Foster, 1983).
certo : Andrew e Foster (1983) introduziram o algoritmo ótimo.
```

Uma prática recomendável na escrita de textos é descrever as legendas das figuras e tabelas em forma auto-contida: as legendas devem ser razoavelmente completas, de modo que o leitor possa entender a figura sem ler o texto onde a figura ou tabela é citada.

Apresentar os resultados de forma simples, clara e completa é uma tarefa que requer inspiração. Nesse sentido, o livro de Tufte (2001), *The Visual Display of Quantitative Information*, serve de ajuda na criação de figuras que permitam entender e interpretar dados/resultados de forma eficiente.

Capítulo 2

FPGA

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.

2.3 Componentes

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

- Blocos lógicos
- Rotas de interconexão
- Blocos de entrada e saída

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 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 noss 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.

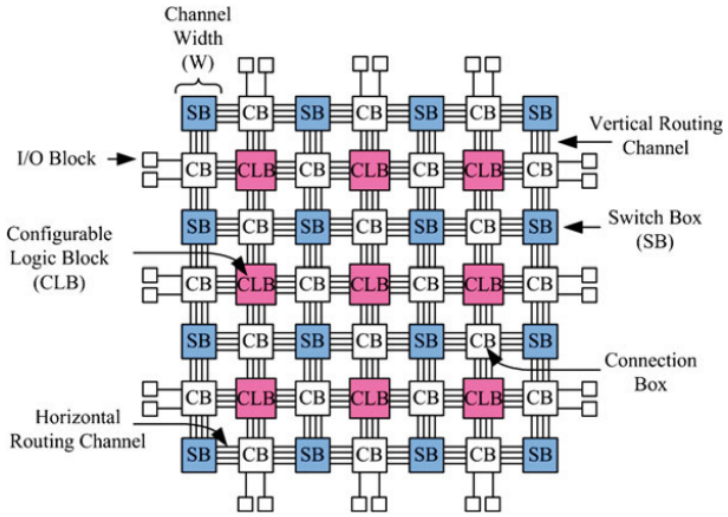

Figura 2.1: *Exemplo de um FPGA com rede de interconexão em malha.*

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

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 disto, os FPGAs são mais lentos e consomem mais energia do que os *ASIC*s.

Um exemplo de figura está na figura 4.1.

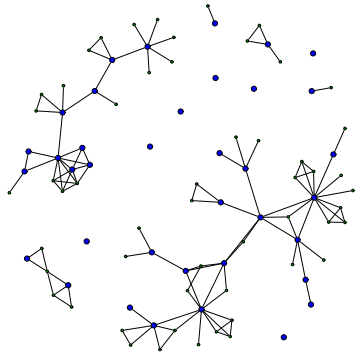

Figura 2.2: *Exemplo de uma figura.*

Capítulo 3

Síntese de alto nível

Síntese de alto nível ('High level synthesis', ou HLS) é o processo de transformação linguagens de programação de alto nível para sintetizar arquiteturas RTL ('*Register-transfer level*' ou 'nível de registrador e transferência'), isto é, sintetizar arquiteturas de circuitos digitais síncronos a partir de descrições comportamentais (ou algorítmicas) do *hardware*. As saídas geradas são, em sua maioria, arquivos em linguagens HDL ('*Hardware description language*' ou linguagem de descrição de *hardware*), que são usadas para configurar o *hardware*. Tais ferramentas realizam o mesmo fluxo básico na sintetização dos circuitos, desde a especificação comportamental desejada até a geração dos arquivos em HDLs.

3.1 Fluxo de síntese

O fluxo de síntese das ferramentas de HLS segue o mesmo padrão, que envolve:

- Modelagem ou especificação
- Compilação
- Alocação
- Escalonamento
- emparelhamento
- Geração

3.2 Modelagem

A etapa de modelagem consiste em descrever o comportamento desejado do *hardware* em questão, onde as entradas, saídas e forma de processamento dos dados são especificadas através do uso de linguagens programação de alto nível. As ferramentas de HLS, tais como o OpenCL(https://www.khronos.org/opencl/) e o LegUp(http://legup.eecg.utoronto.ca/), comumente usam linguagens com sintaxes baseadas em C.

A descrição é dita comportamental ou atemporal, onde o comportamento descrito recebe todos os dados de entrada simultaneamente, realiza seu processamento de forma instantânea, e retorna todos os dados de saída de uma vez. Esse tipo de descrição não é condizente com a forma como um sistema de *hardware* funciona, então há a necessidade de conversão do tipo atemporal para temporal.

É importante ressaltar que nem todos os algoritmos podem ser descritos diretamente em *hardware*. Um bom exemplo são algoritmos recursivos, que não são convertidos para formas iterativas de maneira automatizada.

3.3 Compilação

A modelagem atemporal deve ser compilada 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 de 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 (i.e `for (int i = 0; i < n; i++)`) ou trechos condicionais (`if-elses`). Para tanto, seria necessária transformações no grafo que, dependendo da complexidade da implementação, poderia gastar muito mais memória para armazenamento e muito 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's e 'goto's) e os nós são chamados de *blocos básicos*. Blocos básicos são blocos de código sequenciais sem pontos de saída (por exemplo, um 'return' ou 'goto' no meio do código) ou ramificações (causadas por fluxos condicionais, por exemplo) (podemos dizer que blocos básicos são os blocos de código entre um 'jump target' e um 'jump'). Os CDFGs são mais expressivos por conseguirem representar tanto o fluxo de dados quanto o de controle; entretando, faz-se necessária uma análise mais profunda para explorar o paralelismo dentro dos blocos básicos e expor o parelelismo entre os blocos.

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

Vale lembrar que certas componentes a serem alocadas, principalmente as de comunicação (como os barramentos), podem ser deixadas para serem alocadas posteriormente a fim de otimização, como 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).

3.5 Escalonamento

Como apontado anteriormente, o processo de HLS transforma uma descrição atemporal para uma temporal, ou seja, que considera os ciclos de *clock* do sistema de *hardware*. A fase de escalonamento se encarrega de planejar o processamento dos dados de entrada a cada um desses ciclos, levando em consideração os dados especificados, as operações, as restrições desejadas do modelo (tais como área ou consumo de energia máximos) e os componentes alocados.

Durante essa fase, a representação do modelo em um CDFG é de extrema valia. Isso porque com o CDFG, evidencia-se o paralelismo entre blocos básicos (e, dependendo da profundidade da análise, o paralelismo dentro deles), e este é aproveitado pelo escalonador para otimizar o processamento de dados dentro das restrições estabelecidas. Aproveitam-se possíveis falta de dependência entre dados para realizar múltiplas operações por ciclo de *clock*, sob a restrição de haver unidades funcionais suficientes para tal (vê-se aí, por exemplo, a relação entre aumentar a área implementada de circuito implementada, número de recursos alocados e energia consumida, e diminuir o consumo de tempo e aumentar a taxa de processamento de dados). Dessa forma, uma operação pode ser escalonada pra ser executada ao longo de um ou mais ciclos de *clock*.

É também durante essa fase que pode ocorrer a comunicação entre a alocação e o emparelhamento para otimizar aspectos do *layout* do circuito digital, pois estas 3 fases estão intimamente ligadas por lidarem com o circuito de fato (diferente da modelagem e compilação, que lidam com o comportamento de forma ainda abstrata).

3.6 Emparelhamento

Para cada operação que nosso algoritmo descreve, precisamos não só alocar os recursos necessários para efetuar a operação como também ligar tanto as operações quanto as variáveis aos recursos alocados. A fase de emparelhamento (do inglês *binding*) 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, como já citado) para diminuir a área utilizada. Por exemplo: se uma mesma operação é feita em ciclos diferentes, pode-se reutilizar a unidade funcional daquela operação. Da mesma forma, unidades de memória podem guardar valores de variáveis que possuem um tempo de vida diferente.

3.7 Geração

Finalmente, após o algoritmo de síntese ter realizado todas as suas operações, é gerado um arquivo com uma arquitetura RTL representando o comportamento descrito pelo modelo. O arquivo de saída pode ser de diversos formatos, tais como SystemC, Verilog e VHDL. Vale ressaltar que cada *framework* geralmente trabalha com um número limitado de modelos de placa FPGA, uma vez que estão cada vez mais frequentes o uso de FPGAs em placas integradas, tornando-se um SoC FPGA (do inglês "*System-on-a-Chip FPGA*"). Mais informações estão disponíveis no capítulo 2, sobre FPGAs.

3.8 Alocação, Escalonamento e emparelhamento: 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 programação em uma representação intermediária e vice-versa. Já essas três fases manipulam a representação intermediária com o objetivo de dizer, de forma concreta, de quais recursos do *chip* e quando o algoritmo precisará deles. Essas etapas podem ocorrer de forma concorrente ou sequencial, dependendo da arquitetura da ferramenta, e essas formas de execução podem 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.

- 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. Restrições de recurso (*e.g.* área de implementação, quantidade de unidades funcionais etc) 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 do processamento dos dados e, em seguida, ocorrem as otimizações possíveis em cima desse primeiro resultado.

Um exemplo de figura está na figura 4.1.

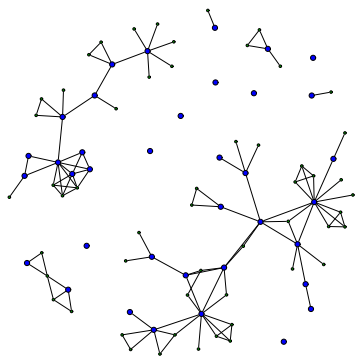

Figura 3.1: *Exemplo de uma figura.*

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).

Atualmente em sua versão 4.0, 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 Fluxo de execução

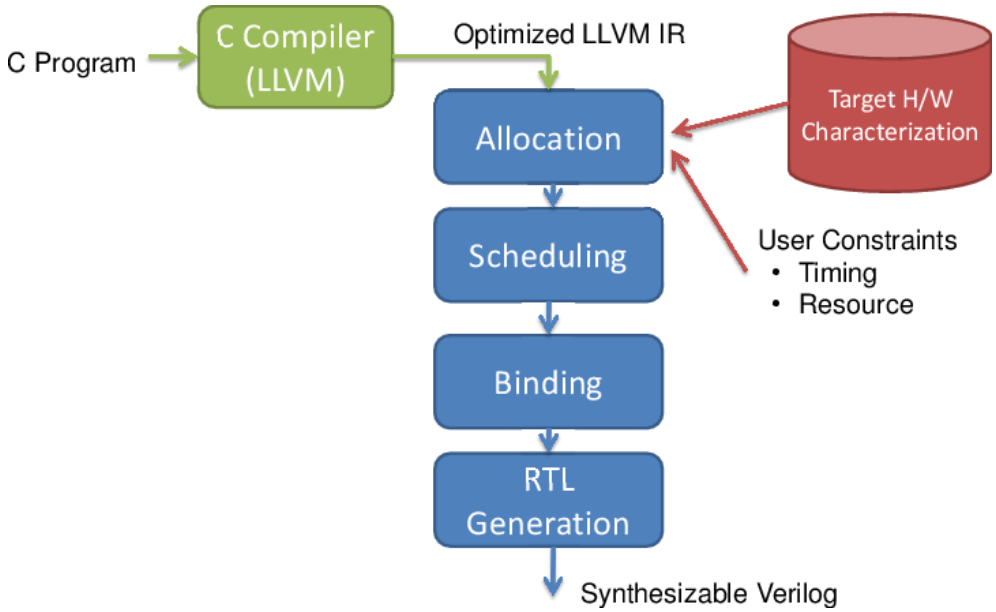

Figura 4.1: *Fluxo de execução do LegUp.*

A figura acima 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 (vide apêndice A). 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 esse 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++, 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.

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 (**colocar referencia**). 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 suporta 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 compliador 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 (https://www.cl.cam.ac.uk/t
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 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 customização do processador, o uso desses tipos de processador aumenta a eficiência do FPGA em termos de energia, latência e área.

Nota: a partir da versão 5.0, o LegUp tornou-se comercial. Veja mais aqui(https://www.legupcomputi

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 retorna 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 em (J. Cong and Z. Zhang, ?An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation,? Proceedings of the 2006 Design Automation Conference, San Francisco, CA, pp. 433-438, July 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 ciclo 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 sobrecarregar uma única unidade funcional com muitas entradas, remetendo ao item anterior sobre multiplexadores com muitas entradas.

- 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 comaprtilhamento 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 [COLOCAR REF AQUI] em tempo polinomial. 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 de existir um controlador de memória. Além disso, como cada módulo tem sua memória local, há a paralelização do acesso à memória, inexistente na memória global por conta de sua natureza compartilhada.

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 (COLOCAR REF PARA O SITE DO TCC).

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 (https://investor.twitterinc.com/static-files/4bfbf376-fefd-43cc-901e-aedd6a7f1daf). 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 (REFERENCIAS AQUI: https://www.quora.com/How-much-data-does-Twitter-store-daily).

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 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* (http://www.gzip.org/).

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, retirado de (COLOCAR REFERENCIA AQUI: ALGORITHMS 4TH EDITION, DO SEDGEWICK), pode ser visto na figura 5.1. A entrada utilizada foi a sequência de caracteres `ABRACADABRA!`, cujo alfabeto é o código ASCII.

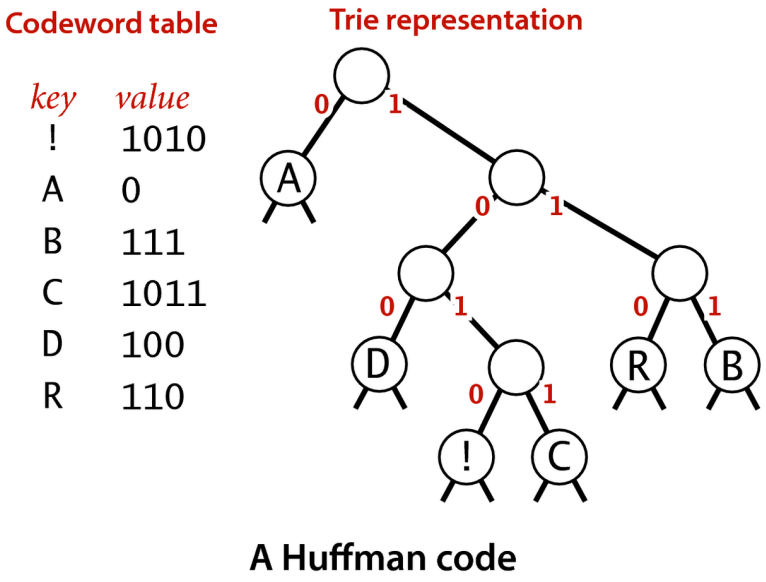

Figura 5.1: *Trie de Huffman para a frase* `ABRACADABRA!`

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 **??**.

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 evidencia 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 em (COLOCAR REFERENCIA AQUI).

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

A função `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 em (COLOCAR REFERENCIA: CORMEN). Calculada a árvore geradora mínima, dobram-se as arestas, isto é, cada aresta $i\bar{j} \in T$ possui dois elementos 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 (COLOCAR REFERENCIA). Por fim, calcula-se um caminho hamiltoniano a partir do grafo P usando o algoritmo 3.

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

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

No algoritmo 3, U 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 $v_n\bar{v}_0$.

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. Note que todos os campos são do tipo `short int` na tentativa de diminuir o tamanho das entradas do circuito gerado.

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

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

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

Conclusões

Texto texto[1].

[1]Exemplo de referência para página Web: www.vision.ime.usp.br/~jmena/stuff/tese-exemplo

Apêndice A

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)(http://clang.llvm.org/), um compilador de C/C++/Objective-C alternativo ao GCC, que (pode apresentar perfomances superiores a este)(http://clang.llvm.org/features.html#performance).

Neste apêndice, serão apontadas características do projeto de forma direcionada ao entendimento do LegUp, descrito no capítulo **??**.

A.1 Estrutura

Figura A.1: *Estrutura básica de um compilador.*

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 A.2.

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

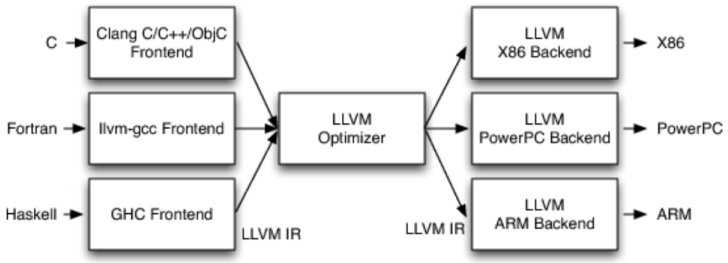

Figura A.2: *Abstração da implementação do Projeto LLVM.*

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 na prática, são muito acopladas, com dependências difíceis de serem desfeitas sem alterar partes críticas e variadas do sistema.

A.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](https://scratch.mit.edu/about), 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 retorna 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.

A.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:

```
...
%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 e busca de padrões desejados em instruções e suas possíveis alterações; 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.

Referências Bibliográficas

Tufte(2001) Edward Tufte. *The Visual Display of Quantitative Information.* Graphics Pr, 2nd edição. Citado na pág. 1

Wazlawick(2009) Raul S. Wazlawick. *Metodologia de Pesquisa em Ciencia da Computação.* Campus, primeira edição. Citado na pág. 1

Zobel(2004) Justin Zobel. *Writing for Computer Science: The art of effective communication.* Springer, segunda edição. Citado na pág. 1