TRABALHO DE GRADUAÇÃO

SÍNTESE DE ARQUITETURAS DEDICADAS A PARTIR DE LINGUAGENS FUNCIONAIS

Luiz Gustavo Soares de Sá

Brasília, Julho de 2017

ENGENHARIA
MECATRÔNICA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

SÍNTESE DE ARQUITETURAS DEDICADAS A PARTIR DE LINGUAGENS FUNCIONAIS

Luiz Gustavo Soares de Sá

Relatório submetido como requisito parcial de obtenção de grau de Engenheiro de Controle e Automação

Banca Examinadora

Prof. Ricardo Pezzoul Jacobi, CIC/UnB _____
Orientador

Prof. Marcelo Grandi Mandelli, CIC/UnB _____
Examinador interno

Prof. Carlos Humberto Llanos Quintero, ENM/UnB _____
Examinador interno

Brasília, Julho de 2017

FICHA CATALOGRÁFICA

LUIZ, DE SÁ
Síntese de arquiteturas dedicadas a partir de linguagens funcionais

[Distrito Federal] 2017.

x, 50p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2017). Trabalho de Graduação – Universidade de Brasília.Faculdade de Tecnologia.

1. Síntese de Alto Nível 2. Linguagens funcionais

I. Mecatrônica/FT/UnB II. Título (Série)

REFERÊNCIA BIBLIOGRÁFICA

LUIZ, DE SÁ, (2017). Síntese de arquiteturas dedicadas a partir de linguagens funcionais. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-$n°$015, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 50p.

CESSÃO DE DIREITOS

AUTOR: Luiz Gustavo Soares de Sá

TÍTULO DO TRABALHO DE GRADUAÇÃO: Síntese de arquiteturas dedicadas a partir de linguagens funcionais

GRAU: Engenheiro ANO: 2017

É concedida à Universidade de Brasília permissão para reproduzir cópias deste Trabalho de Graduação e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desse Trabalho de Graduação pode ser reproduzida sem autorização por escrito do autor.

Luiz Gustavo Soares de Sá

Endereço de email: luizgustavolgss@gmail.com

Campus Darcy Ribeiro, CIC, Universidade de Brasília

Brasília – DF – Brasil.

Dedicatória

Dedico esse trabalho à minha mãe, que é a pessoa mais importante da minha vida.

Luiz Gustavo Soares de Sá

Agradecimentos

Agradeço aos meus colegas de curso. Graças à ajuda deles a graduação pareceu mais tranquila do que provavelmente é.

Luiz Gustavo Soares de Sá

RESUMO

Este trabalho detalha o estudo e a implementação de um sistema de síntese de alto nível baseado em programas puramente funcionais. O sistema tem como entrada um programa constituído de funções puras e tem como saída uma implementação em SystemC da arquitetura resultante. Os resultados obtidos são funcionais mas necessitam de otimizações para serem utilizados na indústria.

ABSTRACT

This work details the study and implementation of a High Level Synthesys system based on purely functional programs. The system uses pure functions as input and produces a SystemC implementation of the resultant architecture as output. The obtained results are functional but need optimizations in order to be used in the industry.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE SÍMBOLOS

Siglas

ADT	*Algebraic Data Type* (Tipo de Dado Algébrico)
BNF	*Bakus Norm Form*
CI	Circuito Integrado
FIFO	*First-In First-Out*
HDL	*Hardware Description Language*
RTL	*Register Transfer Level*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*
VLSI	*Very Large-Scale Integration*

Capítulo 1

Introdução

Visão geral sobre o fluxo de design de hardware e a motivação por trás do projeto.

1.1 A importância da Síntese de Alto Nível

Um projeto de hardware atual começa com a implementação de um modelo funcional. Modelos funcionais de hardware não possuem detalhes da arquitetura e são geralmente desenvolvidos em `C` ou `C++`. Após testado e aprovado, o modelo funcional passa por uma série de etapas que transformam o modelo funcional em uma implementação em RTL (Register Transfer Level) do sistema.

A etapa mais importante ao transformar descrições funcionais em RTL é a definição da arquitetura do sistema. Após definida, uma descrição RTL desta arquitetura, escrita em uma linguagem de descrição de hardware como `VHDL`, `Verilog` ou `SystemC`, é desenvolvida. Quando a implementação em RTL passar a fase de testes o projeto da arquitetura pode ser considerado como finalizado [1].

No entanto, achar uma arquitetura que satisfaz o modelo funcional é a maior fonte de erros em um projeto de hardware. Achar uma arquitetura ótima é ainda mais complexo. É comum projetos de circuitos integrados entrarem em uma fase cíclica de detecção e correção de erros que modifica constantemente a arquitetura do sistema até que todos os requisitos sejam atingidos. Essa fase iterativa é, na maioria dos casos, o gargalo do projeto por ser a mais demorada. Como projetos são comumente regidos por *deadlines* muitas arquiteturas são consideradas prontas mesmo não tendo sua corretude completamente comprovada [1].

O fluxo de *design* de hardware apresentado é aceitável para circuitos menores mas parece ser inadequado para circuitos mais complexos. As inovações tecnológicas na fabricação de altíssima densidade de integração têm trazido um notório aumento na complexidade e no grau de integração de circuitos digitais — é lógico estipular que circuitos serão cada vez mais complexos e densos. O desenvolvimento de implementações em RTL não é alto nível o suficiente para acompanhar a evolução dos CI's [1][9].

A síntese de alto nível é uma maneira diferente de projetar hardware baseada na automatização

da transformação de um modelo funcional em uma implementação RTL (otimizada) do sistema. A síntese de alto nível promete eliminar o maior gargalo do *design* de hardware atual, gerando resultados com menos erros e acelerando drasticamente o desenvolvimento de hardware [1].

A síntese de alto nível é vista como o próximo passo na evolução do *design* de hardware. A indústria historicamente adota ferramentas mais produtivas (com nível de abstração maior) quando as ferramentas do momento não suprem as necessidades da indústria (tempo de desenvolvimento de projeto e qualidade do produto). Foi assim que as ferramentas de *design* foram evoluindo: desde projetos em transistores, para projetos em portas lógicas e atualmente projetos em RTL [1].

1.2 Síntese de Alto Nível de Programas Funcionais

Empresas influentes na indústria de hardware como a Mentor (com Catapult), a Cadence (com Stratus) e a Xilynx (com Vivado), por exemplo, investem em projetos de síntese de alto nível. Esses projetos têm em comum, entre outras características, serem baseados em linguagens tipo-C (`C`, `C++`, `SystemC`). O sistema proposto nesse projeto é baseado em uma linguagem puramente funcional.

Linguagens imperativas, como são as tipo-C, não parecem ser a melhor escolha quando se trata de síntese de alto nível como detalhado em [2]. Entre as características de linguagens imperativas que dificultam a síntese se destaca a natureza sequencial dessas linguagens, incentivando o programador a definir algoritmos também sequenciais que não mapeiam bem para hardware (pelo menos não trivialmente).

Linguagens puramente funcionais, ao contrário, parecem mapear bem para hardware. Por não definir estruturas de controle de fluxo e por não determinar uma ordem de execução, algoritmos definidos por funções são naturalmente concorrentes. A interpretação de funções como hardware muitas vezes é trivial e muitas vezes definições de funções se assemelham, surpreendentemente, à implementação da arquitetura do hardware. Exemplos de transformações triviais são funções combinacionais — interpretadas como conexões entre módulos — e funções recursivas — interpretadas como máquinas de estados. Linguagens funcionais parecem ser, pelo menos à primeira vista, uma entrada mais apropriada para síntese de alto nível.

1.3 Objetivo do trabalho

Este trabalho tem como objetivo a implementação de um sistema de síntese de alto nível de um programa funcional. A entrada do sistema é um subconjunto da linguagem `Haskell`: uma linguagem puramente funcional e fortemente tipada. A saída do sistema é uma arquitetura de hardware descrita em `SystemC`: uma biblioteca do `C++` utilizada para definição de circuitos em RTL usando construções apropriadas. O sistema deve ser capaz de sintetizar funções com base apenas em suas definições e nada mais.

O sistema de síntese será desenvolvido para sintetizar qualquer tipo de função porém foco

maior será dado para funções que sintetizam circuitos de processamento de dados estilo *dataflow* — ou seja, será dado foco para funções que tem como entradas e/ou saídas elementos do tipo lista.

1.4 Conteúdo dos capítulos

Os capítulos, seguindo a ordem de exibição, foram escolhidos de maneira à introduzir, primeiramente, os conceitos necessários — sistemas integrados no capítulo 2 e linguagens funcionais no capítulo 3 — para depois detalhar o sistema de síntese — capítulo 4 — e mostrar seus resultados — capítulo 5. Seguem as descrições resumidas de cada capítulo:

Capítulo 2 - Sistemas Integrados Discussão à respeito de como projetos de sistemas integrados são implementados. Ênfase em síntese de alto nível e modelagem em `SystemC`

Capítulo 3 - Linguagens Funcionais Introdução sobre as características mais importantes das linguagens funcionais exploradas neste projeto

Capítulo 4 - Sistema Proposto Detalha o sistema de síntese proposto desde a entrada até a saída explicando os métodos utilizados

Capítulo 5 - Resultados Obtidos Discute os resultados obtidos pelo sistema proposto e mostra os pontos positivos e negativos da implementação

Capítulo 6 - Conclusão Conclusões e trabalhos futuros

Capítulo 2

Sistemas Integrados

Visão geral o processo de design de sistemas integrados e a importância da síntese de alto nível.

2.1 Projeto de circuitos integrados

O projeto de circuitos integrados se inicia com o desenvolvimento de um modelo funcional. Esse modelo funcional não contém detalhamento de hardware e pode ser um executável em uma linguagem de alto nível (`C`, `C++` e `SystemC` são as mais utilizadas para isso). A partir daí o resto do projeto depende do fluxo de *design* utilizado.

O fluxo de *design* mais utilizado pela indústria atualmente se baseia na implementação RTL do circuito usando uma linguagem HDL, como `VHDL`, `Verilog` ou `SystemC` e é mostrado na figura 2.1. Após desenvolvido e verificado, o modelo funcional deve ser transformado em um modelo em RTL.

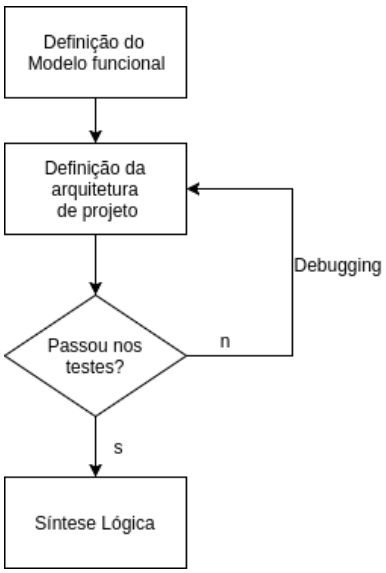

Figura 2.1: Fluxo tradicional de projeto de hardware

O caminho de descrição funcional até descrição RTL não é simples e passa por definir a arquitetura do sistema, que pode precisar ser ótima dependendo das particularidades do projeto. A definição da arquitetura do sistema é a maior fonte de erros em um projeto que utiliza esse fluxo de *design*. Após definir e implementar uma arquitetura a equipe de projeto entra na etapa de verificação de funcionamento (via testes) e correção de erros (*debugging*) que é cíclica e é o gargalo da maioria dos projetos.

A medida que as tecnologias de fabricação de CI's evoluem, circuitos ficam cada vez mais complexos e cada vez mais integrados. O aumento da complexidade dos circuitos faz com que a fase iterativa do fluxo apresentado (figura 2.1) seja cada vez mais demorada criando um gargalo inaceitável pela indústria — gerando custos elevados de projeto e ineficiência nos produtos. Fica claro que a indústria precisa adotar um novo fluxo de *design*.

2.2 Síntese no projeto de hardware

Não existe por enquanto um fluxo de *design* de hardware adotado por padrão na indústria que não seja baseado no nível RTL. Vários fluxos de *design* diferentes estão surgindo. O `SystemC`, por exemplo, ataca o problema de projeto de hardware permitindo, na mesmo código, implementações de sistemas (possivelmente comunicantes) em diferentes níveis de abstração - desde o sistêmico e algorítmico até o RTL. Apesar de integrar diferentes níveis de abstração, transformar uma descrição algorítmica para uma em RTL continua sendo um gargalo de desenvolvimento e uma fonte de erros.

Um fluxo de *design* alternativo baseado no "compilador de silício", em específico, é relevante para este projeto (mostrado na figura 2.2). Um compilador de silício é um sistema que tem como entrada um algoritmo descrito em alto nível e tem como saída um circuito digital em seu nível mais baixo. Todos os processos intermediários são automatizados por sistemas de síntese [9].

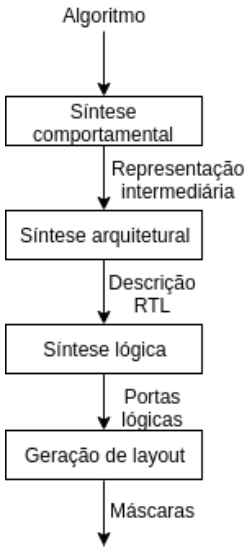

Figura 2.2: Fluxo do compilador de silício

O compilador de silício completo, funcional e performático ainda não existe. Apesar disso, partes do fluxo, como síntese lógica e síntese arquitetural são muito relevantes. As sínteses arquitetural e comportamental usando descrições puramente funcionais, que são o tema deste trabalho, prometem automatizar a etapa de transformação de descrição de alto nível para RTL facilitando muito o fluxo de projeto de circuitos integrados. A síntese lógica gera circuitos otimizados ao nível de portas lógicas melhorando, ainda mais, o resultado gerado pela síntese de RTL (mas esse não é o assunto desse trabalho).

2.3 Síntese de alto nível atualmente

Para que a indústria comece a utilizar ferramentas de síntese de alto nível como padrão e comece a modificar o fluxo de projeto de CI's é necessário que as ferramentas atuais obtenham melhores resultados ou que surjam ferramentas novas que modifiquem a maneira como a síntese é pensada.

Uma ferramenta de síntese de alto nível tem, idealmente, como entrada do usuário dois tipos diferentes de informação: a descrição funcional e restrições que são usados para guiar as otimizações da síntese. Esses dois tipos de informação devem ser suficientes para qualquer sistema de síntese.

As ferramentas atuais, no entanto, não produzem resultados de qualidade apenas com essas duas informações. A maioria das ferramentas provê, baseado em construções especiais, alternativas para o usuário interagir com o sistema de síntese, dando dicas de como sintetizar melhor o sistema. Conclui-se que a síntese de alto nível precisa principalmente de mais estudos e mais diversificação para que seja, de uma vez por todas, adotada como padrão pela indústria. Segundo [3]:

> Pesquisa adicional é vital, pois, ainda estamos longe da síntese de alto nível que pesquisa automaticamente o espaço de projeto sem precisar do direcionamento do projetista e entrega resultados ótimos para diferentes restrições do projeto e de tecnologia.

2.4 Definição de síntese de alto nível

A síntese de alto nível consiste em gerar uma representação RTL de uma arquitetura otimizada que implementa o mesmo comportamento da descrição alto nível que serve como entrada. A síntese de alto nível propicia para o usuário, como mostrado no fluxo 2.2, tanto a síntese comportamental quanto a arquitetural.

Ferramentas de síntese de alto nível mais comuns utilizam linguagens tipo-C (`C`,`C++`,`SystemC`) como entrada. A síntese de alto nível de linguagens tipo-C (o que pode ser estendido, até certo ponto, para linguagens imperativas) tem etapas e algoritmos relativamente conhecidos e detalhados pela literatura — principalmente por [1]. As partes mais importantes que constituem a síntese de alto nível de descrições imperativas estão mostradas na figura 2.3 e são, segundo [3] e [1]:

1) Compilar a especificação: Tem o objetivo de gerar um modelo formal do hardware, explicitando as dependências de dados e controle entre as operações

2) Alocar recursos de hardware (*allocation*): Unidades funcionais e componentes de armazenamento por exemplo

3) Escalonamento (*scheduling*): Fazer o escalonamento das operações em ciclos de *clock*

4) Transformar operações em unidades funcionais (*binding*): Que podem ser reutilizadas pelo sistema

5) Transformar variáveis em elementos de armazenamento: Variáveis são geralmente interpretadas como registradores (*placeholders* de informação) em uma máquina de estados (FMEA)

6) Gerar a arquitetura RTL

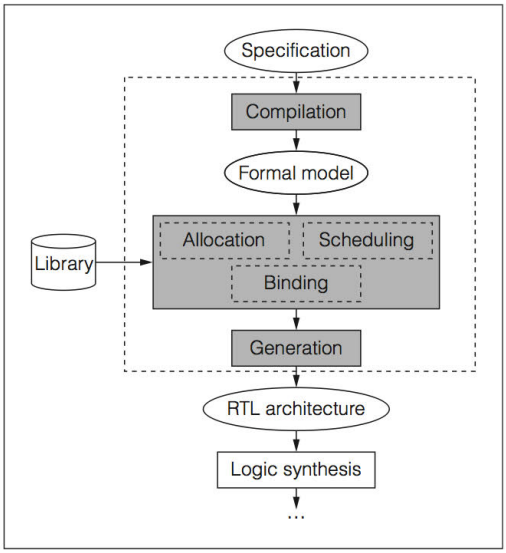

Figura 2.3: Síntese de descrições imperativas. Fonte: [1]

O algoritmo que sintetiza programas funcionais é completamente diferente do algoritmo que sintetiza programas imperativos. As etapas de alocação de recursos e de escalonamento não são essenciais na síntese de programas funcionais porém podem ser modificadas e implementadas no final do processo de síntese como otimizações.

A etapa de `3)` não é necessária pois a própria definição da função, por não ser sequencial, já define um "escalonamento" natural. Como as conexões entre funções são interpretadas com base em filas (como será visto no capítulo 4) não existe necessidade de definir um ciclo específico para cada operação/função como a etapa de escalonamento costuma fazer.

Como programação funcional não utiliza variáveis mutáveis a etapa `5)` não é utilizada (elementos de armazenamento apenas são gerados ao sintetizar na análise de funções recursivas). O algoritmo de síntese de programas funcionais é mostrado no capítulo 4.

2.5 Modelagem em SystemC

O `SystemC` é uma biblioteca do `C++` utilizada para *design* de hardware. O objetivo do SystemC é facilitar o fluxo de *design* de hardware integrando, em um só ambiente, diferentes níveis de abstração desde o sistêmico e algorítmico até o RTL [4].

O fluxo de *design* do `SystemC` [3][4] é baseado em criar sistemas no nível comportamental e ir diminuindo o nível de abstração do sistema:

1. Dividindo o sistema em módulos e implementando suas conexões
2. Definindo os módulos menores no nível comportamental, em alto nível
3. Testando e corrigindo os erros do sistema até então usando simulações
4. Repetindo esse processo para todos os módulos menores do sistema até se chegar no nível RTL

Com certeza o fluxo de *design* do `SystemC` funciona melhor que o fluxo de *design* tradicional, pois a transição do nível comportamental para o nível RTL é gradual e feita em menores escalas. O fluxo que utiliza síntese de alto nível, no entanto, é mais ideal apesar de ser mais difícil de ser implementado.

2.5.1 SystemC como linguagem alvo

O `SystemC` é a linguagem alvo escolhida para o sistema de síntese implementado neste projeto. Poderiam ter sido escolhidas quaisquer linguagens HDL como `VHDL` ou `Verilog`. A vantagem de usar o `SystemC` como linguagem alvo é a mesma de usá-lo em um projeto de hardware: a integração de diferentes níveis de abstração.

A síntese, como será visto no capítulo 4, usa extensivamente filas para conectar módulos e utiliza código em um nível misto (entre comportamental e arquitetural) para implementar o controle dos módulos. Desta forma a síntese pode focar o detalhamento arquitetural das partes mais importantes do circuito enquanto define partes menos importantes no nível comportamental. A implementação do sistema de síntese completamente a nível arquitetural teria que definir filas com múltiplas entradas e saídas de comunicação além de definir complexos controles como máquinas de estados — tudo operando de maneira correta à nível de *clock*.

Capítulo 3

Linguagens Funcionais

Breve introdução sobre linguagens puramente funcionais e a linguagem Haskell.

3.1 Comparação entre linguagens funcionais e imperativas

Linguagens funcionais são chamadas assim pois sua operação fundamental é a aplicação de funções em seus argumentos. Um programa é definido como uma função que recebe seus valores de entrada como argumentos e retorna sua saída como resultado. Uma função é tipicamente definida em termos de outras funções até chegar em funções primitivas da linguagem [10].

Linguagens puramente funcionais descrevem seus programas usando exclusivamente funções. Isso implica na não utilização de ferramentas de controle de fluxo, como `for` e `while`, e de ferramentas que modificam o estado implícito do programa, como a modificação do valor de variáveis por *assignment*. Todo o programa é definido com base em funções, utilizando recursividade e passagem de argumentos ao invés de controle de fluxo e mudanças de estado.

Algumas vantagens das linguagens puramente funcionais podem ser examinadas em comparação com as características das linguagens imperativas. A ausência de *assignment* — todas as variáveis são constantes — e a ausência de *side-effects* — ao aplicar um argumento a uma função o único resultado que interessa é o retorno da função — criam a chamada *transparência referencial*, que faz com que a ordem de execução dos programas seja irrelevante — aplicações de funções podem ser avaliadas a qualquer ordem — e tira a responsabilidade do programador de definir qualquer tipo de controle de fluxo.

Como expressões podem ser avaliadas a qualquer momento, valores também podem ser substituídos pelas expressões que os geraram. Isso faz com que programas funcionais sejam mais facilmente analisáveis matematicamente. Do ponto de vista de síntese de hardware, a *transparência referencial* faz com que a síntese não necessite de etapa de *scheduling* de operações (como geralmente necessita em síntese de linguagens imperativas) [10].

3.2 Haskell

A linguagem utilizada para servir de entrada para o sistema de síntese desse projeto é um *subset* da linguagem `Haskell`. `Haskell` é uma linguagem de programação puramente funcional que se distingue das outras linguagens puramente funcionais por ter um sistema de tipos forte e ao mesmo tempo expressivo, fazendo com que a maioria dos erros sejam pegos no tempo de compilação. Um sistema de tipos forte é ideal para a síntese, visto que todos os erros devem ser pegos no tempo de compilação (erros dinâmicos não são uma possibilidade).

A partir daqui serão introduzidos alguns conceitos importantes para a compreensão da síntese da linguagem `Haskell` e de linguagens funcionais tipadas em geral.

3.2.1 Definição e aplicação de funções

A sintaxe para definição e aplicação de funções é baseada em espaços em branco e parênteses somente quando necessário. A definição de uma função é seu nome (`f`) seguido de seus argumentos (`x` e `y`) separados por espaços em branco até chegar no símbolo de igualdade (`=`) que indica o início do corpo da função (como mostrado abaixo).

O corpo de uma função é uma expressão que define o seu retorno. Expressões podem ser um valor (constante ou variável) ou uma aplicação de função. Sintaticamente, aplicar uma função é escrever seu nome seguido de seus parâmetro(s) de entrada. Abaixo vemos o exemplo da aplicação da função `add` aos argumentos `x` e `y` na definição da função `f` com parâmetros de entrada `x` e `y`.

Algumas funções especiais podem utilizar símbolos ao invés de nomes. Neste caso `f` poderia ser definida como:

Em aplicações aninhadas é necessário o uso de parênteses nos argumentos. Os argumentos, no entanto, continuam separados por espaços em branco. No exemplo abaixo a função `g` recebe como argumento a variável `y` e o resultado da expressão `k x (h y)`, que é a função `k` sendo aplicada aos argumentos `x` e ao resultado da aplicação de `h` na variável `y`.

```
f x y = g y (k x (h y))
```

3.2.2 Currying

Aplicação parcial é quando uma função é definida com 2 ou mais entradas (como a função `g` abaixo) porém a aplicação não dá todos argumentos necessários (como na função `add5` abaixo). Na maioria das linguagens isso geraria um erro. Em `Haskell`, por causa do chamado *currying*, aplicações parciais definem novas funções como se o parâmetro de entrada fosse substituído pelo argumento. Por isso, no exemplo abaixo, a aplicação `add5 3` é reduzida à `8`.

```
g x y = x + y
add5 = g 5
-- add5 3 ⇒ 8
```

Currying será usado para definir algumas funções analisadas nos capítulos 5 e 4 apesar de que *currying* não foi implementado no sistema de síntese.

3.2.3 Tipos de funções

Todos os valores, incluindo funções, têm tipos.

3.2.4 Tipos de dados

`Haskell` baseia suas estruturas de dados em tipos chamados de ADT's (*Algebraic Data Types*), que são a união entre os tipos soma e tipo produto.

Tipos soma são tipos que indicam várias possibilidades. O valor de um tipo soma pode ser qualquer um dos construtores definidos. O tipo mais famoso de tipo soma é o `Bool`, que pode ser `True` ou `False`, mas existem vários exemplos, como o tipo `Color`, mostrado abaixo.

```
data Bool = True | False
data Color = Red | Blue | Green | Brown | Purple
```

Tipos produto são tipos com vários tipos dentro. Sua definição contém um construtor (assim como os tipos soma), seguido de um, ou vários, tipos internos ao tipo produto. Uma pessoa, por exemplo, poderia ser expressada via tipos produtos. O exemplo da definição do tipo `Pessoa` é mostrado abaixo. O termo `Pessoa` antes da igualdade é o nome do tipo e o termo `Pessoa` após a igualdade é o construtor.

```
type Nome = String
type Idade = Int
data Sexo = M | F
data Pessoa = Pessoa Nome Idade Sexo
```

Tipos ADT são a junção entre as idéias de tipos produto e tipos soma. Unificando a idéia de construtores de tipos, pode-se criar tipos soma de produtos (algébricos) onde existem vários casos (soma), cada um com várias informações (produto). Um exemplo de tipo ADT que indica tipos diferentes de geometrias é mostrado abaixo.

```
data Geom = Retangulo Int Int
          | Triangulo Int Int
          | Circulo Int
```

3.2.5 Pattern Matching

Pattern matching é uma técnica geralmente utilizada em conjunto com tipos ADT para definir uma função de maneira concisa "abrindo" o tipo.

A função `and`, mostrada abaixo, usa *pattern matching* com tipos soma para definir retornos diferentes de acordo com o valor de entrada. A função `area`, também mostrada abaixo, usa *pattern matching* assim como a função `not` além de abrir a definição para acessar os tipos internos.

```
and :: Bool → Bool → Bool
and True  True  = True
and True  False = False
and False True  = False
and False False = False

area :: Geom → Int
area (Retangulo b h) =  b * h
area (Triangulo b h) = (b * h)/2
area (Circulo r) = pi * r ^ 2
```

Pattern matching é uma técnica amplamente utilizada em Haskell e no *subset* utilizado nesse trabalho para síntese.

3.2.6 Guards

O *subset* de `Haskell` utilizado por esse projeto não têm expressões `if` nem expressões `case...of`. Para definir uma função condicional o `subset` deve usar *pattern matching* ou *guards*.

Guards são uma maneira de criar condicionais com expressões que retornam `Bool`s. Abaixo é mostrado o exemplo da função fatorial, escrita com *guards*. Caso `n == 0` for `True`, o resultado é `1`, caso contrário é `n * fac (n-1)`.

```
fac :: Int → Int
fac n
  | n == 0 = 1
  | otherwise = n * fac (n-1)
```

Condicionais booleanos só são posíveis com *guards* na linguagem *subset* deste projeto. O capítulo 4 mostra que a linguagem core do sistema é baseada em *guards* pois todo o *pattern matching* é transformado em *guards*.

3.2.7 Síntese de programas funcionais

Programas funcionais são baseados em aplicações de funções. Funções são definidas em termos de outras aplicações até chegar em funções primitivas.

Sintetizar aplicação de funções, funções condicionais, funções recursivas e funções com tipos recursivos é o suficiente para poder sintetizar um gama muito útil de programas funcionais.

O capítulo 4 focará no sistema proposto de síntese e manipulará os elementos mostrados neste capítulo como aplicação e definição de funções, *pattern matching*, *guards*, etc.

Para sintetizar um programa funcional as aplicações são sintetizadas como conexões com FIFOs e as recursividades são sintetizadas como máquinas de estados. Os problemas mais complexos aparecem quando tenta-se sintetizar tipos recursivos, que são interpretados como tipos sequenciais no tempo.

Capítulo 4

Sistema Proposto

Detalhamento do sistema de alto nível proposto: suas entradas, suas saídas e como funciona.

4.1 Descrição

O sistema de síntese de alto nível proposto e implementado por esse trabalho tem como entrada uma descrição de alto nível escrita em uma linguagem puramente funcional que é um subconjunto da linguagem `Haskell` e tem como saída um código `SystemC` que representa a arquitetura resultante da síntese.

O sistema difere em alguns aspectos quando comparado à outros sistemas de síntese propostos para linguagens funcionais. Em comparação com `Lava` [5], que é uma das primeiras tentativas de síntese de hardware utilizando linguagens funcionais, a linguagem de entrada não detalha o nível arquitetural sendo utilizada apenas para especificar a funcionalidade do hardware em alto nível. Diferentemente do compilador `CλaSH` [6] a síntese não é principalmente baseada em reescritas sucessivas das funções até chegar em um código funcional de mais baixo nível. Ao invés disso, o algoritmo de síntese se baseia na coleta de características da função, aplicando reescritas somente quando necessário para coletar mais informações.

Funções recursivas são interpretadas como elementos combinacionais que consomem e geram fluxos de dados armazenados em FIFO's em um modelo semelhante ao *dataflow* (o modelo gerado pode ser visto como uma mistura entre *dataflow* e máquinas de estado) assim como proposto em [8] e [7]. Diferentemente do proposto por [8] e [7], no entanto, o sistema proposto não retira a recursão das funções: a recursão permanece e serve como fonte de informações que auxiliam a síntese.

4.1.1 Linguagem de entrada

4.1.1.1 Características e BNF

Abaixo são listadas algumas características de destaque da linguagem de entrada - que é um subconjunto de `Haskell`:

Puramente funcional: Programas são baseados em aplicações de funções. Estados intermediários, variáveis mutáveis e *side effects* são menos importantes para o programador, que pode ter uma visão mais alto nível de seu programa.

Fortemente tipada: uma linguagem construída em cima de um sistema de tipos é capaz de detectar erros em tempo de compilação. Isso é importante para síntese de hardware pois nesse caso todos erros devem ser, idealmente, detectados antes de usar o hardware.

Definição de *Algebraic Data Types* (ADTs): são tipos "soma de produto"(vistos no capítulo 3) que em conjunto com *pattern matching* simplificam a definição de funções.

Uso de guards em funções: estruturas condicionais primitivas que permitem à função escolher que expressão rodar dependendo do valor das entradas.

Várias construções e funcionalidades da linguagem `Haskell` não foram implementadas por não estarem no foco do projeto ou por não serem relevantes para síntese de alto nível. Entre as ausências mais significativas estão a falta das construções `let` e `where`, que possibilitam a definição de variáveis dentro do escopo de visão de uma única função, a falta do *currying*, ou aplicação parcial de argumentos (como explicado no capítulo 3) e a impossibilidade de definição de funções polimórficas (que atuam em vários tipos).

Dentro das ausências apresentadas é importante citar a importância de *pattern matching* e *guards* pois essas duas construções substituem as expressões `if...then...else...` e `case...of...`, que também estão ausentes no subconjunto de entrada.

Neste subconjunto é possível a definição (e aplicação) de funções:

- recursivas
- que operam em tipos ADT (recursivos ou não)
- de alta ordem (ter como argumento)
- que combinam todos os aspectos acima

Funções primitivas são as aritméticas (`+`,`-`,`*`,`/`), as lógicas (`and`,`or`,`not`) e a igualdade (que pode ser usada em quaisquer dois tipos que não sejam recursivos).

Os tipos primitivos são o inteiro `Int`, o número de ponto fixo `Fixed` e a lista `List`. Tanto o `Int` quanto o `Fixed` têm, por *default* 32 bits mas podem ter como argumento opcional um número que indica a quantidade de bits (exemplo: `Int 64`, `Fixed 85`) e a `List` sempre recebe como argumento o tipo dos valores internos da lista (exemplo: `List Int`, `List Bool`).

O tipo `List` foi escolhido como primitivo visto o uso extensivo dele em programação funcional e sua utilidade em um grande número de funções. É possível, no entanto, que o programador defina um tipo semelhante ao tipo `List` em seus códigos e o resultado para o sistema de síntese seria o mesmo (a diferença é que o tipo definido pelo programador não poderia ser polimórfico em

relação ao tipo do elemento da lista). Além disso, funções que tem como entrada e/ou saída tipos `List` geralmente implementam hardwares úteis em processamento de dados como será mostrado adiante.

Apesar de ser um pequeno subconjunto de `Haskell`, a linguagem de entrada é bastante expressiva — sendo possível a declaração de uma variedade grande de funções. Isso é devido ao caráter minimalista de descrições puramente funcionais: poucas construções carregam muito poder de expressividade. Por completude, a BNF da linguagem é mostrada nos anexos.

4.2 O método de síntese

Do programa funcional até a saída em `SystemC` o sistema é composto por diversas etapas (figura 4.1).

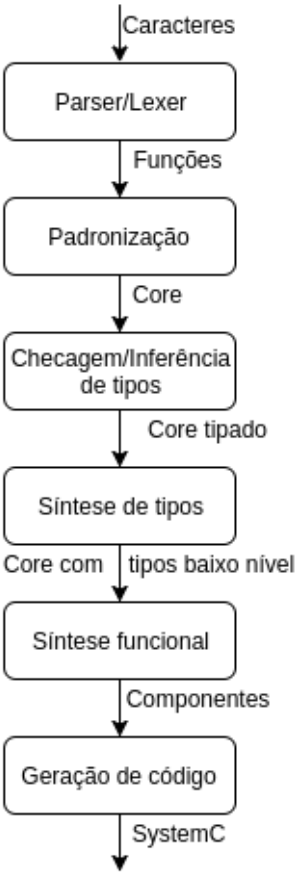

Figura 4.1: Fluxo tradicional de projeto de hardware

A entrada do programa começa com um texto, uma cadeia de caracteres, que é transformado em uma lista de *Tokens* pelo *Lexer* que é transformado em expressões pelo Parser. Essas expressões são então interpretadas para formar um tipo de dado interno que descreve uma função. A função então passa pelo processo de padronização que transforma a linguagem na chamada linguagem Core, que é mais simples de analisar e transformar.

A linguagem Core então passa por um processo de checagem e inferência de tipos que tem como objetivo descobrir os tipos de todas as expressões do programa. Caso haja algum tipo de *mismatch* entre tipos um erro de tipagem é gerado. Essa etapa gera o chamado Core tipado.

O Core tipado é a entrada para o sistema de síntese. A primeira etapa da síntese propriamente dita é a síntese de tipos, que transforma todos os tipos de alto nível — definidos ou não pelo programados — em tipos de baixo nível, mais próximos do hardware. Além disso, a transformação do nível dos tipos também requer uma transformação das funções (que operam estes tipos). O resultado desta etapa gera uma linguagem ainda recursiva, porém com uma gama reduzida de tipos.

O Core com tipos em baixo nível é a entrada para a etapa de síntese funcional — a principal do sistema — que transforma funções em componentes. Componentes são módulos de hardware que podem ser transformados facilmente em `SystemC` na etapa final.

4.2.1 Lexer e Parser

A etapa inicial do sistema é a transformação de caracteres em *tokens*, feita pelo *Lexer*, e a transformação de *tokens* em expressões, realizada pelo Parser. As expressões de saída do parser podem representar declarações de funções, do tipo das funções ou de tipos de dado. A figura 4.2 demonstra esse graficamente esta etapa.

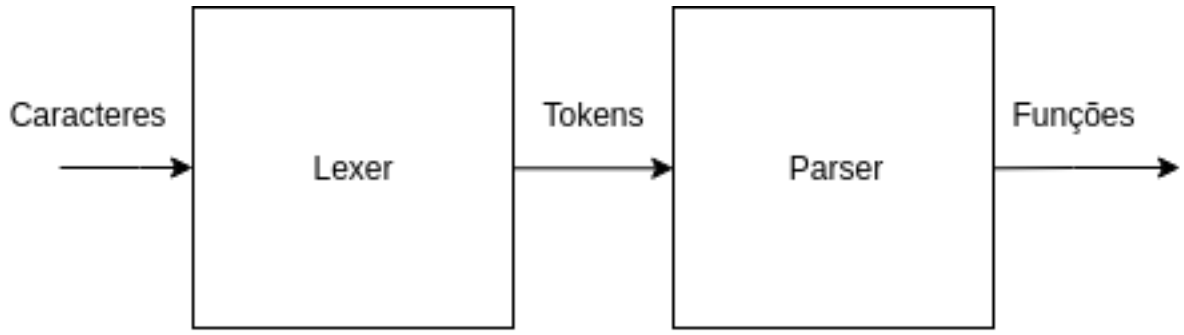

Figura 4.2: Parsing esquematizado

4.2.2 Padronização

A padronização é a etapa que tira o açúcar sintático existente na linguagem de entrada e padroniza a maneira como as funções são definidas. Uma linguagem padronizada é mais simples de ser trabalhada, transformada e analisada.

A maior diferença entre a linguagem core e a linguagem de entrada é a retirada do *Pattern Matching*. O *pattern matching* é muito utilizado em programação funcional e especialmente na linguagem *Haskell*. Grande parte da simplicidade em usar tipos ADT's vem da prática de *pattern matching* (como mostrado no capítulo 3).

Um exemplo é a função `map`. A função `map` é comumente definida fazendo *pattern matching*

nos dois casos possíveis de uma lista: o caso de lista vazia (`Nil`) ou de um valor seguido de outra lista (`Cons`). A maneira como isso é feito em pseudo-`Haskell` é mostrada abaixo:

```
-- map com pattern matching
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

A etapa de padronização transforma *pattern matching* em aplicações de funções e condições (usando *guards*). Ao invés de "abrir" o dado para descobrir se é `Nil` ou `Cons`, funções especiais `is__Nil` e `is__Cons`, que tem como entrada elementos do tipo `List`, podem retornar um `Bool` (`True` ou `False`) caso o caso se aplique. O resultado da padronização da função `map` é mostrada abaixo:

```
-- map sem pattern matching
map f s
  | is__Nil s = Nil
  | is__Cons s = Cons (f x) (map f xs)
```

Todas as funções, após esta etapa, têm a mesma estrutura mostrada abaixo (onde `cond's` representam expressões condicionais e `expr's` representam expressões de retorno da função):

```
func a1 a2 ... an
  | cond1 a1 a2 ... an = expr1 a1 a2 ... an
  | cond2 a1 a2 ... an = expr2 a1 a2 ... an
  ...
  | condn a1 a2 ... an = exprn a1 a2 ... an
```

4.2.3 Checagem e inferência de tipos

A inferência e checagem de tipos tem como objetivo definir os tipos de todas as expressões e variáveis do programa.

A parte da inferência descobre o tipo de qualquer elemento polimórfico da linguagem. Elementos polimórficos incluem números literais e algumas funções especiais que podem atuar em mais de um tipo diferentes. A lista mostrada a seguir mostra todos os elementos polimórficos do sistema e seus tipos, onde, por exemplo, o tipo do `(+)` é lido como "para todo tipo 'a' que seja um número, a função recebe dois elementos do tipo 'a' e retorna outro elemento do mesmo tipo 'a"'.

```
Nil :: List a
Cons :: a → List a → List a
(==) :: (NotRec a) ⇒ a → a → Bool
(+) :: (Num a) ⇒ a → a → a
(-) :: (Num a) ⇒ a → a → a
(*) :: (Num a) ⇒ a → a → a
```

A parte da checagem de tipos verifica se as aplicações de funções respeitam os tipos da função e, se esse não for o caso, ela gera um erro de tipagem. Um exemplo de erro de tipagem é tentar

somar um `Int` com um `Fixed`: apesar de ambos serem numéricos, segundo o tipo de `(+)` os tipos de entrada (e de saída) da função adição devem ser iguais (pois todos são 'a').

A inferência e verificação de tipos ocorrem ao mesmo tempo, em um mesmo algoritmo. Esse algoritmo se baseia, brevemente, em pegar os tipos de todas as funções previamente e caminhar por todas as aplicações verificando, e inferindo ao mesmo tempo, se os tipos batem.

Formalmente, o sistema de tipos implementado é uma versão do sistema do tipos Hindley-Milner [11] (também utilizado na linguagem `Haskell`), que tem como característica especial a inferência de tipos.

4.2.4 Síntese de tipos

A síntese de tipos é considerada a primeira etapa da síntese. Até então, as etapas poderiam servir também para geração de software.

A síntese de tipos consiste em transformar tipos de alto nível, abstratos em tipos mais próximos do hardware. Os tipos mais próximos do hardware em questão são bits (`Bit`), vetores de n bits (`Vec n`) e um tipo fluxo, (`Stream`) que indica que a máquina a ser sintetizada receberá a entrada estendida no tempo ao invés de recebê-la toda de uma só vez. Streams de streams (`Stream (Stream x)`) não fazem sentido neste contexto. Abaixo é apresentada a definição formal dos tipos baixo nível:

```
type : Bit
     | Vec nat
     | Stream type
```

O tipo `Stream Bit`, por exemplo, indica que a máquina receberá vários `Bit`'s seguidos no tempo enquanto o tipo `Stream (Vec 5)` indica o recebimento de vários vetores de 5 bits em sequência temporal e o tipo `Vec 2` indica o recebimento, em uma única vez, de 2 bits (após esse recebimento a entrada já foi completamente consumida).

4.2.4.1 Síntese de tipos primitivos

Tipos primitivos da linguagem, como `Int` e `Fixed`, são sintetizados de maneira especial. Simplesmente os dois são transformados em `Vec n` dependendo de qual for o numero de bits indicado pelo programador (caso não determinado, `n = 32`).

Os tipos `List` e `Bool` apesar de serem primitivos, não recebem tratamento especial na síntese de tipos. Ambos são sintetizados como se tivessem sido definidos pelo programador.

4.2.4.2 Síntese de tipos produto

Tipos produto são tipos que contém, dentro de si, informações de outros tipos. Um exemplo é o tipo `Retangulo` mostrado na imagem abaixo.

```
data Retangulo = Retangulo Int Int
```

É comum realizar *pattern matching* em tipos produto para extrair as informações de dentro deles, como na função `area` definida abaixo.

```
area :: Retangulo → Int
area (Retangulo b h) = b * h
```

Um tipos produto é sintetizado, na implementação atual, como a concatenação do resultado da síntese de todos os tipos contidos nele. Caso os `Int`'s do tipo `Retangulo` sejam sintetizados como `Vec 32`'s, o tipo `Retangulo` será sintetizado como um `Vec (32+32) = Vec 64` pois esse é o resultado da concatenação dos dois tipos contidos nele. A representação gráfica da função `area` é apresentada na figura 4.3.

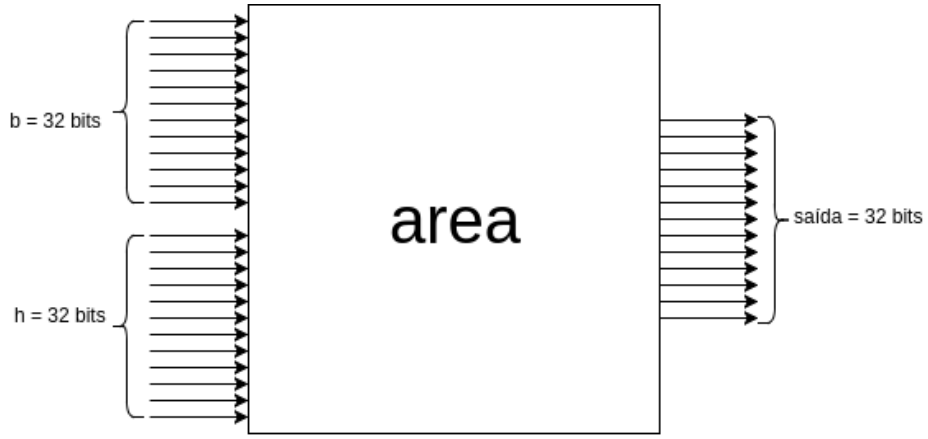

Figura 4.3: Portas da máquina area

Outra maneira de sintetizar tipos produto, que não é utilizada na implementação atual, é fazer com que cada elemento seja entregado à máquina em sequência temporal. Assim, o tipo `Retangulo` teria 32 bits e a máquina teria que ter um mínimo de controle para saber se a entrada atual é o primeiro ou o segundo valor.

4.2.4.3 Síntese de tipos soma

Tipos soma são tipos que indicam várias possibilidades. O exemplo mais clássico de tipo soma é o booleano (`Bool`).

```
data Bool = True | False
```

Uma função que recebe um booleano pode receber tanto o valor `True` quanto o valor `False`. Um exemplo mais extenso de tipo soma é o tipo `Color`, que retrata cores diversas.

```
data Color = Red | Blue | Green | Brown | Purple
```

Uma função que recebe tipos soma geralmente utilizam da técnica do *pattern matching* para elevar a expressividade da função.

```
mix :: Color → Color
mix Red Red = Red
mix Red Blue = Purple
...
```

Tipos soma são sintetizados como vetores de bits do tamanho necessário para que se possa representar todos os elementos do tipo. O tipo `Bool`, por exemplo, necessita apenas de um bit para ser completamente representado enquanto o tipo `Color` necessita de, no mínimo, 3 bits (por ter 5 construtores). A figura 4.4 mostra uma representação possível para o tipo `Color`.

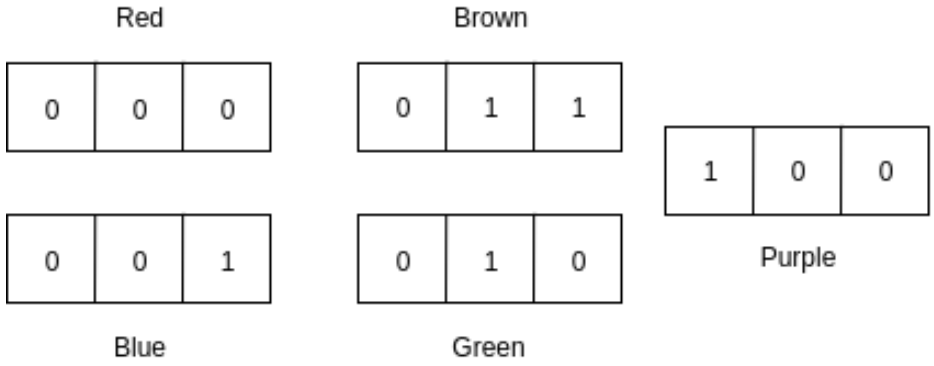

Figura 4.4: Síntese do tipo Color

4.2.4.4 Síntese de ADT's

Tipos de dado algébricos (ADT's, em inglês) são a junção entre os conceitos de tipos soma e tipos produto. Tipos algébricos são tipos soma de produtos, onde um tipo tem várias possibilidades (soma) contendo cada uma delas outros tipos integrantes (produto). Um exemplo de ADT é a própria definição de lista `List`. Outro exemplo também seria o tipo `Geom`, que representa as dimensões de diferentes tipos geométricos.

```
data List a = Nil
            | Cons a (List a)

data Geom = Retangulo Int Int
          | Triangulo Int Int
          | Circulo Int
```

A síntese de uma ADT é a junção entre a síntese de um tipo soma e de um tipo produto. O vetor binário resultante terá uma parte de síntese da soma concatenada com uma parte de síntese do produto. O tamanho final do tipo será o maior tamanho entre o tamanho resultante para todos os casos (em muitos casos ocorrem bits inutilizados). Um exemplo da síntese do tipo `Geom` é mostrada na figura 4.5.

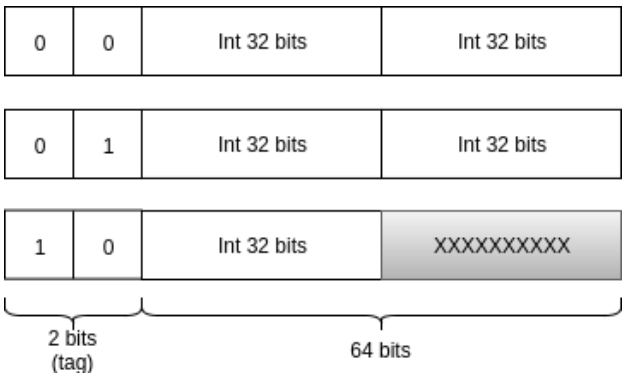

Figura 4.5: Síntese do tipo Geom

A síntese de tipos recursivos segue o mesmo princípio da síntese de ADT's comuns, porém ao invés de sintetizar um tipo vetor, sintetiza-se um tipo *stream* de vetores de maneira que indica que a máquina receberá vários vetores em sequência temporal. O vetor binário, resultante da síntese de uma lista `List` é apresentada na figura 4.6.

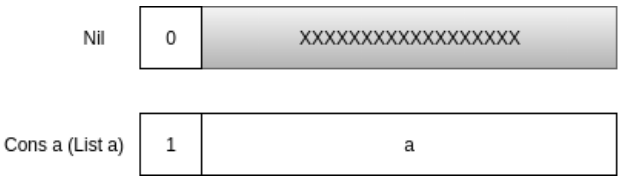

Figura 4.6: Síntese do tipo Lista

4.2.5 Síntese de funções

A síntese de funções é a etapa mais importante do sistema, pois ela gera a arquitetura do resultado. A saída desta etapa são componentes. Um componente é um dado que representa um módulo da arquitetura resultante.

O maior objetivo nesta etapa é definição dos módulos, das conexões e do processamento interno de cada módulo. O algoritmo de síntese funcional parte do princípio de que, ao sintetizar uma função, todas as dependências (outras funções utilizadas em sua definição) já foram sintetizadas e, logo, já existe um componente que as representa.

O algoritmo de síntese começa, por tanto, sintetizando todas as dependências de uma função. Como a maioria das funções são definidas em termos de outras funções esse processo recursivo vai parar quando for achada uma função terminal (uma função definida sem o auxilio de outra função) ou uma função primitiva do sistema. Funções primitivas, como as aritméticas, já tem módulo pré-definido. Funções terminais devem passar pelo processo de síntese comum.

Antes de sintetizar uma função, o algoritmo a classifica de acordo com características importantes. A classificação da função define como ela será analisada e sintetizada. As classificações de cada função são:

Não recursivas Funções não recursivas e sem tipos *streams* são geralmente funções combinacionais (aplicação de outras funções apenas) ou funções terminais.

Não recursivas contendo tipo Stream Apesar de não terem recursão a análise é dificultada pelo fato de terem como argumentos ou saída elementos de tipo `Stream`. Tirando esse detalhe, a análise é similar a de funções não recursivas comuns.

Recursivas Funções recursivas são sintetizadas como máquinas de estado finito apesar de que nem toda recursão pode ser sintetizada pelo sistema atual.

Recursivas contendo tipo Stream Funções recursivas com tipos *streams* são sintetizadas analisando o uso, a cada passo, dos elementos das *streams* de entrada em relação com como a função gera a *stream* de saída.

4.2.5.1 Funções sem recursão

Funções sem recursão são normalmente sintetizadas como módulos de maneira intuitiva. Analisando cada aplicação que compõe uma função é possível criar conexões entre os módulos. Uma função sem recursão e sem *guards* (condições) é chamada de função combinacional, pois é baseada apenas na combinação de outras funções já existentes. Um exemplo de função combinacional e sua respectiva síntese é mostrada abaixo (figura 4.7).

```
f x y = g y (k x (h y))
```

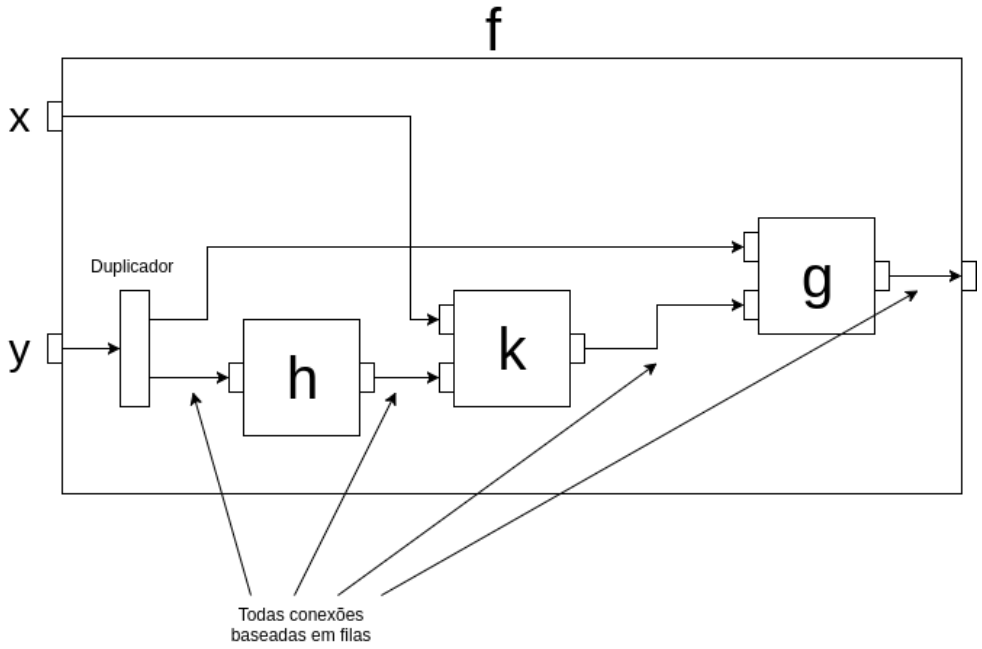

Figura 4.7: Função combinacional

Funções combinacionais podem ser definidas apenas por conexões entre módulos. Funções condicionais como a da figura 4.8, no entanto, necessitam, além das conexões, de um controle

adicional para que o funcionamento da máquina ocorra de maneira correta. Os valores de entrada devem primeiramente entrar em todos módulos condicionais (definidos pelas expressões `cond`) e depois entrar em um dos módulos de expressão (definidos por uma das expressões `expr`) cujo resultado será a saída do módulo. Em suma, a parte de controle é responsável por:

1. Colocar as entradas nas máquinas condicionais
2. Coletar o resultado das máquinas condicionais que define o módulo correto para ser ativado
3. Colocar as entradas no módulo correto
4. Coletar o resultado da máquina e colocar na saída do módulo

```
f x y
  | cond1 x y = expr1 x y
  | cond2 x y = expr2 x y
  ...
  | condn x y = exprn x y
```

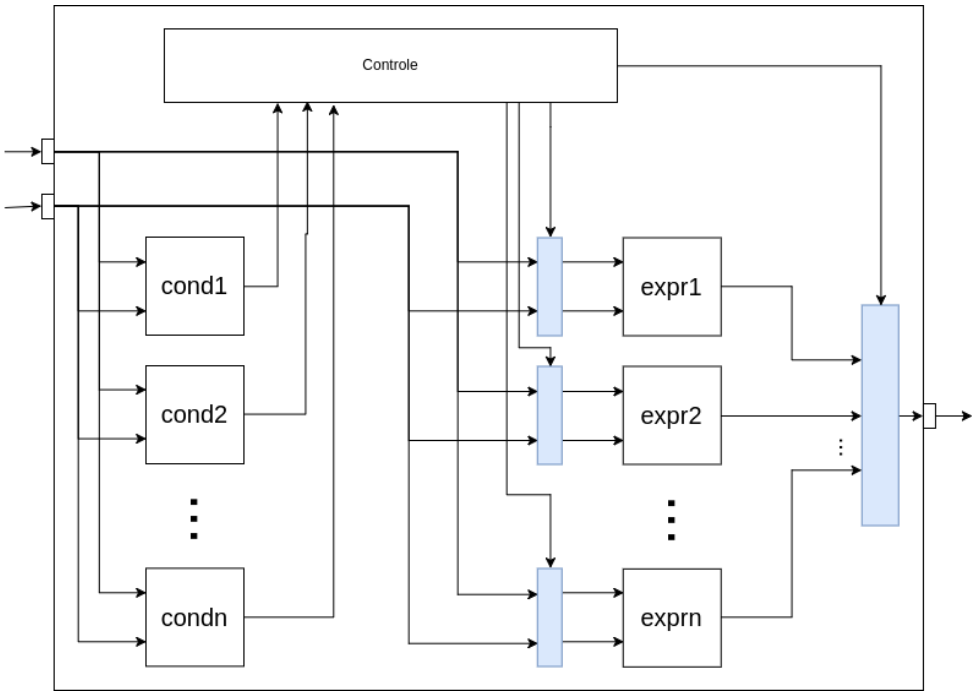

Figura 4.8: Função condicional

É importante deixar claro que conectar a entrada em todos os módulos `expr` e apenas colocar na saída o resultado da `expr` correta de acordo com o resultado dos módulos de condição não é possível pelo fato do sistema ser baseado em filas consumíveis. Como um mesmo valor de entrada só pode ser consumido uma vez, o controle é responsável por guardar os valores e suprir os módulos auxiliares (que representam tanto as expressões `cond` quanto as `expr` da figura 4.8). Graficamente (figura 4.8), o controlador é representado como um módulo que conecta outros módulos. Na seção 4.2.6 será visto que, em `SystemC`, o controle é implementado via `SC_THREAD`s.

4.2.5.2 Funções recursivas

A recursão é uma parte essencial da programação funcional além de ser uma forma simples de escrever definições concisas de funções.

Entre todos tipos de diferentes de definir uma função recursiva apenas funções simples com recursividade à esquerda são traduzidas facilmente para hardware (via inspeção). Os outros tipos de recursividade exigem que a função tenha acesso à memória (pilhas) e use de convenções de chamadas para executar a função.

Felizmente, como mostrado em [8], funções recursivas de quaisquer tipos podem ser transformadas em funções com recursividade simples à esquerda utilizando um método de reescrita que envolve tipos de dado recursivos. A partir deste fato, ao encontrar qualquer função recursiva, primeiro ela deve ser reescrita para a sua forma simples à esquerda, para depois ser sintetizada.

O sistema implementado é capaz de sintetizar funções com recursividade simples à esquerda e à direita (funções com recursividade à direita são primeiramente reescritas).

Recursão à direita

Uma função simples com recursão à direita é uma função que tem sua chamada recursiva à direita de outra função. Um exemplo de função à direita é a função fatorial.

```
fac 0 = 1
fac n = n * fac (n-1)
```

O algoritmo de transformação de recursão à direita para esquerda é baseado em inspeção e reescrita. O resultado, ao exemplo do fatorial é mostrado abaixo.

```
fac = facleft 1

facleft 0 a = a
facleft n a = facleft (n-1) (a*n)
```

Recursão à esquerda

Para sintetizar uma função com recursividade à esquerda, como por exemplo a função `facleft`, basta verificar quais são suas condições de parada, sua saída, seus estados e as funções de transição de cada estado. Todas essas informações podem ser encontradas na definição da função.

No caso da função `facleft` a função de transição para o estado `n` é `n-1` e a função de transição para o estado `a` é `a * n`. A saída do sistema é a variável de estado `a` e o sistema pára quando a variável `n` é igual à zero. Além disso a função `fac` define os estados iniciais do sistema. O resultado é graficamente demonstrado na figura 4.9.

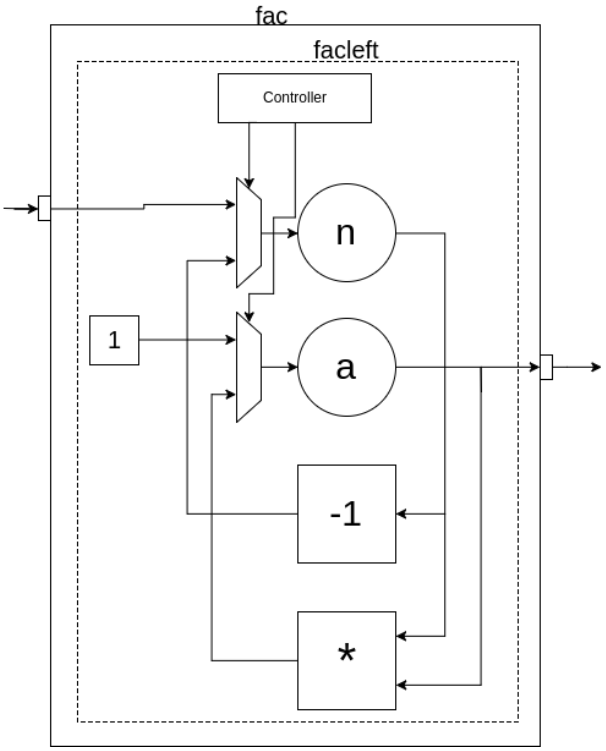

Figura 4.9: Implementação do fatorial

4.2.5.3 Funções recursivas com tipos Stream

A síntese de tipos converte listas em *streams*, funções com recursividade que atuam em listas como `map`, `zipWith` e `sum` fazem parte deste caso. Até certo ponto, a síntese de funções recursivas com *streams* é igual à síntese de funções recursivas comuns: os estados, funções de transição de estados, condição de parada e saída são analisados da definição. Funções que tem como saída `Stream`s podem colocar resultados na saída antes de acabar todo o processamento. Isso faz com que o que era analisado anteriormente como função de transição de estado vire também uma função de saída.

Um exemplo disso é a função `map` e sua versão recursiva à esquerda `mapleft` (mostradas abaixo). As funções de transição são `get__List__1 s` e `Cons (f x) a`. Sabendo que se tratam de variáveis de estado do tipo `Stream` o sistema é capaz de inferir que a transição `Cons (f x) a` define a saída da máquina (o valor `f x`) e que a transição `get__List__1 s` indica apenas a continuação da *stream* de entrada.

```
map f s = mapleft f s Nil

mapleft f Nil a = a
mapleft f (Cons x xs) a = mapleft f xs (Cons (f x) a)
```

A síntese do `map` não necessita de nenhuma variável de estado pois a função consome a lista de entrada no mesmo ritmo que gera a lista de saída. Quando uma função consome elementos num ritmo mais elevado são necessários guardar os valores anteriores da *stream* para o funcionamento

correto do componente. O mesmo tipo de análise é válida para funções com várias entradas de *streams* e para funções com apenas algumas das entradas do tipo `Stream`. O resultado da síntese da função `map`, graficamente representado, é mostrado na figura 4.10.

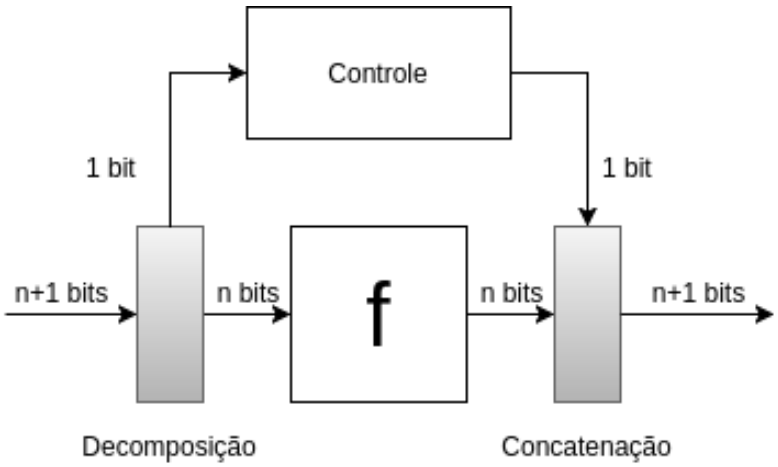

Figura 4.10: Função map

Outros exemplos são a função `zipWith` e a função `sum`. A `zipWith` é similar ao `map` porém consome duas listas ao mesmo tempo usando uma função de duas entradas (resultado da síntese na figura 4.11). A função `sum` soma todos os elementos da lista (resultado da síntese na figura 4.12). Quando o controle da máquina `sum` detecta o fim da lista, o valor da variável de estado `a` é colocado na saída — este processo é representado pelo bloco em azul, conectado ao controle.

```
zipWith f [] ys = []
zipWith f xs [] = []
zipWith f (Cons x xs) (Cons y ys) = Cons (f x y) (zipWith f xs ys)
```

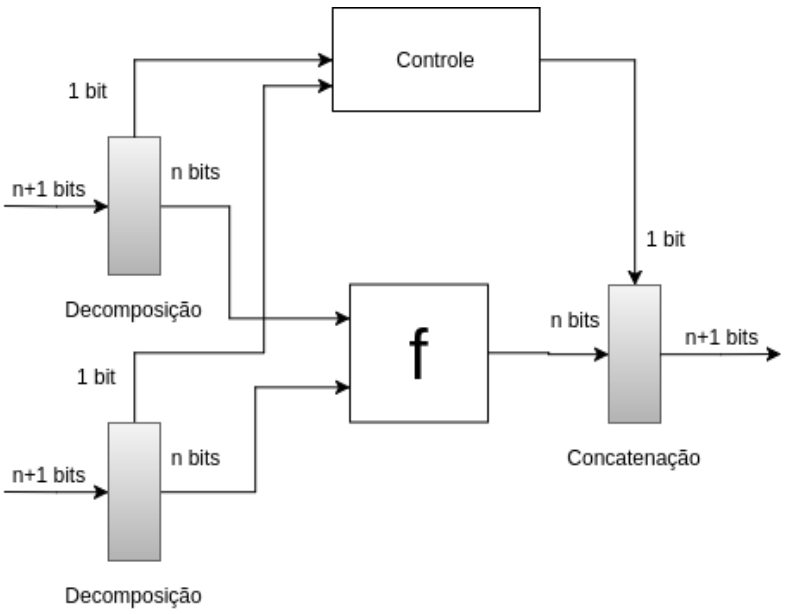

Figura 4.11: Função zipWith

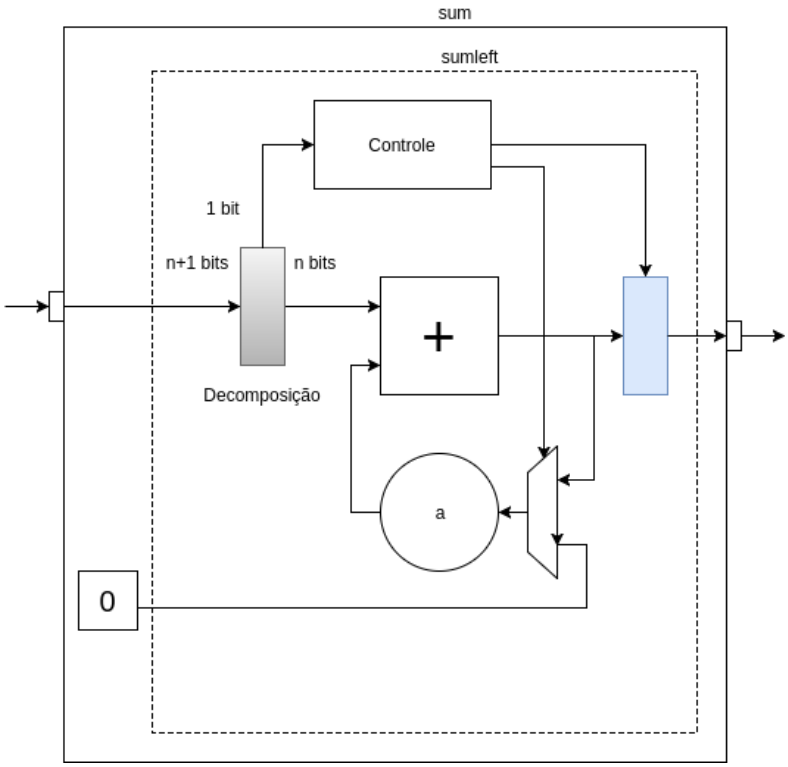

Figura 4.12: Função sum

4.2.5.4 Componentes

A saída da etapa de síntese é um componente. Um componente um tipo de dado que representa um módulo de hardware e sua estrutura é semelhante à de um módulo em `SystemC`. Uma representação esquemática de um componente é mostrada na figura 4.13. Cada componente contém:

- Entradas e saídas conectadas à listas
- Instâncias de outros componentes (as vezes mais de uma instância de um mesmo componente)
- Conexões entre instâncias e entre as portas de entrada e saída do componente
- Um controle, que realiza a partes mais complexas das decisões em um componente

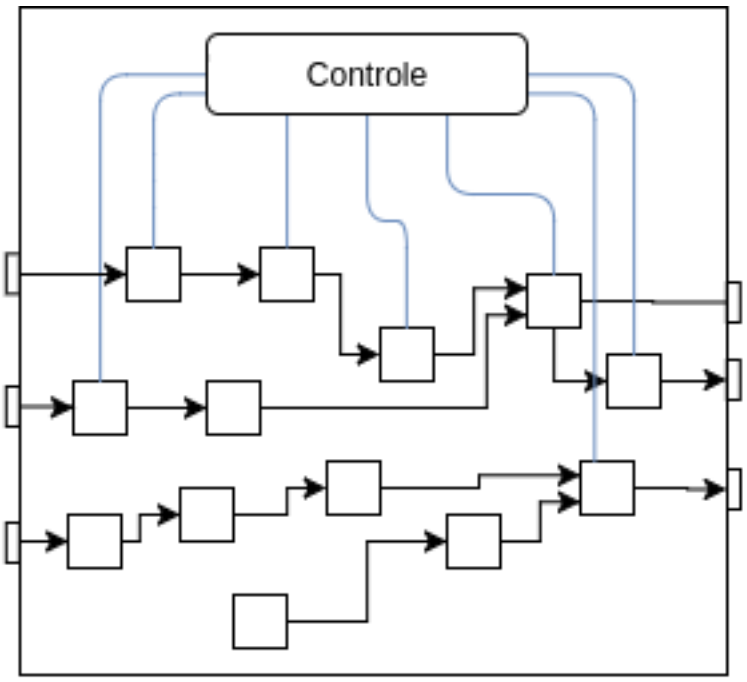

Figura 4.13: Esquema geral de um componente

Um componente pode ser visto, em geral, como uma parte operativa conectada à um controle. A parte operativa é composta por todas as instâncias de módulos e suas conexões. O controle é a parte que é responsável por fazer com que a parte operativa opere de forma correta. O controle, em geral é responsável por:

- Controle de fluxo — em funções condicionais
- Detectar parada da função (fim do processamento)
- Guardar valores como estados quando necessário e descartá-los quando puder

O modelo computacional resultante ao conectar vários componentes não é bem definido. O modelo é uma mistura de *dataflow*, por ser baseado em conexões por filas, e máquinas de estado, por ter componentes com estado e transições de estado. Felizmente, o `SystemC` faz com que a implementação de modelos mistos seja relativamente trivial visto que um dos objetivos do `SystemC` é exatamente integrar diferentes tipos de modelos de computação [4].

4.2.6 Geração do SystemC

Gerar o `SystemC` é a etapa final e mais simples devido à semelhança entre componentes e módulos do `SystemC`. Instâncias de outros componentes viram declarações de variáveis dos tipos adequados. Entradas, saídas e sinais intermediários são declarados do tipo `sc_fifo<>`, com exceção de variáveis de estado. Conexões entre módulos são implementadas usando notação normal do `SystemC` sendo necessário, na maioria delas, a definição de variáveis intermediárias. Por fim, o

controle do sistema é implementado usando SC_THREADs. O esquema a seguir é uma simplificação do resultado do módulo resultante da função sum com comentários que indicam a funcionalidade da parte do código. Todos os códigos gerados em SystemC se assemelham com esse.

```
#include "systemc.h"

// inclusao dos arquivos dos modulos necessarios
#include "const_dec_12_.h"
...
#include "equ1_.h"

SC_MODULE(sum) {

  // entradas e saidas
  sc_fifo_in<sc_lv<32> > s;
  sc_fifo_in<sc_lv<31> > a;
  sc_fifo_out<sc_lv<31> > out;

  // declaracao das instancias do modulo
  const_dec_12_ const_dec_11;
  ...
  equ1_ equ1;

  // declaracao de sinais intermediarios
  sc_lv<31> a__aux;
  sc_lv<32> s__now__1;
  ...
  sc_fifo<sc_lv<31> > __fifo__sum__rec__expr__2__2__out;

  void proc();

  // declaracao das conexoes do modulo
  SC_CTOR(sum) : const_dec_11("const_dec_11") ..., equ1("equ1") {

    // conexoes
    add1.in2(__fifo__sum__rec__expr__2__2__in__a);
    ...
    equ1.in2(const_dec_01_out__equ1_in2);

    // chamada ao processo (controle) do modulo
    SC_THREAD(proc);

  }
};

void sum::proc() {
  while(true) {
    // implementacao do controle
```

```
    }
}
```

Capítulo 5

Resultados Obtidos

Resultados obtidos da implementação do sistema de síntese

5.1 Características do sistema

O sistema de síntese implementado tem como entrada um arquivo contendo a especificação em linguagem funcional (*subset* de `Haskell`) e tem como saída a criação de um diretório contendo todos arquivos em `SystemC` que implementam a arquitetura resultante do sistema. Adicionalmente é gerado um arquivo de *testbench* para testar o resultado — compilar os arquivos do diretório e rodar o executável roda o teste.

O sistema implementado consegue sintetizar uma grande gama de funções. O escopo do sistema abrange a parte mais comumente utilizada da programação funcional — funções recursivas com tipos recursivos. É possível implementar uma grande quantidade de circuitos úteis (principalmente para processamento de dados) somente usando esse escopo.

5.2 Exemplos

Este seção mostrará a geração de dois circuitos utilizados na indústria para processamento de dados: o SAD e o filtro FIR. O anexo do filtro FIR é disponibilizado como anexo.

5.2.0.1 SAD

O SAD (*Sum of Absolute Differences* ou Soma das Diferenças Absolutas) é usado para medir a similaridade entre blocos de imagens e é calculado sendo a soma das diferenças absolutas entre *pixels* em posições iguais. A implementação do SAD é mostrada abaixo e é descrita como a soma (`sum`) do absoluto (`map abs`) das diferenças (`zipWith sub`) entre as duas imagens que neste caso são representadas como listas de inteiros (`List Int`).

```
sad :: List Int → List Int → Int
sad xs ys = sum (map abs (zipWith (-) xs ys))
```

A função SAD, que é combinacional, geraria o circuito mostrado da figura 5.1.

Figura 5.1: Síntese da função SAD

É importante lembrar que como os inteiros de entrada são compostos de 32 bits, a entrada do SAD terá 33 bits, onde o bit adicional igual à zero indicará o fim da lista. Saber o fim da lista é importante para o módulo `sum` que gera a saída apenas quando a lista acabar, diferentemente das funções `map` e `zipWith` que sempre geram saídas mesmo antes do fim das listas de entrada.

Na prática, a o arquivo que implementa a função SAD deveria implementar também as dependências da função `sad` (as funções `sum`, `map` e `zipWith`). Como são funções clássicas da programação funcional estas funções poderiam estar pré definidas em uma biblioteca padrão da linguagem. Como um sistema de bibliotecas não foi implementado essas funções devem ser definidas no arquivo pelo usuário.

5.2.0.2 Filtro FIR

O filtro FIR discreto é um filtro também muito utilizado em processamento de dados. Este filtro é baseado em convolução e por isso é tem código um pouco mais complexo que o SAD. Apesar disso é possível perceber padrões no código (mostrado abaixo).

```
fir :: List Int → List Int → Int
fir xs ys = map (y xs) [1..]

y :: List Int → Int → Int
y xs n = sum (zipWith (*) mask (window n xs))

mask :: List Int
mask = {- coeficientes -}

window :: Int → List Int → List Int
window n s = reverse (take s n)
```

A função `window` é responsável por gerar os elementos que serão multiplicados de acordo com suas posições `zipWith mul` com os elementos da máscara `mask` e somados `sum`. O resultado da função `y` é o enésimo resultado do filtro enquanto o `map (y xs) [1..]` é a lista que representa os valores do filtro FIR em todas as posições. A imagem 5.2 tenta explicar graficamente este processo.

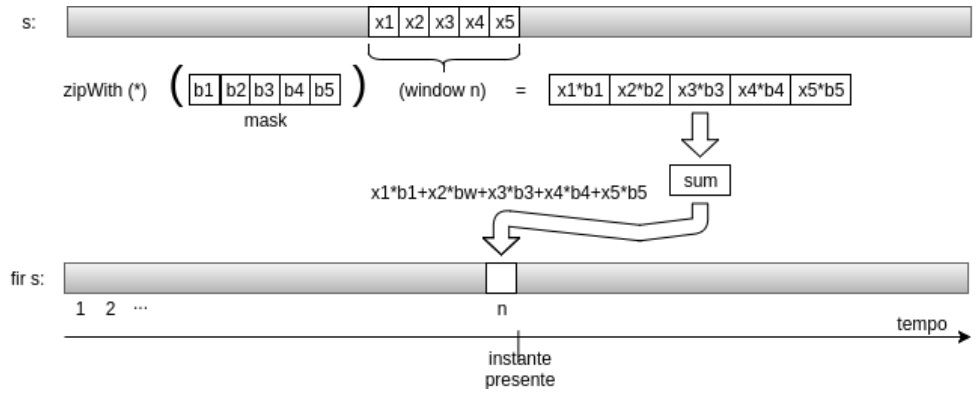

Figura 5.2: Explicação gráfica do filtro FIR

O resultado esquemático em componentes do filtro FIR é o mostrado na imagem 5.3 e ainda não corresponde à uma implementação ideal de filtro FIR. Um ponto positivo dessa implementação é que ela independe do número de *taps* visto que entrada e a máscara são listas. Na seção 5.4 é feita uma análise a respeito do desempenho desta implementação.

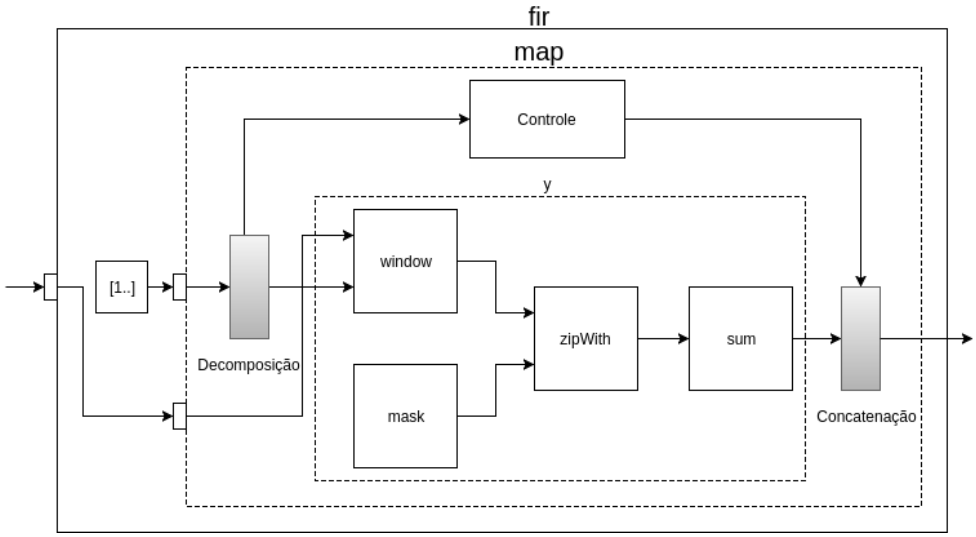

Figura 5.3: Síntese do filtro FIR

O código do filtro FIR e o resultado de sua síntese está em anexo. A definição atual do filtro FIR que é aceita pelo sistema de síntese é um pouco diferente da mostrada acima. Como o sistema não aceita construções de listas infinitas, como `[1..]` — que representa a lista `[1,2,3,4,5,...]` — esta lista será disponibilizada como uma entrada do módulo (ao invés de variável interna). Além disso, a falta de *currying*, faz com que a expressão `map (y xs) [1..]` — mais especificamente a aplicação `(y xs)` — seja inválida. Para definir a mesma funcionalidade, a aplicação da função `y` deve ser colocada dentro da definição da função `map` que, por sua vez, deve ter como argumento adicional a variável `xs`.

Além disso, todas as dependências devem ser definidas pelo programador no mesmo arquivo por causa da ausência de um sistema de bibliotecas e da ausência de uma biblioteca padrão com

funções clássicas da programação funcional (como o módulo `Prelude`, em `Haskell`). A definição dessas dependências fazem com que o arquivo fique maior.

5.3 Limitações do Sistema

São listadas e descritas abaixo algumas das limitações do sistema implementado:

Falta de um sistema de módulos: Todas funções necessárias são definidas em um só arquivo.

Falta de algumas funcionalidades que deixam as funções mais legíveis: *Currying*, listas infinitas, `where`, `let` entre outras funcionalidades fazem com que as funções fiquem mais simples e legíveis.

O sistema é incapaz de sintetizar recursões não simples: Funções irregulares e múltiplas recursões não são permitidas na implementação apesar de ser possível sintetizá-las via reescrita como visto na seção 4.2.5.2.

O sistema é incapaz de sintetizar tipos recursivos não simples: Tipos árvore ou outros tipos com recursões complexas não são sintetizáveis na implementação atual.

Falta de polimorfismo na linguagem de entrada: Não é possível para o usuário definir funções polimórficas prejudicando a expressividade da linguagem.

Código SystemC gerado não é legível e é maior que o necessário algumas vezes: Geração de variáveis e grande tamanho do código faz com que os arquivos `SystemC` sejam difíceis de ler.

Falta de otimizações: A falta de otimizações leva insegurança para o usuário pois o resultado da síntese pode ficar desotimizado dependendo da entrada.

5.4 A importância de Otimizações

Otimizações talvez sejam a parte mais importante de qualquer sistema de síntese visto que o custo e a eficiência (em gasto de energia ou *throughput*) de circuitos digitais são os fatores mais importantes no design de uma arquitetura de hardware. Circuitos ineficientes, mesmo que gerados de implementações de altíssimo nível, não são aceitos pela indústria.

Dito isso, a maior limitação do sistema implementado é a falta de otimizações. Todos resultados (quando aceitos pela linguagem) funcionam, porém muitos deles são ineficientes. A maior ineficiência identificada aparece ao sintetizar funções que repetidamente passam como argumento *streams* inteiras para outras funções. Nestes casos, e em alguns outros, é necessária a cópia dos elementos da *stream* (o que ocasionaria, em uma implementação prática em uso de memórias).

Essa observação, no entanto, não descarta a utilidade do sistema implementado, apenas reitera a importância das otimizações. A implementação de algumas poucas otimizações já podem trazer grandes melhoras nos resultados.

5.4.1 Otimizando o filtro FIR

O filtro FIR implementado na seção 5.2.0.2 gerou o circuito 5.3. Este circuito difere de implementações em VHDL e é realmente menos eficiente. Esta seção procura mostrar que a implementação de apenas uma otimização já pode gerar uma implementação otimizada da síntese do filtro FIR.

Uma otimização possível para qualquer componente que tem como entrada vários valores no tempo (entrada do tipo `Stream`) é, ao invés de pegar um valor por vez, pegar mais de um por vez. Essa otimização pode, ou não, melhorar a eficiência do circuito.

No caso do filtro FIR essa otimização aumenta a eficiência do sistema. Caso o componente window produza, ao invés de 1 saída de cada vez, n de cada vez — sendo n o número de *taps* ou, na função, o número de elementos da lista `mask`. Os módulos `zipWith` e `sum` devem também modificar suas entradas/saídas para aceitar 5 valores simultâneos. O resultado final do filtro FIR após a otimização é mostrado na imagem 5.4 e uma comparação com o filtro FIR geralmente especificado em RTL é feita em 5.5.

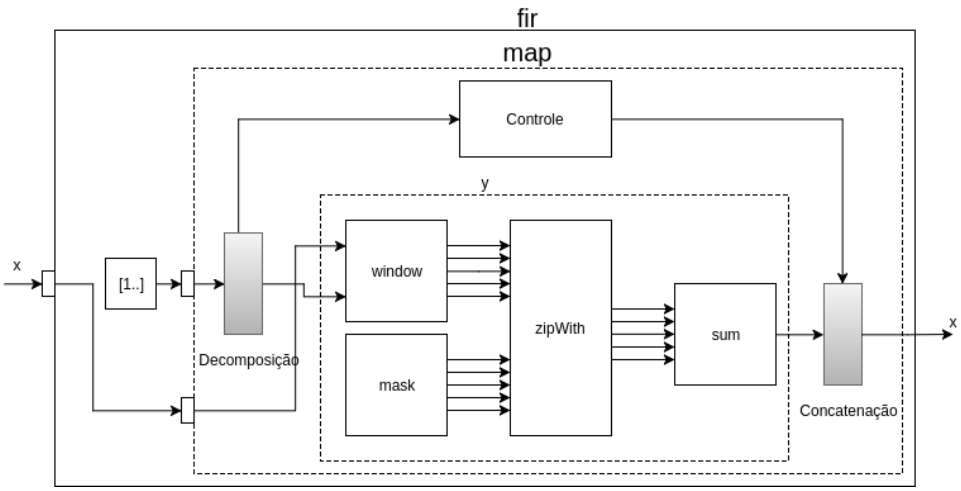

Figura 5.4: Síntese do FIR após otimização

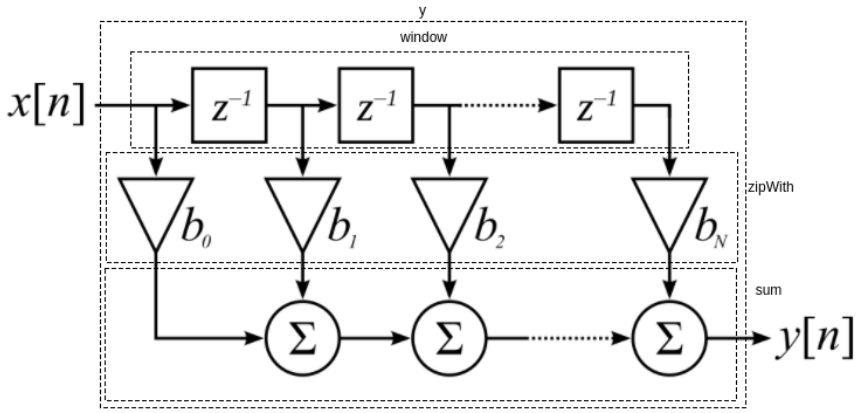

Figura 5.5: Implementação comum manual do filtro FIR

Verifica-se que dependendo da forma como se agrupam os módulos, as implementações são similares. A imagem 5.3 possuí *bounding boxes* que sugerem similaridades com os blocos gerados pelo sistema de síntese em 5.4.

A diferença entre a implementação manual e a implementação otimizada é uso de 1 bit adicional na entrada da implementação sintetizada que é utilizado para indicar o fim da lista de entrada. Esse bit é necessário caso quisermos reutilizar o módulo `fir` dentro de outras funções do sistema. Se fosse necessário, por exemplo, conectar o resultado do filtro FIR à um módulo `sum`, que somaria todos os valores da saída, isso não seria possível sem o uso do bit adicional. No entanto, caso o filtro FIR seja o resultado final da síntese, o bit adicional pode ser considerado obsoleto.

Conclui-se então que apesar de gerar resultados não otimizados, uma extensão do sistema com otimizações é viável.

Capítulo 6

Conclusões

Conclusões, aprendizados e trabalhos futuros.

6.1 Objetivos atingidos

O objetivo do projeto era implementar um sistema de síntese de alto nível que fosse capaz de sintetizar funções analisando as suas definições. O sistema de síntese foi implementado e sintetiza um grupo pequeno, porém muito expressivo, de funções. É possível, com esse sistema, implementar funções de processamento de dados utilizando funções com entradas e/ou saídas de tipos recursivos (como `List`). O resultado do sistema não recebe otimizações e por isso não deve ser utilizado ainda pela indústria. Como o objetivo era implementar um sistema de síntese não necessariamente ótimo, esse objetivo foi alcançado.

Outro objetivo do projeto era reforçar a ideia de que descrições funcionais são mais adequadas que descrições imperativas quando se trata de síntese de alto nível. Os exemplos e as imagens mostradas neste texto tentaram mostrar como a transformação de funções em hardware pode ser muito simples quando os menores detalhes da implementação são ignorados. Detalhes esses que constituíram a maior parte das dificuldades de desenvolvimento do sistema.

A escolha do `SystemC` como linguagem alvo foi essencial para o desenvolvimento das funcionalidades do sistema. A geração do código final é uma simples transformação de componentes (resultado da etapa de síntese) para strings (conteúdo dos arquivos do `SystemC`) — assim mais esforço foi gasto na parte de síntese, que é o principal tema do projeto, e menos tempo foi gasto na geração de código. Caso `VHDL` fosse escolhido como linguagem alvo, o sistema provavelmente não conseguiria sintetizar toda a gama de funções que consegue.

Além disso, acredito que o aprofundamento em temas como projeto de circuitos integrados, e programas funcionais e o desenvolvimento de um sistema prático sustentado por uma teoria ainda não muito consolidada (a teoria de síntese de descrições funcionais) foram importantes para o desenvolvimento da minha visão a respeito da implementação prática de projetos de engenharia.

6.2 Pontos positivos e negativos do sistema

O sistema de síntese implementado tem como pontos positivos:

- Nível de abstração elevado na especificação de entrada
- Muitas funções da linguagem `Haskell` podem ser utilizadas sem nenhuma ou mínima diferença
- Possibilidade de expressar sistemas relativamente complexos compondo funções simples

E pontos negativos:

- Falta de otimizações faz com que o código de saída não seja ótimo em muitos casos
- Problema da falta de otimização é pior para sistemas compostos de mais funções

6.3 Trabalhos futuros

Tentou-se mostrar, ao longo trabalho, as vantagens do uso de descrições funcionais em síntese de alto nível. Programas funcionais são muito maleáveis com poucas construções. Qualquer restrição muito grande no escopo das funções permitidas na síntese não pareceria programação funcional. A escolha do escopo como funções recursivas com tipos ADT's simples foi uma boa escolha pois engloba uma gama muito expressiva de funções.

O sistema não sintetiza, no entanto, muitas outras técnicas utilizadas na programação funcional moderna. Tipos de dado que contém funções, por exemplo, não podem ser sintetizados apesar de serem a base de vários estilos de programação funcional em software — como `Monad`'s, `Functor`'s e `Arrow`'s, citando os mais comuns. Como seria, afinal, a síntese de um tipo que contém uma função? Neste sentido, aumentar o escopo de funções permitidas da linguagem, tendo como base o grande escopo da linguagem `Haskell`, é uma melhora natural do sistema.

Em relação aos detalhes de implementação do código final, seria interessante melhorar o sistema de erros, e, principalmente melhorar o sistema de geração de *testbenches*. Também seria interessante pensar na ideia de pré simulação do sistema e acabar com a necessidade dos *testbenches*.

Em relação à sintaxe da linguagem de entrada, a inclusão de `where`, `let` e *currying* deixariam o código mais conciso e legível para o usuário e a inclusão de tipos polimórficos fariam com que as funções feitas pelo usuário fossem reutilizáveis. Adicionar um sistema de módulos também seria interessante. Com módulos e polimorfismo, a linguagem pode ser possivelmente utilizada para projetos mais complexos.

Otimização é o tópico mais importante para o melhoramento do sistema. Na minha visão, o desenvolvimento de um sistema de otimização é o que define se um sistema pode ou não ser

usado na indústria. Várias otimizações podem ser implementadas a nível de componente. Na seção 5 foi mostrada como uma simples otimização pode aumentar muito a performance de um módulo sintetizado. Otimizações podem usar as definições funcionais da linguagem como objetos auxiliadores da otimização: técnicas matemáticas de reescrita de funções e provas matemáticas ajudam a otimizar o sistema tanto no nível funcional quanto no nível de componentes (e de hardware).

Sobre a geração de código, uma melhora significativa seria mudar a linguagem alvo de `SystemC` para `VHDL` (ou `Verilog`). Para um sistema automatizado de síntese de alto nível a geração de um código mais baixo nível, como o da linguagem `VHDL`, significa um controle maior do processo da síntese. Por outro lado, a parte de síntese de componentes, em especial a parte de síntese do controle dos módulos e das FIFO's, teria que ser adaptado.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Gajski, Daniel D. (et al.) *High-Level Synthesis:* Introduction to Chip and System Design. Kluwer Academics, Springer, 1992.

[2] Stephen A. Edwards *The Challenges of Synthesizing Hardware from C-Like Languages*. IEEE Design & Test of Computers, 2006

[3] Coussy, Philippe; Gajski, Daniel D. (et al.) *An Introduction to High-Level Synthesis*. IEEE Design & Test of Computers, 2009.

[4] Grotker, Thorsten. *System Design with SystemC*. Hingham, MA, USA: Kluwer Academic Publishers, 2002.

[5] Per Bjesse, Koen Claessen, Mary Sheeran. *Lava:* Hardware Design in Haskell. Proceedings of the third ACM SIGPLAN international conference on Functional programming - ICFP '98, 1998.

[6] Christiaan Baaij, Jan Kuper. *Using Rewriting to Synthesize Functional Languages to Digital Circuits*. Lecture Notes in Computer Science,Trends in Functional Programming, 2014.

[7] Richard Townsend, Martha A. Kim, Stephen A. Edwards. *From Functional Programs to Pipelined Dataflow Circuits*. Proceedings of the 26th International Conference on Compiler Construction - CC 2017, 2017.

[8] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Stephen A. Edwards. *Hardware Synthesis from a Recursive Functional Language*. 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2015.

[9] Jacobi, Ricardo Pezzuol, Charles Trullemans. *A Study of the Application of Binary Decision Diagrams to Multi-level Logic Synthesis*. Universite Catholique de Louvain, 1993.

[10] John Hughes. *Why Functional Programming Matters*. "Research Topics in Functional Programming" ed. D. Turner, Addison-Wesley, 1990, pp 17–42

[11] Mark P. Jones. *From Hindley-Milner Types to First-Class Structures*. Proceedings of the Haskell Workshop, La Jolla, California, Yale University Research Report YALEU/DCS/RR-1075, 1995.