Utilizando FPGAs com ferramentas *OpenSource*

Julio Nunes Avelar [1]

Departamento de sistemas de computação, Universidade estadual de campinas

30 de Agosto de 2024

[1]Endereço de e-mail: julio.avelar@students.ic.unicamp.br

Para minha querida paixão platônica: mesmo que você provavelmente nunca saiba disso, desde que te conheci, seu lindo sorriso e toda a sua perfeição têm me incentivado a ser uma pessoa melhor a cada dia.

Prefácio

Nos últimos anos, a indústria esteve sempre dois passos à frente da academia no campo da síntese de hardware, com a maioria das ferramentas de design sendo de origem proprietária. No entanto, esse cenário começou a mudar recentemente, com o surgimento de ferramentas *open-source* para o desenvolvimento de hardware e o rápido crescimento da comunidade de Hardware Livre.

Graças a essas mudanças, hoje temos a possibilidade de utilizar FPGAs (Field-Programmable Gate Arrays) e desenvolver ASICs (Application-Specific Integrated Circuits) por meio de fluxos completamente *open-source*. Este manual foi escrito com o intuito de facilitar o primeiro contato com essas ferramentas, oferecendo um caminho claro para o leitor explorar esse vasto universo.

Como base, foi utilizado a toolchain *open-source* OSS-CAD-Suite, que inclui o sintetizador Yosys, a ferramenta de placement and routing NextPNR, e os utilitários de geração de bitstream associados. Além disso, a principal plataforma abordada é a placa de desenvolvimento Colorlight i9, equipada com a FPGA Lattice ECP5 45f, amplamente suportada pelas ferramentas *open-source*, e com o design de sua PCI disponibilizado de forma aberta.

Este manual, além de ser focado nessa plataforma, também será útil para a utilização de outras plataformas, fornecendo uma base sólida para explorar diferentes possibilidades no desenvolvimento de hardware.

Por último, gostaria de esclarecer que, embora em alguns momentos seja citado um pouco sobre design de hardware, este manual não tem como objetivo ensinar o desenvolvimento de hardware em si, mas sim como utilizar as ferramentas *open-source* para esse fim. Para aprender sobre desenvolvimento de hardware, recomendo o material do professor Rodolfo Azevedo, do Instituto de Computação da Unicamp[1].

Aproveito para expressar minha gratidão a todos que contribuíram para a realização deste trabalho. Em especial, agradeço ao meu orientador, Rodolfo Azevedo, por me proporcionar a oportunidade de utilizar todas as FPGAs descritas neste manual, além de seu apoio e orientação inestimáveis ao longo do processo. Também sou grato aos meus colegas de laboratório e amigos que me ajudaram com suas sugestões e feedbacks. Por fim, agradeço à minha família, que sempre esteve ao meu lado.

Boa leitura e boas implementações!

Julio Nunes Avelar

[1] Disponível em: `https://www.ic.unicamp.br/~rodolfo/Cursos/verilog`

Conteúdo

Prefácio **2**

1 Conhecendo o Ferramental **4**
1.1 Introdução . . . 4
1.2 O fluxo de Design de circuitos digitais . . . 4
1.3 As ferramentas que compõem esse fluxo . . . 5
1.4 Instalação . . . 6
1.4.1 Vantagens e Desvantagens de Cada Método . . . 6
1.4.2 Instalação através do OSS-CAD-Suite . . . 6
1.4.3 Instalação através do Gerenciador de Pacotes . . . 7

2 A Placa Colorlight i9 **8**
2.1 Introdução . . . 8
2.2 Recursos . . . 8
2.3 Guia rápido de pinos . . . 8

3 Iniciando pela Teoria **10**
3.1 Simulando com IVerilog . . . 10
3.1.1 Flags Úteis do IVerilog . . . 11
3.2 Utilizando o GTKWave . . . 12

4 Iniciando o Desenvolvimento **13**
4.1 Introdução . . . 13
4.2 Definindo *constraints* . . . 13
4.3 Um breve *Blink* . . . 13
4.4 Construindo o *Blink* . . . 14
4.4.1 Automatizando o Processo . . . 15
4.5 Utilizando PLLs . . . 15
4.6 Funcionalidades Úteis do Yosys e do NextPnR . . . 17
4.6.1 Flags Adicionais e Comandos Úteis do Yosys . . . 17
4.6.2 *Flags* Adicionais e Casos de Uso do NextPnR . . . 17

5 Aplicações Práticas e Casos de Uso **19**
5.1 Exemplos Interessantes . . . 19
5.2 Utilizando o LiteX . . . 19
5.2.1 Instalando o LiteX . . . 19
5.2.2 Utilizando o LiteX com a Colorlight i9 . . . 20
5.2.3 Linux on VexRiscV . . . 20

6 Apêndice **21**
6.1 Utilizando FPGAs Gowin . . . 21
6.2 Utilizando FPGAs Xilinx . . . 21
6.3 Desenvolvendo um Chip com TinyTapeout . . . 21

Capítulo 1

Conhecendo o Ferramental

1.1 Introdução

Antes de iniciarmos a utilização das ferramentas para o desenvolvimento de hardware, é essencial que conheçamos cada uma delas e tenhamos em mente o que são capazes de fazer e para que servem. Ao contrário das ferramentas proprietárias, que geralmente possuem poucos utilitários responsáveis por uma grande quantidade de funções, as ferramentas *open-source* oferecem maior flexibilidade, mas também são mais fragmentadas, com o fluxo sendo composto por diversas ferramentas, cada uma responsável por uma pequena parte do processo.

Neste capítulo, exploraremos parte do conjunto de ferramentas disponível no OSS-CAD-Suite, abrangendo desde a síntese, passando pelo posicionamento e roteamento, até chegar ao carregamento na FPGA.

Falando um pouco mais sobre o OSS-CAD-Suite: trata-se de um conjunto de ferramentas para desenvolvimento de hardware, sendo que todas as ferramentas incluídas são *open-source*, ou seja, não há qualquer dependência de ferramentas proprietárias.

1.2 O fluxo de Design de circuitos digitais

O fluxo de design de circuitos digitais, de maneira geral, inicia-se com a descrição do circuito a ser construído por meio de uma linguagem de descrição de hardware (Hardware Description Language - HDL), através de um nível de descrição que chamamos de RTL (Register Transfer Level), onde o circuito é descrito de forma comportamental. As linguagens mais utilizadas para essa tarefa atualmente são Verilog, SystemVerilog e VHDL.

Após a descrição do circuito, ele passa por um processo de síntese, onde é convertido para um nível de descrição chamado Gate Level. Nesse nível, o hardware é representado por uma lista de malhas ou redes (Netlist, em inglês) de portas lógicas e/ou outros elementos de maior complexidade. Antes da síntese, é realizada uma verificação formal para garantir que o hardware descrito esteja livre de bugs ou falhas de especificação. Após o processo de síntese, geralmente é realizada uma verificação de equivalência entre a Netlist gerada e o circuito descrito em nível RTL, garantindo a correção da transformação.

Os elementos de uma Netlist variam conforme os componentes físicos disponíveis na arquitetura alvo. No caso de ASICs, por exemplo, a Netlist pode conter apenas portas lógicas simples, como NAND, NOR e NOT, além de flip-flops tipo D. Já em FPGAs, a Netlist é composta por LUTs (Look-Up Tables) e outras células complexas presentes na FPGA alvo, como somadores, multiplicadores, divisores, memórias e barrel shifters.

Com uma Netlist gerada, o fluxo se divide em dois caminhos: o primeiro para FPGAs e o segundo para ASICs. Neste manual, focaremos no primeiro caminho, deixando o segundo parcialmente descrito no apêndice.

Após a geração da Netlist, é realizado o processo de posicionamento e roteamento. Nessa etapa, decide-se onde posicionar os elementos da Netlist dentro das células limitadas da FPGA. Em seguida, é realizado o roteamento, que define como os elementos posicionados serão interligados na estrutura da FPGA.

Com o posicionamento e roteamento finalizados, podemos gerar um *bitstream*, que consiste em um arquivo binário contendo a configuração dos elementos internos da FPGA. De posse deste *bitstream*, podemos carregá-lo na FPGA, que, a partir desse momento, começará a se comportar conforme o circuito que descrevemos. O diagrama de todo esse processo pode ser visto na Figura 1.1.

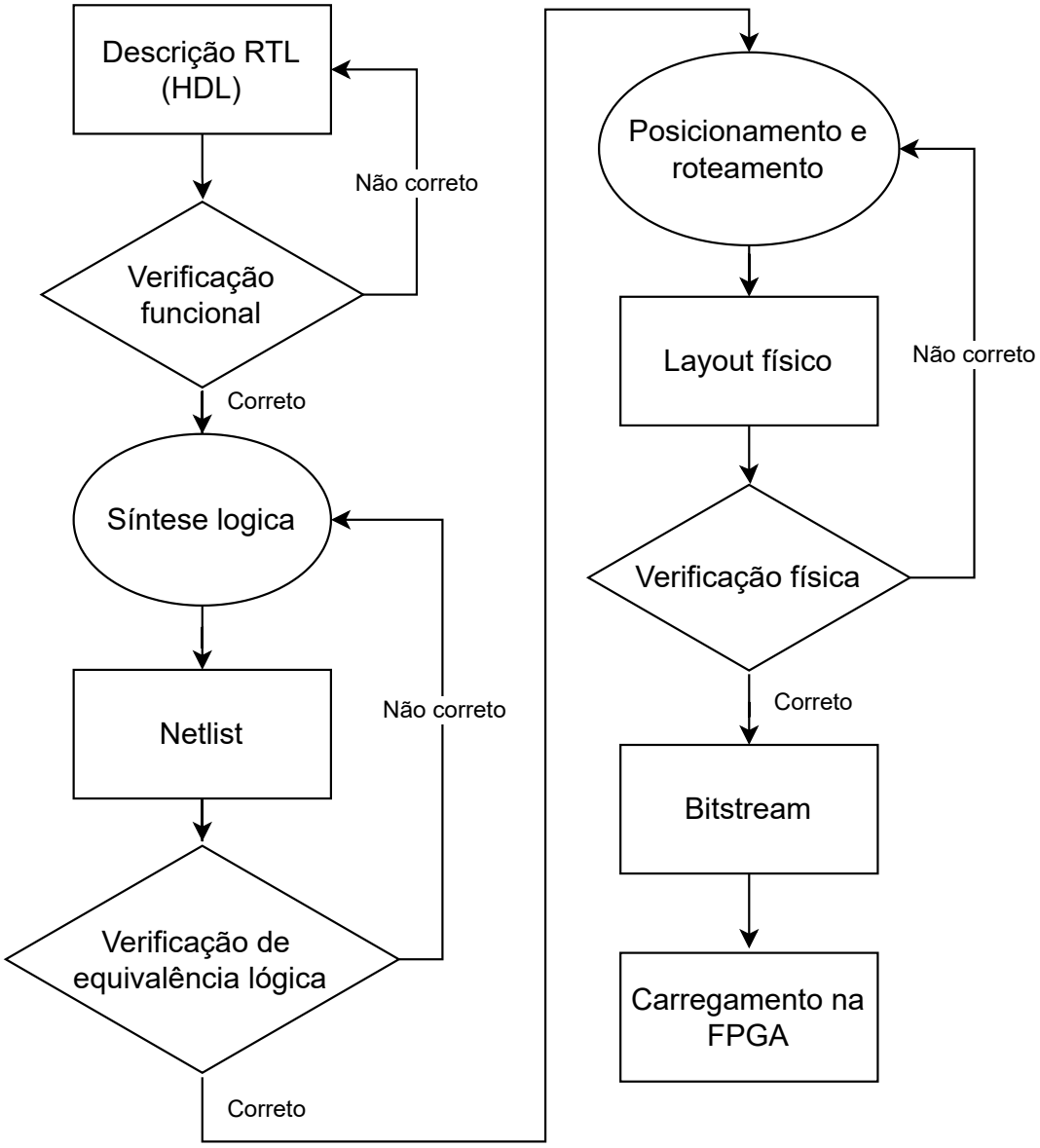

Figura 1.1: Diagrama de blocos do fluxo de design de circuitos digitais.

1.3 As ferramentas que compõem esse fluxo

Para a realização do fluxo descrito acima, utilizaremos quatro ferramentas distintas do OSS-CAD-Suite, cada uma responsável por uma etapa do processo.

O processo de síntese e verificação de equivalência lógica é realizado pela ferramenta Yosys (Yosys Open SYnthesis Suite) em conjunto com a ferramenta ABC (System for Sequential Logic Synthesis and Formal Verification), integrada internamente ao Yosys. O Yosys recebe como entrada um conjunto de arquivos em Verilog ou SystemVerilog. Com o auxílio de um plugin baseado no GHDL, também é possível utilizar arquivos em VHDL. A saída do Yosys é uma Netlist gerada para a arquitetura alvo, que é especificada durante o comando de síntese.

Para o posicionamento e roteamento, utilizamos o NextPnR. O NextPnR suporta diferentes FPGAs, organizadas por subprojetos dedicados a cada família de dispositivos. Assim, FPGAs de fabricantes e famílias diferentes podem ter níveis variados de suporte, dependendo do estágio de desenvolvimento do subprojeto responsável. No momento em que este manual foi escrito, o NextPnR oferece suporte às seguintes FPGAs:

1. Dispositivos Lattice iCE40 - projeto IceStorm
2. Dispositivos Lattice ECP5 - projeto Trellis
3. Dispositivos Lattice Nexus - projeto Oxide
4. Dispositivos Gowin LittleBee - projeto Apicula
5. Dispositivos Altera Cyclone V - projeto Mistral
6. Dispositivos Lattice MachXO2 - projeto Trellis

Junto aos subprojetos do NextPnR, para cada família de FPGA, estão incluídas as ferramentas responsáveis pela geração de *bitstream* para essas famílias. Como este manual utiliza a placa de desenvolvimento Colorlight i9, equipada com uma FPGA Lattice ECP5 45f, utilizaremos a ferramenta `ecppack`, incluída no projeto Trellis.

Por fim, para o carregamento do *bitstream* na FPGA, utilizaremos o utilitário OpenFPGALoader, que suporta uma vasta gama de FPGAs de diferentes fabricantes.

1.4 Instalação

A instalação das ferramentas pode ser feita de várias maneiras distintas. Neste manual, descreveremos duas opções principais: 1) a instalação através do download do pacote completo do OSS-CAD-Suite, que contém todas as ferramentas; e 2) a instalação através do gerenciador de pacotes da sua distribuição Linux.

1.4.1 Vantagens e Desvantagens de Cada Método

A instalação do OSS-CAD-Suite oferece a principal vantagem da praticidade, pois inclui todo o conjunto de ferramentas em um único pacote. No entanto, essa abordagem apresenta desvantagens, como a necessidade de atualização manual das ferramentas e a utilização de versões *nightly*, além de ocupar mais espaço em disco devido ao amplo conjunto de ferramentas incluídas.

Por outro lado, ao utilizar a versão do gerenciador de pacotes, podemos contar com versões estáveis e atualizações automáticas junto ao sistema, além de um menor consumo de espaço em disco. Entretanto, a desvantagem é a necessidade de instalar manualmente todos os pacotes necessários, exigindo que o usuário saiba exatamente o nome de cada um deles.

1.4.2 Instalação através do OSS-CAD-Suite

Para instalar o OSS-CAD-Suite, basta baixar o pacote disponibilizado na última *release* do projeto. As *releases* podem ser encontradas em: `https://github.com/YosysHQ/oss-cad-suite-build/releases`. Na data da redação deste manual, a última *release* disponível é a de 2024-10-01. A instalação pode ser realizada da seguinte forma:

```
$ cd /tmp
$ wget https://github.com/YosysHQ/oss-cad-suite-build/\
releases/download/2024-10-01/oss-cad-suite-linux-x64-20241001.tgz
$ tar xvf oss-cad-suite-linux-x64-20241001.tgz
$ mkdir -p ~/eda
$ mv oss-cad-suite ~/eda/oss-cad-suite
$ # Para usuarios de bash
$ echo "$PATH=$PATH:~/eda/oss-cad-suite/bin" >> ~/.bashrc
$ # Para usuarios de zsh
$ echo "$PATH=$PATH:~/eda/oss-cad-suite/bin" >> ~/.zshrc
```

Lembre-se de, ao realizar o download, substituir o link pelo da *release* do dia.

1.4.3 Instalação através do Gerenciador de Pacotes

Para simplificar, instalaremos apenas as ferramentas necessárias para a utilização da FPGA Lattice ECP5. Além destas, também instalaremos o simulador Iverilog e a ferramenta de visualização de *waveform* GTKWave.

Para usuários de ArchLinux e derivados, utilize o seguinte comando:

```
$ sudo pacman -S yosys prjtrellis gtkwave iverilog openfpgaloader
$ cd /tmp
$ git clone https://aur.archlinux.org/nextpnr-ecp5-nightly.git
$ cd nextpnr-ecp5-nightly
$ makepkg -si
```

Para usuários de Debian e derivados, utilize o seguinte comando:

```
$ sudo apt install openfpgaloader yosys gtkwave iverilog nextpnr-ecp5
```

Vale ressaltar que, no momento da redação deste manual, o pacote nextpnr-ecp5 está disponível apenas nas versões *testing* e *sid* para usuários de Debian. Para usuários de Arch, o nextpnr-ecp5 está disponível apenas na versão *nightly* pelo AUR.

Capítulo 2

A Placa Colorlight i9

2.1 Introdução

A Colorlight i9 é uma placa de desenvolvimento *open-source* participante do projeto Colorlight. Ela contém uma FPGA Lattice ECP5 45f e possui uma irmã com capacidades menores, a Colorlight i5, além de uma versão maior, a Colorlight i9 Plus. A Colorlight i5 conta com uma FPGA Lattice ECP5 25f, enquanto a i9 Plus possui uma FPGA Xilinx Arty A7 50T. As placas Colorlight i9 e i5 apresentam compatibilidade total de pinos em suas placas de expansão.

A placa de expansão possui poucos recursos integrados, delegando a funcionalidade a 10 conectores PMOD de expansão e alguns pinos de I/O fora do padrão PMOD. Além desses conectores, a placa contém um conector HDMI(somente saída de video) e alguns pinos reservados para a conexão de um módulo Ethernet.

2.2 Recursos

Além da FPGA Lattice ECP5 45f, a Colorlight i9 e sua placa de expansão incluem:

1. JTAG
2. UART
3. SPI-Flash
4. SDRAM
5. 1 LED de usuário
6. *Clock* de 25MHz
7. 2 ETH-PHY Broadcom B50612D 1Gb

2.3 Guia rápido de pinos

O pinout da placa de extensão da Colorlight i9 pode ser visto na Figura 2.1, retirada do repositório oficial do projeto Colorlight[1].

[1]Disponível em: `https://github.com/wuxx/Colorlight-FPGA-Projects`

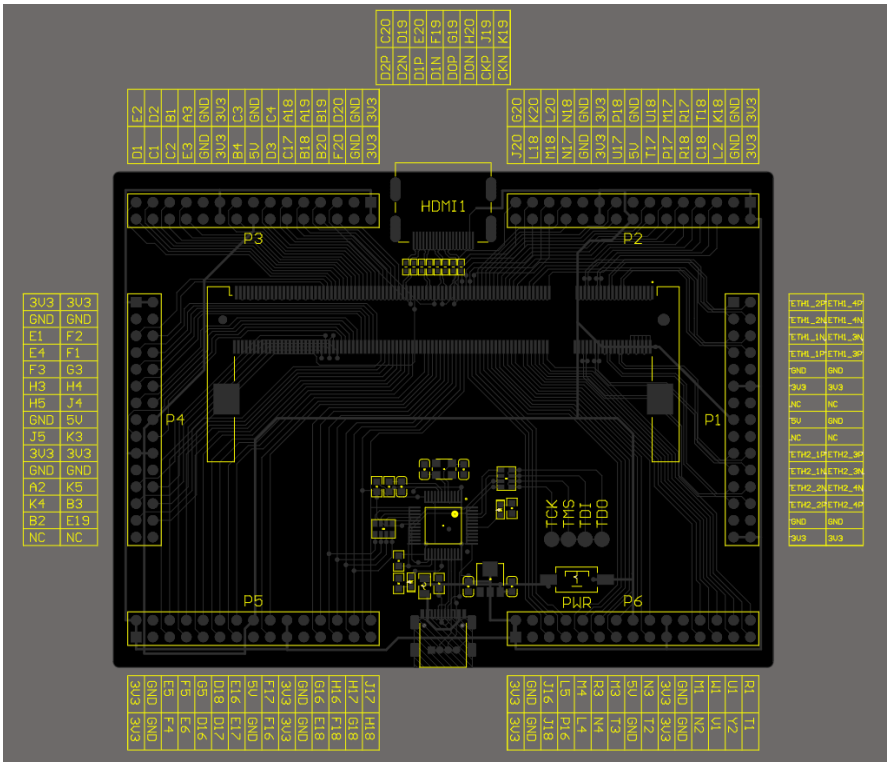

Figura 2.1: Pinout da placa de extensão da Colorlight i9

Os demais pinos são apresentados a seguir:

Pin	Função	Pin	Função
JTAG			
J27	TCK	J30	TDO
J31	TMS	J32	TDI
Clock			
P3	25 MHz clk		
LED			
L2	D2		
SPI-FLASH			
R2	CS	U3	SCK
V2	MISO	W2	MOSI

Tabela 2.1: Pinout para JTAG, Clock, LED e SPI-FLASH

Pin	Função	Pin	Função
SDRAM			
B9	CLK	A11	A9
VCC	CKE	B12	A10
GND	CS	B6	DQ0
B10	RAS	A5	DQ1
A9	CAS	A6	DQ2
A10	WE	A7	DQ3
GND	DQM0	C7	DQ4
GND	DQM1	B8	DQ5
GND	DQM2	B5	DQ6
GND	DQM3	A8	DQ7
B11	BA0	D8	DQ8
C8	BA	D7	DQ9
B13	A0	E8	DQ10
C14	A1	D6	DQ11
A16	A2	C6	DQ12
A17	A3	D5	DQ13
B16	A4	E7	DQ14
B15	A5	C5	DQ15
A14	A6	C10	NC
A13	A7	C9	NC
A12	A8	GND	NC

Tabela 2.2: Pinout do SDRAM

A especificação de cada componente e o datasheet dos mesmos podem ser encontrados no repositório oficial do projeto Colorlight.

Capítulo 3

Iniciando pela Teoria

Durante o processo de desenvolvimento de hardware, uma das ferramentas mais valiosas é a simulação. Simular um design RTL e verificar o seu comportamento, seja visualmente ou formalmente, pode ser de grande utilidade ao iniciar o desenvolvimento de novos módulos de hardware ou ao procurar *bugs* em um design já existente.

Para essa tarefa, o OSS-CAD-Suite fornece ferramentas de simulação como o IVerilog, Verilator e GHDL. As duas primeiras são voltadas para linguagens Verilog e SystemVerilog, enquanto o GHDL é destinado a VHDL.

Neste manual, utilizaremos o IVerilog devido à sua simplicidade e por ser perfeitamente adequado para trabalhar com Verilog sem a necessidade de outras linguagens auxiliares, como é o caso do Verilator. Apesar de suas limitações, o IVerilog é suficiente para as simulações que realizaremos.

Neste capítulo, será apresentada uma introdução ao uso do IVerilog, explicando sua sintaxe básica e como criar um *testbench*, entre outros aspectos relevantes.

3.1 Simulando com IVerilog

O IVerilog utiliza arquivos de *testbench* para gerar as simulações. Um *testbench* nada mais é que um arquivo Verilog convencional contendo um módulo superior (*top module*) e algumas diretivas específicas para simulação, como definição da unidade de tempo, geração de um sinal de *clock*, ou criação de um arquivo VCD para análise dos sinais. Abaixo está um exemplo de *testbench* simples:

```
1  module tb_test();
2
3  reg clk;
4
5  always #1 clk = ~clk;
6
7  initial begin
8      $dumpfile("test.vcd");
9      $dumpvars;
10
11     clk = 1'b0;
12
13     #50
14
15     $finish;
16 end
17
18 endmodule
```

Listing 3.1: Exemplo de Testbench - arquivo testbench_1.v

Para simularmos o *testbench* acima, podemos utilizar os seguintes comandos:

```
1 $ iverilog -o test.o -s tb_test testbench_1.v
2 $ vvp test.o
```

No exemplo acima, o *testbench* contém um bloco always responsável por gerar o sinal de *clock*, onde o registrador clk tem seu valor invertido a cada unidade de tempo. No bloco *initial*, utilizamos as diretivas $dumpfile e $dumpvars, onde $dumpfile especifica o nome do arquivo de saída para o *waveform* e $dumpvars define quais sinais serão "gravados"ao longo da simulação. Após um atraso de 50 unidades de tempo (definido por #50), a simulação é finalizada com a diretiva $finish.

Ao executar os dois comandos acima, serão gerados dois arquivos: test.o, contendo o design "compilado", e test.vcd, que armazena o *waveform* com o comportamento dos sinais ao longo do tempo. O comando iverilog recebe duas *flags*: -o, que define o arquivo de saída, e -s, que especifica o *top module* a ser simulado.

Também é possível utilizar o IVerilog com múltiplos arquivos Verilog. Abaixo, e possível visualizar um exemplo com um módulo somador de 4 bits e seu respectivo *testbench*:

```
module somador (
    input wire [3:0] A,
    input wire [3:0] B,
    output reg [3:0] S
);

always @(*) begin
    S <= A + B;
end

endmodule
```

Listing 3.2: Somador de 4 bits - arquivo somador.v

```
module tb_somador();

reg [3:0] A, B;
wire [3:0] S;

initial begin
    $dumpfile("somador.vcd");
    $dumpvars;

    A = 4'd7;
    B = 4'd5;
    #1
    if(S == 4'd12)
        $display("Correto");
    else
        $display("Incorreto");

    $finish;
end

somador U1(
    .A(A),
    .B(B),
    .S(S)
);

endmodule
```

Listing 3.3: Testbench para o Somador - arquivo testbench_2.v

Para simular o design acima, basta rodar os seguintes comandos:

```
$ iverilog -o somador.o -s tb_somador testbench_2.v somador.v
$ vvp somador.o
```

3.1.1 Flags Úteis do IVerilog

Aqui estão algumas *flags* importantes para o uso do IVerilog:

1. `-s`: Define o *top module* da simulação.

2. `-o`: Especifica o arquivo de saída.
3. `-g`: Permite definir a versão da linguagem Verilog. As opções incluem: `-g1995`, `-g2001`, `-g2005`, `-g2005-sv`, `-g2009`, `-g2012`.
4. `-D`: Permite definir macros de pré-processador para o código Verilog. Pode ser útil para passar variáveis de ambiente.
5. `-I`: Adiciona diretórios à lista de diretórios onde o IVerilog procura por arquivos de inclusão (include).

Essas opções podem ser combinadas conforme necessário, dependendo da complexidade do projeto. A *flag* -D, por exemplo, é muito útil quando se deseja passar variáveis de ambiente diretamente na linha de comando, como demonstrado abaixo:

```
$ iverilog -DDEBUG_MODE=1 -o debug.o testbench_debug.v
```

Aqui, a variável DEBUG_MODE é definida como 1, permitindo comportamentos condicionais no código com base nessa variável.

3.2 Utilizando o GTKWave

Ao realizar uma simulação com o IVerilog, o arquivo gerado no formato VCD (*Value Change Dump*) pode ser visualizado utilizando o *GTKWave*, uma ferramenta de visualização de formas de onda.

Para abrir o arquivo gerado pela simulação, utilize o comando:

```
$ gtkwave test.vcd
```

O *GTKWave* permite que você inspecione os sinais do design ao longo do tempo, facilitando a depuração e compreensão do comportamento do circuito.

Principais Recursos do GTKWave:

1. Zoom: Aproximação e afastamento para observar sinais em detalhes.
2. Sinal de Interesse: Adição e remoção de sinais à janela principal para análise.
3. Bus de Sinais: Análise de múltiplos sinais em conjunto.
4. Anotação de Tempo: Adição de marcadores de tempo para facilitar o rastreamento de eventos na simulação.

A Figura 3.1 mostra um exemplo de utilização do GTKWave.

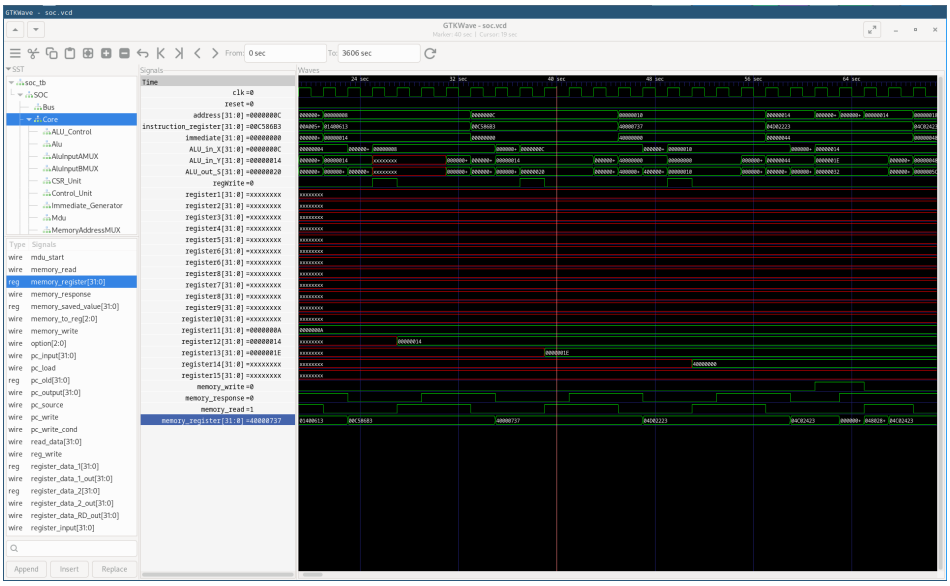

Figura 3.1: Captura de tela do GTKWave

Capítulo 4

Iniciando o Desenvolvimento

4.1 Introdução

Até o momento, as únicas atividades realizadas foram a instalação e configuração de ferramentas, bem como a utilização de ferramentas de simulação. Neste capítulo, finalmente utilizaremos a FPGA.

4.2 Definindo *constraints*

Um dos principais tipos de células disponíveis em FPGAs são as células de entrada e saída. Por meio dessas células, conseguimos fazer com que nosso design interaja com o ambiente externo, recebendo e enviando sinais.

Para utilizarmos pinos físicos da FPGA em nossos designs, precisamos informar à ferramenta de roteamento quais pinos desejamos utilizar e quais nomes daremos a eles em nosso design. Além disso, é necessário especificar informações como a tensão de operação e a frequência, caso se trate de um pino de *clock*. Isso se traduz na seguinte diretriz: utilizaremos determinado pino, queremos que ele tenha um determinado alias, terá a tensão X e, se um sinal de *clock* passar por ele, sua frequência será Y.

Para as FPGAs da família ECP5 da Lattice, conseguimos definir essas ligações, que chamamos de *constraints*, através de um arquivo LPF. Um exemplo de arquivo LPF pode ser visto abaixo:

```
1 LOCATE COMP "clk" SITE "P3";
2 IOBUFFER PORT "clk" IO_TYPE=LVCMOS33;
3 FREQUENCY PORT "clk" 25.0 MHz;
4
5 LOCATE COMP "reset" SITE "P4";
6 IOBUFFER PORT "reset" IO_TYPE=LVCMOS33;
7
8 LOCATE COMP "led" SITE "L2";
9 IOBUFFER PORT "led" IO_TYPE=LVCMOS33;
```

Listing 4.1: Exemplo de arquivo .lpf

Como pode ser observado no exemplo acima, definimos o nome que será atribuído a um pino físico da FPGA através da diretiva LOCATE COMP ”nome a ser dado ao pino”SITE ”pino físico”. Em seguida, definimos o tipo do pino com IOBUFFER PORT ”nome a ser dado ao pino”IO_TYPE=tipo, e, se necessário, especificamos a frequência que passará pelo pino com FREQUENCY PORT ”nome a ser dado ao pino”frequência.

4.3 Um breve *Blink*

Para esclarecer a utilização das *constraints* em nosso design, construiremos um exemplo onde piscamos o LED *onboard* da placa, localizado no pino L2.

```
LOCATE COMP "clk" SITE "P3";
IOBUFFER PORT "clk" IO_TYPE=LVCMOS33;
FREQUENCY PORT "clk" 25.0 MHz;

LOCATE COMP "led" SITE "L2";
IOBUFFER PORT "led" IO_TYPE=LVCMOS33;
```

Listing 4.2: Arquivo blink.lpf

```
module top (
    input wire clk,
    output reg led
);

reg [31:0] counter;

initial begin
    counter = 32'h0;
    led = 1'b0;
end

always @(posedge clk) begin
    if (counter < 32'd12500000) begin
        counter <= counter + 1'b1;
    end else begin
        counter <= 32'h0;
        led <= ~led;
    end
end

endmodule
```

Listing 4.3: Arquivo blink.v

Como podemos ver no exemplo acima, o nome dado ao pino no arquivo LPF é utilizado no design em Verilog. A definição de se o pino atuará como entrada ou saída fica a cargo do design e das limitações físicas da placa. Assim, ao utilizar um pino específico, é sempre aconselhável consultar o *datasheet* e verificar se o pino não está associado a algum recurso específico ou possui alguma propriedade particular.

4.4 Construindo o *Blink*

Na seção anterior, construímos um pequeno circuito que faz um LED piscar. Nesta seção, utilizaremos as ferramentas *open-source* para sintetizar este circuito e carregá-lo na FPGA.

Para realizar a síntese do circuito, utilizaremos o seguinte comando:

```
$ yosys -p "read_verilog␣blink.v;␣synth_ecp5␣-json␣out.json␣-abc9"
```

Ao invocarmos o Yosys com a flag -p, estamos solicitando a execução dos comandos passados como parâmetros. Poderíamos realizar essa ação invocando o Yosys sem utilizar a flag -p, digitando comando por comando através de um *shell* disponibilizado pelo Yosys, ou através de um *script* TCL, utilizando a flag -c e passando o caminho para o script como parâmetro. Observando os comandos passados ao Yosys, temos um comando read_verilog e um comando synth_ecp5. O comando read_verilog é responsável por ler um arquivo Verilog; assim, poderíamos utilizá-lo várias vezes para ler múltiplos arquivos Verilog. O synth_ecp5 é responsável por iniciar a síntese. Ao passarmos a *flag* -json out.json, estamos informando que queremos que a *netlist* resultante da síntese seja salva no arquivo out.json. Por fim, a *flag* -abc9 solicita que a versão mais recente do ABC seja utilizada. Caso não utilizemos essa *flag*, apenas a versão mais antiga será utilizada. A versão mais recente do ABC é mais eficiente na simplificação da *netlist*, mas, ao mesmo tempo, pode ser mais custosa e demorada em termos de tempo de síntese.

Após gerar a *netlist* com o Yosys, utilizaremos o Nextpnr para realizar o posicionamento e roteamento. Para isso, utilizaremos o comando abaixo:

```
$ nextpnr-ecp5 --json out.json --lpf blink.lpf --textcfg out.config \
    --package CABGA381 --45k --speed 6
```

Ao invocar o Nextpnr-ECP5 acima, fornecemos como entrada a *netlist* contida no arquivo out.json, gerada pelo Yosys. Temos como saída um arquivo out.config, que contém a configuração da FPGA. Além disso, utilizamos a *flag* –package para definir o encapsulamento da FPGA. No caso da Colorlight i9, esse encapsulamento é o CABGA381. Passamos a *flag* –45k para definir a arquitetura da FPGA; caso fosse a Colorlight i5, por exemplo, seria –25k. Passamos também o arquivo com as *constraints* com a *flag* –lpf e o limite de frequência do FPGA com –speed 6. Essa constante é interna ao NextPnR e seu valor está relacionado ao modelo da FPGA; para a FPGA utilizada na Colorlight i9, o valor padrão é 6.

Com o arquivo de configuração pronto, podemos gerar nosso *bitstream* utilizando o Ecppack.

```
$ ecppack --input out.config --bit out.bit
```

Como opcional podemos passar ao ecppack a *flag* –compress, está *flag* realiza a compreensão do *bitstream* e diminui drasticamente o tempo de carregamento do mesmo na FPGA.

Com o *bitstream* em mãos, basta carregá-lo na FPGA com o OpenFPGALoader, utilizando:

```
$ openFPGALoader -b colorlight-i9 out.bit
```

Após todo esse processo, se tudo ocorreu bem, o LED da FPGA estará piscando.

4.4.1 Automatizando o Processo

O processo de síntese descrito anteriormente é realizado completamente por linha de comando. Por esse motivo, pode ser automatizado com a utilização de um Makefile. O exemplo abaixo ilustra um Makefile que permite essa automação:

```
all: out.bit

out.bit: out.config
    ecppack --compress --input out.config --bit out.bit

out.config: out.json blink.lpf
    nextpnr-ecp5 --json out.json --lpf blink.lpf --textcfg out.config \
    --package CABGA381 --45k --speed 6

out.json: blink.v
    yosys -p "read_verilog blink.v; synth_ecp5 -json out.json -abc9"

load:
    openFPGALoader -b colorlight-i9 out.bit
```

Listing 4.4: Makefile para automatização do processo de síntese.

Com o Makefile acima, é possível realizar todo o processo de síntese, posicionamento, roteamento e geração do *bitstream* com um simples comando no terminal. Ao digitar `make`, você iniciará automaticamente todas as etapas necessárias até a geração do arquivo `out.bit`. Para carregar o *bitstream* na FPGA, basta utilizar o comando `make load`. Essa abordagem não só simplifica o fluxo de trabalho, mas também minimiza a possibilidade de erros, garantindo que cada etapa do processo seja executada de maneira sequencial e na ordem correta.

4.5 Utilizando PLLs

Em muitos cenários, é necessário trabalhar com uma frequência de *clock* diferente da fornecida como referência pela placa de desenvolvimento. Para obter frequências de *clock* ajustadas, utilizamos PLLs (*Phase-Locked Loops*), que permitem multiplicar ou dividir a frequência de um *clock* de entrada.

O projeto Trellis oferece uma ferramenta chamada `ecppll`, que facilita a geração de configurações de PLL. Com essa ferramenta, é possível especificar um *clock* de entrada e o *clock* de saída desejado, e o `ecppll` gera um arquivo Verilog com a configuração do PLL que melhor se aproxima da frequência solicitada.

Abaixo, e mostrado um exemplo do uso da ferramenta para gerar um *clock* de 50 MHz a partir de um *clock* de referência de 25 MHz, com o resultado salvo no arquivo `pll_50.v`.

```
$ ecppll -i 25 --clkin_name clki --clkout0_name clko -o 50 -f pll_50.v
```

A seguir, o código Verilog gerado pelo `ecppll`:

```
module pll
(
    input clki, // 25 MHz, 0 deg
    output clko, // 50 MHz, 0 deg
    output locked
);
(* FREQUENCY_PIN_CLKI="25" *)
(* FREQUENCY_PIN_CLKOP="50" *)
(* ICP_CURRENT="12" *) (* LPF_RESISTOR="8" *) (* MFG_ENABLE_FILTEROPAMP="1" *) (*
    MFG_GMCREF_SEL="2" *)
EHXPLLL #(
        .PLLRST_ENA("DISABLED"),
        .INTFB_WAKE("DISABLED"),
        .STDBY_ENABLE("DISABLED"),
        .DPHASE_SOURCE("DISABLED"),
        .OUTDIVIDER_MUXA("DIVA"),
        .OUTDIVIDER_MUXB("DIVB"),
        .OUTDIVIDER_MUXC("DIVC"),
        .OUTDIVIDER_MUXD("DIVD"),
        .CLKI_DIV(1),
        .CLKOP_ENABLE("ENABLED"),
        .CLKOP_DIV(12),
        .CLKOP_CPHASE(5),
        .CLKOP_FPHASE(0),
        .FEEDBK_PATH("CLKOP"),
        .CLKFB_DIV(2)
    ) pll_i (
        .RST(1'b0),
        .STDBY(1'b0),
        .CLKI(clki),
        .CLKOP(clko),
        .CLKFB(clko),
        .LOCK(locked)
    );
endmodule
```

Listing 4.5: Arquivo pll_50.v

O `ecppll` suporta a configuração de até quatro *clocks* de saída. A ferramenta oferece diversos parâmetros, sendo os principais:

- `-i`: define a frequência do *clock* de entrada.
- `--clkin_name`: define o nome do sinal de entrada.
- `-o`: define a frequência do *clock* de saída 0.
- `--clkountN`: define a frequência do *clock* de saída N, onde `N` é o número da saída (0 a 3).
- `--clkout0_name`: define o nome do *clock* de saída 0. Para configurar saídas adicionais (até 3), use `clkoutN_name`, onde `N` é o número da saída (0 a 3).
- `-f`: especifica o arquivo de saída.
- `--highres`: aumenta a precisão da frequência gerada.

4.6 Funcionalidades Úteis do Yosys e do NextPnR

Além das funcionalidades abordadas nos exemplos anteriores, tanto o Yosys quanto o NextPnR oferecem uma série de outras opções que podem ser extremamente úteis no desenvolvimento de projetos. Nas subseções a seguir, está destacado algumas *flags* e comandos adicionais que permitem maior controle e flexibilidade durante o processo de síntese e implementação de circuitos.

4.6.1 Flags Adicionais e Comandos Úteis do Yosys

- `-noabc`: O Yosys normalmente usa o otimizador ABC para simplificar a lógica combinacional. No entanto, em alguns casos, pode ser útil desativar essa etapa. A flag `-noabc` pode ser utilizada para evitar a execução do ABC.

```
$ yosys -p "read_verilog mydesign.v; synth_ecp5 \
    -json out.json -noabc"
```

- `-flatten`: O comando `flatten` no Yosys pode ser usado para converter módulos hierárquicos em um único módulo, simplificando a análise de circuitos mais complexos.

```
$ yosys -p "read_verilog top.v; hierarchy -check; flatten; \
    synth_ecp5 -json out.json"
```

- `write_json` e `write_verilog`: Após a síntese, é possível exportar o circuito sintetizado em diferentes formatos. O comando `write_json` gera uma *netlist* em formato JSON, enquanto `write_verilog` escreve uma versão sintetizada do Verilog.

```
$ yosys -p "synth_ecp5; write_json netlist.json"
```

- `-D<macro>`: O Yosys permite passar macros definidos na linha de comando usando a *flag* `-D`. Isso é útil para definir parâmetros de tempo de compilação, como constantes globais ou configurações específicas do projeto, diretamente no fluxo de síntese, sem precisar alterar o código Verilog.

```
$ yosys -DDEBUG=1 -p "synth_ecp5 -json out.json"
```

- `chparam`: Para ajustar parâmetros de módulos Verilog sem modificar o código original, o comando `chparam` pode ser usado para alterar parâmetros diretamente no fluxo de síntese.

```
$ yosys -p "chparam -set WIDTH 16 mymodule; \
    synth_ecp5 -json out.json"
```

- `show`: O Yosys possui um comando visual que gera um diagrama do circuito após a síntese. O comando `show` permite a visualização da netlist em formato gráfico, ajudando a inspecionar o design sintetizado.

```
$ yosys -p "synth_ecp5; show"
```

4.6.2 *Flags* Adicionais e Casos de Uso do NextPnR

- `--timing-allow-fail`: Essa *flag* é utilizada para permitir que o processo de *place and route* continue, mesmo que não atinja as metas de *timing*. Isso é útil em designs experimentais ou quando se deseja uma primeira visualização do resultado.

```
$ nextpnr-ecp5 --json out.json --lpf constraints.lpf \
    --textcfg out.config --timing-allow-fail
```

- `--seed`: A *flag* `--seed` pode ser utilizada para fornecer uma semente ao algoritmo de roteamento, o que pode ajudar a obter diferentes resultados de roteamento em execuções subsequentes, caso o primeiro não seja satisfatório.

```
$ nextpnr-ecp5 --json out.json --lpf constraints.lpf \
    --textcfg out.config --seed 42
```

- `--placer heap`: O NextPnR suporta diferentes algoritmos de posicionamento. A *flag* `--placer heap` utiliza um algoritmo baseado em *heap*, que pode ser mais eficiente em certos designs.

```
$ nextpnr-ecp5 --json out.json --lpf constraints.lpf \
    --textcfg out.config --placer heap
```

- `--lpf-allow-unconstrained`: Essa *flag* permite que o NextPnR continue o processo, mesmo se existirem sinais que não estão explicitamente definidos no arquivo de restrições *constraints*. Isso pode ser útil em designs onde nem todos os sinais estão restritos ou quando se está desenvolvendo um protótipo.

```
$ nextpnr-ecp5 --json out.json --lpf constraints.lpf \
    --textcfg out.config --lpf-allow-unconstrained
```

- `--report`: Gera relatórios detalhados sobre o design, incluindo o uso de recursos e a análise de *timing*. Isso é útil para verificar a eficiência do posicionamento e para ajustar o design para melhorias.

```
$ nextpnr-ecp5 --json out.json --lpf constraints.lpf \
    --textcfg out.config --report detailed_report.txt
```

- `--detailed-timing-report`: Essa *flag* gera um relatório de *timing* detalhado, que pode ajudar na análise de desempenho, revelando atrasos críticos no design.

```
$ nextpnr-ecp5 --json out.json --lpf constraints.lpf \
    --textcfg out.config --detailed-timing-report
```

- `--analysis`: Essa *flag* ativa uma análise adicional no design, fornecendo mais detalhes sobre possíveis problemas de *timing* ou sobre a alocação de recursos.

```
$ nextpnr-ecp5 --json out.json --lpf constraints.lpf \
    --textcfg out.config --analysis
```

Capítulo 5

Aplicações Práticas e Casos de Uso

5.1 Exemplos Interessantes

Com a Colorlight i9 e um fluxo de trabalho completamente *open-source*, podemos explorar diversas aplicações práticas que permitem a utilização efetiva dos recursos da placa. Abaixo estão alguns exemplos que podem servir como inspiração:

1. Controle de brilho de um LED utilizando PWM.
2. Reprodução de músicas via PWM: `https://github.com/JN513/Simple-FPGA-Music-Player-with-PWM`.
3. Integração de um ADC externo.
4. Implementação de um eco na UART.
5. Criação de um dispositivo SPI-Slave: `https://github.com/Unicamp-Odhin/SPI-Slave`.
6. Desenvolvimento de um dispositivo SPI-Master: `https://github.com/Unicamp-Odhin/SPI-Master`.
7. Comunicação entre duas FPGAs utilizando SPI.
8. Exibição de imagens em um monitor via HDMI.
9. Utilizacão de um RISC-V na FPGA: `https://github.com/JN513/Risco-5`.

5.2 Utilizando o LiteX

O LiteX é um framework robusto que facilita a criação e uso de cores e sistemas em chip (SoCs) em FPGAs. Ele oferece suporte a várias FPGAs e inclui uma rica biblioteca de IPs e cores *open-source*.

5.2.1 Instalando o LiteX

A instalação do LiteX em um sistema Linux é um processo simples. Recomenda-se criar uma pasta dedicada chamada eda (Electronic Design Automation) para gerenciar seus projetos. Para instalar o LiteX, siga os passos abaixo, conforme indicado na documentação oficial:

```
$ cd
$ mkdir -p eda/litex
$ cd eda/litex
$ python3 -m venv litex_env
$ wget https://raw.githubusercontent.com/enjoy-digital/\
litex/master/litex_setup.py
$ . litex_env/bin/activate
$ python3 ./litex_setup.py --init --install --config=standard
$ echo 'alias␣get_litex=".␣$HOME/eda/litex/litex_env/bin/activate"'\
>> ~/.bashrc
```

```
$ echo 'alias get_litex=". $HOME/eda/litex/litex_env/bin/activate"'\
>> ~/.zshrc
```

A instalação ocorre em um ambiente virtual Python, e para utilizá-lo, você deve ativar o ambiente com o comando get_litex. O processo pode levar alguns minutos, pois o LiteX clona diversos repositórios necessários para o funcionamento do framework e das ferramentas associadas.

Para aqueles que desejam se aprofundar no LiteX, os desenvolvedores disponibilizam alguns tutoriais: `https://github.com/litex-hub/fpga_101`.

5.2.2 Utilizando o LiteX com a Colorlight i9

Durante a instalação do LiteX, um diretório chamado litex_boards é criado. Este diretório contém definições para as placas FPGA suportadas pelo LiteX, além de exemplos de utilização.

Para nosso teste, utilizaremos o arquivo colorlight_i5.py, que pode ser encontrado em: litex-boards/litex_boards/targets. Navegue até este diretório com o seguinte comando:

```
$ cd ~/eda/litex/litex-boards/litex_boards/targets
```

Uma vez no diretório, ative o ambiente do LiteX e execute o script Python colorlight_i5.py com as opções indicadas abaixo:

```
$ get_litex
$ python3 colorlight_i5.py --board i9 --revision 7.2 --build --load \
    --with-video-framebuffer
```

Se tudo ocorrer como esperado, ao conectar a FPGA a um monitor HDMI, algumas cores serão exibidas na tela.

5.2.3 Linux on VexRiscV

O LiteX também oferece uma demonstração do Linux em execução no processador RISC-V VexRiscV para FPGAs, disponível em `https://github.com/litex-hub/linux-on-litex-vexriscv`. Essa implementação representa uma excelente oportunidade para explorar o funcionamento do Linux em um ambiente FPGA, ampliando as possibilidades de uso da Colorlight i9. Ao testar a demonstração, você terá acesso a um shell Linux através da UART e verá uma imagem semelhante à da Figura 5.1 na saída de vídeo da FPGA.

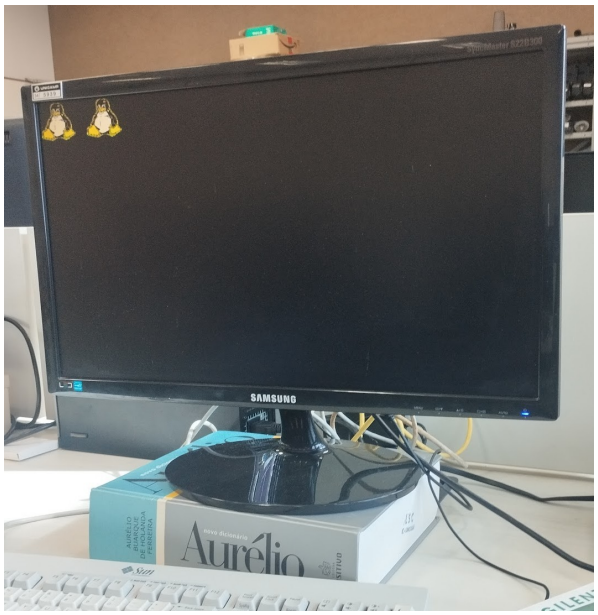

Figura 5.1: Saída de vídeo gerada pelo Linux rodando no softcore na FPGA.

Capítulo 6

Apêndice

6.1 Utilizando FPGAs Gowin

6.2 Utilizando FPGAs Xilinx

6.3 Desenvolvendo um Chip com TinyTapeout