SAPHO & AURORA

Manual de uso e referência técnica

Versão 6.4.2

NIPS-CERN

Núcleo de Inovação e Pesquisa em Sistemas Computacionais

Agosto de 2026

Sumário

Parte I

Primeiros passos

O que é o SAPHO

SAPHO (*Scalable-Architecture Processor for Hardware Optimization*) é uma plataforma para criar processadores dedicados em FPGA. Você descreve um algoritmo em uma linguagem parecida com C, e a plataforma gera um processador em Verilog dimensionado para aquele algoritmo: com a largura de palavra que você escolheu e apenas os blocos de hardware que o programa realmente usa.

A plataforma também serve para o caminho inverso: escrever Verilog à mão, validar, simular e visualizar, tudo na mesma janela. É assim que este manual começa.

1.1 As peças

A IDE. O programa que você instala e abre. Nela vivem o editor, o gerenciador de projetos, os botões de compilação e simulação, os terminais e os visualizadores. É a única interface gráfica da plataforma.

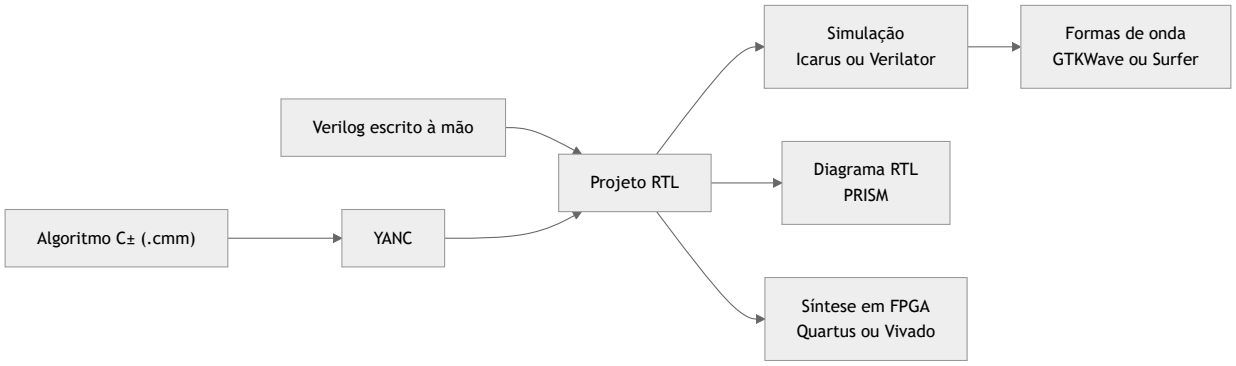
A suíte de compiladores que trabalha por baixo. Traduz o programa C± no processador em Verilog, nas imagens de memória e no testbench. Você nunca a chama diretamente.

O circuito que o YANC emite: acumulador único, arquitetura Harvard, pipeline de três estágios. Só os blocos que o seu programa usa são sintetizados.

O dialeto de C no qual você escreve o algoritmo. Arquivos com extensão `.cmm`. A versão essencial está no capítulo *A linguagem C± essencial*; os recursos de pós-graduação, nos *estudos avançados*.

Em volta desse núcleo, a instalação traz as ferramentas de apoio, todas de código aberto: Icarus Verilog e Verilator para simulação, GTKWave e Surfer para formas de onda, Yosys para análise estrutural, PRISM para visualizar o circuito, cocotb para testes em Python.

1.2 O caminho completo



A única etapa fora da AURORA é a última: levar o Verilog e as imagens de memória à ferramenta do fabricante do FPGA para a gravação física. A AURORA valida, simula e desenha o circuito, mas não gera bitstream.

1.3 Como este manual está organizado

O manual segue a ordem de uma disciplina: primeiro o fluxo Verilog puro (Parte II), depois a criação de processadores em C± (Parte III), depois os dois juntos no mesmo projeto (Parte IV). A Parte V reúne os estudos avançados, voltados à pós-graduação: ponto flutuante configurável, números complexos, notação de Dirac, FFT e a arquitetura interna do processador.

Ver também

A arquitetura e os resultados em FPGA estão no artigo da plataforma, SAPHO: An FPGA Customizable Implementation (IEEE, 2026)[1]. Mais leituras em *Publicações*.

Se esta é a sua primeira vez, siga em frente: *Instalação*.

[1] https://cdn.nipscern.com/publications/ieee-2026-soft-core-processors.pdf

Capítulo 2

Instalação

O SAPHO roda em Windows 10 e 11, 64 bits. A instalação traz tudo: a AURORA, os compiladores do YANC, os simuladores, os visualizadores de onda e um Python embarcado. Não é preciso instalar nada além do pacote.

2.1 Baixar

Baixe o instalador no site do NIPS-CERN:

https://www.nipscern.com/projects/sapho

O botão *Download Latest Release* baixa diretamente o instalador da versão mais recente, um arquivo com nome no formato `sapho-aurora-Setup-vX.Y.Z.exe`, com cerca de 500 MB. Se preferir, as versões anteriores ficam na página de releases do GitHub, no mesmo lugar.

2.2 Instalar

1. Execute o instalador baixado.
2. Se o Windows exibir o aviso do SmartScreen dizendo que o aplicativo não é reconhecido, clique em *Mais informações* e depois em *Executar assim mesmo*. O aviso aparece porque o executável ainda não carrega assinatura digital; a assinatura pela SignPath Foundation está em andamento e o aviso deixará de existir.
3. Siga o assistente: aceite a licença, escolha a pasta de destino e conclua.

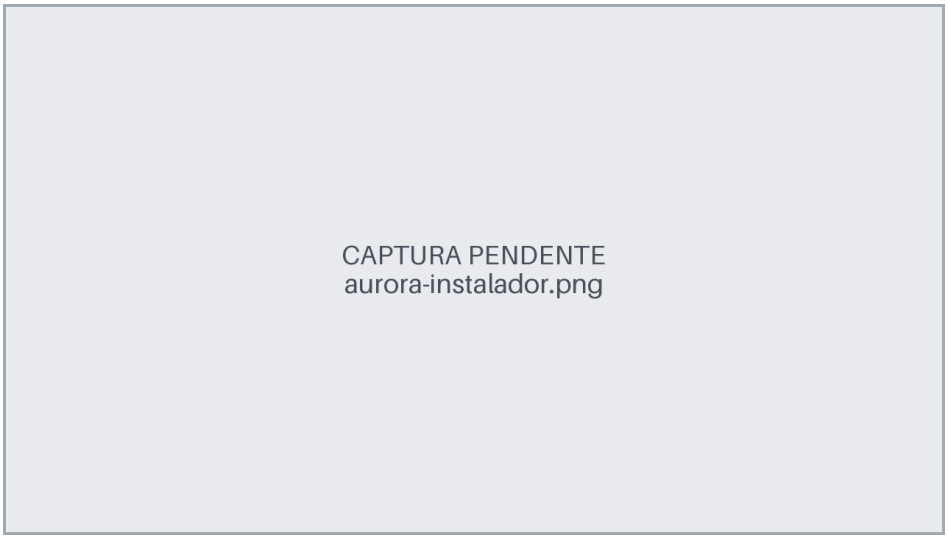


Figura2.1: O assistente de instalação. O padrão instala para o usuário atual, sem exigir privilégios de administrador.

> **⚠ Aviso**
>
> Não execute o instalador como administrador. A atualização automática usa o mesmo caminho da instalação, e uma instalação feita como administrador impede o atualizador de trabalhar depois.

2.3 Primeiro início

Abra o SAPHO pelo menu Iniciar. Após a tela de abertura, você chega à tela de boas-vindas:

Figura2.2: A tela de boas-vindas. À esquerda, criar ou abrir projeto; à direita, a lista de projetos recentes, vazia no primeiro uso.

Confirme que está tudo pronto:

- A barra de status, no rodapé, mostra *Não Pronto*. É o esperado sem projeto aberto; clicar nela abre o diálogo de projeto.
- Em *Configurações* (ícone de controles deslizantes na barra superior), a aba *Sobre* mostra a versão instalada e a situação do atualizador.

2.4 Idioma

A interface nasce em inglês ou português conforme o sistema. Para trocar: *Configurações*, aba *Idioma*. A escolha vale também para as mensagens dos compiladores, que falam português ou inglês conforme essa opção.

2.5 Atualizações

A AURORA verifica atualizações sozinha, alguns segundos após abrir e depois a cada três horas. Quando há versão nova, uma janela mostra as novidades e pergunta se você quer baixar; nada é baixado sem a sua confirmação. Depois de baixada, a atualização se aplica ao fechar o aplicativo.

Próximo passo: *O mapa da janela*.

Capítulo 3

O mapa da janela

Antes do primeiro projeto, só o essencial: onde fica cada coisa. Os detalhes de cada região aparecem nos tutoriais, no momento em que você os usa, e o tour completo está em *Tour pela interface*.

Figura3.1: A janela com um projeto aberto.

Cinco regiões:

Barra de ferramentas
: No topo. Projeto à esquerda; os botões de compilar, sintetizar, simular e abrir o PRISM no centro; controle de versão, Python, configurações e a assistente de IA à direita. Botão cinza é pré-requisito faltando, e o tooltip diz qual.

Árvore de arquivos
: À esquerda. Três visões alternáveis: Arquivos (os fontes por papel), Pastas (o disco) e Hierarquia (as instâncias do design).

Editor
: No centro. Abas, realce para todas as linguagens da plataforma, análise de Verilog enquanto digita.

Terminais
: Embaixo. Uma aba por etapa do fluxo, mais um PowerShell de verdade (TCMD). Mensagens classificadas e clicáveis.

Barra de status
: No rodapé. O resumo do estado: projeto pronto, processador ativo, Top Level, Testbench Top e motor de simulação.

É o suficiente para começar. Siga para o primeiro tutorial: *Tutorial: um contador em Verilog*.

Parte II

Verilog primeiro

Tutorial: um contador em Verilog

Este é o primeiro tutorial do manual. Ao final, você terá criado um projeto, escrito um módulo Verilog e um testbench, validado o design, simulado e lido o resultado na forma de onda. Reserve uns vinte minutos.

O que vamos construir

Um contador de 4 bits com habilitação: conta a cada borda de clock enquanto `habilita` estiver em 1, e zera no reset. É o circuito sequencial mais simples que exercita o fluxo inteiro: clock, reset, registrador e testbench.

4.1 Passo 1: criar o projeto

1. Clique em *Novo Projeto*, na barra superior ou na tela de boas-vindas.
2. Nomeie `LabContador`. O nome aceita letras, números, sublinhado e hífen; sem espaços nem acentos.
3. Clique em *Procurar*, escolha uma pasta com permissão de escrita e clique em *Gerar Projeto*.

Figura4.1: O formulário pede só nome e local. O campo de local é preenchido pelo botão Procurar.

Confira: a árvore mostra `LabContador`, vazio, e a barra de status passou a *Pronto*.

4.2 Passo 2: escrever o módulo

1. Na árvore, clique com o botão direito na área vazia e escolha *Novo arquivo*.
2. Nomeie `contador.v` e salve dentro da pasta do projeto.
3. Digite o módulo:

Listing 4.1: contador.v

```
module contador (
    input  wire       clk,
    input  wire       rst,
    input  wire       habilita,
    output reg  [3:0] conta
);

    always @(posedge clk) begin
        if (rst)
            conta <= 4'd0;
        else if (habilita)
            conta <= conta + 4'd1;
    end

endmodule
```

Salve com `Ctrl+S`.

Figura4.2: Enquanto você digita, dois analisadores conferem o código: erros de sintaxe aparecem sublinhados na hora, e a análise semântica aponta sinais não declarados e portas incompatíveis.

Confira: na visão Arquivos, `contador.v` apareceu com o ícone de fonte sintetizável. A AURORA o classificou sozinha, lendo o conteúdo: sem `initial`, sem `$finish`, é circuito.

4.3 Passo 3: escrever o testbench

Crie um segundo arquivo, `tb_contador.v`:

Listing 4.2: tb_contador.v

```
`timescale 1ns/1ps

module tb_contador;

```

(continua na próxima página)

(continuação da página anterior)

```
    reg        clk = 0;
    reg        rst = 1;
    reg        habilita = 0;
    wire [3:0] conta;

    contador dut (
        .clk      (clk),
        .rst      (rst),
        .habilita (habilita),
        .conta    (conta)
    );

    always #5 clk = ~clk;      // clock de 100 MHz

    initial begin
        $dumpfile("tb_contador.fst");
        $dumpvars(0, tb_contador);

        #12 rst = 0;           // solta o reset fora da borda
        #8  habilita = 1;      // conta por 20 ciclos
        #200 habilita = 0;     // congela
        #40 $finish;
    end

endmodule
```

Repare nos ingredientes que todo testbench tem: o clock gerado por `always #5`, o reset solto depois de alguns nanossegundos, o `$dumpvars` que grava os sinais para a forma de onda e o `$finish` que encerra a simulação.

Confira: o arquivo entrou na árvore com o ícone de testbench. `initial`, `$finish` e o nome começando com `tb_` denunciaram a categoria.

4.4 Passo 4: definir os papéis

1. Botão direito em `contador.v`, escolha *Definir como Top Level*.
2. Botão direito em `tb_contador.v`, escolha *Marcar como Testbench*.

Confira: a barra de status agora mostra os dois nomes, e os botões *Sintetizar Verilog* e *Analisar Verilog* acenderam.

4.5 Passo 5: validar

Clique em *Sintetizar Verilog*.

A AURORA elabora o projeto inteiro a partir do Top Level: resolve os módulos, confere portas e conexões, e monta a hierarquia de instâncias. Não é a síntese física do FPGA; é a prova de que o design fecha.

Confira: alterne a árvore para a visão *Hierarquia*. Deve aparecer a instância do contador. Clicar nela abre o fonte na linha da definição.

Figura4.3: O terminal TVERI relata a validação. Depois dela, o botão de visões da árvore ganha a opção Hierarquia.

4.6 Passo 6: simular e ler a onda

1. Confirme na barra de ferramentas: simulador **Icarus Verilog**, visualizador **GTKWave**.
2. Clique em *Analisar Verilog*.

A AURORA compila os fontes com o testbench, roda a simulação e abre o GTKWave com os sinais já organizados. Acompanhe o terminal TWAVE.

Procure na onda:

- `rst` alto no início e a contagem presa em 0;
- `conta` subindo um a um a cada borda de subida do clock enquanto `habilita` está em 1;
- o estouro: depois de 15, o contador de 4 bits volta a 0;
- a contagem congelada quando `habilita` cai.

4.7 O que você aprendeu

- Um projeto é uma pasta com um `.spf`; os fontes entram por criação ou arraste.
- A AURORA separa circuito de testbench sozinha, pelo conteúdo do arquivo.
- Top Level e Testbench Top são os dois papéis que você marca, e a barra de status os exibe sempre.
- Sintetizar valida a elaboração e constrói a hierarquia; Analisar simula e abre a onda.
- O testbench manda na simulação: clock, reset, estímulos, `$dumpvars` e `$finish` são seus.

4.8 Um segundo padrão: máquina de estados

O contador é o primeiro padrão sequencial; o segundo é a máquina de estados finitos, e vale construí-la já. Um semáforo simples: verde, amarelo, vermelho, cada um com sua duração.

Crie `semaforo.v` no mesmo projeto:

Figura4.4: A forma de onda do contador: o reset solta, habilita sobe e `conta` incrementa a cada borda de clock, de 0 a 15 e de volta a 0.

Listing 4.3: semaforo.v

```
module semaforo (
    input  wire       clk,
    input  wire       rst,
    output reg  [2:0] luz        // {verde, amarelo, vermelho}
);

    localparam VERDE    = 2'd0;
    localparam AMARELO  = 2'd1;
    localparam VERMELHO = 2'd2;

    localparam T_VERDE    = 4'd8;
    localparam T_AMARELO  = 4'd2;
    localparam T_VERMELHO = 4'd6;

    reg [1:0] estado;
    reg [3:0] tempo;

    always @(posedge clk) begin
        if (rst) begin
            estado <= VERDE;
            tempo  <= 4'd0;
        end else begin
            tempo <= tempo + 4'd1;
            case (estado)
                VERDE:    if (tempo == T_VERDE    - 1) begin estado <= AMARELO;  tempo <= 0; end
                AMARELO:  if (tempo == T_AMARELO  - 1) begin estado <= VERMELHO; tempo <= 0; end
                VERMELHO: if (tempo == T_VERMELHO - 1) begin estado <= VERDE;    tempo <= 0; end
                default:  estado <= VERDE;
```

(continua na próxima página)

(continuação da página anterior)

```
            endcase
        end
    end

    always @(*) begin
        case (estado)
            VERDE:    luz = 3'b100;
            AMARELO:  luz = 3'b010;
            default:  luz = 3'b001;
        endcase
    end

endmodule
```

O padrão em três blocos: os estados nomeados com `localparam`, um `always` sequencial com o registrador de estado e o temporizador, e um `always` combinacional com a saída em função do estado. Escreva um testbench nos moldes do anterior (solte o reset e deixe rodar uns 200 ciclos), marque os papéis e observe na onda o ciclo verde, amarelo, vermelho se repetindo. A visão *Hierarquia* agora mostra dois módulos de topo possíveis; o Top Level marca qual vale.

4.9 Exercícios

1. Faça o contador contar de 0 a 9 e reiniciar, virando um contador de década.
2. Acrescente um pino `sobe_desce` ao contador: 1 conta para cima, 0 para baixo. Confirme na onda os dois sentidos.
3. Dê ao contador uma saída `estouro`, em 1 apenas no ciclo em que a contagem volta a zero. Grave-a na onda.
4. No semáforo, acrescente um pino de pedestre que, pressionado durante o verde, encurta o tempo restante para no máximo 2 ciclos. Cuidado com o caso do tempo já estar abaixo disso.

4.10 Para onde ir

As formas de onda em detalhe estão em *Formas de onda*, o fluxo em *O fluxo Verilog em detalhe* e os testbenches, incluindo cocotb, em *Testbenches: Verilog e cocotb*. Quando quiser gerar um processador em vez de escrever o circuito à mão, siga para a Parte III: *Tutorial: um processador SAPHO*.

Formas de onda

A forma de onda é o osciloscópio do projetista digital: cada sinal ao longo do tempo, ciclo a ciclo. Este capítulo cobre as escolhas que valem para qualquer projeto; o que é específico de processadores SAPHO (as trilhas de assembly e de linha C±) está em *Simulação do processador*.

5.1 Icarus ou Verilator

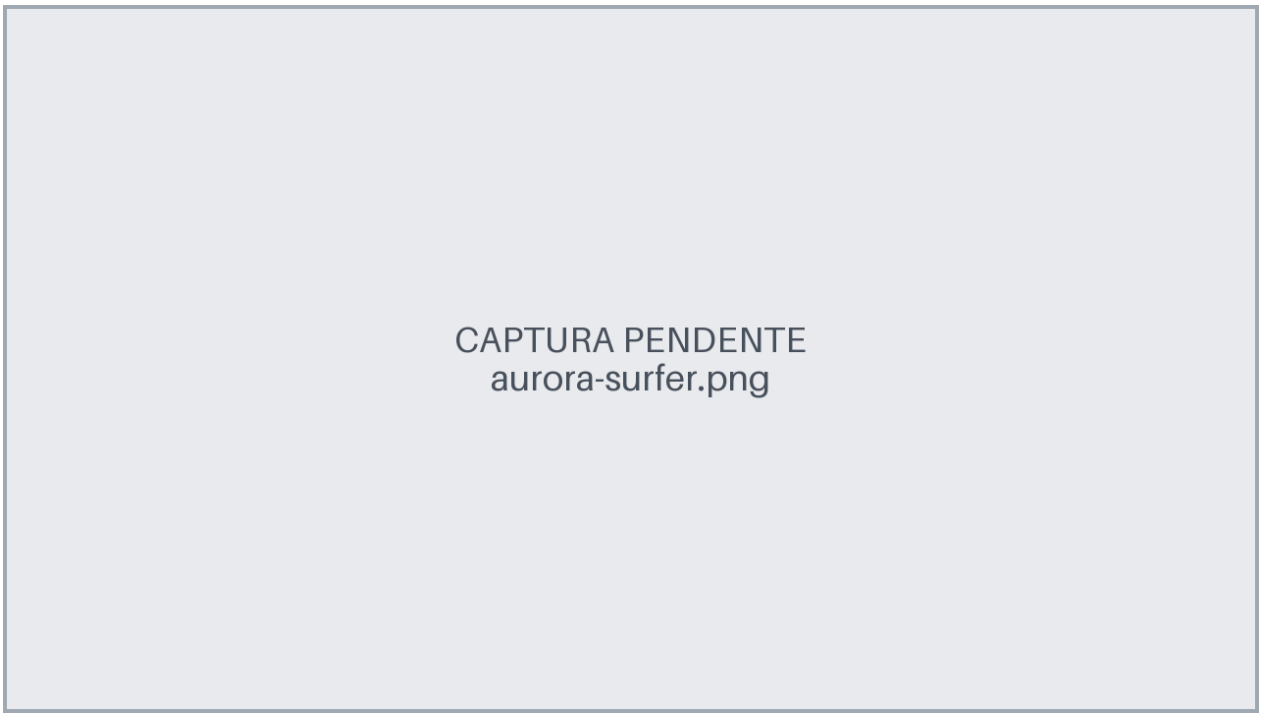
A chave na barra de ferramentas escolhe o motor de simulação:

	Icarus Verilog	Verilator
Velocidade	referência	10 a 100 vezes mais rápido
Sinais visíveis	todos	sinais internos de processadores SAPHO ficam de fora
Uso típico	ondas curtas, depuração fina	testbenches longos, regressões

A troca vale na próxima simulação, e a barra de status mostra o motor atual.

5.2 GTKWave ou Surfer

A segunda chave escolhe o visualizador. O GTKWave é o clássico; o Surfer é o alternativo, moderno e rápido. Os dois abrem em janela própria com o layout preparado pela AURORA: sinais agrupados, nomeados e coloridos, em vez do despejo bruto.



5.3 Escolher o que gravar

O modal *Configuração de ondas* define quais sinais a simulação grava. Menos sinais, simulação mais leve e arquivo menor.

Figura5.1: A árvore reflete a hierarquia do projeto; o filtro aceita texto e expressão regular.

A seleção vale por testbench e segue uma precedência: um arquivo de layout ativo vence a configuração do modal, que vence um `$dumpvars` escrito à mão no testbench, que vence o padrão (tudo no escopo do módulo do testbench).

5.4 Layouts salvos

Organizou os sinais do seu jeito no GTKWave? Salve um `.gtkw` (File, Write Save File) e registre-o no seletor da barra de ferramentas: ele vira o layout daquele testbench. O item *padrão* volta ao layout automático. Com o Surfer, o mesmo seletor gerencia os arquivos de layout dele.

5.5 Exercícios

1. No projeto do contador, abra a *Configuração de ondas*, grave só `clk`, `habilita` e `conta`, e compare o tamanho do arquivo de onda com a gravação completa.
2. No GTKWave, reordene os sinais, troque a base de `conta` para decimal, salve um `.gtkw` e registre-o no

seletor. Rode de novo e confirme que o layout voltou como você deixou.

3. Rode a mesma simulação com Icarus e com Verilator e compare o tempo relatado no terminal TWAVE.

Capítulo 6

O fluxo Verilog em detalhe

O tutorial mostrou o caminho feliz. Este capítulo explica o que cada etapa faz de verdade e o que fazer quando algo foge do roteiro.

6.1 O que "Sintetizar Verilog" executa

O botão roda a elaboração do projeto: todos os fontes sintetizáveis são compilados juntos, com o Top Level como raiz, usando o Icarus Verilog em modo de checagem (elabora sem gerar simulação). A biblioteca de módulos do SAPHO entra automaticamente na busca, então um projeto que instancie um processador gerado não precisa listar os módulos internos dele.

Passando a checagem, o Yosys constrói a hierarquia de instâncias, que alimenta a visão *Hierarquia* da árvore. O testbench fica de fora dessa etapa de propósito: construções de simulação como `$dumpvars`, atrasos e `$finish` não pertencem ao circuito.

> **Nota**
>
> Validar não é sintetizar para FPGA. A AURORA prova que o design elabora e desenha a estrutura; a síntese física, com mapeamento em células e bitstream, acontece no Quartus ou no Vivado, como descrito em *Levar ao FPGA*.

6.2 A classificação automática das fontes

Cada `.v` importado é lido e pontuado: gravação de onda, `$finish` ou `$stop`, módulo sem portas e blocos `initial` pesam mais; `$display`, atrasos `#` e nome terminando em `_tb` pesam menos. Passando do limiar, o arquivo é testbench; na dúvida, é sintetizável. A pontuação se refaz a cada atualização da árvore.

Na prática isso significa: escreva testbenches com cara de testbench (um `initial`, um `$finish`) e módulos com cara de módulo, e a árvore se organiza sozinha. Se um arquivo cair na categoria errada, o menu de contexto permite marcá-lo como testbench manualmente.

6.3 Diagnósticos enquanto você digita

Dois analisadores acompanham a edição de Verilog e SystemVerilog:

- O analisador sintático aponta erros de forma e estilo na hora, e é ele que atende o *Formatar* (`Shift+Alt+F`).
- O analisador semântico elabora o projeto inteiro em segundo plano e marca o que só aparece na elaboração: identificadores não declarados, incompatibilidade de tipos e de portas, sinais nunca usados. Ele pode ser ligado e desligado com `Ctrl+Alt+S`.

Os dois escrevem na mesma lista de problemas do editor, com origens identificadas.

6.4 Múltiplos arquivos e módulos

O conjunto de fontes da elaboração é a lista de sintetizáveis do projeto, inteira. Você pode dividir o design em quantos arquivos quiser; a resolução de módulos cruza todos. O que importa é um único Top Level marcado, do qual a elaboração parte. Módulos não alcançáveis a partir dele simplesmente não participam.

6.5 Quando a validação falha

As mensagens chegam no terminal TVERI com link para arquivo e linha. Os casos mais comuns:

`Unknown module type: X`
: Um módulo instanciado não existe em nenhum fonte do projeto. Confira o nome e se o arquivo que o define está na lista de sintetizáveis.

Portas incompatíveis
: A instância não bate com a definição do módulo. O analisador semântico costuma apontar isso antes mesmo de você clicar no botão.

Botão desabilitado
: Falta o Top Level. Marque-o pelo menu de contexto na visão Arquivos.

A lista completa de mensagens por fluxo está em *Diagnóstico: sintomas e causas*.

Capítulo 7

Testbenches: Verilog e cocotb

Um testbench é o programa que exercita o circuito na simulação: gera clock e reset, aplica estímulos, observa saídas e decide quando parar. Na AURORA ele pode ser escrito em Verilog ou em Python, com o cocotb. Este capítulo cobre os dois.

7.1 Anatomia de um testbench Verilog

O do tutorial serve de modelo:

- **Clock**: `always #5 clk = ~clk;` gera 100 MHz com `timescale 1ns/1ps`.
- **Reset**: comece em 1 e solte fora da borda de clock, para o circuito nascer em estado conhecido.
- **Estímulos**: um bloco `initial` com atrasos `#` sequenciando os eventos.
- **Gravação**: `$dumpfile` e `$dumpvars` escolhem o arquivo e os sinais gravados. Se o seu testbench já os traz, a AURORA os respeita; se não traz, ela os injeta com uma seleção padrão, e o modal *Configuração de ondas* permite refinar sem editar o arquivo.
- **Término**: `$finish` encerra. Sem ele, a simulação corre até o limite e o visualizador abre com o que houver.

> **Dica**
>
> O testbench original nunca é modificado pela AURORA. Quando é preciso instrumentar (para injetar a gravação de ondas, por exemplo), uma cópia é gerada na área temporária e é ela que compila. O seu arquivo permanece como você escreveu.

A escolha entre os motores Icarus e Verilator, os visualizadores e a seleção de sinais estão no capítulo anterior, *Formas de onda*.

7.2 Testbench em Python: cocotb

Com o cocotb, o testbench é um módulo Python: o simulador roda o circuito e o Python dirige os sinais. A vantagem é usar a linguagem inteira na verificação, incluindo bibliotecas de análise.

Crie pelo menu de contexto da árvore: *Novo testbench cocotb (.py)*. O modelo gerado já traz a estrutura:

Listing 7.1: teste_contador.py

```python
# aurora-toplevel: contador

import cocotb
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge


@cocotb.test()
async def conta_ate_quinze(dut):
    cocotb.start_soon(Clock(dut.clk, 10, units="ns").start())

```

(continua na próxima página)

(continuação da página anterior)

```
    dut.rst.value = 1
    dut.habilita.value = 0
    for _ in range(2):
        await RisingEdge(dut.clk)
    dut.rst.value = 0

    dut.habilita.value = 1
    for esperado in range(1, 16):
        await RisingEdge(dut.clk)
        assert dut.conta.value == esperado, (
            f"esperava {esperado}, veio {int(dut.conta.value)}"
        )
```

Figura7.1: A primeira linha importa: o comentário `# aurora-toplevel: contador` diz qual módulo Verilog é o alvo do teste. Fora da AURORA a linha é um comentário inerte.

Marque o `.py` como Testbench Top e use os mesmos botões de sempre: *Analisar Verilog* roda os testes e abre a onda; *Execução rápida* roda sem gravar onda, ideal para o ciclo de ajuste.

Três coisas que valem saber:

- Nada de Makefile: a AURORA monta e executa o projeto cocotb sozinha, com o Python embarcado. Você escreve só as funções `@cocotb.test()`.
- Teste que falha não esconde a onda: se a simulação rodou e as asserções falharam, o erro aparece em vermelho e a forma de onda abre mesmo assim, porque é nela que se investiga.
- Bibliotecas extras (pyuvm, extensões de barramento, análise de VCD) se instalam pelo painel *Bibliotecas Python*, descrito em *Controle de versão, Python e configurações*.

7.3 Verilog ou cocotb?

Para exercícios curtos e primeiros contatos, o testbench Verilog é mais direto: tudo em um arquivo, sem camadas. O cocotb compensa quando a verificação cresce: laços de referência em Python, comparação com modelos, geração de estímulos complexos. Os dois convivem no mesmo projeto; o Testbench Top decide qual roda.

Figura7.2: Na execução rápida, o veredito de cada teste sai no terminal TWAVE.

Parte III

C± e o processador

Capítulo 8

Tutorial: um processador SAPHO

No tutorial anterior você escreveu o circuito à mão. Agora o caminho é outro: descrever um algoritmo em C± e deixar a plataforma gerar o processador. Ao final, você terá criado um processador sob medida, compilado até Verilog, simulado e lido o resultado na onda. Reserve uns trinta minutos.

O que vamos construir

Um filtro de média móvel de quatro amostras: lê um valor pela porta de entrada, guarda as quatro últimas leituras, soma e devolve a média pela porta de saída. É o filtro mais simples do processamento de sinais, e exercita em vinte linhas tudo o que um projeto SAPHO tem: entrada e saída, um vetor na memória, aritmética e um laço infinito.

8.1 Passo 1: criar o projeto

Crie um projeto chamado `MeuFiltro`, como no tutorial anterior: *Novo Projeto*, nome, local, *Gerar Projeto*.

8.2 Passo 2: criar o processador

1. Clique em *Hub de Processadores*, na barra superior.
2. Preencha conforme a tabela e clique em *Gerar Processador*.

Tabela8.1: Valores para este tutorial

Campo	Valor	Por quê
Nome do Processador	`media_movel`	Vira o nome da pasta, do módulo Verilog e da diretiva `#PROCESSOR_NAME`
Total de Bits	`16`	Largura da palavra; 16 bits bastam para sinais de instrumentação
Bits da Mantissa	`10`	Precisão do ponto flutuante
Bits do Expoente	`5`	Faixa do ponto flutuante
Ganho	`128`	Usado só por `norm()`; deve ser potência de dois
Pilha de Instruções	`2`	Profundidade de chamadas de função aninhadas
Pilha de Dados	`4`	Complexidade das expressões
Portas de Entrada	`1`	Uma porta para receber as amostras
Portas de Saída	`1`	Uma porta para publicar a média

Importante

A regra que mais reprova o formulário: o total de bits deve ser a soma da mantissa, do expoente e do bit de sinal. Aqui, $16 = 10 + 5 + 1$. É uma exigência estrutural do hardware de ponto flutuante, explicada em *Ponto flutuante customizado*.

Confira: a árvore mostra `media_movel` com as pastas `Software`, `Hardware` (vazia) e `Simulation` (vazia). O arquivo `media_movel.cmm` nasceu com o cabeçalho preenchido pelo formulário.

Figura8.1: A validação roda enquanto você digita, e o botão só habilita com tudo consistente.

Figura8.2: Um campo fora da regra fica com a borda vermelha e trava o botão.

8.3 Passo 3: escrever o algoritmo

Abra `Software/media_movel.cmm` e substitua o conteúdo por:

Listing 8.1: media_movel.cmm

```
#PRNAME media_movel
#NUBITS 16
#NDSTAC 4
#SDEPTH 2
#NUIOIN 1
#NUIOOU 1
#NBMANT 10
#NBEXPO 5
#NUGAIN 128

void main()
{
    int x[4];    // historico das ultimas 4 amostras
    int soma;

    while (1)
    {
        x[3] = x[2];          // desloca o historico
        x[2] = x[1];
        x[1] = x[0];
        x[0] = in(0);         // le nova amostra da porta 0

        soma = x[0] + x[1] + x[2] + x[3];
        out(0, soma >> 2);    // media = soma/4, sem divisor
    }
}
```

Salve com `Ctrl+S`. Três observações sobre o programa:

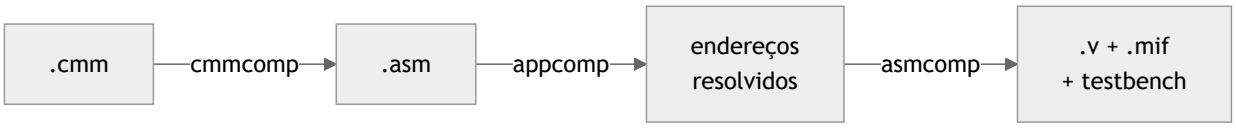
As nove primeiras linhas são diretivas
Não são código executável: configuram o hardware que será gerado. O formulário as escreveu, e a partir de agora o arquivo é a fonte da verdade. Editar uma diretiva muda o processador na próxima compilação.

O `while (1)` é a forma canônica
O programa modela um circuito, e circuitos não terminam: processam amostras para sempre.

A divisão virou deslocamento
`soma >> 2` dá o mesmo resultado de `soma / 4`, mas `/` instanciaria um divisor inteiro, um dos blocos mais caros da ULA. O deslocamento sai quase de graça. Você vai ver essa diferença com os próprios olhos no passo 7.

8.4 Passo 4: compilar

Com o `.cmm` em foco, clique em *Compilar C±*. Três compiladores rodam em sequência:



O terminal TCMM mostra a tradução para assembly; o TASM mostra a montagem e, o mais interessante, os avisos de recurso instanciado: cada bloco de hardware que o seu programa ligou no circuito.

Confira: a pasta `Hardware` ganhou `media_movel.v` (o processador), `media_movel_inst.mif` e `media_movel_data.mif` (as memórias). Em `Simulation` apareceu o testbench gerado.

Se a compilação falhar, vá à primeira mensagem de erro, não à última. A referência de linha é um link que abre o editor no ponto exato.

8.5 Passo 5: preparar o estímulo e simular

O testbench gerado alimenta cada porta de entrada com um arquivo de texto, um valor por linha, e grava cada porta de saída em outro.

1. Na visão *Pastas*, botão direito em `media_movel/Simulation`, *Novo arquivo*, nomeie `input_0.txt`.
2. Escreva um degrau: quatro zeros e depois valores 100.

Listing 8.2: Simulation/input_0.txt

```
0
0
0
0
100
100
100
100
100
100
100
100
```

3. Confirme na barra: simulador **Icarus Verilog**, visualizador **GTKWave**.
4. Clique em *Analisar Verilog*.

Se a simulação acabar antes do resultado

Quase sempre é o número de ciclos. Clique na engrenagem ao lado do botão C± e aumente o valor, que por padrão é 2000. O mesmo painel ajusta a frequência de clock e mostra o tempo simulado estimado.

8.6 Passo 6: ler a forma de onda

O visualizador abre com os sinais já agrupados, nomeados e coloridos, uma curadoria feita pela AURORA.

Figura8.3: O degrau entra e sai suavizado em quatro passos, o comportamento esperado de uma média móvel de quatro amostras.

Procure na janela:

- as variáveis `x[0]` a `x[3]` deslizando a cada amostra;
- `soma` subindo em degraus de 100 conforme o histórico se preenche;
- a porta de saída com o degrau suavizado;
- as trilhas de texto que mostram, a cada ciclo, a instrução assembly executada e a linha do seu C± correspondente. É o seu programa rodando dentro do hardware, em sincronia com o clock.

Os valores de saída também ficam em `Simulation/output_0.txt`, prontos para uma planilha ou um script.

8.7 Passo 7: ver o circuito por dentro

1. Botão direito em `Hardware/media_movel.v`, *Definir como Top Level*.
2. Clique em *Abrir PRISM*.

O PRISM desenha o circuito como um diagrama navegável: clique em um módulo para entrar nele.

Dica

O experimento que mais ensina: troque `soma >> 2` por `soma / 4`, recompile e clique em *Recompile* no PRISM. Um divisor inteiro aparece no diagrama, e o TASM anuncia o bloco novo. Desfaça e ele some. É a relação entre construção de linguagem e custo de hardware, tornada visível.

8.8 O que você aprendeu

- Um processador é uma pasta com `Software`, `Hardware` e `Simulation`, criada pelo Hub.
- As diretivas no topo do `.cmm` definem o hardware; o corpo define o comportamento.
- Compilar produz Verilog, imagens de memória e um testbench, por três compiladores encadeados.
- A simulação lê `input_N.txt` e grava `output_N.txt`.
- A onda mostra as suas variáveis pelo nome e a linha do seu código a cada ciclo.
- Escolhas de linguagem têm custo em hardware, e o PRISM mostra esse custo.

8.9 Exercícios

1. Mude o filtro para média de 8 amostras. O que precisa mudar além do vetor e do deslocamento?
2. Acrescente uma segunda porta de saída (`#NUIOOU 2`) publicando a amostra crua em paralelo com a média, e compare as duas curvas na onda.
3. Troque `soma >> 2` por `soma / 4`, recompile e meça a diferença: no relatório do TASM e no diagrama do PRISM.
4. Sature a saída com `pset()` para o filtro nunca publicar valor negativo, e teste com um degrau que desce.

Siga para a linguagem: *A linguagem C± essencial*. Ou, se preferir ver os detalhes da cadeia de compilação, *Compilação e artefatos*.

Capítulo 9

A linguagem C± essencial

C± é um dialeto de C reduzido ao que faz sentido virar hardware. Se você conhece C, aprende C± em uma tarde, e a leitura mais útil é a das ausências: elas existem porque cada construção da linguagem precisa mapear em circuito de tamanho conhecido em tempo de compilação.

Este capítulo cobre o essencial para a graduação. Complexos, notação de Dirac e os detalhes do ponto flutuante estão nos *estudos avançados*.

9.1 Tipos

Tipo	O que é
`int`	inteiro com sinal, largura definida por `#NUBITS`
`float`	ponto flutuante customizado, formato definido por `#NBMANT` e `#NBEXPO`
`comp`	número complexo (dois floats); assunto de *Números complexos*
`void`	apenas como retorno de função

Não existem `char`, `double`, `unsigned`, `bool`, `struct`, ponteiros nem strings. O identificador `i` é reservado para a parte imaginária dos complexos; não o use como variável.

9.2 Variáveis e vetores

```
int a, b;
float ganho = 0.5;
int v[128];                     // vetor
float M[4][4];                  // matriz (ate 2 dimensoes)
int tabela[64] "valores.txt";   // inicializado por arquivo, um valor por linha
```

Toda memória é estática: sem `malloc`, sem ponteiro, cada variável tem endereço fixo. É isso que faz cada variável aparecer pelo nome na forma de onda. A inicialização com chaves `{1, 2, 3}` não existe; vetores inicializam por arquivo.

9.3 Operadores

```
aritmeticos     + - * / %          (% so entre inteiros)
relacionais     < > <= >= == !=
logicos         && || !
bits            & | ^ ~ << >> >>>  (>>> preserva o sinal)
pos-incremento  a++  v[i]++
```

Não existem `--`, operadores compostos (`+=`, `-=` e família), operador ternário `?:`, cast explícito nem `sizeof`. Conversões entre `int` e `float` acontecem sozinhas, com aviso do compilador.

> **Dica**
>
> Deslocamentos substituem multiplicações e divisões por potências de dois: `x >> 2` no lugar de `x / 4`. A diferença não é estilo: `/` instancia um divisor no circuito, `>>` quase nada. O PRISM mostra isso ao vivo.

9.4 Controle de fluxo

`if/else`, `while`, `do while`, `for`, `switch/case/default` (com fall-through como em C), `break`, `continue` e `return`. Sem `goto`.

O `for` aceita as formas simples: `for (i = 0; i < 8; i++)`, com declaração no início ou cláusulas vazias. Não aceita múltiplas cláusulas separadas por vírgula.

9.5 Funções

```
float media(float a, float b)
{
    return (a + b) * 0.5;
}
```

- `main()` é obrigatória e é onde o programa vive.
- Até 16 parâmetros, passados por valor. Vetores não podem ser parâmetros.
- Sem recursão: as variáveis locais têm endereço fixo, então uma função não pode chamar a si mesma.
- Sem protótipos: defina a função antes de usar, em um único arquivo `.cmm`.

9.6 Entrada e saída

O processador conversa com o mundo por portas numeradas, definidas por `#NUIOIN` e `#NUIOOU`:

```
int  a = in(0);      // le um inteiro da porta 0
float g = fin(1);    // le da porta 1 convertendo para float
out(0, a + 1);       // escreve na porta de saida 0
fout(0, g);          // escreve mantendo o formato float
```

O número da porta precisa ser um literal, não uma variável. Na simulação, cada porta de entrada lê de `Simulation/input_N.txt` e cada saída grava em `Simulation/output_N.txt`.

9.7 Diretivas

O bloco no topo do arquivo configura o processador. As nove que o Hub escreve:

Diretiva	Configura
`#PRNAME`	nome do processador e do módulo Verilog
`#NUBITS`	largura da palavra
`#NBMANT`, `#NBEXPO`	formato do ponto flutuante (mantissa e expoente)
`#NDSTAC`, `#SDEPTH`	profundidade das pilhas de dados e de chamadas
`#NUIOIN`, `#NUIOOU`	número de portas de entrada e de saída
`#NUGAIN`	constante da função `norm()`, potência de dois

A regra estrutural: `#NUBITS` deve ser igual a `#NBMANT + #NBEXPO + 1`. A tabela completa, com as diretivas avançadas, está em *Diretivas do C±*.

Também existe `#define` para constantes simples (`#define TAM 8`), sem macros com argumentos, sem `#include` e sem compilação condicional.

9.8 Biblioteca

Funções que custam pouco, mapeadas quase uma a uma em instruções:

Função	Faz
`abs(x)`	valor absoluto
`pset(x)`	zera se negativo
`sign(x, y)`	devolve `y` com o sinal de `x`
`norm(x)`	divide pelo `#NUGAIN`, sem instanciar divisor

E as matemáticas, que custam mais porque viram rotinas de várias instruções: `sqrt`, `sin`, `cos`, `tan`, `atan`, `exp`, `log`, `pow`, hiperbólicas e arredondamentos. Use quando precisar; cada uma anexada ao programa aparece no relatório do TASM. A referência completa, com assinaturas e restrições, está em *Biblioteca do C±*.

9.9 O que não existe, em resumo

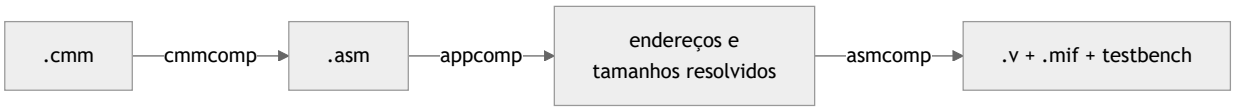
Ponteiros, structs, strings, recursão, alocação dinâmica, `#include`, múltiplos arquivos-fonte, vetores com mais de duas dimensões, vetores como parâmetro de função. Se o seu algoritmo parece exigir algum desses, a pergunta certa costuma ser: como um circuito faria isso com memória fixa? A resposta cabe em C±.

Capítulo 10

Compilação e artefatos

O botão *Compilar C±* esconde uma cadeia de três compiladores. Conhecê-la ajuda a ler os terminais e a saber onde procurar cada arquivo gerado.

10.1 A cadeia



cmmcomp
: O compilador da linguagem: traduz o C± em assembly do processador SAPHO, resolve variáveis e expressões, aplica otimizações e anexa as rotinas de biblioteca usadas. Escreve no terminal **TCMM**.

appcomp
: Uma pré-passagem sobre o assembly: conta instruções e dados e resolve os endereços de rótulos, informações de que o montador precisa antes de emitir a primeira palavra.

asmcomp
: O montador e gerador de hardware: produz as imagens de memória, o módulo Verilog do processador e o testbench. Escreve no terminal **TASM**, incluindo os avisos de recurso instanciado e a estimativa de ocupação do conjunto de instruções.

Os três falam o idioma escolhido nas configurações da AURORA, português ou inglês.

10.2 O que cada compilação gera

Para um processador chamado `media_movel`:

Arquivo	Onde	O que é
`media_movel.asm`	`Software/`	o assembly gerado, legível; vale abrir e comparar com o seu C±
`media_movel.v`	`Hardware/`	o processador em Verilog, pronto para simulação e síntese
`media_movel_inst.mif`	`Hardware/`	a imagem da memória de programa, uma instrução binária por linha
`media_movel_data.mif`	`Hardware/`	a imagem da memória de dados, com variáveis e constantes
`media_movel_tb.v`	`Simulation/`	o testbench gerado, com clock, reset e a leitura e escrita das portas

O `.v` e os dois `.mif` são exatamente o que se leva ao Quartus ou ao Vivado na hora de gravar o FPGA, como descreve *Levar ao FPGA*.

> **Nota**
>
> O testbench gerado só é copiado para `Simulation/` se você ainda não tiver um testbench próprio para o processador. O seu nunca é sobrescrito.

> **⚠ Aviso**
>
> Um arquivo presente em `Hardware/` não prova que a última compilação passou: pode ser sobra de uma tentativa anterior. A prova é o terminal, com as duas etapas terminando sem erro.

10.3 Um programa, um processador sob medida

A parte mais importante de toda a cadeia acontece em silêncio no asmcomp: o módulo Verilog gerado liga apenas os blocos de hardware que o seu programa usa. Cada instrução que o assembly contém vira um parâmetro ligado na instância do processador; cada instrução ausente deixa o bloco correspondente fora da síntese.

Por isso o TASM anuncia os recursos instanciados: aquela lista é literalmente o inventário do seu circuito. Um programa só de inteiros não carrega somador de ponto flutuante; um programa sem divisão não carrega divisor. O tamanho do processador acompanha o algoritmo, e é isso que torna o SAPHO um soft-core otimizado. O mecanismo por dentro está em *Os módulos HDL do SAPHO por dentro*.

10.4 Lendo erros de compilação

- As mensagens do cmmcomp apontam a linha do `.cmm`; a referência é um link clicável no terminal.
- Vá sempre à primeira mensagem: as seguintes costumam ser consequência dela.
- Constantes `float` aproximadas geram um aviso com o erro de representação. Não é defeito: é o formato de ponto flutuante escolhido, e *Ponto flutuante customizado* explica como dimensioná-lo.

As mensagens mais comuns, com causas e correções, estão em *Diagnóstico: sintomas e causas*.

Simulação do processador

Um processador SAPHO se simula pelos mesmos botões e chaves de *Formas de onda*, com três superpoderes específicos: as suas variáveis aparecem pelo nome, duas trilhas de texto acompanham a execução, e existe um modo de teste rápido só de entrada e saída.

11.1 Os três jeitos de rodar

Analisar Verilog
: O caminho completo: compila, simula o Testbench Top e abre a forma de onda.

Execução rápida
: Roda sem gravar onda. Para quando você só quer os arquivos de saída ou o veredito dos testes cocotb.

Teste do processador sintetizado
: O mais direto para processadores: pega o processador ativo, monta um executor nativo com o Verilator e roda só a interface de entrada e saída, lendo `input_N.txt` e escrevendo `output_N.txt`, com barra de progresso no terminal **THTEST**. Serve para validar o comportamento com muitos dados, rápido. O fim do programa é detectado por um marcador que a AURORA injeta no `.cmm` (a diretiva `#TOAQUI`, que cria o pino `cheguei`).

Nota

Com o Verilator, os sinais internos do processador (núcleo, pilhas, ULA) ficam fora da onda; as suas variáveis continuam visíveis. Para ver o processador por dentro, use o Icarus.

11.2 As variáveis pelo nome

Como toda memória do C± é estática, cada variável do seu programa vira um sinal nomeado na onda: `soma`, `x[0]`, `x[1]`. Vetores aparecem elemento a elemento quando a opção *Mostrar arrays* está ligada na engrenagem do processador. Floats aparecem decodificados como número real, e complexos no formato `a + bi`.

11.3 As trilhas Assembly e C±

Na onda de um processador, duas trilhas de texto acompanham o clock: o mnemônico da instrução em execução e a linha do `.cmm` correspondente. É o vínculo mais direto entre o código e o circuito: dá para seguir um `while` acontecendo, ciclo a ciclo.

11.4 Onde os resultados ficam

Arquivo	Conteúdo
`Simulation/input_N.txt`	estímulo da porta de entrada N, um inteiro por linha (você escreve)
`Simulation/output_N.txt`	o que o programa escreveu na porta N (a simulação gera)
o arquivo de onda	os sinais gravados, aberto pelo visualizador

Os `output_N.txt` são a ponte para a análise externa: planilha, script Python, comparação com um modelo de referência.

11.5 Exercícios

1. Rode o filtro do tutorial com o *Teste do processador sintetizado* alimentando mil amostras (gere o `input_0.txt` com um script) e confira o `output_0.txt` contra uma média móvel calculada em Python.
2. Na onda, siga uma iteração completa do `while` pela trilha C±: identifique em que ciclos acontecem as quatro cópias do histórico, a leitura da porta e a escrita.
3. Aumente o clock configurado de 100 para 200 MHz na engrenagem e observe o que muda (e o que não muda) na simulação.

PRISM: o circuito desenhado

O PRISM transforma o projeto em um diagrama de circuito navegável. Ele responde à pergunta que a forma de onda não responde: em que estrutura o meu código virou hardware?

12.1 Abrir

O botão *Abrir PRISM* exige um Top Level definido. Ele valida o projeto como o botão Verilog faria, sintetiza a estrutura com o Yosys e abre a janela do diagrama.

Figura12.1: O nível do topo. Processadores SAPHO aparecem com símbolo próprio; os demais módulos, como caixas com suas portas.

12.2 Navegar

- **Clique** em um módulo entra nele: o diagrama desce um nível na hierarquia, e a trilha no topo da janela mostra o caminho. *Back* sobe.
- **Duplo clique** em uma célula abre o código-fonte correspondente no editor da janela principal, na linha exata.
- Zoom com a roda do mouse, arraste para mover, *Fit* reenquadra.
- *Download* salva o diagrama atual como SVG, pronto para relatório ou slide.
- *Recompile* refaz tudo após uma mudança no código, sem fechar a janela.

Figura12.2: Dentro de um processador SAPHO: a ULA, as memórias e as pilhas têm desenho dedicado.

12.3 O uso didático

O PRISM fecha o ciclo pedagógico da plataforma: cada construção do C± tem um custo em blocos, e aqui os blocos aparecem. O experimento do tutorial (trocar um deslocamento por uma divisão e ver o divisor surgir no diagrama) vale para qualquer recurso: chame `sqrt()` e observe o que muda, acrescente uma variável `float` e compare.

12.4 Simulação interativa

O botão *Simular* abre o modo interativo, que anima o circuito com estímulos clicáveis. Ele funciona bem para designs pequenos e médios; acima de alguns milhares de células, o PRISM recusa o modo interativo e sugere o diagrama estático, que não tem limite prático. A simulação séria continua sendo a dos capítulos anteriores; o modo interativo é uma lupa para entender trechos.

12.5 Limitações

- O diagrama parte sempre do Top Level do projeto. Sem ele, o PRISM não abre.
- Módulos sem desenho dedicado aparecem como caixas genéricas, o que é cosmético, não funcional.
- O PRISM mostra estrutura, não tempo: quem mostra o comportamento ao longo dos ciclos é a forma de onda.

Parte IV

Os dois juntos

O processador dentro do seu Verilog

Até aqui os dois mundos andaram separados: Verilog escrito à mão na Parte II, processador gerado na Parte III. Este capítulo os junta: o processador vira um componente dentro de um circuito seu.

É o arranjo típico de um projeto real: o processador cuida do algoritmo, e a lógica em volta cuida do resto, como interfaces, sincronização e pré-processamento.

13.1 As portas do processador gerado

Abra `Hardware/media_movel.v` e olhe o cabeçalho do módulo. As portas que ele expõe:

Porta	Direção	Largura	Papel
`clk`	entrada	1	clock
`rst`	entrada	1	reset síncrono
`in`	entrada	`NUBITS`	dado de entrada
`out`	saída	`NUBITS`	dado de saída
`req_in`	saída	depende das portas	qual porta de entrada o programa está lendo agora
`out_en`	saída	depende das portas	qual porta de saída o programa está escrevendo agora

O aperto de mão é simples: quando o programa executa `in(k)`, o processador expõe `k` em `req_in` e espera o dado presente em `in`; quando executa `out(k, v)`, expõe `k` em `out_en` e `v` em `out` por um ciclo. Com uma porta só, `req_in` e `out_en` funcionam como sinais de "lendo agora" e "válido agora".

> **Dica**
>
> O melhor guia de fiação é o testbench gerado, `Simulation/media_movel_tb.v`: ele mostra exatamente como alimentar `in` em função de `req_in` e quando capturar `out`. Copiar a fiação dele para o seu top-level é o caminho seguro.

13.2 Um top-level de exemplo

Um gerador de estímulo em Verilog alimentando o filtro do tutorial: um contador produz uma rampa, e o processador devolve a média móvel dela.

Listing 13.1: top_filtro.v

```
module top_filtro (
    input  wire        clk,
    input  wire        rst,
    output wire [15:0] media,
    output wire        media_valida
);

    // gerador de estimulo: uma rampa que sobe de 8 em 8
    reg [15:0] rampa;
    wire       lendo;
```

(continua na próxima página)

(continuação da página anterior)

```
    always @(posedge clk) begin
        if (rst)
            rampa <= 16'd0;
        else if (lendo)
            rampa <= rampa + 16'd8;
    end

    // o processador gerado pela Parte III
    media_movel u_filtro (
        .clk    (clk),
        .rst    (rst),
        .in     (rampa),
        .out    (media),
        .req_in (lendo),
        .out_en (media_valida)
    );

endmodule
```

Passos na AURORA:

1. Crie `top_filtro.v` no projeto `MeuFiltro` (a classificação o marcará como sintetizável).
2. Botão direito, *Definir como Top Level*. O processador deixa de ser o topo e vira um componente.
3. Clique em *Sintetizar Verilog*. A elaboração resolve o módulo `media_movel` sozinha, porque o `.v` gerado está no projeto.
4. Escreva um testbench para o `top_filtro` (clock, reset, `$dumpvars`, `$finish`), marque como Testbench Top e clique em *Analisar Verilog*.

Na onda, você verá a rampa entrando, o aperto de mão pulsando e a média saindo suavizada: dois mundos no mesmo diagrama de tempo.

> **⚠ Aviso**
>
> O processador carrega as memórias por `$readmemb` com os arquivos `.mif`. Recompile o C± antes de simular o conjunto, para as imagens de memória estarem atualizadas com o programa.

13.3 Hierarquia e PRISM

Com o top-level seu, a visão *Hierarquia* mostra o processador como uma instância entre as outras, e o PRISM desenha o conjunto: sua lógica e o processador no mesmo diagrama, cada um navegável.

Vários processadores no mesmo projeto seguem a mesma receita: crie cada um no Hub, compile cada um, e instancie todos no seu top-level. Um estudo de caso completo com dois processadores sincronizados está em *Interrupção, marcador de fim e multiprocessador*.

13.4 Exercícios

1. Troque a rampa por um gerador pseudoaleatório (um LFSR de 16 bits) e observe o filtro suavizando o ruído.
2. Insira um decimador entre o gerador e o processador: só uma a cada duas amostras chega ao filtro.
3. Coloque dois processadores em cascata: a saída do filtro de média alimenta um segundo processador

que detecta quando o valor cruza um limiar e publica 0 ou 1.

Falta só um passo para o hardware de verdade: *Levar ao FPGA*.

Levar ao FPGA

A AURORA valida, simula e desenha. A síntese física, com mapeamento em células, posicionamento, roteamento e bitstream, acontece na ferramenta do fabricante: Quartus para Intel/Altera, Vivado para AMD/Xilinx. Este capítulo lista o que levar e os cuidados do caminho.

14.1 O que levar

Para um projeto com processador:

1. Os seus fontes Verilog (top-level e módulos próprios).
2. O processador gerado: `Hardware/<proc>.v`.
3. As imagens de memória: `Hardware/<proc>_inst.mif` e `<proc>_data.mif`.
4. A biblioteca de módulos do SAPHO, que o `<proc>.v` instancia. Ela acompanha a instalação da AURORA, na pasta `components/HDL` dentro da pasta de instalação: `processor.v`, `core.v`, `ula.v`, `instr_dec.v`, `addr_dec.v`.

Copie tudo para o projeto da ferramenta de síntese e adicione os arquivos ao projeto dela.

> **⚠ Aviso**
>
> As memórias são carregadas por `$readmemb` com o caminho dos `.mif`. Ao mover os arquivos, confira se o caminho gravado no `<proc>.v` continua resolvendo, ou ajuste-o para o layout do projeto de síntese. Um caminho quebrado sintetiza memória vazia, e o processador executa nada.

14.2 Passos típicos

1. Crie o projeto na ferramenta, aponte o dispositivo FPGA da sua placa.
2. Adicione os fontes e defina o seu top-level como entidade de topo.
3. Associe os pinos físicos: clock da placa em `clk`, botão em `rst`, e os sinais de dado nos periféricos que o seu top-level expõe.
4. Restrinja o clock (arquivo de constraints com o período) para a análise de tempo valer.
5. Sintetize, grave, teste.

14.3 Conferências que evitam sofrimento

- **Simule antes.** O comportamento visto na onda da AURORA é o contrato; a síntese não conserta algoritmo.
- **Reset**: o processador usa reset síncrono. Um botão da placa costuma precisar de sincronização antes de entrar no circuito.
- **Frequência**: a análise de tempo da ferramenta dirá a frequência máxima do conjunto. Se o relatório reprovar o seu clock, reduza a frequência da placa ou revise o caminho crítico.
- **Tamanho**: o relatório de utilização mostra o custo real de cada escolha do C±. É o fechamento do experimento do PRISM, agora em números de células.

14.4 O que a AURORA não faz

Não gera bitstream, não faz place and route, não escreve constraints. O Yosys embutido serve à visualização e à hierarquia, nunca à síntese física. Essa fronteira é de projeto: a plataforma termina onde a ferramenta do fabricante, que conhece o silício, começa.

Parte V

Estudos avançados

Ponto flutuante customizado

Estudos avançados

A Parte V pressupõe os tutoriais das Partes II e III. O conteúdo daqui em diante é voltado à pós-graduação.

No SAPHO, o formato de ponto flutuante é um parâmetro de projeto. Em vez de aceitar o IEEE 754 de 32 bits, você escolhe quantos bits de mantissa e de expoente o seu problema pede, e o hardware é gerado exatamente nesse tamanho. Menos bits, menos células, mais frequência; a troca é precisão.

15.1 O formato

Uma palavra `float` de `NUBITS` bits se divide em três campos:

```
[ sinal: 1 ][ expoente: NBEXPO ][ mantissa: NBMANT ]
```

com o valor dado por $(-1)^{sinal} \times mantissa \times 2^{expoente}$. As diferenças para o IEEE 754, todas deliberadas, todas baratas em hardware:

	SAPHO	IEEE 754
Mantissa	inteiro sem bit implícito	fração com 1 implícito
Expoente	complemento de dois, sem viés	com viés
Sinal	sinal e magnitude	sinal e magnitude
NaN, infinitos, subnormais	não existem	existem

A regra estrutural que o Hub impõe vem daqui: `NUBITS = NBMANT + NBEXPO + 1`, um bit para cada campo, sem sobra.

15.2 Faixa e precisão

- Maior magnitude: $(2^{NBMANT} - 1) \cdot 2^{2^{NBEXPO-1}-1}$
- Menor magnitude não nula: $2^{-2^{NBEXPO-1}}$
- Precisão relativa: cerca de $NBMANT \cdot \log_{10} 2$ dígitos decimais

Configurações de referência:

NUBITS	NBMANT	NBEXPO	Comparável a
16	10	5	meia precisão
23	16	6	o padrão do Hub
32	23	8	precisão simples IEEE em bits

15.3 Dimensionando para o seu problema

O procedimento honesto é experimental:

1. Estime a faixa dinâmica do sinal (o expoente cuida dela) e a precisão exigida (a mantissa cuida dela).
2. Gere o processador com a configuração candidata.
3. Compile: toda constante que não cabe exatamente no formato gera um aviso com o erro de representação. Leia esses avisos; eles são o primeiro veredito.
4. Simule e compare as saídas com um modelo de referência (um script Python com o mesmo algoritmo em dupla precisão, por exemplo).

15.4 Vendo o erro de arredondamento na onda

Na simulação com Icarus, o processador expõe dois sinais didáticos: `delta_float` e `delta_int`, o erro de arredondamento cometido em cada operação da ULA, calculado contra o valor exato. Adicione-os na configuração de ondas e observe o erro se acumulando ao longo de um laço: é a aula de análise numérica acontecendo no seu próprio circuito.

15.5 O custo em hardware

As operações de soma, multiplicação e divisão em ponto flutuante têm blocos dedicados na ULA, instanciados apenas se o programa as usa. As funções matemáticas (`sqrt`, `exp`, trigonométricas) não têm bloco: viram rotinas de software sobre essas operações básicas, e o custo delas é tempo de execução, não área. O relatório do TASM mostra as duas coisas.

Figura15.1: Para comparar formatos, gere dois processadores iguais mudando apenas mantissa e expoente, rode o mesmo estímulo e compare os `output_N.txt`.

Números complexos

O C± tem números complexos como tipo nativo: `comp`. Um complexo é um par de floats (parte real e imaginária) no formato de ponto flutuante do processador, e a aritmética sobre ele é gerada pelo compilador, sem biblioteca externa.

16.1 Declarar e operar

```
comp z;
comp w = 3.0 + 4.0i;
comp data[8];              // vetor de complexos

z = w * conj(w);           // 25 + 0i
float m = mod2(w);         // 25.0, o modulo ao quadrado
float f = fase(w);         // 0.927 rad
comp  u = complex(a, b);   // monta a + bi a partir de dois floats
```

O literal exige as duas partes: `3.0 + 4.0i` funciona, `4.0i` sozinho não. O identificador `i` é reservado pela linguagem exatamente por isso.

16.2 O que funciona e o que não

Operação	Com `comp`
`+ - * /` e o `-` unário	sim, com a álgebra completa
comparações (`<`, `==`, ...)	comparam o módulo, com aviso
`%`, bits (`&`, `|`, `^`, `~`), `deslocamentos`, `++`	não
`abs`	sim: devolve o módulo
`sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `atan`	sim, nas versões complexas
`pow`, hiperbólicas, arredondamentos, `pset`, `sign`, `norm`	não
`out()` direto de um `comp`	não: escreva `fout(p, real(z))` e `fout(p, imag(z))`
notação de Dirac	não; vetores Dirac são de `int` ou `float`

As funções de acesso: `real(z)`, `imag(z)`, `mod2(z)`, `fase(z)`, `complex(a, b)`, `conj(z)`.

16.3 Custo e representação

Um `comp` ocupa duas células de memória. Uma multiplicação complexa custa quatro multiplicações e duas somas reais; uma divisão, isso mais a divisão pelo módulo ao quadrado. O compilador gera essas sequências e o TASM relata os blocos instanciados, como sempre.

16.4 Complexos na forma de onda

Na simulação, cada variável `comp` aparece como um único sinal decodificado no formato `a + bi`, legível diretamente:

Figura16.1: A decodificação funciona no GTKWave e no Surfer, com o formato de ponto flutuante do processador aplicado automaticamente.

16.5 Exemplo rápido

O clássico $e^{i\pi} = -1$, direto no processador:

Listing 16.1: euler.cmm (trecho)

```
comp z = 0.0 + 3.14159i;
comp e = exp(z);
fout(0, real(e));    // -1.000 (com o erro do formato escolhido)
fout(0, imag(e));    //  0.000
```

Compare o resultado com diferentes configurações de mantissa: é um bom exercício para fechar este capítulo com o anterior. O uso pesado de complexos aparece no capítulo da *FFT*.

Capítulo 17

Notação de Dirac: álgebra linear em hardware

O recurso mais singular do C±: vetores e matrizes se operam com a notação bra-ket da física, e cada expressão vira código totalmente desenrolado, sem laços. Um produto interno de 4 elementos gera cerca de dez instruções em linha reta; as dimensões precisam ser constantes conhecidas na compilação, e é exatamente isso que permite o desenrolamento.

17.1 As formas suportadas

Com vetores `a`, `b`, `d`, matrizes `A`, `B`, `M`, `P` e escalares `c`, `g`, `e`:

Escrita	Calcula
`e = d - ▯w|x▯;`	produto interno $\langle w \mid x \rangle$ dentro de expressão
`a # |M|b▯;`	$a = Mb$ (matriz vezes vetor)
`a # c|b▯;`	$a = c\,b$ (escalar vezes vetor)
`a # |b▯ + c|d▯;`	$a = b + c\,d$ (soma ponderada)
`A # |a▯▯b|;`	$A = a\,b^T$ (produto externo)
`A # |B| - |a▯▯b|;`	$A = B - a\,b^T$
`A # c|B|;`	$A = c\,B$
`A # c|I|;`	$A = c\,I$ (identidade escalada)
`a # |0▯;`	zera o vetor
`a # c|in(p)▯;`	preenche `a` lendo a porta `p`, cada leitura vezes `c`
`a # c -> |a▯;`	registrador de deslocamento: desloca `a` e insere `c`
`out(p, c|a▯);`	escreve o vetor inteiro na porta, cada elemento vezes `c`

O `#` marca a atribuição vetorial. As declarações também aceitam inicialização por Dirac: `float Px[4] # |P|x▯;`.

Regras verificadas na compilação: dimensões compatíveis, tipos iguais dos dois lados (`int` com `int`, `float` com `float`), e nada de `comp`. O registrador de deslocamento exige o mesmo vetor dos dois lados.

17.2 Tutorial: um filtro RLS em 66 linhas

O filtro adaptativo RLS (*Recursive Least Squares*) é o caso de uso perfeito: seu miolo é puro produto de matriz e vetor, e em C com laços ele viraria uma página de índices. Em Dirac, cada linha do algoritmo matemático vira uma linha de código.

Crie um processador `proc_rls` (32 bits, mantissa 23, expoente 8, uma porta de entrada e uma de saída, pilha de dados 8) e use o programa:

Listing 17.1: proc_rls.cmm, o nucleo do algoritmo

```
#PRNAME proc_rls
#NUBITS 32
#NBMANT 23
#NBEXPO 8
#NDSTAC 8
```

(continua na próxima página)

(continuação da página anterior)

```
#SDEPTH 2
#NUIOIN 1
#NUIOOU 1

#define N 4          // ordem do filtro

void main()
{
    float x[N];      // entradas recentes
    float w[N];      // coeficientes do filtro
    float P[N][N];   // matriz de covariancia inversa
    float Px[N];
    float K[N];
    float d, y, e, g;

    P # 1000.0|I|;               // P = 1000 * identidade
    w # |0□;                     // coeficientes zerados

    while (1)
    {
        x # fin(0) -> |x□;       // desloca x e insere a amostra nova
        d = fin(0);              // sinal desejado

        y = □w|x□;               // saida do filtro
        e = d - y;               // erro

        Px # |P|x□;              // P x
        g = 1.0 / (1.0 + □x|Px□);
        K # g|Px□;               // ganho de Kalman

        w # |w□ + e|K□;          // atualiza os coeficientes
        P # |P| - |K□□Px|;       // atualiza a covariancia
        P # 1.0101|P|;           // fator de esquecimento

        fout(0, y);
    }
}
```

Leia o miolo do `while` ao lado das equações do RLS em qualquer livro de filtragem adaptativa: é uma transcrição. Esse é o argumento do recurso.

Para digitar `□` e `□`, use os caracteres U+27E8 e U+27E9 (o editor os aceita normalmente; vale montar atalhos de teclado do sistema ou copiar do próprio código de exemplo).

17.3 Quando usar

Dirac compensa quando o algoritmo é álgebra linear de dimensão pequena e fixa: filtros adaptativos, projeções, transformações de coordenadas, redes pequenas. Para dimensões grandes, o desenrolamento explode o número de instruções; o compilador avisa o tamanho gerado, e o relatório do TASM mostra o total. A alternativa nesses casos é escrever os laços em C± comum.

Figura17.1: O editor entende os brackets: realce e espaçamento próprios para a notação.

Capítulo 18

FFT em hardware

A FFT é o algoritmo que junta tudo o que a Parte V apresentou: números complexos, ponto flutuante dimensionado e um recurso que só o SAPHO tem na linguagem: o índice bit-reverso.

18.1 O problema do embaralhamento

A FFT radix-2 com decimação no tempo consome as amostras em ordem bit-reversa: para 8 pontos, a posição binária `001` vira `100`, então `x[1]` é lido como `x[4]`, e assim por diante. Em software comum, isso custa uma rotina de embaralhamento antes da transformada.

No SAPHO, o embaralhamento custa zero: a linguagem tem um modo de indexação que inverte os bits do índice no próprio hardware de endereçamento.

```
data[j]     // acesso normal
data(j)     // acesso com os bits de j invertidos
```

O parêntese no lugar do colchete final é a sintaxe. Quantos bits são invertidos é definido pela diretiva `#FFTSIZ`: para uma FFT de 8 pontos, `#FFTSIZ 3`.

18.2 O programa

Crie um processador `proc_fft` (32 bits, mantissa 23, expoente 8) e escreva a FFT de 8 pontos:

Listing 18.1: proc_fft.cmm, FFT radix-2 de 8 pontos

```
#PRNAME proc_fft
#NUBITS 32
#NBMANT 23
#NBEXPO 8
#NDSTAC 8
#SDEPTH 2
#NUIOIN 1
#NUIOOU 1
#FFTSIZ 3

#define N  8
#define NL 3            // log2(N)

void main()
{
    comp  data[N];
    comp  tw, t, u;
    int   i, j, k, par, salto;
    float ang;

    while (1)
    {
        // carrega as amostras ja em ordem bit-reversa:
```

(continua na próxima página)

(continuação da página anterior)

```
24          // data[i) grava na posicao com os bits de i invertidos
25          for (i = 0; i < N; i++)
26          {
27              data[i) = complex(fin(0), 0.0);
28          }
29
30          // as tres etapas de borboletas
31          salto = 1;
32          for (k = 0; k < NL; k++)
33          {
34              for (par = 0; par < N; par = par + 2 * salto)
35              {
36                  for (j = 0; j < salto; j++)
37                  {
38                      ang = -3.14159265 * j / salto;
39                      tw  = exp(complex(0.0, ang));
40
41                      u = data[par + j];
42                      t = tw * data[par + j + salto];
43
44                      data[par + j]         = u + t;
45                      data[par + j + salto] = u - t;
46                  }
47              }
48              salto = salto * 2;
49          }
50
51          // publica o espectro: modulo de cada raia
52          for (i = 0; i < N; i++)
53          {
54              fout(0, abs(data[i]));
55          }
56      }
57  }
```

Os pontos de atenção:

- A linha 27 é o truque inteiro: `data[i)` grava a amostra `i` já na posição embaralhada. Nenhuma rotina de reordenação, nenhum ciclo gasto.
- O twiddle `tw` sai de `exp` complexo, que o compilador implementa por rotina. Uma versão otimizada usaria uma tabela pré-calculada em arquivo (`comp tw[4] "twiddles.txt"` não existe para `comp`; use dois vetores `float` com as partes real e imaginária), bom exercício para a turma.
- `abs` de complexo devolve o módulo, direto para a saída.

18.3 Verificando

Alimente `input_0.txt` com 8 amostras de uma senoide que caiba em um período da janela e simule. No espectro de saída, duas raias simétricas devem se destacar. Compare com o `numpy.fft.fft` das mesmas amostras: os módulos devem bater dentro do erro do formato de ponto flutuante escolhido, o que fecha o laço com o capítulo de *ponto flutuante*.

Para tamanhos maiores, ajuste `N`, `NL` e `#FFTSIZ` juntos, e acompanhe no TASM o crescimento do programa.

Ver também

O índice bit-reverso nasceu de um projeto de processador dedicado à FFT: Projeto de um Processador Embarcado Otimizado para Aplicação com Transformada de Fourier (2016)[2].

[2] https://cdn.nipscern.com/publications/tcc-2016-leandro-silva.pdf

Capítulo 19

Os módulos HDL do SAPHO por dentro

Este capítulo abre a caixa: o que exatamente é o processador que o YANC gera, e por que ele merece o rótulo de soft-core otimizado. É leitura para quem vai estudar ou estender a arquitetura; o uso normal da plataforma não exige nada daqui.

19.1 A biblioteca de módulos

O processador é montado a partir de uma biblioteca fixa de módulos Verilog, instalada com a AURORA em `components/HDL`:

Arquivo	Contém
`processor.v`	as memórias de instrução e de dados, e o módulo `processor` que amarra tudo
`core.v`	o contador de programa, o prefetch, as duas pilhas, o controle de memória e de I/O, e o núcleo
`instr_dec.v`	o decodificador de instruções
`ula.v`	a unidade lógica e aritmética, um submódulo por operação
`addr_dec.v`	o decodificador one-hot das portas de I/O

O arquivo gerado por projeto, `Hardware/<proc>.v`, é um invólucro fino: instancia `processor` com os parâmetros do seu programa e liga as portas externas.

19.2 A arquitetura

Um processador de **acumulador único**, **Harvard**, com **três estágios de pipeline** efetivos e uma instrução por ciclo:

- Memórias separadas de programa e de dados, com larguras independentes. As duas são inicializadas pelos `.mif` via `$readmemb`.
- Não há banco de registradores: toda operação passa pelo acumulador. Expressões aninhadas usam a pilha de dados (profundidade `#NDSTAC`); chamadas de função usam a pilha de retorno (profundidade `#SDEPTH`).
- Os desvios resolvem no estágio de busca, sem penalidade. Não há forwarding nem stall porque a arquitetura não cria hazards: o caminho de dados e o de controle são casados por construção.
- Reset síncrono em tudo, uma escolha amigável a FPGA.
- O conjunto completo de instruções, com os 108 opcodes, está em *Conjunto de instruções do processador*.

19.3 A otimização que dá nome à plataforma

Aqui está o mecanismo central. Abra um `<proc>.v` gerado e olhe a instância do `processor`: além dos parâmetros numéricos, há uma lista de parâmetros com nomes de instruções, cada um ligado em 1:

```
processor #(
    .NUBITS(16), .NBMANT(10), .NBEXPO(5),
```

(continua na próxima página)

(continuação da página anterior)

```
    // ...
    .ADD(1), .S_ADD(1), .SHR(1), .INN(1), .OUT(1)
    // as demais instrucoes ficam em 0
) u_proc ( /* ... */ );
```

Cada instrução que o seu programa usa vira um parâmetro em 1. Dentro dos módulos da biblioteca, cada bloco está embrulhado em um `generate if` condicionado a esse parâmetro: **instrução não usada, bloco não sintetizado**. O somador de ponto flutuante, o divisor, o deslocador, cada um só existe no circuito se alguma linha do seu C± o exigir.

É por isso que o TASM imprime o relatório de recursos instanciados e a estimativa de uso do conjunto de instruções: aquilo é a lista dos `generate` que ligaram. E é por isso que trocar `/ 4` por `>> 2` no tutorial muda o diagrama do PRISM: o parâmetro `DIV` foi de 1 a 0, e o divisor evaporou.

O resultado prático: dois programas diferentes geram dois processadores de tamanhos diferentes sobre a mesma biblioteca. O processador é do tamanho do algoritmo, e o custo de área em FPGA acompanha o que o código pede, não o pior caso da arquitetura.

19.4 A visibilidade de simulação

Os módulos carregam um arnês de simulação, ativado apenas nos simuladores, que expõe as suas variáveis pelo nome, as trilhas de assembly e de linha C±, e os sinais `delta_float` e `delta_int` de erro de arredondamento. Nada disso existe na síntese física: o `<proc>.v` sintetizado é só o circuito.

19.5 Parâmetros derivados

Dois parâmetros são calculados pelo compilador, não escolhidos por você: a largura do campo de operando (que cresce com o tamanho do programa e dos dados) e os tamanhos das memórias (o número exato de instruções e de células de dados do programa). O processador não carrega memória sobrando.

Quem quiser ir além, o código-fonte da biblioteca e dos compiladores é aberto, no repositório do YANC na organização nipscernlab.

Capítulo 20

Interrupção, marcador de fim e multiprocessador

Três recursos que aparecem quando o processador deixa de ser um exercício e vira parte de um sistema.

20.1 Interrupção: `#PRACA`

A diretiva `#PRACA`, colocada como um comando dentro de `main()`, marca o endereço de atendimento de interrupção e cria o pino `itr` no processador. Enquanto `itr` estiver em 1, o contador de programa salta para o ponto marcado; quando volta a 0, o programa retoma de onde estava.

Listing 20.1: esqueleto com atendimento de interrupcao

```
void main()
{
    int amostra;

    while (1)
    {
        // laco principal
        amostra = in(0);
        out(0, amostra);

        #PRACA              // daqui em diante, o atendimento
        out(1, 1);          // sinaliza o evento na porta 1
    }
}
```

Só pode haver um `#PRACA` por programa. O uso típico: um periférico externo (o seu Verilog da Parte IV) levanta `itr` quando precisa de atenção, e o processador responde sem varrer a porta o tempo todo.

20.2 Marcador de fim: `#TOAQUI`

A diretiva `#TOAQUI` marca um endereço e cria o pino `cheguei`, que sobe quando o contador de programa passa por ali. É o mecanismo que o botão *Teste do processador sintetizado* usa para saber que o programa terminou: a AURORA injeta um `#TOAQUI` no final do `main()` automaticamente nesse fluxo.

Em um sistema seu, o pino serve de sincronização barata: "o processador chegou na fase tal". Também só pode haver um por programa.

20.3 Multiprocessador

Vários processadores SAPHO convivem no mesmo projeto e no mesmo circuito. A receita é a da Parte IV, repetida:

1. Crie cada processador no Hub, cada um com seu nome e seus parâmetros. Cada um vive na sua pasta, com seu `.cmm`.

2. Compile cada um com *Compilar C±*. Ao usar os botões de síntese e simulação, a AURORA recompila todos os processadores do projeto antes, sozinha.
3. Escreva um top-level Verilog que instancia todos e a lógica de interconexão: quem alimenta quem, quem sincroniza quem.
4. O testbench do conjunto é seu. O testbench gerado automaticamente dirige um processador isolado; um sistema com dois ou mais pede um testbench que conheça a interconexão.

O padrão de interconexão mais comum é o produtor e consumidor: a porta de saída de um processador alimenta a porta de entrada do outro, com o aperto de mão `out_en` e `req_in` fazendo a sincronização, possivelmente com uma FIFO entre eles quando os ritmos diferem.

Estudo de caso: DTW com dois processadores

O repositório do YANC traz um sistema completo de detecção de novidade em sinais de 60 Hz usando dois processadores: um `ZeroCross`, que detecta cruzamentos de zero e segmenta o sinal, e um `ProcDTW`, que calcula a distância DTW contra um padrão. Uma máquina de estados em Verilog puro coordena os dois, e o conjunto foi validado em FPGA real.

Os fontes estão em `Compilers/CMMComp/Tests/DTW` do repositório do YANC: os dois `.cmm`, o top-level, a máquina de estados e o testbench do sistema. É o melhor material de estudo para o padrão multiprocessador.

O padrão multiprocessador tem literatura própria do laboratório: a arquitetura multi-core para reconstrução online de energia (CBA, 2020)[3] e o simulador de pulsos do TileCal com SAPHO (SBAI, 2025)[4], este rodando na eletrônica do experimento ATLAS.

Na onda, cada processador aparece com seu grupo de sinais e suas próprias trilhas de assembly e de linha C±, então dá para acompanhar os dois programas executando em paralelo, ciclo a ciclo.

[3] https://cdn.nipscern.com/publications/cba-2020-arquitetura-multi-core.pdf
[4] https://cdn.nipscern.com/publications/sbai-2025-simulador-de-pulsos-do-tilecal.pdf

Parte VI

A AURORA no dia a dia

Tour pela interface

Uma janela, cinco regiões: a barra de ferramentas no topo, a árvore de arquivos à esquerda, o editor no centro, os terminais embaixo e a barra de status no rodapé. O painel da Aurora Intelligence, quando aberto, entra à direita.

Figura21.1: A janela com um projeto aberto. Esta captura serve de mapa para o restante do capítulo.

21.1 A barra de ferramentas

Os botões se agrupam por assunto, da esquerda para a direita. Um botão cinza está desabilitado porque falta um pré-requisito; o tooltip diz qual.

21.1.1 Projeto

Novo Projeto cria um projeto do zero; *Abrir Projeto* abre um arquivo `.spf` existente. O primeiro ícone recolhe e restaura a árvore lateral.

21.1.2 Processador

Hub de Processadores cria um processador novo. *Compilar C±* compila o arquivo `.cmm` em foco no editor; só habilita quando um `.cmm` está aberto e em foco. A engrenagem abre a configuração de simulação do processador ativo: frequência de clock, número de ciclos e exibição de arrays na onda.

CAPTURA PENDENTE
aurora-toolbar-projeto.png

CAPTURA PENDENTE
aurora-toolbar-processador.png

21.1.3 Síntese

Sintetizar Verilog valida o projeto: elabora todos os fontes a partir do Top Level e gera a hierarquia de módulos. *Abrir PRISM* faz o mesmo e abre o diagrama do circuito. Os dois exigem um Top Level definido.

21.1.4 Ondas

Da esquerda para a direita: as chaves de simulador (Icarus ou Verilator) e de visualizador (GTKWave ou Surfer); *Analisar Verilog*, que simula e abre a onda; *Execução rápida*, sem onda; *Cancelar*; a *Configuração de ondas*; o seletor de layout; e o *Teste do processador sintetizado*. As escolhas deste grupo estão explicadas em *Formas de onda* e *Simulação do processador*.

21.1.5 Ferramentas

Controle de Versão abre o painel git (o badge mostra quantos arquivos mudaram); *Bibliotecas Python* gerencia pacotes para testbenches cocotb; *Configurações* abre as preferências; *Assistente IA* abre a Aurora Intelligence (atalho `Ctrl+K`).

21.2 A árvore de arquivos

A árvore tem três visões, alternadas pelo botão no seu cabeçalho:

Arquivos
: Os fontes do projeto agrupados por processador, mais os importados. É a visão de trabalho: aqui se define o Top Level e o Testbench Top pelo menu de contexto.

Pastas
: O disco como ele é, com criar, renomear, mover, copiar e excluir, arrastar e soltar, e `Ctrl+Z` para desfazer.

Hierarquia
: A árvore de instâncias do design, disponível depois de uma sintetização Verilog. Clicar em um módulo abre o fonte na linha da definição.

O cabeçalho da árvore ainda traz: novo arquivo, atualizar, busca nos arquivos (`Ctrl+Shift+F`), abrir a pasta no explorador, backup do projeto (gera um zip datado em `Backup/`) e fechar o projeto.

CAPTURA PENDENTE
aurora-toolbar-ondas.png

CAPTURA PENDENTE
aurora-toolbar-direita.png

CAPTURA PENDENTE
aurora-arvore-arquivos.png

21.3 O editor

O editor é o Monaco, o mesmo do VS Code, com realce para C±, assembly SAPHO, Verilog, SystemVerilog e Python. Clique simples na árvore abre o arquivo em modo prévia (aba em itálico, substituída pela próxima prévia); duplo clique fixa a aba.

Três botões flutuam no painel em foco: dividir o editor (até três painéis), pré-visualizar Markdown e HTML, e formatar o arquivo (`Shift+Alt+F`). Para Verilog, dois analisadores trabalham enquanto você digita: um sintático, que também formata, e um semântico, que elabora o projeto inteiro e aponta sinais não declarados e incompatibilidades de porta.

Selecionar um trecho de código faz aparecer uma estrela: é o acesso rápido à Aurora Intelligence sobre aquela seleção (explicar, corrigir, melhorar, comentar).

21.4 Os terminais

Seis abas, uma por etapa: **TCMM** (compilação C±), **TASM** (montagem e geração do Verilog), **TVERI** (validação Verilog, hierarquia, PRISM), **TWAVE** (simulação e ondas), **THTEST** (teste do processador sintetizado) e **TCMD**, que é um PowerShell de verdade dentro da AURORA.

As mensagens chegam classificadas (erro, aviso, sucesso, dica) e podem ser filtradas pelos botões com contadores. Referências como `arquivo.v:15` são links: o clique abre o arquivo naquela linha. O botão de exportar salva o conteúdo de todos os terminais em um `.txt`.

O TCMD merece um capítulo próprio de dicas em *Controle de versão, Python e configurações*; por ora, dois comandos: `apython` chama o Python embarcado da AURORA, e `Use-Python aurora` faz `python` significar o embarcado naquela sessão.

21.5 A barra de status

Da esquerda para a direita: *Pronto* ou *Não Pronto* (clicável quando não há projeto: abre o diálogo de abrir); o processador ativo; o andamento da compilação; o Top Level; o Testbench Top; o motor de simulação escolhido; a posição do cursor com o controle de zoom; e a conta GitHub, que abre o painel de controle de versão.

21.6 Paleta de comandos e busca

`Ctrl+Shift+K` (ou `Ctrl+Shift+P`) abre a paleta de comandos, com tudo o que os botões fazem, pesquisável pelo nome. `Ctrl+Shift+F` abre a busca em todos os arquivos do projeto, com opções de maiúsculas, palavra inteira e expressão regular.

Com o mapa em mãos, o próximo capítulo explica como um projeto se organiza no disco: *Organização de um projeto*.

Capítulo 22

Organização de um projeto

Um projeto é uma pasta no disco governada por um arquivo `.spf` (*SAPHO Project File*). Tudo o que a AURORA sabe sobre o projeto está nele: quais arquivos participam, qual é o Top Level, qual é o Testbench Top e quais processadores existem.

22.1 A pasta

Um projeto que já tem um processador e alguns fontes Verilog fica assim:

```
MeuProjeto/
├── MeuProjeto.spf              o arquivo do projeto
├── contador.v                  fonte Verilog importado
├── tb_contador.v               testbench importado
├── media_movel/                um processador SAPHO
│   ├── Software/               o que você escreve
│   │   ├── media_movel.cmm
│   │   └── media_movel.asm     gerado na compilação
│   ├── Hardware/               o que o YANC gera
│   │   ├── media_movel.v
│   │   ├── media_movel_inst.mif
│   │   └── media_movel_data.mif
│   └── Simulation/             estímulos e resultados
│       ├── media_movel_tb.v
│       ├── input_0.txt
│       └── output_0.txt
├── testbench/                  estado de ondas por testbench
└── Backup/                     zips gerados pelo botão de backup
```

O projeto nasce só com o `.spf`. As pastas de processador aparecem quando você cria um processador no Hub; as demais, conforme o uso.

22.2 O arquivo .spf

É um JSON legível. Guarda o nome e o caminho base do projeto, a lista de processadores com suas configurações de simulação, as listas de fontes sintetizáveis e de testbenches, e os dois ponteiros centrais: `topLevelFile` e `testbenchFile`. Caminhos dentro do projeto são gravados relativos, então o projeto pode ser movido ou copiado de máquina para máquina sem quebrar.

> **Dica**
>
> O `.spf` abre no editor como JSON com realce. Ler o seu é uma boa forma de entender o que a AURORA registra. Editar à mão raramente é necessário; a interface cuida dele.

> **⚠ Aviso**
>
> Os parâmetros de arquitetura do processador (largura de bits, mantissa, portas) não ficam no `.spf`. Eles vivem nas diretivas no topo do arquivo `.cmm`, que é a fonte da verdade: editar uma diretiva muda o processador na próxima compilação. No `.spf` ficam apenas as preferências de simulação de cada processador (clock, número de ciclos).

22.3 Sintetizável ou testbench: quem decide é o conteúdo

Ao importar um `.v`, a AURORA o classifica sozinha lendo o conteúdo: sinais típicos de testbench (gravação de onda, `$finish`, módulo sem portas, blocos `initial`, atrasos `#`, nome terminando em `_tb`) somam pontos; passando do limiar, o arquivo é testbench, senão é sintetizável. Arquivos `.py` são sempre testbenches cocotb. A classificação se refaz a cada atualização da árvore, então um arquivo editado pode mudar de categoria sozinho.

O que a classificação não escolhe é o papel de raiz, e isso é seu:

Top Level
: O módulo raiz do circuito sintetizável. Define de onde a elaboração parte e o que o PRISM desenha. Marque pelo menu de contexto do arquivo na visão Arquivos: *Definir como Top Level*.

Testbench Top
: O arquivo que comanda a simulação, `.v` ou `.py`. Define o que roda quando você clica em *Analisar Verilog*. Marque por *Marcar como Testbench*.

Figura22.1: Os dois papéis são exclusivos: marcar um arquivo desmarca o anterior. A barra de status mostra os dois o tempo todo.

22.4 Importar e criar arquivos

Arraste arquivos `.v`, `.sv`, `.vh` ou `.py` de fora para a árvore, ou use o menu de contexto da área vazia: *Novo arquivo*, *Novo testbench cocotb (.py)*, *Novo .gitignore*. Na visão Pastas, o menu de contexto oferece o conjunto completo de operações de disco, com lixeira e desfazer.

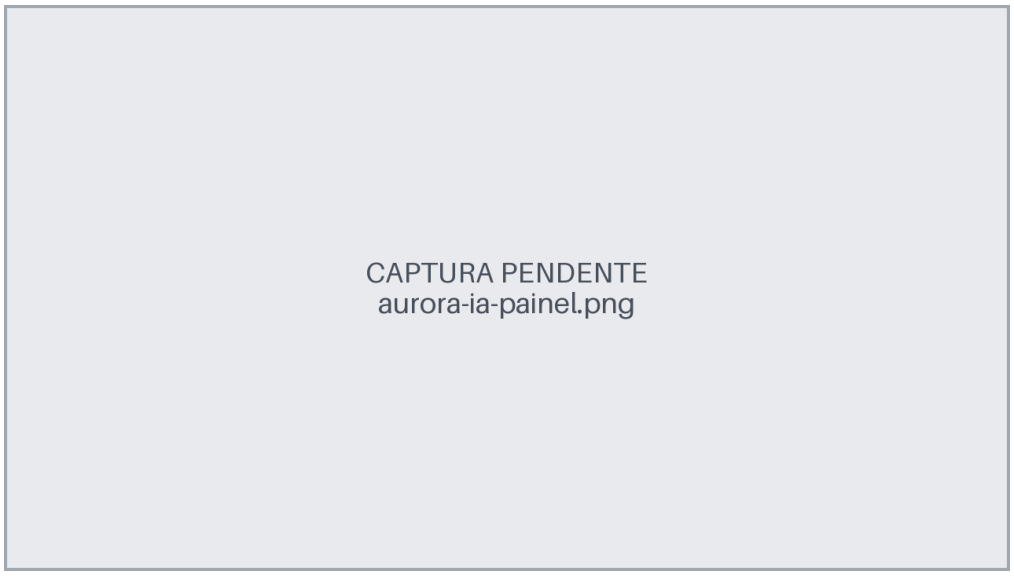
22.5 Backup

O botão de backup no cabeçalho da árvore gera `Backup/<projeto>_<data>.zip` com tudo, exceto os backups anteriores. É a forma rápida de congelar um estado antes de uma mudança grande. Para histórico de verdade, o painel de controle de versão está em *Controle de versão, Python e configurações*.

Pronto para trabalhar. A Parte II começa criando um projeto Verilog do zero: *Tutorial: um contador em Verilog*.

Aurora Intelligence

A assistente de IA integrada. Ela conhece o projeto aberto, a linguagem C± e o fluxo da plataforma, e pode agir sobre a IDE: abrir arquivos, editar código, compilar, ler terminais. Cada ação passa pelo seu controle. Abra por `Ctrl+K` ou pelo botão *Assistente IA*.



23.1 Configurar

Funciona no modelo BYOK: você traz a chave ou a assinatura. Em *Configurações*, aba *Assistente IA*, três caminhos:

- **Chave de API**: OpenAI, Anthropic, Google, DeepSeek ou Groq. Cole a chave no cartão do provedor, teste, salve. As chaves ficam cifradas no cofre do sistema.
- **Assinatura**: Claude Code ou Codex, com login pela conta, sem chave.
- **Local**: Ollama, rodando um modelo na própria máquina, sem nada sair do computador.

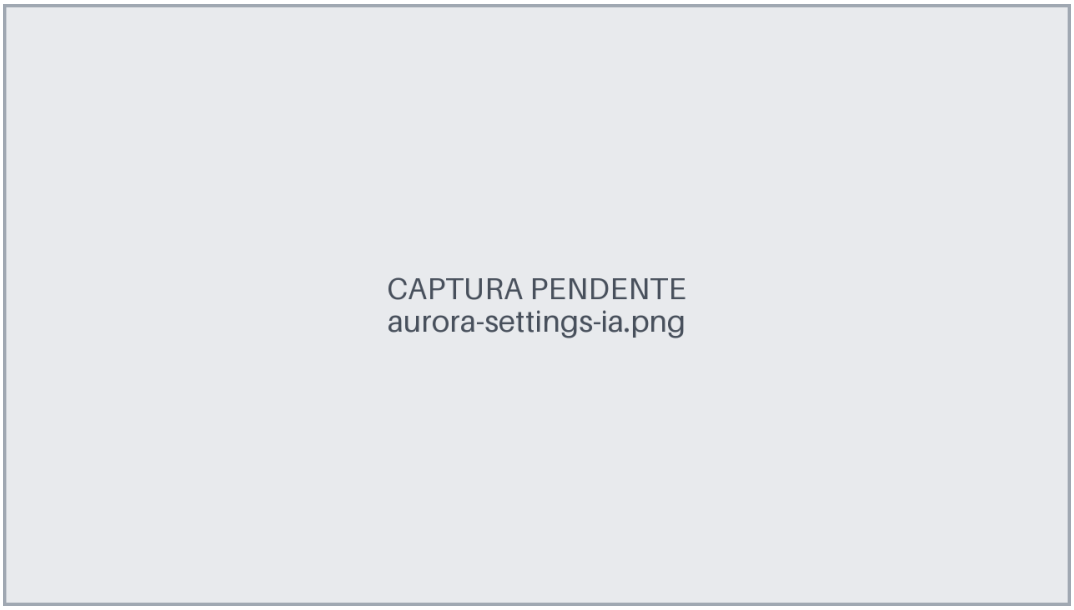
23.2 Usar

Pedidos que funcionam bem: "explique o que este `.cmm` faz e onde está o custo em hardware", "compile e me diga por que o TASM reclamou", "crie um testbench cocotb para o módulo contador". Selecionar código no editor faz aparecer uma estrela com atalhos: explicar, procurar defeitos, melhorar, comentar.

23.3 Permissões e limites

O seletor de permissões do painel controla a autonomia. O padrão, *Perguntar antes de alterar*, deixa as leituras livres e pede confirmação para qualquer escrita, com o cartão *Permitir* ou *Negar* mostrando exatamente o que será feito.

Limites fixos, em qualquer modo: sem shell arbitrário, sem trocar os binários da cadeia de compilação, links


CAPTURA PENDENTE
aurora-settings-ia.png


CAPTURA PENDENTE
aurora-ia-selecao.png


CAPTURA PENDENTE
aurora-ia-permissao.png

externos só abrem com o seu aval, e toda ação fica registrada em um log local.

Em sala: a assistente é ótima para explicar erros e revisar testbench, e má ideia para fazer o exercício pelo aluno. O modo padrão deixa o aluno no comando de cada mudança. Sem rede ou sem chave, a plataforma funciona normalmente; só o painel fica indisponível.

Controle de versão, Python e configurações

As três ferramentas de apoio que completam a IDE.

24.1 Controle de versão (Dagr)

O painel git da AURORA, aberto pelo botão *Controle de Versão* ou pelo item do GitHub na barra de status. O badge do botão mostra quantos arquivos mudaram.

O essencial:

- **Alterações**: marque os arquivos, escreva o resumo e clique em *Commit*. Cada arquivo tem o diff inline e o botão de descartar.
- **Histórico**: a lista de commits, com o diff de cada arquivo por demanda.
- **Sincronizar**: *Fetch*, *Pull*, *Push*. O menu do branch cria, troca e mescla branches, com stash automático quando a troca esbarra em alterações não commitadas.
- **GitHub**: o login usa o fluxo de dispositivo (um código digitado no navegador, sem senha na IDE) ou um token pessoal. Logado, dá para publicar o projeto direto (*Privado* ou *Público*) e clonar os seus repositórios.
- Um projeto sem repositório oferece *Inicializar* na hora; o menu da árvore cria um `.gitignore` adequado ao SAPHO.

A árvore de arquivos mostra o estado git de cada arquivo (modificado, novo, excluído) por letras e cores, nas duas visões.

24.2 Bibliotecas Python (PyLibs)

O painel *Bibliotecas Python* instala pacotes para os testbenches cocotb e para scripts de análise, no Python embarcado da AURORA, sem tocar no Python da sua máquina.

- O catálogo é curado: verificação de UVM e barramentos, leitura de ondas por script, gráficos, matemática, formatos de memória e comunicação serial. Cada item instala com um clique, com o download conferido por hash.
- Fora do catálogo, a seção *Outra biblioteca da PyPI* aceita qualquer pacote puro Python: a AURORA verifica a compatibilidade antes de baixar. Pacotes com código compilado (numpy, scipy, pandas) não rodam no Python embarcado; para esses, use o seu próprio Python pelo terminal.
- O isolamento é a regra: as bibliotecas do painel valem para o Python embarcado e para mais ninguém. No terminal TCMD, `apython script.py` roda com elas; `python` continua sendo o do sistema, com os seus pacotes do pip, a menos que você troque com `Use-Python aurora` (a troca vale só naquela sessão).
- Um vigia confere a integridade das bibliotecas de tempos em tempos; uma biblioteca quebrada (antivírus é a causa clássica) acende o badge do botão e ganha a ação *Reparar*.

24.3 Configurações

Configurações abre as preferências, em sete abas:

Aba	O que tem
Geral	tooltips da interface; confiar em links externos do chat da IA
Aparência	o fundo animado da tela de boas-vindas (desligado por padrão); trocar o ícone do aplicativo
Idioma	Português ou English, para a interface e para as mensagens dos compiladores
Terminal	modo verboso: mostra as linhas de comando completas de cada etapa
Atalhos de Teclado	regravar os atalhos principais; clique no atalho e pressione a combinação nova
Assistente IA	os cartões de provedores do capítulo anterior
Sobre	versão, situação do atualizador, equipe, manual online e offline, relatar problema, licenças

Detalhe que evita surpresa: fechar o painel sem *Salvar Alterações* descarta o que foi mudado.

24.4 Este manual, dentro da AURORA

A aba *Sobre* traz o *Manual (online)*, que abre este site, e o *Manual (offline)*, uma cópia completa instalada com o aplicativo, para a bancada sem internet. A cópia offline se atualiza sozinha quando uma versão nova do manual é publicada.

24.5 Relatar um problema

Também na aba *Sobre*: o botão monta um e-mail para a equipe já com o diagnóstico da instalação (versão, sistema, o que estava aberto), no seu webmail de preferência. Quanto mais contexto no relato, melhor a resposta.

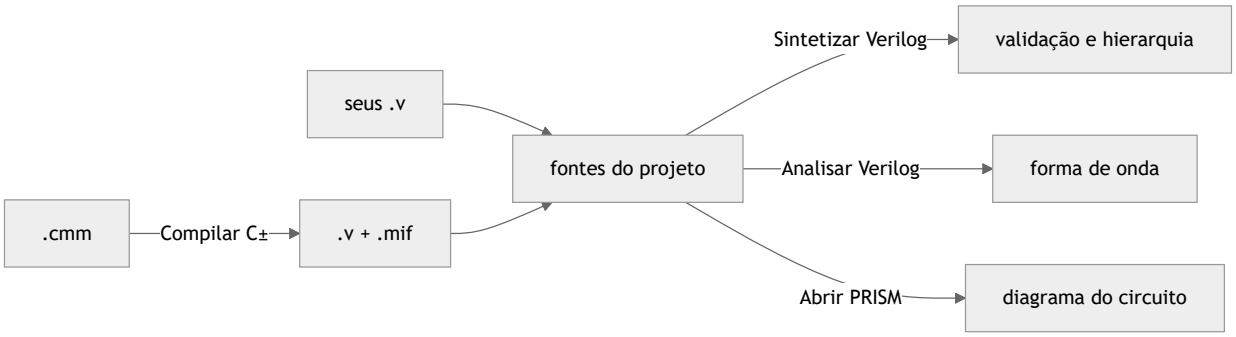
Parte VII

Referência

Folha de consulta rápida

Uma página para deixar na bancada. Imprima pelo navegador ou pelo PDF do manual.

25.1 O fluxo



25.2 Os botões e seus pré-requisitos

Botão	Faz	Exige
Compilar C±	C± até Verilog e memórias	um `.cmm` em foco
Sintetizar Verilog	valida a elaboração, gera a hierarquia	Top Level
Analisar Verilog	simula e abre a onda	Testbench Top
Execução rápida	simula sem onda	Testbench Top e Verilator (ou testbench .py)
Teste do processador sintetizado	roda só a E/S, com arquivos	processador ativo
Abrir PRISM	desenha o circuito	Top Level
Cancelar	interrompe o que roda	nada

25.3 Os papéis

Termo	Significa	Onde se define
Top Level	módulo raiz do circuito	botão direito no `.v`, visão Arquivos
Testbench Top	quem comanda a simulação	botão direito no `.v` ou `.py`
Processador ativo	o `.cmm` em foco no editor	abrir o arquivo

A barra de status mostra os três o tempo todo.

25.4 Arquivos de um processador

```
<proc>/Software/<proc>.cmm       voce escreve
<proc>/Hardware/<proc>.v + .mif  a compilacao gera (levar ao FPGA)
```

(continua na próxima página)

(continuação da página anterior)

```
<proc>/Simulation/input_N.txt    estimulo, um inteiro por linha
<proc>/Simulation/output_N.txt   resultado da simulacao
```

25.5 A regra de ouro do Hub

```
Total de bits = mantissa + expoente + 1
```

25.6 Atalhos que mais valem

`Ctrl+S` salvar	`Ctrl+Shift+F` buscar nos arquivos
`Ctrl+Shift+K` paleta de comandos	`Ctrl+K` assistente de IA
`Shift+Alt+F` formatar	`F12` ir à definição

25.7 Quando algo der errado

Leia a primeira mensagem de erro, não a última; clique no link de linha; e *Diagnóstico: sintomas e causas* tem os sintomas mais comuns.

Diretivas do C±

26.1 De configuração

Vivem no topo do arquivo, uma por linha, e definem o hardware gerado.

Diretiva	Padrão	Define
`#PRNAME nome`	obrigatória	nome do processador e do módulo Verilog
`#NUBITS n`	23	largura da palavra, em bits
`#NBMANT n`	16	bits de mantissa do ponto flutuante
`#NBEXPO n`	6	bits de expoente do ponto flutuante
`#NDSTAC n`	10	profundidade da pilha de dados (expressões)
`#SDEPTH n`	10	profundidade da pilha de chamadas
`#NUIOIN n`	1	portas de entrada
`#NUIOOU n`	1	portas de saída
`#NUGAIN n`	64	divisor fixo da função `norm()`; potência de dois
`#FFTSIZ n`	8	quantos bits do índice o acesso bit-reverso `x[i]` inverte

Regra estrutural, verificada na compilação: `NUBITS = NBMANT + NBEXPO + 1`.

Os padrões acima são os do compilador, aplicados quando a diretiva é omitida. O Hub de Processadores escreve as nove primeiras explicitamente no arquivo que gera.

26.2 De comportamento

Aparecem dentro de `main()`, como comandos, no máximo uma de cada por programa.

Diretiva	Cria o pino	Faz
`#PRACA`	`itr`	marca o ponto de atendimento de interrupção; com `itr` em 1, a execução salta para lá
`#TOAQUI`	`cheguei`	o pino sobe quando a execução passa pelo ponto marcado

Detalhes de uso em *Interrupção, marcador de fim e multiprocessador*.

26.3 Constantes

```
#define NOME valor
```

Substituição textual simples, sem argumentos. Não existem `#include`, `#ifdef` nem macros com parâmetros.

Biblioteca do C±

Todas as funções embutidas da linguagem. "T" indica que a função aceita `int` ou `float` e devolve o mesmo tipo.

27.1 Entrada e saída

Função	Devolve	Faz
`in(p)`	`int`	lê a porta de entrada `p`
`fin(p)`	`float`	lê a porta `p` convertendo para float
`out(p, e)`	nada	escreve `e` na porta de saída `p` (float é convertido para int, com aviso)
`fout(p, e)`	nada	escreve mantendo o formato float

O número da porta é sempre um literal inteiro, validado contra `#NUIOIN` e `#NUIOOU`.

27.2 Funções baratas

Mapeiam em uma ou poucas instruções.

Função	Devolve	Faz
`abs(x)`	T (ou `float` para `comp`)	valor absoluto; para complexo, o módulo
`pset(x)`	T	`x` se positivo, senão 0
`sign(x, y)`	T	`y` com o sinal de `x`
`norm(x)`	`int`	divide pelo `#NUGAIN` sem instanciar divisor; só `int`
`copy(x, id)`	nada	copia `x` na variável `id` sem checagem de tipo

27.3 Funções matemáticas

Implementadas como rotinas anexadas ao programa quando usadas; cada uma aparece no relatório do TASM.

Função	Aceita `comp`?
`sqrt(x)`	sim
`sin(x)`, `cos(x)`, `tan(x)`, `atan(x)`	sim
`exp(x)`, `log(x)`	sim
`pow(x, y)`	não
`sinh(x)`, `cosh(x)`, `tanh(x)`	não
`floor(x)`, `ceil(x)`, `round(x)`	não

Todas devolvem `float` (ou `comp` nas versões complexas). O método por trás delas está publicado em Implementação de Funções Não-Lineares em Processador Soft-Core (ENMC, 2025)[5].

[5] https://cdn.nipscern.com/publications/enemc-2025-implementacao-de-funcoes.pdf

27.4 Funções de números complexos

Função	Devolve	Faz
`real(z)`	`float`	parte real
`imag(z)`	`float`	parte imaginária
`mod2(z)`	`float`	módulo ao quadrado
`fase(z)`	`float`	fase em radianos, quatro quadrantes
`complex(a, b)`	`comp`	monta `a + bi` de dois floats
`conj(z)`	`comp`	conjugado; um real vira `x + 0i`

27.5 Operações vetoriais (notação de Dirac)

A tabela completa das formas `▯a|b▯`, `a # |M|b▯` e demais está em *Notação de Dirac: álgebra linear em hardware*. Restrições gerais: dimensões constantes, tipos iguais dos dois lados, sem `comp`.

Conjunto de instruções do processador

A referência do assembly SAPHO, útil para ler o `.asm` gerado e a trilha de instruções na forma de onda.

Convenções de nome: prefixo `P_` empilha o acumulador antes; `F_` opera em ponto flutuante; `S_` toma o segundo operando da pilha em vez da memória; sufixo `_M` aplica a operação na memória; sufixo `_V` é pseudo-instrução com deslocamento constante para acesso a vetores.

28.1 Memória e pilha

Instrução	Faz
`LOD`, `P_LOD`	carrega da memória no acumulador
`LDI`, `ILI`	carrega indireto (endereço = operando + índice); `ILI` inverte os bits do índice
`SET`, `SET_P`	grava o acumulador na memória
`STI`, `ISI`	grava indireto; `ISI` com índice bit-reverso
`PSH`, `POP`	empilha e desempilha o acumulador
`LDA`, `STA`	carrega e grava com endereço vindo do acumulador ou da pilha
`LEA`	carrega o endereço de uma variável como constante

28.2 Entrada e saída

Instrução	Faz
`INN`, `P_INN`	lê a porta de entrada indicada
`F_INN`, `PF_INN`	lê convertendo para float
`OUT`	escreve o acumulador na porta de saída

28.3 Controle de fluxo

Instrução	Faz
`JMP`	salto incondicional
`JIZ`	salta se o acumulador é zero (palavra inteira)
`CAL`, `RET`	chamada e retorno de sub-rotina
`NOP`	nada

28.4 Aritmética

Grupo	Instruções
soma e subtração	`ADD`, `S_ADD`, `F_ADD`, `SF_ADD`, `F_SU1`, `F_SU2`, `SF_SU1`, `SF_SU2`
multiplicação	`MLT`, `S_MLT`, `F_MLT`, `SF_MLT`
divisão e resto	`DIV`, `S_DIV`, `F_DIV`, `SF_DIV`, `MOD`, `S_MOD`
negação e absoluto	`NEG`, `F_NEG`, `ABS`, `F_ABS` e variantes `_M`
saturação e escala	`PST`, `F_PST`, `NRM` (divide pelo `#NUGAIN`) e variantes
conversão	`I2F`, `F2I` (trunca em direção a zero) e variantes
sinal	`SGN`, `S_SGN`, `F_SGN`, `SF_SGN`

28.5 Lógica, bits e comparações

Grupo	Instruções
bit a bit	`AND`, `ORR`, `XOR`, `INV` e variantes
lógicos	`LAN`, `LOR`, `LIN` e variantes
deslocamentos	`SHL`, `SHR` (lógico), `SRS` (aritmético) e variantes `S_`
comparações	`LES`, `GRE`, `EQU` nas formas inteira, float e de pilha; resultado 0 ou 1

28.6 Cirurgia de expoente

Usadas pelas rotinas matemáticas para operar direto no campo de expoente do float:

Instrução	Faz
`F_ROT`	aproxima a raiz quadrada pela potência de dois mais próxima (semente de Newton)
`F_SCL`, `SF_SCL`	multiplica por 2^k somando no expoente
`XPO`, `XPO_M`	extrai $\lfloor \log_2 \lvert x \rvert \rfloor$ como inteiro

São 108 opcodes ao todo. O mecanismo que sintetiza apenas os usados está descrito em *Os módulos HDL do SAPHO por dentro*.

Atalhos de teclado

29.1 Personalizáveis

Regraváveis em *Configurações*, aba *Atalhos de Teclado*.

Padrão	Ação
`Ctrl+N`	novo arquivo
`Ctrl+S`	salvar o arquivo atual
`Ctrl+Shift+S`	salvar todos
`Ctrl+W`	fechar a aba ativa
`Ctrl+Shift+T`	reabrir a última aba fechada
`Ctrl+Alt+S`	ligar e desligar a análise semântica de Verilog

29.2 Fixos

Atalho	Ação
`Ctrl+K`	abre e fecha a Aurora Intelligence
`Ctrl+Shift+K` ou `Ctrl+Shift+P`	paleta de comandos
`Ctrl+Shift+F`	buscar nos arquivos do projeto
`F2`	abre a pasta do projeto no explorador
`Ctrl` + `+` / `−` / `0`	zoom da janela
`Esc`	fecha o modal aberto

29.3 No editor

Atalho	Ação
`Ctrl+F`	localizar no arquivo
`Ctrl+H`	localizar e substituir
`Ctrl+G`	ir para a linha
`Shift+Alt+F`	formatar o documento
`F12`	ir para a definição
`Alt+F12`	espiar a definição
`Shift+F12`	listar referências

29.4 Na árvore, visão Pastas

Atalho	Ação
`F2`	renomear
`Delete`	mover para a lixeira (`Shift+Delete` apaga em definitivo)
`Ctrl+C` / `Ctrl+X` / `Ctrl+V`	copiar, recortar, colar
`Ctrl+Z` / `Ctrl+Y`	desfazer e refazer

Arrastar com `Ctrl` pressionado copia em vez de mover.

29.5 No painel da IA e nos terminais

Atalho	Ação
`Enter`	envia a mensagem (`Shift+Enter` quebra linha)
`Ctrl+C` no TCMD	copia a seleção; sem seleção, interrompe o comando
`Ctrl+V` no TCMD	cola

Diagnóstico: sintomas e causas

As falhas mais comuns de cada fluxo, com a causa e a correção. Regra geral que resolve metade dos casos: leia a primeira mensagem de erro, não a última, e clique no link de linha.

30.1 Botões desabilitados

Botão cinza	Falta
Compilar C±	um arquivo `.cmm` aberto e em foco no editor
Sintetizar Verilog, *Abrir PRISM*	um Top Level definido
Analisar Verilog, *Configuração de ondas*	um Testbench Top definido
Execução rápida	testbench definido, e Verilator selecionado ou testbench em Python
Teste do processador sintetizado	um processador ativo (o `.cmm` dele em foco)
engrenagem de configuração	idem

30.2 Projeto

O nome foi recusado
: Nomes de projeto e de processador aceitam letras, números, hífen e sublinhado. Sem espaços e sem acentos.

Aviso de arquivos ausentes na árvore
: O `.spf` lista arquivos que sumiram do disco (movidos ou apagados por fora). O botão do aviso remove as referências, com confirmação.

30.3 Compilação C±

O compilador aponta uma linha
: O link abre o editor no ponto. Erros em cascata quase sempre derivam do primeiro.

`Constante aproximada`
: A constante não cabe exata no formato de ponto flutuante escolhido. Informativo; se a precisão incomodar, aumente a mantissa (*Ponto flutuante customizado*).

O total de bits foi recusado
: `NUBITS` deve ser igual a `NBMANT + NBEXPO + 1`. Vale no Hub e nas diretivas.

Arquivo em `Hardware/` mas o terminal mostra erro
: O arquivo é de uma compilação anterior. Vale o terminal.

30.4 Validação Verilog

`Unknown module type: X`
: Um módulo instanciado não está em nenhum fonte do projeto. Se `X` é um processador, compile o C± dele antes; o `.v` gerado precisa existir.

Portas incompatíveis
: A instância diverge da definição. O analisador semântico do editor aponta antes do botão.

30.5 Simulação e ondas

A simulação termina antes do resultado
Poucos ciclos. Engrenagem do processador, aumente o número de clocks.

A onda abre sem os sinais internos do processador
Comportamento esperado com o Verilator. Troque para o Icarus para ver o processador por dentro.

Sinal selecionado não aparece
A seleção de ondas referencia um sinal que não existe mais (código mudou). Reabra a *Configuração de ondas* e salve de novo; a AURORA remove as referências mortas.

O testbench não encontra os dados
`input_N.txt` ausente ou com linha inválida. Um inteiro por linha, sem vazios.

cocotb reprova os testes mas a onda abre
Comportamento intencional: a onda é a ferramenta de investigação da falha. O veredito de cada teste está no TWAVE.

cocotb não acha o módulo alvo
Falta a linha `# aurora-toplevel: nome` no `.py`, ou o nome não confere.

30.6 PRISM

Recusa por tamanho no modo interativo
O diagrama estático não tem limite; a simulação interativa é para designs pequenos.

Diagrama desatualizado
Recompile na própria janela do PRISM.

30.7 Quando nada explica

- Ligue o modo verboso (*Configurações*, *Terminal*) e repita: os comandos completos de cada etapa aparecem no terminal.
- Exporte o log (botão na área de terminais) e anexe ao relato pelo *Relatar um problema* da aba *Sobre*.

Parte VIII

Apêndices

Capítulo 31

Glossário

AURORA
A IDE da plataforma SAPHO: editor, projetos, compilação, simulação e visualização em uma janela.

C±
A linguagem de descrição de algoritmos, dialeto reduzido de C, em arquivos `.cmm`.

cocotb
Framework de testbenches em Python: o simulador roda o circuito, o Python dirige os sinais.

Fonte sintetizável
Arquivo Verilog classificado como parte do circuito. A classificação é automática, pelo conteúdo.

Icarus Verilog
Simulador interpretado, o padrão para ondas curtas; enxerga todos os sinais.

MIF
Imagem de memória em texto (`_inst.mif` para o programa, `_data.mif` para os dados), carregada pelo processador na inicialização e usada também na síntese em FPGA.

Notação de Dirac
Sintaxe vetorial do C± (`□a|b□`, `a # |M|b□`) que desenrola álgebra linear em código de linha reta.

Processador ativo
O processador cujo `.cmm` está em foco no editor. É o alvo do botão C±, da engrenagem e do teste de hardware, e aparece na barra de status.

PRISM
O visualizador de circuito: desenha o RTL elaborado como diagrama navegável.

Processador SAPHO
O núcleo gerado pelo YANC: acumulador único, memórias separadas de programa e dados, e apenas os blocos que o programa usa.

SAPHO
A plataforma completa (e o nome do processador que ela gera).

spf
SAPHO Project File, o JSON que define um projeto: fontes, papéis, processadores.

Surfer
O visualizador de ondas alternativo ao GTKWave.

Testbench
O código que exercita o circuito na simulação, em Verilog ou Python.

Testbench Top
O testbench marcado como raiz da simulação; alvo do botão *Analisar Verilog*.

Top Level
O módulo raiz do circuito sintetizável; ponto de partida da elaboração e do PRISM.

Verilator
Simulador compilado, muito mais rápido; não expõe os sinais internos dos processadores na onda.

YANC
A suíte de compiladores que transforma C± em processador: tradutor, pré-montador e montador gerador de hardware.

Publicações

A plataforma nasceu e evolui dentro da pesquisa do NIPS-CERN. Esta é uma seleção do que vale ler junto com o manual; a lista completa do laboratório está em https://www.nipscern.com/publications.

32.1 O artigo da plataforma

- **SAPHO, Scalable-Architecture Processor for Hardware Optimization: An FPGA Customizable Implementation** (IEEE, 2026). O texto de referência: a arquitetura, a alocação de recursos dirigida pelo programa e os resultados em FPGA. PDF[6]

32.2 Sobre os recursos que este manual ensina

- **Implementação de Funções Não-Lineares em Processador Soft-Core Baseado em FPGA** (ENMC, 2025). Como `sqrt`, trigonométricas e exponenciais viram rotinas sobre a ULA, o assunto da biblioteca do C±. PDF[7]
- **Otimizações para Processador Soft-Core Embarcado em FPGA Visando Implementação de Métodos Iterativos** (TCC, 2023). As otimizações de geração de código que o TASM aplica. PDF[8]
- **Projeto de um Processador Embarcado Otimizado para Aplicação com Transformada de Fourier** (TCC, 2016). A origem do índice bit-reverso do capítulo de FFT. PDF[9]
- **Arquitetura Multi-Core de Processadores Reconfiguráveis para Reconstrução Online de Energia** (CBA, 2020). Vários processadores cooperando, o padrão da Parte de estudos avançados. PDF[10]

32.3 Aplicações reais

- **Simulador de Pulsos do TileCal em Tempo Real Utilizando o Processador SAPHO e FPGA** (SBAI, 2025). O SAPHO em produção na eletrônica do calorímetro do experimento ATLAS, no CERN. PDF[11]
- **Implementação de Fractal em FPGA Usando o Processador Soft-Core SAPHO** (ENMC, 2025). Um caso visual e didático de uso. PDF[12]
- **Implementação Embarcada em FPGA de Métodos Visando a Reconstrução Online de Energia no Calorímetro** (dissertação, 2023). Deconvolução iterativa multicore com SAPHO. PDF[13]

32.4 No prelo

Os trabalhos sobre a AURORA (a IDE e sua cadeia de compilação e simulação), a Aurora Intelligence e a integração do cocotb foram aceitos para o ENMC 2026 e entram na página de publicações após o congresso, em outubro de 2026.

[6] https://cdn.nipscern.com/publications/ieee-2026-soft-core-processors.pdf
[7] https://cdn.nipscern.com/publications/enemc-2025-implementacao-de-funcoes.pdf
[8] https://cdn.nipscern.com/publications/tcc-2023-lucca-oliveira.pdf
[9] https://cdn.nipscern.com/publications/tcc-2016-leandro-silva.pdf
[10] https://cdn.nipscern.com/publications/cba-2020-arquitetura-multi-core.pdf
[11] https://cdn.nipscern.com/publications/sbai-2025-simulador-de-pulsos-do-tilecal.pdf
[12] https://cdn.nipscern.com/publications/enemc-2025-implementacao-de-fractal.pdf
[13] https://cdn.nipscern.com/publications/mest-2023-melissa-santos.pdf

Capítulo 33

Escopo, versão e método

33.1 O que foi documentado

Esta documentação descreve a aplicação desktop AURORA no estado observado no repositório local. O conteúdo é orientado ao uso da aplicação por:

- **Estudantes e projetistas**, que precisam criar projetos, processadores, testbenches e interpretar resultados.
- **Docentes e equipes de laboratório**, que precisam explicar o fluxo SAPHO e reproduzir ambientes.
- **Usuários avançados**, que precisam configurar simulação, formas de onda, controle de versão e Aurora Intelligence.

Item	Valor congelado
Produto	AURORA IDE / SAPHO
Versão do pacote	6.4.2
Commit	`f71b2f4551bb59f20a2ae33dcd715eea3bd78050`
Branch	`main`
Plataforma primária	Windows 10/11
Runtime	Electron 39.8.10 e Node.js 18+
Editor	Monaco Editor 0.52.2, fixado exatamente
Data da análise	9 de julho de 2026

33.2 Fontes e precedência

As informações foram consolidadas nesta ordem de confiança:

1. Comportamento implementado no código-fonte.
2. Testes unitários e E2E.
3. Arquivos de configuração, scripts de bootstrap, empacotamento e release.
4. Documentos `ARCHITECTURE.md`, `README.md`, `RELEASE.md`, `SECURITY.md` e documentação interna.
5. Wiki local de apoio em `C:\Users\Computador\Documents\Arthur\NIPSCERN\Backup_Last_Project\Project_Wiki`, especialmente a rebaseline e o inventário funcional de 9 de julho de 2026.
6. Páginas públicas oficiais do NIPS-CERN sobre AURORA, SAPHO e YANC.
7. Documentação oficial das tecnologias integradas.

Nota

A wiki local é usada como apoio de rastreabilidade, não como substituta do código. Quando a wiki e o código divergem, o comportamento implementado no commit documentado prevalece.

33.3 Convenções

- **C±** e **CMM** referem-se à linguagem de entrada usada pelo SAPHO e à extensão `.cmm`.
- **Processador** é o núcleo de hardware gerado a partir de um algoritmo C±.
- **Top Level** é o módulo Verilog raiz da síntese.
- **Testbench Top** é o arquivo de simulação selecionado; pode ser Verilog (`.v`) ou cocotb/Python (`.py`).
- Caminhos entre `<...>` são exemplos e devem ser substituídos pelo valor local.
- Opções marcadas como *legadas* existem no DOM ou em configurações antigas, mas não constituem um fluxo ativo confiável nesta versão.

33.4 Limites

Este manual não reproduz a especificação completa da linguagem C± nem os manuais de cada ferramenta externa. Ele explica como executar tarefas na AURORA, compreender os arquivos produzidos e verificar os resultados observáveis de cada fluxo.

Não foram alterados arquivos do código-fonte e nenhum commit foi criado durante a produção deste material.

33.5 Créditos

Manual escrito e mantido por Chrysthofer A. A. Afonso, com orientação de Luciano M. A. Filho, no NIPS-CERN (UFJF).