

SAPHO & AURORA

Manual de uso e referência técnica

Versão 7.1.0

NIPS-CERN

Núcleo de Instrumentação e Processamento de Sinais

UFJF e CERN

Agosto de 2026

Sumário

I Primeiros passos 1

1 O que é o SAPHO 2
- 1.1 De onde ela vem . . . 2
- 1.2 As peças . . . 2
- 1.3 As ferramentas que vêm juntas . . . 2
- 1.4 O caminho completo . . . 3
- 1.5 Como este manual está organizado . . . 3

2 Instalação 5
- 2.1 Baixar . . . 5
- 2.2 Instalar . . . 5
- 2.3 A cadeia de compilação . . . 6
- 2.4 Primeiro início . . . 6
- 2.5 Idioma . . . 9
- 2.6 Atualizações . . . 9

3 O mapa da janela 11

II Verilog primeiro 12

4 Tutorial: um contador em Verilog 13
- 4.1 Passo 1: criar o projeto . . . 13
- 4.2 Passo 2: escrever o módulo . . . 14
- 4.3 Passo 3: escrever o testbench . . . 15
- 4.4 Passo 4: definir os papéis . . . 15
- 4.5 Passo 5: validar . . . 16
- 4.6 Passo 6: simular e ler a onda . . . 16
- 4.7 O que você aprendeu . . . 17
- 4.8 Um segundo padrão: máquina de estados . . . 17
- 4.9 Exercícios . . . 18
- 4.10 Para onde ir . . . 18

5 Formas de onda 19
- 5.1 Icarus ou Verilator . . . 19
- 5.2 GTKWave ou Surfer . . . 19
- 5.3 Escolher o que gravar . . . 19
- 5.4 Layouts salvos . . . 20
- 5.5 Exercícios . . . 20

6 O fluxo Verilog em detalhe 22
- 6.1 O que “Sintetizar Verilog” executa . . . 22
- 6.2 O papel de cada fonte . . . 22
- 6.3 Diagnósticos enquanto você digita . . . 22
- 6.4 Múltiplos arquivos e módulos . . . 23
- 6.5 Quando a validação falha . . . 23

7 Testbenches: Verilog e cocotb 24
- 7.1 Anatomia de um testbench Verilog . . . 24
- 7.2 Testbench em Python: cocotb . . . 24

7.3 Verilog ou cocotb? . . . 26

III C± e o processador 27

8 Tutorial: um processador SAPHO 28
8.1 Passo 1: criar o projeto . . . 28
8.2 Passo 2: criar o processador . . . 28
8.3 Passo 3: escrever o algoritmo . . . 30
8.4 Passo 4: compilar . . . 31
8.5 Passo 5: preparar o estímulo e simular . . . 33
8.6 Passo 6: ler a forma de onda . . . 34
8.7 Passo 7: ver o circuito por dentro . . . 34
8.8 O que você aprendeu . . . 36
8.9 Exercícios . . . 36

9 C± em uma página 38
9.1 Tipos e variáveis . . . 38
9.2 Operadores, controle e funções . . . 38
9.3 Entrada e saída . . . 39
9.4 Links entre processadores . . . 39
9.5 Diretivas . . . 39
9.6 Biblioteca . . . 40
9.7 C++ . . . 40

10 Compilação e artefatos 41
10.1 A cadeia . . . 41
10.2 O que cada compilação gera . . . 41
10.3 Um programa, um processador sob medida . . . 42
10.4 Cada clique fica registrado . . . 42
10.5 Lendo erros de compilação . . . 43

11 Simulação do processador 44
11.1 Os três jeitos de rodar . . . 44
11.2 As variáveis pelo nome . . . 44
11.3 As trilhas Assembly e C± . . . 45
11.4 Onde os resultados ficam . . . 45
11.5 Exercícios . . . 45

12 PRISM: o visualizador de RTL 46
12.1 Abrir . . . 46
12.2 Navegar . . . 48
12.3 O uso didático . . . 48
12.4 Simular . . . 48
12.5 Limitações . . . 52

IV Os dois juntos 53

13 O processador dentro do seu Verilog 54
13.1 As portas do processador gerado . . . 54
13.2 Um top-level de exemplo . . . 54
13.3 Hierarquia e PRISM . . . 55
13.4 Exercícios . . . 55

14 Levar ao FPGA 57
14.1 O que levar . . . 57
14.2 Passos típicos . . . 57

14.3 Conferências que evitam sofrimento . . . 57
14.4 O que a AURORA não faz . . . 58

V Estudos avançados **59**

15 Ponto flutuante customizado **60**
15.1 O formato . . . 60
15.2 Faixa e precisão . . . 60
15.3 Dimensionando para o seu problema . . . 61
15.4 Vendo o erro de arredondamento na onda . . . 61
15.5 O custo em hardware . . . 61

16 Números complexos **63**
16.1 Declarar e operar . . . 63
16.2 O que funciona e o que não . . . 63
16.3 Custo e representação . . . 63
16.4 Complexos na forma de onda . . . 63
16.5 Exemplo rápido . . . 64

17 Notação de Dirac: álgebra linear em hardware **65**
17.1 Os símbolos, e como digitá-los . . . 65
17.2 As formas suportadas . . . 66
17.3 Tutorial: um filtro RLS em 66 linhas . . . 66
17.4 Quando usar . . . 67

18 FFT em hardware **69**
18.1 O problema do embaralhamento . . . 69
18.2 O programa . . . 69
18.3 Verificando . . . 70

19 Os módulos HDL do SAPHO por dentro **72**
19.1 A biblioteca de módulos . . . 72
19.2 A arquitetura . . . 72
19.3 A otimização que dá nome à plataforma . . . 73
19.4 A visibilidade de simulação . . . 73
19.5 Parâmetros derivados . . . 73

20 Interrupção, marcador de fim e multiprocessador **74**
20.1 Interrupção: `#PRACA` . . . 74
20.2 Marcador de fim: `#TOAQUI` . . . 74
20.3 Multiprocessador . . . 75

VI A AURORA no dia a dia **77**

21 Tour pela interface **78**
21.1 A barra de ferramentas . . . 78
21.2 A árvore de arquivos . . . 79
21.3 O editor . . . 80
21.4 Os terminais . . . 81
21.5 A barra de status . . . 82
21.6 Paleta de comandos e busca . . . 82

22 Organização de um projeto **84**
22.1 A pasta . . . 84
22.2 Projetos de versões antigas . . . 85
22.3 Projetos de exemplo . . . 85

22.4 O arquivo .spf . . . 86
22.5 Sintetizável ou testbench: quem decide é você . . . 87
22.6 Importar e criar arquivos . . . 88
22.7 Backup . . . 88

23 Aurora Intelligence **89**
23.1 Como ela age na IDE . . . 89
23.2 A simulação que ela consegue ler . . . 90
23.3 Tutorial guiado . . . 90
23.4 Configurar . . . 90
23.5 Usar . . . 92
23.6 Permissões e limites . . . 93

24 Controle de versão, Python, componentes e configurações **94**
24.1 Controle de versão (Git-D) . . . 94
24.2 Bibliotecas Python (PyLibs) . . . 95
24.3 Componentes . . . 97
24.4 Configurações . . . 98
24.5 Este manual, dentro da AURORA . . . 98
24.6 Relatar um problema . . . 101

VII Referência **103**

25 Folha de consulta rápida **104**
25.1 O fluxo . . . 104
25.2 Os botões e seus pré-requisitos . . . 104
25.3 Os papéis . . . 104
25.4 Arquivos de um processador . . . 105
25.5 A regra de ouro do Hub . . . 105
25.6 Atalhos que mais valem . . . 105
25.7 Quando algo der errado . . . 105

26 Atalhos de teclado **106**
26.1 Personalizáveis . . . 106
26.2 Fixos . . . 106
26.3 No Simular do PRISM . . . 106
26.4 No editor . . . 107
26.5 Na árvore, visão Pastas . . . 107
26.6 No painel da IA e nos terminais . . . 107

27 Diagnóstico: sintomas e causas **108**
27.1 Botões desabilitados . . . 108
27.2 Projeto . . . 108
27.3 Compilação C± . . . 108
27.4 Validação Verilog . . . 109
27.5 Simulação e ondas . . . 109
27.6 PRISM . . . 109
27.7 Quando nada explica . . . 110

VIII Apêndices **111**

28 Glossário **112**

29 Publicações **114**
29.1 O artigo da plataforma . . . 114
29.2 Sobre os recursos que este manual ensina . . . 114

29.3 Aplicações reais . . . 114
29.4 No prelo . . . 114

30 Links e onde encontrar a gente **115**
30.1 As duas casas . . . 115
30.2 Na rede . . . 115
30.3 A plataforma . . . 115
30.4 Leitura . . . 116
30.5 As ferramentas que vêm na caixa . . . 116
30.6 Falar com a gente . . . 117

31 Escopo, versão e método **118**
31.1 O que foi documentado . . . 118
31.2 Fontes e precedência . . . 118
31.3 Convenções . . . 119
31.4 Limites . . . 119
31.5 Créditos . . . 119

Parte I

Primeiros passos

Capítulo 1

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 De onde ela vem

O SAPHO nasceu no NIPS-CERN, o Núcleo de Instrumentação e Processamento de Sinais da UFJF, que mantém desde 2001 uma colaboração com o CERN, o laboratório europeu de física de partículas, em Genebra. Lá, no experimento ATLAS do LHC, o calorímetro de telhas (TileCal) mede a energia das partículas produzidas nas colisões, e a eletrônica que lê esses sinais precisa de processamento em tempo real, dentro do FPGA, com resposta em poucos ciclos de relógio.

Foi esse problema que deu forma à plataforma. Um processador SAPHO comanda hoje o simulador de pulsos usado nas bancadas de teste da eletrônica do TileCal, e a linguagem C± carrega as marcas dessa origem: aritmética de ponto flutuante com formato escolhido por você, números complexos como tipo nativo, notação de Dirac para produtos internos, índice com bits invertidos para FFT. São recursos de processamento de sinais e de física, não de programação de propósito geral.

A mesma plataforma é a base da disciplina de Dispositivos Lógicos Programáveis na UFJF, onde cada estudante projeta, compila e simula um processador próprio.

1.2 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. Tudo passa por ela: os compiladores e as ferramentas de apoio rodam por baixo, chamados pelos seus botões, e você não precisa de linha de comando para nada do fluxo.

O conjunto de compiladores que trabalha por baixo. O YANC traduz o programa C± no processador em Verilog, nas imagens de memória e no testbench. Você nunca o 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.

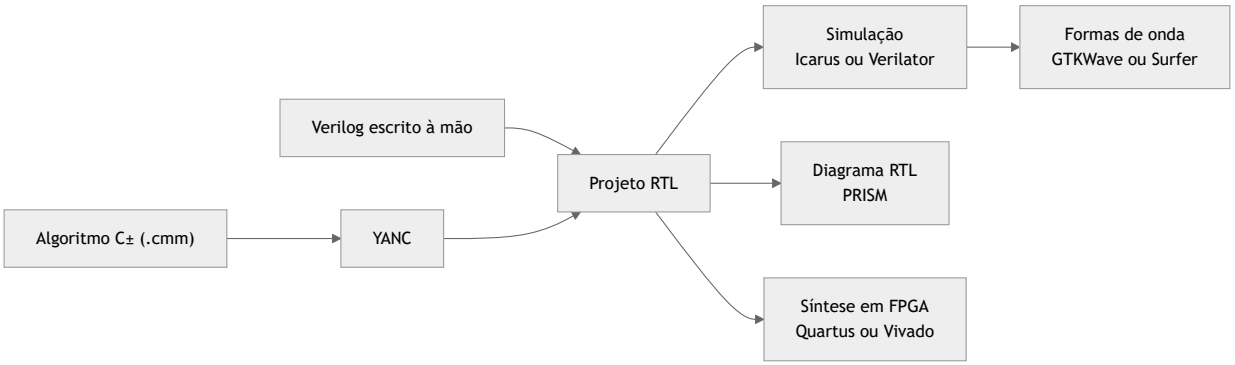
Um subconjunto de C, de médio nível, com adições voltadas a processamento de sinais e física. Arquivos com extensão `.cmm`. A versão essencial está no capítulo *C± em uma página*; os recursos de pós-graduação, nos *estudos avançados*.

1.3 As ferramentas que vêm juntas

Em volta desse núcleo trabalha um conjunto de ferramentas de código aberto, cada uma consagrada no que faz. Você não precisa instalar, configurar nem chamar nenhuma delas por fora: a AURORA baixa as versões certas pelo painel de Componentes (*Controle de versão, Python, componentes e configurações*) e as aciona pelos botões.

Ferramenta	Para que serve
Icarus Verilog	Simulador Verilog interpretado, o mais fiel ao padrão. É o motor padrão da AURORA: enxerga todos os sinais internos e gera o `.vcd` completo. Mais lento em simulações longas.
Verilator	Compila o Verilog para C++ e roda muito mais rápido, ao custo de só aceitar código sintetizável e de expor menos sinais. É a escolha para varreduras longas e para testes automatizados.
GTKWa	Visualizador de formas de onda clássico, o padrão de fato no mundo Verilog. Abre o `.vcd`, guarda layouts em `.gtkw` e é onde as trilhas de assembly e de linha C± aparecem.
Surfer	Visualizador de ondas moderno, escrito em Rust, com navegação mais fluida. A AURORA traz um fork nosso, integrado ao projeto, que abre como uma aba dentro do próprio editor.
Yosys	Ferramenta de síntese lógica. Aqui ela não gera bitstream: elabora o projeto, resolve a hierarquia e produz a estrutura que o PRISM desenha.
PRISM	Visualizador de RTL da casa. Transforma a saída do Yosys em um diagrama navegável do circuito, com um clique para descer na hierarquia e outro para voltar ao código-fonte. No modo Simular, o diagrama vira um circuito vivo, com relógio e entradas clicáveis.
cocotb	Biblioteca que permite escrever testbenches em Python em vez de Verilog, com `async/await`. Roda sobre o Icarus ou o Verilator.
Pylibs	O gerenciador de bibliotecas Python da AURORA. Um catálogo curado de bibliotecas para testbenches e análise — cocotb, verificação UVM, leitura de ondas por script, gráficos —, instaláveis com um clique no Python embarcado, sem mexer no Python do sistema.

1.4 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.

1.5 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

Instalação

O SAPHO roda em Windows 10 e 11, 64 bits. O instalador traz a AURORA e os compiladores do YANC, que são o SAPHO em si; o resto — simuladores, visualizadores de onda e o Python embarcado — são componentes que a própria AURORA baixa quando você for usar, sem instalar nada por fora. O primeiro download é guiado, como mostra a seção *A cadeia de compilação* abaixo.

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 140 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.

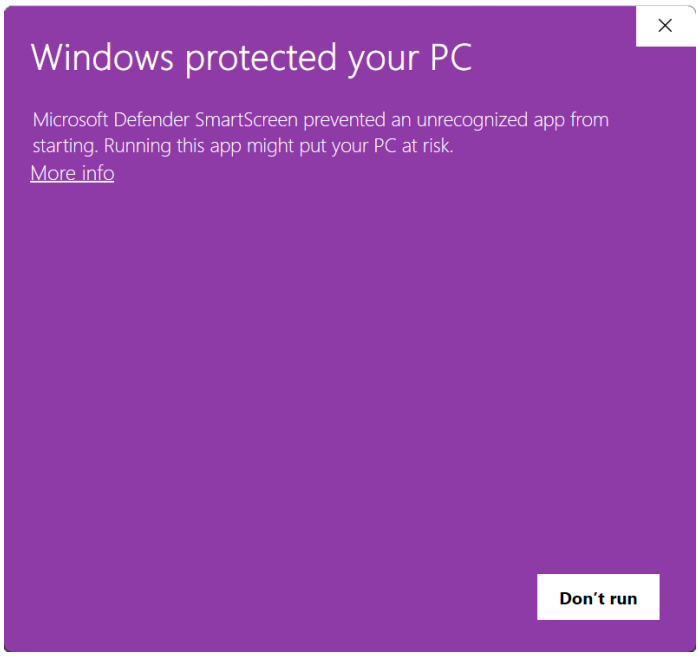


Figura2.1: O aviso do SmartScreen. *Mais informações* revela o botão *Executar assim mesmo*.

3. Leia e aceite a licença. Com o aceite, a instalação segue sozinha até o fim: instala para o usuário atual, sem pedir pasta nem senha de administrador.

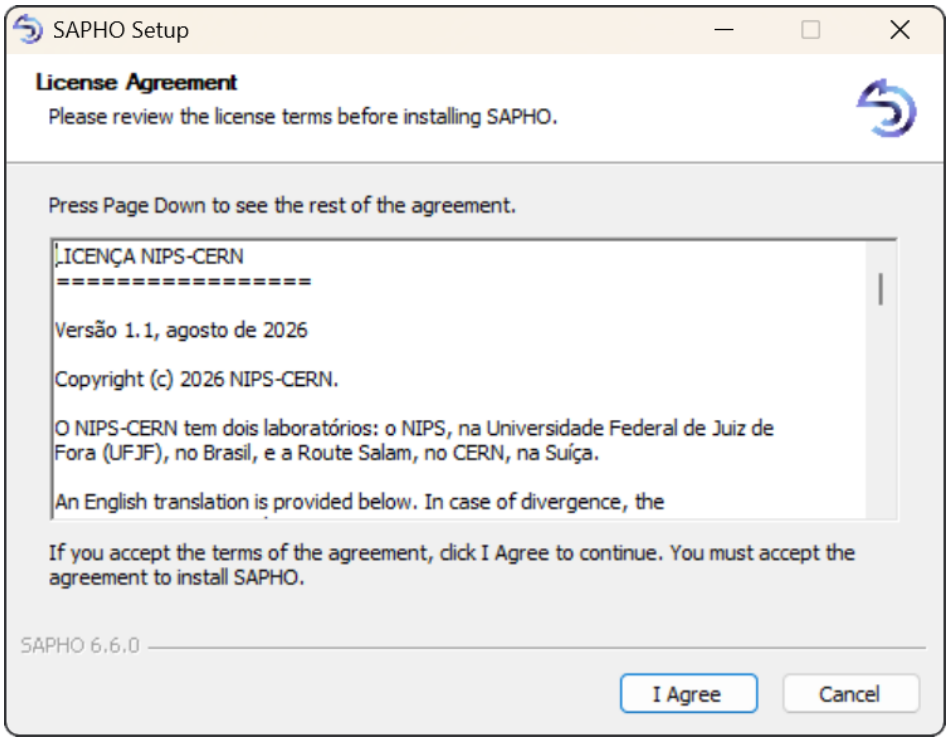


Figura2.2: A página de licença do instalador; o aceite é obrigatório. O padrão instala para o usuário atual, sem exigir privilégios de administrador.

> **⚠ Aviso**
>
> Não execute o instalador como administrador. Rodado com "Executar como administrador", o SAPHO se instala no perfil do administrador, e o aluno que fizer login depois não encontra nada; além disso, a atualização automática deixa de funcionar. Se a máquina do laboratório pede senha para instalar, a senha serve para liberar a política de execução, não para elevar o instalador.

O instalador também registra as extensões `.spf`, `.cmm` e `.v`: um duplo clique num `.spf` abre o projeto, e num `.cmm` ou `.v` abre o arquivo solto no editor.

2.3 A cadeia de compilação

Na primeira abertura, a AURORA avisa que a máquina ainda não compila: a cadeia de compilação — Icarus Verilog, Verilator, Yosys e o Python embarcado — não vem no instalador, para que ele caiba em 140 MB. Um diálogo oferece o download (cerca de 272 MB), e aceitar já abre a aba *Componentes* das Configurações com o download andando.

Até esse download terminar, os botões de compilar e simular não funcionam, e a engrenagem de *Configurações* na barra superior fica com um ponto de aviso aceso. Tudo o que se baixa, se conserta e se remove por ali está descrito em *Controle de versão, Python, componentes e configurações*.

2.4 Primeiro início

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

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.

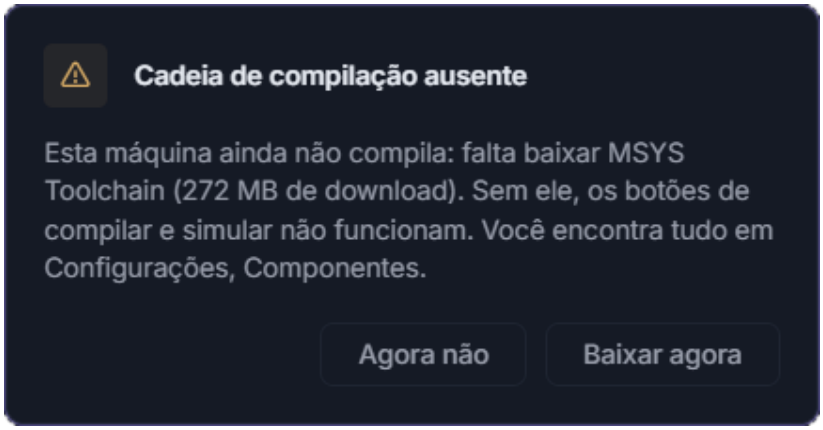


Figura2.3: O aviso do primeiro início. *Baixar agora* resolve; *Agora não* deixa para depois, e o aviso volta na próxima abertura.



Figura2.4: A tela de abertura enquanto os componentes são conferidos. O rodapé mostra o progresso e a versão.

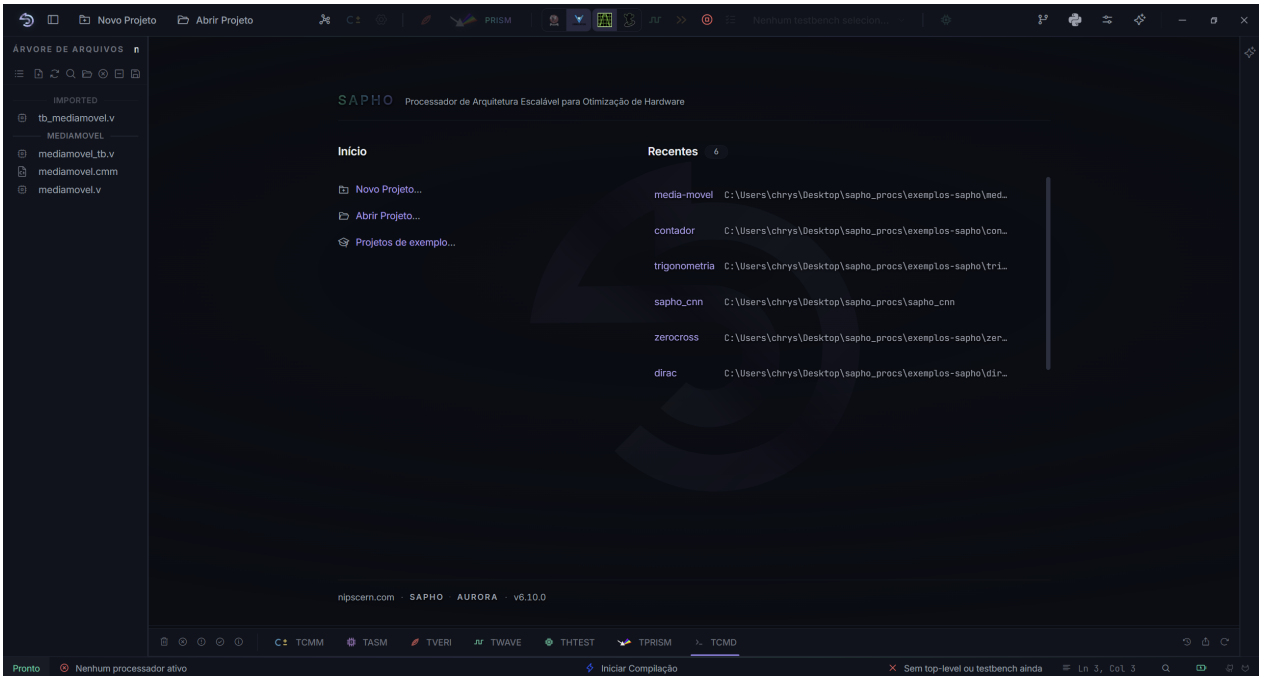


Figura2.5: A tela de boas-vindas. Os projetos abertos recentemente ficam à direita; os que sumiram do disco aparecem riscados, com uma lupa que os procura no computador e o botão de esquecê-los.

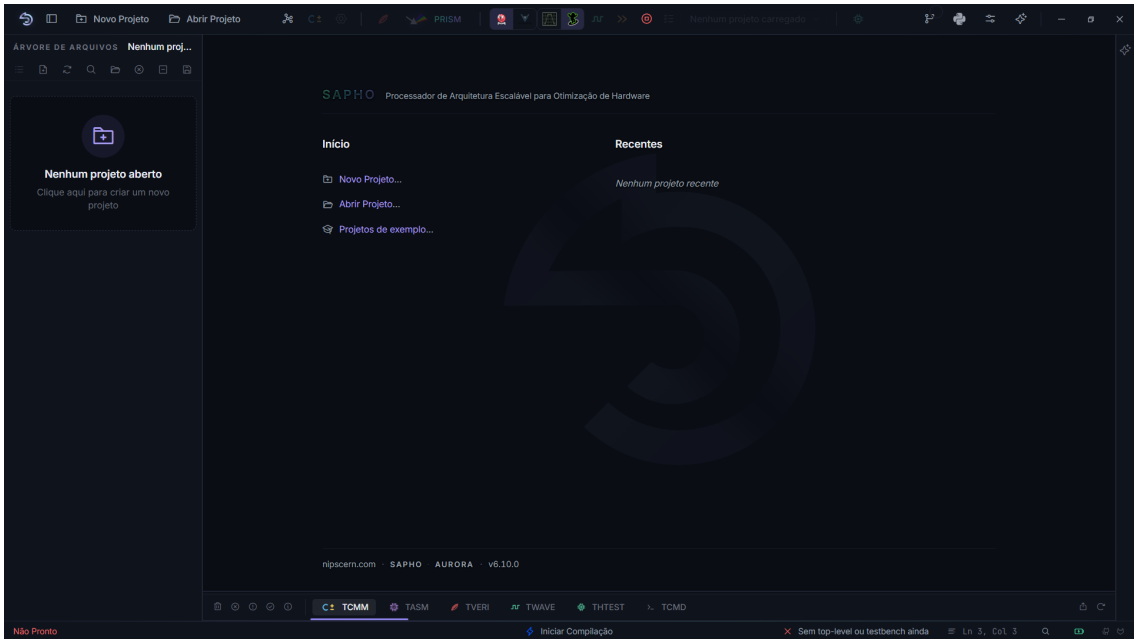


Figura2.6: A tela de boas-vindas. À esquerda, criar ou abrir projeto e o botão *Projetos de exemplo…*, que instala cinco projetos prontos; à direita, a lista de projetos recentes, vazia no primeiro uso.

- Em *Configurações* (ícone de controles deslizantes na barra superior), a aba *Sobre* mostra a versão instalada e a situação do atualizador.

2.5 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.

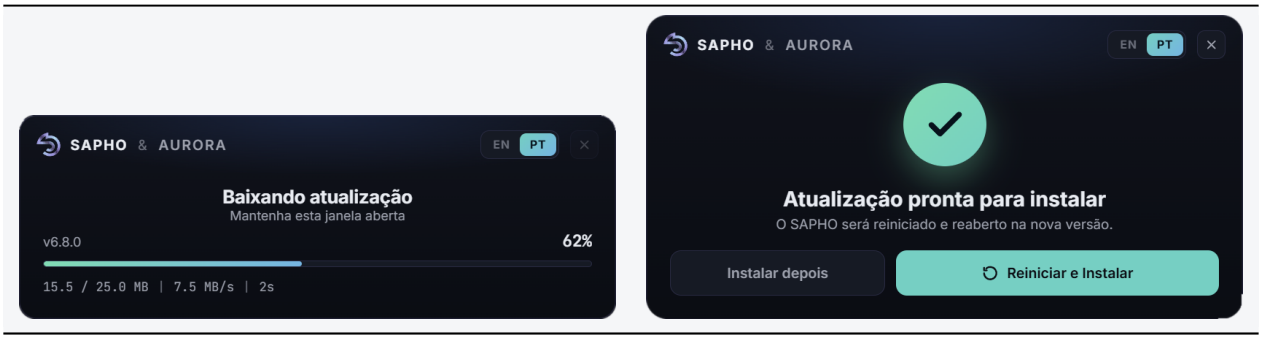
2.6 Atualizações

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.



Figura2.7: O aviso traz a versão atual, a nova, o tamanho do pacote e o changelog da release.

O download acontece em segundo plano, com progresso à vista, e ao terminar a AURORA pergunta se pode reiniciar. Quem responder *Instalar depois* não perde nada: a atualização baixada fica guardada e se instala sozinha na próxima vez que a AURORA abrir, antes de a janela aparecer.



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*.

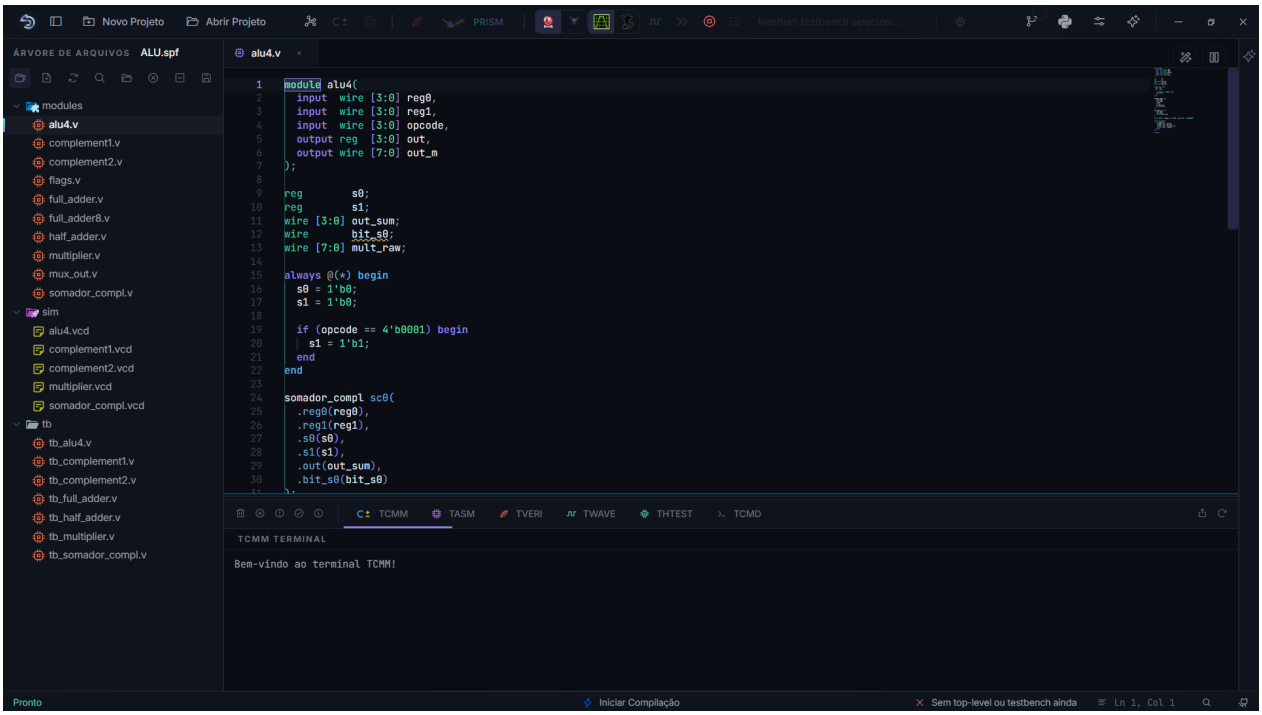


Figura3.1: A janela com um projeto aberto.

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.

Nota

Com pressa? O botão *Projetos de exemplo…* da tela de boas-vindas cria um contador equivalente já pronto. O tutorial vale mais construído à mão, mas o exemplo serve de gabarito para conferir o seu.

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*.

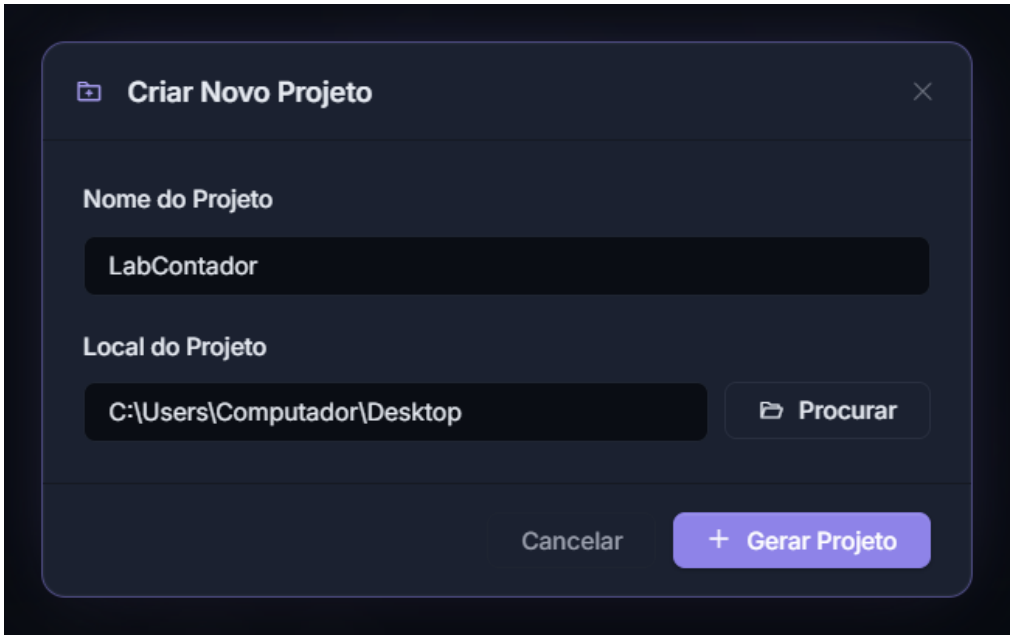


Figura4.1: O formulário pede só nome e local. O campo de local é preenchido pelo botão Procurar.

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:

Listagem 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`.

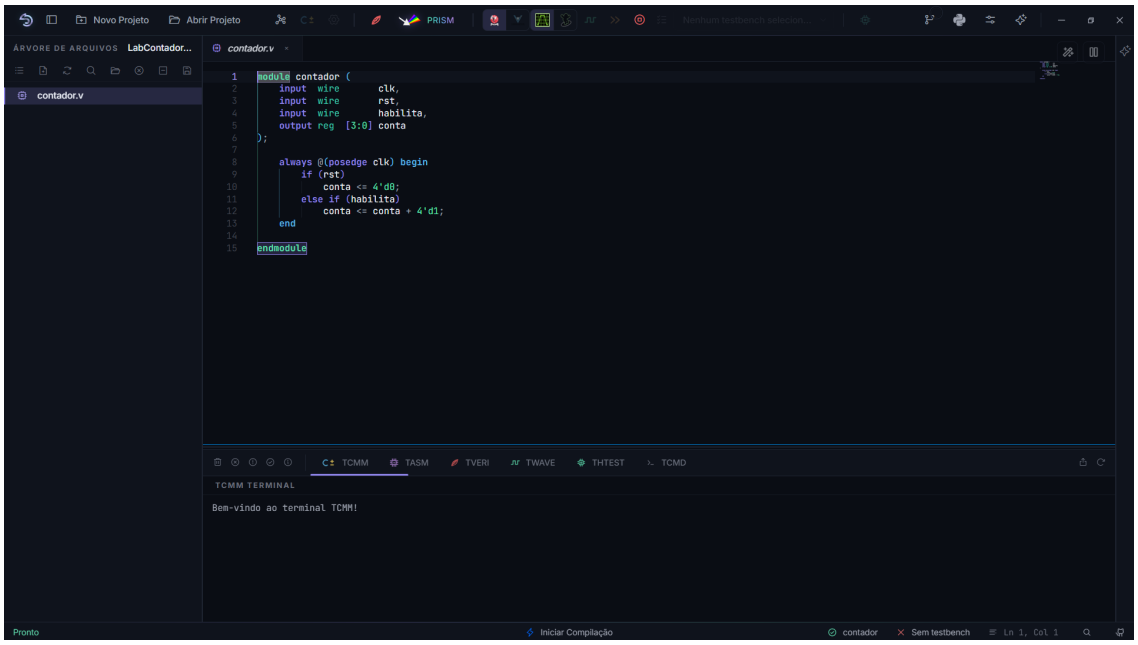


Figura4.2: Enquanto você digita, dois analisadores conferem o código: erros de sintaxe aparecem sublinhados na hora, e a análise semântica aponta sinais não declarados e portas incompatíveis.

Confira: na visão Arquivos, `contador.v` apareceu com um ponto de interrogação no ícone: ainda não tem papel. Botão direito nele, *Marcar como síntese*, e o ícone vira o de fonte sintetizável. Até ganhar um papel, o arquivo fica fora de toda compilação.

4.3 Passo 3: escrever o testbench

Crie um segundo arquivo, `tb_contador.v`:

Listagem 4.2: tb_contador.v

```
`timescale 1ns/1ps

module tb_contador;

    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 sem papel, com o ponto de interrogação. Botão direito, *Marcar como testbench*, e o ícone vira o de testbench.

4.4 Passo 4: definir os papéis

Um projeto pode ter dezenas de arquivos, e a AURORA precisa saber dois deles pelo nome: qual é o circuito e qual é o teste.

Top Level
: O módulo que está no alto da hierarquia, a raiz do seu projeto. É dele que a elaboração parte, e tudo o que ele instancia, direta ou indiretamente, faz parte do design. O que não for alcançável a partir do Top Level não participa: fica no projeto, mas fora do circuito. É também o módulo que você levaria à ferramenta do fabricante para gravar no FPGA, e o que o PRISM desenha quando abre. Aqui é o `contador.v`, porque o contador é o circuito.

Testbench Top
: O módulo que comanda a simulação, o que gera clock, reset e estímulos. Ele instancia o Top Level como um componente e o observa de fora. Não vai para o FPGA: existe só para exercitar o circuito na bancada.

Aqui é o `tb_contador.v`.

Marque cada um:

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

Os dois papéis são exclusivos: marcar um arquivo novo desmarca o anterior. Se um dia a compilação reclamar de módulo não encontrado, ou a onda vier vazia, o primeiro lugar a conferir é este par.

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.

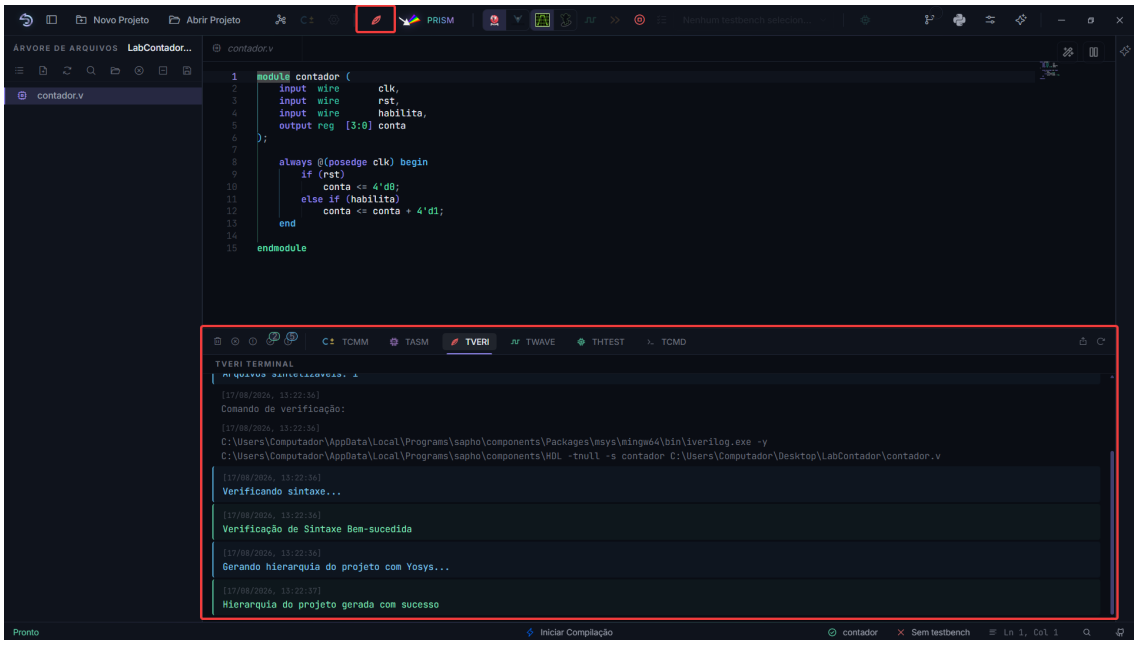


Figura4.3: O terminal TVERI relata a validação. Depois dela, o botão de visões da árvore ganha a opção Hierarquia.

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

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;

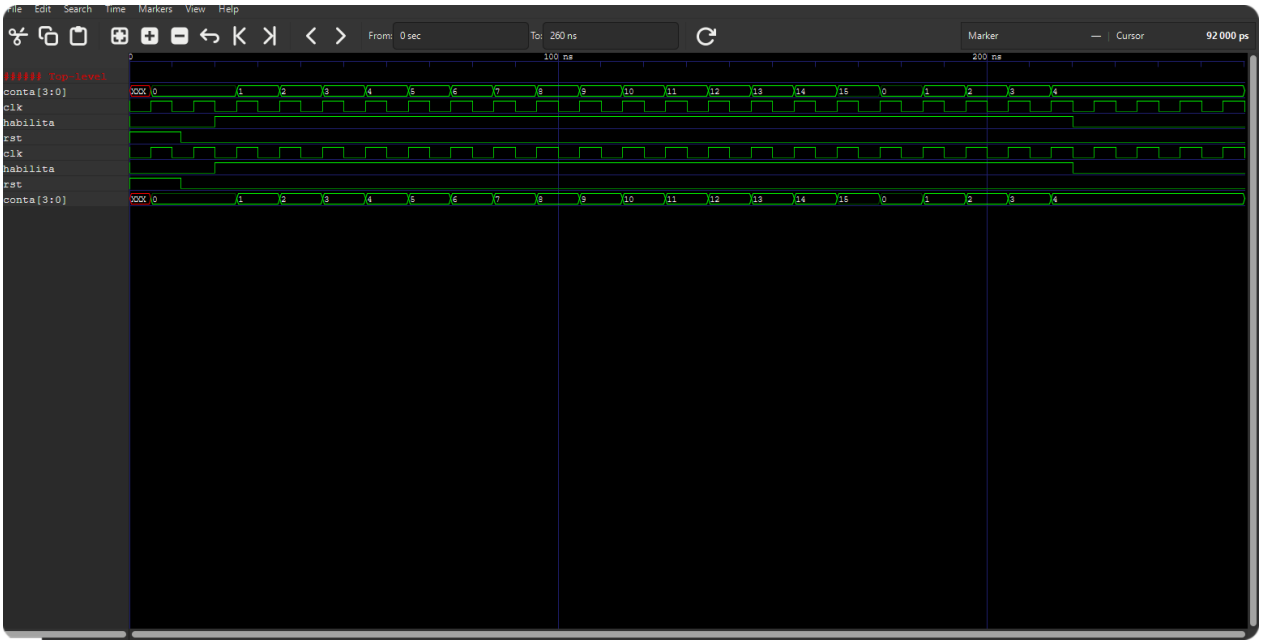


Figura4.4: A forma de onda do contador: o reset solta, habilita sobe e `conta` incrementa a cada borda de clock, de 0 a 15 e de volta a 0.

- 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:

Listagem 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;
```

(continua na próxima página)

(continuação da página anterior)

```
    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;
            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*.

Capítulo 5

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	quase todos: os monitores didáticos dos processadores SAPHO (pilhas, ULA) entram por espelhos no testbench; o miolo mais profundo fica 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 e abre em janela própria; o Surfer é o moderno e abre como **aba do próprio editor**, ao lado dos fontes — a onda e o código na mesma janela. Nos dois casos o layout chega preparado pela AURORA: sinais agrupados, nomeados e coloridos, em vez do despejo bruto.

Quem preferir o Surfer em janela própria troca em *Configurações*, aba *Geral*, *Onde o Surfer abre*; a aba é o padrão. E trabalhar na aba tem uma vantagem a mais: salvar ali de dentro grava o estado no projeto e o registra como layout ativo daquele testbench, e a próxima simulação abre a onda do jeito que você deixou.

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.

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).

Sob o Verilator há uma diferença de grão, que o próprio modal avisa: ele grava por **escopo**, não por sinal. Todo sinal público de um módulo com ao menos um sinal selecionado entra no dump; módulos sem nada selecionado ficam fora; o escopo do testbench é sempre gravado. A seleção continua valendo — ela decide quais módulos entram —, só não desce ao sinal individual.



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. A Aurora Intelligence também sabe **criar** um layout sob encomenda: peça no chat os sinais e a organização que quer, e ela grava o arquivo e o registra no seletor.

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.

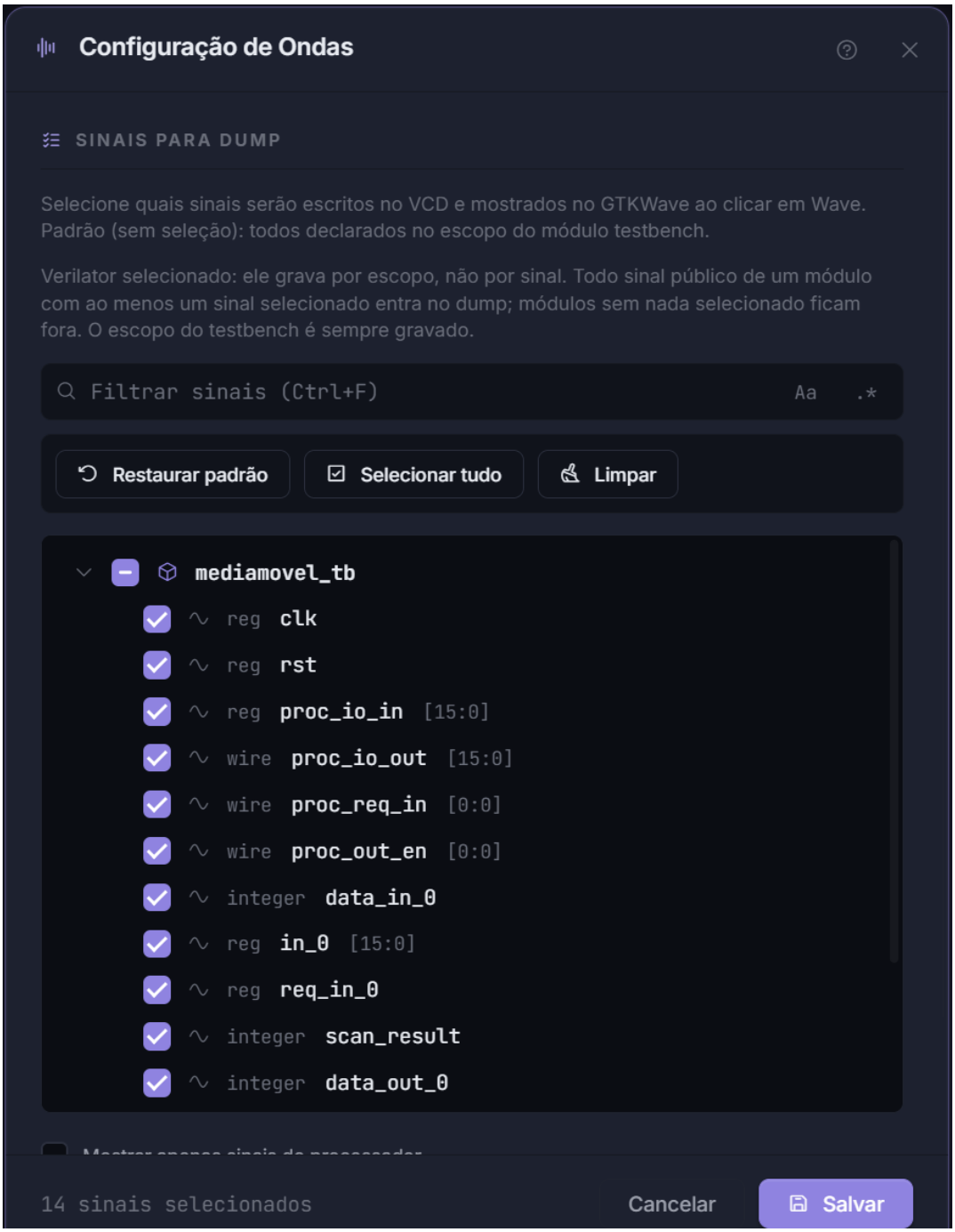
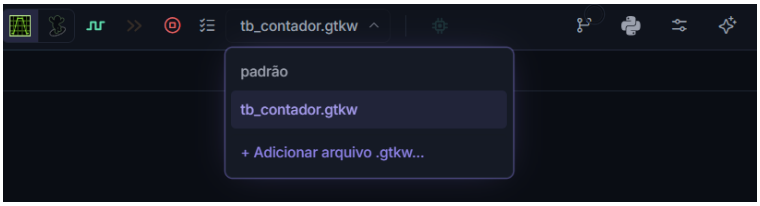


Figura5.1: A árvore reflete a hierarquia do projeto; o filtro aceita texto e expressão regular.



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 O papel de cada fonte

Cada `.v` que entra no projeto, criado ou importado, chega sem papel: aparece com um ponto de interrogação no ícone e fica fora da elaboração e da simulação até você escolher, pelo botão direito, *Marcar como síntese* ou *Marcar como testbench*. Antes de compilar, o terminal lista os arquivos que ficaram de fora por não ter papel.

Os `.py` entram como testbenches cocotb, e o que o YANC gera já nasce com o papel certo: o `<proc>.v` é sintetizável e o `<proc>_tb.v` é testbench. Nada muda de papel sozinho; para trocar, use o mesmo menu.

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:

> **Nota**
>
> Na primeira execução de um testbench cocotb o simulador precisa ser construído, e a AURORA gera um executável a partir do seu Verilog antes de rodar o Python. Isso leva alguns segundos e aparece no terminal TWAVE; da segunda vez em diante o executável é reaproveitado, e só é refeito quando o Verilog muda. Se o antivírus do Windows perguntar sobre esse executável recém-criado dentro da pasta do projeto, é ele.

Listagem 7.1: teste_contador.py

```python
# aurora-toplevel: contador

import cocotb
from cocotb.clock import Clock
from cocotb.triggers import Timer


@cocotb.test()
async def test_contador(dut):
    dut.clk.value = 0
    dut.rst.value = 1
    dut.habilita.value = 0
    clock = Clock(dut.clk, 10, unit="ns")
    cocotb.start_soon(clock.start())

    await Timer(12, unit="ns")
    dut.rst.value = 0

    await Timer(8, unit="ns")
    dut.habilita.value = 1

    await Timer(200, unit="ns")
    dut.habilita.value = 0

    await Timer(40, unit="ns")
```

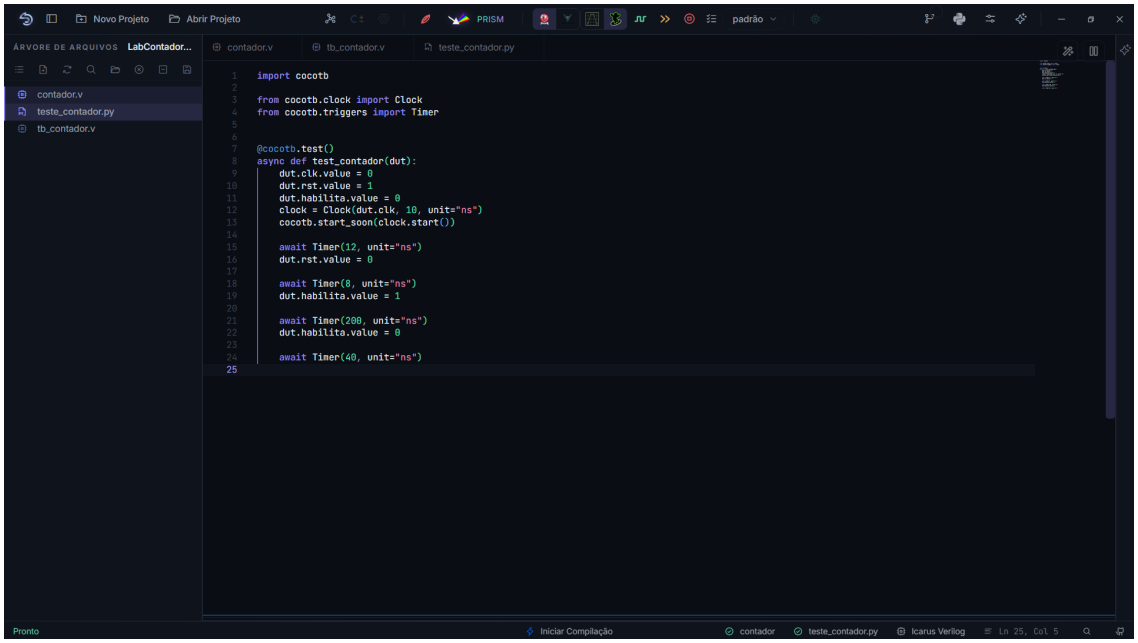


Figura7.1: A primeira linha importa: o comentário `# aurora-toplevel: contador` diz qual módulo Verilog é o alvo do teste. Fora da AURORA a linha é um comentário inerte.

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

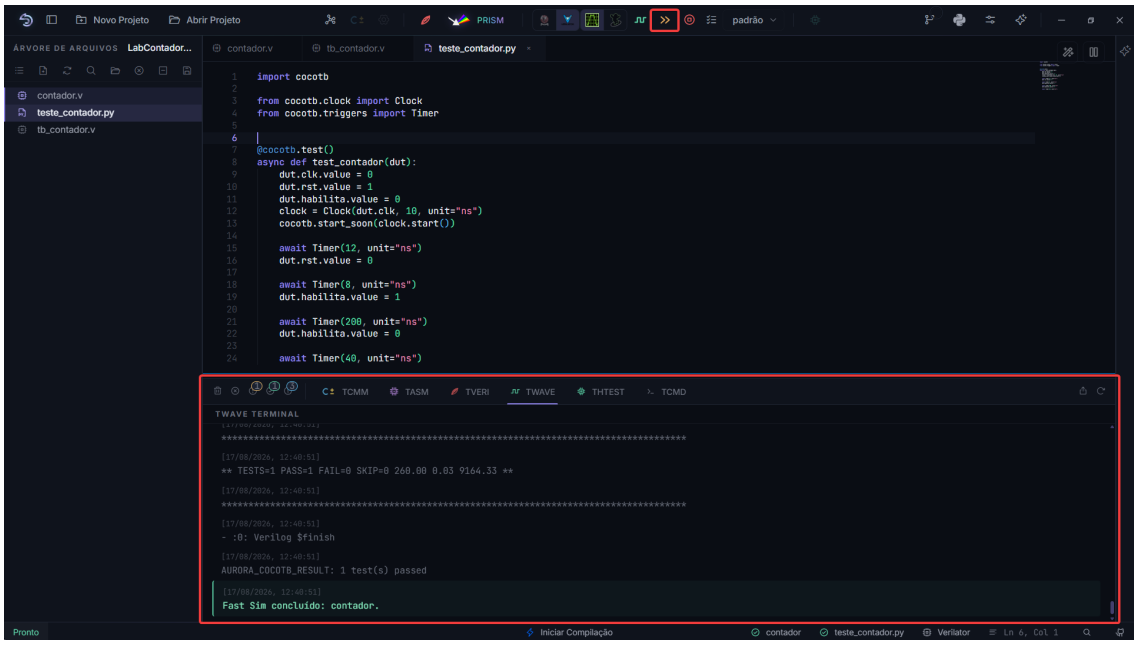


Figura7.2: Na execução rápida, o veredito de cada teste sai no terminal TWAVE.

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, componentes 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.

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*.

Aviso

Se você ainda estiver com o projeto do contador aberto, feche-o ou confira os papéis antes de simular. Top Level e Testbench Top são marcações do projeto ativo, e um `contador.v` esquecido no papel de Top Level faz a compilação do filtro simular o circuito errado, sem erro nenhum. A barra de status mostra os dois o tempo todo: neste tutorial, os dois nomes têm que ser do `media_movel`.

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 arquivo fonte e do módulo Verilog; só letras sem acento, dígitos e _
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. 128 é o padrão, e o formulário só o escreve no fonte se for outro valor
Portas de Entrada	`1`	Uma porta para receber as amostras
Portas de Saída	`1`	Uma porta para publicar a média

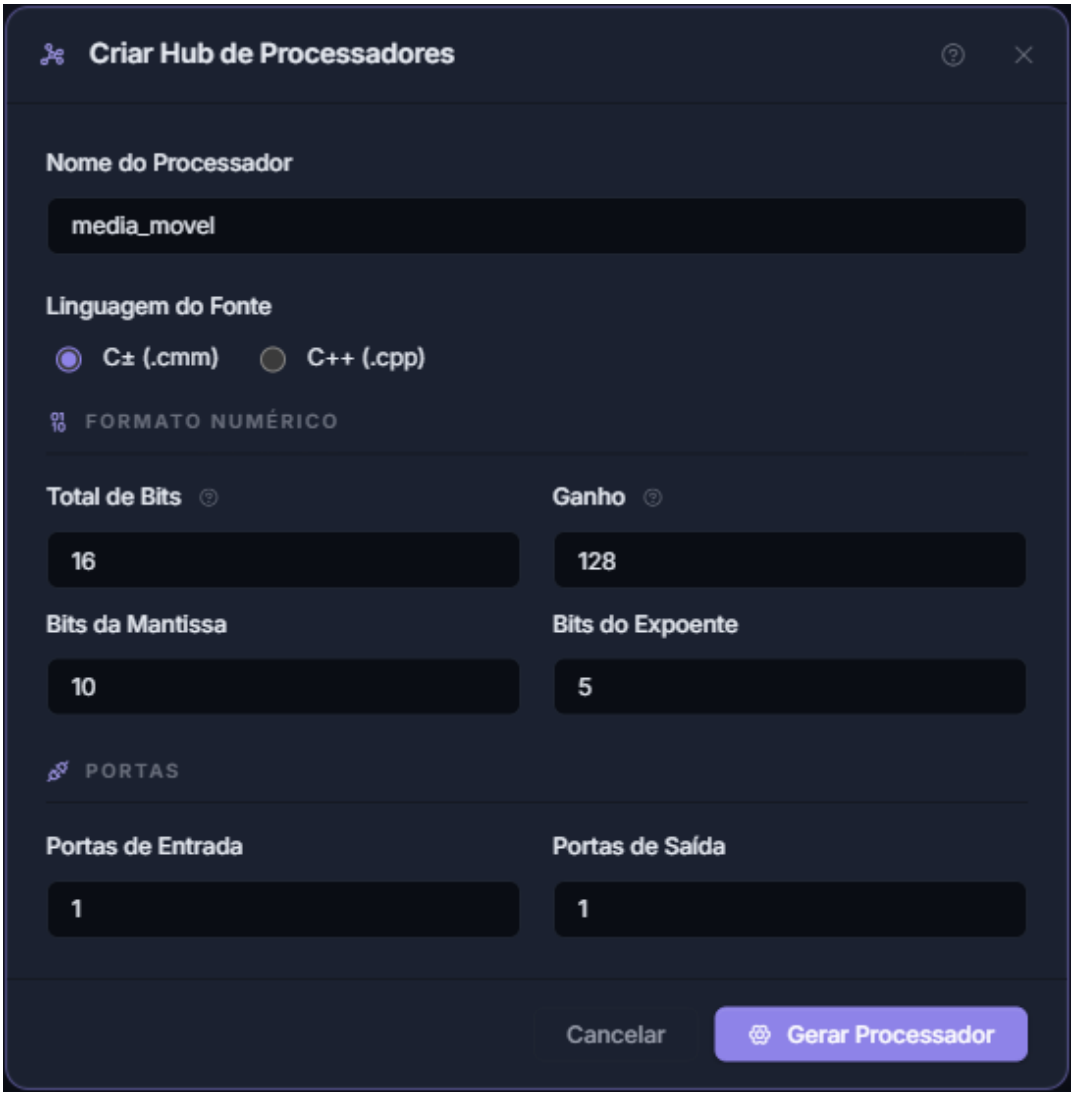


Figura8.1: A validação roda enquanto você digita, e o botão só habilita com tudo consistente.

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*.

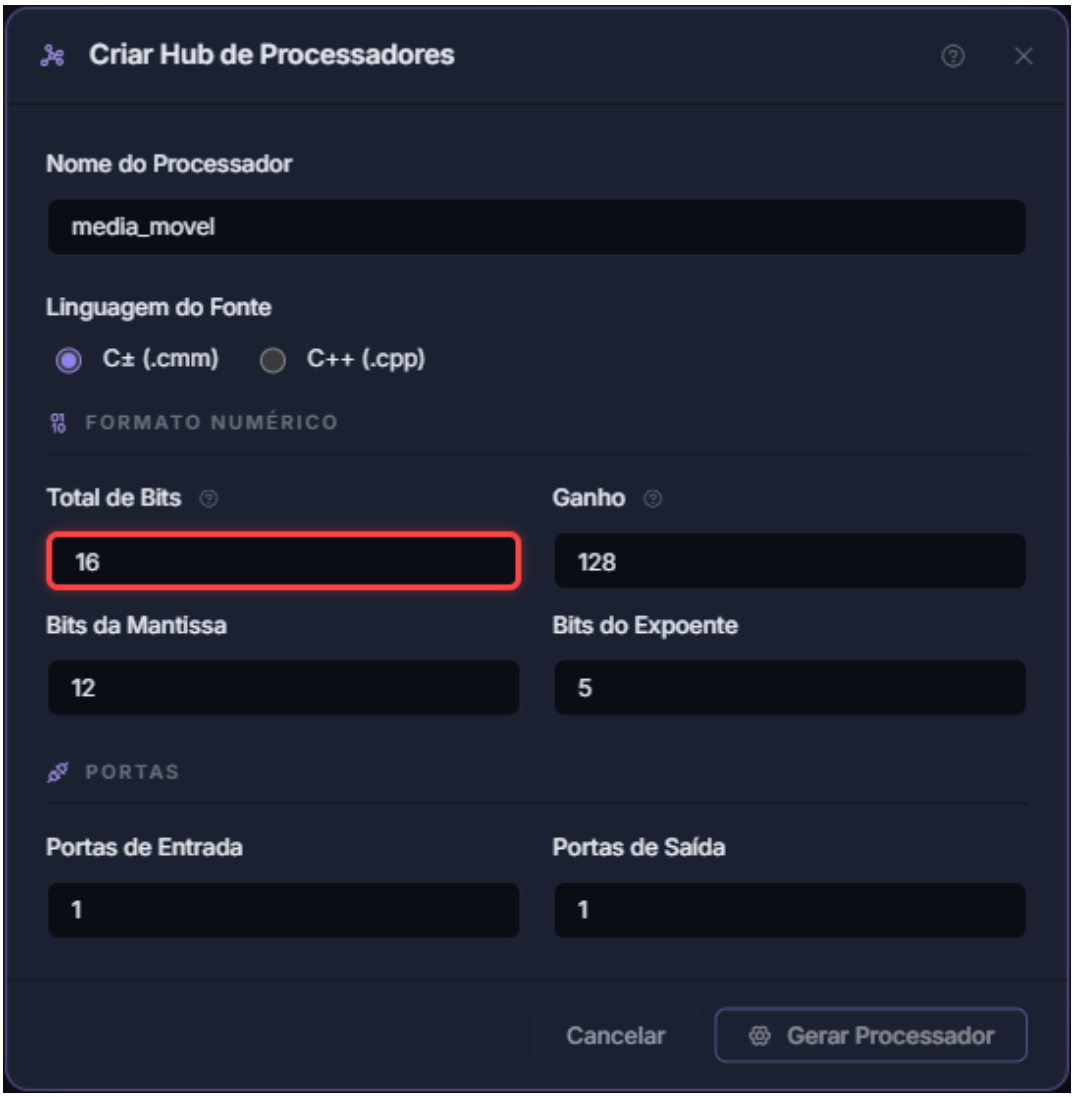


Figura8.2: Um campo fora da regra fica com a borda vermelha e trava o botão.

O formulário não pergunta o tamanho das pilhas: o compilador calcula a profundidade de cada uma pelo programa.

Confira: a árvore mostra o processador `media_movel`, com o arquivo `media_movel.cmm` dentro, nascido com o cabeçalho preenchido pelo formulário. A pasta de saída, `media_movel/media_movel/`, aparece na primeira compilação.

8.3 Passo 3: escrever o algoritmo

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

Listagem 8.1: media_movel.cmm

```
#NUBITS 16
#NUIOIN 1
#NUIOOU 1
#NBMANT 10
#NBEXPO 5

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:

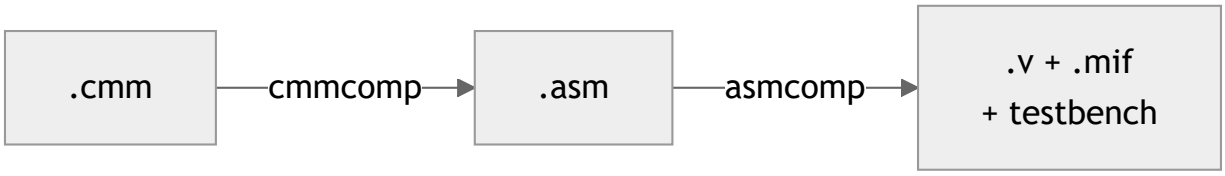
As cinco primeiras linhas são diretivas
Não são código executável: configuram o hardware que será gerado. O formulário as escreveu, e a partir de agora o arquivo é a fonte da verdade. Editar uma diretiva muda o processador na próxima compilação. O nome do processador não aparece aqui: ele é o nome do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

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

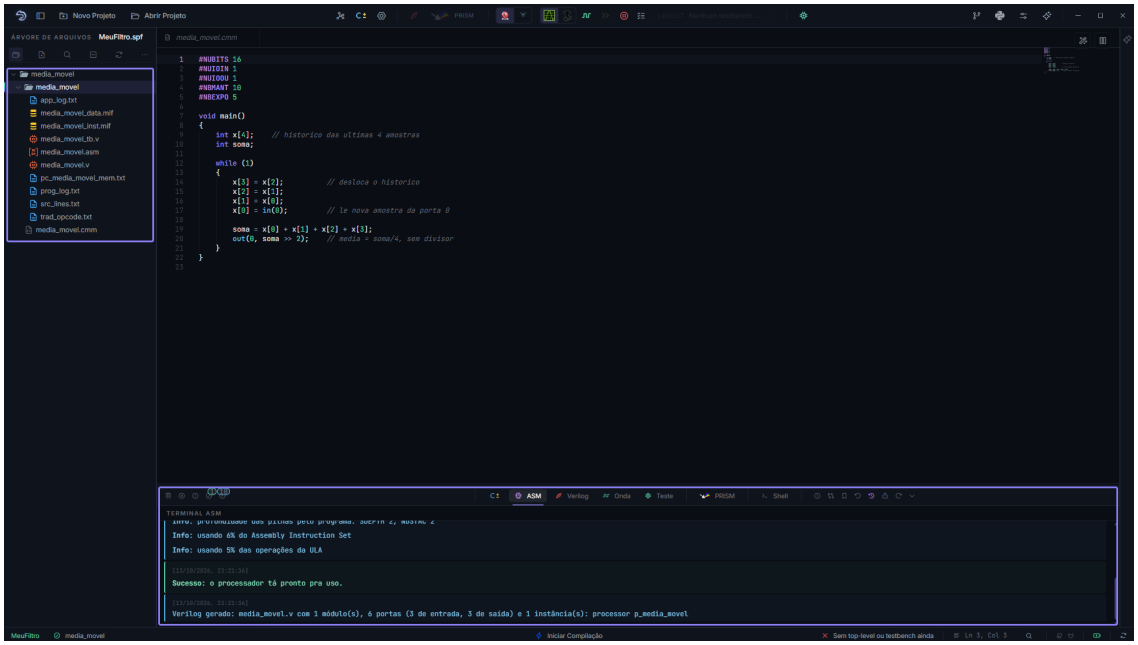
Com o `.cmm` em foco, clique em *Compilar C±*. Dois compiladores rodam em sequência:

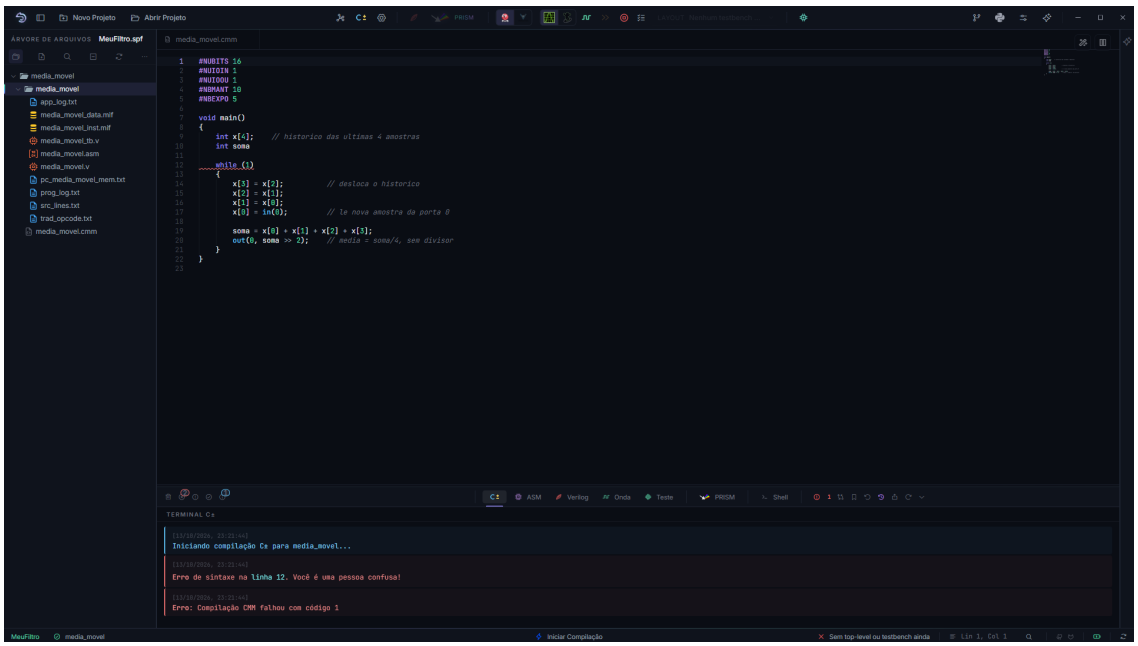


O terminal TCMM mostra a tradução para assembly; o TASM mostra a montagem e, o mais interessante, os avisos de recurso instanciado: cada bloco de hardware que o seu programa ligou no circuito.

Confira: apareceu a pasta de saída `media_movel/media_movel/`, com `media_movel.v` (o processador), `media_movel_inst.mif` e `media_movel_data.mif` (as memórias) e o testbench gerado, `media_movel_tb.v`.

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.


Novo Projeto
Abrir Projeto
ÁRVORE DE ARQUIVOS
MeuFiltro.spf
media_movel
app_log.txt
media_movel_data.mif
media_movel_inst.mif
media_movel_tb.v
media_movel.asm
media_movel.v
pc_media_movel_mem.txt
prog_log.txt
src_lines.txt
trad_opcode.txt
media_movel.cmm
#NUBITS 16
#NUIOIN 1
#NUIOOU 1
#NBMANT 10
#NBEXPO 5
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
}
}
ASM
Verilog
Onda
Teste
PRISM
Shell
TERMINAL ASM
Info: usando 6% do Assembly Instruction Set
Info: usando 5% das operações da ULA
Sucesso: o processador tá pronto pra uso.
Verilog gerado: media_movel.v com 1 módulo(s), 6 portas (3 de entrada, 3 de saída) e 1 instância(s): processor p_media_movel
MeuFiltro
media_movel
Iniciar Compilação
Sem top-level ou testbench ainda


Novo Projeto
Abrir Projeto
ÁRVORE DE ARQUIVOS
MeuFiltro.spf
media_movel
app_log.txt
media_movel_data.mif
media_movel_inst.mif
media_movel_tb.v
media_movel.asm
media_movel.v
pc_media_movel_mem.txt
prog_log.txt
src_lines.txt
trad_opcode.txt
media_movel.cmm
#NUBITS 16
#NUIOIN 1
#NUIOOU 1
#NBMANT 10
#NBEXPO 5
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
}
}
ASM
Verilog
Onda
Teste
PRISM
Shell
TERMINAL C±
Iniciando compilação C± para media_movel...
Erro de sintaxe na linha 12. Você é uma pessoa confusa!
Erro: Compilação CMM falhou com código 1
MeuFiltro
media_movel
Iniciar Compilação
Sem top-level ou testbench ainda

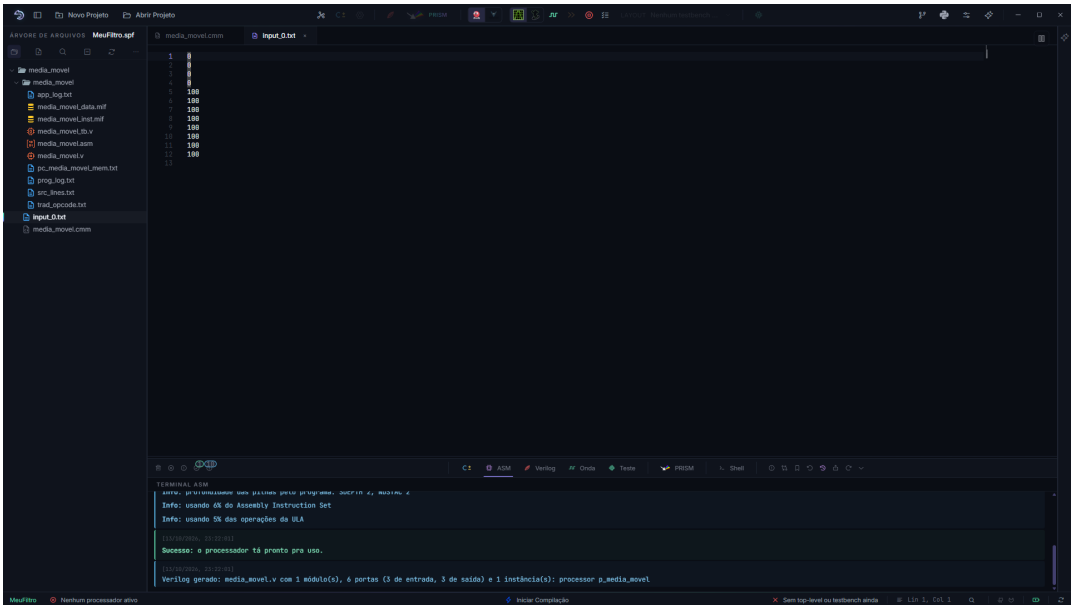
8.5 Passo 5: preparar o estímulo e simular

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 na pasta `media_movel` (a do fonte, não a de saída), *Novo arquivo*, nomeie `input_0.txt`.
2. Escreva um degrau: quatro zeros e depois valores 100.

Listagem 8.2: media_movel/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.

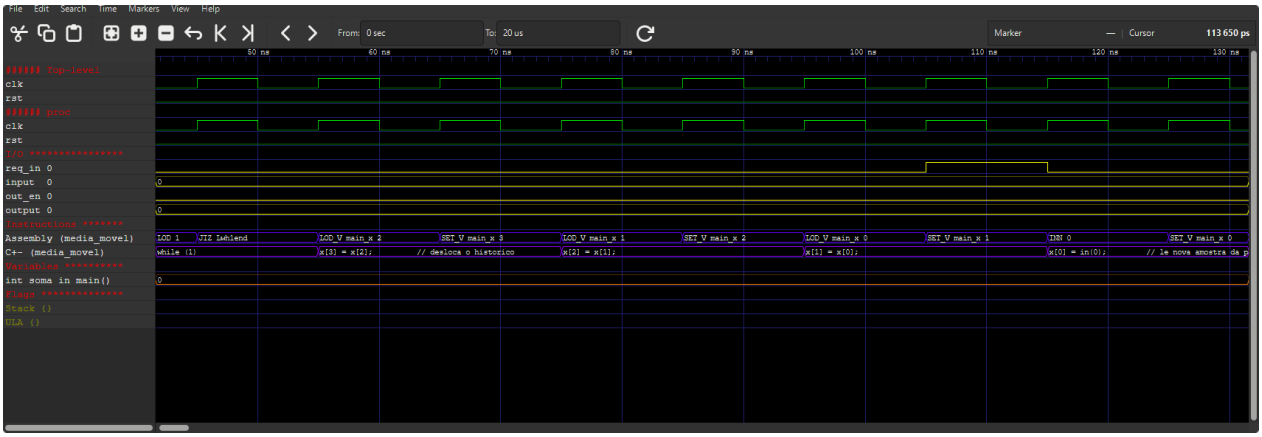


Figura8.3: O degrau entra e sai suavizado em quatro passos, o comportamento esperado de uma média móvel de quatro amostras.

Procure na janela:

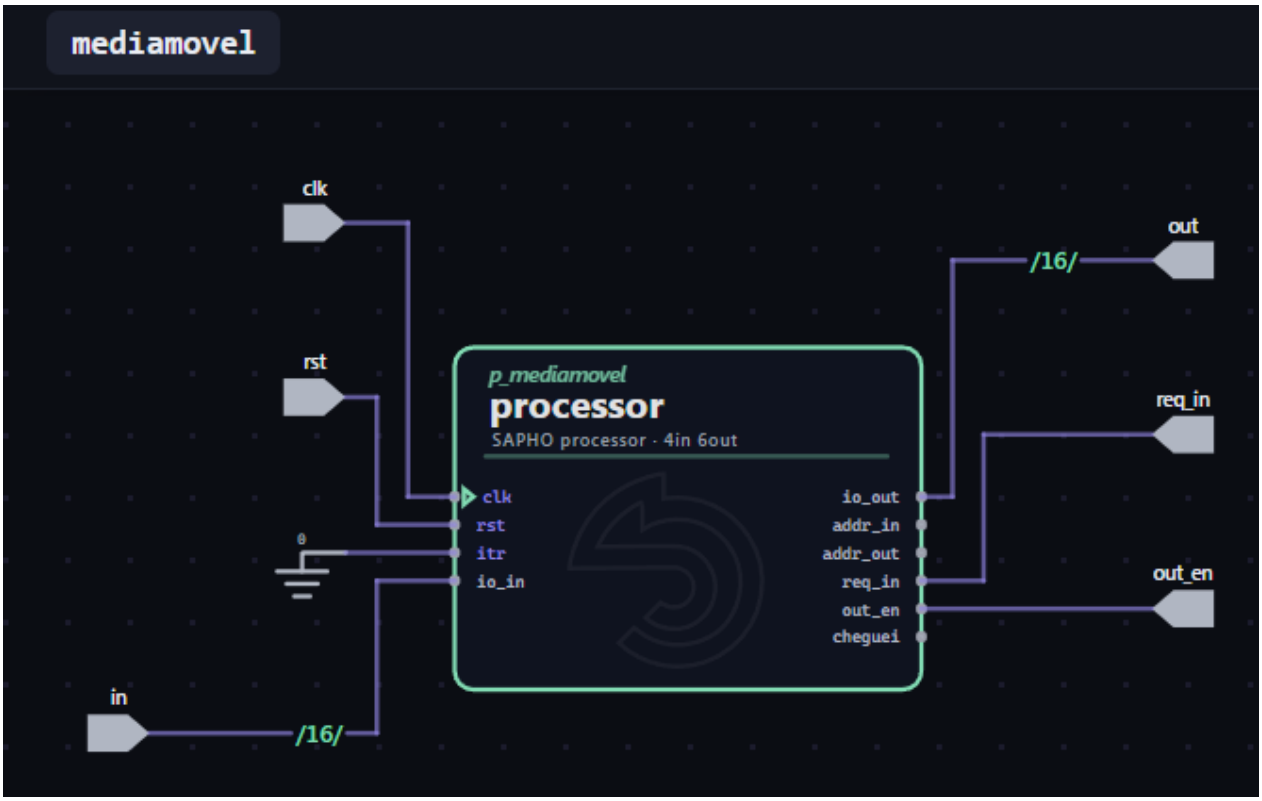
- as variáveis `x[0]` a `x[3]` deslizando a cada amostra;
- `soma` subindo em degraus de 100 conforme o histórico se preenche;
- a porta de saída com o degrau suavizado;
- as trilhas de texto que mostram, a cada ciclo, a instrução assembly executada e a linha do seu C± correspondente. É o seu programa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

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

8.7 Passo 7: ver o circuito por dentro

1. Botão direito em `media_movel/media_movel/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.



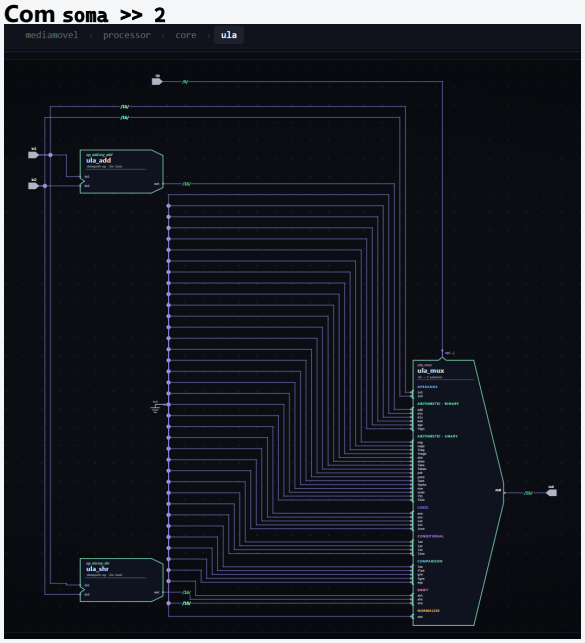
Dica

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.

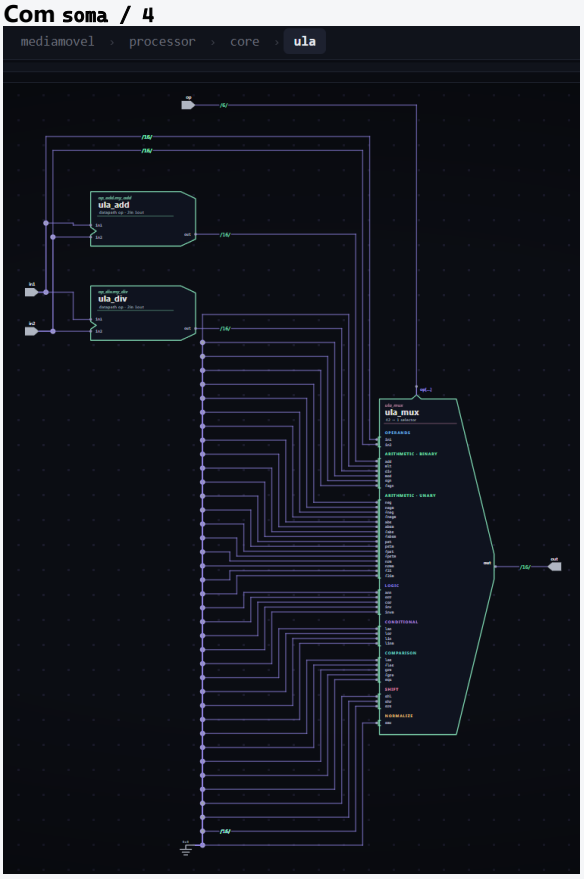
Para enxergar o divisor, desça até a ULA: no diagrama recompilado, clique no processador para entrar nele e depois em *ula*. No nível do topo nada muda de aparência, porque as portas externas do processador continuam as mesmas; o que muda é o conteúdo da unidade aritmética.

A comparação é direta. Com `soma >> 2`, dentro da `ula` moram o somador e o deslocador; com `soma / 4`, o deslocador dá lugar ao divisor, um bloco visivelmente mais caro. É o `generate if` do capítulo *Os módulos HDL do SAPHO por dentro* acontecendo à sua frente: o parâmetro `SHR` foi a zero e o `DIV` foi a um.

Com `soma >> 2`



Com `soma / 4`



Fica o convite para o capítulo do PRISM: além do diagrama, ele tem um modo *Simular*, que liga o módulo da tela com relógio e entradas clicáveis. A `ula` é um bom alvo, pequena o bastante para o modo aceitar, e dá para ver o divisor dividindo. O tour completo está em *PRISM: o visualizador de RTL*.

8.8 O que você aprendeu

- Um processador é uma pasta com o fonte e os dados, criada pelo Hub, e o que o YANC gera fica numa pasta de saída com o mesmo nome, dentro dela.
- As diretivas no topo do `.cmm` definem o hardware; o corpo define o comportamento.
- Compilar produz Verilog, imagens de memória e um testbench, por dois 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: *C± em uma página*. Ou, se preferir ver os detalhes da cadeia de compilação, *Compilação e artefatos*.

Capítulo 9

C± em uma página

C± é um dialeto de C reduzido ao que faz sentido virar hardware. Esta página traz o mínimo para seguir os tutoriais e ler os exemplos do manual. A referência completa da linguagem, das diretivas, da biblioteca e das instruções está no manual do YANC[2], o compilador do SAPHO, escrito junto com ele e atualizado a cada versão.

9.1 Tipos e variáveis

`int` é inteiro com sinal, na largura de `#NUBITS`; `float` é o ponto flutuante do processador, no formato de `#NBMANT` e `#NBEXPO`; `comp` é um número complexo. Não existem `char`, `double`, `bool`, `struct`, ponteiros nem strings. O identificador `i` é a unidade imaginária: usá-lo como nome de variável não compila.

```
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: cada variável tem endereço fixo, e é por isso que ela aparece pelo nome na forma de onda.

Detalhes: tipos[3], variáveis e vetores[4], complexos[5].

9.2 Operadores, controle e funções

Os operadores de C, menos `--`, os compostos (`+=` e família), o ternário, o cast e o `sizeof`. `>>>` é o deslocamento que preserva o sinal. `if`, `while`, `do while`, `for`, `switch`, `break`, `continue` e `return` funcionam como em C. `main()` é obrigatória; funções recebem até nove parâmetros por valor, não aceitam vetor como parâmetro e não podem ser recursivas.

[2] https://www.nipscern.com/library/yanc/
[3] https://www.nipscern.com/library/yanc/language-cmm.html#types
[4] https://www.nipscern.com/library/yanc/language-cmm.html#variables-and-arrays
[5] https://www.nipscern.com/library/yanc/language-cmm.html#complex-numbers

> **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.

Detalhes: operadores[6], controle de fluxo[7], funções[8], o que não existe[9].

9.3 Entrada e saída

O processador conversa com o mundo por portas numeradas, em número definido 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 é um literal. Na simulação do processador sozinho, cada porta de entrada lê de `input_N.txt`, ao lado do fonte, e cada saída grava em `output_N.txt`, na pasta de saída.

9.4 Links entre processadores

Dois processadores de um mesmo projeto trocam valores por links numerados: `toma(k, x)` entrega `x` pelo link `k` e espera o outro lado pegar; `cade(k)` e `fcade(k)` esperam um valor e o recebem como `int` ou `float`. Simulado sozinho, o processador lê os valores de cada `cade(k)` de `cade_k.txt`, ao lado do fonte, e grava cada `toma(k, x)` em `toma_k.txt`, na pasta de saída.

Os pinos que cada link cria, como ligá-los no top level e as formas `_sync`: links no manual do YANC[10].

9.5 Diretivas

O bloco no topo do arquivo configura o processador. O nome do processador não é diretiva: é o nome do arquivo. O Hub escreve sempre estas cinco:

Diretiva	Configura
`#NUBITS`	largura da palavra
`#NBMANT`, `#NBEXPO`	formato do ponto flutuante
`#NUIOIN`, `#NUIOOU`	portas de entrada e de saída

`#NUBITS` deve ser igual a `#NBMANT + #NBEXPO + 1`. Faltar uma diretiva não dá erro: o processador nasce com o padrão do compilador. As pilhas o compilador calcula pelo programa.

Todas as diretivas, com os padrões: diretivas no manual do YANC[11].

[6] https://www.nipscern.com/library/yanc/language-cmm.html#operators
[7] https://www.nipscern.com/library/yanc/language-cmm.html#control-flow
[8] https://www.nipscern.com/library/yanc/language-cmm.html#functions
[9] https://www.nipscern.com/library/yanc/language-cmm.html#what-does-not-exist
[10] https://www.nipscern.com/library/yanc/multiprocessor.html#links-toma-and-cade
[11] https://www.nipscern.com/library/yanc/directives.html

9.6 Biblioteca

As funções embutidas vão das baratas, que viram uma ou poucas instruções (`abs`, `pset`, `sign`, `norm`), às matemáticas, que viram rotinas anexadas ao programa (`sqrt`, `sin`, `exp`, `log`…). Cada uma usada aparece no relatório do TASM.

A lista completa, com o custo de cada uma: biblioteca no manual do YANC[12].

9.7 C++

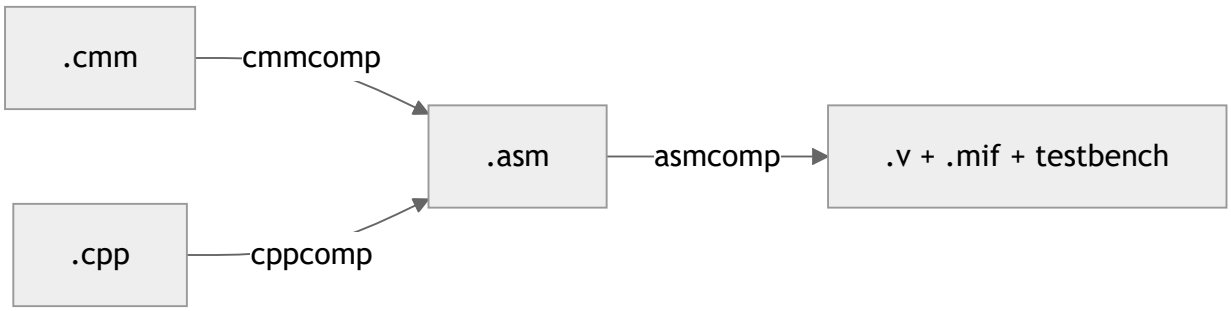
O mesmo processador também se programa em C++, com classes, referências e parte da biblioteca padrão; os parâmetros entram por `#pragma yanc`. O que o compilador aceita: o C++ do YANC[13].

[12] https://www.nipscern.com/library/yanc/library.html
[13] https://www.nipscern.com/library/yanc/language-cpp.html

Compilação e artefatos

O botão *Compilar C±* esconde o YANC, o compilador do SAPHO. A AURORA chama um programa só, o `yanc`, em duas etapas, e ele chama os compiladores que estão ao lado dele. Conhecer a cadeia ajuda a ler os terminais e a saber onde procurar cada arquivo gerado.

10.1 A cadeia



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

cppcomp
: O compilador do C++, a outra linguagem do SAPHO: aceita um subconjunto de C++ com classes, `for`, referências e parte da biblioteca padrão, e produz o mesmo `.asm`. Ele mesmo resolve `#include` e `#define`, e as mensagens de erro apontam o arquivo e a linha que você escreveu, inclusive dentro de um header. Os parâmetros do processador entram por `#pragma yanc` em vez das diretivas `#`; sem pragmas, o processador nasce com 32 bits e ponto flutuante IEEE 754 de precisão simples. Também escreve no **TCMM**.

asmcomp
: O montador e gerador de hardware. Uma primeira passada conta instruções e dados e resolve os endereços dos rótulos; a segunda 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.

Qual dos dois compiladores roda, a AURORA decide pela extensão do fonte: o mesmo botão compila um `.cmm` e um `.cpp`, e o Processor Hub cria processadores nas duas linguagens.

As mensagens saem no idioma escolhido nas configurações da AURORA, português ou inglês. As do cppcomp saem sempre em inglês, e o terminal avisa disso uma vez. O que cada ferramenta faz, as opções da linha de comando e o que cada mensagem quer dizer estão no manual do YANC[14].

10.2 O que cada compilação gera

Para um processador chamado `media_movel`, tudo vai para a pasta de saída, `media_movel/media_movel/`:

[14] https://www.nipscern.com/library/yanc/command-line.html

Arquivo	O que é
`media_movel.asm`	o assembly gerado, legível; vale abrir e comparar com o seu C±
`media_movel.v`	o processador em Verilog, pronto para simulação e síntese
`media_movel_inst.mif`	a imagem da memória de programa, uma instrução binária por linha
`media_movel_data.mif`	a imagem da memória de dados, com variáveis e constantes
`media_movel_tb.v`	o testbench gerado, com clock, reset e a leitura e escrita das portas
`prog_log.txt`, `src_lines.txt`, `pc_media_movel_mem.txt`, `trad_opcode.txt`, `app_log.txt`	as tabelas que a forma de onda usa para mostrar o assembly e a linha do fonte

O nome de tudo vem do nome do arquivo fonte: `media_movel.cmm` gera o processador `media_movel` e o módulo Verilog `media_movel`.

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 lê os `input_N.txt` da pasta do processador, um nível acima, e grava as saídas na pasta de saída. Ele roda com a pasta de saída como diretório de trabalho, e a AURORA cuida disso. Um testbench seu fica onde você o pôs e nunca é sobrescrito.

Aviso

Um arquivo presente na pasta de saída 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 Cada clique fica registrado

Todo botão de compilação grava uma execução: o que foi pedido, quando, quanto durou, que ferramentas rodaram e como terminou, junto com um retrato do projeto naquele momento (Top Level, testbench, simulador, visualizador, processadores e fontes). O botão do relógio na barra dos terminais (`Ctrl+Shift+H`) abre o *Histórico de compilações*, que lista as execuções e se atualiza enquanto uma compilação acontece: a execução viva aparece no topo, com os passos e o tempo crescendo.

Serve para responder o que o terminal sozinho não responde: o que exatamente foi clicado antes de dar errado, com que Top Level, em que ordem. Os registros ficam no próprio projeto, em `.aurora/execucoes/`,

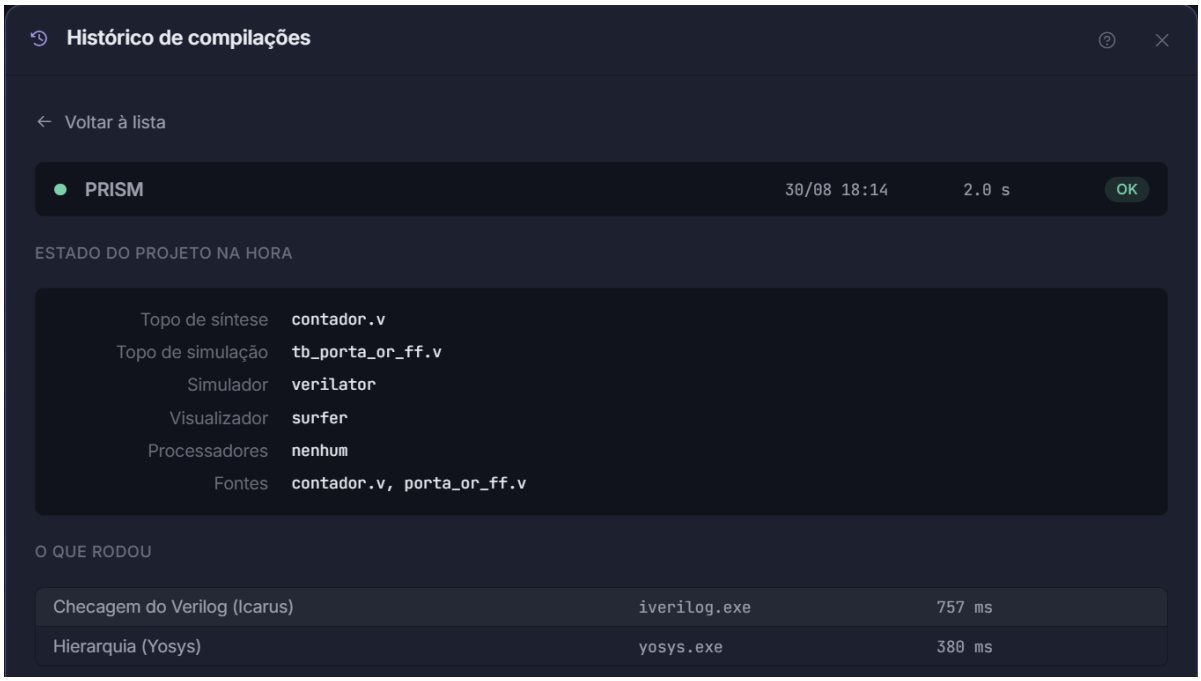


Figura10.1: Uma execução aberta: o estado do projeto na hora e a cadeia de ferramentas que rodou, com a duração de cada passo.

um JSON por execução. Duas compilações ao mesmo tempo geram dois registros; um passo que não dá para atribuir com certeza a uma delas sai marcado como sobreposto, e não como erro.

10.5 Lendo erros de compilação

- As mensagens do cmmcomp apontam a linha do `.cmm`, e as do cppcomp, o arquivo e a linha (inclusive de um header); nos dois casos 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, as suas variáveis continuam visíveis, e os monitores didáticos do processador — as pilhas e os erros de arredondamento da ULA — entram na onda por espelhos que a AURORA declara no testbench. Um programa sem funções não tem pilha de instruções, então o monitor dela nem aparece; não é defeito. Para enxergar qualquer sinal interno sem exceção, use o Icarus.

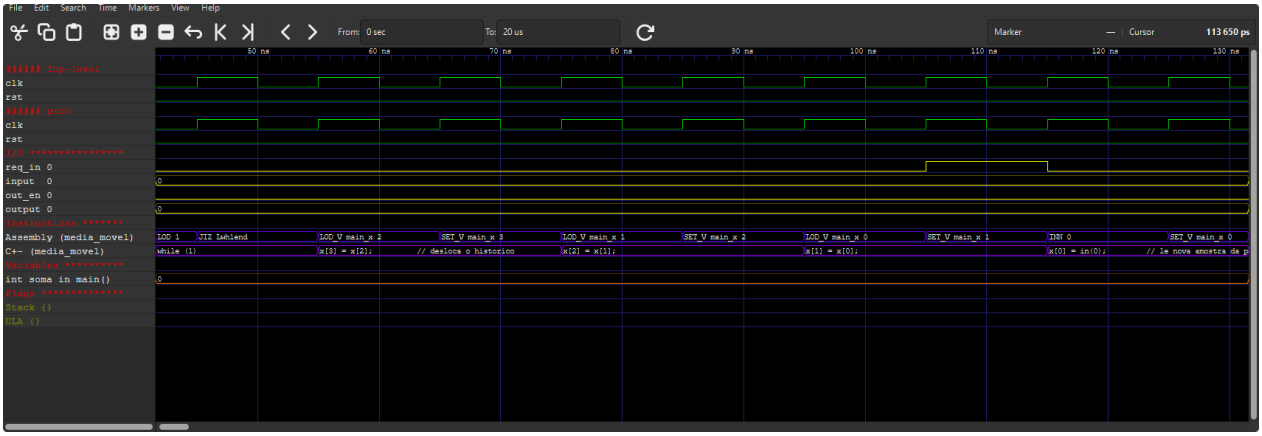
Há ainda uma quarta lupa, fora desta página: o modo *Simular* do PRISM, que liga um módulo isolado com relógio e entradas clicáveis (*PRISM: o visualizador de RTL*). Vale para entender um pedaço do circuito, não para rodar o programa.

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
`<proc>/input_N.txt`	estímulo da porta de entrada N, um inteiro por linha (você escreve, ao lado do fonte)
`<proc>/<proc>/output_N.txt`	o que o programa escreveu na porta N (a simulação gera, na pasta de saída)
`<proc>/cade_k.txt`, `<proc>/<proc>/toma_k.txt`	o mesmo para os links entre processadores, quando o processador é simulado sozinho
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 visualizador de RTL

O PRISM é o visualizador de RTL da plataforma: ele lê o Verilog do projeto e o desenha como um diagrama de circuito navegável. Responde à pergunta que a forma de onda não responde: em que estrutura o meu código virou hardware? E, com o modo *Simular*, responde também a seguinte: o que essa estrutura faz quando recebe valores?

O desenho é sempre um retrato do Verilog atual. Quem produz a estrutura é o Yosys, elaborando os mesmos arquivos que iriam para a síntese, então qualquer mudança no código muda o diagrama: acrescente um módulo e ele aparece, troque um operador e o bloco correspondente entra ou some, renomeie um sinal e o rótulo acompanha. Nada ali é decorativo ou desenhado à mão.

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 o diagrama. O andamento dessa síntese, com seus avisos, sai no terminal **TPRISM**.

Por padrão o PRISM abre em janela própria. Quem prefere o diagrama ao lado do código troca em *Configurações*, aba *Geral*, a opção *Onde o PRISM abre*: ele passa a abrir numa aba do editor, como um arquivo.

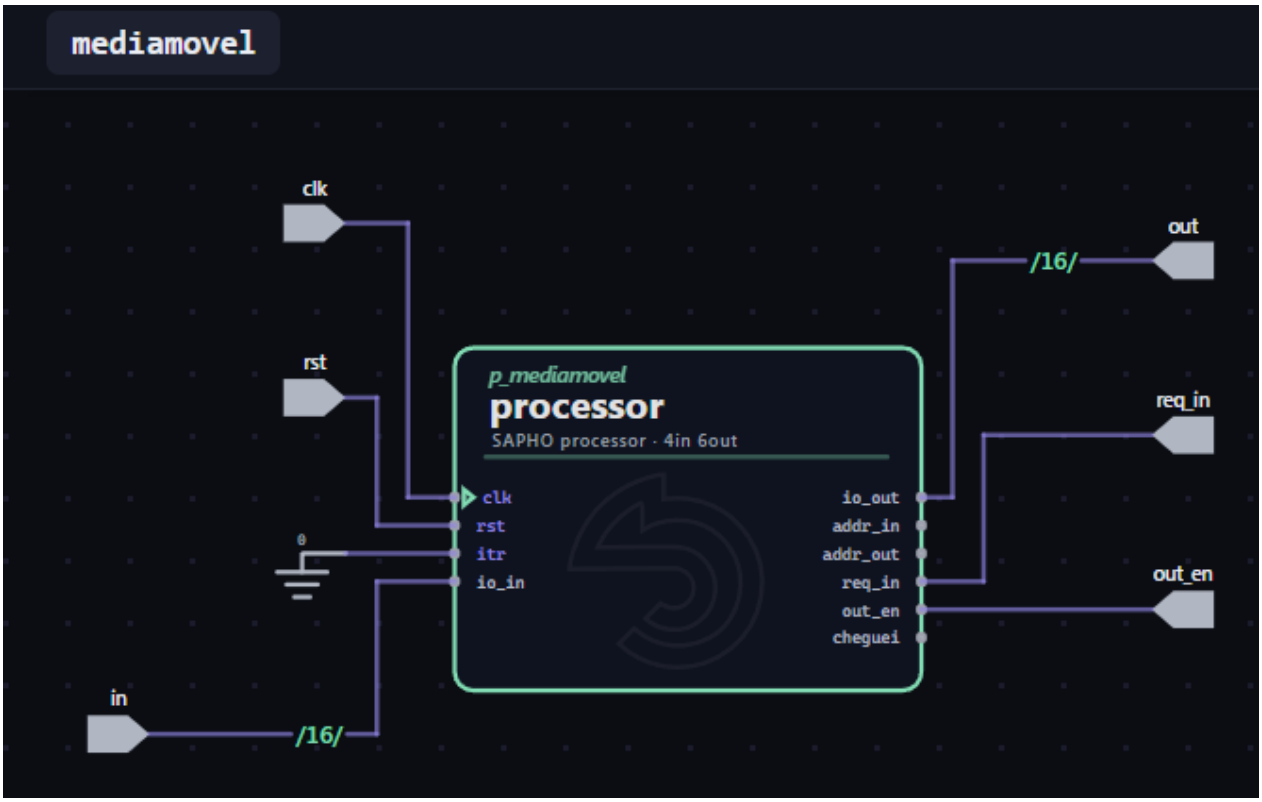


Figura12.1: O nível do topo. Processadores SAPHO aparecem com símbolo próprio; os demais módulos, como caixas com suas portas.

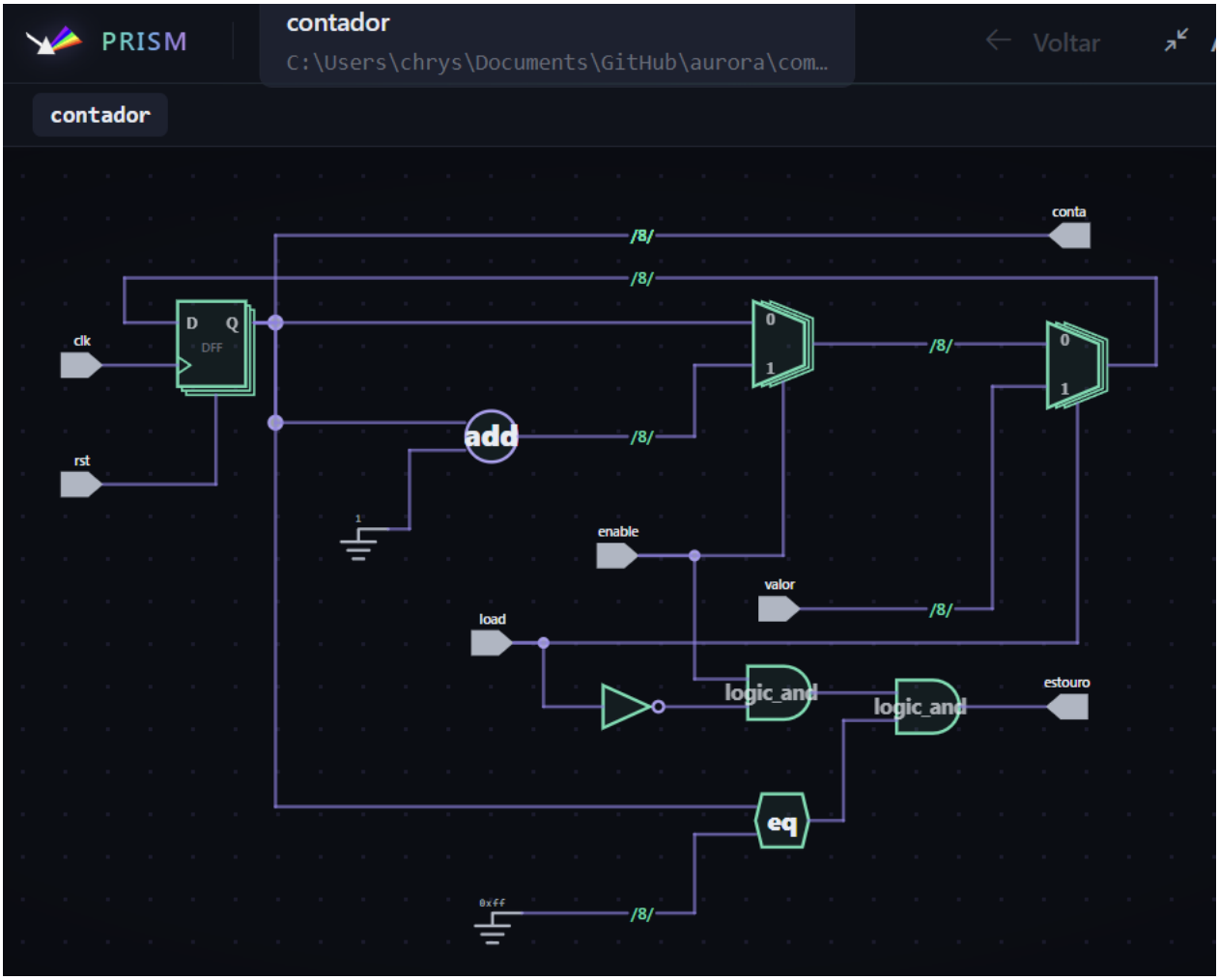


Figura12.2: O mesmo diagrama numa aba do editor, para trabalhar com o código e o circuito lado a lado.

12.2 Navegar

- **Clique** em qualquer ponto de um módulo entra nele: o diagrama desce um nível na hierarquia, e a trilha no topo mostra o caminho. *Back* sobe.
- **Shift+clique** em uma célula acende todas as conexões que a tocam; o menu de contexto oferece o mesmo em *Destacar conexões*. Clicar em um fio destaca só aquele fio.
- **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.
- Cada barramento carrega uma etiqueta com a largura sobre o próprio fio (`/32/`), e células que o Yosys batiza com nomes gerados, cheios de `$`, aparecem rotuladas pelo tipo (`mem read`, `dff`) em vez do nome ilegível.
- *Download* salva o diagrama atual como SVG, pronto para relatório ou slide.
- *Recompile* refaz tudo após uma mudança no código, sem fechar o PRISM.

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.

> **Dica**
>
> **Em que nível olhar.** O divisor, o multiplicador e os demais blocos de operação moram dentro da ULA, e não no topo. Depois do *Recompile*, clique no processador para descer um nível, e então em *ula*: é ali que o bloco novo aparece, ao lado dos que já existiam. No nível do topo você só veria o mesmo retângulo de antes, porque as portas externas do processador não mudam quando a aritmética interna muda.

12.4 Simular

O botão *Simular* troca o desenho pelo circuito vivo. Vale ser preciso sobre o que ele liga: o módulo que está na tela naquele momento, e não o Top Level do projeto. Desça até um contador e clique em *Simular*, e é o contador que acorda, com as portas dele viradas para você. O mesmo botão, agora *Esquemático*, devolve o desenho.

A montagem roda o Yosys de novo sobre o código atual e entrega a estrutura a um simulador lógico que vive dentro da própria AURORA; um véu cobre a tela enquanto isso acontece. Não é o Icarus nem o Verilator: é uma simulação de lógica, sem noção de nanossegundos, feita para se ver e se tocar. A simulação séria continua sendo a dos capítulos anteriores; esta é uma lupa.

12.4.1 O tempo

Uma barra sobre o circuito comanda o tempo: *Rodar* e *Pausar* (também com `Espaço`), *Tick* avança um passo (a seta para a direita), *Próximo evento* salta até a próxima mudança de sinal (`Shift` mais a seta), *Rápido* corre sem espera entre os ticks e *Reiniciar* volta ao tick zero com os registradores no valor inicial, sem recompilar e sem mexer nas chaves das entradas. Ao lado ficam o contador de ticks, o meio período do relógio, em ticks, e a velocidade, em ticks por segundo.

Uma entrada de um bit chamada `clk` ou `clock` no módulo simulado vira um relógio de verdade, que bate sozinho; os registradores nascem em zero, como depois de ligar a placa. Módulo sem relógio não tem o que bater: avance com *Tick*.

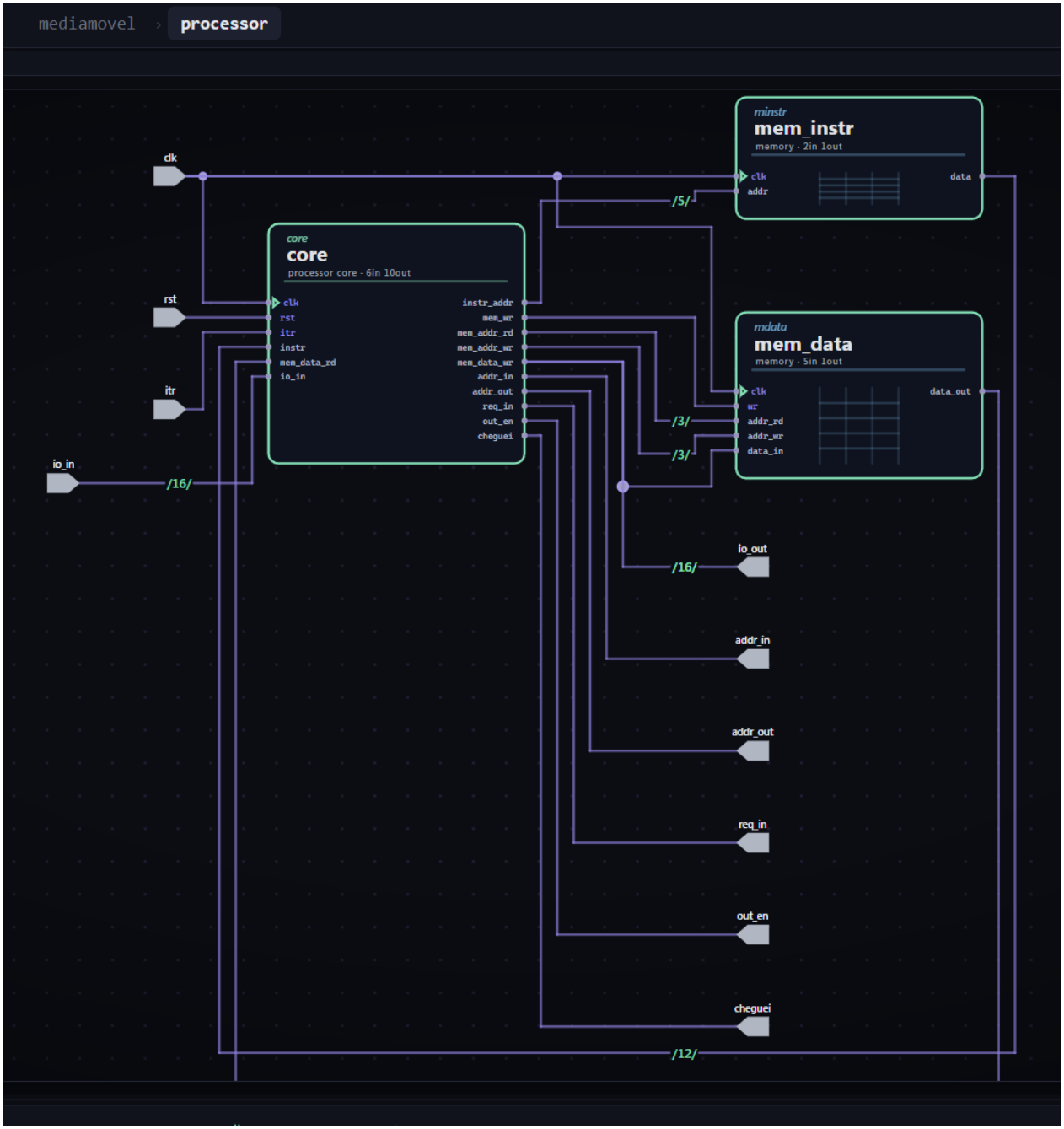


Figura12.3: Dentro de um processador SAPHO: a ULA, as memórias e as pilhas têm desenho dedicado.

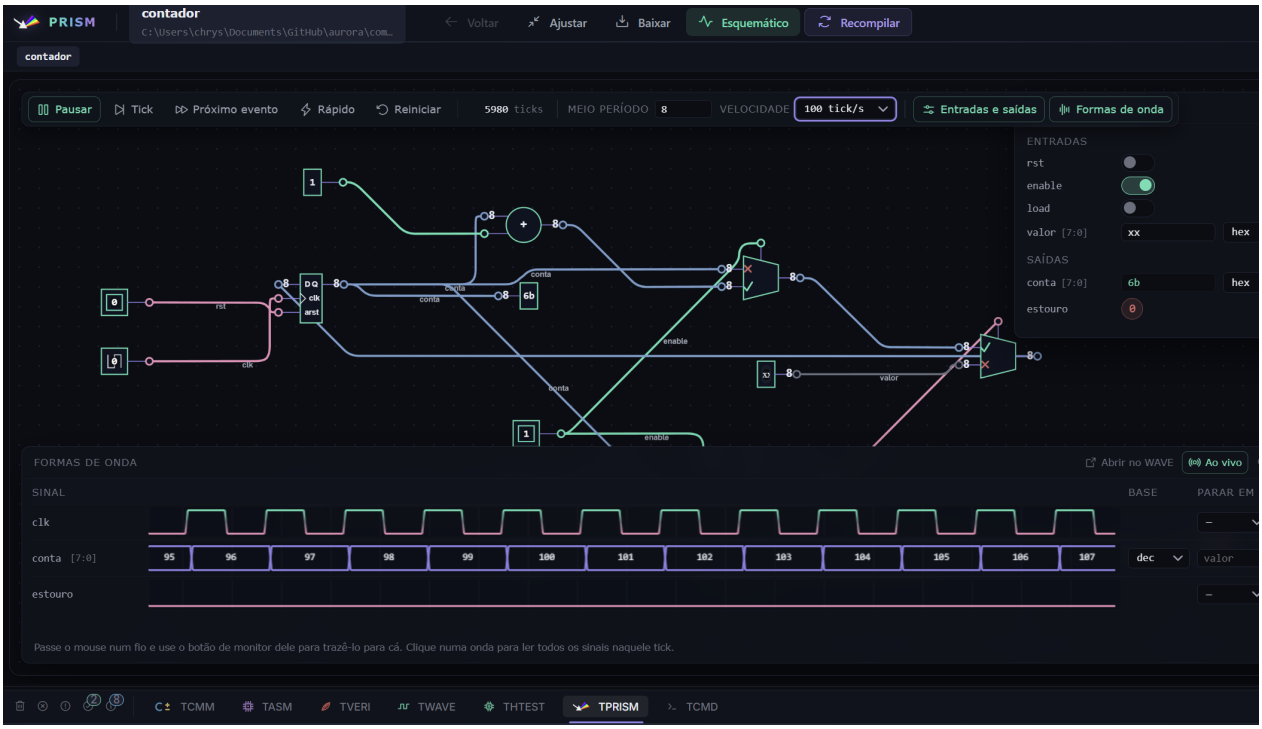


Figura12.4: O contador do tutorial Verilog em simulação: a barra de tempo sobre o circuito, as entradas como interruptores e o monitor de formas de onda.

12.4.2 Entradas e saídas

O painel *Entradas e saídas* lista as portas do módulo com a faixa de bits de cada barramento: as entradas de um bit são interruptores, as de vários bits são campos em que se digita o valor na base escolhida, e as saídas mostram o que o circuito responde.

12.4.3 Formas de onda

O painel *Formas de onda* é o monitor: o relógio e as saídas entram nele de saída, e qualquer outro fio entra pelo botão de monitor que aparece ao passar o mouse sobre ele. Cada linha tem a base em que o valor é lido e um *parar em*, que interrompe a simulação quando o sinal chega ao valor dado, com um aviso dizendo qual sinal parou, em que tick e com que valor.

Um clique numa onda põe um cursor naquele tick, e cada linha passa a mostrar o valor que tinha ali; `Esc` tira o cursor. *Ao vivo* acompanha o presente, e desliga sozinho quando você arrasta a onda para olhar o passado; a lupa aproxima e afasta no tempo. Passar o mouse sobre um fio do diagrama mostra o nome e o valor dele, em hexadecimal, decimal e binário.

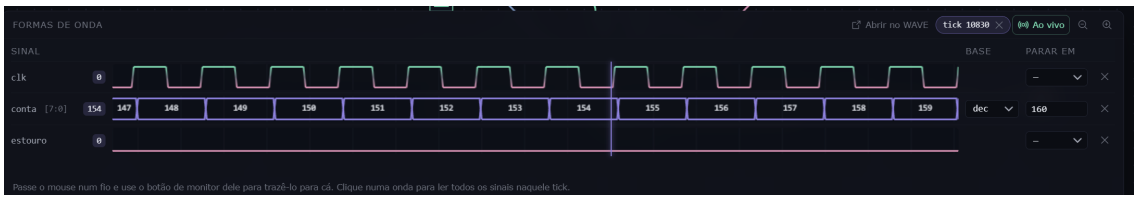
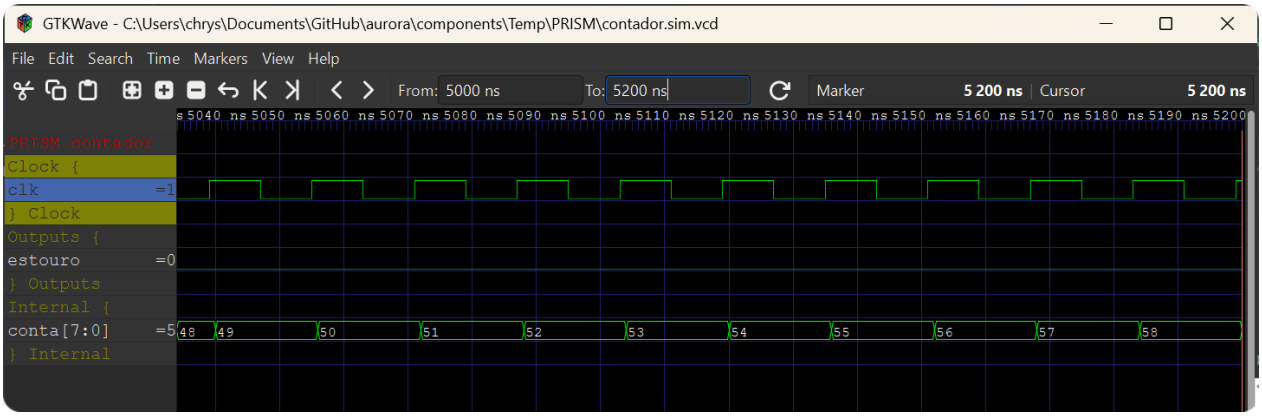


Figura12.5: Um *parar em* disparou: o aviso diz o sinal, o tick e o valor, e o cursor mostra o que cada linha valia ali.



12.4.4 Da lupa para o osciloscópio

Abrir no WAVE grava os sinais do monitor num `.vcd` e o abre no seu visualizador de ondas, GTKWave ou Surfer conforme a preferência, já com o layout montado: relógio, entradas, saídas e internos agrupados, cada sinal na base que estava no monitor. É o caminho para medir com o cursor e o zoom de sempre aquilo que você acabou de ver acontecer.



12.4.5 Submódulos

A lupa na caixa de um submódulo abre o interior dele no lugar, com a trilha no topo mostrando o caminho; *Voltar* ou `Esc` sobe um nível, e a simulação continua correndo em todos os níveis.

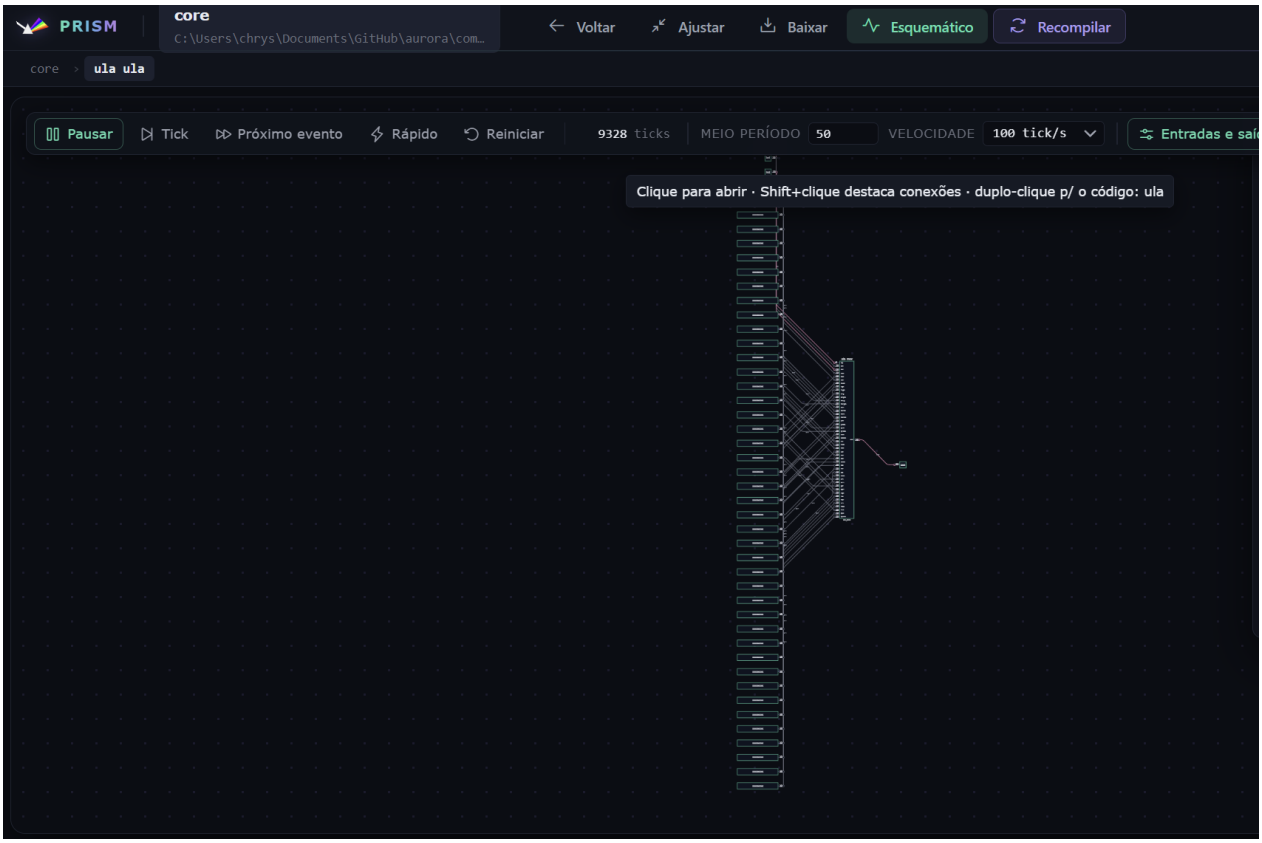
12.4.6 A memória das escolhas

O que se escolheu numa simulação fica guardado por módulo e volta sozinho na próxima vez, e também depois de *Reiniciar*: os sinais do monitor, com base e *parar em*, a velocidade, o meio período e os painéis abertos.

> **Ver também**
>
> A Aurora Intelligence opera esta simulação por ferramentas próprias, e é a única simulação cujos valores ela consegue ler: *Aurora Intelligence*.

12.4.7 Os limites do modo

- A simulação é lógica, sem tempo físico: não existem nanossegundos, atrasos nem `timescale`. Quem mede tempo é a forma de onda.
- Acima de alguns milhares de células o PRISM recusa o modo e sugere abrir um submódulo menor no esquemático e simular aquele. O diagrama estático não tem limite prático.
- Um processador SAPHO inteiro executando o programa continua sendo trabalho do testbench; a lupa serve aos módulos e aos pedaços.



12.5 Limitações

- O diagrama parte sempre do Top Level do projeto. Sem ele, o PRISM não abre (a simulação, depois de aberto, é do módulo na tela).
- Módulos sem desenho dedicado aparecem como caixas genéricas, o que é cosmético, não funcional.
- O PRISM mostra estrutura e comportamento lógico; o comportamento no tempo, com atrasos e clock de verdade, é assunto da forma de onda.

Parte IV

Os dois juntos

Capítulo 13

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 `media_movel/media_movel/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, `media_movel/media_movel/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.

Listagem 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`. Ele entra sem papel: botão direito, *Marcar como síntese*.
2. Botão direito de novo, *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-o com *Marcar como testbench* e depois *Definir como testbench atual*, 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: `<proc>/<proc>/<proc>.v`.
3. As imagens de memória: `<proc>/<proc>/<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/SAPHO` dentro da pasta de instalação: `processor.v`, `core.v`, `ula.v`, `instr_dec.v`, `addr_dec.v` e os demais.

Copie tudo para o projeto da ferramenta de síntese e adicione os arquivos ao projeto dela. O `<proc>.v` nomeia as memórias pelo nome puro (`.IFILE("<proc>_inst.mif")`), então os dois `.mif` têm de ficar na pasta do projeto da ferramenta; se preferir deixá-los em outro lugar, troque os nomes por caminhos completos nos parâmetros `IFILE` e `DFILE`.

> **⚠ 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. O formato em detalhe, as constantes e os níveis de arredondamento de `#FROUND` estão no manual do YANC[15].

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] https://www.nipscern.com/library/yanc/floating-point.html

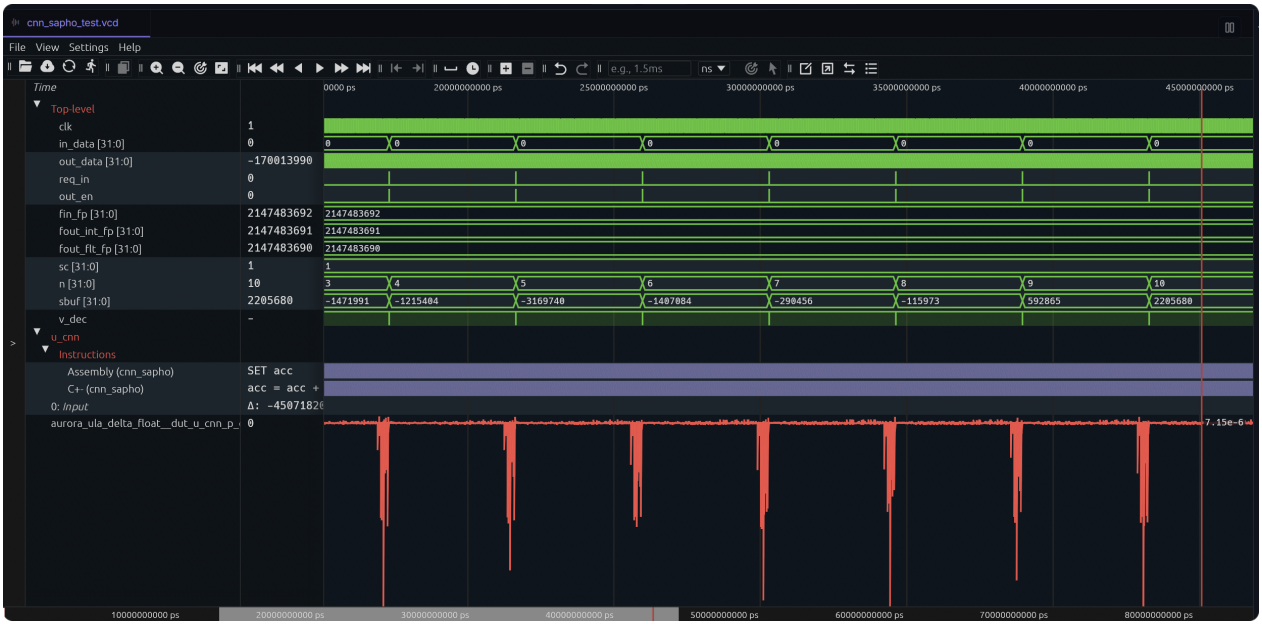
15.3 Dimensionando para o seu problema

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. Algumas funções matemáticas também: `sqrt` usa um bloco de rotação, e `exp` e `log` usam blocos de escala por potência de dois e de extração de expoente, todos condicionados ao mesmo `generate if` das demais operações. As trigonométricas (`sin`, `cos`, `tan`, `atan`) são rotinas de software, avaliadas por polinômios sobre as operações básicas: o custo delas é tempo de execução, não área. O relatório do TASM mostra as duas coisas: os recursos da ULA instanciados e o tamanho do programa.

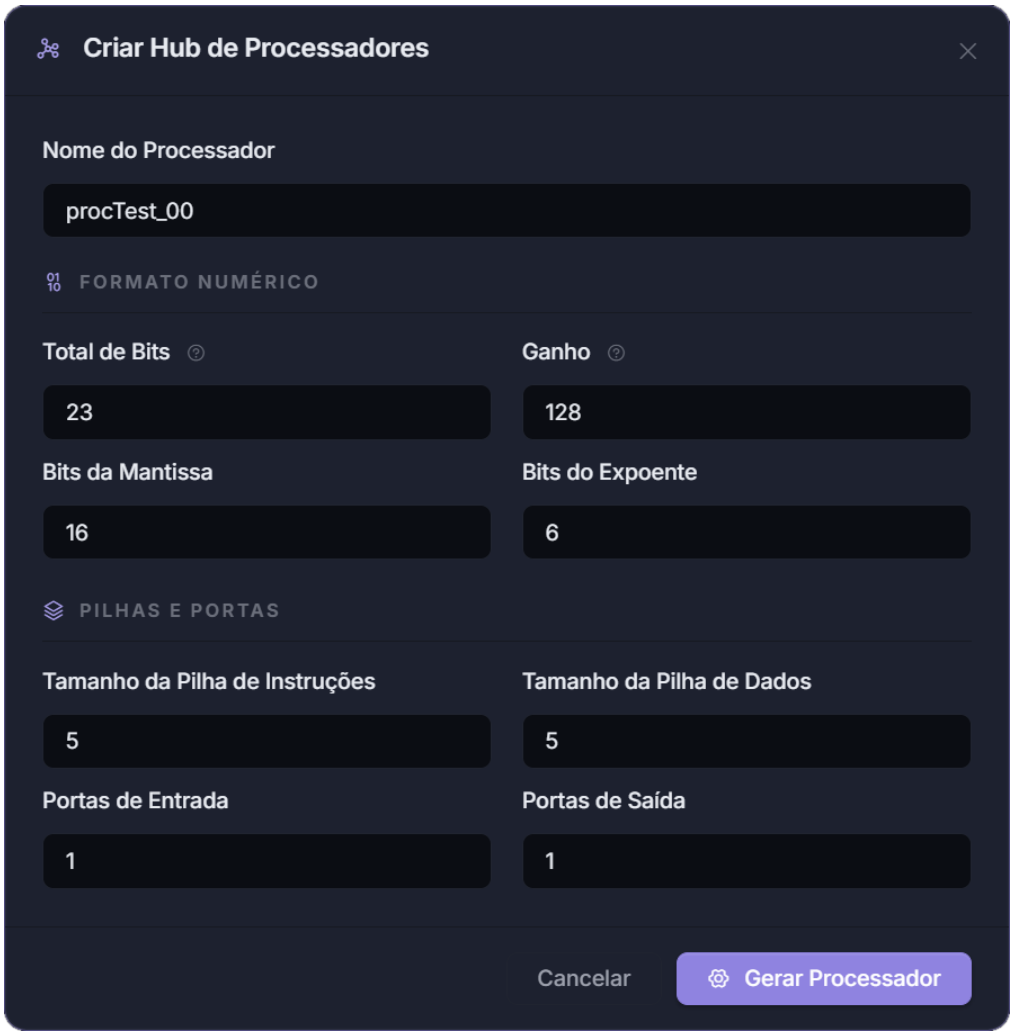


Figura15.1: Os três campos que definem o formato ficam juntos no Hub. Aqui está a configuração padrão, 23 bits com mantissa 16 e expoente 6; para comparar formatos, gere dois processadores iguais mudando apenas mantissa e expoente, rode o mesmo estímulo e compare os `output_N.txt`.

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

A soma, a subtração, a multiplicação e a divisão têm a álgebra completa, e `abs` devolve o módulo. Operadores de bits, `%`, `++` e a notação de Dirac não aceitam `comp`, e `out()` não escreve um complexo direto: escreva `fout(p, real(z))` e `fout(p, imag(z))`.

O que vale para cada operador e função, e as funções de acesso (`real`, `imag`, `mod2`, `fase`, `complex`, `conj`): complexos na linguagem[16] e na biblioteca[17], no manual do YANC.

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:

[16] https://www.nipscern.com/library/yanc/language-cmm.html#complex-numbers
[17] https://www.nipscern.com/library/yanc/library.html#complex-numbers

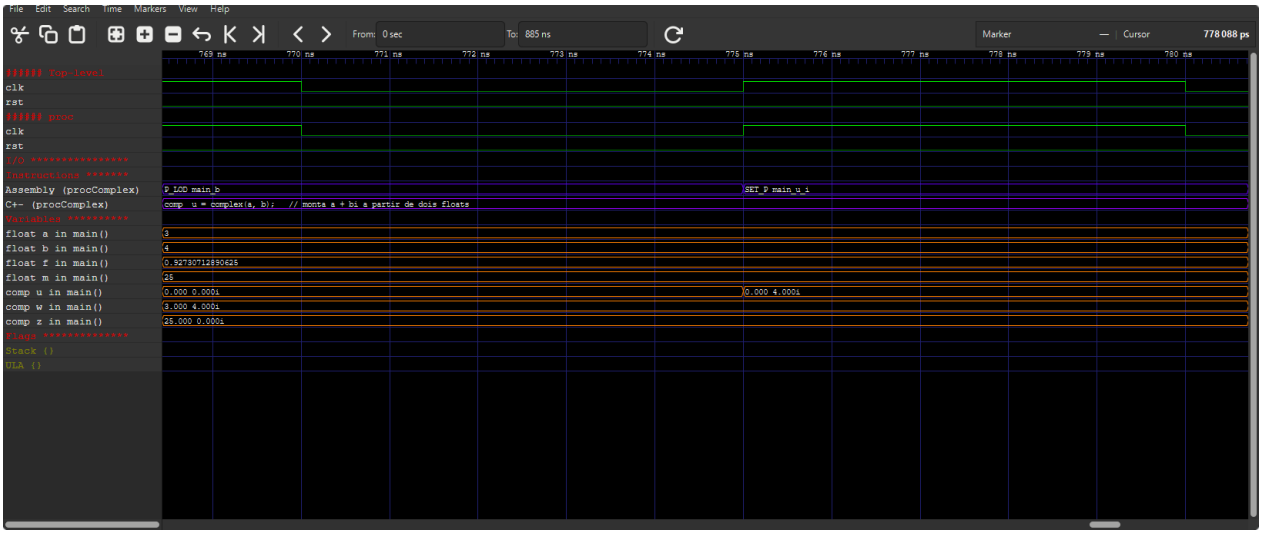


Figura16.1: A decodificação funciona no GTKWave e no Surfer, com o formato de ponto flutuante do processador aplicado automaticamente.

16.5 Exemplo rápido

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

Listagem 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 Os símbolos, e como digitá-los

Antes das formas, o obstáculo prático: dois dos símbolos não estão no teclado. O compilador reconhece exatamente estes quatro, e nenhuma variação deles.

Símbolo	Nome	Código Unicode	O que é
`⟩`	*ket*, fecha	U+27E9	Fecha um vetor coluna, como em `|b⟩`
`⟨`	*bra*, abre	U+27E8	Abre um vetor linha, como em `⟨b|`
`|0⟩`	vetor nulo	usa o U+27E9	Zera o vetor inteiro
`|I|`	identidade	só ASCII	Matriz identidade, esta se digita normalmente

O ponto que causa mais perda de tempo: `⟨` e `⟩` **não são** os sinais de menor e maior do teclado. São caracteres diferentes, com códigos diferentes. Escrever `a # |M|b>;` não compila, e a mensagem de erro fala de sintaxe, sem dizer que o problema é o caractere. Os sinais `<` e `>` continuam sendo comparação em C±, e é por isso que o compilador não pode aceitá-los nos dois papéis.

17.1.1 Na AURORA: o editor completa

Não é preciso caçar o caractere. No editor, dentro de um arquivo `.cmm`, digite o nome da operação e aceite a sugestão:

Digite	E aceite para escrever
`ket`	`|v⟩` com o cursor sobre o nome do vetor
`bra`	`⟨v|`
`braket`	`⟨a|b⟩`, o produto interno
`dirac`	a lista inteira: as quatro operações, a identidade, o vetor nulo e a leitura de porta
`>>` ou `<<`	o caractere `⟩` ou `⟨` sozinho, para inserir no meio de algo já escrito

A lista sai da mesma tabela de símbolos que o compilador usa, extraída do yanc, então ela não pode ensinar um símbolo que o compilador não aceite.

17.1.2 Fora da AURORA

Em qualquer outro editor, três caminhos:

- copiar de um exemplo que já use a notação, que é o mais rápido;
- no Windows, o Mapa de Caracteres (`charmap`), procurando pelo código U+27E8 ou U+27E9;

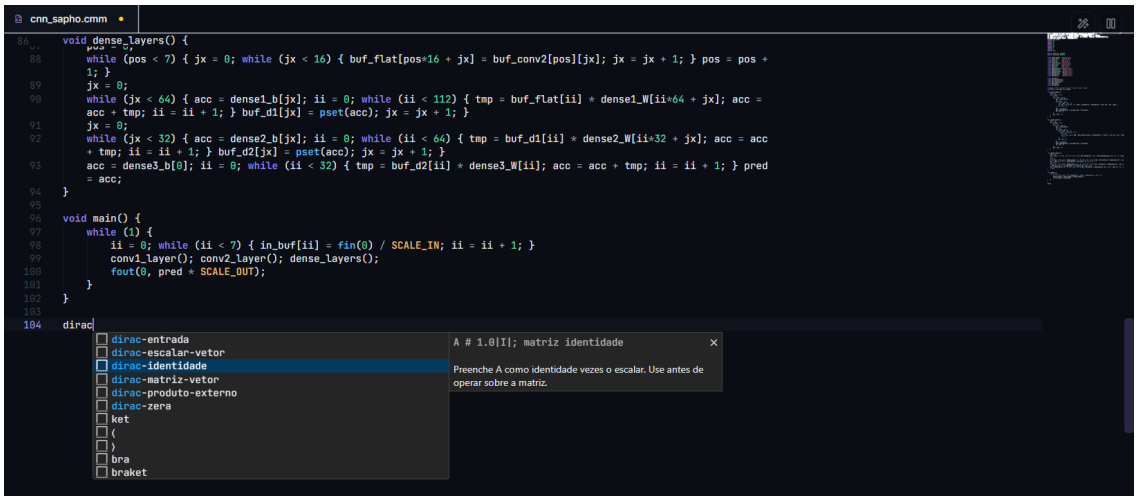


Figura17.1: Digitar `ket` num `.cmm` traz as formas da notação; aceitar a sugestão insere o caractere correto.

- em editores que aceitam entrada Unicode, digitar o código e converter (no Word e no LibreOffice, `27E9` seguido de `Alt+X`).

17.2 As formas suportadas

O `#` marca a atribuição vetorial: `a # |M|b□;` faz $a = Mb$, `A # |a□□b|;` faz o produto externo, e `□w|x□` é o produto interno dentro de qualquer expressão. As dimensões precisam ser compatíveis e os tipos iguais dos dois lados (`int` com `int`, `float` com `float`), sem `comp`.

Todas as formas, com o que cada uma calcula: vetores em notação de Dirac, no manual do YANC[18]. As usadas no tutorial abaixo:

Escrita	Calcula
`e = d - □w|x□;`	produto interno 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 # |B| - |a□□b|;`	$A = B - a\,b^T$
`A # c|B|;`, `A # c|I|;`	escalar vezes matriz, identidade escalada
`a # |0□;`	zera o vetor
`a # c -> |a□;`	registrador de deslocamento: desloca `a` e insere `c`

17.3 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) e use o programa, que fixa a pilha de dados em 8:

Listagem 17.1: proc_rls.cmm, o nucleo do algoritmo

```
#NUBITS 32
#NBMANT 23
#NBEXPO 8
```

(continua na próxima página)

[18] https://www.nipscern.com/library/yanc/library.html#vectors-in-dirac-notation

(continuação da página anterior)

```
#NDSTAC 8
#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.

17.4 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.

proc_rsl.cmm

```
#PRNAME proc_rsl
#NUBITS 32
#NBMANT 23
#NBEXPO 8
#NDSTAC 8
#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);
    }
}
```

Figura17.2: 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` (diretivas no manual do YANC[19]).

18.2 O programa

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

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

```
#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   n, j, k, par, salto;
    float ang;

    while (1)
    {
```

(continua na próxima página)

[19] https://www.nipscern.com/library/yanc/directives.html

(continuação da página anterior)

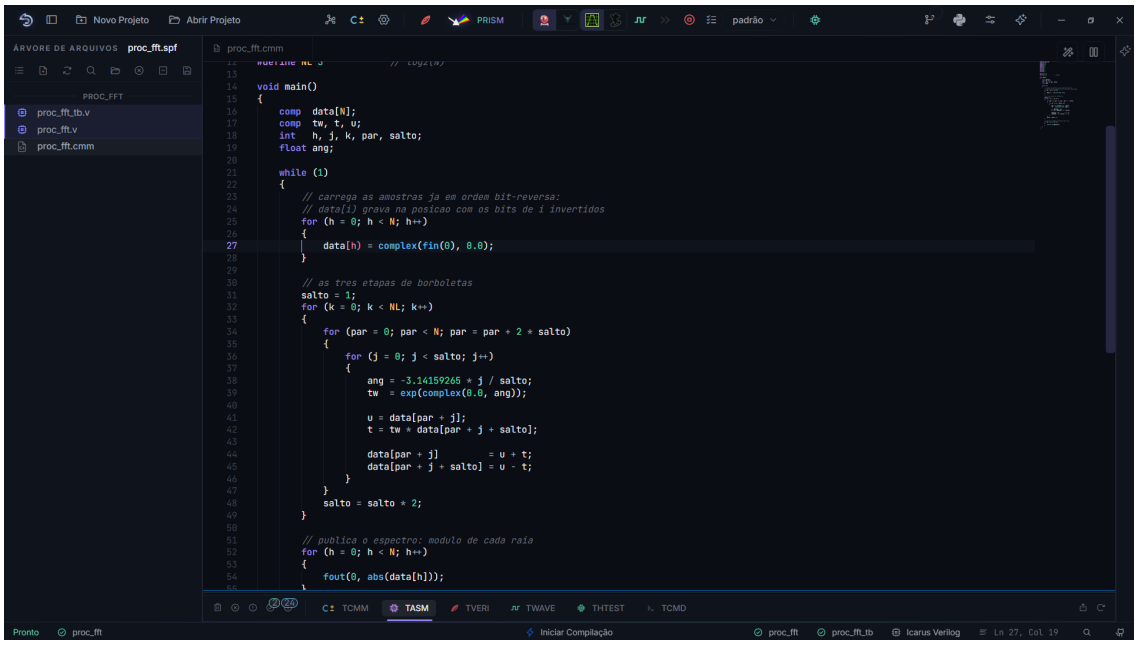
```
22          // carrega as amostras ja em ordem bit-reversa:
23          // data[n) grava na posicao com os bits de n invertidos
24          for (n = 0; n < N; n++)
25          {
26              data[n) = complex(fin(0), 0.0);
27          }
28
29          // as tres etapas de borboletas
30       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

Os pontos de atenção:

- A linha 27 é o truque inteiro: `data[n)` grava a amostra `n` já na posição embaralhada. Nenhuma rotina de reordenação, nenhum ciclo gasto.
- O índice de amostra se chama `n`, e não `i`, porque `i` é reservado no C± e usá-lo como variável é erro de compilação (*C± em uma página*).
- 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)[20].

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

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/SAPHO`:

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
`myFIFO.v`	uma FIFO de uso geral, útil como amortecedor entre processadores no padrão produtor e consumidor de *Interrupção, marcador de fim e multiprocessador*
`myDCFIFO.v`	a mesma FIFO para dois relógios, quando os processadores rodam em clocks diferentes

O arquivo gerado por processador, `<proc>/<proc>/<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 todos os registradores de estado, uma escolha amigável a FPGA. A exceção deliberada são as memórias: para não impedir a inferência de block-RAM, as memórias de instrução e de dados não são resetadas. Depois de um `rst`, o contador de programa volta a zero e o programa recomeça, mas as variáveis guardam o valor da rodada anterior; um programa que dependa de estado inicial deve atribuí-lo explicitamente no começo do `main()`.
- O conjunto completo de instruções, com os 108 opcodes, está no manual do YANC[21], gerado da própria

[21] https://www.nipscern.com/library/yanc/reference/instructions.html

tabela do compilador.

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),
    // ...
    .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.

Listagem 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 (marcadores no manual do YANC[22]). 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.

[22] https://www.nipscern.com/library/yanc/directives.html#markers

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.

Os processadores conversam de dois jeitos. O mais direto são os links: `toma(k, x)` num e `cade(k)` no outro, com a espera e a entrega resolvidas pelo próprio hardware (veja *C± em uma página* e os links no manual do YANC[23], que mostra os pinos e a ligação no top level). O outro é o produtor e consumidor pelas portas: a saída de um alimenta a entrada do outro, com o aperto de mão `out_en` e `req_in`, possivelmente com uma FIFO entre eles quando os ritmos diferem (FIFOs no manual do YANC[24]).

20.3.1 O top level como sistema

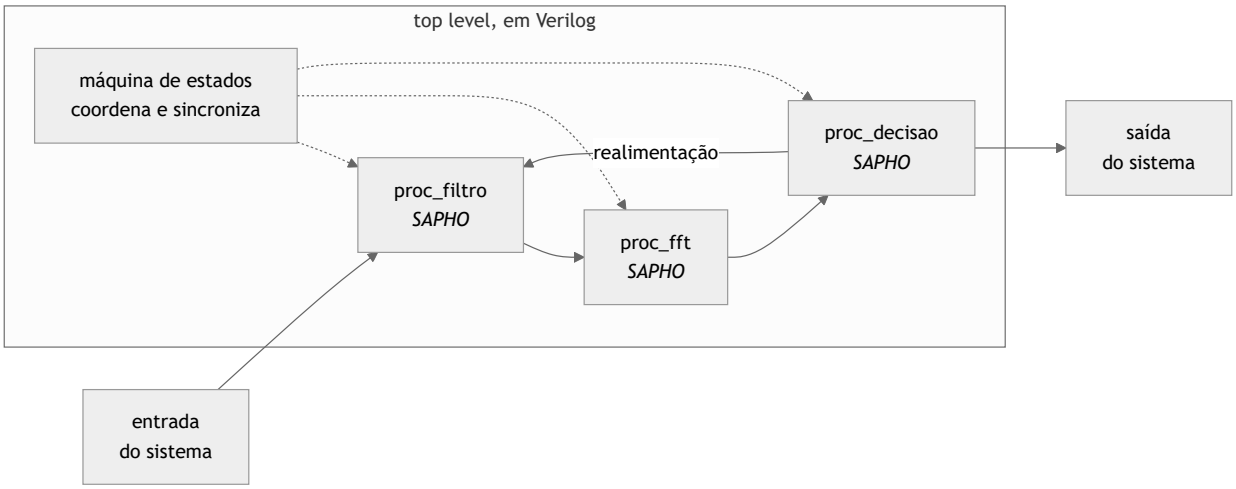
Vale insistir num ponto que muda a forma de pensar o projeto: o Top Level não precisa ser um processador. Ele é o módulo raiz do circuito, e um processador SAPHO gerado pelo YANC é apenas mais um módulo Verilog, com portas de entrada e de saída como qualquer outro. Nada impede que o seu top level instancie três, cinco, dez deles.

A partir daí o encadeamento é livre, e é o que abre a porta para arquiteturas de verdade:

- **Cascata**: a saída de um alimenta a entrada do seguinte, cada estágio fazendo uma parte do processamento. Um filtra, o próximo transforma, o último decide.
- **Paralelo com divisão de dados**: o mesmo programa replicado em vários processadores, cada um cuidando de um canal, com um módulo Verilog distribuindo as amostras e outro juntando os resultados.
- **Realimentação**: a saída de um estágio posterior volta como entrada de um anterior, que é como se escreve um controle em malha fechada ou um algoritmo iterativo, com o próprio circuito repetindo o ciclo até convergir.
- **Hierarquia**: um processador coordenador lê resultados dos outros e decide o que fazer, funcionando como supervisor de um conjunto de processadores especializados.

Cada processador continua sendo do tamanho do seu próprio programa, então um sistema de cinco processadores pequenos pode custar menos área que um único processador genérico que fizesse tudo. Como o circuito inteiro nasce do mesmo projeto, a simulação também é uma só: você vê todos eles na mesma onda, cada um com suas trilhas.

[23] https://www.nipscern.com/library/yanc/multiprocessor.html#links-toma-and-cade
[24] https://www.nipscern.com/library/yanc/multiprocessor.html#fifos-myfifo-and-mydcfifo



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 no repositório do YANC, repartidos em três pastas de `Compilers/CMMComp/Tests`: os `.cmm` moram em `ZeroCross` e `ProcDTW`, e a pasta `DTW` traz 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)[25] e o simulador de pulsos do TileCal com SAPHO (SBAI, 2025)[26], 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.

[25] https://cdn.nipscern.com/publications/cba-2020-arquitetura-multi-core.pdf
[26] 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.

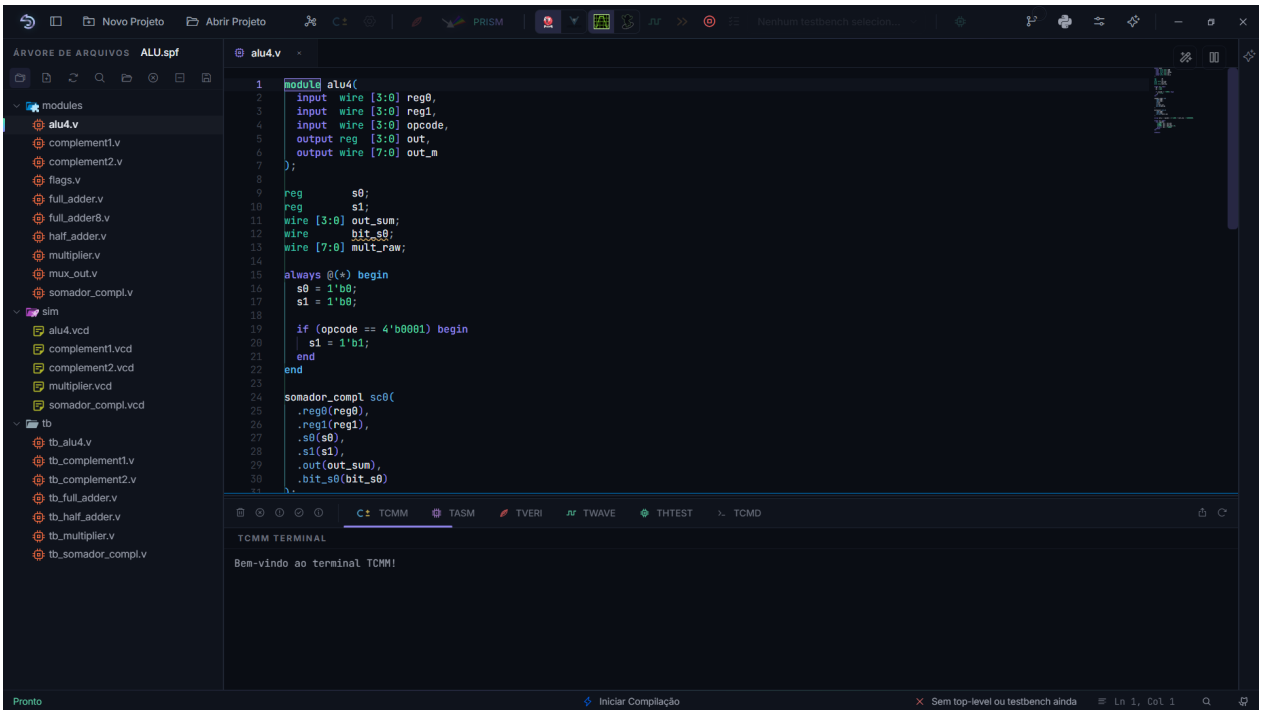
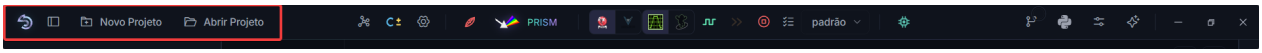


Figura21.1: A janela com um projeto aberto. Esta captura serve de mapa para o restante do capítulo.

21.1 A barra de ferramentas

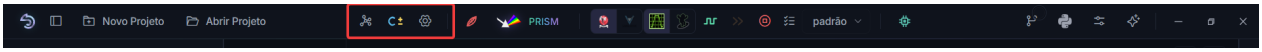
Os botões se agrupam por assunto, da esquerda para a direita. Um botão cinza está desabilitado porque falta um pré-requisito; o tooltip diz qual.

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.

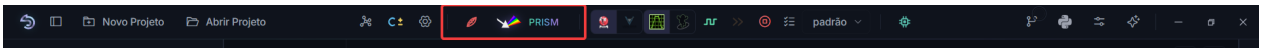
21.1.2 Processador



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

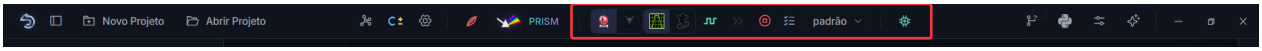
processador ativo: frequência de clock, número de ciclos e exibição de arrays na onda.

21.1.3 Síntese



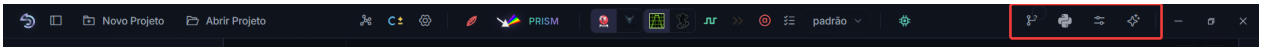
Sintetizar Verilog valida o projeto: elabora todos os fontes a partir do Top Level e gera a hierarquia de módulos. *Abrir PRISM* faz o mesmo e abre o diagrama do circuito. Os dois exigem um Top Level definido.

21.1.4 Ondas



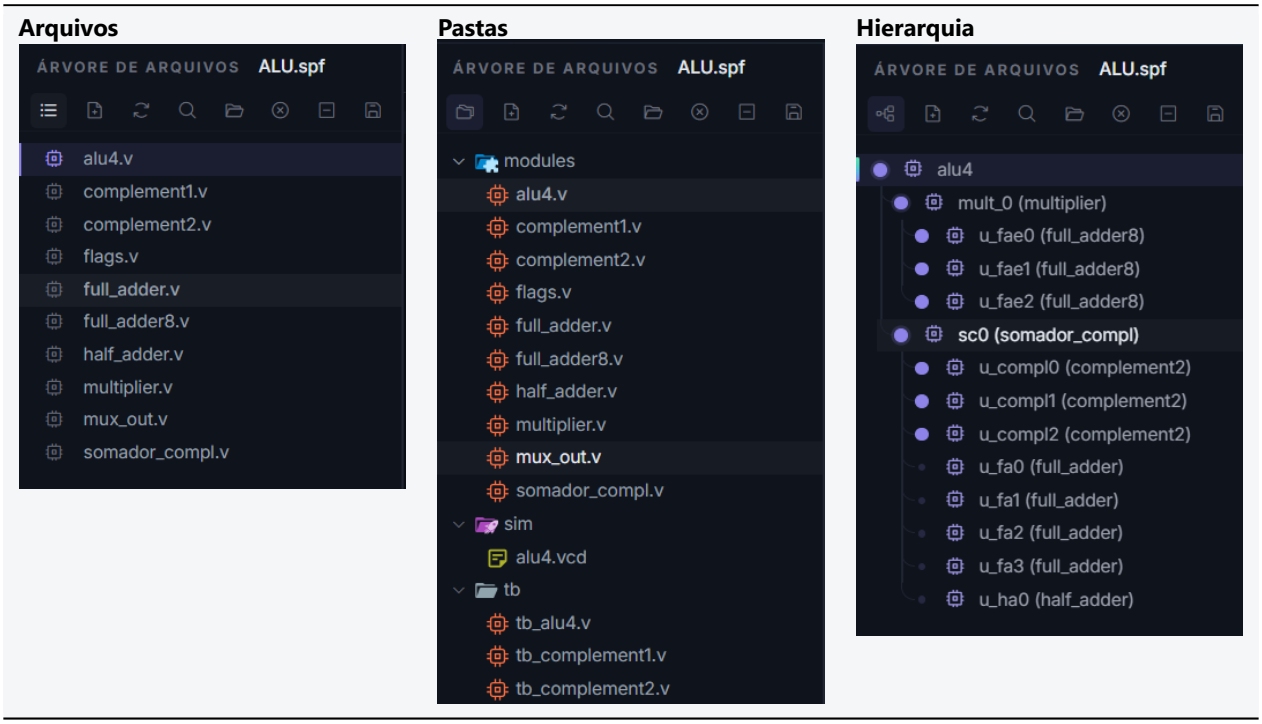
Da esquerda para a direita: as chaves de simulador (Icarus ou Verilator) e de visualizador (GTKWave ou Surfer); *Analisar Verilog*, que simula e abre a onda; *Execução rápida*, sem onda; *Cancelar*; a *Configuração de ondas*; o seletor de layout; e o *Teste do processador sintetizado*. As escolhas deste 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. A seleção é múltipla: `Ctrl` soma e tira, `Shift` pega o intervalo, `Ctrl+A` marca o que está visível. Renomear ou mover um arquivo referenciado pelo projeto atualiza o `.spf` no mesmo gesto, e renomear um arquivo aberto devolve o cursor e a rolagem ao mesmo ponto.

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.

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).

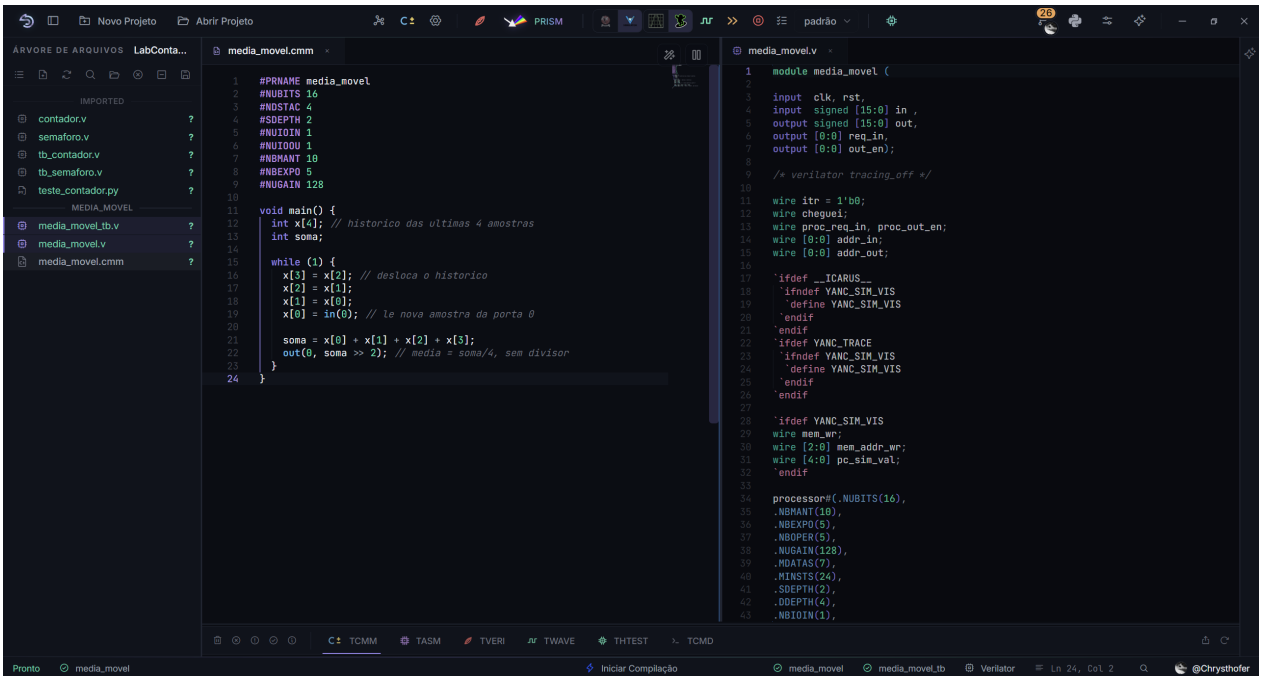
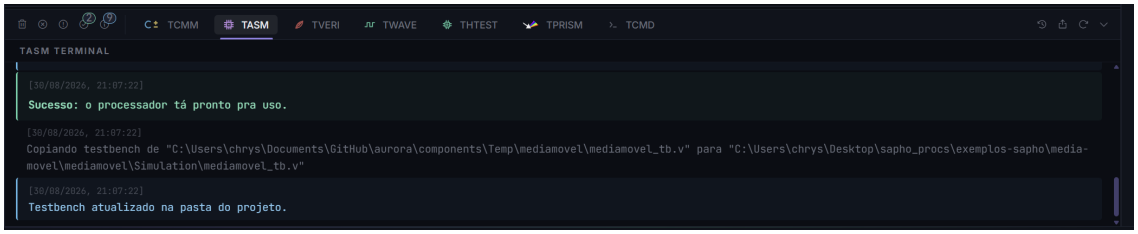


Figura21.2: O uso mais proveitoso da divisão: o `.cmm` de um lado e o `.v` que o YANC gerou dele do outro, com a instância do `processor` e seus parâmetros à vista.

21.4 Os terminais



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

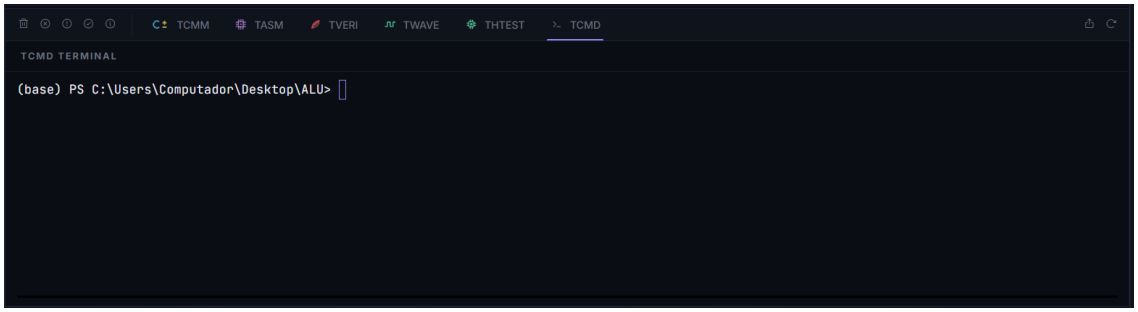
As mensagens chegam classificadas (erro, aviso, sucesso, dica) e podem ser filtradas pelos botões com contadores. Toda referência de linha que uma ferramenta imprime vira link, no formato de cada uma: o caminho com linha do Icarus, o caminho com linha e coluna do Verilator, o `Erro na linha N` do compilador C± e o traceback do cocotb; o clique abre o arquivo naquele ponto. O botão de limpar vale para o terminal aberto ou, no modo todos, para todos de uma vez; o de exportar salva o conteúdo de todos os terminais em um `.txt`; e o do relógio abre o *Histórico de compilações*, descrito em *Compilação e artefatos*.

Três gentilezas evitam a rolagem infinita: qualquer contador que sobe — do simulador, do cocotb ou de um `$display` seu no testbench — vira uma barra de progresso em vez de uma linha por atualização; uma linha repetida em sequência vira um contador `xN` na própria linha; e, durante uma simulação, um marcador no TWAVE mostra o arquivo de onda crescendo ao vivo e congela no tamanho final quando ela termina (o mouse em cima revela o caminho do arquivo). Quando uma mensagem cita um componente que ainda não foi baixado, a linha ganha o botão *Abrir Componentes*.

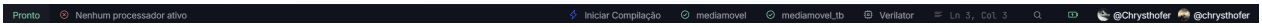
O TCMD não é um console de mentira nem um espelho de saída: é o seu próprio shell, um PowerShell de verdade rodando na máquina, com as suas variáveis de ambiente, o seu histórico e as suas permissões. Vale ali tudo o que valeria na janela do sistema: `git`, `pip`, um script seu, navegar até outra pasta. A AURORA só acrescenta o contexto do projeto ao prompt (processador ativo, diretório, branch) e alguns comandos próprios.

Por ora, dois deles: `apython` chama o Python embarcado da AURORA, e `Use-Python aurora` faz `python` significar

o embarcado naquela sessão. O TCMD merece um capítulo próprio de dicas em *Controle de versão, Python, componentes e configurações*.



21.5 A barra de status



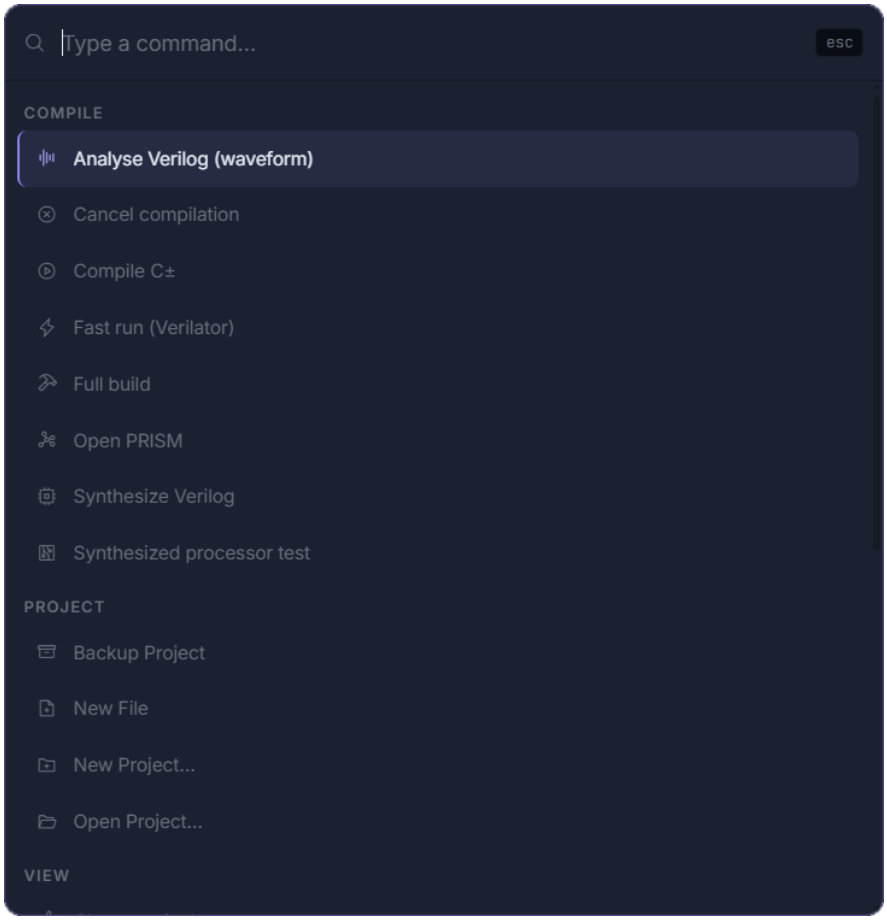
Da esquerda para a direita: *Pronto* ou *Não Pronto* (clicável quando não há projeto: abre o diálogo de abrir); o processador ativo; o andamento da compilação; o Top Level; o Testbench Top; o motor de simulação escolhido; a posição do cursor com o controle de zoom; e as fichas das contas GitHub e GitLab, apagadas quando desconectadas, que abrem o painel de controle de versão. Num laptop entra ainda um indicador de energia: verde na tomada, vermelho na bateria — na bateria o Windows reduz o clock da CPU e a simulação demora mais, e o clique no indicador explica e leva às configurações de energia do Windows.

21.6 Paleta de comandos e busca

Um detalhe que economiza tempo: quase todo modal traz um ponto de interrogação ao lado do X, que abre o capítulo deste manual sobre aquela tela, sem sair da AURORA (*Controle de versão, Python, componentes e configurações*).

`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*.



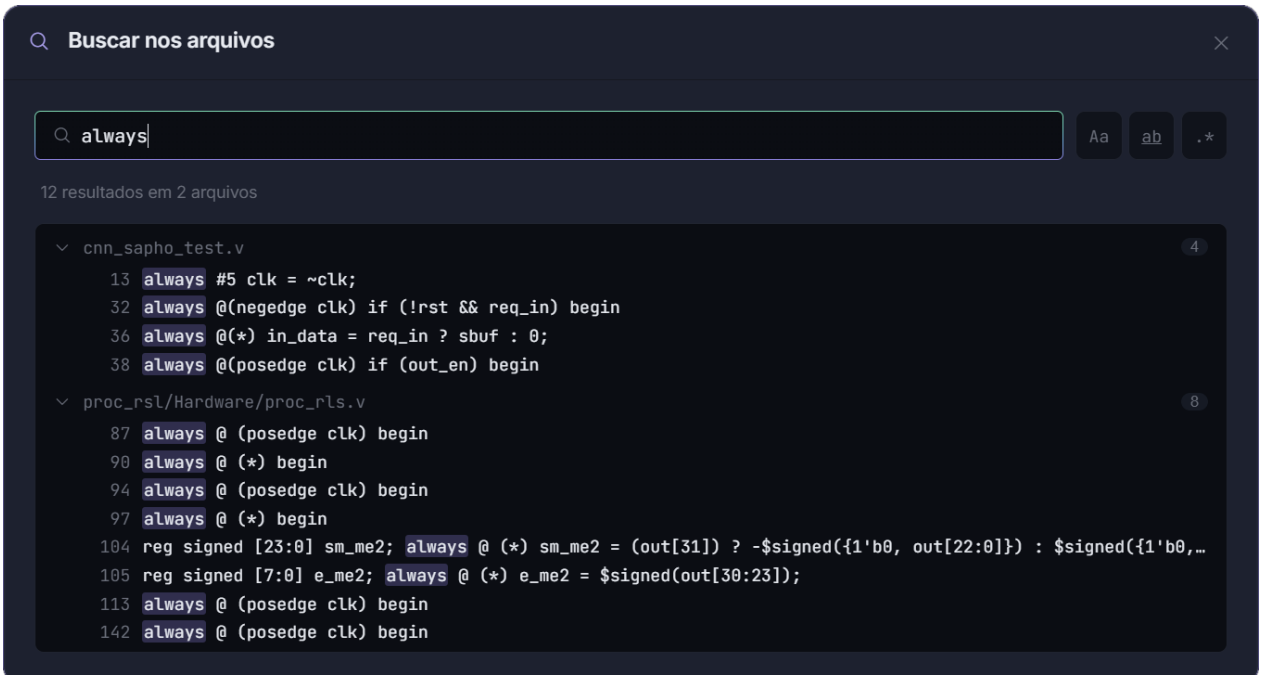


Figura21.3: A busca varre o projeto inteiro e agrupa por arquivo, com a contagem de ocorrências de cada um. O clique numa linha abre o arquivo naquele ponto.

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: o que você escreve
│   ├── media_movel.cmm         o programa (ou media_movel.cpp)
│   ├── input_0.txt             estímulo da porta de entrada 0
│   └── media_movel/            a pasta de saída: tudo o que o YANC gera
│       ├── media_movel.asm
│       ├── media_movel.v
│       ├── media_movel_inst.mif
│       ├── media_movel_data.mif
│       ├── media_movel_tb.v
│       └── output_0.txt        o que a simulação do processador escreveu
├── testbench/                  estado de ondas por testbench
├── .aurora/                    registros da AURORA (histórico de execuções)
└── Backup/                     zips gerados pelo botão de backup
```

O projeto nasce só com o `.spf`. A pasta de um processador aparece quando você o cria no Hub, só com o fonte dentro; a pasta de saída, com o mesmo nome do processador, o YANC cria na primeira compilação. As demais aparecem conforme o uso.

Cada processador tem, então, duas pastas:

- `media_movel/` guarda o que **você escreve**: o fonte, os arquivos que inicializam vetores e os `input_N.txt` (e os `cade_k.txt`, se o processador usar links; veja *C± em uma página*);
- `media_movel/media_movel/` guarda o que **o YANC gera**: o assembly, o Verilog, as memórias, o testbench, as tabelas da forma de onda e as saídas da simulação. Não escreva nada lá dentro: a próxima compilação sobrescreve.

O nome do processador é o nome do arquivo fonte, e vira o nome do módulo Verilog. Por isso ele só aceita letras sem acento, dígitos e _, não pode começar com dígito nem com __, e não pode ser uma palavra reservada do Verilog (`wire`, `module`, `begin`...). A AURORA confere isso ao criar e ao renomear. Para renomear um processador, use a opção da árvore ou do Hub: ela troca a pasta, o fonte e a pasta de saída juntos.

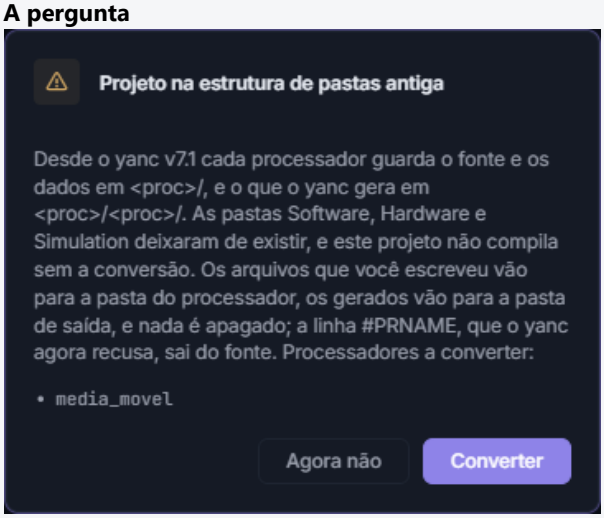
22.2 Projetos de versões antigas

Antes do YANC 7.1, cada processador tinha três subpastas, `Software/`, `Hardware/` e `Simulation/`, e o nome do módulo vinha da diretiva `#PRNAME`. O compilador novo não aceita mais nenhuma das duas coisas. Ao abrir um projeto assim, a AURORA pergunta, uma vez, se pode convertê-lo:

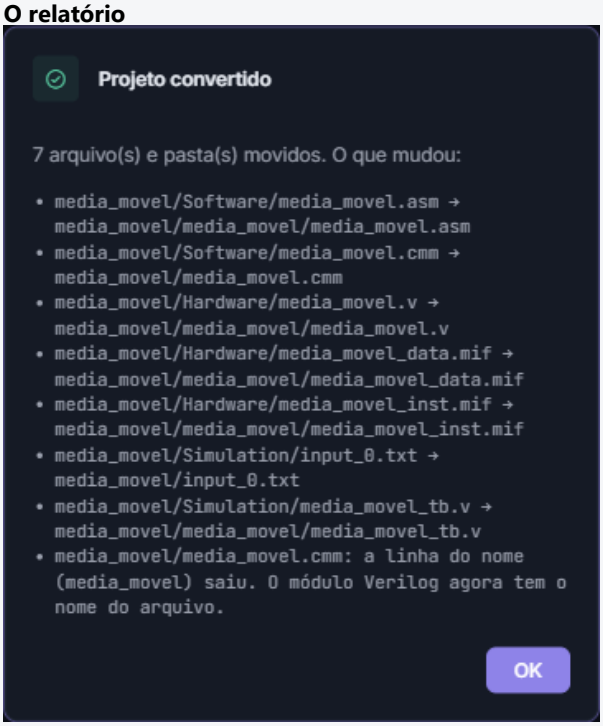
- o que você escreveu (o fonte, os arquivos de vetor, os `input_N.txt`, um `.v` seu que estivesse em `Hardware/`) vai para a pasta do processador;
- o que o YANC tinha gerado vai para a pasta de saída;
- nada é apagado e nada é sobrescrito: se o destino já existe, o arquivo fica onde estava e aparece no relatório;
- a linha `#PRNAME` sai do fonte, e o `.spf` passa a apontar para os lugares novos.

No fim, um relatório lista o que foi movido e o que ficou. Se o `#PRNAME` antigo dava ao módulo um nome diferente do nome da pasta, o relatório oferece renomear o processador para esse nome, para o seu top-level continuar achando o módulo. O mesmo vale para um nome que não serve mais (com hífen ou acento). Quem responder *Agora não* pode converter depois, abrindo o projeto de novo; até lá ele não compila.

A pergunta



O relatório



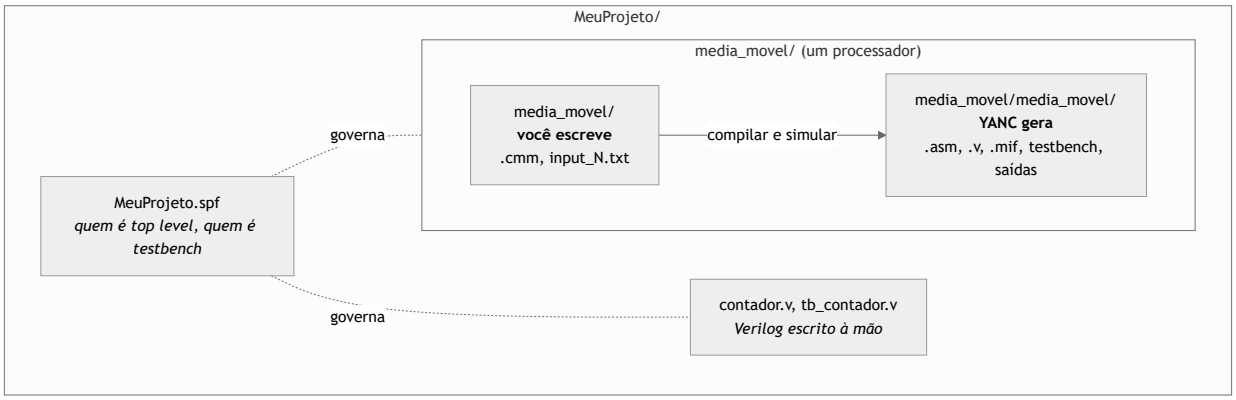
22.3 Projetos de exemplo

Quem prefere partir de algo pronto tem o botão *Projetos de exemplo…* na tela de boas-vindas: ele pergunta só onde salvar e cria **cinco projetos completos**, cada um com o seu `.spf` e o testbench já registrado, todos dentro de uma pasta `exemplos-sapho`. Os cinco entram na lista de recentes, do mais simples para o mais completo:

1. **Contador em Verilog** — Verilog puro, sem processador; o caminho mais curto para ver a AURORA elaborar e simular.
2. **Média móvel** — o processador do *tutorial*, em aritmética inteira.
3. **Trigonometria** — ponto flutuante: seno, cosseno e arco-tangente.
4. **Notação de Dirac** — a álgebra linear escrita como na física, uma operação por linha.
5. **Cruzamento por zero** — duas entradas, cinco saídas, filtros IIR e o endereçamento de porta.

Os fontes viajam dentro do aplicativo, então o botão funciona sem internet. Instalar de novo na mesma pasta não sobrescreve nada: o que já existe é respeitado. A Aurora Intelligence conhece os cinco e também sabe criá-los, a pedido, pelo chat.

Em desenho, o caminho de um arquivo até o hardware:



Aviso

Escolha bem o caminho da pasta. As ferramentas de linha de comando que rodam por baixo (compiladores, Icarus, Verilator, Yosys) recebem esse caminho como argumento, e várias delas engasgam com acentos, espaços duplos, cedilha, `#`, `&`, `%` ou parênteses. O sintoma é ruim de diagnosticar: um erro estranho de arquivo não encontrado, vindo de uma etapa que não tem nada a ver com o seu código.

Use letras sem acento, números, hífen e sublinhado. `C:\Projetos\meu_filtro` está bom; `C:\Meus Projetos (2026)\Simulação #1` não. Vale para o nome do projeto e para toda a árvore de pastas acima dele, inclusive o nome de usuário do Windows. O compilador e os simuladores já aceitam acento no caminho; o espaço continua sendo o que mais atrapalha, porque o Verilator não aceita espaço no caminho do projeto. O nome do processador tem regra mais estrita, descrita acima.

22.4 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` (ou nos `#pragma yanc` do `.cpp`), 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).

22.5 Sintetizável ou testbench: quem decide é você

Todo arquivo Verilog do projeto tem um papel, guardado no `.spf`: **sintetizável** (entra na elaboração e no PRISM), **testbench**, ou **sem papel**. Um arquivo novo ou importado entra sem papel, com um ponto de interrogação no ícone, e fica fora de toda compilação até você escolher, pelo botão direito do mouse. Antes de compilar, o terminal lista os arquivos que ficaram de fora por não ter papel. Arquivos `.py` entram como testbenches cocotb, e o que o YANC gera já nasce com o papel certo: o `<proc>.v` é sintetizável e o `<proc>_tb.v` é testbench. Nada muda de papel sozinho.

Além do papel, há o papel de raiz, que também é seu:

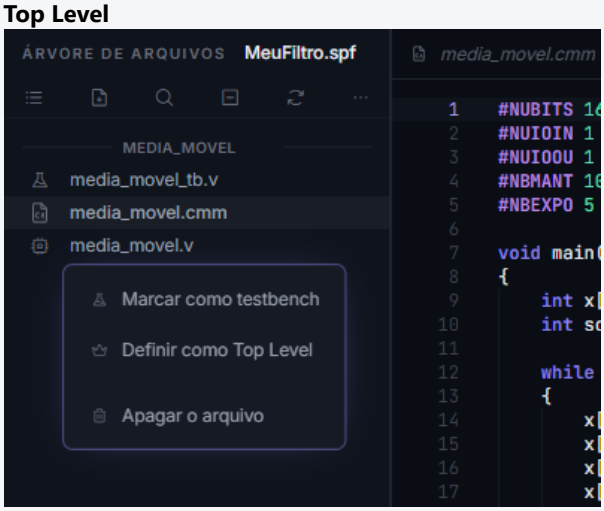


Figura22.1: O menu de contexto de um `.v` ainda sem papel, na visão Arquivos: as duas escolhas de papel.

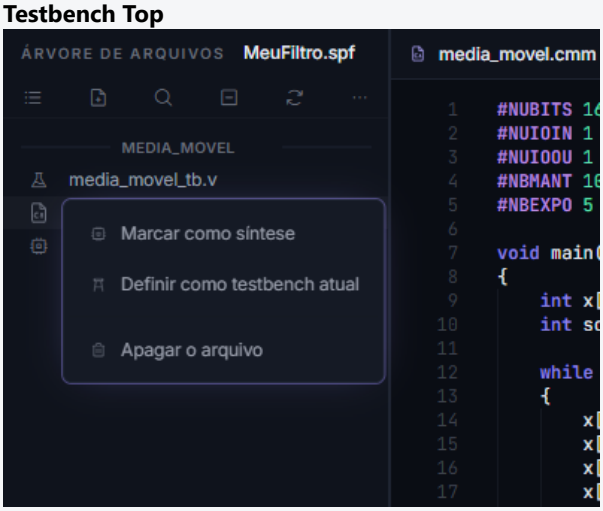
Top Level
: O módulo raiz do circuito sintetizável. Define de onde a elaboração parte e o que o PRISM desenha. Marque pelo menu de contexto do arquivo na visão Arquivos: *Definir como Top Level*.

Testbench Top
: O arquivo que comanda a simulação, `.v` ou `.py`. Define o que roda quando você clica em *Analisar Verilog*. Marque por *Definir como testbench atual*, no menu de um arquivo que já é testbench.

Top Level



Testbench Top



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

22.6 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.

22.7 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, componentes e configurações*.

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

Capítulo 23

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 Como ela age na IDE

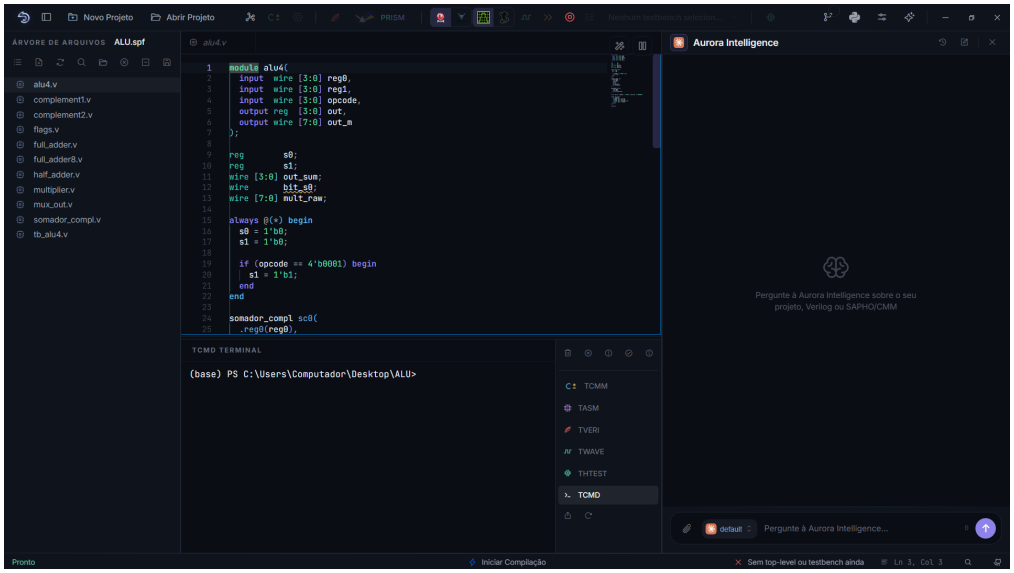
O que separa a Aurora Intelligence de um chat comum é o acesso: ela conversa com as mesmas funções internas que os seus botões chamam. A AURORA expõe à assistente um conjunto de ferramentas, e o modelo escolhe qual usar a cada passo.

São as operações do dia a dia da IDE: listar e ler os arquivos do projeto, escrever ou alterar um arquivo, abrir um arquivo no editor, disparar a compilação C± ou a análise Verilog, perguntar se uma execução ainda está rodando, ler o que saiu em cada terminal, consultar a configuração do processador ativo, rodar um comando no TCMD, abrir o PRISM ou um visualizador de ondas, criar um layout de ondas sob encomenda, conhecer e instalar os *projetos de exemplo*, e operar a simulação do PRISM.

Ela também pesquisa e lê **este manual**: perguntas sobre a plataforma são respondidas citando as páginas certas, em vez de de memória. E sabe o que a máquina tem — recebe a lista de componentes ainda não baixados, e não gasta o seu tempo propondo uma ferramenta que não vai rodar.

O ciclo é sempre o mesmo: você descreve o objetivo, o modelo pede uma ferramenta, a AURORA executa e devolve o resultado, e o modelo decide o passo seguinte com esse resultado em mãos. Por isso um pedido como "compile e me diga por que o TASM reclamou" funciona: ela compila de verdade, lê o terminal de verdade e responde sobre o seu erro, não sobre um erro genérico.

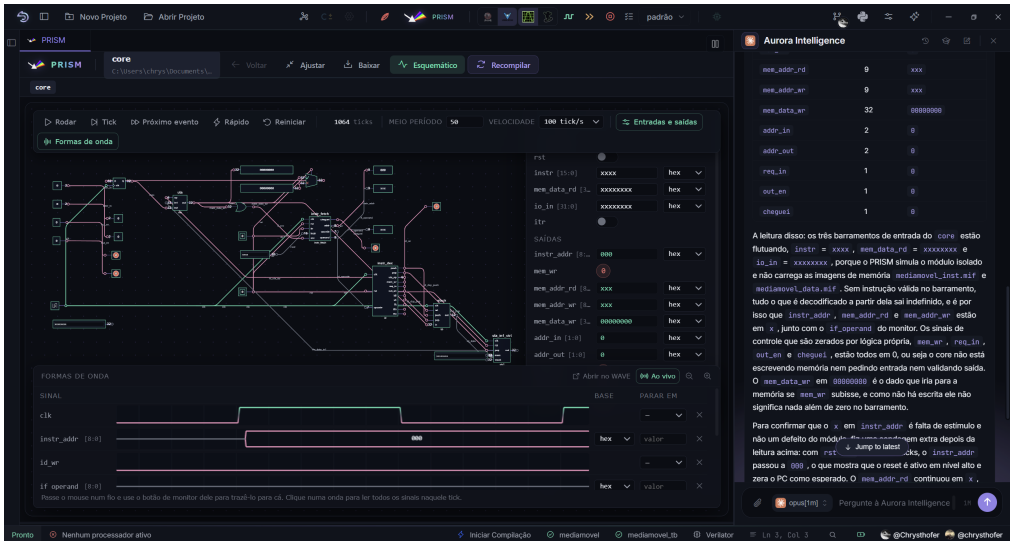
Nada disso acontece pelas suas costas. Toda ferramenta que modifica o projeto ou executa algo aparece antes como um cartão de confirmação, com *Permitir* ou *Negar*, e as de leitura podem ser liberadas de vez nas configurações. O modelo nunca alcança nada fora do projeto aberto: os caminhos são validados no processo principal da AURORA antes de qualquer operação de disco.



23.2 A simulação que ela consegue ler

O GTKWave e o Surfer são visualizadores, e a assistente não enxerga dentro deles: a onda de uma simulação comum é uma figura para você olhar, não um dado que ela possa consultar. A exceção é o modo *Simular* do PRISM (*PRISM: o visualizador de RTL*), que roda dentro da própria AURORA. Por ele a assistente liga o circuito de um módulo, escreve nas entradas, avança um número exato de ticks ou até um sinal chegar a um valor, e lê o que cada porta e cada fio interno valem naquele instante. É a diferença entre responder o que o Verilog parece fazer e responder o que o circuito fez.

Na prática isso permite pedir coisas como verificar se um contador conta mesmo, se o reset zera o registrador, ou qual saída um módulo dá para uma certa entrada, e receber a resposta com os números observados. Ela pode ainda trazer sinais internos para o monitor, parar a simulação quando um deles atingir um valor, entrar num submódulo e, no fim, gravar tudo num `.vcd` e abrir no seu visualizador de ondas. Os limites são os do próprio modo: lógica apenas, sem tempo, e módulos pequenos. Um processador SAPHO inteiro executando o programa continua sendo trabalho do testbench.



23.3 Tutorial guiado

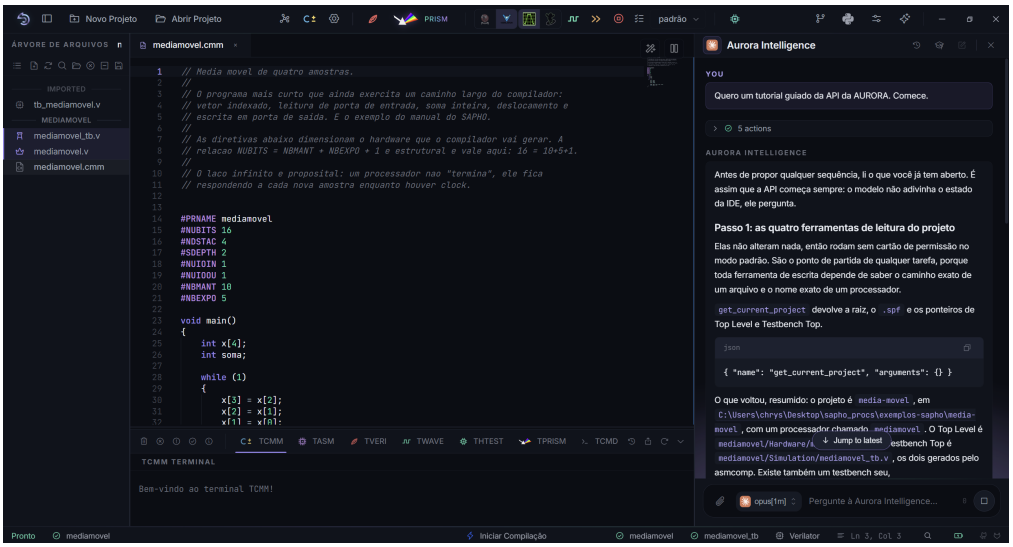
O botão de capelo no cabeçalho do painel inicia um tutorial guiado da API da AURORA: uma conversa nova em que a própria assistente vira instrutora, com o manual instalado por trás, e apresenta o que ela sabe fazer na IDE, com exemplos para você experimentar em seguida. É o jeito mais rápido de descobrir o que dá para pedir.

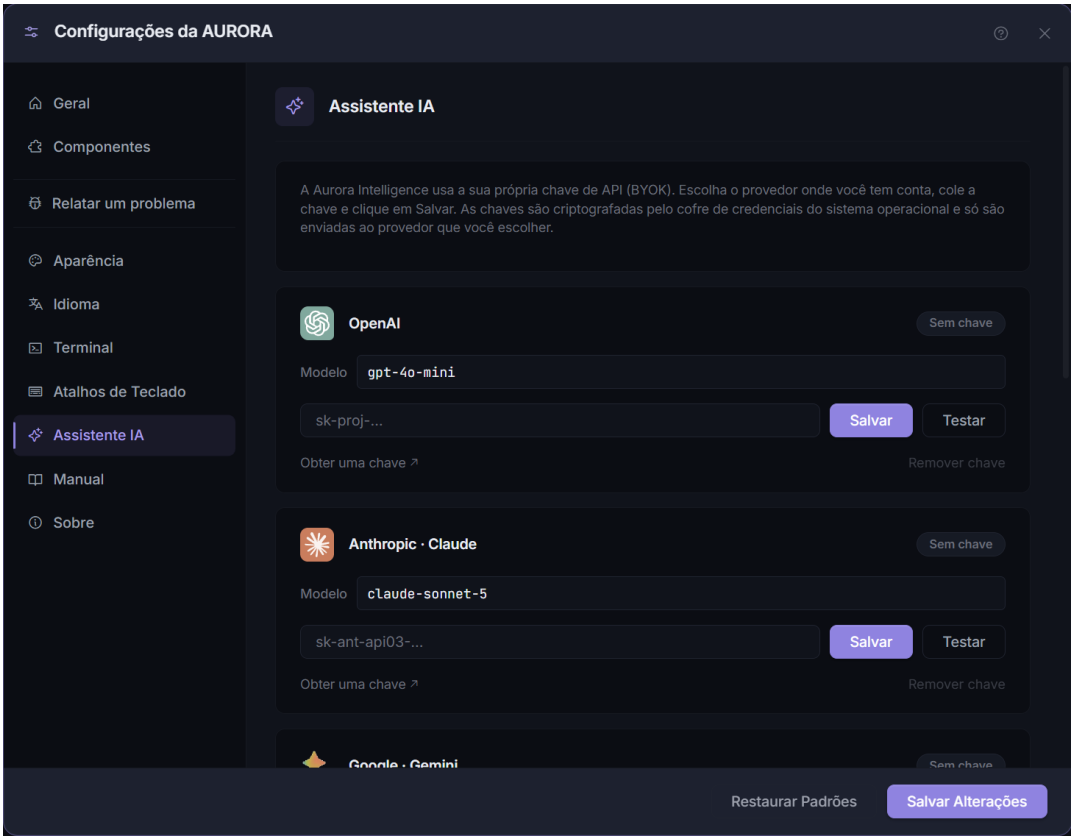
23.4 Configurar

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.

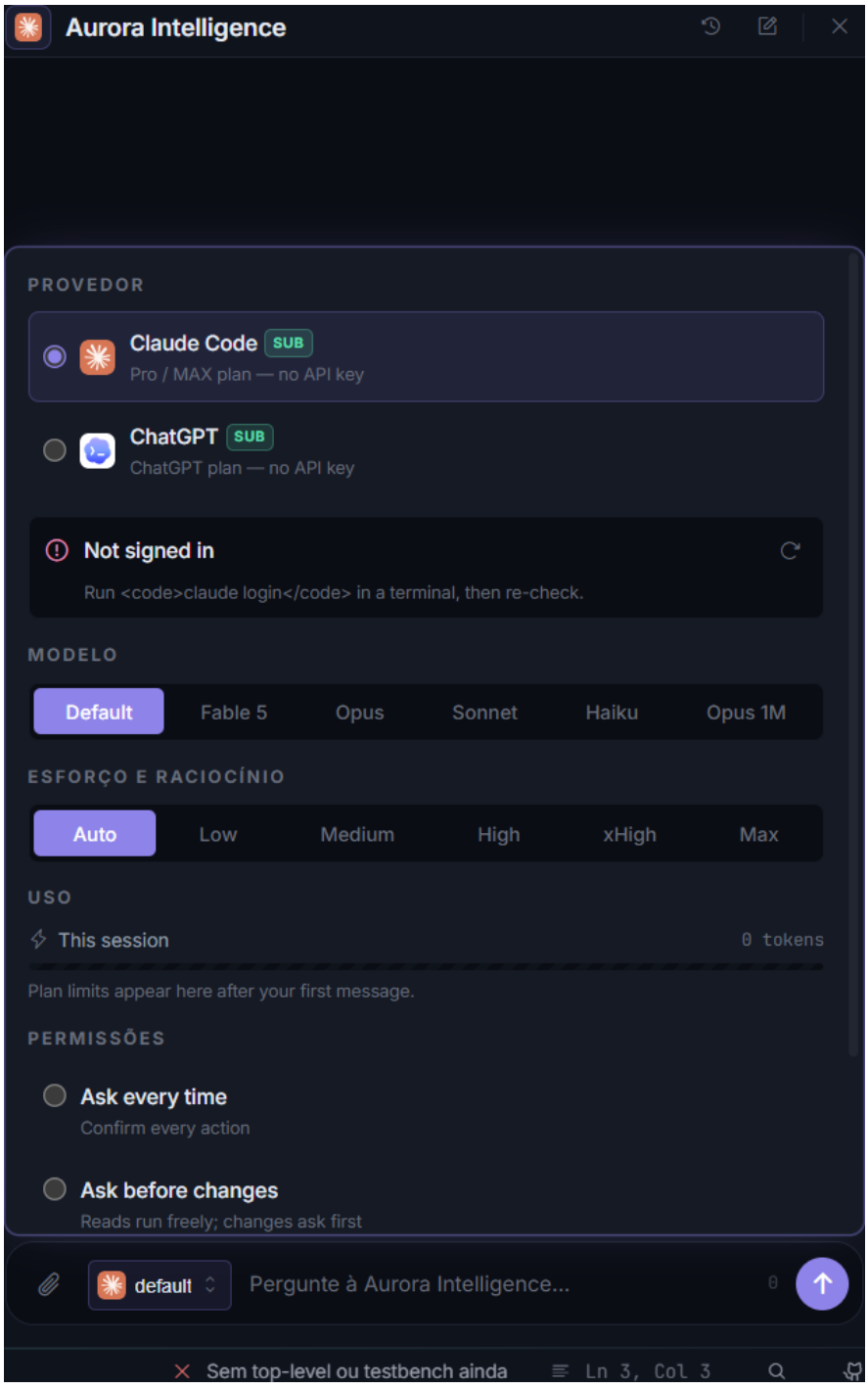
No cartão da Anthropic, o modelo padrão é o Claude Sonnet 5, e a lista traz também o Opus 5, o Fable 5 e o Haiku 4.5, cada um com o preço por milhão de tokens ao lado. O campo *Esforço e raciocínio* controla quanto o modelo pensa antes de responder, de *Low* a *Max*, com *Auto* deixando a escolha com ele. O cache de prompt é aplicado sozinho, o que barateia as conversas longas sobre o mesmo projeto.


Novo Projeto
Abrir Projeto
PRISM
padrão
ÁRVORE DE ARQUIVOS
mediamovel.cmm
IMPORTED
tb_mediamovel.v
MEDIAMOVEL
mediamovel_tb.v
mediamovel.v
mediamovel.cmm
// Media movel de quatro amostras.
// O programa mais curto que ainda exercita um caminho largo do compilador:
// vetor indexado, leitura de porta de entrada, soma inteira, deslocamento e
// escrita em porta de saida. E o exemplo do manual do SAPHO.
// As diretivas abaixo dimensionam o hardware que o compilador vai gerar. A
// relacao NUBITS = NBMANT + NBEXPO + 1 e estrutural e vale aqui: 16 = 10+5+1.
// O laco infinito e proposital: um processador nao "termina", ele fica
// respondendo a cada nova amostra enquanto houver clock.
#PRNAME mediamovel
#NUBITS 16
#NOSTAC 4
#SDEPTH 2
#NUIDIN 1
#NUIODU 1
#NBMANT 10
#NBEXPO 5
void main()
int x[4];
int soma;
while (1)
x[3] = x[2];
x[2] = x[1];
C: TCMM
TASM
TVERI
TWAVE
THTEST
TPRISM
TCMD
TCMM TERMINAL
Bem-vindo ao terminal TCMM!
Aurora Intelligence
YOU
Quero um tutorial guiado da API da AURORA. Comece.
5 actions
AURORA INTELLIGENCE
Antes de propor qualquer sequência, li o que você já tem aberto. É assim que a API começa sempre: o modelo não adivinha o estado da IDE, ele pergunta.
Passo 1: as quatro ferramentas de leitura do projeto
Elas não alteram nada, então rodam sem cartão de permissão no modo padrão. São o ponto de partida de qualquer tarefa, porque toda ferramenta de escrita depende de saber o caminho exato de um arquivo e o nome exato de um processador.
get_current_project devolve a raiz, o .spf e os ponteiros de Top Level e Testbench Top.
json
{ "name": "get_current_project", "arguments": {} }
O que voltou, resumido: o projeto é media-movel , em C:\Users\chrys\Desktop\sapho_procs\exemplos-sapho\media-movel , com um processador chamado mediamovel . O Top Level é mediamovel/Hardware/k
Jump to latest
estbench Top é mediamovel/Simulation/mediamovel_tb.v , os dois gerados pelo asmcomp. Existe também um testbench seu,
opus[1m]
Pergunte à Aurora Intelligence...
Pronto
mediamovel
Iniciar Compilação
mediamovel
mediamovel_tb
Verilator
Ln 3, Col 3


Configurações da AURORA
Geral
Componentes
Relatar um problema
Aparência
Idioma
Terminal
Atalhos de Teclado
Assistente IA
Manual
Sobre
Assistente IA
A Aurora Intelligence usa a sua própria chave de API (BYOK). Escolha o provedor onde você tem conta, cole a chave e clique em Salvar. As chaves são criptografadas pelo cofre de credenciais do sistema operacional e só são enviadas ao provedor que você escolher.
OpenAI
Sem chave
Modelo
gpt-4o-mini
sk-proj-...
Salvar
Testar
Obter uma chave ↗
Remover chave
Anthropic · Claude
Sem chave
Modelo
claude-sonnet-5
sk-ant-api03-...
Salvar
Testar
Obter uma chave ↗
Remover chave
Google · Gemini
Sem chave
Restaurar Padrões
Salvar Alterações

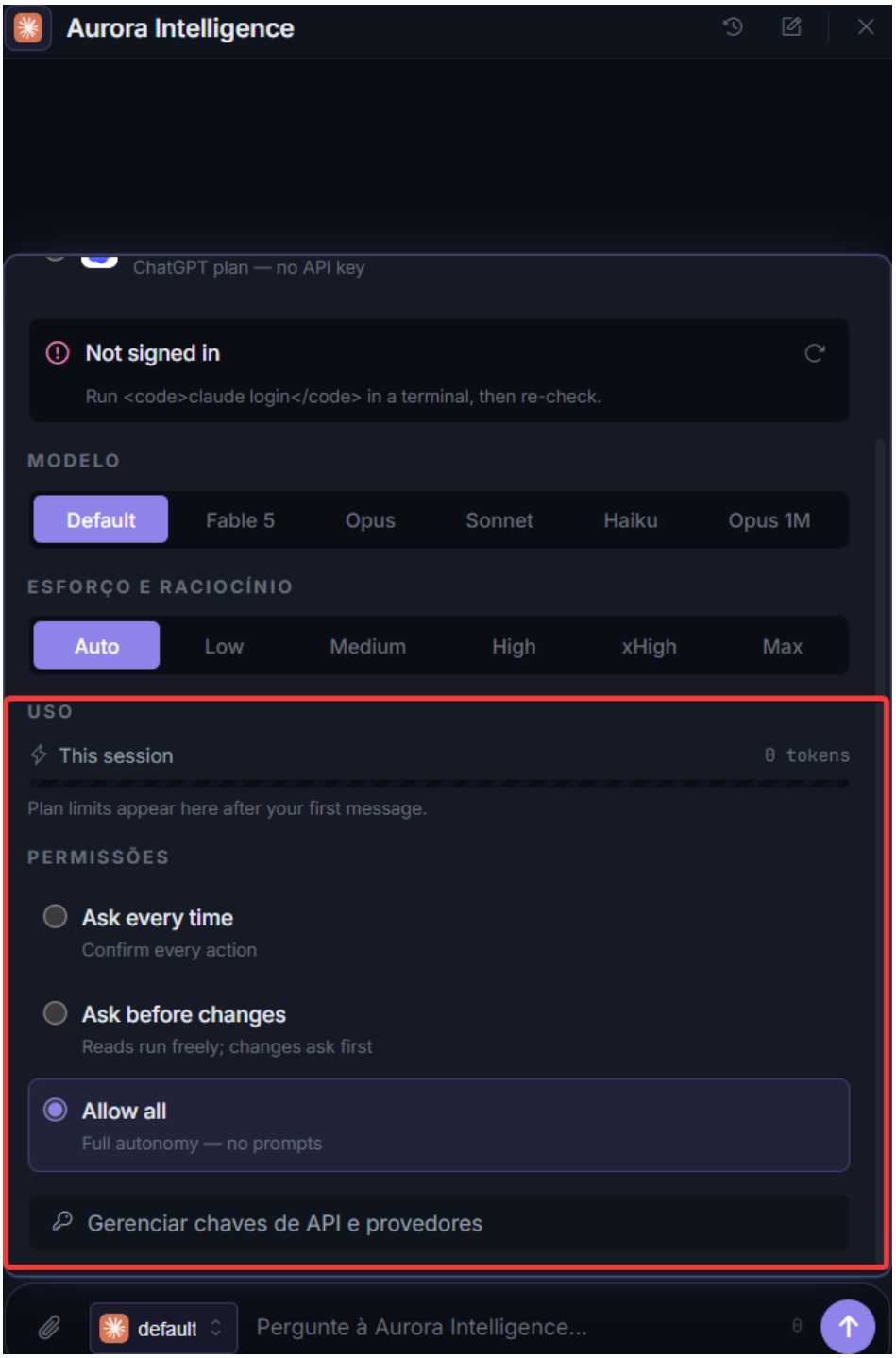
23.5 Usar

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", "simule a ula no PRISM e me diga quanto sai para 12 e 5". Selecionar código no editor faz aparecer uma estrela com atalhos: explicar, procurar defeitos, melhorar, comentar.



23.6 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 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.

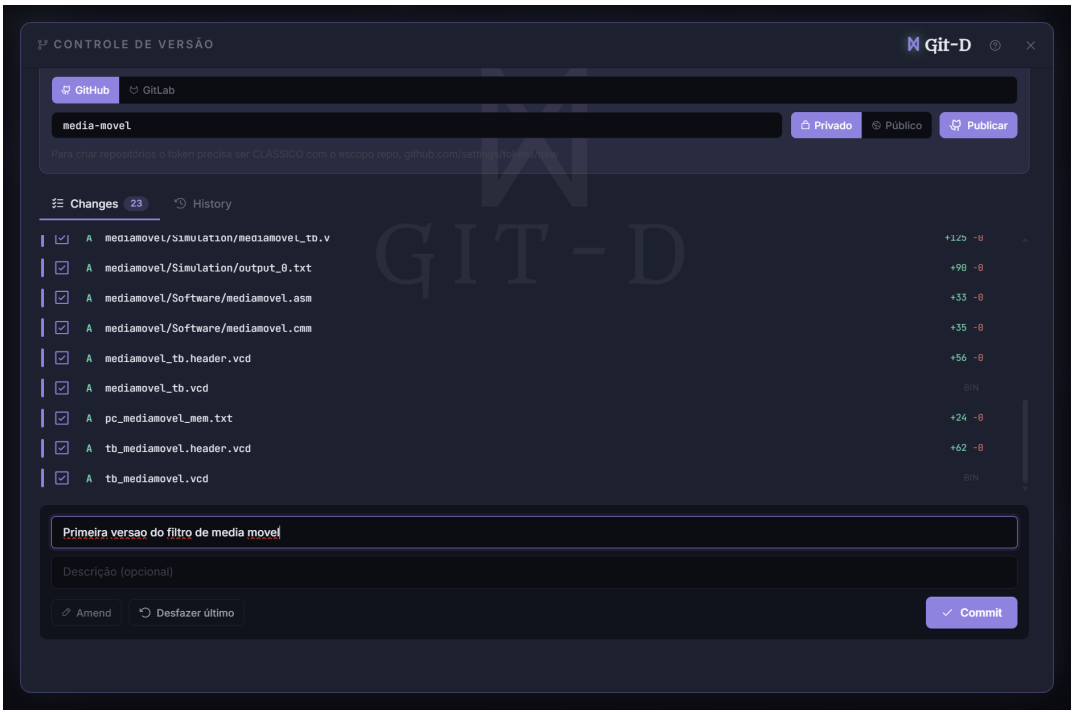
Controle de versão, Python, componentes e configurações

As ferramentas de apoio que completam a IDE.

24.1 Controle de versão (Git-D)

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

> **Nota**
>
> Quem usa a AURORA há mais tempo conheceu este painel como **Dagr**. É a mesma ferramenta: apenas abreviamos o nome para **Git-D**, que diz na hora do que se trata.



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.
- **Contas**: GitHub e GitLab dividem a mesma fileira do painel, cada um com o seu cartão, sem hierarquia entre as forjas. O login de qualquer um usa o fluxo de dispositivo (o botão *Entrar* abre o navegador e

mostra um código para digitar, sem senha na IDE) ou um token pessoal; no GitLab, o token precisa do escopo `api`, e um campo de instância aceita servidores próprios além do `gitlab.com`.

- **Repositórios**: logado, dá para publicar o projeto direto (*Privado* ou *Público*) e clonar os seus repositórios. A lista mistura as duas origens, ordenada por atividade, com um selo dizendo de onde vem cada uma; ao publicar com as duas contas conectadas, um seletor pergunta a forja.
- Um projeto sem repositório oferece *Inicializar* na hora; o menu da árvore cria um `.gitignore` adequado ao SAPHO.
- A barra de status mostra as duas fichas o tempo todo: apagada quando a conta está desconectada, com a foto e o `@usuario` quando conectada.

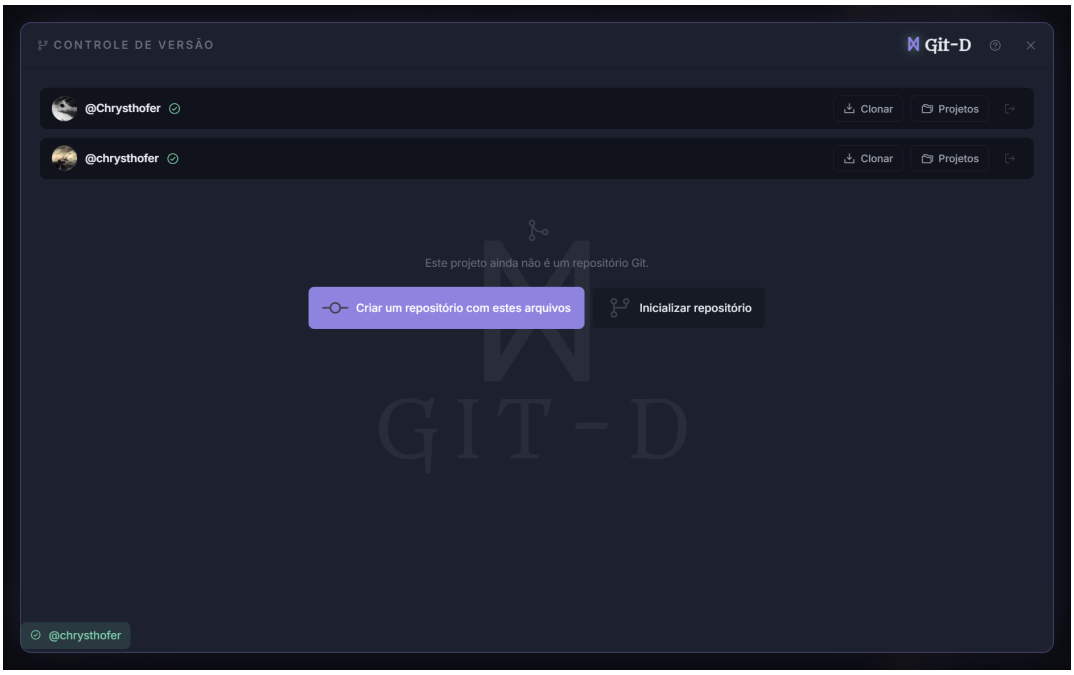


Figura24.1: As duas forjas no mesmo painel. O cartão muda de forma conforme a largura, mas o conteúdo é o mesmo.

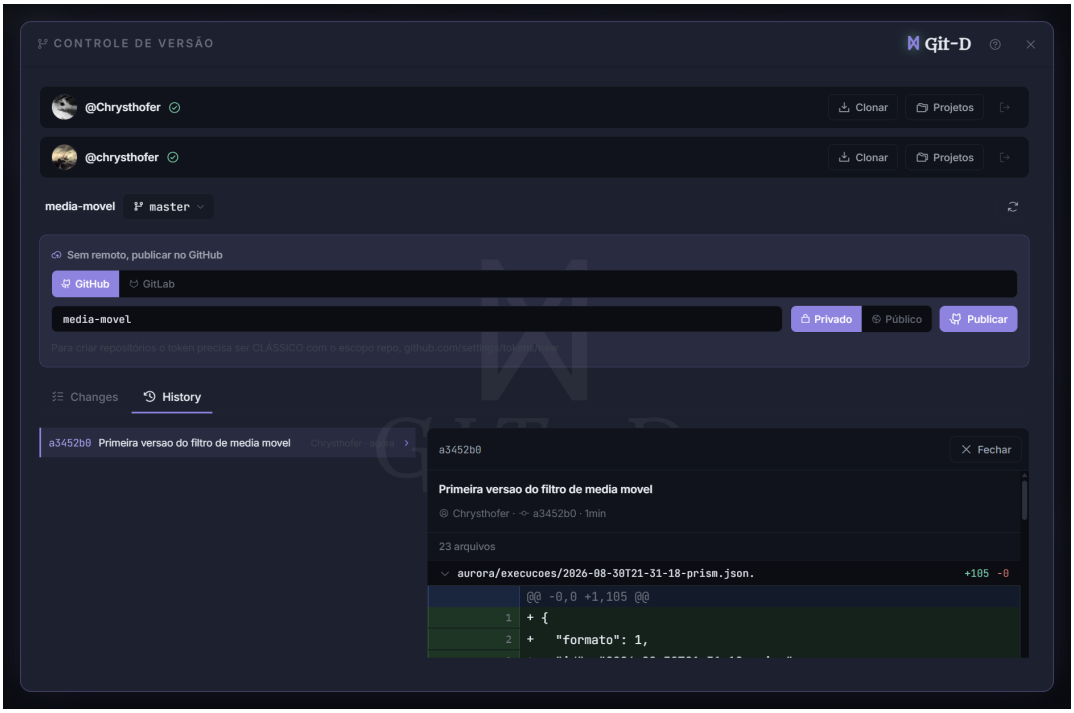
> **Dica**
>
> Em computador compartilhado, repare na opção *Limpar o acesso ao GitHub e ao GitLab ao fechar a AURORA*, na aba *Geral* das Configurações. Ela vem **ligada de fábrica**: ao fechar a AURORA, as duas contas saem do cofre da IDE, do arquivo de credenciais do git e do Gerenciador de Credenciais do Windows, sem depender de alguém lembrar. O botão *Limpar agora*, logo abaixo, faz o mesmo sem esperar o fechamento.

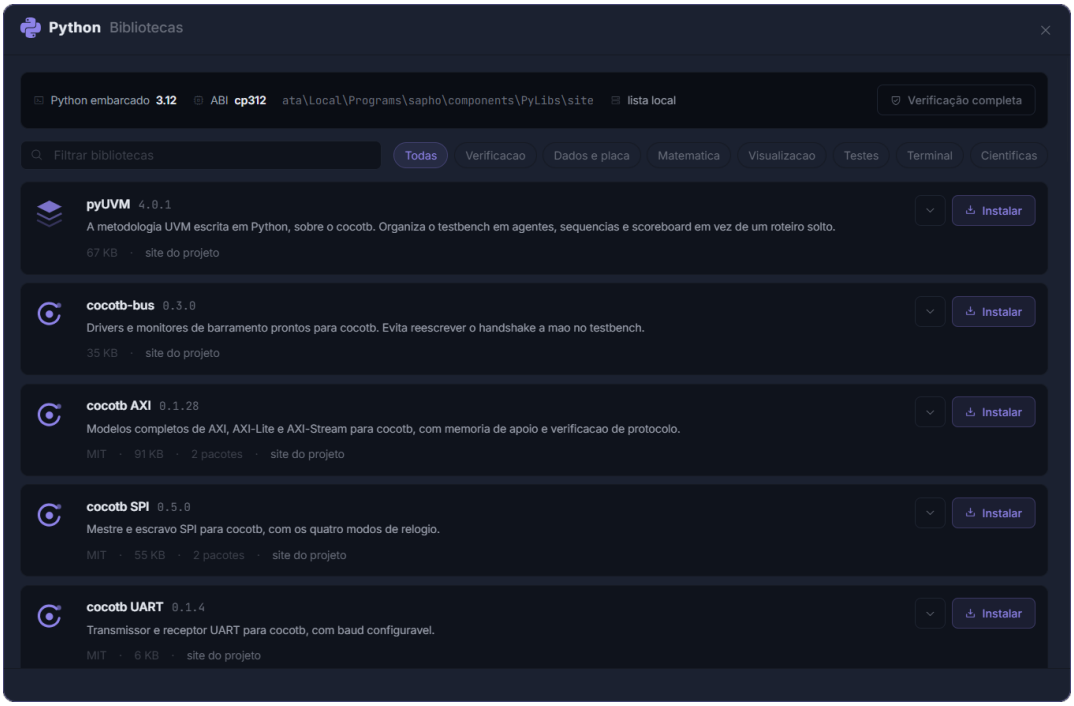
A árvore de arquivos mostra o estado git de cada arquivo (modificado, novo, excluído) por letras e cores, nas duas visões.

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


CONTROLE DE VERSÃO
Git-D
@Chrysthofer
Clonar
Projetos
@chrysthofer
Clonar
Projetos
media-movel
master
Sem remoto, publicar no GitHub
GitHub
GitLab
media-movel
Privado
Público
Publicar
Changes
History
a3452b0 Primeira versao do filtro de media movel
a3452b0
Fechar
Primeira versao do filtro de media movel
Chrysthofer · a3452b0 · 1min
23 arquivos
aurora/execucoes/2026-08-30T21-31-18-prism.json.
+105 -0
@@ -0,0 +1,105 @@
+ {
+ "formato": 1,


Python Bibliotecas
Python embarcado 3.12
ABI cp312
ata\Local\Programs\sapho\components\PyLibs\site
lista local
Verificação completa
Filtrar bibliotecas
Todas
Verificacao
Dados e placa
Matematica
Visualizacao
Testes
Terminal
Cientificas
pyUVM 4.0.1
A metodologia UVM escrita em Python, sobre o cocotb. Organiza o testbench em agentes, sequencias e scoreboard em vez de um roteiro solto.
67 KB · site do projeto
Instalar
cocotb-bus 0.3.0
Drivers e monitores de barramento prontos para cocotb. Evita reescrever o handshake a mao no testbench.
35 KB · site do projeto
Instalar
cocotb AXI 0.1.28
Modelos completos de AXI, AXI-Lite e AXI-Stream para cocotb, com memoria de apoio e verificacao de protocolo.
MIT · 91 KB · 2 pacotes · site do projeto
Instalar
cocotb SPI 0.5.0
Mestre e escravo SPI para cocotb, com os quatro modos de relogio.
MIT · 55 KB · 2 pacotes · site do projeto
Instalar
cocotb UART 0.1.4
Transmissor e receptor UART para cocotb, com baud configuravel.
MIT · 6 KB · site do projeto
Instalar

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 Componentes

O instalador da AURORA traz só o que é o SAPHO em si: a IDE e os compiladores do YANC. O resto — simuladores, visualizadores, formatadores, os agentes de IA — são **componentes**: ferramentas que você baixa quando for usar e pode remover para recuperar espaço. O painel mora em *Configurações*, aba *Componentes*, a segunda da lista.

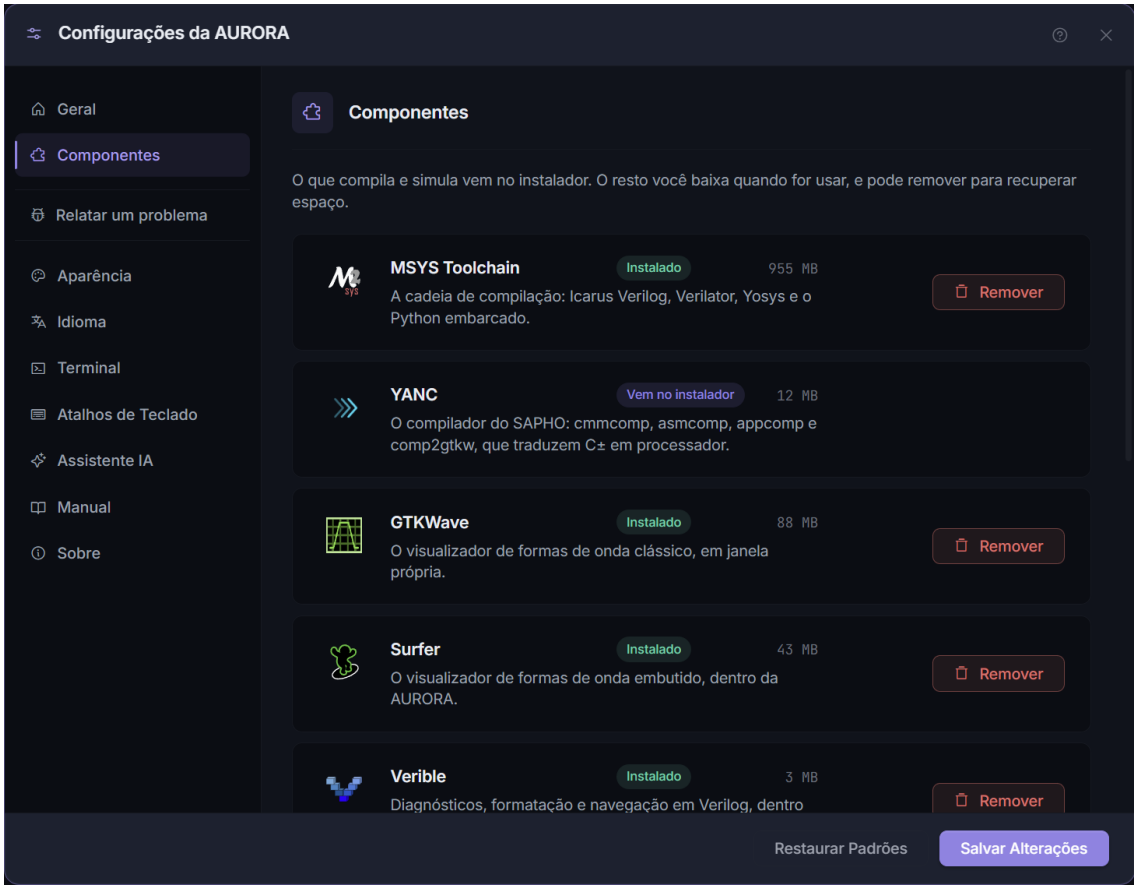


Figura24.2: Cada componente é um cartão com a marca, um selo de estado, o tamanho e a ação. O rodapé resume o que falta baixar.

Componente	O que é	Download
MSYS Toolchain	a cadeia de compilação: Icarus Verilog, Verilator, Yosys e o Python embarcado	272 MB
YANC	os compiladores do SAPHO; vem no instalador e não pode ser removido	—
GTKWave	o visualizador de ondas clássico, em janela própria	30 MB
Surfer	o visualizador de ondas embutido, dentro da AURORA	16 MB
Verible	diagnósticos, formatação e navegação em Verilog, dentro do editor	2 MB
slang	análise semântica de SystemVerilog	3 MB
clang-format	formatação de C, C++ e C± com `Shift+Alt+F`	2 MB
Claude Code	o agente da Anthropic, usado pela Aurora Intelligence no modo Claude Code	90 MB
Codex	o agente da OpenAI, usado pela Aurora Intelligence no modo Codex	132 MB

Como o painel se comporta:

- Cada cartão carrega **um selo**: *Instalado*, *Não instalado*, *Atualização disponível*, *Vem no instalador* (o YANC) ou *Necessário para compilar*, o selo urgente do MSYS e do YANC.
- As ações são *Baixar*, *Atualizar* e *Remover*. Remover mostra o preço antes: o que depende do componente para de funcionar, e tê-lo de volta custa o download de novo. Os seus projetos nunca são tocados.
- Tentar usar uma ferramenta cujo componente falta não dá erro seco: um diálogo explica o que falta e oferece *Baixar agora*, e as linhas de terminal que citam um componente ausente ganham um botão *Abrir Componentes*.
- O botão *Verificar e consertar* do rodapé — a maleta de primeiros socorros — limpa os caches, confere os arquivos de cada componente e baixa de novo o que estiver incompleto ou quebrado. Componente saudável não é tocado.
- *Abrir a pasta* leva ao lar dos componentes, `%LOCALAPPDATA%\SAPHO\components`. A atualização da AURORA preserva essa pasta; desinstalar o SAPHO a apaga.

24.4 Configurações

Configurações abre as preferências, em nove abas:

Aba	O que tem
Geral	tooltips da interface; confiar em links externos do chat da IA; onde o PRISM e o Surfer abrem, janela própria ou aba do editor; limpar o acesso ao GitHub e ao GitLab ao fechar (ligado por padrão) e *Limpar agora*; avisar quando a internet cair
Componentes	baixar, atualizar, consertar e remover as ferramentas; a seção acima
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
Manual	o manual do SAPHO neste computador; a seção abaixo
Sobre	versão, situação do atualizador, equipe, licenças

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

24.5 Este manual, dentro da AURORA

O manual tem aba própria nas Configurações: *Manual*. Um cartão de estado diz se a cópia offline está *Neste computador*, com a versão dela, e três botões fazem o resto: *Abrir na AURORA* abre a cópia local, sem internet; *Abrir no navegador* abre este site; *Procurar atualização* busca uma versão nova na hora. A cópia offline também se atualiza sozinha quando saem correções, sem esperar uma versão nova do aplicativo.

24.5.1 O ponto de interrogação dos modais

Você quase nunca vai precisar abrir o manual pelo começo. Os modais que exigem alguma decisão trazem um ponto de interrogação ao lado do X, com a dica dizendo qual assunto ele abre: são o Hub de Processadores, o Novo Projeto, a Configuração de ondas, o Histórico de compilações, a busca nos arquivos, as Bibliotecas Python, o Controle de Versão e as próprias Configurações.

O clique não abre um índice para você procurar: abre o capítulo que trata daquela tela, direto. O manual

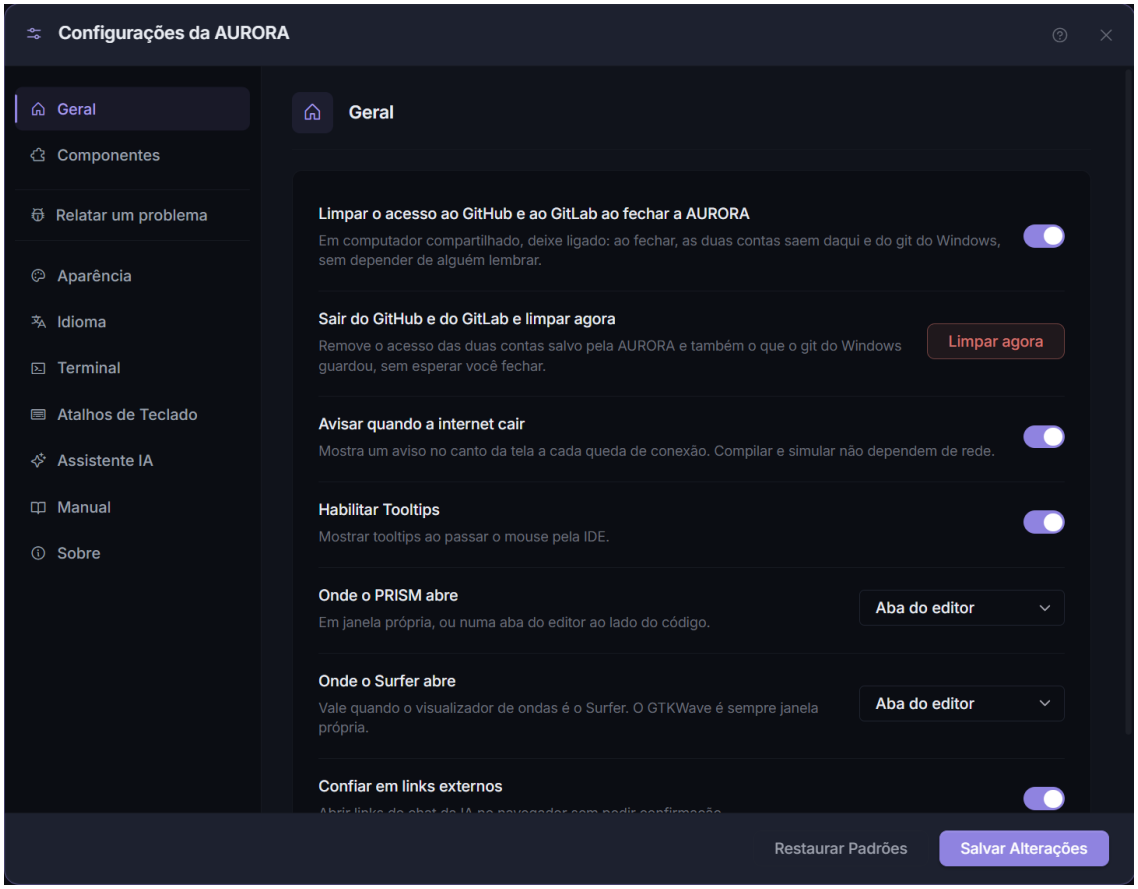
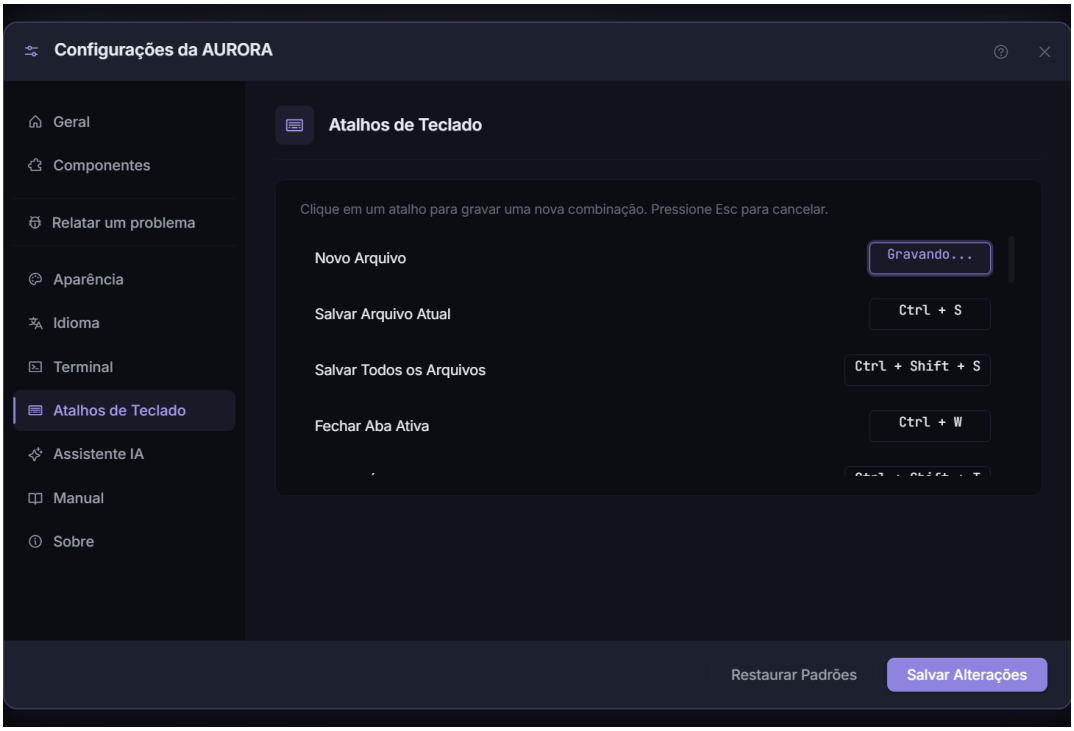
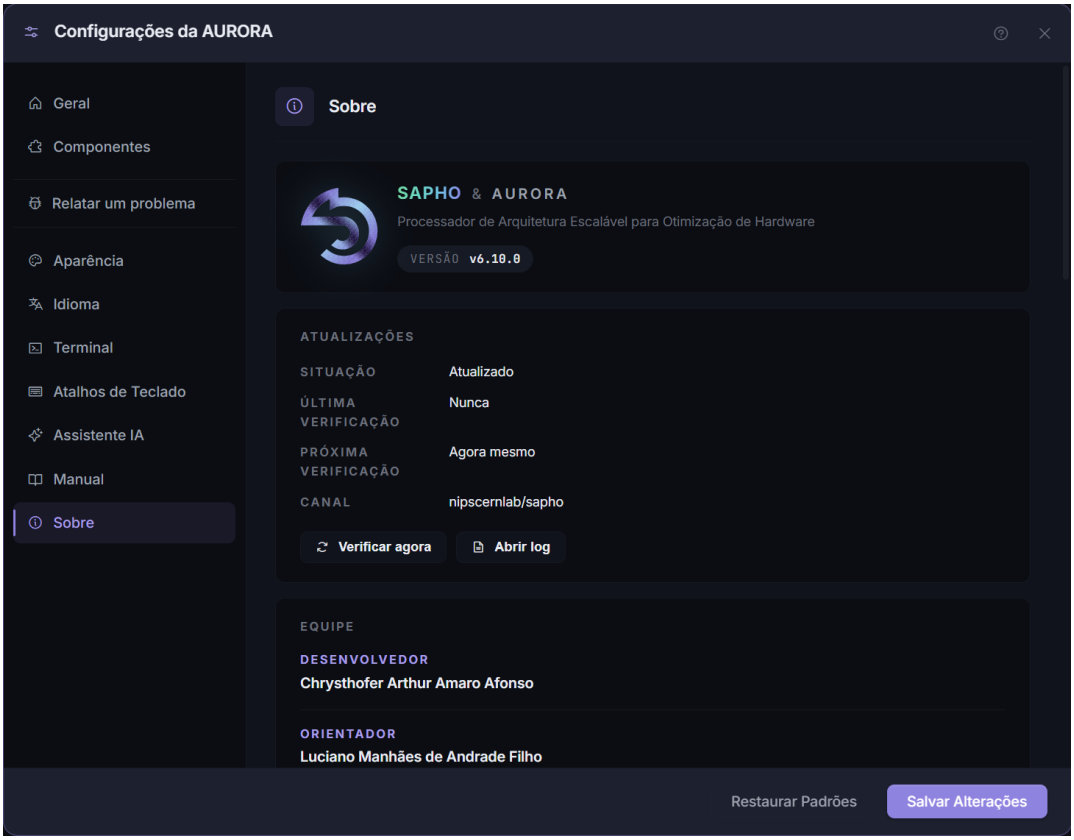
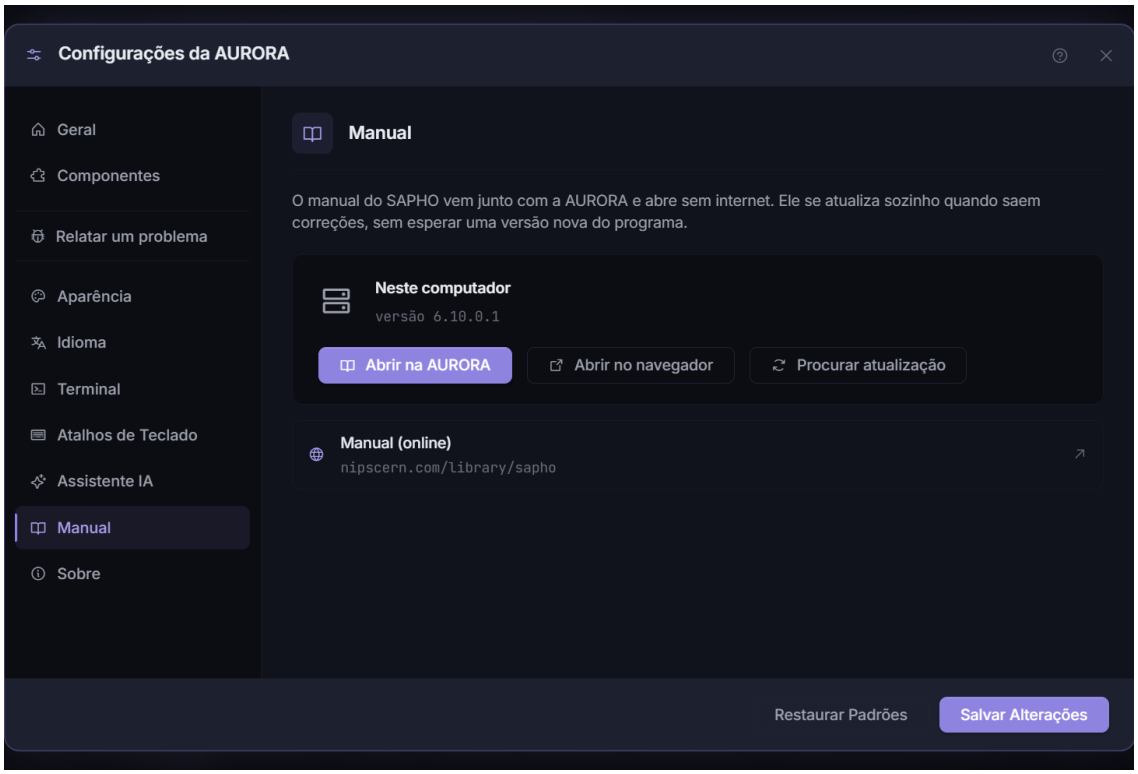


Figura24.3: A aba Geral. As opções de conta do GitHub e do GitLab são as primeiras, porque são as que mais importam em computador compartilhado.




Configurações da AURORA
Geral
Componentes
Relatar um problema
Aparência
Idioma
Terminal
Atalhos de Teclado
Assistente IA
Manual
Sobre
Sobre
SAPHO & AURORA
Processador de Arquitetura Escalável para Otimização de Hardware
VERSÃO v6.10.0
ATUALIZAÇÕES
SITUAÇÃO Atualizado
ÚLTIMA VERIFICAÇÃO Nunca
PRÓXIMA VERIFICAÇÃO Agora mesmo
CANAL nipscernlab/sapho
Verificar agora
Abrir log
EQUIPE
DESENVOLVEDOR
Chrysthofer Arthur Amaro Afonso
ORIENTADOR
Luciano Manhães de Andrade Filho
Restaurar Padrões
Salvar Alterações


Configurações da AURORA
Geral
Componentes
Relatar um problema
Aparência
Idioma
Terminal
Atalhos de Teclado
Assistente IA
Manual
Sobre
Manual
O manual do SAPHO vem junto com a AURORA e abre sem internet. Ele se atualiza sozinho quando saem correções, sem esperar uma versão nova do programa.
Neste computador
versão 6.10.0.1
Abrir na AURORA
Abrir no navegador
Procurar atualização
Manual (online)
nipscern.com/library/sapho
Restaurar Padrões
Salvar Alterações

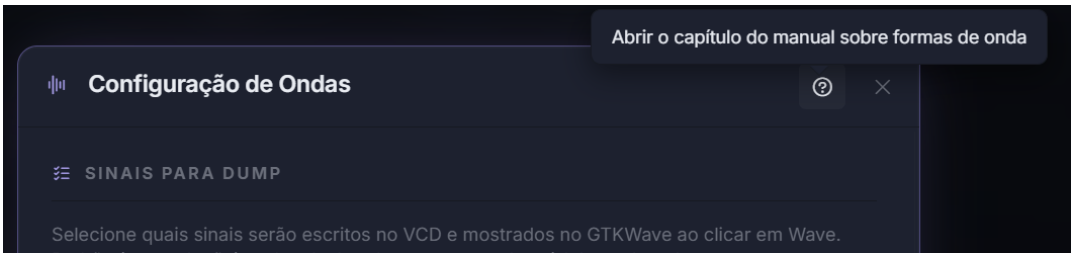


Figura24.4: O botão no cabeçalho do modal de ondas. A dica diz aonde ele leva, antes do clique.

aparece numa janela da própria AURORA, com busca, sumário e os botões de voltar e avançar, e funciona sem internet, porque é a cópia instalada que está sendo lida. Se a cópia offline não estiver na máquina, o mesmo botão abre a página equivalente no site.

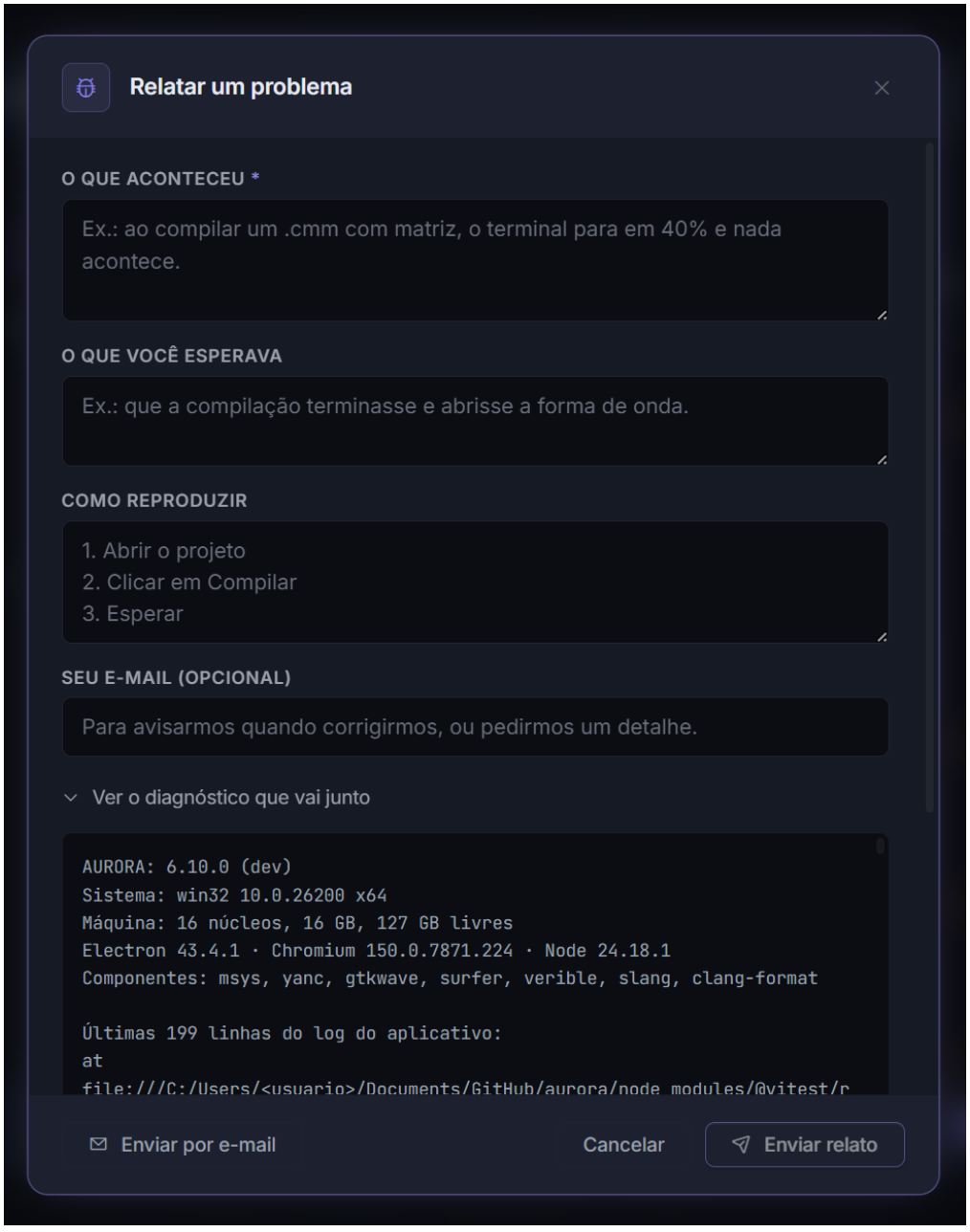


Figura24.5: O que o botão do modal de ondas abriu: este capítulo, na cópia instalada, dentro da AURORA.

24.6 Relatar um problema

O botão *Relatar um problema*, na barra lateral das Configurações, abre o formulário de relato: o que aconteceu, o que você esperava e como reproduzir, mais um e-mail opcional para receber a resposta. Antes de enviar, a seção *Ver o diagnóstico que vai junto* mostra exatamente o que acompanha o relato: versão, sistema, quais componentes a máquina tem e o terminal recortado em volta dos erros, com a vizinhança de cada um. Conteúdo de arquivos, senhas e conversas com a Aurora Intelligence nunca vão junto, e o nome de usuário é removido dos caminhos.

Enviar relato manda direto para a equipe; *Enviar por e-mail* monta a mensagem no seu webmail, para quem preferir. Quanto mais contexto no relato, melhor a resposta.


Relatar um problema
O QUE ACONTECEU *
Ex.: ao compilar um .cmm com matriz, o terminal para em 40% e nada acontece.
O QUE VOCÊ ESPERAVA
Ex.: que a compilação terminasse e abrisse a forma de onda.
COMO REPRODUZIR
1. Abrir o projeto
2. Clicar em Compilar
3. Esperar
SEU E-MAIL (OPCIONAL)
Para avisarmos quando corrigirmos, ou pedirmos um detalhe.
Ver o diagnóstico que vai junto
AURORA: 6.10.0 (dev)
Sistema: win32 10.0.26200 x64
Máquina: 16 núcleos, 16 GB, 127 GB livres
Electron 43.4.1 · Chromium 150.0.7871.224 · Node 24.18.1
Componentes: msys, yanc, gtkwave, surfer, verible, slang, clang-format
Últimas 199 linhas do log do aplicativo:
at
file:///C:/Users/<usuario>/Documents/GitHub/aurora/node_modules/@vitest/r
Enviar por e-mail
Cancelar
Enviar relato

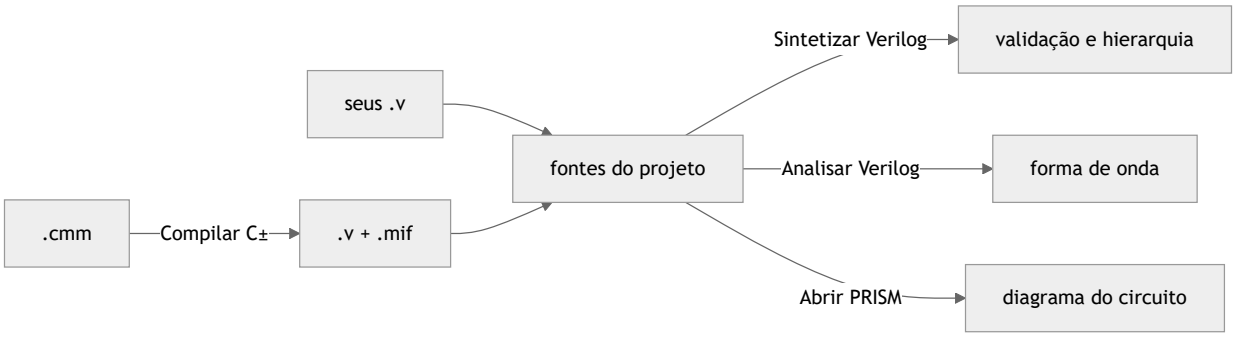
Parte VII

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± ou C++ até Verilog e memórias	o fonte do processador 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
Simular, no PRISM	liga o módulo da tela, com relógio e chaves	módulo pequeno o bastante
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>/<proc>.cmm                voce escreve (o nome do arquivo e o do processador)
<proc>/input_N.txt               estimulo, um inteiro por linha
<proc>/<proc>/<proc>.v + .mif    a compilacao gera (levar ao FPGA)
<proc>/<proc>/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
`F6` compilar C±	`F7` sintetizar Verilog
`F8` simular com onda	`F10` abrir o PRISM
`Ctrl+Shift+K` paleta de comandos	`Ctrl+K` assistente de IA
`Shift+Alt+F` formatar	`Ctrl+Shift+H` histórico de compilações

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. A linguagem, as diretivas e as mensagens do compilador estão no manual do YANC[27].

[27] https://www.nipscern.com/library/yanc/

Atalhos de teclado

26.1 Personalizáveis

Uma tabela só, regravável em *Configurações*, aba *Atalhos de Teclado*: clique no atalho e pressione a combinação nova. A mesma tabela alimenta a paleta de comandos, então tudo aqui também se acha pelo nome.

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+N`	novo projeto
`Ctrl+Shift+O`	abrir projeto
`Ctrl+Alt+B`	backup do projeto
`F5`	compilar tudo
`F6`	compilar C±
`F7`	sintetizar Verilog
`F8`	simular com forma de onda (*Analisar Verilog*)
`F9`	simulação rápida, sem onda
`F10`	abrir o PRISM
`Shift+F5`	cancelar a compilação
`Ctrl+Shift+H`	histórico de compilações
`Ctrl+Shift+L`	limpar o terminal
`Ctrl+Alt+P`	Hub de Processadores
`Ctrl+Shift+C`	abrir as Configurações
`Ctrl+Alt+S`	ligar e desligar a análise semântica de Verilog

26.2 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

26.3 No Simular do PRISM

Valem com o foco no diagrama, fora de campos de texto.

Atalho	Ação
`Espaço`	rodar e pausar
`→`	avança um tick
`Shift+→`	salta ao próximo evento
`Esc`	tira o cursor de tempo; sem cursor, sobe um nível de submódulo

26.4 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

26.5 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
`Ctrl+clique` / `Shift+clique`	seleção múltipla: soma um, ou pega o intervalo
`Ctrl+A`	seleciona o que está visível

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

26.6 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

Capítulo 27

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.

27.1 Botões desabilitados

Botão cinza	Falta
Compilar C±	um fonte do processador (`.cmm` ou `.cpp`) 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

Um caso à parte: se o clique abre o diálogo *Componente não instalado*, não falta pré-requisito de projeto — falta a ferramenta. *Baixar agora* resolve na hora, e o painel completo está em *Controle de versão, Python, componentes e configurações*.

27.2 Projeto

O nome foi recusado
: Nomes de projeto aceitam letras, números, hífen e sublinhado, sem espaços e sem acentos. O nome do processador é mais estrito, porque vira módulo Verilog: só letras sem acento, dígitos e _, sem hífen, sem dígito nem __ no começo, e nenhuma palavra reservada do Verilog.

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.

27.3 Compilação C±

Cada mensagem do compilador, com a causa e a correção, está no manual do YANC: as mensagens mais comuns, explicadas[28] e a lista completa[29]. Aqui ficam os casos em que a AURORA entra no meio.

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 na pasta de saída mas o terminal mostra erro
: O arquivo é de uma compilação anterior. Vale o terminal.

[28] https://www.nipscern.com/library/yanc/messages.html
[29] https://www.nipscern.com/library/yanc/reference/messages.html

`#PRNAME não existe mais`
Desde o YANC 7.1 o nome do processador é o nome do arquivo. Apague a linha; para outro nome, renomeie o processador.

O nome do arquivo não serve de processador
O nome vira módulo Verilog: só letras sem acento, dígitos e _, sem dígito nem __ no começo, e nenhuma palavra reservada do Verilog. Renomeie o processador.

O fonte não tem o nome do processador
Um `outro.cmm` dentro da pasta do processador `filtro` geraria o módulo `outro`. Renomeie o arquivo para `filtro.cmm`, ou mova-o para um processador com o nome dele.

27.4 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.

27.5 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
Com o Verilator, os monitores de pilha e ULA entram por espelhos no testbench, mas o miolo mais profundo fica de fora; e um programa sem funções não tem pilha de instruções para monitorar. Para ver tudo, troque para o Icarus.

A simulação aborta citando o arquivo de onda
O arquivo da rodada anterior está aberto em outro programa ou ficou somente-leitura, e a AURORA prefere abortar a mostrar uma onda velha como se fosse nova. Feche o visualizador (GTKWave/Surfer) e simule de novo.

`$fopen devolveu 0` ou "invalid file descriptor"
O testbench abre um arquivo que não existe ou não pode ser escrito. A AURORA avisa antes de rodar quando o caminho de leitura não existe; confira os caminhos dos `$fopen` do testbench.

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.

27.6 PRISM

Grande demais para simular ao vivo
O módulo passa do limite de células do modo *Simular*. Abra um submódulo menor no esquemático e simule aquele; o diagrama estático não tem limite prático.

O relógio não bate na simulação do PRISM
O modo só cria relógio para uma entrada de um bit chamada `clk` ou `clock` no módulo simulado. Sem

ela, avance com *Tick*.

Diagrama desatualizado
Recompile na própria janela do PRISM.

27.7 Quando nada explica

- Ligue o modo verboso (*Configurações*, *Terminal*) e repita: os comandos completos de cada etapa aparecem no terminal.
- O *Histórico de compilações* (relógio na barra dos terminais, `Ctrl+Shift+H`) mostra o que cada clique fez e o estado do projeto na hora, o que ajuda a reconstruir o caminho até o erro.
- Rode *Verificar e consertar* na aba *Componentes*: ele limpa os caches de compilação, confere os arquivos de cada componente e baixa de novo o que estiver incompleto ou quebrado — o conserto clássico para uma toolchain que o antivírus mordeu.
- Use o *Relatar um problema* da barra lateral das Configurações: o relato já vai com o diagnóstico da instalação e o terminal recortado em volta dos erros, e a tela mostra tudo antes de enviar.

Parte VIII

Apêndices

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.

Componente
Uma ferramenta que a AURORA baixa por demanda em vez de trazer no instalador: a cadeia de compilação, os visualizadores de onda, os agentes de IA. Gerenciados em Configurações, aba Componentes.

Fonte sintetizável
Arquivo Verilog marcado como parte do circuito. O papel é você quem escolhe, pelo menu de contexto da árvore; um arquivo novo entra sem papel.

Git-D
O painel de controle de versão da AURORA, com contas GitHub e GitLab. Chamava-se Dagr até a versão 6.4; só o nome mudou.

Histórico de compilações
O registro que cada clique de compilação deixa: o que rodou, quanto durou e o retrato do projeto na hora. Fica em `.aurora/execucoes/`, dentro do projeto.

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 e, no modo Simular, liga o módulo da tela, com relógio e entradas clicáveis.

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; abre como aba dentro do próprio editor.

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; grava a onda por escopo, e o miolo mais profundo dos processadores fica de fora (os monitores de pilha e ULA entram por espelhos no testbench).

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.

29.1 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[30]

29.2 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[31]
- **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[32]
- **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[33]
- **Arquitetura Multi-Core de Processadores Reconfiguráveis para Reconstrução Online de Energia** (CBA, 2020). Vários processadores cooperando, o padrão da Parte de estudos avançados. PDF[34]

29.3 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[35]
- **Implementação de Fractal em FPGA Usando o Processador Soft-Core SAPHO** (ENMC, 2025). Um caso visual e didático de uso. PDF[36]
- **Implementação Embarcada em FPGA de Métodos Visando a Reconstrução Online de Energia no Calorímetro** (dissertação, 2023). Deconvolução iterativa multicore com SAPHO. PDF[37]

29.4 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.

[30] https://cdn.nipscern.com/publications/ieee-2026-soft-core-processors.pdf
[31] https://cdn.nipscern.com/publications/enemc-2025-implementacao-de-funcoes.pdf
[32] https://cdn.nipscern.com/publications/tcc-2023-lucca-oliveira.pdf
[33] https://cdn.nipscern.com/publications/tcc-2016-leandro-silva.pdf
[34] https://cdn.nipscern.com/publications/cba-2020-arquitetura-multi-core.pdf
[35] https://cdn.nipscern.com/publications/sbai-2025-simulador-de-pulsos-do-tilecal.pdf
[36] https://cdn.nipscern.com/publications/enemc-2025-implementacao-de-fractal.pdf
[37] https://cdn.nipscern.com/publications/mest-2023-melissa-santos.pdf

Capítulo 30

Links e onde encontrar a gente

Tudo o que orbita este manual em um lugar só: os endereços oficiais da plataforma, do laboratório e das ferramentas que vêm na caixa.

30.1 As duas casas

O NIPS-CERN, Núcleo de Instrumentação e Processamento de Sinais, tem dois endereços, e é a distância entre eles que explica a plataforma: a pesquisa nasce em Juiz de Fora e é usada em Genebra.

Departamento de Engenharia Elétrica, PPEE
R. José Lourenço Kelmer, s/n
Juiz de Fora, MG 36036-900, Brasil

luciano.andrade@ufjf.br

Esplanade des Particules 1
CH-1211 Genève 23, Suisse

chrysthofer.afonso@cern.ch

Para assuntos gerais, parcerias e vagas: contact@nipscern.com.

30.2 Na rede

nipscern.com. Projetos, publicações, notícias e a equipe. É de lá que sai o instalador do SAPHO.

https://www.nipscern.com

@nipscern. O dia a dia do laboratório: bancada, viagens ao CERN, defesas, resultados novos. É por onde a gente conversa mais rápido.

https://www.instagram.com/nipscern/

A organização nipscernlab: o código do SAPHO, da AURORA, do YANC e deste manual, tudo aberto.

https://github.com/nipscernlab

Os projetos que vivem no GitLab, entre eles o nosso fork do Surfer.

https://gitlab.com/nips-cern

A colaboração com o CERN, o calorímetro TileCal do experimento ATLAS e o que a nossa eletrônica faz lá dentro.

https://www.nipscern.com/cern.html

As duas casas do laboratório: o NIPS, na UFJF, em Juiz de Fora, e a Route Salam, no CERN, em Genebra.

https://www.nipscern.com/about.html

30.3 A plataforma

A apresentação da plataforma, com o processador desenhado bloco a bloco e o download da última versão.

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

O instalador completo, direto do GitHub: IDE, compiladores, simuladores e visualizadores em um arquivo só.

https://github.com/nipscernlab/sapho/releases/latest

A IDE por dentro: editor, terminais, Aurora Intelligence e o resto da janela onde você trabalha.

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

A referência da linguagem C±, do C++ aceito, das diretivas, da biblioteca, das instruções e das mensagens do compilador. Em inglês, atualizado a cada versão.

https://www.nipscern.com/library/yanc/

O compilador que traduz C± e C++ em processador: o yanc, com o cmmcomp, o cppcomp e o asmcomp.

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

Achou um erro ou quer sugerir um capítulo? O manual é um repositório aberto, e uma issue basta.

https://github.com/nipscernlab/docs_aurora

As dúvidas que mais chegam, com resposta curta. Inclui como colaborar e como trabalhar conosco.

https://www.nipscern.com/qa.html

30.4 Leitura

SAPHO: An FPGA Customizable Implementation (IEEE, 2026). A arquitetura e os resultados em FPGA.

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

O simulador de pulsos do TileCal em tempo real (SBAI, 2025), rodando na eletrônica do ATLAS.

https://cdn.nipscern.com/publications/sbai-2025-simulador-de-pulsos-do-tilecal.pdf

Implementação de Fractal em FPGA usando o SAPHO (ENMC, 2025). O caso mais visual de todos, e ótimo para aula.

https://cdn.nipscern.com/publications/enemc-2025-implementacao-de-fractal.pdf

A produção completa do laboratório, de 2001 até hoje, com os PDFs.

https://www.nipscern.com/publications

O que anda acontecendo por aqui: versões novas, congressos, gente chegando.

https://www.nipscern.com/news/

A seleção comentada, capítulo por capítulo, do que vale ler junto com cada parte.

publicacoes.html

30.5 As ferramentas que vêm na caixa

Todas de código aberto, todas instaladas junto com o SAPHO. Os links vão para os projetos originais, que merecem a visita.

O simulador padrão.

https://steveicarus.github.io/iverilog/

Simulação rápida.

https://www.veripool.org/verilator/

As formas de onda.

https://gtkwave.sourceforge.net/

Ondas, no nosso fork.

https://gitlab.com/nips-cern/surfer-aurora

A síntese que o PRISM desenha.

https://yosyshq.net/yosys/

Testbenches em Python.

https://www.cocotb.org/

O editor dentro da AURORA.

https://code.visualstudio.com/api/monaco

A base da aplicação.

https://www.electronjs.org/

30.6 Falar com a gente

Assunto	Onde
Dúvida sobre a plataforma ou o manual	contact@nipscern.com
Coordenação do laboratório	luciano.andrade@ufjf.br
A plataforma e este manual	chrysthofer.afonso@cern.ch
Defeito na AURORA	A aba *Sobre* das configurações monta o relato com o diagnóstico da instalação, ou abra uma issue no repositório da AURORA[38]
Erro neste manual	Uma issue em docs_aurora[39]
Parcerias, pesquisa, vagas	contact@nipscern.com e a página Sobre[40]

O laboratório soma 147 publicações no registro, de 2001 a 2026, com 32 orientações concluídas e 18 pessoas hoje. Os números vivos ficam no site[41].

[38] https://github.com/nipscernlab/aurora/issues
[39] https://github.com/nipscernlab/docs_aurora/issues
[40] https://www.nipscern.com/about.html
[41] https://www.nipscern.com

Escopo, versão e método

31.1 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	6.11.0
Commit	`ee4b8ec52c23e46497e62da9b2ea8beb219d4ab5`
Branch	`main`
Plataforma primária	Windows 10/11
Runtime	Electron 43.4
Editor	Monaco Editor 0.52.2, fixado exatamente
Data da análise	30 de agosto de 2026 (base anterior, 6.10.0, apurada em 27 de agosto de 2026)

31.2 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.

31.3 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.

31.4 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.

31.5 Créditos

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