AURORA

Manual de uso e referência técnica

Versão 6.3.2

NIPSCERN

Núcleo de Inovação e Pesquisa em Sistemas Computacionais

Junho de 2026

Sumário

Parte I

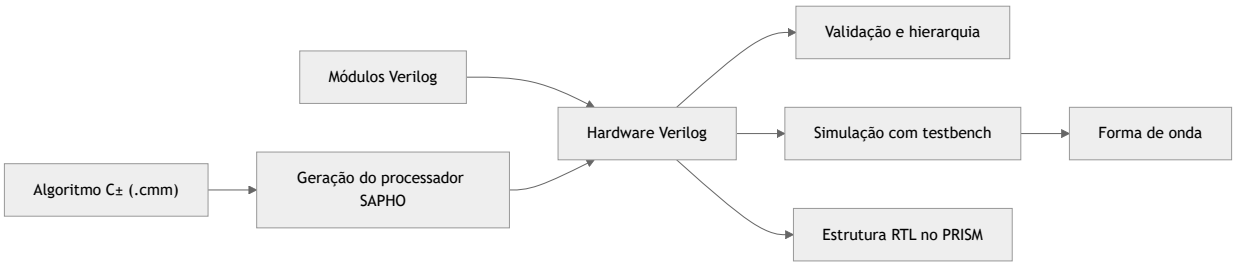
Primeiros passos

Capítulo 1

Entenda os fluxos da AURORA

Esta página apresenta como as etapas de um projeto se conectam. Ao final, você saberá distinguir o arquivo que descreve o circuito, o ponto de entrada da validação e o arquivo que controla a simulação.

A AURORA reúne dois caminhos principais: o desenvolvimento direto de RTL Verilog e a geração de processadores SAPHO a partir de um algoritmo C±, salvo em um arquivo `.cmm`. Os dois caminhos passam a compartilhar as mesmas ferramentas depois que o hardware Verilog está disponível.



1.1 Como as etapas se relacionam

1. No fluxo Verilog, você cria ou importa diretamente os módulos RTL.
2. No fluxo SAPHO, o YANC transforma o algoritmo C± em módulos Verilog na pasta `Hardware`.
3. Você define o **Top Level**, que identifica a raiz do circuito.
4. Para simular, você também define o **Testbench Top**, que contém ou inicia os estímulos e as verificações.
5. Icarus ou Verilator executa a simulação. Um testbench pode ser escrito em Verilog ou em Python com cocotb.
6. A simulação pode gerar uma forma de onda para análise no GTKWave ou no Surfer.
7. O PRISM usa o RTL validado para apresentar módulos e conexões; ele não executa o testbench.

Uma validação bem-sucedida confirma que os módulos podem ser elaborados. A simulação verifica o comportamento definido pelo testbench. A forma de onda permite observar os sinais ao longo do tempo, enquanto o PRISM apresenta a organização estrutural do circuito.

1.2 Conceitos que organizam o projeto

Conceito	Definição	Por que é necessário
Projeto `.spf`	Arquivo que registra fontes, processadores e seleções do projeto.	Permite que a AURORA restaure o contexto de trabalho.
Algoritmo C±	Código-fonte `.cmm` usado para gerar um processador SAPHO.	É a entrada editável do fluxo SAPHO.
Processador ativo	Processador associado ao arquivo `.cmm` aberto.	Determina qual algoritmo será compilado e testado pelas ações de processador.
Fonte sintetizável	Módulo Verilog que pertence ao circuito.	Participa da elaboração, da hierarquia e do PRISM.
Top Level	Módulo Verilog que representa a raiz do circuito completo.	Orienta a validação, a hierarquia, o PRISM e a associação com testbenches cocotb.
Testbench Top	Arquivo `.v` ou `.py` selecionado para iniciar a simulação.	Define os estímulos, as verificações e o término do teste.
Forma de onda	Registro temporal dos sinais gerado pela simulação, normalmente em VCD ou FST.	Permite investigar a sequência de eventos e valores do circuito.

O **Top Level** e o **Testbench Top** cumprem funções diferentes. Em um teste Verilog, o Testbench Top normalmente instancia o módulo definido como Top Level. Em um teste cocotb, o arquivo Python controla o simulador e acessa as portas do Top Level.

> **Dica**
>
> Antes de validar, simular ou abrir o PRISM, confira na barra de status se o processador ativo, o **Top Level** e o **Testbench Top** correspondem ao fluxo que você pretende executar.

Escolha seu roteiro em *Escolha o fluxo de trabalho*.

Capítulo 2

Instalação e primeiro início

Esta página orienta a instalação da AURORA e a verificação inicial do ambiente. Ao final, a aplicação estará pronta para abrir ou criar um projeto.

2.1 Requisitos

- Windows 10 ou Windows 11.
- Espaço para o aplicativo, os projetos e a toolchain.
- Permissão para gravar arquivos na pasta escolhida para os projetos.

2.2 Baixar a AURORA

O instalador da versão solicitada está publicado na página da versão 6.2.0 do repositório da AURORA:

Baixar a AURORA 6.2.0 nas Releases do GitHub[1]

Na página da versão, localize a seção **Assets** e baixe o instalador destinado ao Windows. Use exatamente essa versão quando estiver acompanhando um curso, laboratório ou projeto preparado para a AURORA 6.2.0.

2.3 Instalar

1. Abra o instalador baixado em AURORA 6.2.0 nas Releases do GitHub[2].
2. Siga as etapas apresentadas pelo instalador.
3. Abra a AURORA pelo menu Iniciar ou pelo atalho criado.
4. Aguarde a tela de abertura concluir a preparação do ambiente.
5. Abra **Configurações da AURORA** → **Sobre** e confira a versão instalada.

Na primeira execução, a preparação pode levar mais tempo porque a AURORA configura o ambiente de compilação. Quando a janela principal aparecer e responder aos comandos, a inicialização básica estará concluída.

2.4 Confirmar que está pronto

Após iniciar, verifique:

1. A janela principal abriu sem permanecer na tela de carregamento.
2. **Novo Projeto** e **Abrir Projeto** estão disponíveis.
3. O editor aparece no centro da janela.
4. **Configurações da AURORA** abre normalmente.
5. A barra inferior indica que a aplicação está pronta.

Se esses itens estiverem corretos, siga para *Primeiro projeto: escolha seu fluxo* e escolha entre o fluxo Verilog e o fluxo de processador SAPHO.

[1] https://github.com/nipscernlab/aurora/releases/tag/v6.2.0
[2] https://github.com/nipscernlab/aurora/releases/tag/v6.2.0

2.5 Se a AURORA não iniciar

1. Aguarde alguns segundos para confirmar que a preparação inicial não está apenas em andamento.
2. Verifique se outra janela da AURORA já está aberta.
3. Reinicie a aplicação pelo menu Iniciar.
4. Registre a versão do Windows e a etapa em que a abertura parou antes de solicitar suporte.

Consulte *Solução de problemas* para uma investigação orientada por sintomas.

2.6 Onde seus dados ficam

Você escolhe a pasta de cada projeto. Ela contém o arquivo `.spf`, os fontes e os arquivos gerados.

Preferências, projetos recentes, conversas da IA e outros dados da aplicação ficam no perfil do Windows. Esses dados facilitam a retomada da sessão, mas não substituem a pasta do projeto. Para fazer backup do trabalho, copie a pasta completa que contém o `.spf`, os fontes e os arquivos gerados; veja *Arquivos do projeto e backup*.

Capítulo 3

Primeiro projeto: escolha seu fluxo

A AURORA possui dois fluxos principais. Antes de criar arquivos, escolha se o hardware será escrito diretamente em Verilog ou gerado a partir de um algoritmo C± do ecossistema SAPHO.

Começar com Verilog Use módulos RTL próprios ou importados, adicione um testbench e siga diretamente para validação e simulação.

Fluxo completo para projetos Verilog Começar com um processador SAPHO Configure o processador no **Hub de Processadores**, escreva o algoritmo C± e gere o hardware Verilog antes de simular.

Fluxo completo para processadores SAPHO

3.1 O que os dois fluxos compartilham

Em qualquer opção, você criará ou abrirá um projeto `.spf`, trabalhará com arquivos no editor e definirá os pontos de entrada usados pelas ferramentas:

Top Level
: Módulo Verilog principal do circuito.

Testbench Top
: Arquivo Verilog ou Python/cocotb que executa o teste.

Depois dessa preparação, ambos podem usar Icarus, Verilator, cocotb, GTKWave ou Surfer e PRISM.

3.2 Diferença principal

No *Fluxo completo para projetos Verilog*, o RTL já é a entrada de trabalho: você escreve ou importa os módulos e os valida diretamente.

No *Fluxo completo para processadores SAPHO*, a entrada editável é o algoritmo `.cmm`: o YANC gera os módulos Verilog que seguem para as mesmas etapas de simulação e análise.

Consulte *Escolha o fluxo de trabalho* para uma comparação completa. Para conhecer a interface antes de iniciar, continue em *Interface principal*.

Parte II

Uso da AURORA

Interface principal

Esta página apresenta as áreas da janela principal e explica como o estado do projeto controla as ações disponíveis. A captura principal mostra a janela inteira da AURORA; as capturas detalhadas usam projetos adequados a cada recurso.

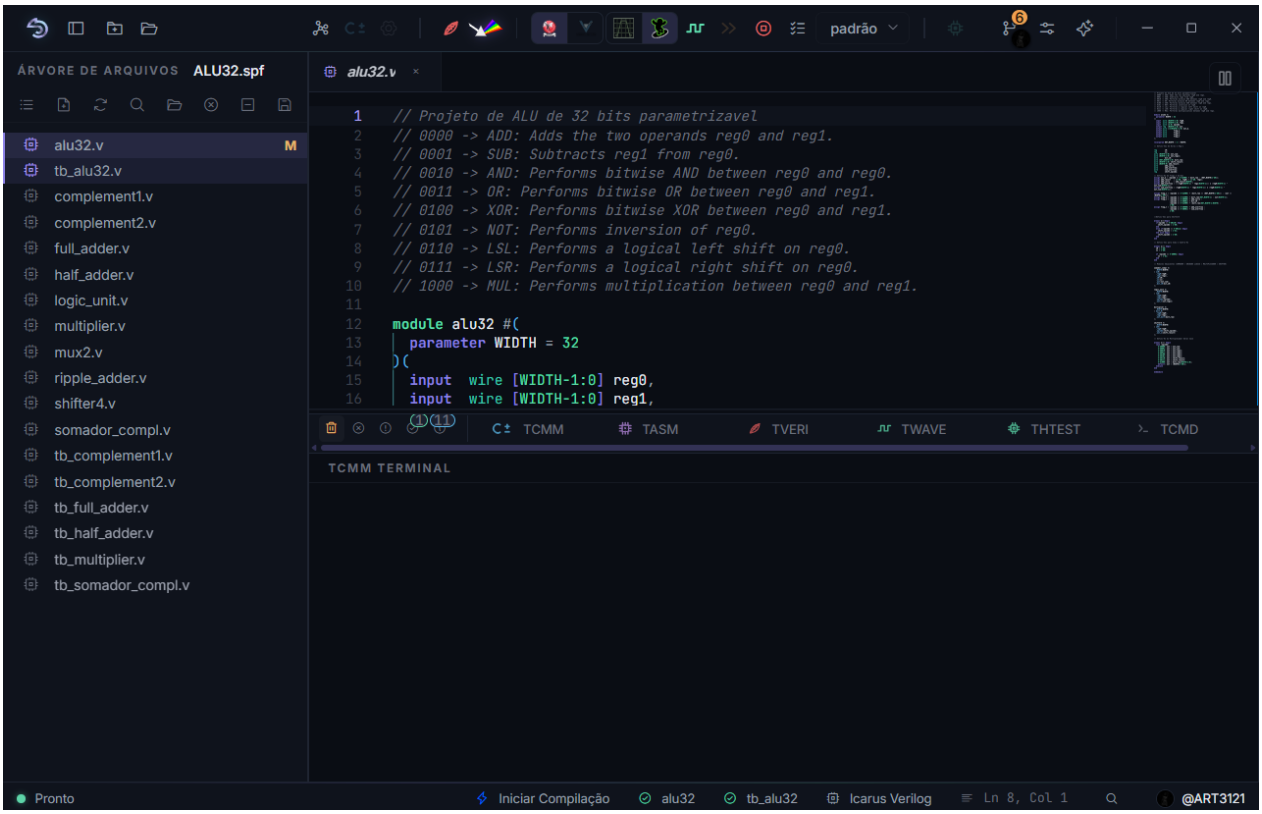


Figura1: A janela principal reúne a barra superior, a árvore do projeto, o editor, os terminais e a barra de status.

4.1 Áreas da janela

Área	Uso principal
Barra superior	Criar projetos, compilar, simular, abrir o PRISM, usar IA e acessar configurações.
Árvore lateral	Abrir arquivos, alternar visualizações e definir Top Level ou Testbench Top.
Editor	Editar C±, Assembly, Verilog, Python e arquivos auxiliares.
Terminais	Acompanhar mensagens de compilação, simulação e teste de hardware.
Barra de status	Conferir processador ativo, Top Level, Testbench Top e simulador.

O fluxo normal começa na árvore ou no editor, continua por um botão da barra superior e produz mensagens nos terminais. Antes de executar uma ação, confira a barra de status para evitar compilar ou simular o arquivo errado.

4.2 Botões principais

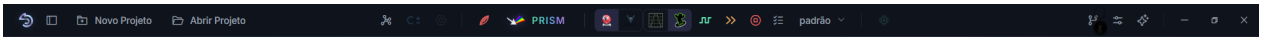


Figura2: A barra superior agrupa ações de projeto, processador, Verilog, simulação, controle de versão, configurações e Aurora Intelligence.

Controle em português	O que faz	Quando fica disponível
Novo Projeto	Cria um projeto.	Sempre que nenhum diálogo impedir a ação.
Abrir Projeto	Abre um arquivo `.spf`.	Sempre que nenhum diálogo impedir a ação.
Hub de Processadores	Cria um processador SAPHO.	Com um projeto aberto.
Compilar C±	Gera o hardware do processador ativo.	Com um arquivo `.cmm` aberto.
Configurações de simulação do processador	Ajusta frequência, quantidade de ciclos e exibição de arrays.	Com um processador ativo.
Compilar Verilog	Valida o conjunto Verilog e atualiza a hierarquia.	Com Top Level definido.
Analisar Verilog (forma de onda)	Executa a simulação, gera VCD ou FST e abre o viewer selecionado.	Com Testbench Top definido.
Execução rápida	Simula sem abrir formas de onda.	Com testbench `.py`, ou com testbench Verilog e Verilator selecionado.
Abrir PRISM	Abre a visualização RTL.	Com Top Level definido.
Teste do processador sintetizado	Executa o processador ativo com Verilator e mostra o resultado no THTEST.	Com um processador ativo e hardware gerado.
Configuração de Ondas	Seleciona os sinais gravados na forma de onda.	Com um projeto aberto.
Cancelar	Interrompe a execução atual.	Durante compilação ou simulação.

4.3 Árvore lateral

Um clique em um arquivo o abre no editor como uma aba de visualização. Um clique duplo mantém essa aba aberta de forma permanente. Nas pastas, um clique expande ou recolhe o conteúdo.

Na visualização **Arquivos**, clique com o botão direito do mouse em um arquivo para abrir o menu de contexto. Conforme o tipo e a classificação do arquivo, o menu permite **Definir como Top Level**, **Marcar como Testbench** ou **Excluir arquivo**. Arquivos Verilog sintetizáveis oferecem a seleção de Top Level; testbenches Verilog ou Python oferecem a seleção de Testbench Top.

A barra de ferramentas da árvore também permite criar arquivos, atualizar a lista, pesquisar, abrir a pasta no Explorer, fazer backup e fechar o projeto.

4.3.1 Visualizações da árvore

Visualização	Uso
Arquivos	Mostra fontes sintetizáveis, testbenches e arquivos associados ao fluxo HDL registrado no `.spf`.
Hierarquia	Mostra a árvore de módulos gerada para o Top Level após a validação Verilog.
Pastas	Espelha a estrutura real de pastas e arquivos do projeto no disco.

Clique no primeiro botão da barra da árvore para alternar entre as visualizações disponíveis.

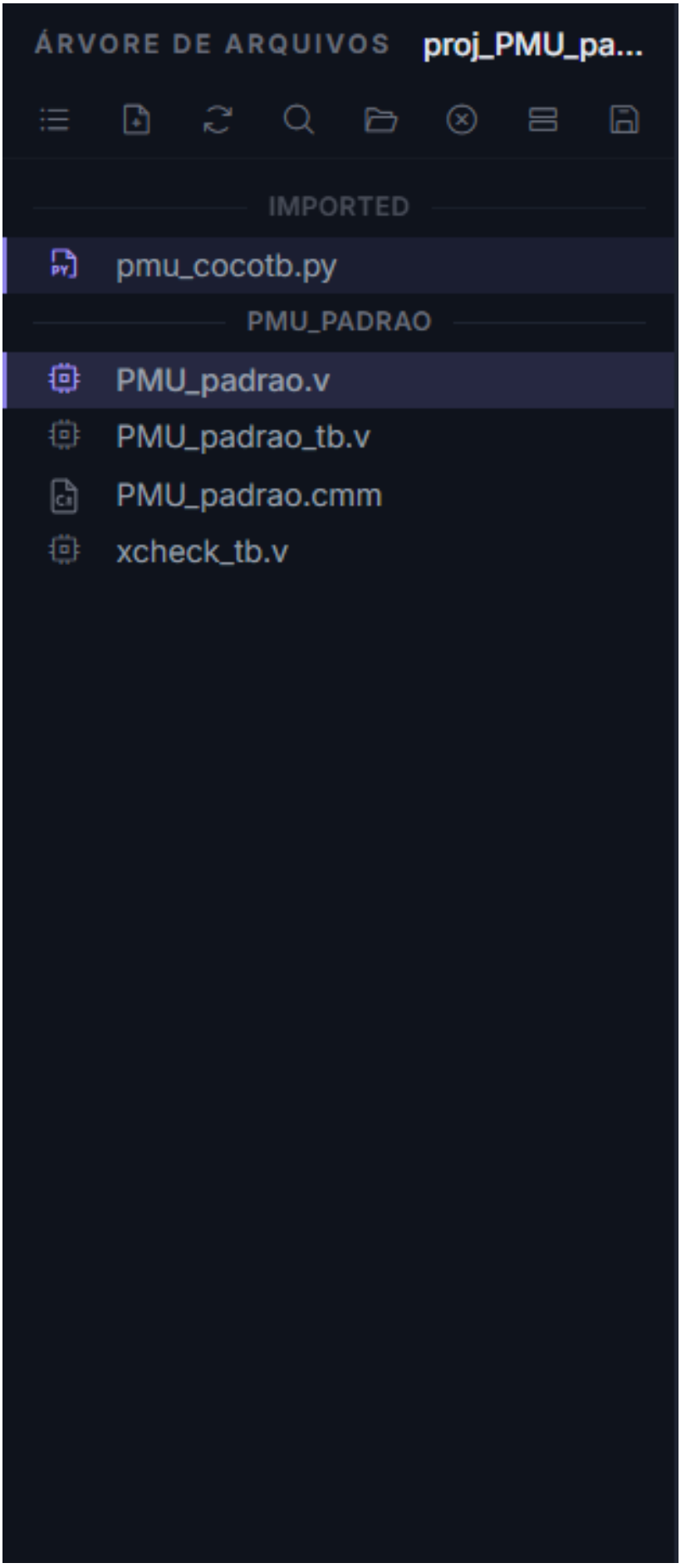


Figura3: Arquivos reúne as fontes e os testbenches registrados no projeto.

Figura4: Hierarquia apresenta `PMU_padrao` e os módulos elaborados a partir dele.

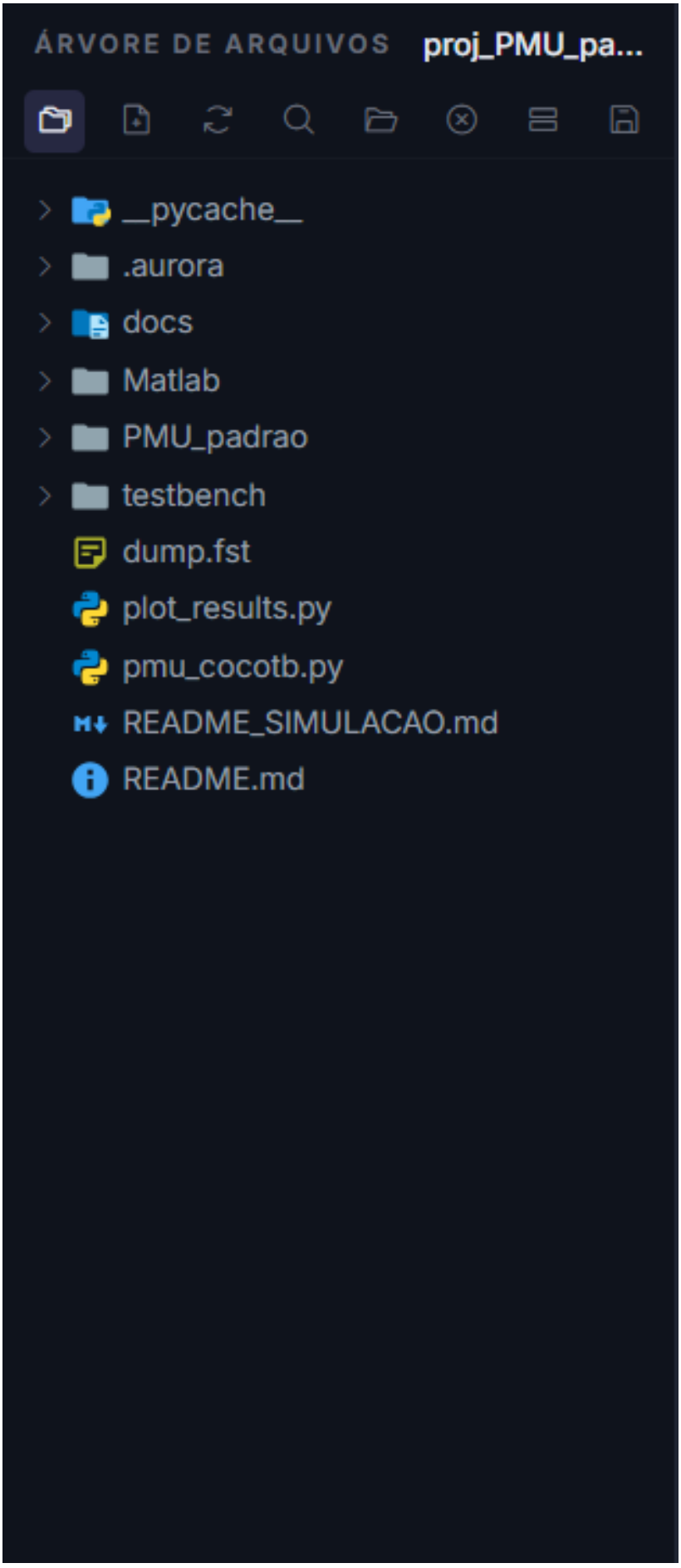


Figura5: **Pastas** apresenta a organização real do projeto no disco.

A visualização **Pastas** carrega diretórios sob demanda e memoriza as pastas abertas. O arquivo `.inv`, quando existe na raiz do projeto, filtra somente essa visualização. Ele não altera o Git nem remove arquivos do disco.

Quando o projeto está em um repositório Git, **Arquivos** e **Pastas** podem exibir indicadores de arquivos modificados, adicionados, removidos, renomeados, copiados, novos ou em conflito. Para operações de Git, veja *Source Control e Git*.

4.4 Terminais

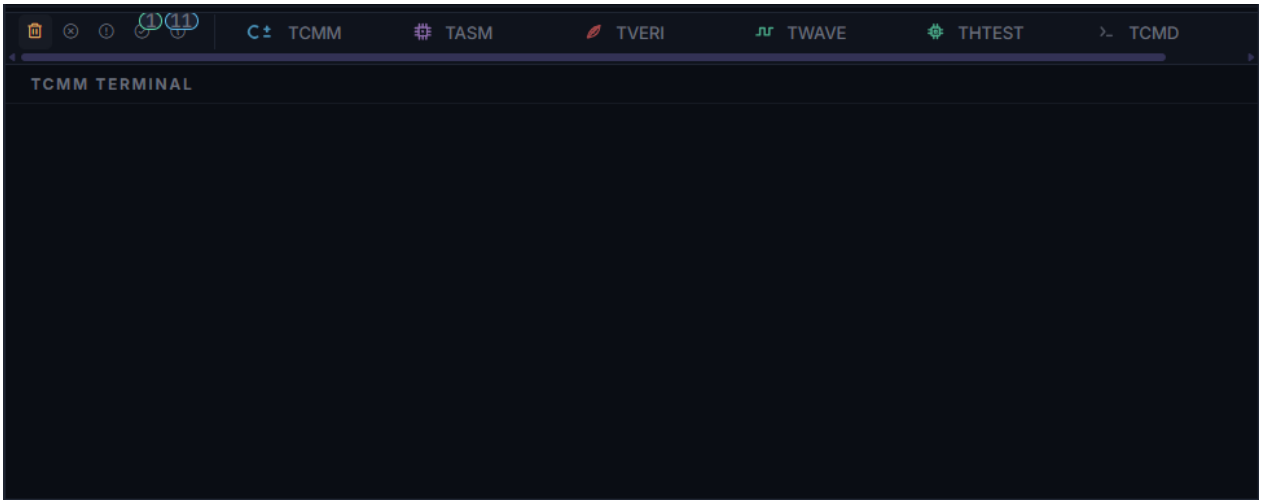


Figura6: Cada ação direciona a saída para o terminal correspondente. Na captura, o **TCMM** está selecionado; o **TCMD** permanece visível apenas como uma das abas disponíveis.

Os terminais são separados por etapa:

- **C±**: Mostra a tradução do algoritmo C±.
- **ASM**: Mostra a geração intermediária e a criação do hardware.
- **Verilog**: Mostra validação, elaboração, hierarquia, Icarus Verilog e Verilator.
- **Wave**: Mostra a execução dos testbenches e a preparação das formas de onda.
- **THTEST**: Mostra o teste de hardware do processador sintetizado.
- **TCMD**: Oferece um terminal PowerShell interativo dentro da AURORA.

O terminal **THTEST** é usado pelo botão **Teste do processador sintetizado**. A AURORA identifica as portas do processador ativo, gera um harness C++, compila o conjunto com Verilator e executa até o limite de ciclos configurado ou até o sinal `cheguei` indicar que o algoritmo alcançou `#TOAQUI`. Durante a execução, o terminal mostra o progresso, o número de ciclos e a quantidade de leituras de entrada. Ao terminar, apresenta as saídas e um link para abrir a pasta `Simulation` na visualização **Pastas**.

O terminal **TCMD** inicia uma sessão PowerShell real sob demanda. Ele abre no diretório do projeto e acompanha o contexto do projeto ou processador ativo. Use-o para consultar arquivos, executar comandos e trabalhar com ferramentas de linha de comando sem sair da AURORA. A ação **Abrir no terminal integrado** também seleciona o TCMD e muda a sessão para a pasta escolhida. Os comandos são executados com as permissões do usuário; confira o diretório exibido no prompt antes de alterar ou remover arquivos.

Use os filtros dos terminais para localizar erros e avisos. Comece pela primeira mensagem de erro, pois as mensagens seguintes frequentemente descrevem apenas consequências.

4.5 Barra de status

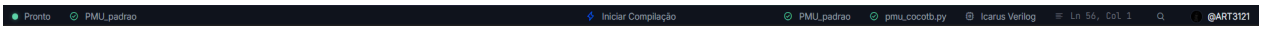


Figura7: A barra de status resume as seleções usadas na próxima compilação ou simulação.

Consulte a barra de status antes de compilar, simular ou abrir o PRISM. Ela mostra o processador ativo, o Top Level, o Testbench Top e o simulador. Se um valor não corresponder ao fluxo pretendido, abra o arquivo correto ou ajuste a seleção na árvore antes de continuar.

4.6 Quando um botão estiver desabilitado

Verifique nesta ordem:

1. Há um projeto aberto.
2. O arquivo correto está aberto no editor.
3. O processador ativo corresponde ao algoritmo C± desejado.
4. O Top Level foi definido.
5. O Testbench Top foi definido.
6. Nenhuma execução incompatível está em andamento.

Na maioria dos casos, um botão desabilitado indica que falta uma dessas seleções. Salve o arquivo ativo e aguarde a conclusão de qualquer operação anterior antes de tentar novamente.

Projetos

Um projeto reúne o arquivo de configuração `.spf`, os algoritmos C±, os módulos Verilog, os testbenches e os resultados gerados. Esta página explica como criar, abrir, mover e preservar esse conjunto sem quebrar referências.

5.1 Criar

1. Clique em **Novo Projeto**.
2. Informe um nome.
3. Escolha a pasta de destino.
4. Confirme.

O nome do projeto é usado na pasta e no arquivo `.spf`. Escolha um nome curto, descritivo e sem caracteres que possam ser rejeitados pelo Windows. Depois da confirmação, aguarde a árvore do projeto aparecer antes de criar processadores ou importar arquivos.

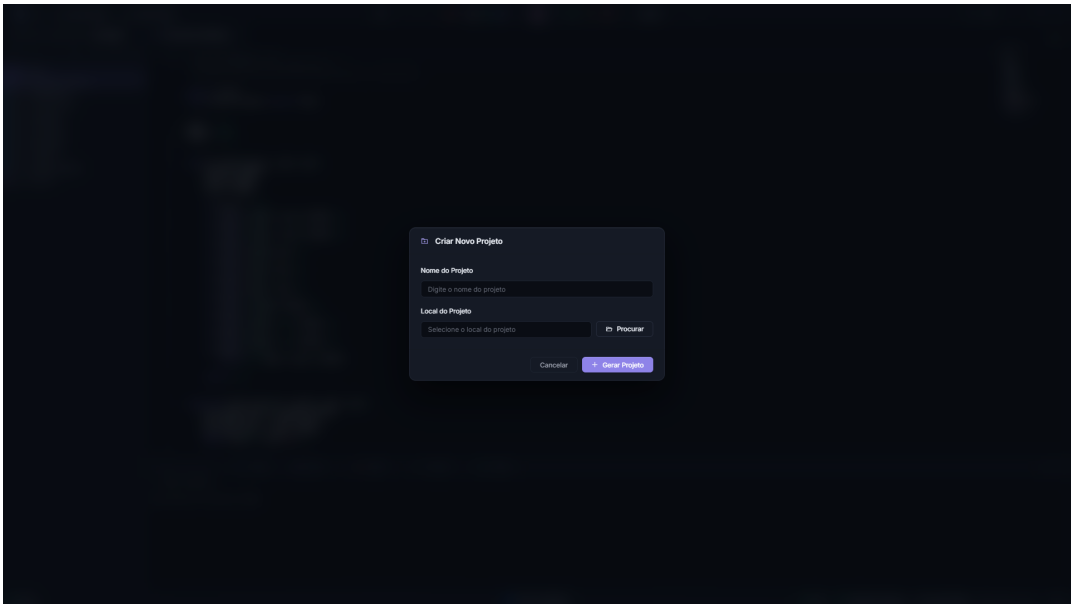


Figura1: Preencha o nome, escolha a pasta de destino em **Procurar** e confirme em **Gerar Projeto**.

5.2 Abrir

Clique em **Abrir Projeto** e selecione o arquivo `.spf`. Projetos usados recentemente também aparecem na tela inicial. Ao abrir, a AURORA restaura a estrutura registrada e verifica os caminhos dos arquivos associados.

Se um projeto recente foi movido ou excluído, abra-o novamente pela nova localização. Uma entrada recente não contém uma cópia do projeto; ela é apenas um atalho para o `.spf` existente no disco.

5.3 O que é o arquivo `.spf`

O `.spf` guarda a configuração do projeto, incluindo:

- Processadores.
- Arquivos Verilog e Testbenches.
- Top Level e Testbench Top.
- Caminhos de arquivos importados.

Não edite esse arquivo manualmente durante o uso normal. Para mover um projeto, feche-o na AURORA, copie a pasta completa e abra o `.spf` na nova localização. Depois, confirme se Top Level, Testbench Top e arquivos importados continuam válidos.

5.4 Navegar pela pasta do projeto

Use a visualização **Pastas** para conferir a estrutura real no disco. Ela mostra pastas e arquivos do projeto mesmo quando eles não fazem parte da lista Verilog registrada no `.spf`.

A visualização **Pastas** é útil para:

- Abrir arquivos auxiliares sem sair da AURORA.
- Conferir resultados gerados em `Hardware/`, `Simulation/` e `testbench/`.
- Revelar ou expandir pastas do projeto.
- Ver badges Git junto dos arquivos no disco.

Arquivos ocultos, metadados internos e alguns arquivos de configuração histórica não aparecem nessa view para reduzir ruído.

5.5 Filtrar a visualização Pastas com `.inv`

Crie um arquivo `.inv` na raiz do projeto para esconder arquivos e pastas somente na navegação da AURORA. A sintaxe é inspirada em `.gitignore`: aceita comentários, linhas vazias, negação com `!`, padrões de diretório, `*`, `?`, `**` e comparação sem diferenciar maiúsculas de minúsculas.

Exemplo:

```
# Esconder resultados locais volumosos.
Simulation/tmp/
*.log

# Manter um relatório visível.
!Simulation/relatorio.log
```

O `.inv` não altera Git, não muda o `.spf` e não impede que um arquivo seja aberto por caminho direto. Ao salvar o `.inv`, a visualização **Pastas** é atualizada.

5.6 Importar arquivos

Você pode usar arquivos dentro ou fora da pasta do projeto.

1. Adicione o arquivo pelo comando de importação ou arraste e solte um ou vários arquivos sobre a árvore **Arquivos**.
2. Confirme se ele é fonte sintetizável ou testbench.
3. Defina Top Level ou Testbench Top quando necessário.

O recurso de arrastar e soltar aceita os formatos suportados pelo fluxo HDL: Verilog e SystemVerilog (`.v`, `.sv` e `.vh`) e testbenches Python/cocotb (`.py`). A AURORA registra os arquivos importados no `.spf`; ela não transforma automaticamente um formato desconhecido em fonte compilável.

Para facilitar backup e compartilhamento, prefira manter as fontes dentro da pasta do projeto.

Arquivos externos permanecem dependentes do caminho original. Se o projeto for enviado para outra máquina sem esses arquivos, a árvore indicará referências ausentes e as etapas de compilação poderão falhar.

5.7 Fechar e reabrir

Use a ação **Fechar Projeto** na árvore. Salve arquivos modificados antes de fechar.

Ao reabrir, confira na barra de status:

- Processador ativo.
- Top Level.
- Testbench Top.
- Simulador selecionado.

Abra também um ou dois arquivos importantes para confirmar que os caminhos foram restaurados. Essa verificação é especialmente importante depois de mover a pasta ou restaurar um backup.

5.8 Backup

Use a ação de backup antes de:

- Renomear ou excluir processadores.
- Importar muitos arquivos.
- Alterar a estrutura Verilog.
- Atualizar a AURORA.

O backup não substitui um sistema de versionamento, mas permite voltar rapidamente a um estado anterior. Identifique a cópia com uma data ou uma descrição da mudança para evitar confundi-la com a versão ativa.

5.9 Renomear ou excluir

Use os comandos da AURORA para renomear ou excluir projetos, processadores e arquivos registrados. Alterar apenas nomes de pastas pelo Explorer pode deixar referências inválidas no `.spf`.

5.10 Versionamento

Para acompanhar alterações ao longo do tempo, mantenha a pasta do projeto em Git. A AURORA pode exibir decorações nas árvores e operar status, diff, stage, commit, branch, stash, pull e push pelo painel de Source Control. Veja *Source Control e Git*.

Capítulo 6

Editor, abas e painéis

O editor é a área de trabalho para algoritmos C±, módulos Verilog, testbenches e arquivos auxiliares. Esta página explica como organizar arquivos, salvar mudanças e comparar conteúdos sem perder o contexto do projeto.

6.1 Abrir arquivos

Clique duas vezes em um arquivo da árvore. Arquivos de texto são abertos em uma aba; imagens e conteúdos reconhecidos usam visualizadores próprios.

O editor oferece realce para C±, Assembly, Verilog, Python, JSON e outros formatos comuns. O realce ajuda na leitura, mas não garante que o conteúdo esteja correto; use as etapas de compilação e simulação para validar o arquivo.

Use `Ctrl+N` para criar um documento vazio chamado `Untitled-N`. A AURORA detecta o tipo pelo conteúdo e sugere a extensão correspondente quando o arquivo é salvo.

6.1.1 Criar um processador SAPHO com `$cmm`

1. Abra um documento vazio com `Ctrl+N`.
2. Digite somente `$cmm`.
3. Aguarde a expansão automática do modelo C±.
4. Use `Ctrl+S` e informe o nome do processador.

A AURORA substitui `$cmm` pelas diretivas padrão e pelo corpo inicial de um algoritmo C±. Ao salvar dentro de um projeto aberto, ela usa o nome escolhido em `#PRNAME`, registra o processador no `.spf` e cria a estrutura `<processador>/Software`, `<processador>/Hardware` e `<processador>/Simulation`. O arquivo `.cmm` é gravado em `Software` e passa a aparecer como processador do projeto.

6.2 Abas

- Clique em uma aba para ativá-la.
- Use `Ctrl+W` para fechar a aba atual.
- Use `Ctrl+Shift+T` para reabrir a última aba fechada.
- O marcador na aba indica alterações ainda não salvas.

Ao fechar uma aba modificada, leia a confirmação antes de descartar o conteúdo. Fechar a aba não remove o arquivo do projeto nem do disco.

6.3 Dividir o editor

Use o botão de divisão para abrir até três painéis. Isso é útil para comparar módulos, manter o testbench ao lado do RTL ou consultar um arquivo enquanto edita outro.

Se o mesmo arquivo estiver aberto em dois painéis, a alteração aparece nos dois porque ambos representam o mesmo documento. Use a divisão para consultar regiões diferentes ou comparar arquivos relacionados, não para criar versões independentes do mesmo conteúdo.

6.4 Localizar e navegar

Atalho	Ação
`Ctrl+F`	Localizar no arquivo
`Ctrl+H`	Localizar e substituir
`Ctrl+G`	Ir para uma linha
`F1`	Abrir a paleta de comandos do Monaco
`Ctrl+Shift+P`	Abrir a paleta de comandos da AURORA
`F12`	Ir para definição quando disponível

6.5 Salvar

- Atalho `Ctrl+S`: Salva o arquivo ativo.
- Atalho `Ctrl+Shift+S`: Salva todos os arquivos modificados.

Antes de compilar, confirme que o marcador de alteração desapareceu da aba.

Salvar antes da compilação é importante porque as ferramentas externas leem o conteúdo gravado no disco. Uma alteração visível no editor, mas ainda não salva, não fará parte da execução.

6.6 Usar a seleção com IA

Selecione um trecho de código para enviá-lo à Aurora Intelligence com o caminho e a linguagem do arquivo. Um pedido específico, como "explique por que este sinal nunca muda", produz um contexto melhor do que uma solicitação genérica. Peça primeiro uma explicação ou revisão. Alterações propostas pela IA exigem confirmação antes de serem aplicadas.

6.7 Mudanças externas

Quando o arquivo muda no disco enquanto possui alterações locais, escolha conscientemente qual versão manter. Em caso de dúvida, copie o trecho local antes de recarregar.

Depois de resolver o conflito, salve o arquivo e verifique se a aba deixou de indicar modificação. Em seguida, compile novamente para garantir que a versão correta foi utilizada.

Processadores SAPHO

Um processador reúne o algoritmo C±, a configuração numérica e os arquivos necessários para gerar o hardware. Esta página explica como criar, selecionar, gerar e testar um processador dentro de um projeto.

7.1 Criar no Hub de Processadores

Abra **Hub de Processadores**, preencha os campos e confirme.

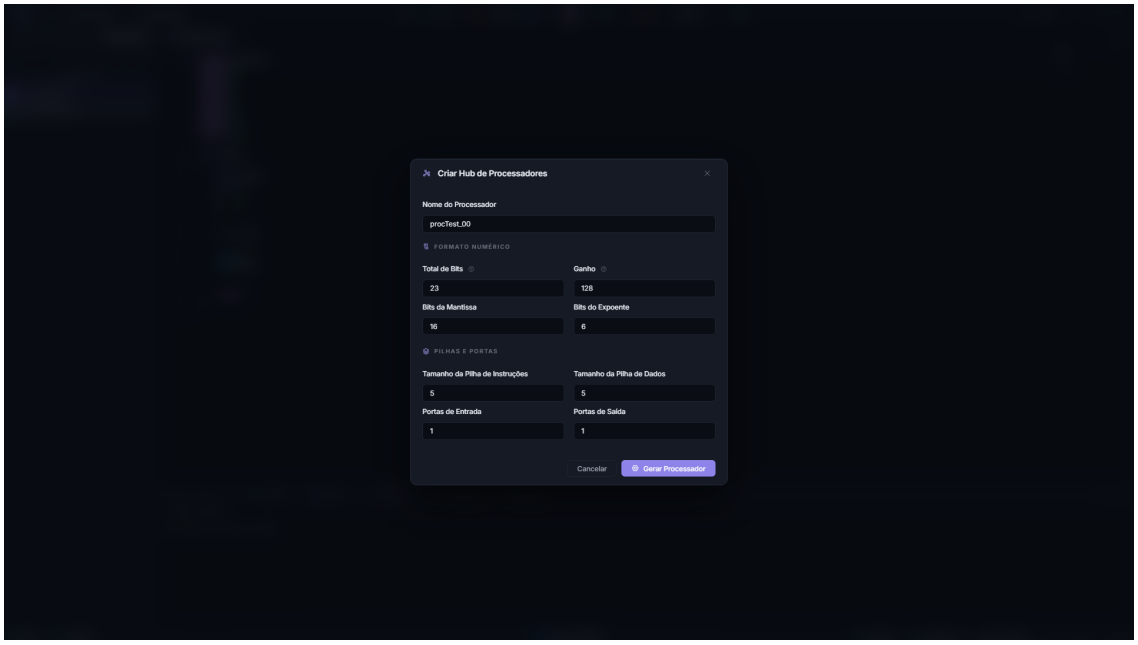


Figura1: O Hub de Processadores concentra os parâmetros usados para criar a estrutura inicial do processador SAPHO.

Campo	Significado
Nome	Identifica o processador e suas pastas
Bits totais	Largura da palavra processada
Ganho	Escala usada pela representação numérica
Mantissa / Expoente	Divisão dos bits de ponto flutuante
Pilha de instruções	Capacidade de chamadas e controle
Pilha de dados	Capacidade para valores temporários
Entradas / Saídas	Quantidade de portas de I/O

Use apenas números inteiros positivos. O ganho deve ser potência de dois e a soma de mantissa, expoente e bit de sinal deve corresponder aos bits totais. Se a configuração for rejeitada, revise primeiro essa relação e os tamanhos das pilhas antes de tentar valores diferentes aleatoriamente.

Os valores determinam características do hardware gerado. Para o primeiro projeto, mantenha uma configuração já validada pelo curso, laboratório ou exemplo de referência. Altere um parâmetro por vez quando precisar avaliar seu efeito.

7.2 Criar rapidamente com `$cmm`

1. Use `Ctrl+N` para criar um arquivo `Untitled-N`.
2. Digite somente `$cmm`.
3. Aguarde a AURORA expandir o modelo inicial do algoritmo C±.
4. Use `Ctrl+S` e informe o nome do processador.

Ao salvar dentro de um projeto aberto, a AURORA atualiza a diretiva `#PRNAME`, registra o processador no `.spf` e cria as pastas `<processador>/Software`, `<processador>/Hardware` e `<processador>/Simulation`. O arquivo `.cmm` é salvo dentro de `Software` e passa a aparecer como processador do projeto.

7.3 Arquivos criados

Cada processador possui:

```
<processador>/
├── Software/    algoritmo C±
├── Hardware/    Verilog gerado
└── Simulation/  entradas, saídas e arquivos de simulação
```

O arquivo `.cmm` inicial contém as diretivas da configuração escolhida. A pasta `Software` guarda o algoritmo editável; `Hardware` recebe os módulos Verilog; e `Simulation` concentra arquivos usados nos testes do processador.

7.4 Processador ativo

Abra o `.cmm` do processador que deseja usar. O nome ativo aparece na barra de status e habilita as ações relacionadas.

Se o botão **C±** estiver desabilitado, confirme que:

1. O projeto está aberto.
2. O arquivo `.cmm` do processador está ativo.
3. Nenhuma compilação está em andamento.

7.5 Configuração de execução

O painel do processador permite ajustar:

- Frequência de clock.
- Quantidade de ciclos.
- Exibição de arrays.

Esses valores afetam simulações e a execução direta do processador. A frequência participa da configuração temporal, enquanto a quantidade de ciclos limita a duração esperada da execução. Comece com os padrões e altere somente quando souber a duração necessária do teste.

7.6 Gerar o hardware

1. Abra o arquivo `.cmm`.
2. Salve as alterações.
3. Clique em **C±**.
4. Acompanhe os terminais.

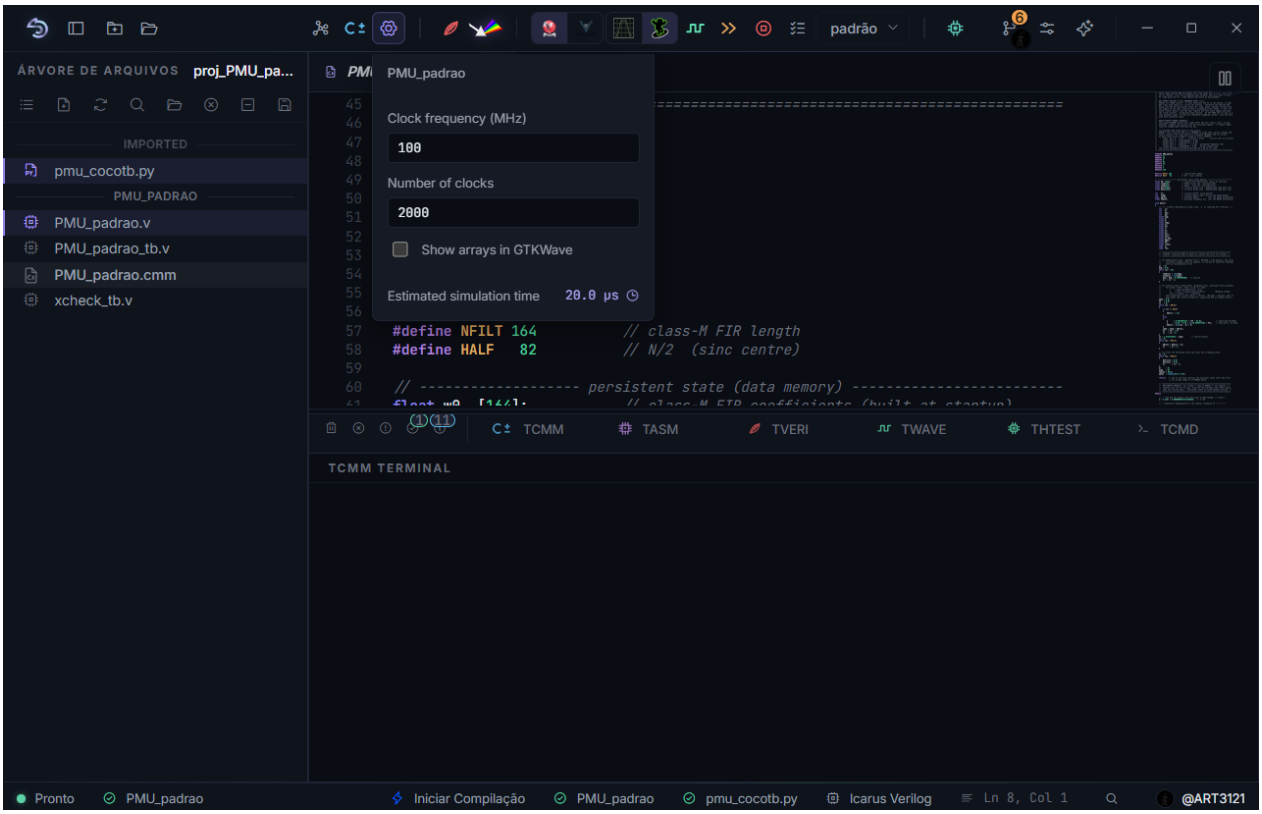


Figura2: No projeto `proj_PMU_padrao`, o painel da engrenagem aparece sobre o algoritmo `PMU_padrao.cmm`. Ele ajusta a frequência, o limite de ciclos e a inclusão de arrays nas formas de onda. O tempo estimado é atualizado com os valores informados.

5. Confirme que os arquivos apareceram em `Hardware`.

Leia o terminal de C± e, em seguida, o terminal de Assembly. O processo deve terminar sem erro antes que os arquivos em `Hardware` sejam considerados atualizados. Se a geração falhar, preserve a primeira mensagem de erro para o diagnóstico.

7.7 Executar diretamente

Teste do processador sintetizado executa o processador ativo pelo fluxo de teste dedicado. Use essa opção quando quiser testar entradas e saídas do processador sem montar uma análise completa no viewer de ondas. O resultado aparece no terminal **THTEST**.

7.8 Renomear ou excluir

Faça backup e use os comandos da AURORA. A exclusão pode remover arquivos do processador; leia a confirmação antes de prosseguir.

Capítulo 8

Arquivos Verilog e Testbenches

Os arquivos Verilog descrevem o circuito e o ambiente usado para testá-lo. Esta página mostra como classificá-los, definir os pontos de entrada e evitar seleções que levem a uma compilação incompleta.

8.1 Tipos de arquivo

A AURORA separa os arquivos em dois grupos:

Fontes sintetizáveis
: Módulos RTL que formam o circuito.

Testbenches
: Arquivos Verilog ou Python/cocotb usados somente para simulação.

Confirme a categoria ao importar. Um testbench não deve ser incluído como fonte sintetizável, pois contém estímulos e construções destinadas apenas à simulação. Da mesma forma, os módulos instanciados pelo circuito precisam estar entre as fontes selecionadas.

8.2 Adicionar arquivos

1. Localize o arquivo na árvore ou use a ação de importação.
2. Marque-o na categoria correta.
3. Repita para todas as dependências do projeto.

Arquivos externos funcionam, mas tornam o projeto mais difícil de mover. Sempre que possível, mantenha-os dentro da pasta do projeto. Depois da importação, abra o arquivo pela árvore para confirmar que o caminho registrado está acessível.

8.3 Definir o Top Level

Escolha o arquivo que contém o módulo principal do circuito, clique nele com o botão direito do mouse e selecione **Definir como Top Level**.

O Top Level deve declarar ou conter o módulo que representa o circuito completo. Ele é necessário para:

- Validar o Verilog.
- Abrir o PRISM.
- Montar a hierarquia do projeto.

Definir um módulo intermediário como Top Level pode produzir uma validação parcial e uma árvore incompleta. Confirme o nome do módulo principal antes de executar o fluxo.

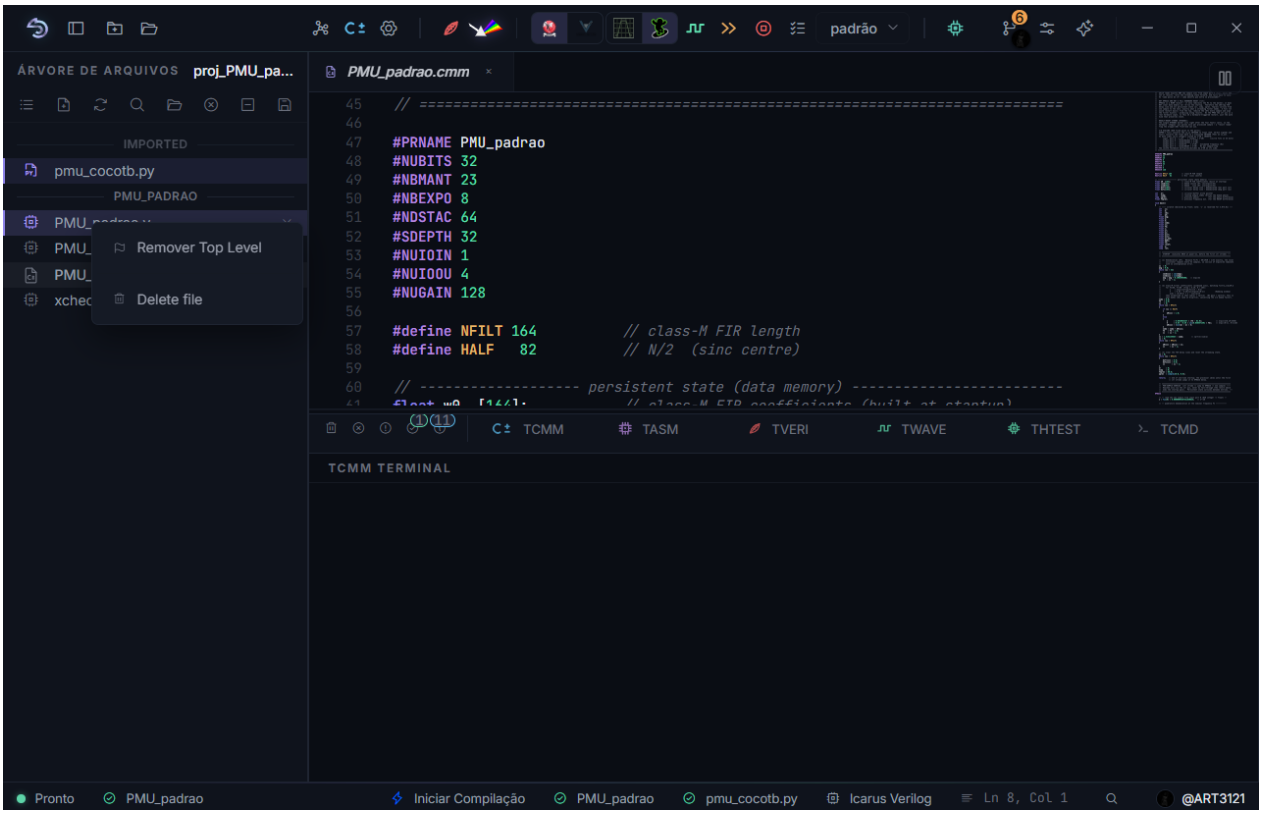


Figura1: No projeto `proj_PMU_padrao`, clique com o botão direito em `PMU_padrao.v` e use a ação de Top Level. A bandeira na árvore e o nome na barra de status confirmam a seleção; quando o arquivo já está selecionado, o menu oferece **Remover Top Level**.

8.4 Definir o Testbench Top

Escolha o testbench que será executado, clique nele com o botão direito do mouse e selecione **Marcar como Testbench**. Essa ação define o Testbench Top usado pela simulação.

- Arquivos `.v` usam testbench Verilog.
- Arquivos `.py` usam cocotb.

O Testbench Top é necessário para **Analisar Verilog (forma de onda)** e outras ações de simulação.

O arquivo selecionado deve conseguir instanciar o circuito e gerar os estímulos do teste. Em cocotb, o arquivo Python contém o módulo de teste, enquanto o Top Level continua apontando para o circuito Verilog.

8.5 Usar a vista de hierarquia

Depois de uma análise bem-sucedida, alterne a árvore para a vista de hierarquia. Ela mostra como os módulos se relacionam e ajuda a localizar instâncias. Use essa vista para verificar se todas as dependências esperadas foram encontradas.

A vista de hierarquia não altera os arquivos selecionados no projeto.

8.6 Corrigir arquivos ausentes

Quando uma referência não existe mais:

1. Restaure o arquivo no caminho anterior.
2. Remova a referência antiga e importe a nova localização.

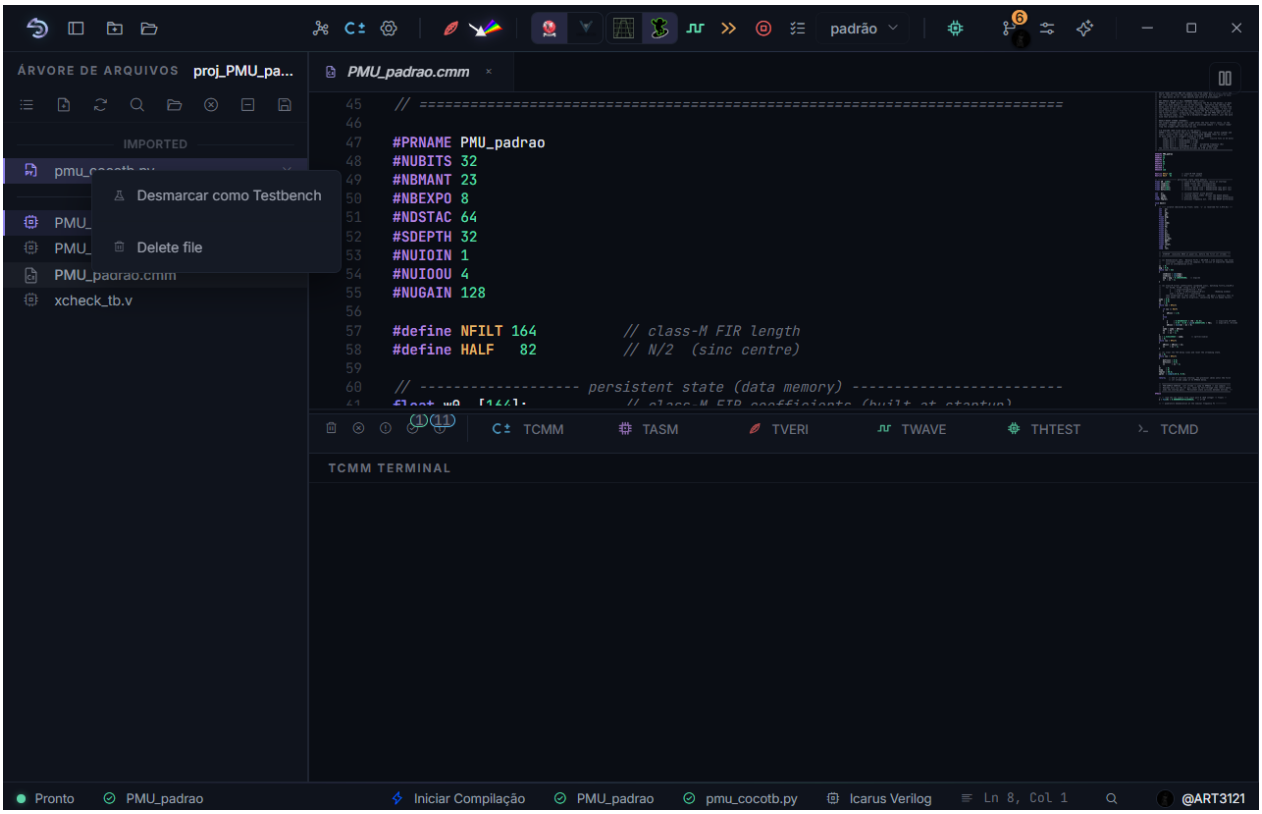


Figura2: No projeto `proj_PMU_padrao`, use **Marcar como Testbench** no arquivo `.v` ou `.py` que será executado. O ícone de frasco identifica o testbench; quando ele já está selecionado, o menu oferece **Desmarcar como Testbench**.

8.7 Evitar erros comuns

- Não inclua duas cópias que declarem o mesmo módulo.
- Mantenha todas as dependências do Top Level selecionadas.
- Defina Top Level e Testbench Top antes de simular.
- Valide com **Verilog** antes de abrir o PRISM.

Quando houver erro de módulo desconhecido, confira primeiro se o arquivo que declara esse módulo foi adicionado como fonte sintetizável. Quando houver declaração duplicada, procure cópias do mesmo módulo em arquivos diferentes.

Para praticar essas seleções com circuitos completos, consulte *Galeria de projetos e testbenches*. A galeria contém fontes Verilog e C± acompanhadas por testbenches Verilog e cocotb equivalentes.

Capítulo 9

Source Control e Git

O painel de **Controle de Versão** integra as operações Git mais frequentes ao projeto aberto na AURORA. Ele permite iniciar um repositório, revisar alterações, criar commits, sincronizar com o GitHub e clonar projetos associados à conta conectada.

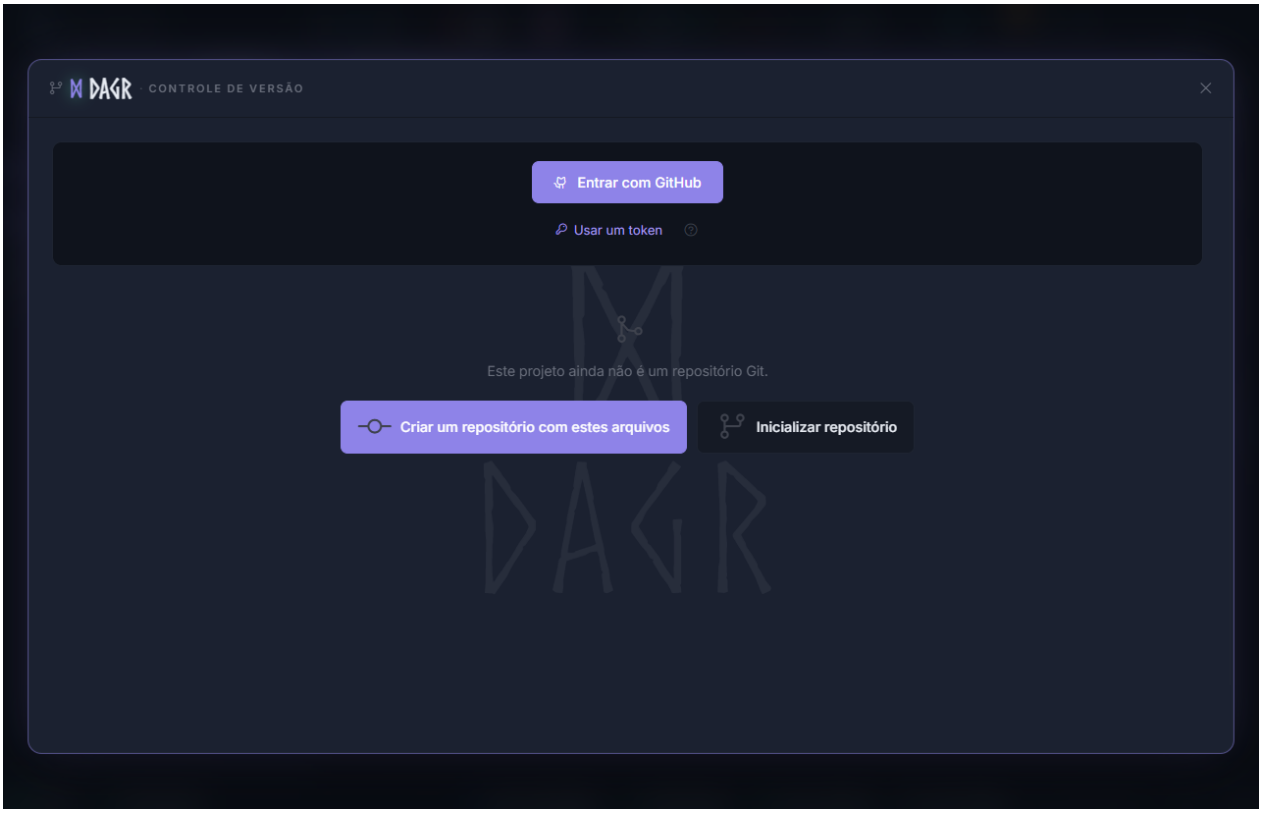


Figura1: Sem uma conta conectada e antes da inicialização local, o painel oferece **Criar um repositório com estes arquivos** e **Inicializar repositório**, sem exibir nomes, avatares ou repositórios pessoais.

9.1 Começar com o projeto aberto

Quando a pasta do projeto ainda não contém um repositório Git, o painel oferece duas ações:

Criar um repositório com estes arquivos
: Inicializa o Git na pasta do projeto, prepara os arquivos, cria o commit inicial e conduz a publicação em um novo repositório remoto quando a conta do GitHub está conectada.

Inicializar repositório
: Cria apenas o repositório Git local. Use esta opção para versionar o projeto sem publicá-lo imediatamente ou para configurar o remoto depois.

O repositório usa a pasta que contém o `.spf`; portanto, fontes, testbenches e demais arquivos do projeto podem ser acompanhados em conjunto. Revise a lista antes do primeiro commit para evitar incluir resultados volumosos ou credenciais.

9.2 Alterações e commits

Na aba **Alterações**, a AURORA separa arquivos modificados, novos e preparados. Clique em um arquivo para abrir o diff e use as caixas de seleção para controlar o próximo commit.

Controle	O que faz
Preparar / Tirar do stage	Inclui ou remove um arquivo do próximo commit
Preparar tudo / Despreparar tudo	Aplica a seleção a todos os arquivos listados
Resumo do commit	Define a mensagem que identifica a alteração
Commit	Registra os arquivos preparados no histórico local
Amend	Atualiza o commit mais recente com a seleção e a mensagem atuais
Desfazer último commit	Retorna o último commit para alterações locais, conforme a confirmação exibida
Atualizar	Recarrega status, branch e diferenças do repositório

Use a aba **Histórico** para consultar commits, autores e arquivos alterados. Ao abrir um commit, a interface carrega o diff de cada arquivo sob demanda.

9.3 Sincronização e branches

Controle	O que faz
Fetch	Consulta o estado do remoto sem integrar as mudanças
Pull	Traz e integra os commits do remoto à branch atual
Push	Envia os commits locais ao remoto configurado
Publicar	Cria ou associa o repositório remoto quando ele ainda não existe
Branches	Troca de branch, cria uma branch, acompanha branches remotas e faz merge
Alterações guardadas	Restaura ou descarta um stash criado durante uma troca de branch

Se uma troca de branch sobrescreveria alterações locais, a AURORA pode guardá-las em stash antes da troca. Leia as confirmações antes de merge, restauração ou descarte, pois essas operações alteram o conteúdo do projeto no disco.

9.4 Conectar e clonar do GitHub

Conecte uma conta do GitHub no próprio painel para listar os repositórios aos quais ela tem acesso. Na aba **Clonar**:

1. Escolha um repositório da conta ou das organizações disponíveis.
2. Selecione a pasta de destino.
3. Clique em **Clonar**.
4. Aguarde a AURORA procurar arquivos `.spf` no conteúdo baixado.
5. Abra o projeto encontrado ou consulte-o em **Projetos**.

A lista **Projetos** oferece ações para abrir o projeto na AURORA, acessar o repositório no GitHub, abrir o terminal, mostrar a pasta no Explorer ou remover a entrada da lista. Remover a entrada não deve ser confundido com excluir o repositório do disco; leia a confirmação apresentada pela interface.

A integração pode usar token armazenado de forma segura ou OAuth, conforme a configuração da instalação. Nunca inclua tokens em arquivos do projeto, capturas de tela ou commits.

9.5 Usar `.inv` sem alterar o Git

O `.inv` é um filtro exclusivo da visualização **Pastas**. Ele não remove arquivos, não muda o `.spf`, não altera o status Git e não substitui `.gitignore`.

O exemplo abaixo foi aplicado ao projeto `porta_AND`:

```
# Oculta artefatos locais apenas na visualização Pastas da AURORA.
__pycache__/
testbench/
arquivo.inv
```

Figura2: À esquerda, pastas auxiliares e `arquivo.inv` aparecem na árvore. À direita, depois de salvar o `.inv`, permanecem somente os arquivos não filtrados.

Use `.inv` para reduzir o ruído visual na AURORA e `.gitignore` para impedir que o Git rastreie arquivos gerados, temporários ou locais. Um item escondido por `.inv` ainda pode aparecer normalmente no painel de **Controle de Versão**.

9.6 Cuidados

- Revise o diff antes de preparar ou confirmar arquivos.
- Salve as abas abertas antes de trocar de branch.
- Faça `Fetch` antes de sincronizar um projeto compartilhado.
- Não publique saídas geradas ou credenciais sem necessidade.
- Preserve uma cópia do projeto antes de operações estruturais importantes.

Parte III

Fluxos principais

Capítulo 10

Escolha o fluxo de trabalho

A AURORA atende a dois fluxos principais. Você pode desenvolver um projeto diretamente em Verilog ou gerar um processador dedicado do ecossistema SAPHO a partir de um algoritmo C±, armazenado em arquivo `.cmm`. Os dois caminhos usam o mesmo editor, a mesma estrutura de projeto e as mesmas ferramentas de validação e análise, mas começam de pontos diferentes.

Fluxo Verilog Crie ou importe módulos RTL, defina o Top Level, adicione um testbench e siga diretamente para validação, simulação, análise de ondas e PRISM.

Fluxo completo para projetos Verilog Fluxo de processador SAPHO Configure um processador, escreva o algoritmo C±, gere o hardware Verilog com o YANC e depois simule ou visualize o RTL produzido.

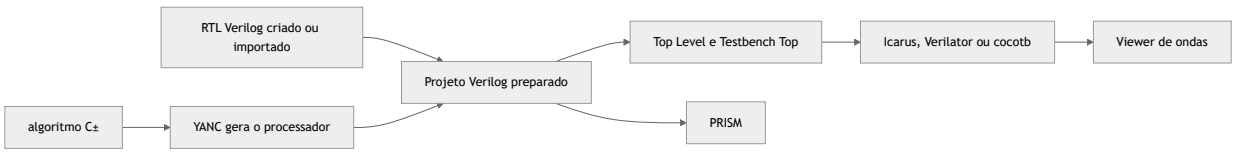
Fluxo completo para processadores SAPHO

10.1 Comparação rápida

Etapa	Fluxo Verilog	Fluxo de processador SAPHO
Entrada principal	Módulos `.v` ou `.sv`	algoritmo C± em arquivo `.cmm`
Geração inicial	Escrita ou importação do RTL	Hub de Processadores e compilação C±
Hardware usado	RTL fornecido pelo usuário	RTL gerado na pasta `Hardware`
Top Level	Módulo principal do projeto	Módulo gerado do processador ou módulo que o instancia
Testbench Top	Testbench `.v` ou cocotb `.py`	Testbench gerado, personalizado `.v` ou cocotb `.py`
Validação	Botão **Verilog**	Botão **C±**, seguido da validação Verilog quando necessária
Simulação	**Analisar Verilog (forma de onda)** ou **Execução rápida**	**Analisar Verilog (forma de onda)**, **Execução rápida** ou **Teste do processador sintetizado**
Análise	Viewer de ondas e PRISM	Viewer de ondas e PRISM

10.2 Etapas compartilhadas

Depois que o hardware Verilog está disponível, os dois fluxos convergem:



Em ambos os caminhos, o **Top Level** identifica o módulo principal do circuito e o **Testbench Top** identifica o teste que controla a simulação. A diferença está na origem do RTL: escrito ou importado no fluxo Verilog; gerado a partir de C± no fluxo SAPHO.

10.3 Qual fluxo escolher

Escolha *Fluxo completo para projetos Verilog* quando já possui módulos RTL, está aprendendo Verilog ou deseja testar um circuito digital sem gerar um processador SAPHO.

Escolha *Fluxo completo para processadores SAPHO* quando deseja transformar um algoritmo C± em uma arquitetura dedicada, configurar largura numérica e portas de entrada e saída ou trabalhar com o fluxo de geração de processadores do ecossistema SAPHO.

As páginas *Compilar Verilog ou gerar um processador SAPHO*, *Simular com Icarus, Verilator ou cocotb*, *Formas de onda, GTKWave e Surfer* e *Visualizar e simular o RTL no PRISM* detalham as ferramentas compartilhadas depois da preparação inicial.

Fluxo completo para projetos Verilog

Este roteiro apresenta a AURORA como ambiente de desenvolvimento Verilog independente do fluxo C±. Ao final, você terá um projeto com fontes RTL, Top Level, testbench, validação, simulação, formas de onda e visualização no PRISM.

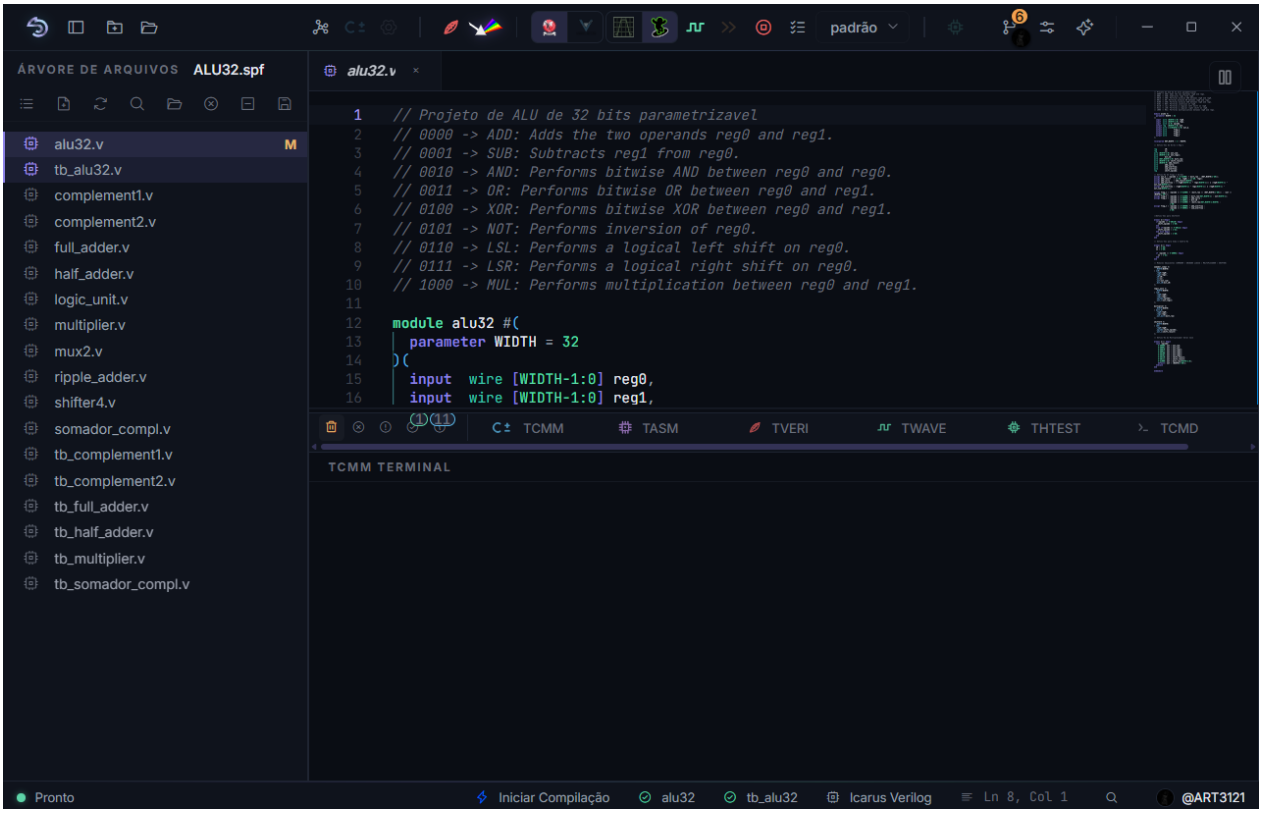


Figura1: O projeto `ALU32` aberto em `alu32.v`, com os módulos RTL e os testbenches registrados na árvore **Arquivos**.

11.1 1. Criar ou abrir o projeto

1. Clique em **Novo Projeto**.
2. Informe um nome para o projeto.
3. Escolha uma pasta com permissão de escrita.
4. Confirme a criação.

A AURORA cria a pasta do projeto e um arquivo `.spf`, usado para registrar os arquivos e as seleções do fluxo. Não é necessário criar um processador no **Hub de Processadores** para trabalhar com Verilog.

11.2 2. Adicionar os módulos RTL

Crie os arquivos no editor ou importe módulos existentes. Você também pode arrastar e soltar um ou vários arquivos sobre a árvore **Arquivos**. Esse recurso aceita módulos Verilog e SystemVerilog (`.v`, `.sv` e `.vh`) e testbenches cocotb (`.py`).

A AURORA classifica o conteúdo importado como fonte sintetizável ou testbench. Confirme a classificação na árvore e mantenha como fontes sintetizáveis todos os módulos que participam do circuito, inclusive os módulos instanciados pelo módulo principal.

Para um primeiro teste, você pode usar o circuito `porta_and.v` disponível na *Galeria de projetos e testbenches*.

11.3 3. Definir o Top Level

Na árvore **Arquivos**, localize o arquivo que contém o módulo principal. Clique nele com o botão direito do mouse e selecione **Definir como Top Level**.

O Top Level é a raiz usada para:

- Validar a elaboração do projeto.
- Gerar a visualização **Hierarquia** da árvore.
- Associar o circuito aos testbenches Verilog ou Python/cocotb.
- Gerar a visualização do PRISM.

A bandeira ao lado do arquivo e o nome exibido na barra de status confirmam a seleção. Se o módulo principal instancia outros módulos, todos eles também devem estar adicionados como fontes sintetizáveis.

11.4 4. Adicionar o testbench

Use uma das alternativas:

- Um testbench Verilog `.v`, que instancia o circuito, gera estímulos e pode incluir as verificações no próprio HDL.
- Um testbench Python `.py` com cocotb, que controla o Top Level pelo simulador.

Adicione o arquivo à árvore, clique nele com o botão direito do mouse e escolha **Marcar como Testbench**. Essa ação define o Testbench Top usado na simulação. O testbench fica separado das fontes sintetizáveis.

11.5 5. Validar o Verilog

1. Salve os arquivos.
2. Clique em **Compilar Verilog**.
3. Leia o terminal **Verilog**.
4. Corrija a primeira mensagem de erro, se houver.

A validação confirma sintaxe, módulos e conexões necessárias para elaborar o Top Level. Ela não substitui o testbench: um circuito pode ser elaborado corretamente e ainda produzir resultados funcionais incorretos.

11.6 6. Simular

1. Escolha Icarus Verilog ou Verilator. A **Execução rápida** com testbench Verilog exige Verilator; testbenches cocotb podem usar o simulador selecionado.
2. Confirme o Top Level e o Testbench Top na barra de status.
3. Use **Execução rápida** para executar o teste sem abrir formas de onda.

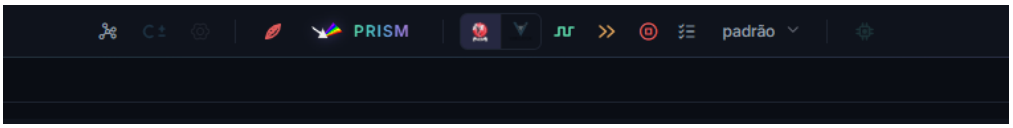


Figura2: Na barra superior, escolha o simulador e use **Execução rápida** ou **Analisar Verilog (forma de onda)** conforme o resultado desejado.

4. Use **Analisar Verilog (forma de onda)** para executar o Testbench Top, gerar a saída temporal e abrir o viewer selecionado.
5. Confirme no terminal se o testbench terminou e se todas as verificações passaram.

Icarus Verilog é uma boa escolha para a primeira execução e para observar sinais internos. Verilator é indicado quando a simulação exige mais desempenho e o projeto utiliza construções compatíveis.

11.7 7. Analisar formas de onda

Abra **Configuração de Ondas** para escolher os sinais e execute **Analisar Verilog (forma de onda)**. O viewer selecionado abre o VCD ou FST produzido pela simulação.

Use a forma de onda para conferir clock, reset, entradas, saídas e estados internos relevantes. A passagem do testbench continua sendo o critério funcional; a forma de onda ajuda a explicar por que um caso passou ou falhou.

11.8 8. Visualizar no PRISM

Com o Top Level definido e o Verilog válido, clique em **Abrir PRISM**. Use a visualização para conferir a hierarquia e as conexões estruturais do circuito.

O PRISM oferece o modo **Esquemático**, para inspecionar a estrutura, e o modo **Simular**, para alterar entradas lógicas e observar saídas diretamente. Essa interação não substitui um testbench nem a análise temporal no viewer de ondas.

11.9 Resultado esperado

O fluxo Verilog está concluído quando:

- Todos os módulos necessários estão registrados como fontes sintetizáveis.
- O Top Level e o Testbench Top aparecem corretamente na barra de status.
- A validação Verilog termina sem erro.
- O testbench Verilog ou cocotb conclui suas verificações.
- A análise de forma de onda abre os sinais esperados, quando solicitada.
- O PRISM apresenta a raiz correta do circuito.

Para exemplos completos em `.v` e cocotb, consulte *Galeria de projetos e testbenches*. Para detalhes de classificação dos arquivos, consulte *Arquivos Verilog e Testbenches*.

Capítulo 12

Fluxo completo para processadores SAPHO

Este roteiro apresenta a geração de hardware do ecossistema SAPHO. Ao final, você terá um processador configurado no **Hub de Processadores**, um algoritmo C± compilado, o Verilog correspondente e um teste executado na AURORA.

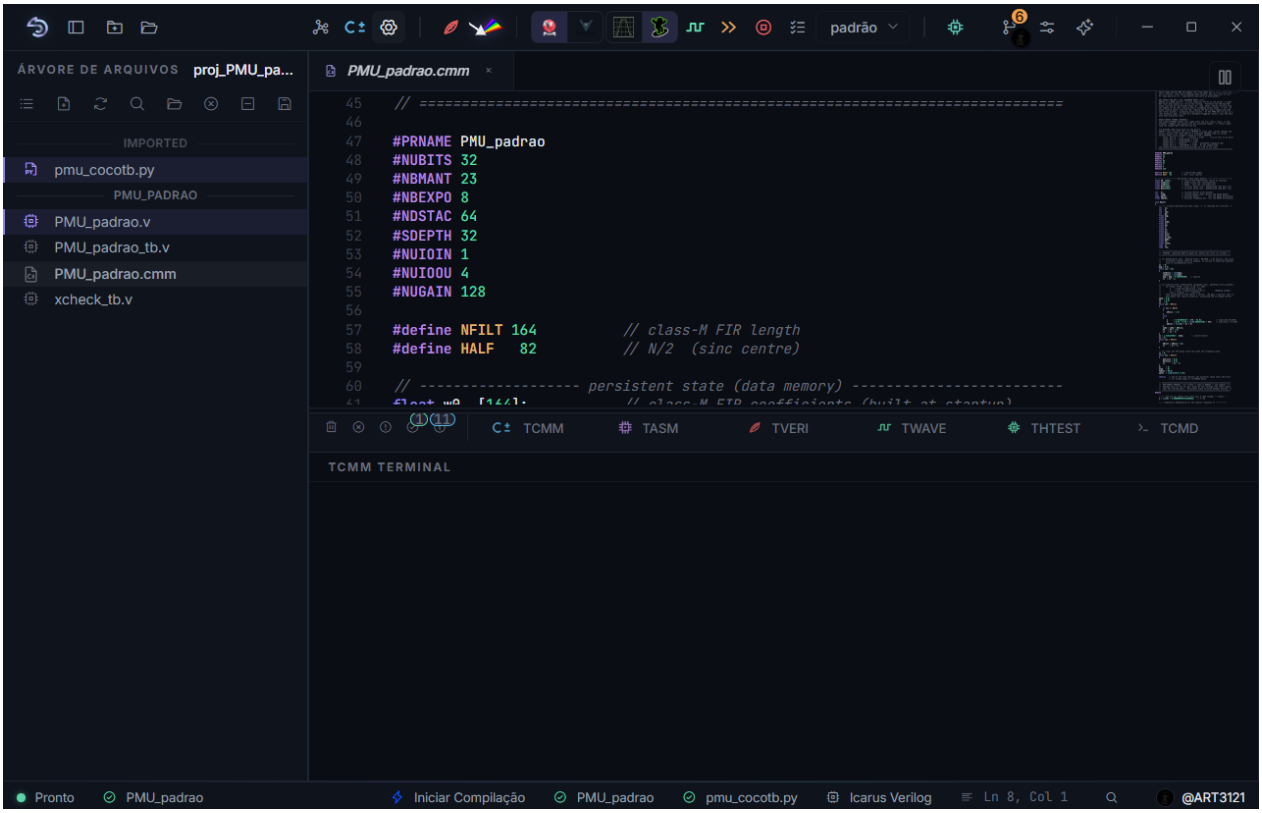


Figura1: O projeto `proj_PMU_padrao` aberto em `PMU_padrao.cmm`. Ao abrir o algoritmo C±, o processador `PMU_padrao` fica ativo e as ações relacionadas ao processador são habilitadas.

12.1 1. Criar ou abrir o projeto

1. Clique em **Novo Projeto**.
2. Informe um nome para o projeto.
3. Escolha uma pasta com permissão de escrita.
4. Confirme a criação.

O arquivo `.spf` criado pela AURORA registra os processadores, as fontes e as seleções do projeto. Não edite esse arquivo manualmente durante o fluxo normal.

12.2 2. Criar o processador

Abra **Hub de Processadores** e informe apenas o nome do processador. No primeiro projeto, mantenha os valores padrão já preenchidos para bits totais, ganho, mantissa, expoente, pilhas e portas de entrada e saída. Em seguida, clique em **Gerar Processador**.

Os padrões formam uma configuração válida e permitem praticar o fluxo antes de alterar a arquitetura. Para compreender cada campo e preparar configurações específicas, consulte *Processadores SAPHO* com o título **Processadores SAPHO**.

Depois da confirmação, o processador aparece na árvore com as pastas:

```
<processador>/
├── Software/
├── Hardware/
└── Simulation/
```

12.3 3. Escrever o algoritmo C±

Abra `Software/<processador>.cmm`, preserve as diretivas geradas e escreva o algoritmo dentro das funções do arquivo.

O arquivo `.cmm` define tanto o comportamento quanto os parâmetros usados na geração. Salve antes de compilar. Para testes que precisam identificar o término do algoritmo, use `#TOAQUI` no ponto adequado.

12.4 4. Gerar o hardware

1. Mantenha o arquivo `.cmm` aberto.
2. Clique em **Compilar C±**.
3. Acompanhe os terminais **C±** e **ASM**.
4. Aguarde a criação ou atualização dos arquivos na pasta `Hardware`.

O YANC transforma o algoritmo em Assembly SAPHO e depois gera o módulo Verilog, as imagens de memória e o testbench padrão. Um arquivo antigo em `Hardware` não comprova que a compilação atual passou; confirme o resultado nos terminais.

12.5 5. Preparar Top Level e testbench

O módulo em `Hardware/<processador>.v` pode ser usado como Top Level quando o projeto testa apenas esse processador. Clique nele com o botão direito do mouse e escolha **Definir como Top Level**. Em sistemas maiores, o Top Level pode ser outro módulo Verilog que instancia um ou mais processadores gerados.

Como Testbench Top, use:

- O testbench padrão criado na pasta `Simulation`.
- Um testbench Verilog personalizado.
- Um arquivo Python/cocotb com a diretiva `# aurora-toplevel: <processador>`.

Clique com o botão direito no testbench escolhido e selecione **Marcar como Testbench**. Confirme o Top Level e o Testbench Top na barra de status antes de simular.

12.6 6. Escolher a forma de execução

Teste do processador sintetizado
: Executa o processador ativo em um harness dedicado do Verilator. Use para validar entradas, saídas e término sem preparar uma inspeção completa no viewer de ondas. O progresso e o resultado aparecem no terminal **THTEST**.

Execução rápida
: Executa o Testbench Top sem abrir formas de onda. Use para verificações automatizadas e regressões curtas.

Analisar Verilog (forma de onda)
: Executa o Testbench Top, gera VCD ou FST e abre o viewer selecionado. Use quando precisa observar sinais, portas e estados internos.

12.7 7. Conferir entradas e saídas

Nos processadores gerados, `out` transporta o valor e `out_en` identifica a porta de saída escrita. O sinal `cheguei` indica que a execução alcançou o marcador `#TOAQUI`.

Um teste adequado deve verificar:

- Se cada porta recebeu os valores esperados.
- Se a quantidade e a ordem das saídas estão corretas.
- Se o processador alcançou o ponto de término.
- Se existe um limite de ciclos para evitar espera infinita.

12.8 8. Analisar o hardware gerado

Use **Analisar Verilog (forma de onda)** para observar o comportamento temporal e **Abrir PRISM** para examinar a estrutura RTL. O algoritmo C± continua sendo a fonte editável; os arquivos da pasta `Hardware` são resultados da geração e podem ser substituídos na próxima compilação.

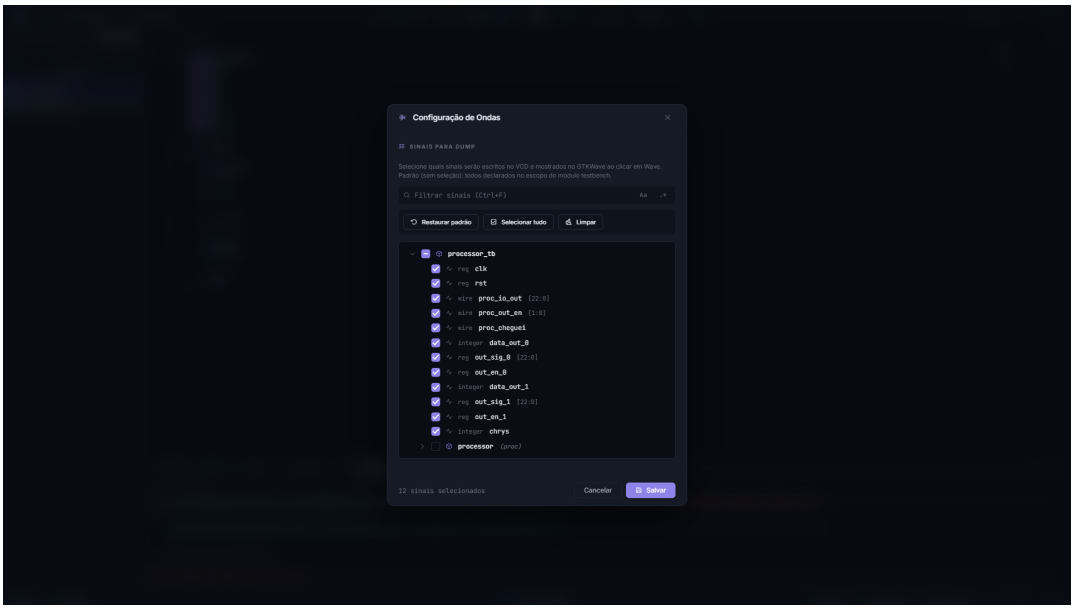


Figura2: No fluxo SAPHO, comece por `clk`, `rst`, `proc_io_out`, `proc_out_en` e `proc_cheguei`. Expanda o processador somente quando precisar investigar sinais internos.

12.9 Resultado esperado

O fluxo SAPHO está concluído quando:

- O Hub de Processadores criou a estrutura do processador.
- O algoritmo `.cmm` compila sem erro.
- O Verilog e os arquivos de memória são atualizados.
- O Top Level e o Testbench Top estão definidos.
- A execução produz as saídas esperadas e alcança o término.
- O viewer de ondas ou o PRISM apresenta o resultado desejado, quando utilizado.

Para compreender os campos do Hub de Processadores, consulte *Processadores SAPHO*. Para exemplos C± com testbenches Verilog e cocotb, consulte *Galeria de projetos e testbenches*.

Parte IV

Compilação, simulação e análise

Capítulo 13

Compilar Verilog ou gerar um processador SAPHO

Esta página explica as etapas de compilação compartilhadas pelos dois fluxos principais. Em um projeto Verilog, a validação verifica diretamente o RTL selecionado. Em um processador SAPHO, a geração C± produz primeiro o hardware Verilog que será usado nas etapas seguintes.

13.1 Antes de começar

Confirme qual fluxo está executando:

- No *Fluxo completo para projetos Verilog*, não é necessário possuir um processador ou arquivo `.cmm`.
- No *Fluxo completo para processadores SAPHO*, o processador e o arquivo `.cmm` ativo determinam o hardware que será regenerado.

Confirme:

1. O projeto está aberto.
2. Todos os arquivos modificados foram salvos.
3. O processador correto está ativo, se houver C±.
4. O Top Level está definido para validar Verilog.

Salvar os arquivos é indispensável porque os compiladores leem o conteúdo no disco. Confira também a barra de status para garantir que o processador e o Top Level pertencem ao fluxo que você deseja executar.

13.2 Gerar um processador C±

1. Abra o arquivo `.cmm` do processador.
2. Clique em **C±**.
3. Acompanhe os terminais de C± e Assembly.
4. Aguarde a conclusão antes de iniciar outra etapa.

O resultado esperado é a criação ou atualização dos arquivos na pasta `Hardware` do processador. Os terminais de C± e Assembly devem concluir sem erro. A existência de um arquivo antigo nessa pasta, por si só, não prova que a geração atual foi bem-sucedida.

Dica

Se você alterou apenas o algoritmo C±, não precisa executar manualmente cada compilador. O botão **C±** executa a sequência necessária.

13.3 Validar o projeto Verilog

1. Selecione todas as fontes sintetizáveis necessárias.
2. Defina o **Top Level**.
3. Clique em **Verilog**.
4. Leia o terminal Verilog.

A validação está correta quando termina sem erros de sintaxe, módulos ausentes ou módulos duplicados. Essa etapa não executa o comportamento do circuito; ela confirma que o Top Level e suas dependências formam um conjunto coerente para as etapas posteriores.

Se a validação falhar, corrija a primeira mensagem de erro e execute **Verilog** novamente. Um único módulo ausente pode produzir várias mensagens secundárias.

13.4 Ações que recompilam dependências

As ações abaixo preparam automaticamente o que precisam:

Ação	Resultado
C±	Atualiza o hardware do processador ativo
Verilog	Valida o Top Level e suas dependências
Analisar Verilog (forma de onda)	Compila e executa a simulação, depois abre as formas de onda.
Execução rápida	Compila e simula sem abrir o viewer de ondas
PRISM	Valida o RTL e gera a visualização

Projetos somente Verilog também são suportados; não é obrigatório criar um processador C±.

Executar uma ação mais completa não elimina a necessidade de ler os terminais. Por exemplo, **Analisar Verilog (forma de onda)** pode recompilar dependências, mas ainda falhar antes da simulação se o Verilog estiver inválido.

13.5 Interromper uma execução

Clique em **Cancelar**. A interrupção pode levar alguns segundos enquanto processos auxiliares são encerrados.

Não feche nem exclua arquivos temporários durante esse período.

Depois que o cancelamento terminar, revise a última mensagem do terminal. Se uma nova execução permanecer bloqueada, feche a ferramenta externa que ainda estiver aberta e consulte o diagnóstico do manual.

13.6 Erros comuns

Botão C± desabilitado
: Abra o `.cmm` do processador desejado.

Top Level não encontrado
: Confirme o módulo principal e inclua todas as dependências Verilog.

Módulo duplicado
Remova da seleção uma das cópias que declaram o mesmo módulo.

Arquivo gerado não mudou
Salve o `.cmm`, confirme o processador ativo e execute C± novamente.

Execução não termina
Use **Cancelar** e aguarde a limpeza. Se persistir, consulte *Solução de problemas*.

Simular com Icarus, Verilator ou cocotb

Uma simulação executa o circuito em um ambiente de teste e compara seu comportamento com o resultado esperado. Nesta página, você aprenderá a preparar o projeto, escolher o simulador e reconhecer quando o teste realmente foi concluído.

14.1 Preparar a simulação

Antes de clicar em **Analisar Verilog (forma de onda)** ou **Execução rápida**:

1. Selecione as fontes Verilog do projeto.
2. Defina o **Top Level**.
3. Defina o **Testbench Top**.
4. Salve os arquivos.
5. Selecione Icarus ou Verilator na barra superior quando o fluxo depender dessa opção.

O Top Level identifica o circuito; o Testbench Top identifica o teste. Antes de continuar, confira os dois nomes na barra de status e verifique se o testbench referencia sinais que existem na versão atual do RTL.

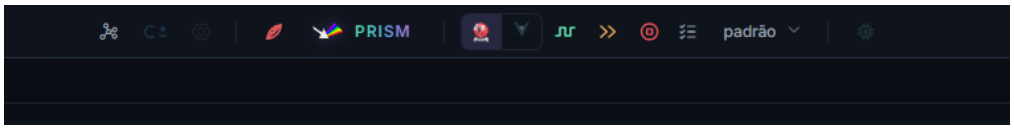


Figura1: Os logotipos selecionam o simulador usado pela análise de forma de onda e por testbenches cocotb. O ícone de onda executa **Analisar Verilog (forma de onda)**, os dois chevrons executam **Execução rápida** e a lista abre a configuração dos sinais.

14.2 Escolher o simulador

Opção	Indicação	Característica principal	Atenção
Icarus Verilog	Primeira execução, testes curtos e investigação de sinais	Simula diretamente o HDL e oferece boa compatibilidade com testbenches Verilog tradicionais	Pode ser mais lento em testes extensos
Verilator	Simulações longas ou projetos maiores	Compila o modelo para obter maior desempenho	Nem toda construção de testbench aceita pelo Icarus é compatível
cocotb	Testes automatizados escritos em Python	Oferece corrotinas, asserções e acesso às portas do circuito	Não é um terceiro simulador: usa Icarus ou Verilator como backend

Comece com Icarus para validar a estrutura básica e observar sinais. Troque para Verilator quando a duração ou o tamanho do projeto exigir mais desempenho. Escolha cocotb quando o teste se beneficiar de Python, automação e asserções legíveis, selecionando antes o backend desejado na barra superior.

A **Execução rápida** com testbench Verilog exige Verilator. Com testbench `.py`, a AURORA executa cocotb em modo headless usando Icarus ou Verilator, conforme a seleção. Como os backends não aceitam necessariamente as mesmas construções, avalie uma falha pelo terminal da opção selecionada.

14.3 Análise de forma de onda ou execução rápida

Analisar Verilog (forma de onda)
: Executa o testbench, gera formas de onda e abre o viewer selecionado.

Execução rápida
: Executa sem abrir viewer de ondas. Com testbench Verilog, usa Verilator em modo headless. Com testbench `.py`, executa cocotb em modo headless com o backend selecionado.

Use **Analisar Verilog (forma de onda)** quando precisar observar a evolução dos sinais. Use **Execução rápida** para testes automatizados, mensagens no terminal ou verificações que já possuem `assert` e não dependem de inspeção visual.

14.4 Testbench Verilog

Selecione um arquivo `.v` como Testbench Top. O testbench deve:

- Instanciar o módulo testado.
- Gerar clock e reset quando necessários.
- Aplicar estímulos.
- Encerrar a execução.
- Produzir dump de sinais ou permitir que a AURORA o configure.

O teste deve possuir uma condição clara de encerramento. Um testbench que gera clock indefinidamente e nunca chama sua condição de término continuará executando até ser cancelado.

O exemplo abaixo testa uma porta AND. O módulo do circuito deve ser adicionado como fonte sintetizável e definido como **Top Level**.

```
module porta_and (
    input  wire a,
    input  wire b,
    output wire y
);
    assign y = a & b;
endmodule
```

O arquivo seguinte deve ser adicionado como testbench e definido como **Testbench Top**.

```
`timescale 1ns/1ps

module tb_porta_and;
    reg a;
    reg b;
    wire y;

    porta_and dut (
        .a(a),
        .b(b),
        .y(y)
    );
```

(continua na próxima página)

(continuação da página anterior)

```

    task verificar;
        input valor_a;
        input valor_b;
        input esperado;
        begin
            a = valor_a;
            b = valor_b;
            #1;

            if (y !== esperado) begin
                $display(
                    "ERRO: a=%0b b=%0b esperado=%0b obtido=%0b",
                    a, b, esperado, y
                );
                $fatal(1);
            end
        end
    endtask

    initial begin
        $dumpfile("tb_porta_and.vcd");
        $dumpvars(0, tb_porta_and);

        verificar(1'b0, 1'b0, 1'b0);
        verificar(1'b0, 1'b1, 1'b0);
        verificar(1'b1, 1'b0, 1'b0);
        verificar(1'b1, 1'b1, 1'b1);

        $display("SUCESSO: tabela verdade validada.");
        $finish;
    end
endmodule
```

O testbench instancia o circuito com o nome `dut`, aplica os quatro casos da tabela-verdade e compara cada saída. Quando um valor está incorreto, `$fatal(1)` interrompe a simulação como falha. Quando todos os casos passam, `$finish` encerra normalmente. As chamadas `$dumpfile` e `$dumpvars` criam o registro usado na visualização das formas de onda.

14.5 Testbench cocotb

Selecione um arquivo `.py` como Testbench Top. A AURORA usa o ambiente Python incluído na toolchain; você não precisa configurar o Python global do Windows.

O mesmo circuito pode ser verificado com o testbench cocotb abaixo:

```
# aurora-toplevel: porta_and

import cocotb
from cocotb.triggers import Timer


async def verificar(dut, valor_a, valor_b, esperado):
    dut.a.value = valor_a
```

(continua na próxima página)

Figura2: No projeto `porta_AND`, o testbench `test_porta_and.py` aparece na mesma árvore da fonte `porta_and.v` e é identificado pelo ícone Python.

(continuação da página anterior)

```
    dut.b.value = valor_b
    await Timer(1, unit="ns")

    obtido = int(dut.y.value)
    assert obtido == esperado, (
        f"a={valor_a} b={valor_b}: esperado={esperado}, obtido={obtido}"
    )


@cocotb.test()
async def testa_tabela_verdade(dut):
    casos = [
        (0, 0, 0),
        (0, 1, 0),
        (1, 0, 0),
        (1, 1, 1),
    ]

    for valor_a, valor_b, esperado in casos:
        await verificar(dut, valor_a, valor_b, esperado)
```

O exemplo apresenta as partes essenciais de um teste cocotb:

- A diretiva `# aurora-toplevel: porta_and` informa explicitamente qual módulo Verilog será elaborado como DUT.
- O decorador `@cocotb.test()` registra a corrotina como um teste executável.
- O parâmetro `dut` representa o módulo Verilog e expõe suas portas pelos nomes `a`, `b` e `y`.
- `Timer(1, unit="ns")` permite que a lógica combinacional atualize a saída antes da leitura.
- A asserção compara resultado esperado e resultado obtido e inclui o caso completo na mensagem de erro.

Mantenha o nome do arquivo Python compatível com um módulo Python, por exemplo, `test_porta_and.py`. Evite espaços, hífens e caracteres especiais. Se um sinal não existir com o nome usado no código, o teste falhará antes de concluir as verificações.

Para circuitos com clock, use `Clock`, aguarde `RisingEdge` e leia os sinais após `ReadOnly`. A galeria apresenta esse padrão em um contador e também mostra como testar o Verilog gerado por algoritmos C±.

Consulte *Galeria de projetos e testbenches* para baixar todos os arquivos e comparar testbenches `.v` e `.py` equivalentes.

14.6 Confirmar o resultado

A simulação está correta quando:

- O terminal termina sem erro.
- O testbench alcança o final esperado.
- As verificações do teste passam.
- **Analisar Verilog (forma de onda)** abre o arquivo de onda correto.

Uma execução sem mensagens de erro não é suficiente quando o testbench deveria verificar valores. Confirme que as asserções foram alcançadas, que o tempo simulado avançou e que a mensagem final corresponde ao teste executado.

14.7 Problemas comuns

Módulo principal não encontrado
: Revise Top Level e arquivos selecionados.

Testbench não inicia
: Confirme Testbench Top, clock, reset e nome do módulo.

O cocotb não encontra o teste
: Confirme o arquivo `.py`, o nome da função decorada e os sinais usados.

Verilator falha, mas Icarus funciona
: O projeto pode usar uma construção não aceita pelo Verilator. Leia a primeira mensagem de erro no terminal.

Nenhuma forma de onda foi criada
: Siga *Formas de onda, GTKWave e Surfer*.

Capítulo 15

Galeria de projetos e testbenches

Esta galeria reúne exemplos pequenos para praticar o fluxo completo de simulação na AURORA. Cada circuito ou algoritmo possui duas alternativas de teste: um testbench Verilog (`.v`) e um testbench cocotb (`.py`). Use apenas uma alternativa como **Testbench Top** em cada execução.

15.1 Como usar os exemplos

Para um exemplo Verilog:

1. Baixe ou copie os três arquivos do conjunto para a pasta do projeto.
2. Adicione o arquivo do circuito como fonte sintetizável.
3. Defina o arquivo do circuito como **Top Level**.
4. Adicione `tb_*.v` ou `test_*.py` como testbench.
5. Defina o testbench escolhido como **Testbench Top**.
6. Execute **Analisar Verilog (forma de onda)** para abrir as formas de onda ou **Execução rápida** para conferir apenas as verificações.

Para um exemplo C±:

1. Crie no **Hub de Processadores** um processador com o mesmo nome indicado por `#PRNAME`.
2. Substitua o conteúdo do arquivo em `Software` pelo exemplo `.cmm`.
3. Execute **C±** para gerar o módulo em `Hardware/<nome>.v` e os arquivos de memória necessários.
4. Adicione o testbench `.v` ou `.py` correspondente ao projeto.
5. Defina `Hardware/<nome>.v` como **Top Level** e o teste escolhido como **Testbench Top**.
6. Execute **Analisar Verilog (forma de onda)** ou **Execução rápida**.

> **Importante**
>
> Os testbenches dos exemplos C± exercitam o Verilog gerado pelo compilador. Compile o `.cmm` antes da simulação e mantenha os arquivos gerados pela AURORA no projeto. O arquivo Python não executa C± diretamente.

15.2 Porta AND em Verilog

Este primeiro conjunto cobre um circuito combinacional. Os quatro pares possíveis de entrada são aplicados, e cada saída é comparada com a tabela-verdade da operação AND.

Arquivos: `porta_and.v`, `tb_porta_and.v` e `test_porta_and.py`.

15.2.1 Circuito `porta_and.v`

```
module porta_and (
    input  wire a,
    input  wire b,
```

(continua na próxima página)

(continuação da página anterior)

```
    output wire y
);
    assign y = a & b;
endmodule
```

15.2.2 Testbench Verilog `tb_porta_and.v`

```
`timescale 1ns/1ps

module tb_porta_and;
    reg a;
    reg b;
    wire y;

    porta_and dut (
        .a(a),
        .b(b),
        .y(y)
    );

    task verificar;
        input valor_a;
        input valor_b;
        input esperado;
        begin
            a = valor_a;
            b = valor_b;
            #1;

            if (y !== esperado) begin
                $display(
                    "ERRO: a=%0b b=%0b esperado=%0b obtido=%0b",
                    a, b, esperado, y
                );
                $fatal(1);
            end
        end
    endtask

    initial begin
        $dumpfile("tb_porta_and.vcd");
        $dumpvars(0, tb_porta_and);

        verificar(1'b0, 1'b0, 1'b0);
        verificar(1'b0, 1'b1, 1'b0);
        verificar(1'b1, 1'b0, 1'b0);
        verificar(1'b1, 1'b1, 1'b1);

        $display("SUCESSO: tabela verdade validada.");
        $finish;
    end
endmodule
```

O bloco `task verificar` evita repetir a sequência de atribuição, espera e comparação. `$fatal(1)` encerra

a execução com falha assim que um caso produz uma saída incorreta. `$dumpfile` e `$dumpvars` registram os sinais para o viewer de ondas.

15.2.3 Testbench cocotb `test_porta_and.py`

```python
# aurora-toplevel: porta_and

import cocotb
from cocotb.triggers import Timer


async def verificar(dut, valor_a, valor_b, esperado):
    dut.a.value = valor_a
    dut.b.value = valor_b
    await Timer(1, unit="ns")

    obtido = int(dut.y.value)
    assert obtido == esperado, (
        f"a={valor_a} b={valor_b}: esperado={esperado}, obtido={obtido}"
    )


@cocotb.test()
async def testa_tabela_verdade(dut):
    casos = [
        (0, 0, 0),
        (0, 1, 0),
        (1, 0, 0),
        (1, 1, 1),
    ]

    for valor_a, valor_b, esperado in casos:
        await verificar(dut, valor_a, valor_b, esperado)
```

A diretiva `# aurora-toplevel: porta_and` seleciona explicitamente o módulo testado. A função auxiliar aplica um caso, aguarda a propagação combinacional e inclui entradas, valor esperado e valor obtido na mensagem da asserção.

15.3 Contador de 4 bits em Verilog

Este conjunto introduz clock, reset e controle de habilitação. O teste confirma que o reset zera o estado, que o contador avança a cada borda quando `enable` vale `1` e que o valor permanece estável quando `enable` vale `0`.

Arquivos: `contador4.v`, `tb_contador4.v` e `test_contador4.py`.

15.3.1 Circuito `contador4.v`

```verilog
module contador4 (
    input  wire       clk,
    input  wire       rst,
    input  wire       enable,
    output reg  [3:0] valor
);
    always @(posedge clk) begin
```

(continua na próxima página)

(continuação da página anterior)

```
        if (rst)
            valor <= 4'd0;
        else if (enable)
            valor <= valor + 4'd1;
    end
endmodule
```

15.3.2 Testbench Verilog `tb_contador4.v`

```
`timescale 1ns/1ps

module tb_contador4;
    reg clk = 1'b0;
    reg rst = 1'b1;
    reg enable = 1'b0;
    wire [3:0] valor;

    contador4 dut (
        .clk(clk),
        .rst(rst),
        .enable(enable),
        .valor(valor)
    );

    always #5 clk = ~clk;

    initial begin
        $dumpfile("tb_contador4.vcd");
        $dumpvars(0, tb_contador4);

        repeat (2) @(posedge clk);
        @(negedge clk);
        rst = 1'b0;
        enable = 1'b1;

        repeat (4) @(posedge clk);
        #1;
        if (valor !== 4'd4) begin
            $display("ERRO: esperado=4 obtido=%0d", valor);
            $fatal(1);
        end

        @(negedge clk);
        enable = 1'b0;
        repeat (2) @(posedge clk);
        #1;
        if (valor !== 4'd4) begin
            $display("ERRO: contador mudou com enable=0. obtido=%0d", valor);
            $fatal(1);
        end

        $display("SUCESSO: reset, contagem e enable validados.");
        $finish;
```

(continua na próxima página)

(continuação da página anterior)

```
    end
endmodule
```

O clock alterna a cada `5 ns`, portanto o período é de `10 ns`. O reset é retirado na borda de descida para estar estável antes da próxima borda de subida. O atraso `#1` após a borda permite observar o valor atualizado pela atribuição não bloqueante do contador.

15.3.3 Testbench cocotb `test_contador4.py`

```python
# aurora-toplevel: contador4

import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge, ReadOnly, RisingEdge


async def ler_apos_borda(dut):
    await RisingEdge(dut.clk)
    await ReadOnly()
    return int(dut.valor.value)


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

    dut.rst.value = 1
    dut.enable.value = 0
    await ler_apos_borda(dut)
    assert await ler_apos_borda(dut) == 0, "O reset nao zerou o contador."

    await FallingEdge(dut.clk)
    dut.rst.value = 0
    dut.enable.value = 1
    for esperado in range(1, 5):
        obtido = await ler_apos_borda(dut)
        assert obtido == esperado, (
            f"Contagem incorreta: esperado={esperado}, obtido={obtido}"
        )

    await FallingEdge(dut.clk)
    dut.enable.value = 0
    for _ in range(2):
        obtido = await ler_apos_borda(dut)
        assert obtido == 4, (
            f"O contador mudou com enable=0: esperado=4, obtido={obtido}"
        )
```

`Clock` gera o clock em uma corrotina separada. A função `ler_apos_borda` aguarda `RisingEdge` e depois `ReadOnly`, fase em que as atualizações daquele instante já podem ser lidas sem disputar com a lógica do simulador. As mudanças de `rst` e `enable` são feitas após `FallingEdge`, fora da fase somente de leitura e antes da próxima borda ativa.

15.4 Soma de constantes em C±

Este exemplo gera um processador chamado `soma_constantes`. O algoritmo soma `3 + 4`, escreve `7` na porta de saída `0` e alcança `#TOAQUI`, que permite ao hardware indicar o término pelo sinal `cheguei`.

Arquivos: `soma_constantes.cmm`, `tb_soma_constantes.v` e `test_soma_constantes.py`.

15.4.1 Algoritmo `soma_constantes.cmm`

```
#PRNAME soma_constantes
#NUBITS 32
#NBMANT 23
#NBEXPO 8
#NDSTAC 8
#SDEPTH 8
#NUIOIN 1
#NUIOOU 1
#NUGAIN 128

void main()
{
    int a = 3;
    int b = 4;
    int resultado;

    resultado = a + b;
    out(0, resultado);

    #TOAQUI
}
```

15.4.2 Testbench Verilog `tb_soma_constantes.v`

```
`timescale 1ns/1ps

module tb_soma_constantes;
    reg clk = 1'b0;
    reg rst = 1'b1;
    wire signed [31:0] out;
    wire [0:0] out_en;
    wire cheguei;

    integer ciclos = 0;
    integer recebeu_saida = 0;
    integer terminou = 0;

    soma_constantes dut (
        .clk(clk),
        .rst(rst),
        .out(out),
        .out_en(out_en),
        .cheguei(cheguei)
    );

    always #5 clk = ~clk;
```

(continua na próxima página)

(continuação da página anterior)

```

    always @(posedge clk) begin
        if (!rst && (out_en == 1)) begin
            recebeu_saida = 1;
            if ($signed(out) !== 32'sd7) begin
                $display("ERRO: esperado=7 obtido=%0d", $signed(out));
                $fatal(1);
            end
        end

        if (cheguei === 1'b1)
            terminou = 1;
    end

    initial begin
        $dumpfile("tb_soma_constantes.vcd");
        $dumpvars(0, tb_soma_constantes);

        repeat (2) @(posedge clk);
        @(negedge clk);
        rst = 1'b0;

        while (!terminou && (ciclos < 200)) begin
            @(posedge clk);
            ciclos = ciclos + 1;
        end

        repeat (2) @(posedge clk);
        #1;

        if (!recebeu_saida) begin
            $display("ERRO: nenhuma escrita foi observada na porta 0.");
            $fatal(1);
        end

        if (!terminou) begin
            $display("ERRO: tempo limite antes de #TOAQUI.");
            $fatal(1);
        end

        $display("SUCESSO: saida 7 e termino do algoritmo validados.");
        $finish;
    end
endmodule
```

O bloco acionado na borda de subida observa `out_en == 1`, condição que identifica uma escrita na porta `0`. A captura ocorre na própria borda porque esse sinal funciona como um pulso de habilitação da escrita. O limite de `200` ciclos evita uma simulação infinita se o algoritmo não produzir a saída ou não alcançar `#TOAQUI`.

15.4.3 Testbench cocotb `test_soma_constantes.py`

```
# aurora-toplevel: soma_constantes

import cocotb
```

(continua na próxima página)

(continuação da página anterior)

```
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge, ReadOnly, RisingEdge


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

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

    saidas = []
    ciclos_apos_termino = 0
    for _ in range(200):
        await FallingEdge(dut.clk)
        await ReadOnly()

        if int(dut.out_en.value) == 1:
            saidas.append(int(dut.out.value))

        if int(dut.cheguei.value) == 1 and ciclos_apos_termino == 0:
            ciclos_apos_termino = 1
        elif ciclos_apos_termino:
            ciclos_apos_termino += 1

        if ciclos_apos_termino >= 3:
            break
    else:
        raise AssertionError("Tempo limite antes de o algoritmo atingir #TOAQUI.")

    assert saidas == [7], f"Esperado [7] na porta 0, obtido {saidas}."
```

O teste amostra `out_en`, `out` e `cheguei` na borda de descida do clock, quando os sinais produzidos na borda ativa já estão estáveis. Ele exige a lista exata `[7]`; dessa forma, detecta ausência de saída, valor incorreto e escritas adicionais inesperadas. O limite de ciclos impede uma espera infinita.

15.5 Sequência de quadrados em C±

Este exemplo usa um laço `while` para escrever `0`, `1`, `4` e `9` na porta `0`. Além do valor de cada amostra, os testes verificam a ordem e a quantidade de escritas.

Arquivos: `sequencia_quadrados.cmm`, `tb_sequencia_quadrados.v` e `test_sequencia_quadrados.py`.

15.5.1 Algoritmo `sequencia_quadrados.cmm`

```
#PRNAME sequencia_quadrados
#NUBITS 32
#NBMANT 23
#NBEXPO 8
#NDSTAC 8
#SDEPTH 8
```

(continua na próxima página)

(continuação da página anterior)

```
#NUIOIN 1
#NUIOOU 1
#NUGAIN 128

void main()
{
    int indice = 0;

    while (indice < 4) {
        out(0, indice * indice);
        indice = indice + 1;
    }

    #TOAQUI
}
```

15.5.2 Testbench Verilog `tb_sequencia_quadrados.v`

```
`timescale 1ns/1ps

module tb_sequencia_quadrados;
    reg clk = 1'b0;
    reg rst = 1'b1;
    wire signed [31:0] out;
    wire [0:0] out_en;
    wire cheguei;

    integer ciclos = 0;
    integer indice = 0;
    integer terminou = 0;
    integer esperado [0:3];

    sequencia_quadrados dut (
        .clk(clk),
        .rst(rst),
        .out(out),
        .out_en(out_en),
        .cheguei(cheguei)
    );

    always #5 clk = ~clk;

    always @(posedge clk) begin
        if (!rst && (out_en == 1)) begin
            if (indice > 3) begin
                $display("ERRO: o processador produziu saidas extras.");
                $fatal(1);
            end

            if ($signed(out) !== esperado[indice]) begin
                $display(
                    "ERRO: indice=%0d esperado=%0d obtido=%0d",
                    indice, esperado[indice], $signed(out)
```

(continua na próxima página)

(continuação da página anterior)

```
                );
                $fatal(1);
            end

            indice = indice + 1;
        end

        if (cheguei === 1'b1)
            terminou = 1;
    end

    initial begin
        esperado[0] = 0;
        esperado[1] = 1;
        esperado[2] = 4;
        esperado[3] = 9;

        $dumpfile("tb_sequencia_quadrados.vcd");
        $dumpvars(0, tb_sequencia_quadrados);

        repeat (2) @(posedge clk);
        @(negedge clk);
        rst = 1'b0;

        while (!terminou && (ciclos < 400)) begin
            @(posedge clk);
            ciclos = ciclos + 1;
        end

        repeat (2) @(posedge clk);
        #1;

        if (indice !== 4) begin
            $display("ERRO: esperadas=4 saidas_recebidas=%0d", indice);
            $fatal(1);
        end

        if (!terminou) begin
            $display("ERRO: tempo limite antes de #TOAQUI.");
            $fatal(1);
        end

        $display("SUCESSO: sequencia 0, 1, 4, 9 validada.");
        $finish;
    end
endmodule
```

O vetor `esperado` funciona como um modelo de referência simples. Cada pulso de `out_en` consome uma posição. O teste falha se houver valor incorreto, quantidade diferente de quatro saídas ou ausência do sinal de término.

15.5.3 Testbench cocotb `test_sequencia_quadrados.py`

```python
# aurora-toplevel: sequencia_quadrados

import cocotb
from cocotb.clock import Clock
from cocotb.triggers import FallingEdge, ReadOnly, RisingEdge


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

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

    saidas = []
    ciclos_apos_termino = 0
    for _ in range(400):
        await FallingEdge(dut.clk)
        await ReadOnly()

        if int(dut.out_en.value) == 1:
            saidas.append(int(dut.out.value))

        if int(dut.cheguei.value) == 1 and ciclos_apos_termino == 0:
            ciclos_apos_termino = 1
        elif ciclos_apos_termino:
            ciclos_apos_termino += 1

        if ciclos_apos_termino >= 3:
            break
    else:
        raise AssertionError("Tempo limite antes de o algoritmo atingir #TOAQUI.")

    assert saidas == [0, 1, 4, 9], (
        f"Esperado [0, 1, 4, 9] na porta 0, obtido {saidas}."
    )
```

A lista Python torna a comparação da sequência inteira direta. Esse padrão pode ser ampliado para filtros, máquinas de estados e processadores que produzam séries de amostras.

15.6 Escolher entre os dois tipos de testbench

Situação	Testbench recomendado
Aprender eventos e tarefas do Verilog	`.v`
Controlar diretamente o dump de sinais	`.v`
Percorrer muitos casos de entrada	`.py` com cocotb
Criar listas, modelos matemáticos e mensagens detalhadas	`.py` com cocotb
Depurar a integração inicial de um módulo	Comece com `.v` e reproduza em `.py` quando necessário

Os dois formatos verificam o mesmo hardware. A escolha altera a linguagem e a organização do teste, não o circuito sintetizável. Para o procedimento de execução e o diagnóstico de falhas, consulte *Simular com Icarus, Verilator ou cocotb*.

Formas de onda, GTKWave e Surfer

As formas de onda mostram como cada sinal muda ao longo do tempo simulado. Esta página explica como escolher sinais, gerar o arquivo de onda e reutilizar layouts no viewer selecionado sem confundir resultados de testbenches diferentes.

16.1 Gerar uma forma de onda

1. Defina o **Testbench Top**.
2. Escolha Icarus ou Verilator.
3. Escolha o viewer de waveform, quando quiser alternar entre GTKWave e Surfer.
4. Abra **Configuração de Ondas**.
5. Selecione os sinais desejados.
6. Clique em **Analisar Verilog (forma de onda)**.

A AURORA executa a simulação e abre o resultado no viewer selecionado. GTKWave é o viewer externo padrão. Surfer é uma opção opt-in; se o executável do Surfer não estiver disponível, a AURORA volta para GTKWave e avisa o usuário. Se a compilação ou o testbench falhar, o visualizador pode não ser aberto; nesse caso, o terminal Wave deve ser analisado antes da configuração visual.

16.2 Escolher o viewer

Use GTKWave quando quiser o fluxo tradicional com janela externa e layouts `.gtkw`. Use Surfer quando quiser usar layouts `.surf.ron` ou scripts `.sucl` e o viewer estiver disponível na instalação.

Viewer	Layouts	Observação
GTKWave	`.gtkw`	Padrão externo para VCD/FST.
Surfer	`.surf.ron` e `.sucl`	Opção opt-in; pode usar mappings em `%APPDATA%/surfer-project/surfer/config/mappings/`.

O arquivo de onda real continua sendo `.vcd` ou `.fst`. Layouts apenas organizam sinais e visualização.

16.3 Selecionar sinais

Na **Configuração de Ondas** você pode:

- Pesquisar por nome.
- Navegar pela hierarquia.
- Selecionar todos, nenhum ou o conjunto padrão.
- Filtrar sinais relacionados aos processadores.

Atalhos:

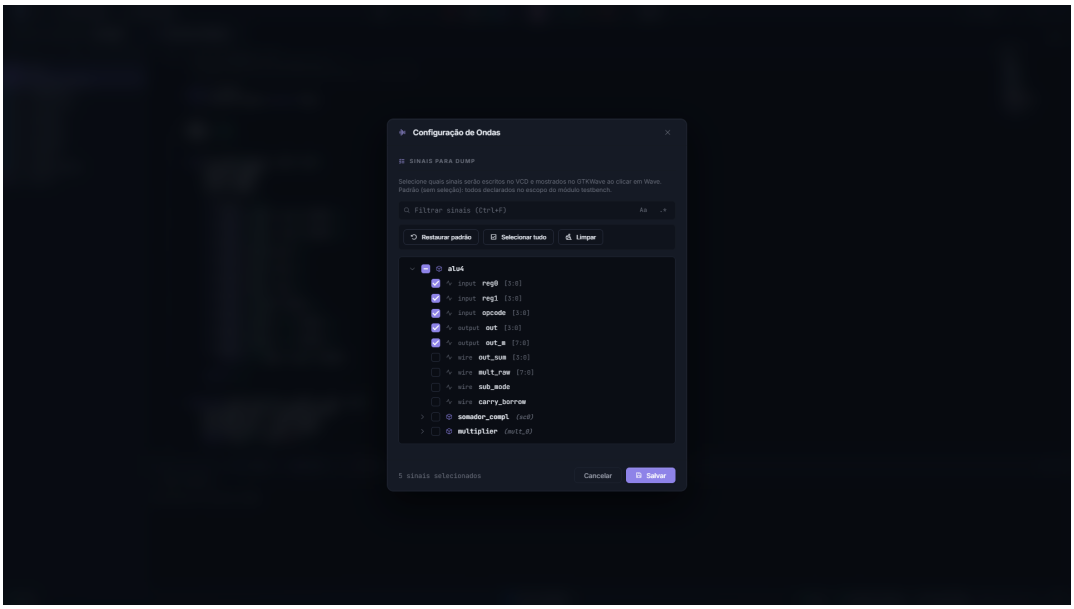


Figura1: Expanda a hierarquia, marque somente os sinais necessários e confira a quantidade selecionada antes de salvar.

Atalho	Ação
`Ctrl+F`	Focar a pesquisa.
`Esc`	Limpar a pesquisa; pressione novamente para fechar.

Selecione primeiro clock, reset, entradas, saídas e os poucos sinais internos necessários para responder à pergunta do teste. Um conjunto menor torna a leitura mais simples, reduz o tamanho do arquivo de onda e evita que detalhes sem relação ocultem o comportamento importante.

Depois da simulação, confira se os nomes exibidos no viewer pertencem à hierarquia esperada. Se o circuito foi renomeado ou reorganizado, seleções antigas podem deixar de corresponder ao RTL atual.

16.4 Testbench com `$dumpfile` e `$dumpvars`

Se o testbench já define o dump manualmente, a AURORA respeita essa configuração enquanto a **Configuração de Ondas** não tiver sido personalizada para esse testbench. Nesse caso, confirme se o nome do arquivo, o escopo passado a `$dumpvars` e os sinais gerados correspondem ao que deseja abrir. Uma configuração manual muito restrita pode produzir um arquivo válido, mas sem os sinais necessários para a análise.

Quando você salva uma seleção própria na **Configuração de Ondas** ou ativa um layout que dita sinais, a AURORA usa essa seleção como fonte do dump. Se houver `$dumpfile` ou `$dumpvars` manuais, eles são neutralizados apenas na cópia temporária usada para simular; o testbench original não é alterado.

16.5 Usar layouts de viewer

O seletor de layout acompanha o viewer ativo. Em GTKWave, ele trabalha com `.gtkw`. Em Surfer, ele trabalha com `.surf.ron` e `.sucl`.

O seletor permite:

- **Default**: Gerar uma organização automática.
- **Add**: Registrar um layout existente.
- **Remove**: Remover a referência do seletor.

Use apenas layouts criados para o mesmo testbench e para uma hierarquia compatível. Um layout organiza nomes de sinais; ele não contém a simulação. Um layout de outro projeto pode apontar para sinais inexistentes mesmo quando o arquivo VCD ou FST foi gerado corretamente.

16.6 Se o viewer abrir sem sinais

Verifique:

1. O testbench executou até o fim.
2. O terminal Wave não informou erro.
3. Existe dump de sinais.
4. Os sinais selecionados ainda existem no RTL.
5. O layout ativo pertence ao testbench atual.

Volte para **Default** e gere novamente para descartar um layout incompatível.

Se os sinais aparecerem, mas permanecerem sem mudança, verifique o clock, o reset, os estímulos e o intervalo de tempo mostrado. O problema pode estar no testbench ou no zoom da janela, e não na geração do arquivo.

16.7 Se o arquivo de onda não for encontrado

- Confira o nome usado em `$dumpfile`.
- Verifique se o testbench encerrou antes de produzir o arquivo.
- Remova arquivos de onda antigos que possam causar ambiguidade.
- Execute novamente e leia o terminal Wave.

Visualizar e simular o RTL no PRISM

O PRISM possui dois modos complementares. **Esquemático** apresenta a estrutura RTL sintetizada; **Simular** abre um circuito lógico interativo no qual entradas podem ser alteradas entre `0` e `1` e as saídas são atualizadas na tela.

17.1 Abrir o PRISM

1. Adicione todas as fontes sintetizáveis ao projeto.
2. Defina o **Top Level**.
3. Salve os arquivos.
4. Execute **Compilar Verilog** e corrija os erros mostrados no terminal.
5. Clique em **Abrir PRISM**.

O diagrama corresponde ao RTL salvo e processado naquele momento. Se o Top Level ou o conjunto de fontes mudar, confirme as seleções na janela principal e use **Recompilar**.

17.2 Projeto Verilog puro

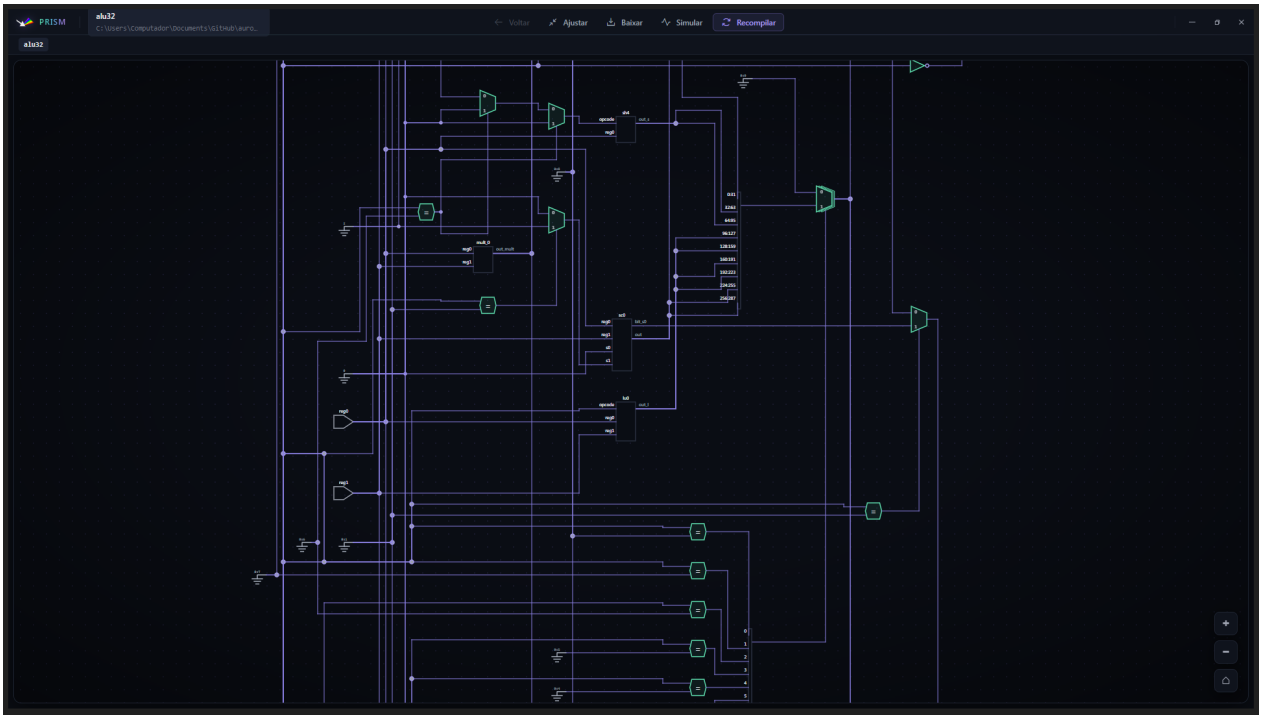


Figura1: O projeto `ALU32` mostra operações, multiplexadores, portas e conexões derivados diretamente do módulo `alu32.v`.

No modo **Esquemático**, use o zoom para aproximar ou afastar, arraste a área livre para navegar e abra instâncias navegáveis para inspecionar módulos internos. **Voltar** retorna ao nível anterior e **Ajustar** enquadra o diagrama na janela.

17.3 Processador SAPHO

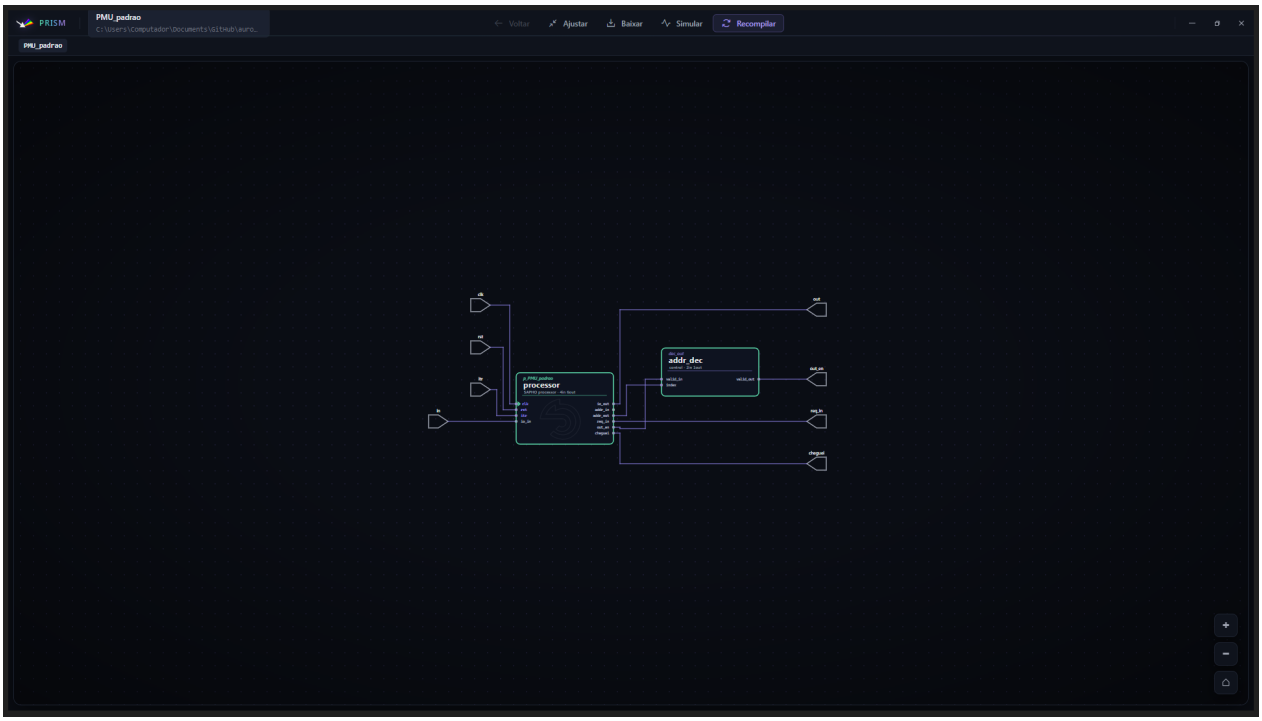


Figura2: No projeto `proj_PMU_padrao`, o processador SAPHO aparece com um ícone próprio no RTL, ao lado dos demais blocos e conexões do Top Level.

A aparência própria do bloco `processor` facilita reconhecer o núcleo SAPHO dentro de um projeto maior. O ícone representa uma instância do módulo; abra a instância quando quiser navegar pela estrutura interna disponível.

17.4 Controles da janela

Controle	O que faz
Voltar	Retorna ao módulo visualizado anteriormente
Ajustar	Centraliza e enquadra todo o circuito
Baixar	Exporta a visualização atual
Simular	Troca do esquemático para a simulação lógica interativa
Esquemático	Retorna da simulação interativa ao diagrama RTL
Recompilar	Processa novamente os arquivos salvos e atualiza a visualização

17.5 Simulação interativa

Para testar um circuito combinacional:

1. Abra o esquemático do Top Level no PRISM.
2. Clique em **Simular** e aguarde a construção do circuito interativo.
3. Clique nos controles de entrada para alternar cada valor entre `0` e `1`.
4. Observe as conexões e os valores das saídas atualizados na própria tela.
5. Clique em **Esquemático** para retornar à estrutura RTL.

Em circuitos sequenciais, a simulação pode disponibilizar controles adicionais de clock. Valores desconheci-

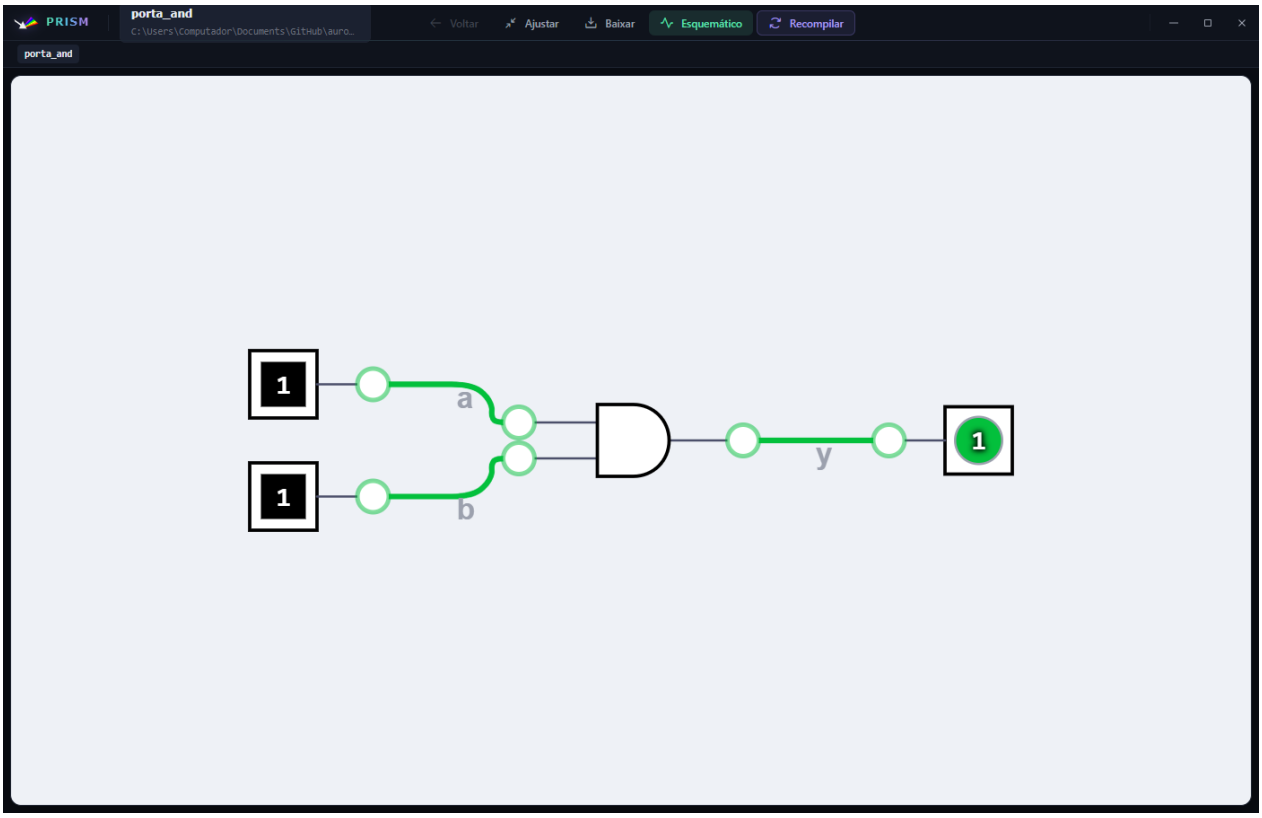


Figura3: Na simulação do projeto `porta_AND`, as entradas `a` e `b` estão em `1`; a saída `y` responde com `1` conforme a tabela-verdade da porta AND.

dos podem aparecer como `x` até que entradas, clock ou reset definam um estado válido.

17.6 O que o PRISM confirma

Use o modo **Esquemático** para conferir hierarquia, instâncias e conexões. Use **Simular** para experimentar estados lógicos diretamente, sobretudo em circuitos pequenos e combinacionais.

A simulação interativa não substitui um testbench Verilog ou cocotb, nem a análise temporal em formas de onda. Para verificar sequências extensas, temporização, clock e asserções automatizadas, use *Simular com Icarus, Verilator ou cocotb* e *Formas de onda, GTKWave e Surfer*.

17.7 Quando o PRISM falhar

Top Level incorreto
: Escolha o módulo raiz do projeto.

Módulo ausente
: Inclua o arquivo que declara a dependência.

Módulo duplicado
: Remova uma das fontes que declaram o mesmo nome.

Diagrama não é gerado
: Execute **Compilar Verilog** primeiro e corrija o erro mais inicial do terminal.

Uma instância não abre
: Ela pode ser uma primitiva, um módulo filtrado ou não possuir informação suficiente para navegação.

A simulação mostra x
: Defina as entradas e aplique reset ou clock quando o circuito exigir estado inicial.

Parte V

Aurora Intelligence

Aurora Intelligence

Aurora Intelligence é o assistente integrado ao projeto. Ele pode explicar arquivos, consultar o estado da IDE, ajudar no diagnóstico e, com sua confirmação, executar ações. Esta página apresenta um modo de uso seguro e previsível, no qual cada alteração é revisada e validada.

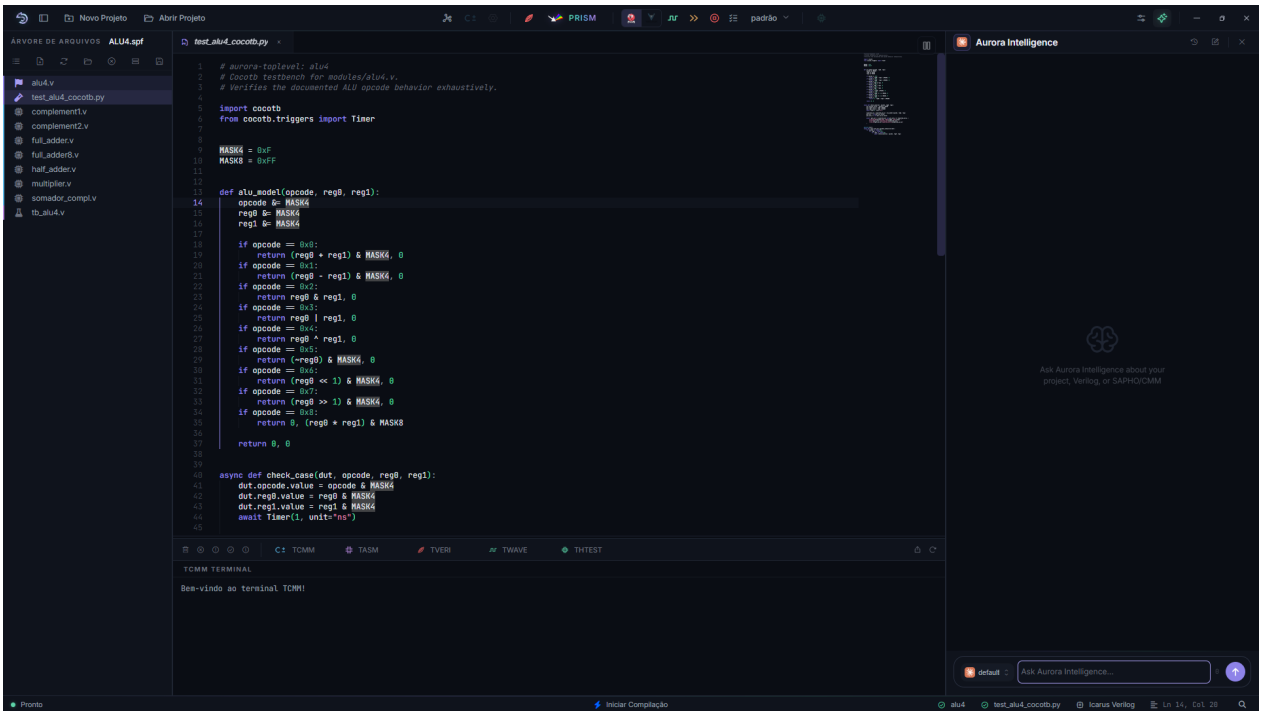


Figura1: O painel permanece ao lado do editor, permitindo formular o pedido enquanto o arquivo e o estado do projeto continuam visíveis.

18.1 O que você pode pedir

- Explicar um arquivo ou trecho selecionado.
- Localizar erros de compilação.
- Listar arquivos, processadores e sinais.
- Orientar a configuração do projeto.
- Criar ou editar arquivos.
- Compilar, simular e abrir ferramentas.
- Preparar uma sequência de diagnóstico.

O assistente trabalha melhor quando o pedido informa o objetivo, o arquivo e a etapa atual. Em vez de pedir apenas "corrija o projeto", indique o erro observado e peça que a primeira causa seja explicada antes de qualquer modificação.

18.2 Leitura e alteração

Pedidos de consulta podem ser executados diretamente. Ações que alteram arquivos, configurações ou executam fluxos apresentam uma confirmação.

Antes de aprovar, verifique:

1. Qual ação será executada.
2. Quais arquivos serão alterados.
3. Se há backup ou versionamento.
4. Se os argumentos correspondem ao seu objetivo.

Negar uma ação não encerra a conversa. O assistente pode explicar outra abordagem, reduzir o escopo ou apresentar a mudança em texto antes de solicitar nova confirmação.

Uma confirmação autoriza a ação mostrada naquele momento; ela não substitui a revisão do resultado. Depois de qualquer escrita, abra o arquivo alterado, confira o conteúdo e execute a validação adequada.

18.3 Fluxo recomendado

1. Peça uma análise.
2. Solicite um plano curto.
3. Aprove uma alteração pequena.
4. Revise o resultado.
5. Compile ou simule.
6. Continue somente depois da validação.

Esse ciclo reduz o risco de acumular várias mudanças antes de descobrir qual delas introduziu um problema. Para tarefas maiores, peça que o assistente divida o trabalho em etapas independentes.

18.4 Exemplos

```
Explique o arquivo ativo e diga qual é o módulo principal.
```

```
Leia o terminal Verilog e identifique o primeiro erro que preciso corrigir.
```

```
Liste os sinais do testbench e sugira apenas clock, reset, entradas e saídas.
```

```
Antes de editar, mostre exatamente quais linhas pretende alterar.
```

18.5 Privacidade

O conteúdo enviado depende do provedor configurado:

- Provedores em nuvem recebem mensagens e contexto necessário.
- Ollama processa o modelo localmente.
- Claude Code e ChatGPT via Codex CLI usam suas próprias contas e CLIs.

Não envie projetos confidenciais sem verificar as regras do provedor. Remova segredos dos arquivos e revise o contexto da conversa.

Chaves, senhas e tokens não devem ser incluídos em mensagens ou arquivos usados como contexto. Quando um erro envolver autenticação, descreva apenas a mensagem recebida e o nome do provedor.

Para configurar um serviço, siga *Configurar o provedor de IA*. Para exemplos de tarefas, consulte *Tarefas com a Aurora Intelligence*.

Configurar o provedor de IA

O provedor determina qual serviço processará as mensagens da Aurora Intelligence. Esta página explica os requisitos de cada opção e como confirmar a conexão antes de utilizar ferramentas no projeto.

Abra **Configurações da AURORA** → **Assistente IA**.

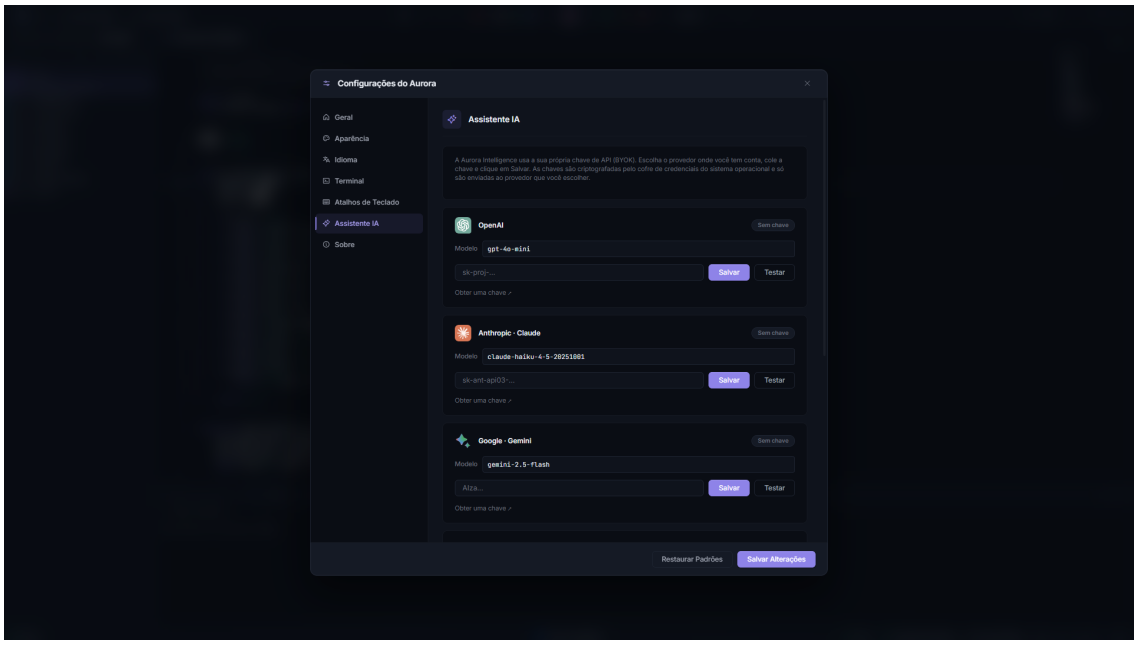


Figura1: Cada cartão reúne o modelo, a credencial e os botões para salvar e testar o provedor. Nunca publique uma captura com uma chave preenchida.

19.1 Escolher uma opção

Tipo	Opções	Requisito
API em nuvem	OpenAI, Anthropic, Google, DeepSeek e Groq	Chave de API e modelo válido.
Local	Ollama	Ollama iniciado e modelo instalado.
Assinatura por CLI	Claude Code e ChatGPT via Codex CLI	Login concluído no CLI correspondente.

19.2 Provedor por API

1. Abra o cartão do provedor.
2. Cole a chave de API.
3. Escolha ou informe o modelo.
4. Clique em **Save**.
5. Clique em **Test connection**.

Depois de salvar, o campo da chave pode ficar vazio. Isso não significa que a chave foi perdida.

O teste deve confirmar que a credencial, o endereço do serviço e o modelo são aceitos. Se a conexão funcionar, inicie uma conversa curta e peça apenas uma explicação antes de autorizar ações no projeto.

> **⚠ Aviso**
>
> As chamadas podem gerar cobrança na conta do provedor. Consulte limites e preços no serviço escolhido.

19.3 Ollama

Instale e inicie o Ollama fora da AURORA. Exemplo:

```
ollama pull llama3.1:8b
ollama serve
```

na AURORA:

1. Informe `http://localhost:11434/v1` como URL.
2. Clique em **Detect models**.
3. Selecione o modelo.
4. Teste a conexão.

Se o modelo responde, mas não consegue executar ações, escolha um modelo com melhor suporte a ferramentas. Modelos locais variam em memória necessária, velocidade e capacidade de seguir estruturas de chamadas; selecione uma opção compatível com o computador utilizado.

19.4 Claude Code

A AURORA usa o Claude Code CLI para este provedor. O executável pode ser encontrado no cache sob demanda em `userData\cli-cache`, na dependência local da aplicação ou no `PATH` do Windows. A versão pinada no manifesto atual é `2.1.196`.

Para autenticar:

```
claude login
claude --version
```

Depois, selecione Claude Code na AURORA e inicie uma conversa. Se o CLI ainda não estiver cacheado, a AURORA pode baixar o pacote previsto pelo manifesto. O login continua sendo responsabilidade do CLI e usa as credenciais locais em `~\.claude\.credentials.json`.

19.5 ChatGPT via Codex CLI

A opção **ChatGPT** usa o Codex CLI local. O executável pode ser encontrado no cache sob demanda em `userData\cli-cache`, na dependência local da aplicação ou no `PATH` do Windows. A versão pinada no manifesto atual é `0.131.0`.

Para autenticar:

```
codex login
codex --version
```

Depois, selecione ChatGPT na AURORA. Se um modelo específico falhar, use a opção **Default**.

O Codex CLI usa `CODEX_HOME\auth.json` quando `CODEX_HOME` está definido, ou `~\.codex\auth.json` no caso padrão. A AURORA não garante que essa conta seja a mesma conta aberta no navegador.

19.6 Se o teste falhar

- Confirme a chave, conta e modelo.
- Verifique conexão, proxy e firewall.
- Para Ollama, confirme que `ollama serve` está ativo.
- Para Claude Code ou Codex, refaça o login no CLI correspondente.
- Se o CLI foi instalado ou autenticado fora da AURORA, reinicie a aplicação para atualizar o ambiente.
- Se o cache sob demanda falhar, verifique conexão com o registro npm e permissões de escrita no perfil do usuário.

Leia a mensagem apresentada pelo teste antes de trocar várias configurações. Erros de autenticação, modelo inexistente, serviço indisponível e bloqueio de rede exigem correções diferentes.

Capítulo 20

Tarefas com a Aurora Intelligence

Em vez de memorizar nomes de ferramentas internas, descreva o resultado esperado. Inclua o arquivo, a etapa ou o erro relevante e informe se o assistente pode apenas analisar ou também propor alterações.

Os exemplos desta página são modelos de pedido. Substitua os nomes genéricos pelo contexto real e mantenha uma tarefa principal por mensagem.

20.1 Entender o projeto

Comece por consultas de leitura. Elas ajudam a confirmar se o assistente reconheceu corretamente o projeto antes de executar qualquer ação.

```
Resuma a estrutura do projeto e identifique Top Level, Testbench Top e processador ativo.
```

```
Explique o arquivo ativo para alguém que está aprendendo Verilog.
```

20.2 Corrigir compilação

Peça que o terminal seja analisado na ordem em que as mensagens ocorreram. Corrigir a primeira falha evita alterações desnecessárias causadas por erros secundários.

```
Leia os terminais e explique somente o primeiro erro que impede a compilação.
```

```
Confira se há módulos duplicados ou dependências ausentes antes de alterar arquivos.
```

20.3 Editar com segurança

Defina limites claros: arquivo, trecho, comportamento esperado e validação que deverá ser executada depois da mudança.

```
Mostre a alteração proposta e aguarde minha confirmação antes de editar.
```

```
Faça apenas a menor mudança necessária e salve o arquivo.
```

20.4 Trabalhar com processadores

```
Liste os processadores e explique a configuração do processador ativo.
```

```
Crie um processador chamado filtro com 16 bits, uma entrada e uma saída.
```

20.5 Simular e analisar ondas

Informe se deseja apenas preparar a seleção, executar a simulação ou interpretar o resultado. Essas ações possuem efeitos e tempos de execução diferentes.

```
Confira Top Level e Testbench Top e diga o que falta para simular.
```

```
Selecione clock, reset, entradas e saídas na **Configuração de Ondas**.
```

```
Execute a simulação e explique o resultado do terminal sem modificar o RTL.
```

20.6 Organizar arquivos

```
Liste referências ausentes e não remova nenhuma sem minha aprovação.
```

```
Crie um backup antes de renomear o processador.
```

20.7 Boas práticas

- Informe uma tarefa por vez.
- Peça análise antes de alterações amplas.
- Leia a confirmação de ações de escrita.
- Valide arquivos editados com compilação ou simulação.
- Mantenha backup ou controle de versão.

Depois de uma ação, peça um resumo objetivo do que foi alterado e do resultado da validação. Não considere uma resposta textual como prova de que a compilação ou a simulação foi concluída.

Capítulo 21

Usar Claude Code e ChatGPT via Codex CLI

Claude Code e ChatGPT via Codex CLI podem ser usados como provedores de assinatura dentro da Aurora Intelligence. A autenticação é feita no CLI de cada serviço. A AURORA inicia o provedor selecionado e disponibiliza ferramentas controladas para que ele consulte ou opere o projeto.

21.1 Como a AURORA encontra os CLIs

A AURORA procura o executável nesta ordem:

1. Cache sob demanda em `userData\cli-cache`.
2. Dependência local instalada com a aplicação.
3. Executável disponível no `PATH` do Windows.

O manifesto atual fixa Claude Code em `2.1.196` e Codex em `0.131.0`. Quando o CLI não está no cache, a AURORA pode baixar o pacote previsto pelo manifesto e reutilizá-lo nas próximas execuções.

21.2 Preparar o Claude Code

1. Execute `claude login` no terminal.
2. Confirme com `claude --version`.
3. Reinicie a AURORA se o login ou instalação tiver ocorrido enquanto ela estava aberta.
4. Selecione Claude Code em **Configurações da AURORA** → **Assistente IA**.

O login fica em `~\.claude\.credentials.json`. Se `claude --version` não for reconhecido no terminal, a AURORA ainda pode usar o cache sob demanda ou a dependência local; use o terminal principalmente para concluir login e diagnosticar credenciais.

21.3 Preparar o ChatGPT via Codex CLI

1. Execute `codex login` no terminal.
2. Confirme com `codex --version`.
3. Reinicie a AURORA se o login ou instalação tiver ocorrido enquanto ela estava aberta.
4. Selecione ChatGPT em **Configurações da AURORA** → **Assistente IA**.

O Codex usa `CODEX_HOME\auth.json` quando `CODEX_HOME` está definido, ou `~\.codex\auth.json` no padrão. A AURORA não comprova que essa conta seja a mesma sessão do ChatGPT no navegador.

21.4 Durante a conversa

A AURORA fornece ao CLI ferramentas controladas para consultar e operar o projeto. Alterações continuam sujeitas às confirmações exibidas pela interface.

Ao criar uma nova conversa, o contexto começa limpo. Informe novamente o objetivo e os arquivos relevantes. Ao continuar uma conversa existente, a AURORA tenta retomar a sessão anterior e preservar o encadeamento do trabalho.

Mesmo quando o CLI possui recursos próprios de execução, use as confirmações exibidas pela AURORA como limite operacional. Revise o nome da ação, os argumentos e os arquivos antes de autorizar.

21.5 Problemas comuns

CLI não encontrado
: Verifique se o download sob demanda conseguiu gravar em `userData\cli-cache`, se a dependência local está presente ou se o executável está no `PATH`.

Login expirado
: Execute novamente `claude login` ou `codex login`.

Conversa antiga não retoma
: Inicie uma nova conversa e repita o contexto necessário.

Ferramenta não responde
: Confirme que a janela principal continua aberta e reinicie a conversa.

Modelo explícito não é aceito
: Selecione **Default** para usar a opção compatível com a conta.

Parte VI

Configuração e ajuda

Capítulo 22

Configurações do aplicativo

Abra **Configurações da AURORA** pela barra superior.

As configurações controlam a apresentação da interface, o nível de detalhes dos terminais e as integrações de IA. Faça uma alteração por vez e confirme o efeito antes de continuar, principalmente ao ajustar opções usadas em diagnóstico.

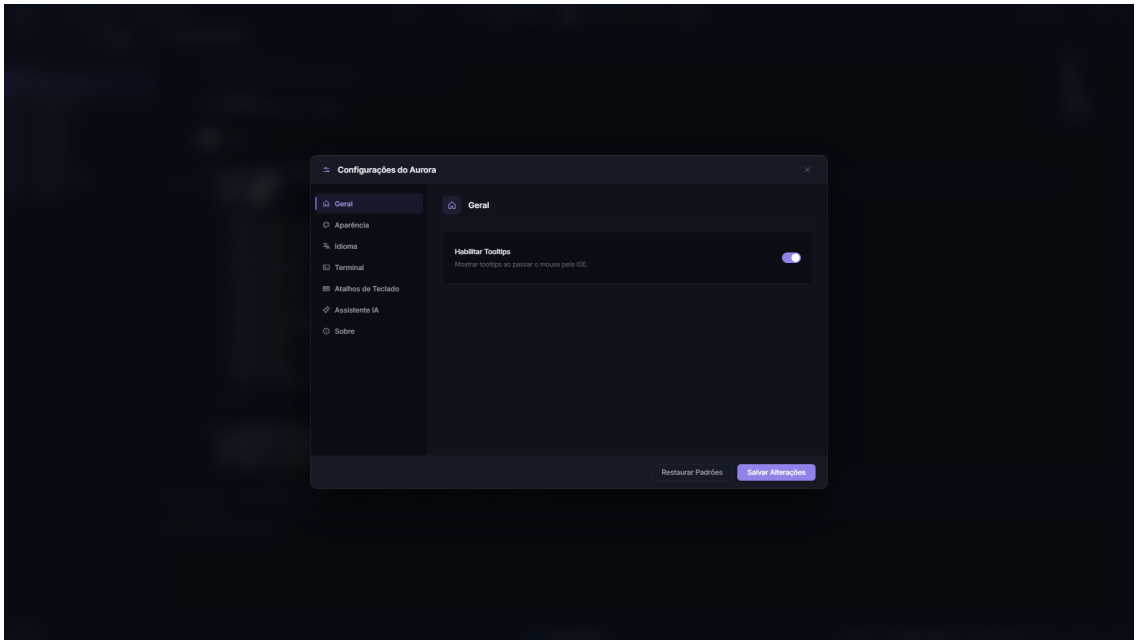


Figura1: As categorias ficam no menu lateral. A seção **Geral** controla as dicas exibidas ao passar o cursor sobre a interface.

22.1 Geral

Idioma
: Altera a interface e as mensagens geradas pela toolchain.

Tooltips
: Exibe explicações curtas ao posicionar o cursor sobre controles.

Ative tooltips enquanto estiver aprendendo a interface.

Depois de se familiarizar com os controles, você pode desativá-los para reduzir elementos sobrepostos. A alteração afeta apenas as dicas visuais; ela não modifica o projeto.

22.2 Aparência

Altere ou restaure o ícone da aplicação, quando a opção estiver disponível. A versão atual não expõe um seletor geral de tema nessa seção.

22.3 Terminal

Ative **verbose** quando precisar de detalhes para diagnóstico. Esse modo pode exibir comandos, caminhos e etapas auxiliares que ajudam a localizar a origem de uma falha. No uso normal, mantenha-o desativado para destacar as mensagens principais.

22.4 Atalhos

Consulte *Atalhos* para os atalhos confirmados nesta versão.

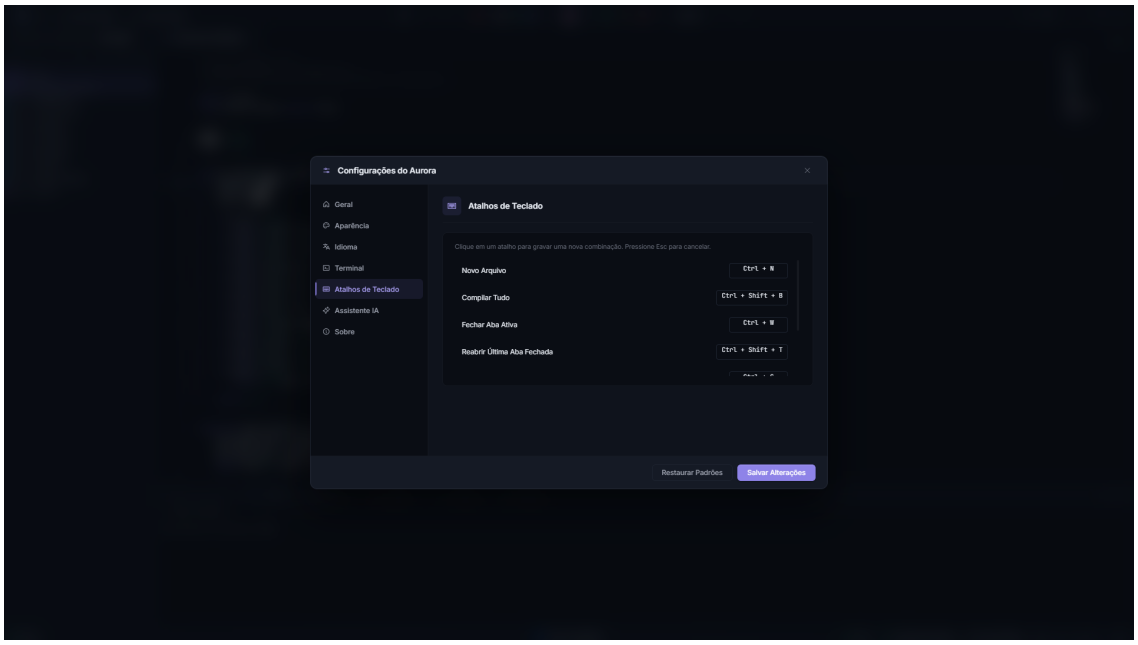


Figura2: Clique em uma combinação para redefini-la. Pressione `Esc` para cancelar a gravação de um novo atalho.

22.5 AI Assistant

Configure chaves, modelos, Ollama, Claude Code e ChatGPT via Codex CLI. Veja *Configurar o provedor de IA*.

Depois de configurar um provedor, use o teste de conexão antes de iniciar uma tarefa no projeto. Não inclua chaves ou tokens nas mensagens da conversa.

22.6 About e atualização

Use **About** para conferir a versão instalada. Essa informação deve acompanhar relatórios de problema. Quando houver atualização disponível, leia as notas, salve os arquivos e faça backup do projeto antes de instalar.

22.7 Zoom e janela

Os controles da barra permitem minimizar, maximizar, fechar e ajustar o zoom. O PRISM possui controles equivalentes em sua própria janela.

O zoom altera a escala da interface, não o conteúdo dos arquivos. Ajuste-o quando painéis ou textos não couberem adequadamente na janela.

Capítulo 23

Arquivos do projeto e backup

Esta página identifica quais arquivos fazem parte do trabalho e como mover ou arquivar um projeto com segurança. O princípio principal é preservar o `.spf` junto com os fontes e recursos aos quais ele se refere.

23.1 O que deve ser preservado

Para copiar ou arquivar um projeto, preserve a pasta completa. Os itens principais são:

Item	Conteúdo
Arquivo `.spf`	Configuração e estrutura do projeto
`Software/`	Algoritmos C±
`Hardware/`	Verilog gerado e fontes relacionadas
`Simulation/`	Arquivos de entrada, saída e testes do processador
Testbenches	Arquivos Verilog ou Python usados na simulação
`testbench/`	Seleção de sinais e layouts de ondas
`.inv`	Filtro visual opcional da visualização **Pastas**

Os arquivos em `Hardware` podem ser regenerados em muitos fluxos, mas preservá-los junto com o projeto facilita comparação, reprodução e diagnóstico. Os fontes C±, módulos adicionados manualmente e testbenches devem sempre ser tratados como dados essenciais.

23.2 Mover um projeto

1. Feche o projeto na AURORA.
2. Copie a pasta completa.
3. Abra o `.spf` na nova localização.
4. Verifique arquivos importados.
5. Confirme Top Level e Testbench Top.

Arquivos importados de fora da pasta continuam apontando para o caminho original.

Depois da cópia, execute uma validação Verilog ou abra os arquivos principais. Isso confirma que o projeto não depende de um caminho que ficou na máquina anterior.

23.3 O que não editar manualmente

Durante o uso normal, evite alterar diretamente:

- O arquivo `.spf`.
- Arquivos de estado dentro de `testbench/`.
- Dados internos da IA no perfil do Windows.
- Arquivos temporários da toolchain.

Use a interface para mudar configurações e referências. O `.inv` é a exceção: ele pode ser criado ou ajustado manualmente quando você quiser filtrar a visualização **Pastas**, pois não altera o projeto nem o Git.

Uma edição manual incorreta pode deixar o arquivo sintaticamente válido, mas inconsistente com as pastas reais. Se precisar investigar o formato, trabalhe sobre uma cópia do projeto.

23.4 Fazer backup

Use o comando de backup da AURORA antes de alterações estruturais. Para projetos importantes, mantenha também uma cópia externa ou repositório de controle de versão.

23.5 Arquivo `.inv`

O `.inv` fica na raiz do projeto e controla apenas a visualização **Pastas**. Ele aceita comentários, linhas vazias, negação com `!`, padrões de diretório, globs e `**`. A comparação é feita sem diferenciar maiúsculas de minúsculas.

Não use `.inv` para segurança, empacotamento ou Git. Para controle de versão, use `.gitignore` e revise o painel de Source Control.

23.6 Layouts de ondas

O diretório `testbench/` guarda um JSON por Testbench Top. Esse estado registra sinais selecionados, inicialização da **Configuração de Ondas**, layouts `.gtkw` e layouts Surfer (`.surf.ron` ou `.sucl`) associados ao testbench.

Layouts `.gtkw` organizam sinais para GTKWave. Layouts `.surf.ron` e `.sucl` configuram o Surfer. Eles não substituem o arquivo de onda (`.vcd` ou `.fst`), que continua sendo a fonte dos valores simulados.

Nunca dependa apenas da pasta de arquivos temporários ou da lista de projetos recentes.

23.7 Verificar um backup

1. Copie o backup para uma pasta de teste.
2. Abra o `.spf` dessa cópia.
3. Confira processadores, Top Level e Testbench Top.
4. Abra os fontes principais.
5. Execute a validação apropriada.

Um backup só deve ser considerado utilizável depois dessa verificação. Evite testar diretamente sobre a única cópia preservada.

Atalhos

Os atalhos aceleram ações frequentes do editor e das janelas auxiliares. Eles seguem o contexto ativo: um comando do editor exige que o foco esteja no arquivo, enquanto um comando da **Configuração de Ondas** exige que essa janela esteja ativa.

24.1 AURORA

Atalho	Ação
`Ctrl+N`	Criar arquivo
`Ctrl+W`	Fechar aba ativa
`Ctrl+Shift+T`	Reabrir última aba fechada
`Ctrl+S`	Salvar arquivo ativo
`Ctrl+Shift+S`	Salvar todos os arquivos
`Ctrl+Shift+F`	Abrir busca nos arquivos
`Ctrl+Shift+K` ou `Ctrl+Shift+P`	Abrir a paleta de comandos da AURORA
`Ctrl+Alt+S`	Alternar análise semântica Slang/SystemVerilog

24.2 Editor

Atalho	Ação
`Ctrl+F`	Localizar
`Ctrl+H`	Substituir
`Ctrl+G`	Ir para linha
`F1`	Paleta de comandos do Monaco
`F12`	Ir para definição quando suportado
`Shift+F12`	Localizar referências quando suportado

24.3 Configuração de Ondas

Atalho	Ação
`Ctrl+F`	Focar a pesquisa
`Esc`	Limpar a pesquisa; pressionar novamente para fechar

Atalhos exibidos por versões antigas podem não estar ativos. Use apenas os listados nesta página.

Se um atalho não responder, clique primeiro na área em que ele deve atuar e tente novamente. Menus, diálogos abertos ou ferramentas externas podem capturar a combinação enquanto estiverem em primeiro plano.

Capítulo 25

Solução de problemas

Use esta página para localizar a etapa que falhou antes de alterar configurações ou reinstalar ferramentas. Um diagnóstico eficiente registra o contexto, reproduz uma única ação e começa pela primeira mensagem de erro.

25.1 Comece por aqui

1. Salve todos os arquivos.
2. Confira projeto, processador, Top Level, Testbench Top e simulador na barra de status.
3. Repita apenas a ação que falhou.
4. Leia a primeira mensagem de erro no terminal correspondente.
5. Ative **verbose** somente se precisar de mais detalhes.

Anote o resultado esperado e o que realmente aconteceu. Essa comparação evita descrições genéricas como "não funciona" e ajuda a decidir se a falha está na interface, na configuração do projeto, na compilação ou no testbench.

25.2 Interface

Um botão está desabilitado
Abra o arquivo esperado e confirme Top Level ou Testbench Top. Veja *Interface principal*.

O editor não carrega
Feche e abra a AURORA. Se continuar, reinicie o Windows e repita a abertura.

Um arquivo mudou fora da AURORA
Escolha conscientemente entre a versão do disco e a versão do editor.

25.3 Compilação

Top Level não encontrado
Confirme o módulo principal e inclua suas dependências.

Módulo duplicado
Remova uma das fontes que declaram o mesmo módulo.

A geração C± não atualiza o hardware
Salve o `.cmm`, confirme o processador ativo e execute C± novamente.

O cancelamento demora
Aguarde alguns segundos para que processos auxiliares sejam encerrados.

Depois de uma falha de compilação, não use arquivos antigos em `Hardware` como evidência de sucesso. Verifique a data de atualização e confirme que o terminal da execução atual terminou corretamente.

25.4 Simulação e ondas

Testbench não inicia
Confirme Testbench Top, clock, reset e nomes de módulos.

Nenhum arquivo de onda
Confirme que o testbench terminou e possui sinais configurados para dump.

Viewer de ondas abre sem sinais
Volte o layout para **padrão**, revise a **Configuração de Ondas** e gere novamente.

O cocotb não encontra um sinal
Compare o nome usado no Python com as portas reais do módulo.

Se a simulação não termina, confira se o testbench possui uma condição de encerramento. Use **Cancelar** antes de iniciar outra execução.

25.5 PRISM

Execute **Verilog** antes do PRISM. Corrija primeiro erros de módulo ausente, módulo duplicado e Top Level incorreto.

Se o Verilog está válido, mas o diagrama continua desatualizado, salve os arquivos e use **Recompile** na janela do PRISM.

25.6 Aurora Intelligence

Chave salva, mas a conexão falha
Confirme conta, modelo, rede e permissões do provedor.

Ollama não detecta modelos
Confirme que `ollama serve` está ativo e use `http://localhost:11434/v1`.

Claude Code ou ChatGPT via Codex CLI não conecta
Execute novamente o login, confirme `--version` e reinicie a AURORA.

25.7 Pedir suporte

Inclua:

- Versão da AURORA e do Windows.
- Ação exata que falhou.
- Resultado esperado e observado.
- Saída exportada do terminal.
- Projeto mínimo sem dados confidenciais.

Inclua apenas o necessário para reproduzir o problema. Remova chaves, dados pessoais e arquivos que não participam do fluxo. Quando possível, descreva uma sequência curta de passos que leve ao mesmo resultado.

Parte VII

Apêndices

Capítulo A

Glossário

Este glossário reúne os termos usados de forma consistente no manual. Quando a interface mantiver um nome em inglês, o termo é preservado para facilitar sua localização na AURORA.

AURORA
Ambiente desktop do ecossistema SAPHO para organizar projetos, editar fontes, gerar processadores, compilar, simular e analisar circuitos.

C± / CMM
Linguagem de entrada para geração de processadores dedicados. Os algoritmos são armazenados em arquivos com extensão `.cmm`.

YANC / SAPHO
YANC é o conjunto de compiladores usado na transformação de C± em hardware. SAPHO é o ecossistema no qual esse fluxo está inserido.

Top Level / Testbench Top
Top Level é o módulo Verilog raiz do circuito. Testbench Top é o arquivo que inicia e controla a simulação.

RTL
Representação do circuito em nível de transferência entre registradores, usada pelos módulos Verilog do projeto.

Icarus / Verilator / cocotb
Icarus e Verilator são opções de simulação do RTL. O cocotb é o framework Python usado para escrever testbenches executados com um simulador compatível.

VCD / FST / GTKWave / Surfer / layouts
VCD e FST armazenam formas de onda. GTKWave e Surfer são viewers. `.gtkw`, `.surf.ron` e `.sucl` registram organização visual ou comandos de visualização, sem conter os dados da simulação.

Yosys / PRISM
Yosys processa a estrutura RTL usada pelo PRISM, que apresenta módulos e conexões em um diagrama navegável.

Aurora Intelligence
Assistente integrado que pode consultar o contexto do projeto e, mediante confirmação, executar ações controladas.

MCP
Model Context Protocol, empregado para disponibilizar ferramentas controladas da AURORA a agentes compatíveis.

SPF
Formato JSON de projeto com extensão `.spf`. Registra estrutura, processadores, arquivos e seleções relevantes.

CommandSpec
Contrato interno que representa executável, argumentos, diretório de trabalho e ambiente sem montar um comando de shell.

Escopo, versão e método

B.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	6.3.2
Commit	`cee4922189e746b83e0198fd998ed50646f18371`
Branch	`main`
Plataforma primária	Windows 10/11
Runtime	Electron 39.8.10 e Node.js 18+
Editor	Monaco Editor 0.52.2, fixado exatamente
Data da análise	9 de julho de 2026

B.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 NIPSCERN 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.

B.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.

B.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.