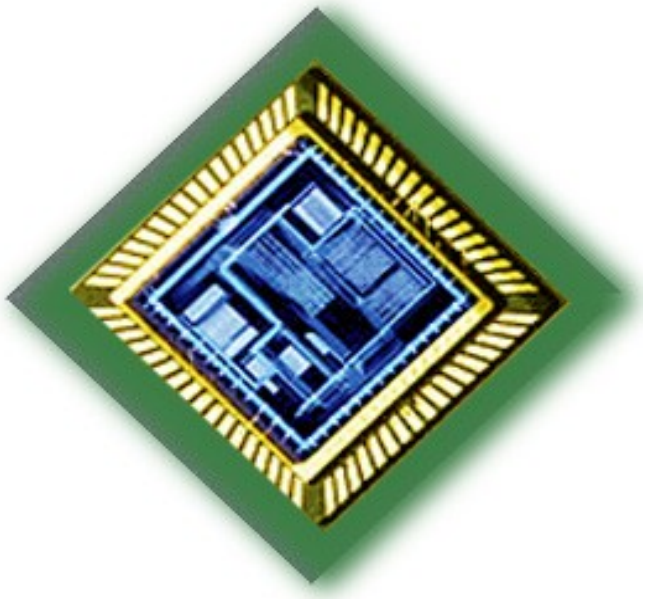


Verificação funcional

Curso do projeto Brazil-IP

Elmar Melcher
UFCG
elmar@dsc.ufcg.edu.br
http://lad.dsc.ufcg.edu.br/ip

Roteiro

- Introdução
  - Motivação
  - Fluxo de projeto
- Verificação
  - Tipos de verificação
  - Verificação funcional
  - Abordagens de verificação
  - Plano de verificação

Roteiro

- Metodologia VeriSC
  - Testbench
  - Elementos básicos
  - Regras de projeto
  - Implementação
  - Tipos de estímulos
  - Cobertura
  - Biblioteca BVE_COVER

Roteiro

- Exemplos
  - Codificador DPCM
  - Decodificador MPEG4
  - Lacre Eletrônico

Introdução

- Ninguém usa um IP core no qual não confia.

IP core: “Lógica ou os dados necessários para construir um dado projeto de hardware.
Idealmente ele é “reusável” e pode ser adaptado a vários tipos de dispositivos de hardware. Pode-se entender o *IP-core* como a implementação de um dado projeto de hardware em uma linguagem especifica para esse objetivo.”

Motivação

Caso famoso: NASA

- Exemplo de problema causado por má verificação na NASA:
  - Mars Climate Orbiter de 1999
  - Erro de verificação de sistema fez com que o Orbiter voasse muito próximo à atmosfera marciana e queimasse.
  - Problema era uma mistura de unidades métricas e unidades inglesas que causou um erro no cálculo da trajetória.

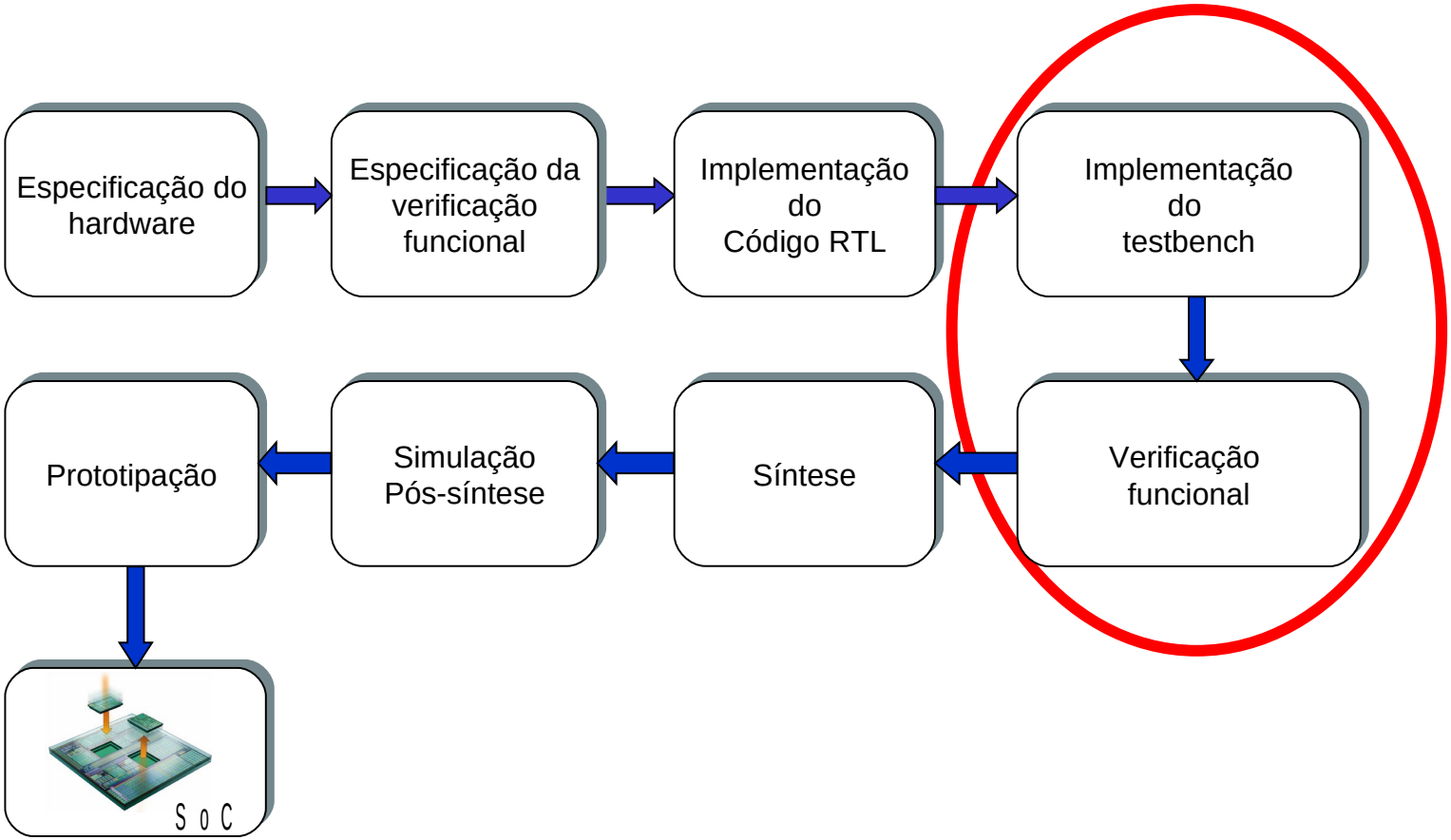
Caso famoso: INTEL

- Problema descoberto no Pentium em 1994 por um professor de matemática durante uma pesquisa matemática.
  - A FPU produzia resultados de divisão errôneos no oitavo digito significativo para certos argumentos.
  - Embora no máximo 1 usuário em 1 milhão jamais foi afetado, a confiança na INTEL caiu.
  - Abriu o caminho para crescimento maior da AMD.
  - Prejuízo de 500 MUS$ (?)

Verificação de IP cores

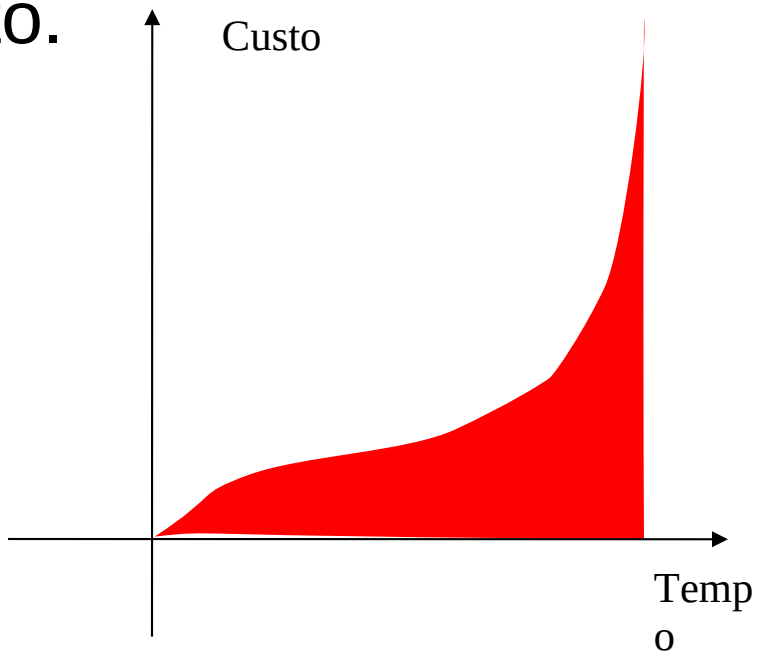
- Como alcançar confiança em um projeto?
  - Processo de verificação bem executado e documentado.
- IP cores precisam ser verificados mais amplamente,
  - Todas propriedades e utilizações possíveis devem ser verificadas,
  - Não somente um ambiente específico.

Fases de um projeto de hardware



Custo versus Tempo

- Entregar produtos e ganhar dinheiro.
- **Custo** e tempo de corrigir um defeito cresce quando descoberto mais tarde no ciclo de vida do produto.
  - na especificação
  - na simulação
  - na prototipagem
  - na fabricação em volume
  - no uso pelo cliente



Verificação

Tipos de verificação

- Verificação:
  - Dinâmica ou Funcional.
  - Estática ou Formal.
  - Híbrida.

Verificação funcional

“Verificação funcional é um processo usado para demonstrar que o objetivo do projeto é preservado em sua implementação” **[bergeron2003]**

Verificação funcional x Teste

Verificação Funcional: confrontar um modelo a ser verificado a outro modelo padrão, comparando a funcionalidade por simulação.

X

Teste: verificar se um CI está sem erro de fabricação.

Verificação funcional

- Mais da metade do esforço de projeto está na verificação.
  - Um testbench muitas vezes contém mais linhas que a própria descrição do projeto.
  - A equipe de engenheiros de verificação é maior do que a equipe de projetistas.
- Verificação é difícil.

Verificação funcional

- Engenheiros da Motorola no SBCCI 2002: “Estamos procurando trabalhos sobre verificação”

- Processo de projeto está indo direito.

- Procura por “bons” engenheiros de verificação em todas as empresas.

Custo da verificação

- Um mal necessário
  - sempre leva tempo demais e custa caro demais
- mas indispensável.
  - porque afeta diretamente os três requisitos:
    - cronograma
    - custo
    - qualidade

Isso funciona mesmo ?

- A verificação funcional deve responder a esta pergunta.
- “isso” é uma descrição RTL de um projeto.
- “funciona” se refere a simulação.
- O funcionário de uma empresa deve poder **dormir tranqüilo** se a verificação responde “sim”.

Como saber fazer ?

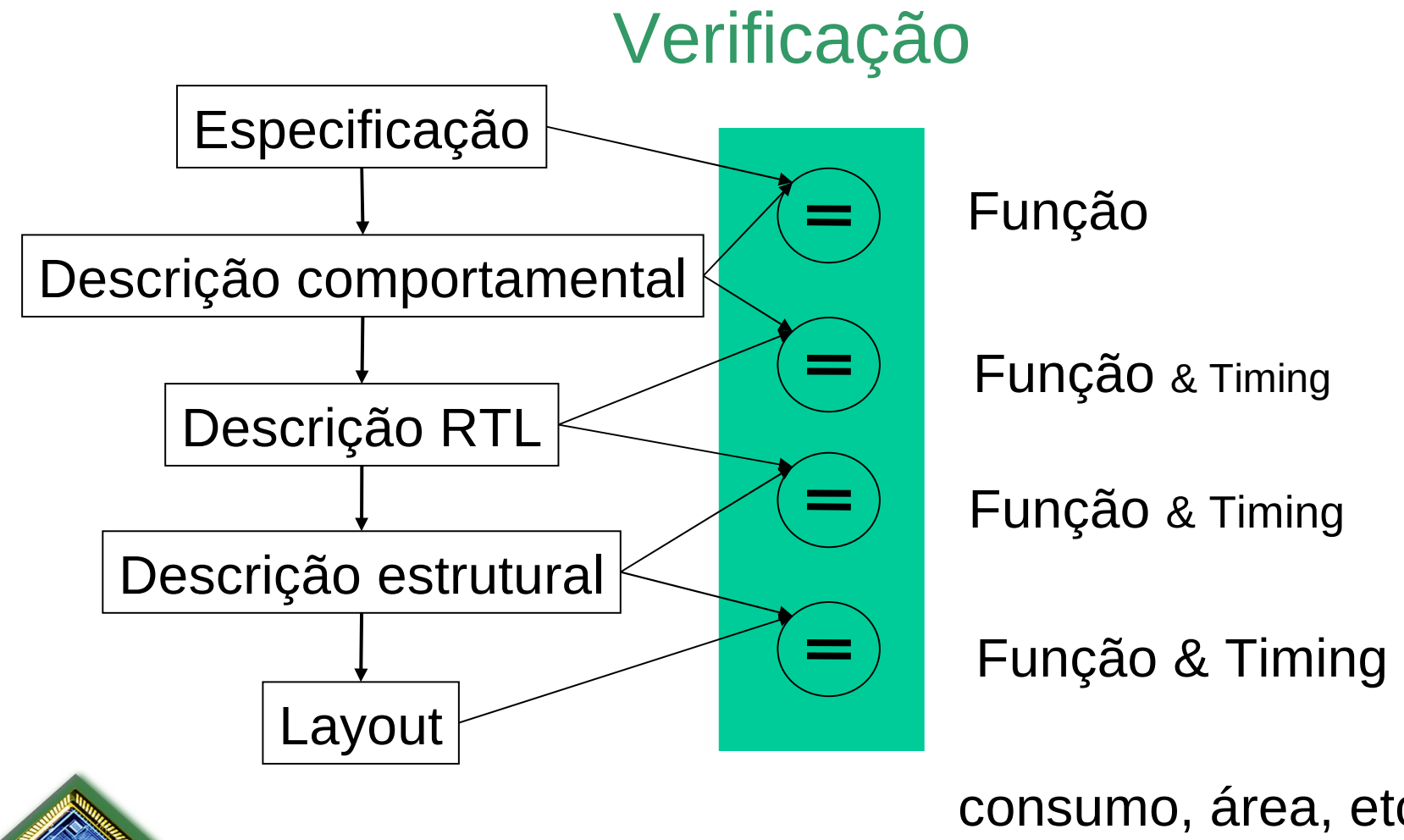
- Muitos livros falam sobre implementação.

- Poucos falam sobre verificação.
  - *Writing Testbenches: Functional Verification of HDL Models* by Janick Bergeron, 2nd edition, KluwerAcademic Publishers, 2003
  - Piziali, *Functional verification Coverage Measurement and Analysis*, Kluwer 2004.

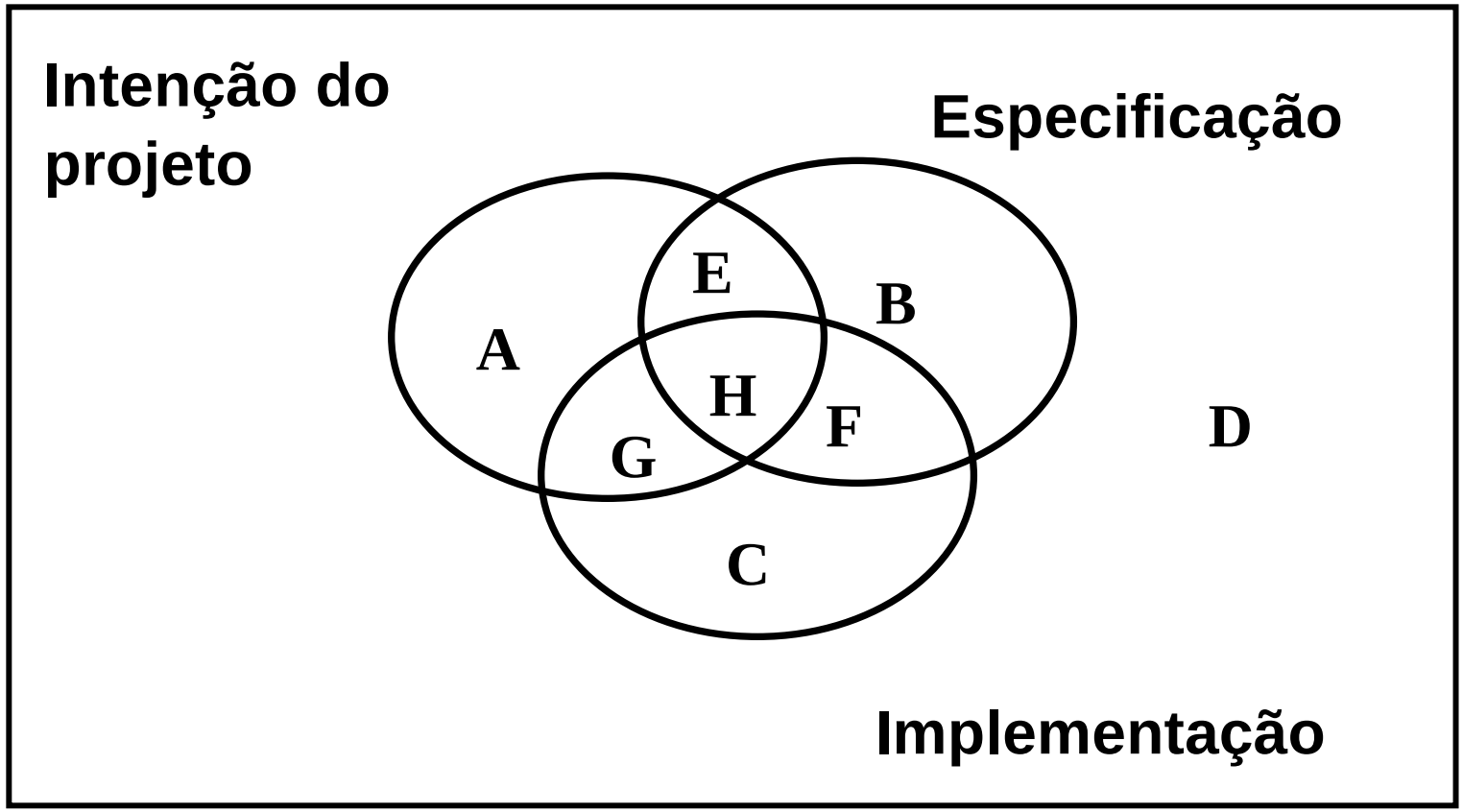
Verificação estática

- Analisadores de código HDL, *Lint* ou *linting tools*
- Tipos de problemas detectáveis
  - **`case`** incompleto
  - atribuições em **`if...else`** inconsistente
  - falta de sinais na lista de sensibilidade
  - reset síncrono / assíncrono
  - falta de sinais a serem resetados

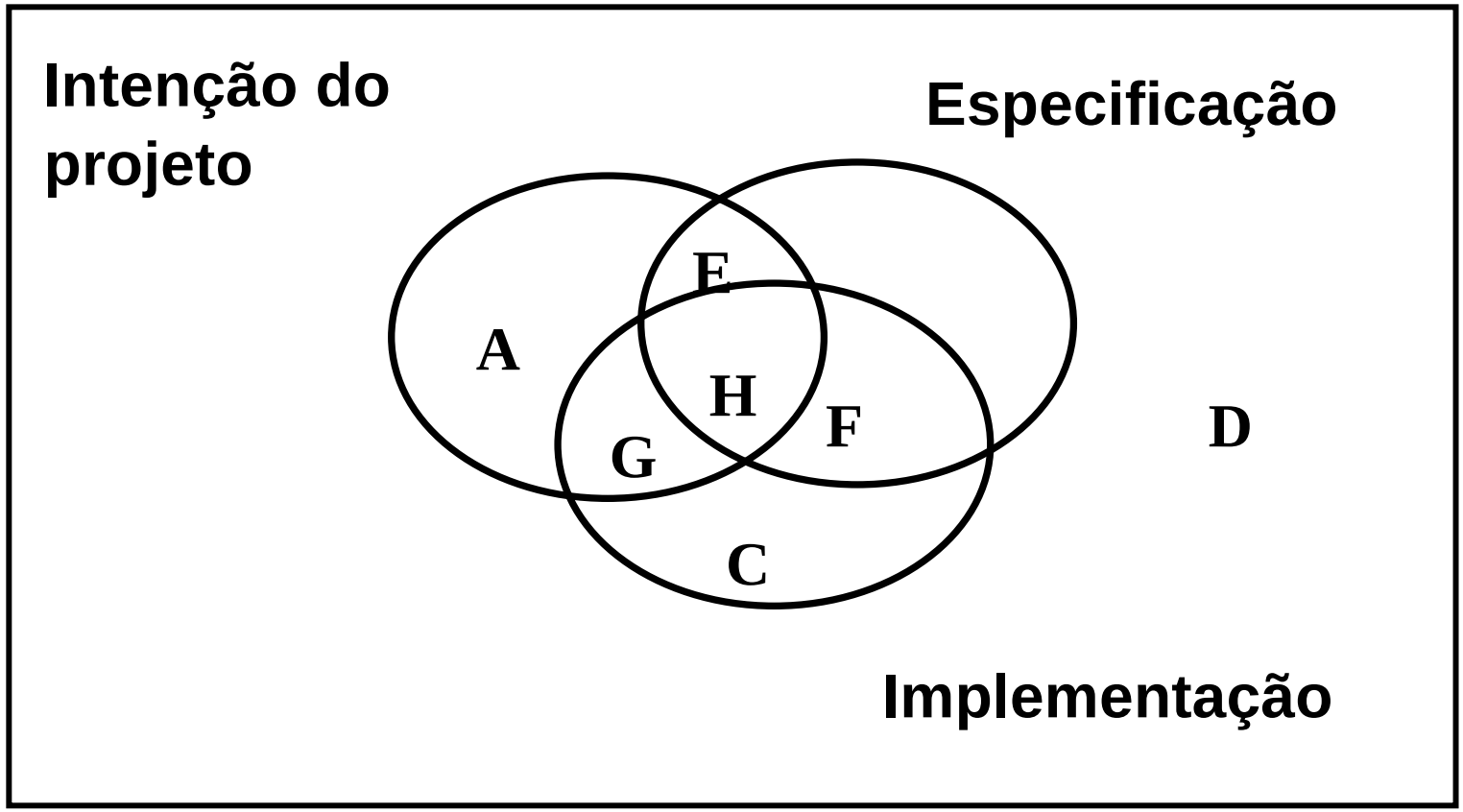
Fluxo de projeto



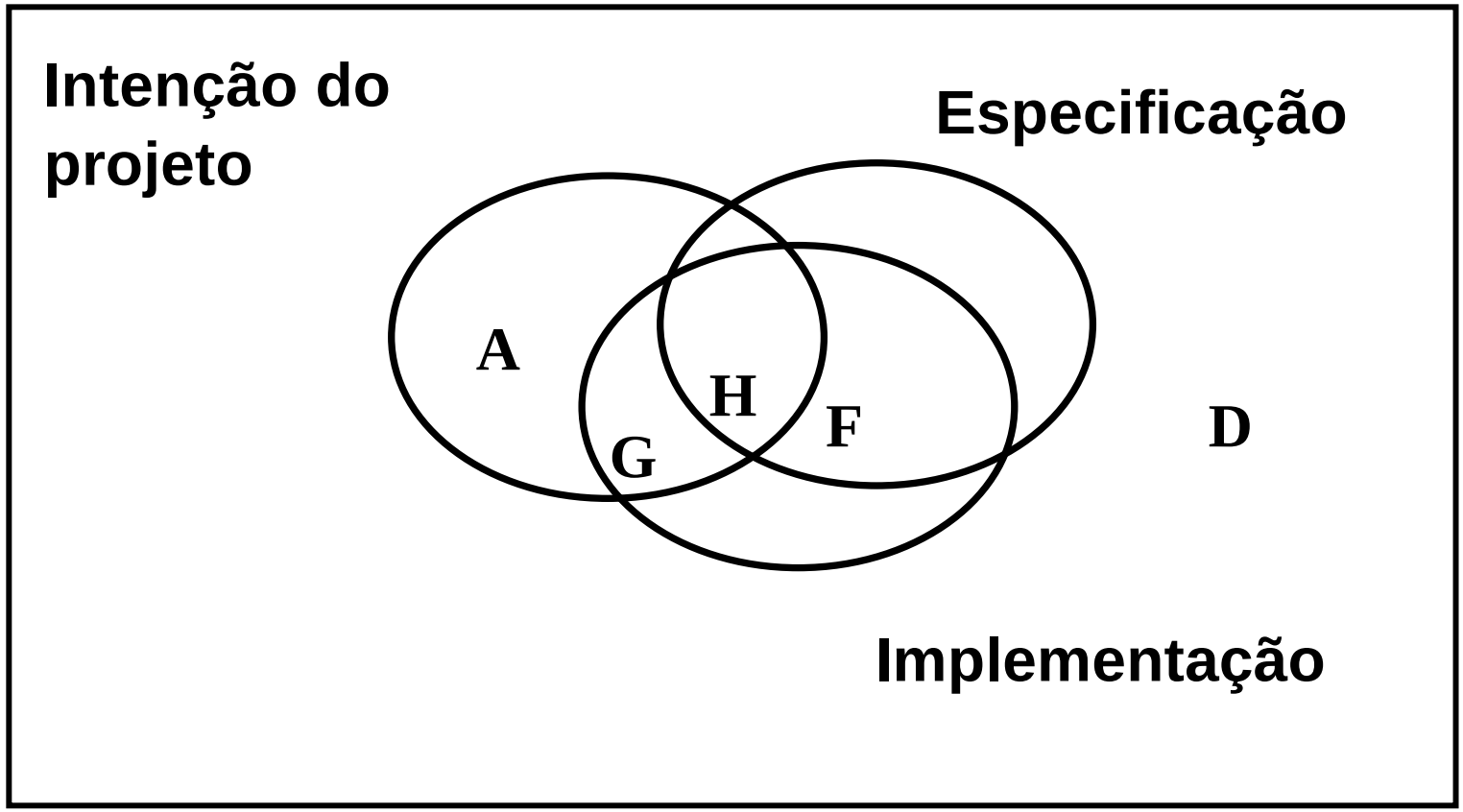
Projeto x Verificação



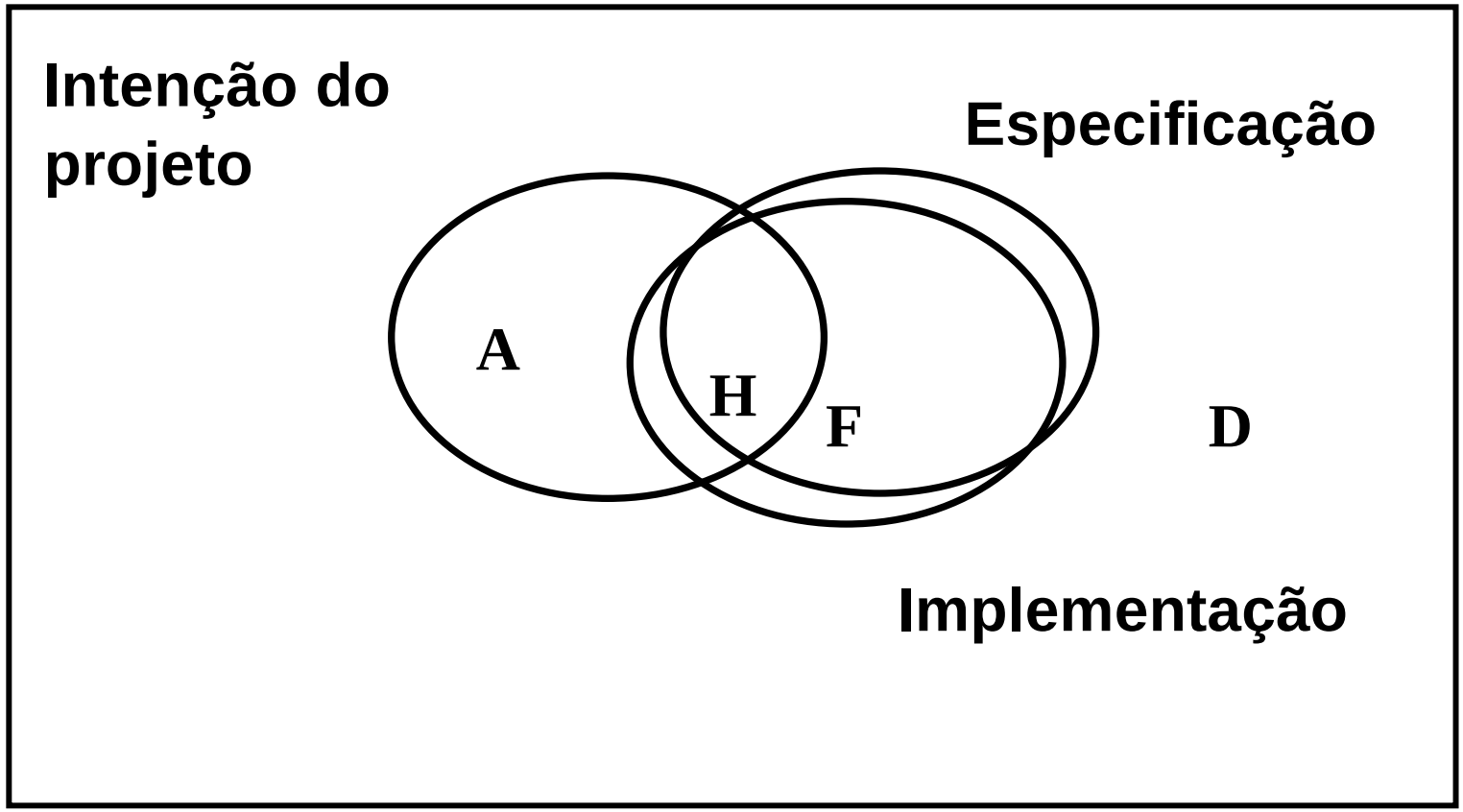
Projeto x Verificação



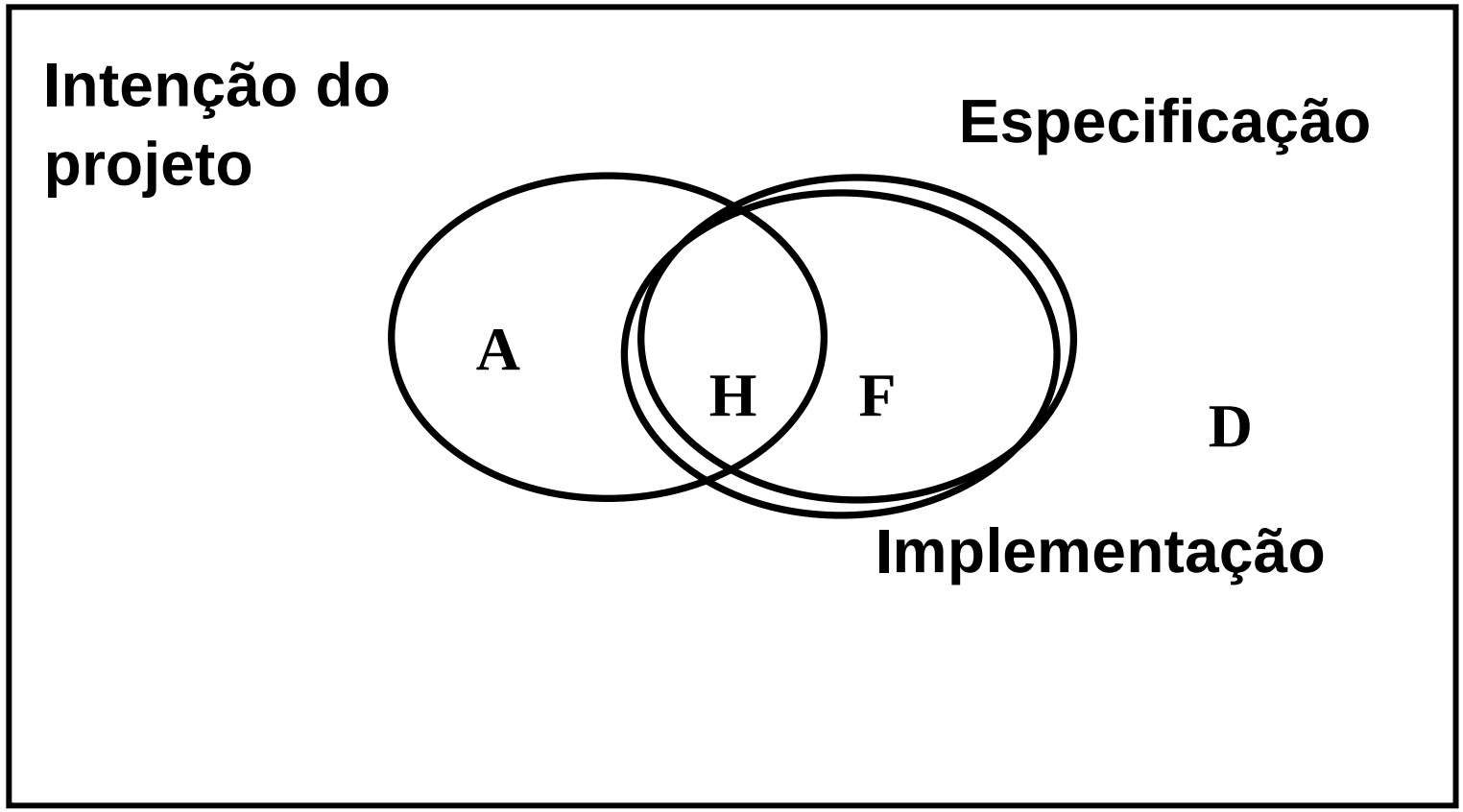
Projeto x Verificação



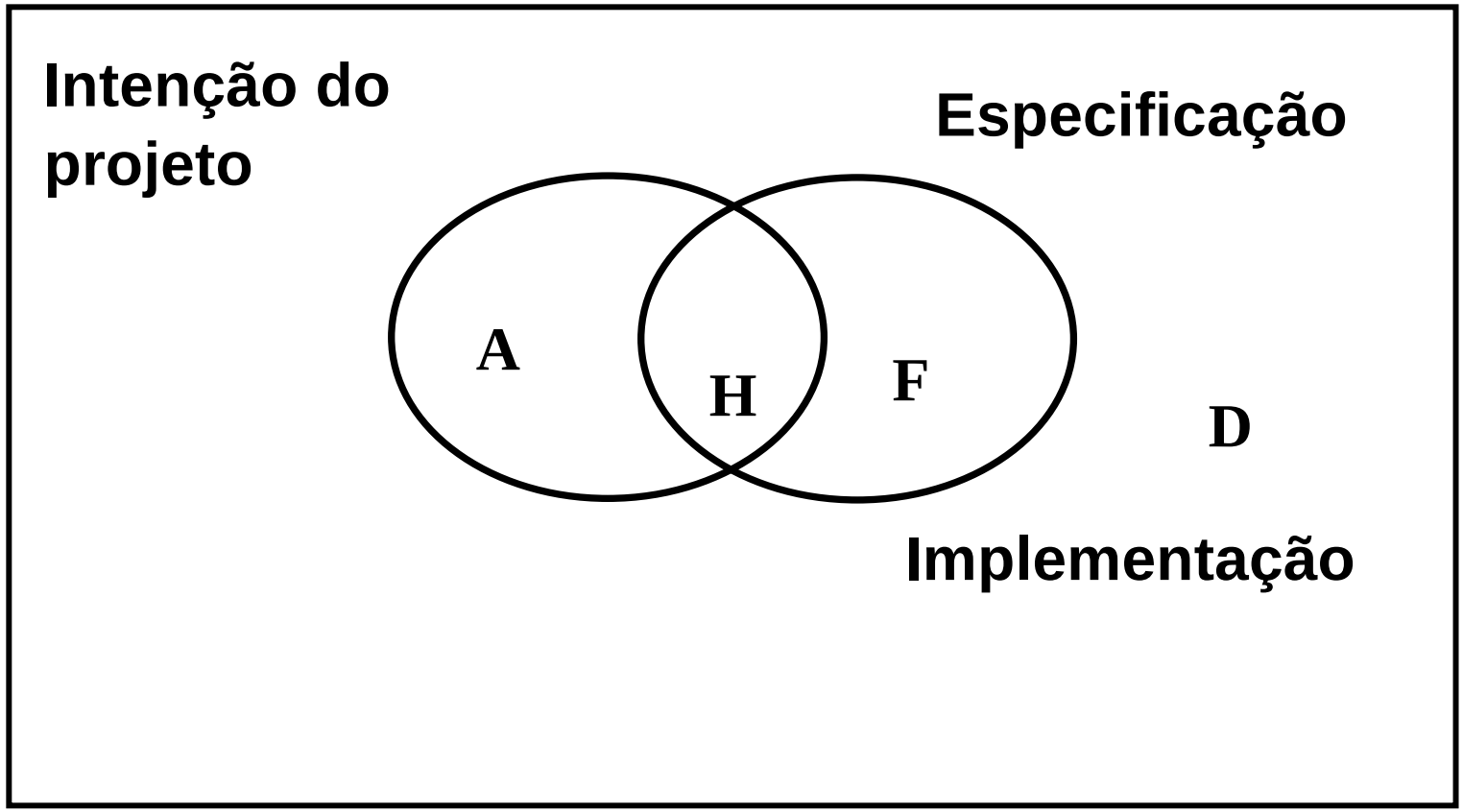
Projeto x Verificação



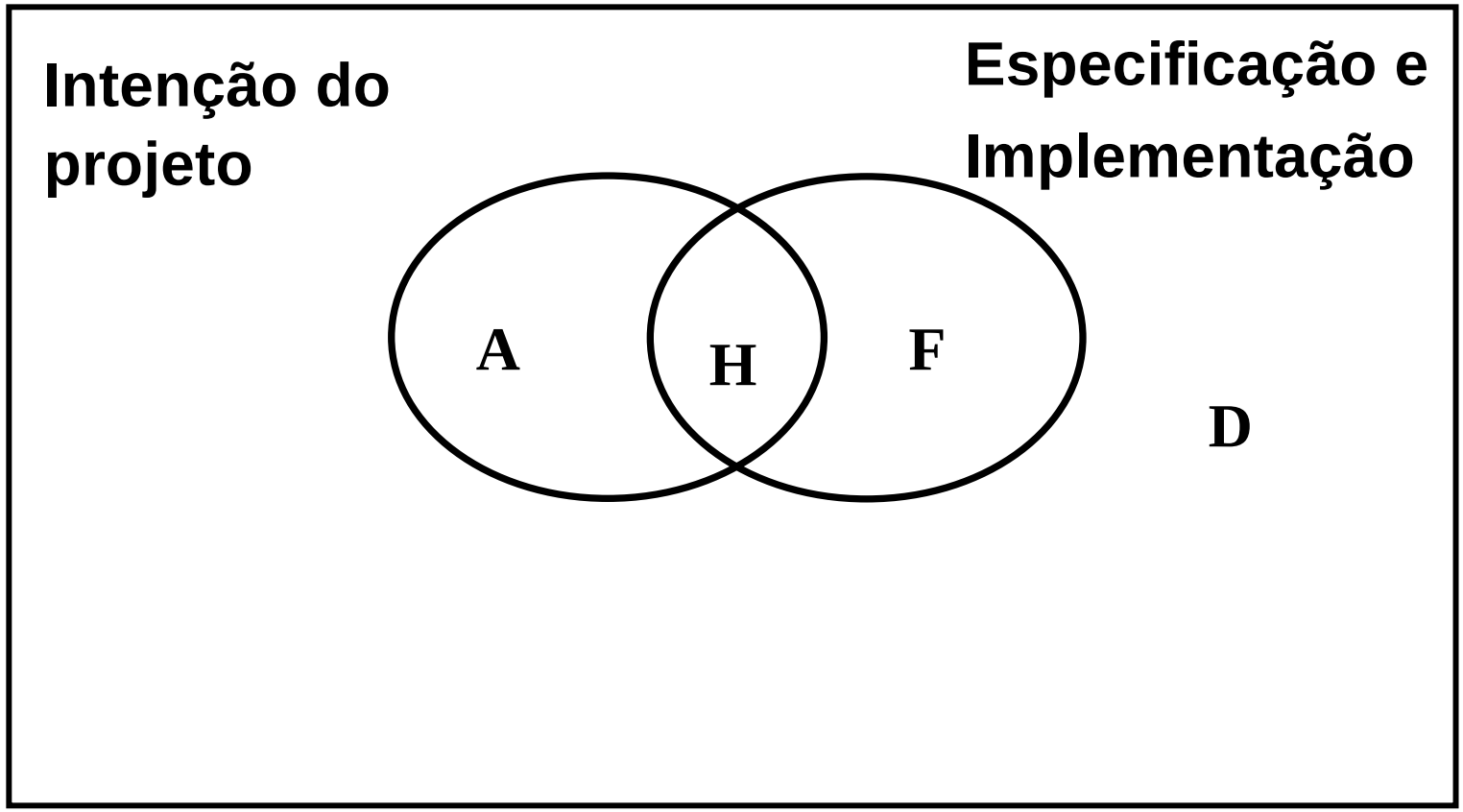
Projeto x Verificação



Projeto x Verificação



Projeto x Verificação



Nível hierárquico adequado

- Verificação funcional pode ser realizada a vários níveis:
  - componente/unidade/sub-unidade, ...
  - ASIC/FPGA/IP
  - Sistema/SOC
  - placa

Nível hierárquico adequado

- Como decidir qual nível?
  - Nível mais baixo fornece mais observabilidade mas exige mais esforço,
  - A nível mais alto os elementos menores são verificados implicitamente com menos esforço desde que os cenários e vetores de entrada sejam completos.
- Decisão depende do projeto.
- Faz parte do plano de verificação
  - para cada elemento existe uma seção nele.

Verificação em vários níveis

- Na verificação no nível de sistema é suficiente verificar a interação dos componentes se uma verificação correta a nível de componentes foi feita.
- Verificação entre dois níveis adjacentes, do topo até a base.

Verificação funcional

- Para realizar a verificação funcional necessita-se de:
  - Um projeto a ser verificado denominado DUV (Design Under Verification).
  - Um Modelo de Referência que implementa as funcionalidades do DUV de forma ideal.
  - Um ambiente de verificação (testbench).
    - Responsável por gerar estímulos para o DUV e comparar as respostas do DUV com as respostas do Modelo de Referência.

Verificação funcional

- Tudo a mesma coisa:
  - UUV (Unit Under Test).
    - unidade a ser testada.
  - MUT (Model Under Test).
  - DUT (Device Under Test, Design Under Test).
  - EUV (*Entity Under Verification*).
  - DUV (Design Under Verification).

Verificação funcional

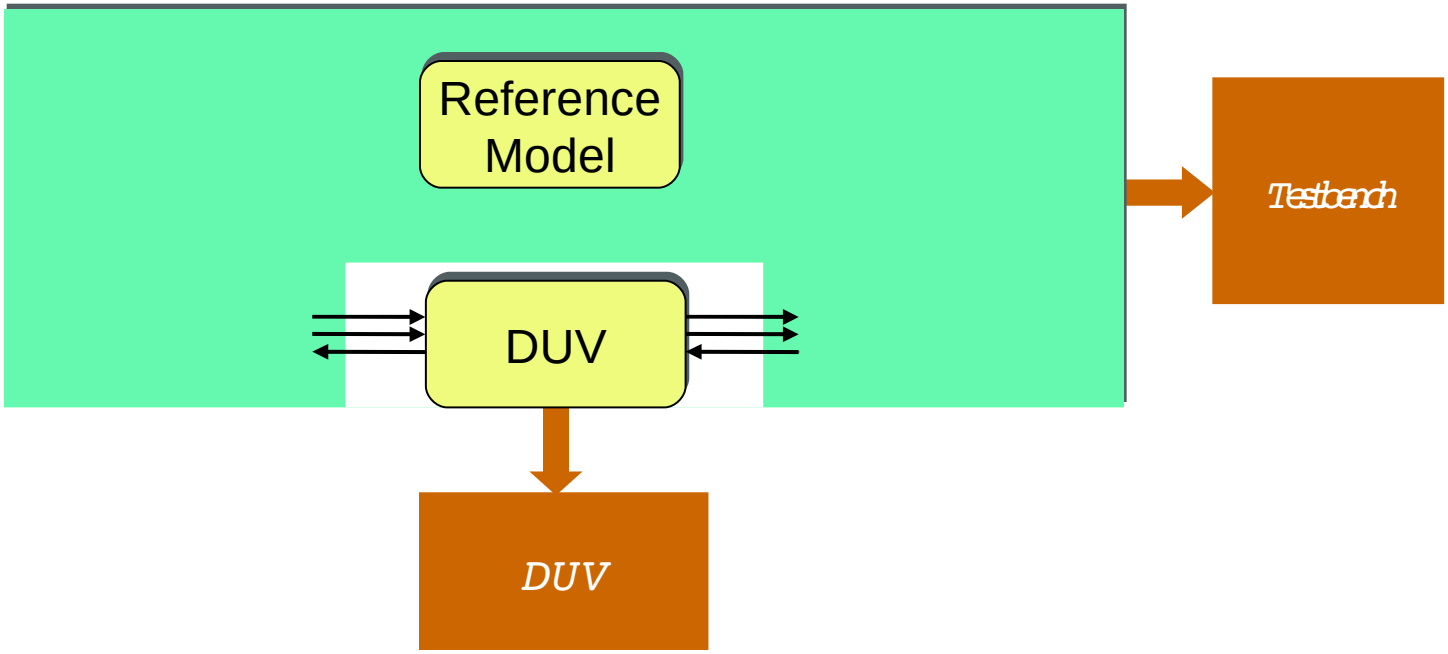
- Modelo de Referência pode ser escrito em qualquer linguagem que possa se comunicar com SystemC.

- Normalmente escrito em C, C++, podendo ser escrito em outras linguagens.

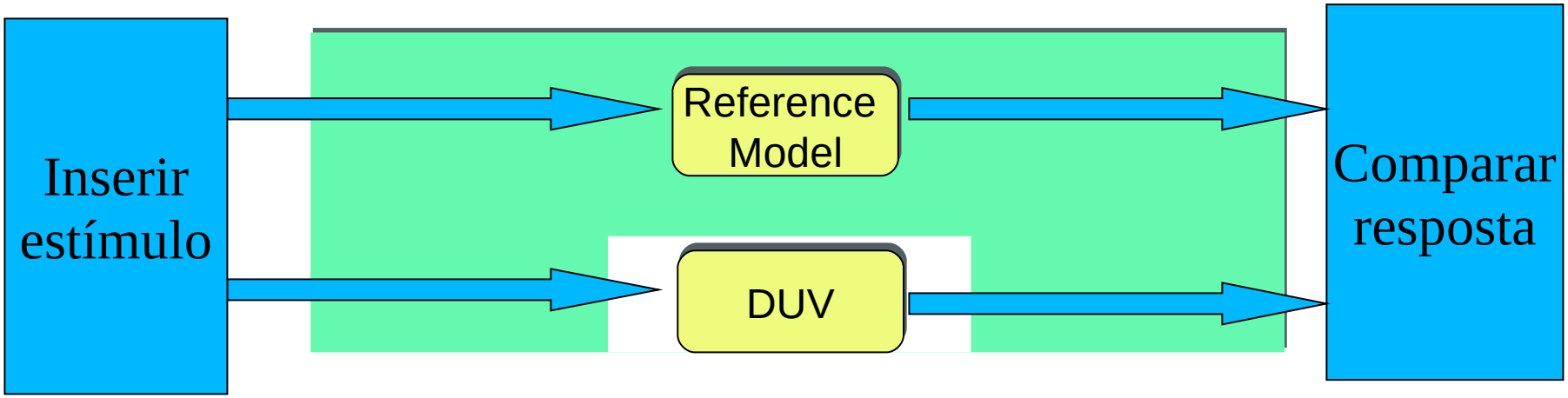
Verificação funcional

- Testbench
  - Montagem de teste para simulação.
  - Código escrito em SystemC.
  - Cria estímulos e verifica a resposta.
  - Não tem entrada nem saída.
  - Um modelo do universo em volta do projeto.
  - Imprime mensagens quando o DUV apresenta comportamento inesperado.
  - Caso tudo está ok imprime uma única mensagem no final.

Verificação funcional



Verificação funcional



Verificação funcional

- Verificação funcional deve:
  - **Ser dirigida pela cobertura (Coverage-driven)**
    - Quando parar de simular ?
    - Cobertura - processo que mostra que as funcionalidades especificadas estão sendo exercitadas.
    - Auxilia na análise de redirecionamento de estímulo.
    - Mostra o progresso da verificação.
    - Determina quando terminar a verificação.
  - Possuir estímulos
    - Direcionados.
    - Corner-cases.
    - Reais.

Verificação funcional

- Verificação funcional deve:
  - **Apresentar estímulos com aleatoriedade direcionada (Random-constrained)**
    - Estímulos são gerados baseados em uma distribuição de probabilidade direcionada para o DUV.
    - Visa alcançar os critérios de cobertura.
    - Encontra erros não previstos.

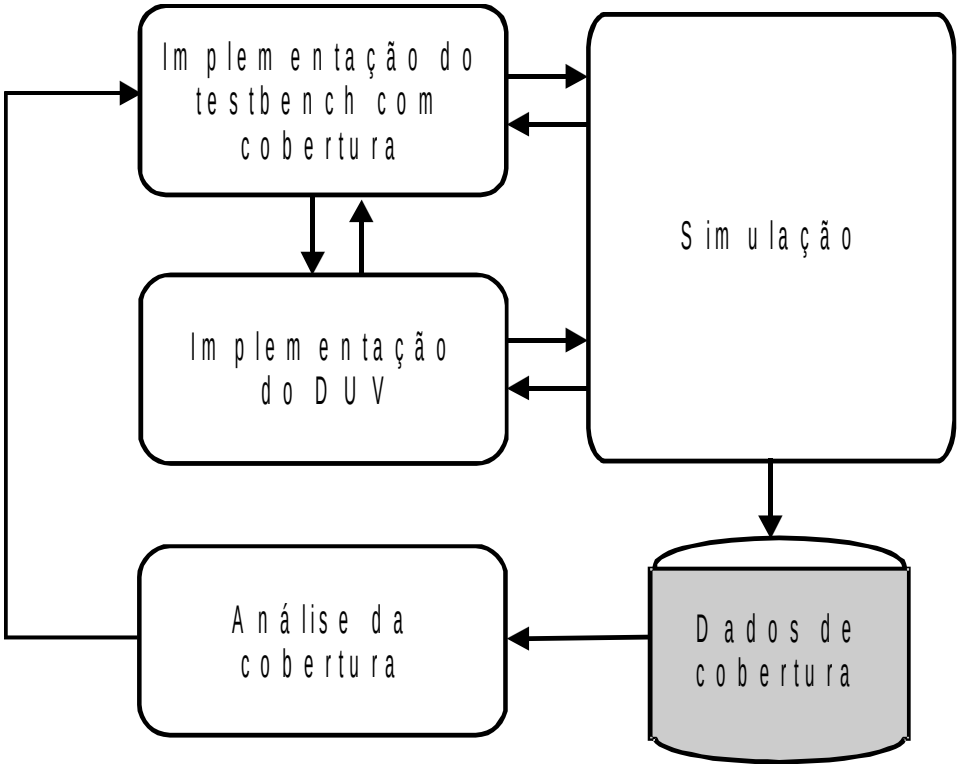
Verificação funcional

- Verificação funcional deve:
  - **Ser auto verificável (Self-checking)**
    - O ambiente de verificação deve comparar as respostas do DUV com as respostas do Modelo de Referência sem intervenção humana.

Verificação funcional

- Verificação funcional deve:
  - **Ter o testbench implementado no nível de transação (Transaction-level)**
    - Transação: uma estrutura de instruções e/ou dados que tem início e fim no tempo.
      - Exemplos:
        - » Pacote Ethernet
        - » Frame

Resumo


Implementação do testbench com cobertura
Simulação
Implementação do DUV
Análise da cobertura
Dados de cobertura

Verificação

- Automação
  - Eliminar intervenção humana no processo.
  - Realidade mostra que não é possível:
    - processos mal definidos,
    - precisando de invenção e criatividade humana.
- Redundância
  - Usar dois engenheiros (ou grupos) para um verificar o outro.
    - Projetista
    - Engenheiro de verificação

Quem pode errar ?

- Um projetista pode implementar uma funcionalidade de forma errada ?

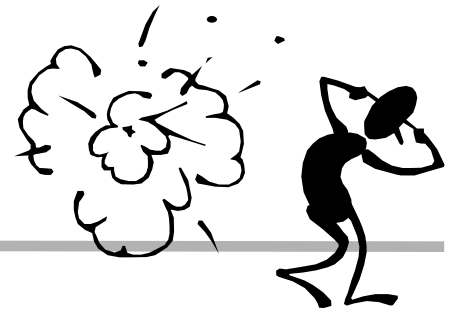
Sim, o erro será descoberto por um teste.

Um engenheiro de verificação pode testar uma funcionalidade de forma errada ?

Sim, um erro falso aparecerá no teste.

O projetista e o engenheiro de verificação podem cometer o mesmo erro ?

Não, o erro será aceito no teste.



Quem pode errar ?

- Um projetista pode esquecer de implementar alguma funcionalidade ?

Sim, a falha será descoberta por um teste.

Um engenheiro de verificação pode esquecer de testar alguma funcionalidade ?

Não, um possível erro do projetista passará despercebido.

Verificação funcional

- Pode provar a presença de erros, mas não pode provar a ausência de erros.

- É preciso saber quando pode-se terminar o processo de verificação.

  medição de ***cobertura***

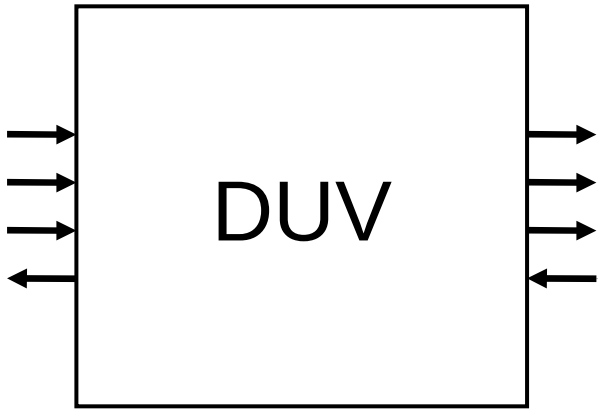
Abordagens de Verificação

- Black Box
- Grey Box
- White Box

Black Box

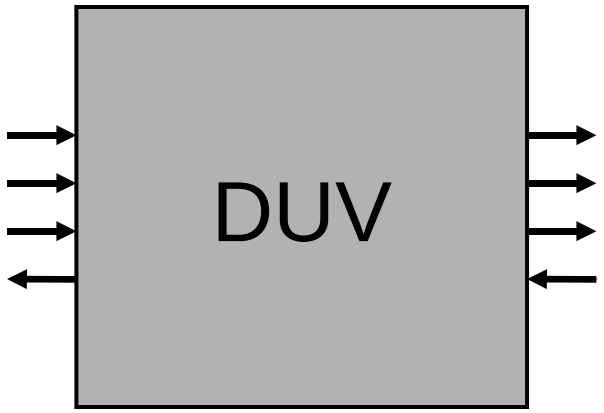
- Entradas, saídas, função
- Função bem documentada (ou não...)
- Para verificar, é preciso entender a função e prever as saídas sabendo as entradas.

White Box



- Todas as variáveis internas visíveis.
- Podem ser acessadas para verificação.
- Para teste de unidades pequenas nas folhas da hierarquia

Grey Box



- Uma seleção restrita de variáveis internas pode ser usado para verificação.
- Exemplo: registradores de um processador

Plano de Verificação

- Especificação do processo de verificação;
- Define o quê será verificado e como.
- Abordagem tradicional:
  - faça como quiser
- Abordagem nova:
  - uso de métricas para saber quando a verificação estiver completa: **Cobertura da Verificação**
  - idealmente a definição de sucesso “de primeira”.

Plano de Verificação

- Feito a partir da especificação do DUV.
- Define os cenários de teste (testbenches a serem escritos):
  - define a complexidade deles,
  - as dependências entre eles.
- A partir daí é feito um cronograma:
  - recursos (humanos, máquinas, etc.) necessários,
  - recursos disponíveis

Plano de Verificação

- Pertence à equipe:
  - todo mundo envolvido é responsável,
  - tudo mundo deve contribuir.
- Plano de Verificação não é algo novo, já é usado por:
  - NASA
  - FAA
  - Software

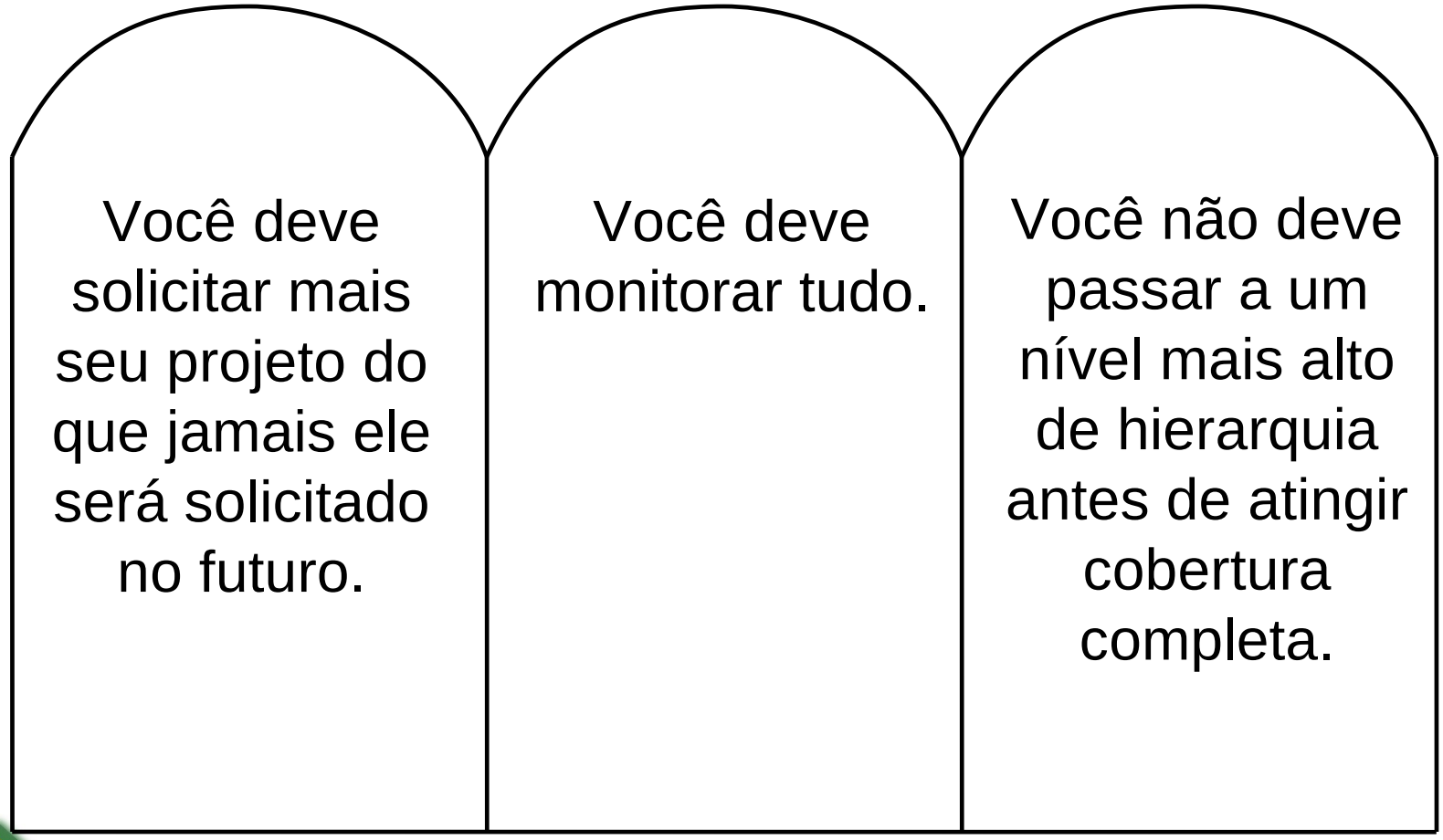
Conteúdo do Plano de Verificação

- Resumo do sistema
- Níveis de abstração
- Tecnologias de Verificação
- Modelos de referência a serem usados
- Fluxograma da verificação
- Definição dos estímulos
- Testes de regressão

Conteúdo do Plano de Verificação

- Gerência de falhas
- Plano de recursos
- Cronograma

Os três mandamentos da verificação funcional

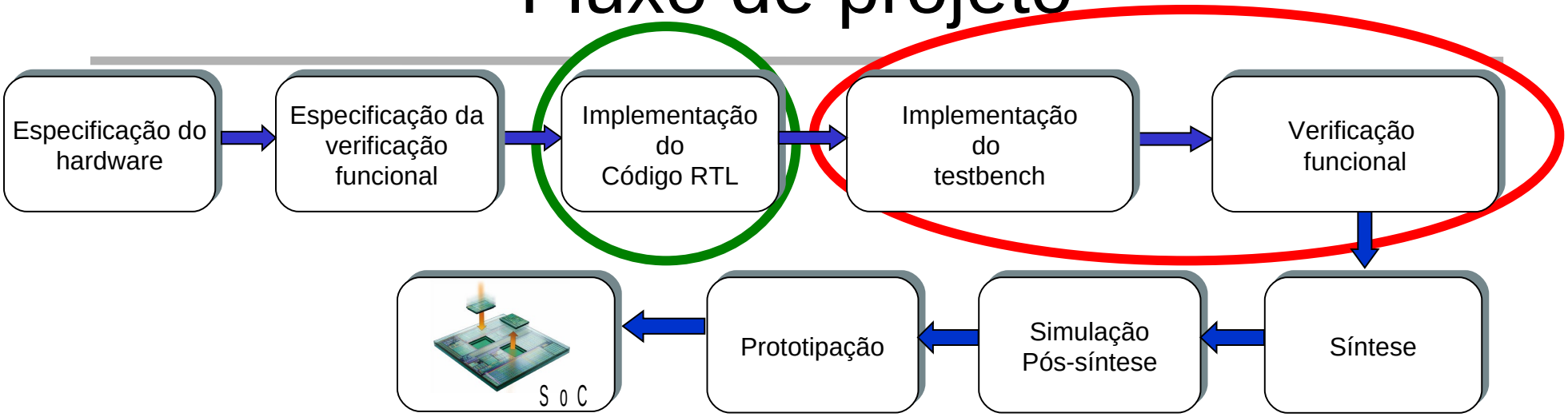


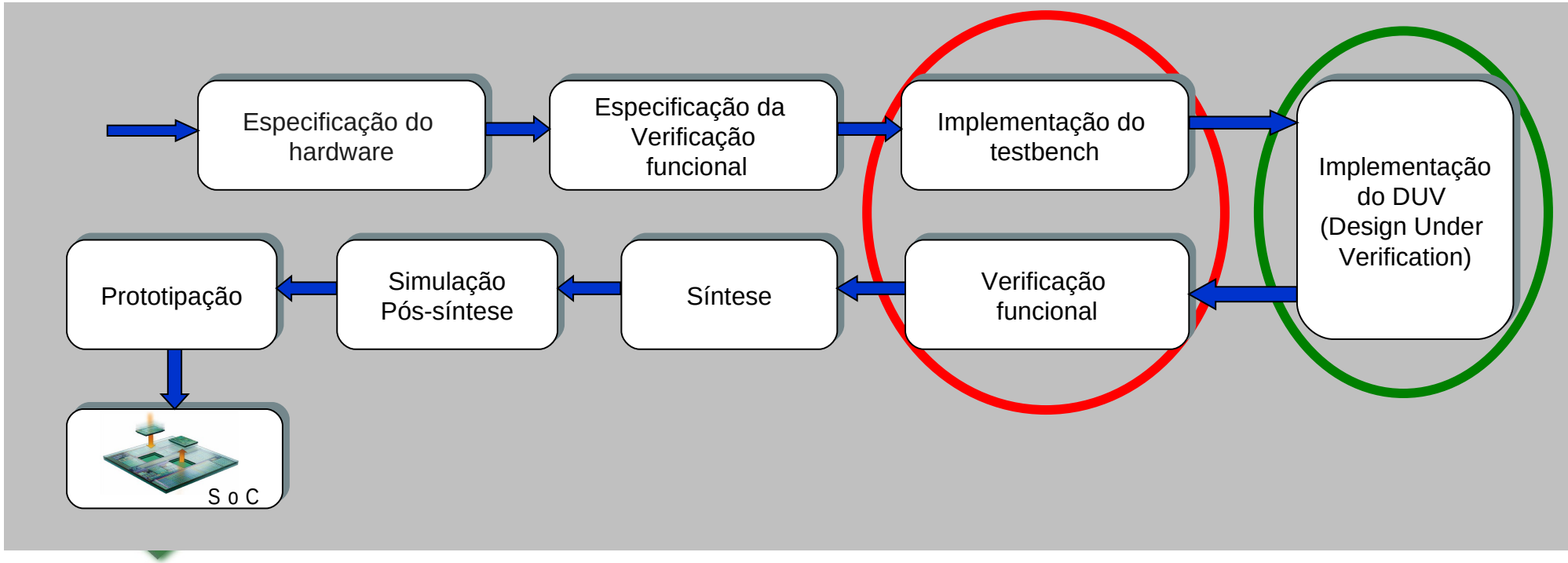
Metodologia VeriSC

Brazil-IP

- Metodologia VeriSC surgiu no contexto do projeto Brazil-IP.
- Brazil-IP é um esforço de universidades brasileiras para a capacitação de pessoas para a produção de IPs.
- Foi defendida como tese de doutorado.

Fluxo de projeto

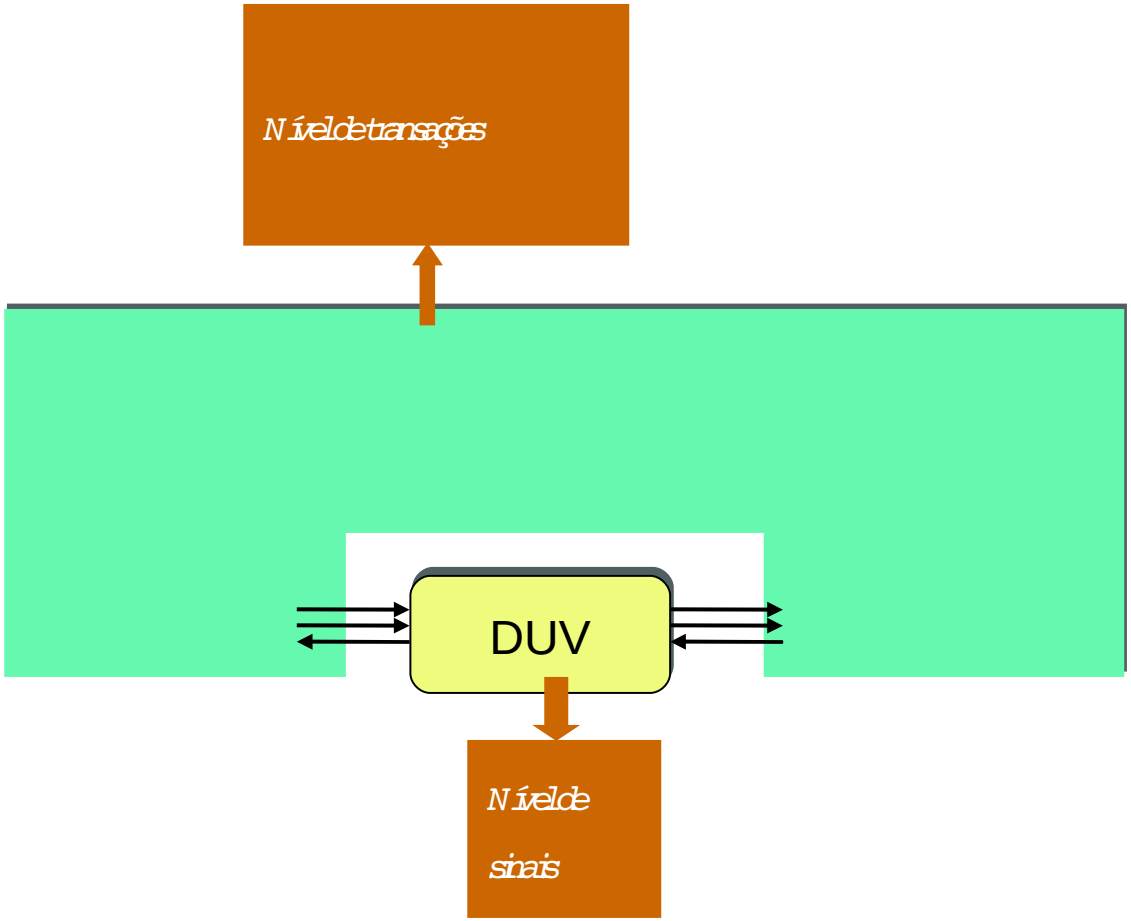




Metodologia VeriSC

- Inicia pela implementação do testbench.
- Implementa um mecanismo para simular a presença do DUV unicamente com os elementos do testbench, sem usar nenhum artifício extra.
- Todas as partes do testbench podem estar prontas e simuladas antes do início do desenvolvimento do DUV.

Testbench da metodologia VeriSC



Testbench

- **Definição:**
  - Montagem em volta do *Design Under Verification*

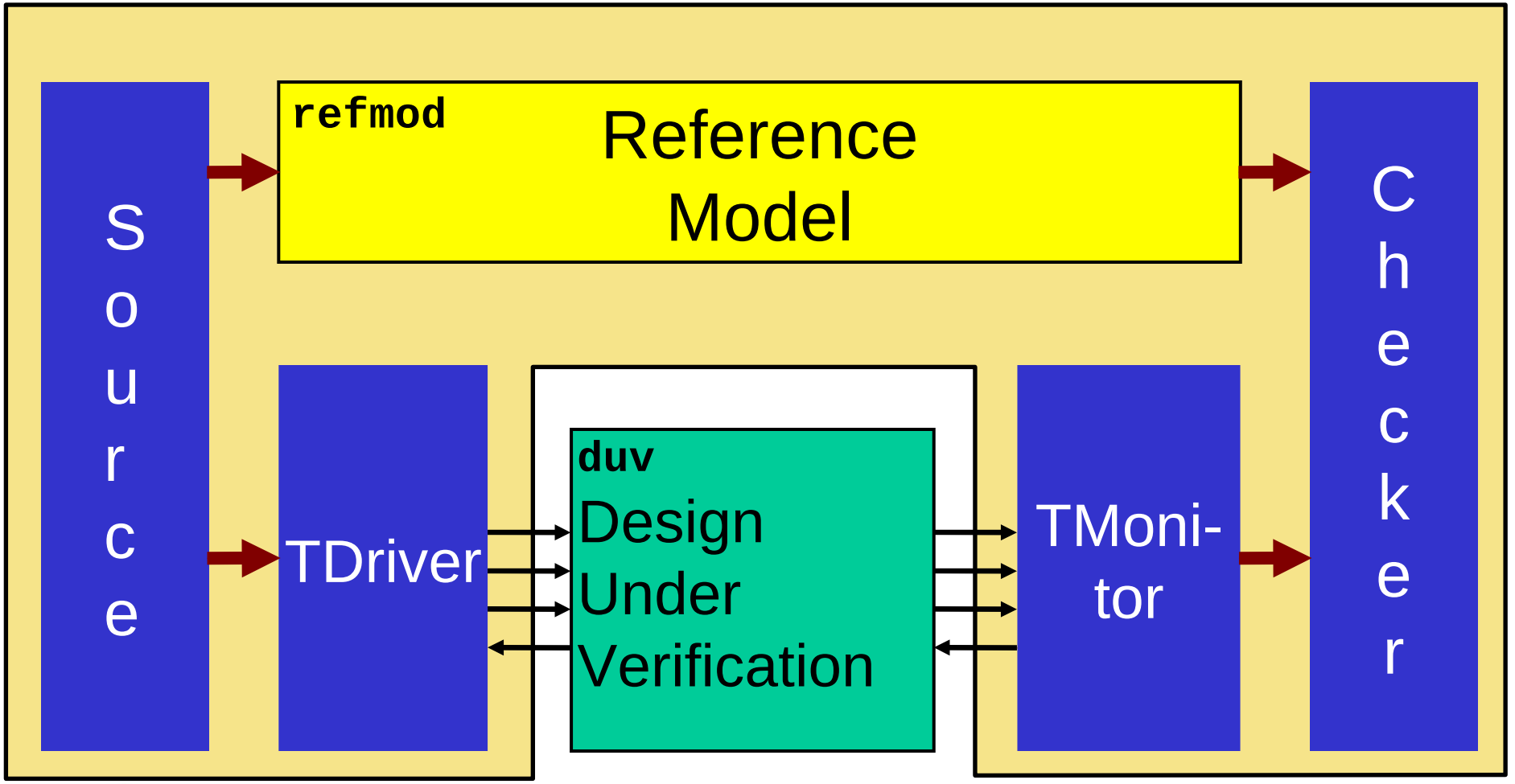
Transação

- **Definição:**
  - Uma operação que inicia num determinado momento no tempo e termina em outro.
- É caracterizada pelo conjunto de instruções e dados necessárias para realizar a operação.
- Exemplos:
  - transmissão de um pacote ethernet
  - recepção de uma imagem
  - uma escrita num barramento
  - execução de uma instrução de máquina de um processador

channel

sinal

Testbench



Elementos de um testbench

- Source
  - Envia transações de entrada para o driver e para o reference model.
- Checker
  - Compara as transações de saída recebidas do monitor com as de um reference model.
  - É bom ser reutilizável, ou seja, depender pouco do DUV.
- Reference Model
  - Tipicamente *timeless*

Elementos de um testbench

- TDriver
  - Recebe transações de entrada e as converte em transições de sinais da interface de entrada do DUV.
- TMonitor
  - Observa sinais da interface de saída do DUV, implementa o protocolo de sinalização e gera transações de saída que ele repassa para o checker.

Regras de projeto

- TDriver
  - Acesso ao DUV somente pela interface do mesmo;
  - Transação flui do source para o driver mas nunca na direção oposta.
- TMonitor
  - Acesso somente pela interface do DUV;
  - Faz verificação de protocolo (baixo nível);
  - Envia transações ao checker;
  - Independente de driver e checker.

Regras de projeto

- Source
  - Não envia sinais diretamente para o DUV
- Checker
  - Nunca escreve (força sinal) dentro do DUV;
  - Pode eventualmente ler informação do DUV (por exemplo registradores internos);
- Reference Model
  - Modela a funcionalidade, mas não a interface

Ferramentas usadas na Metodologia VeriSC

- SystemC.
- SCV (SystemC Verification Library).
- Ferramenta eTBc (Easy Testbench Creator) de geração semi-automática de testbenches.

Passos da metodologia VeriSC

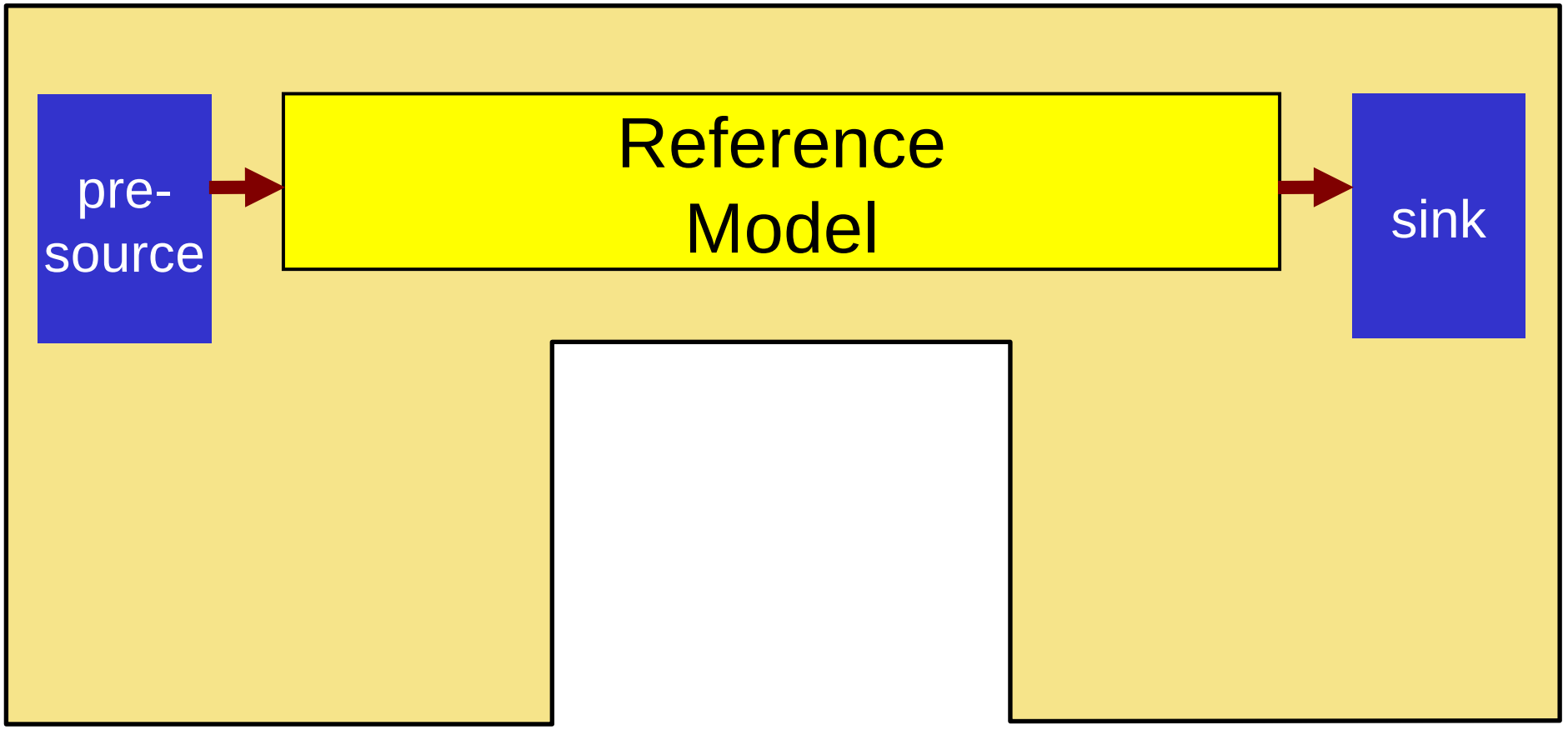
- Passo 1: Testbench Conception
  - 1.1 Single Refmod
  - 1.2 Double Refmod
  - 1.3 DUV Emulation
- Passo 2: Hierarchical Refmod Decomposition
  - 2.1 Refmod Decomposition
  - 2.2 Single Hierarchical Refmods
  - 2.3 Hierarchical Refmods Verifications

Passos da metodologia VeriSC

- Passo 3: Hierarchical Testbench
  - 3.1 Double Hierarchical Refmods
  - 3.2 Hierarchical DUV Emulation
  - 3.3 Hierarchical DUV
- Passo 4: Full Testbench

Passo 1: Testbench Conception

- 1.1 Single Refmod

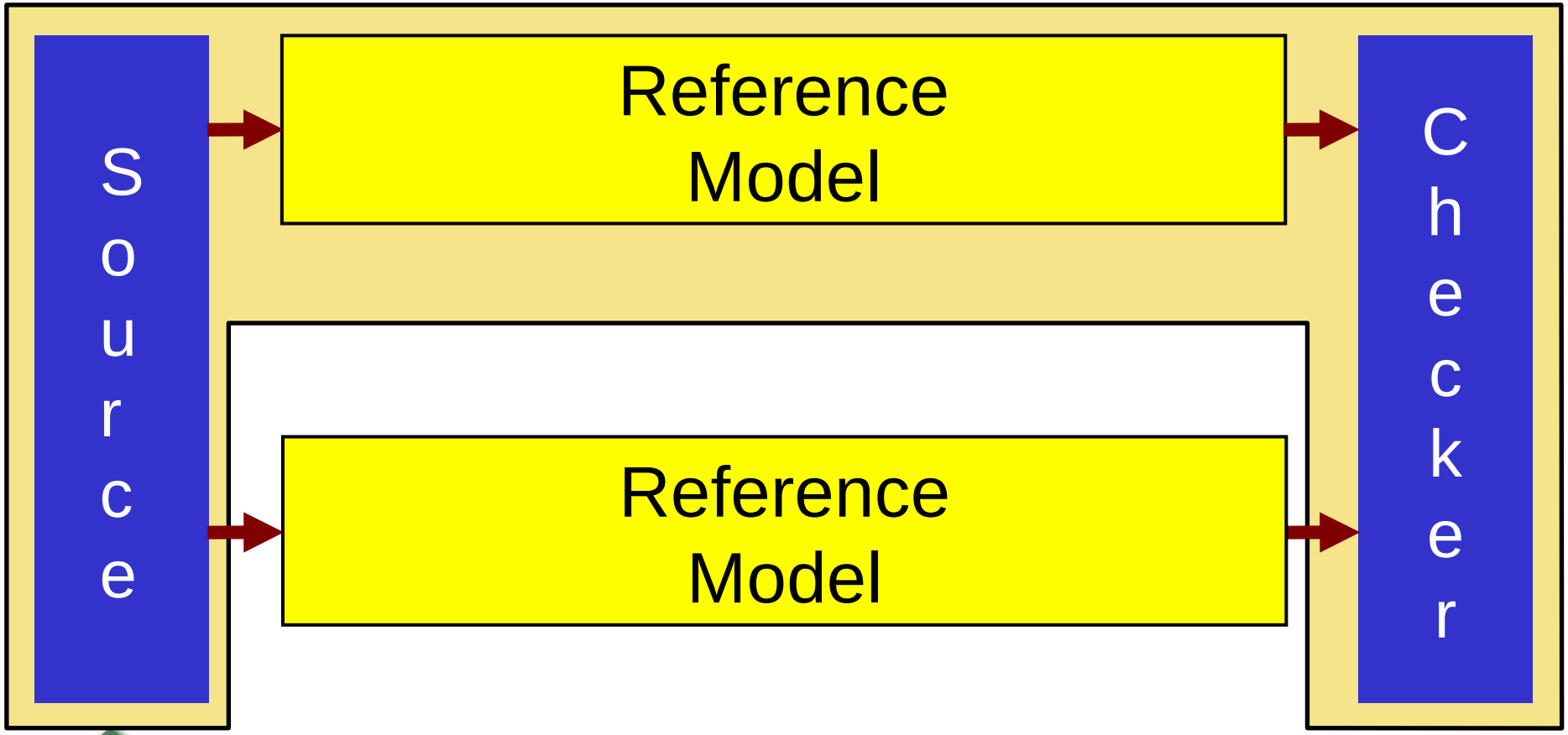


Passo 1: Testbench Conception

- 1.1 Single Refmod
  - Reference model é testado em sua capacidade de interagir com o testbench.
  - Pré-Source é um subconjunto do Source com quase as mesmas funcionalidades.
  - Sink possui um subconjunto das funcionalidades do Checker.

Passo 1: Testbench Conception

- 1.2 Double Refmod

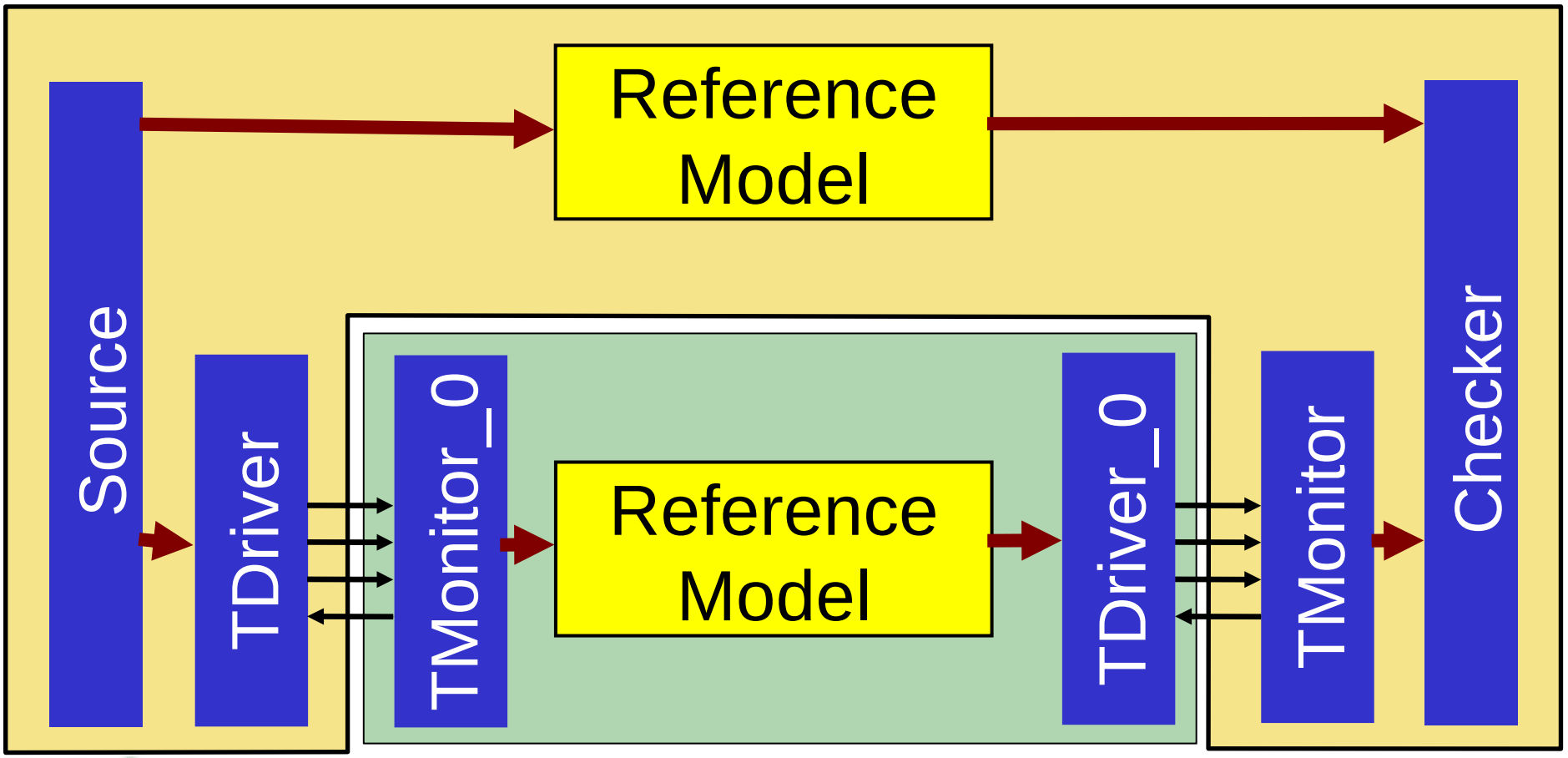


Passo 1: Testbench Conception

- 1.2 Double Refmod
  - Testar Source e Checker.
  - Pode-se inserir erros em uma das instâncias do reference model para testar a capacidade do Checker de detectá-los.
  - Checker emite mensagem somente em caso de ERRO.

Passo 1: Testbench Conception

- 1.3 DUV Emulation



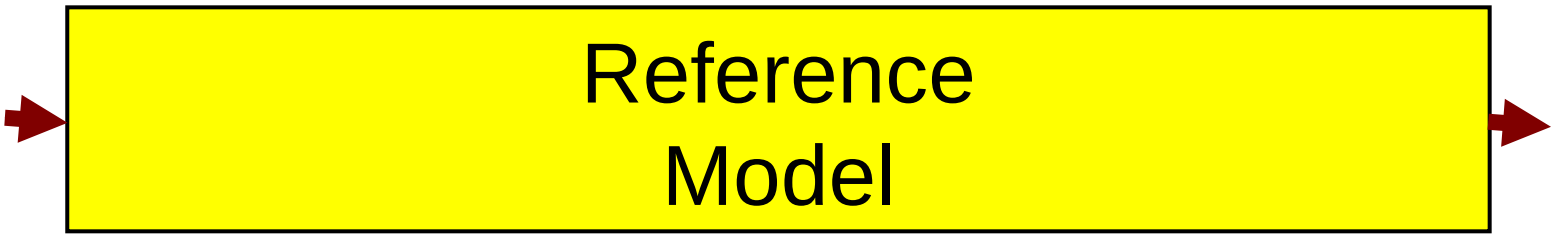
Passo 1: Testbench Conception

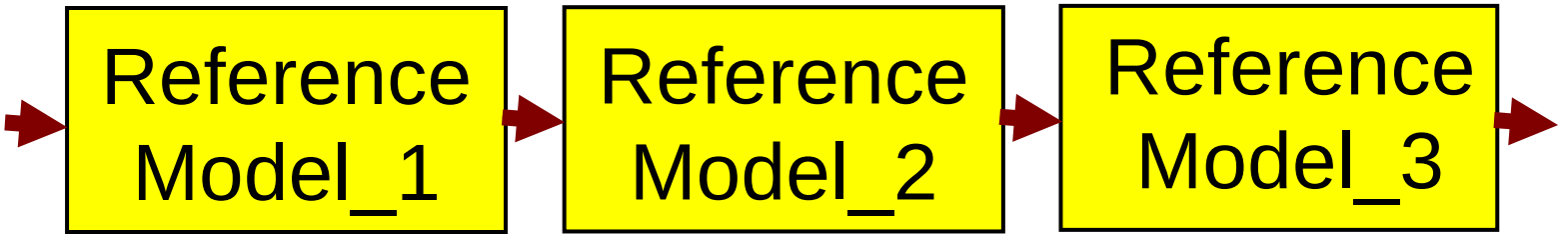
- 1.3 DUV Emulation
  - Testar TDriver(s) e TMonitor(es)
  - Tdriver-TMonitor_0 e Tdriver_0-TMonitor precisam ser simétricos

channel
sinal

Passo 2: Hierarchical Refmod Decomposition

- 2.1 Refmod Decomposition



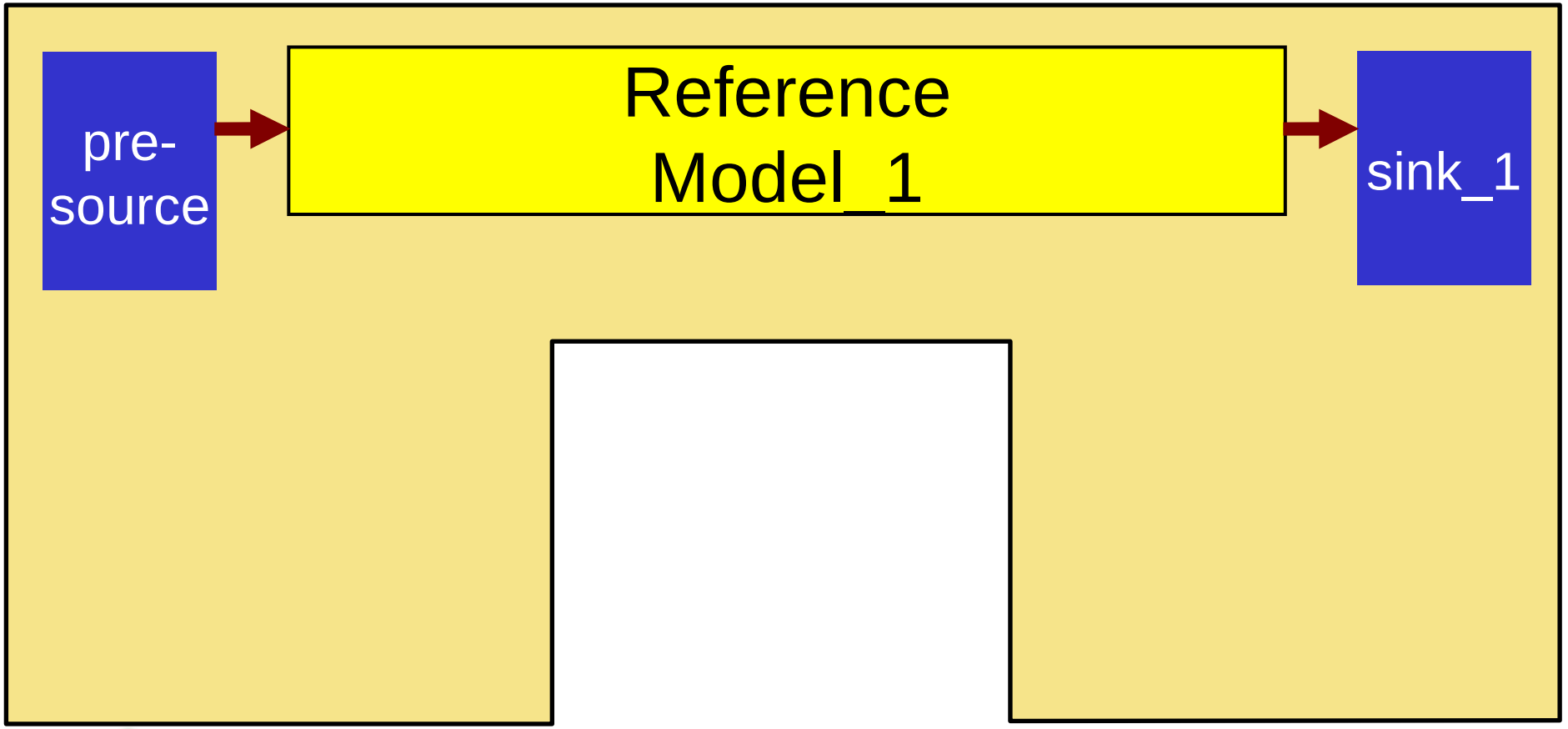


Passo 2: Hierarchical Refmod Decomposition

- 2.1 Refmod Decomposition
  - Reference Model é dividido hierarquicamente como o DUV.

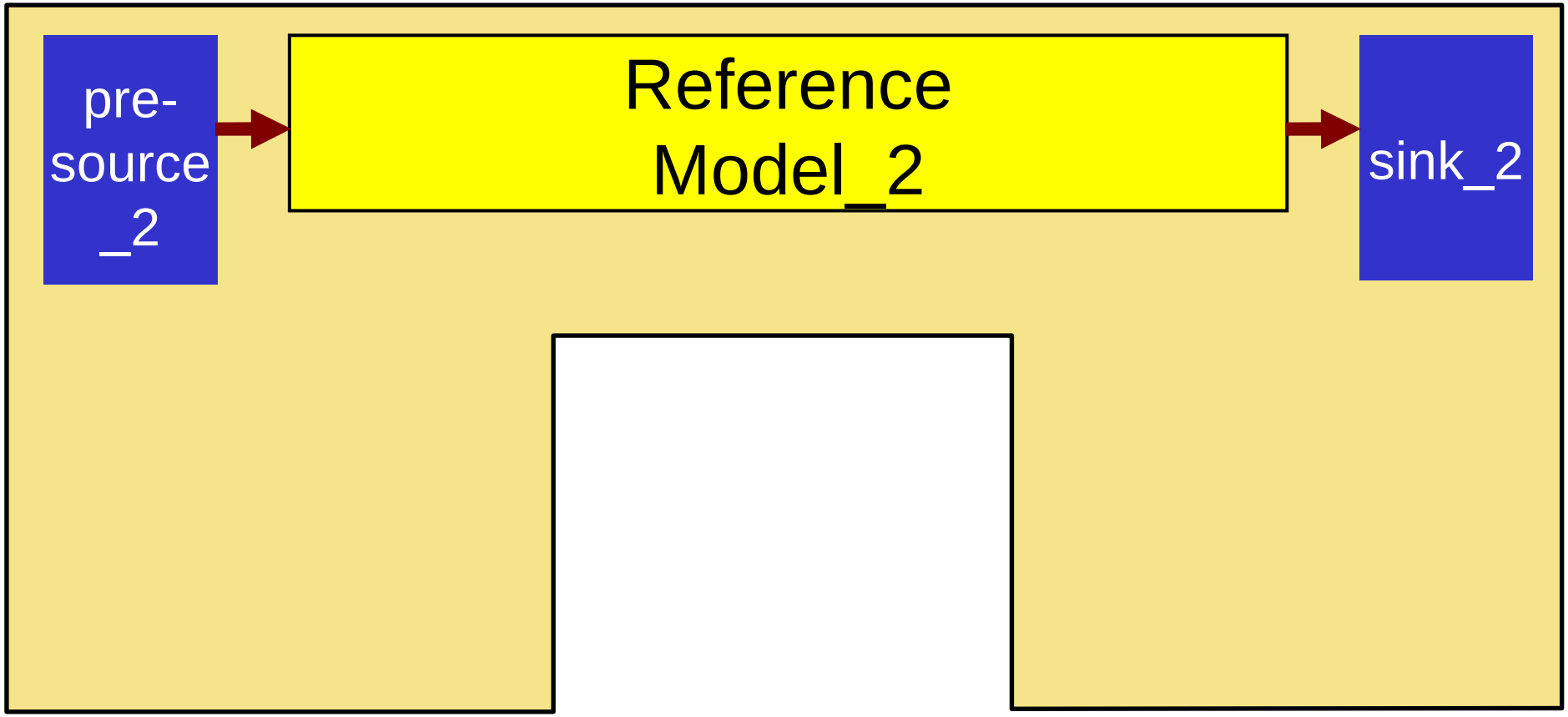
Passo 2: Hierarchical Refmod Decomposition

- 2.2 Single Hierarchical Refmods



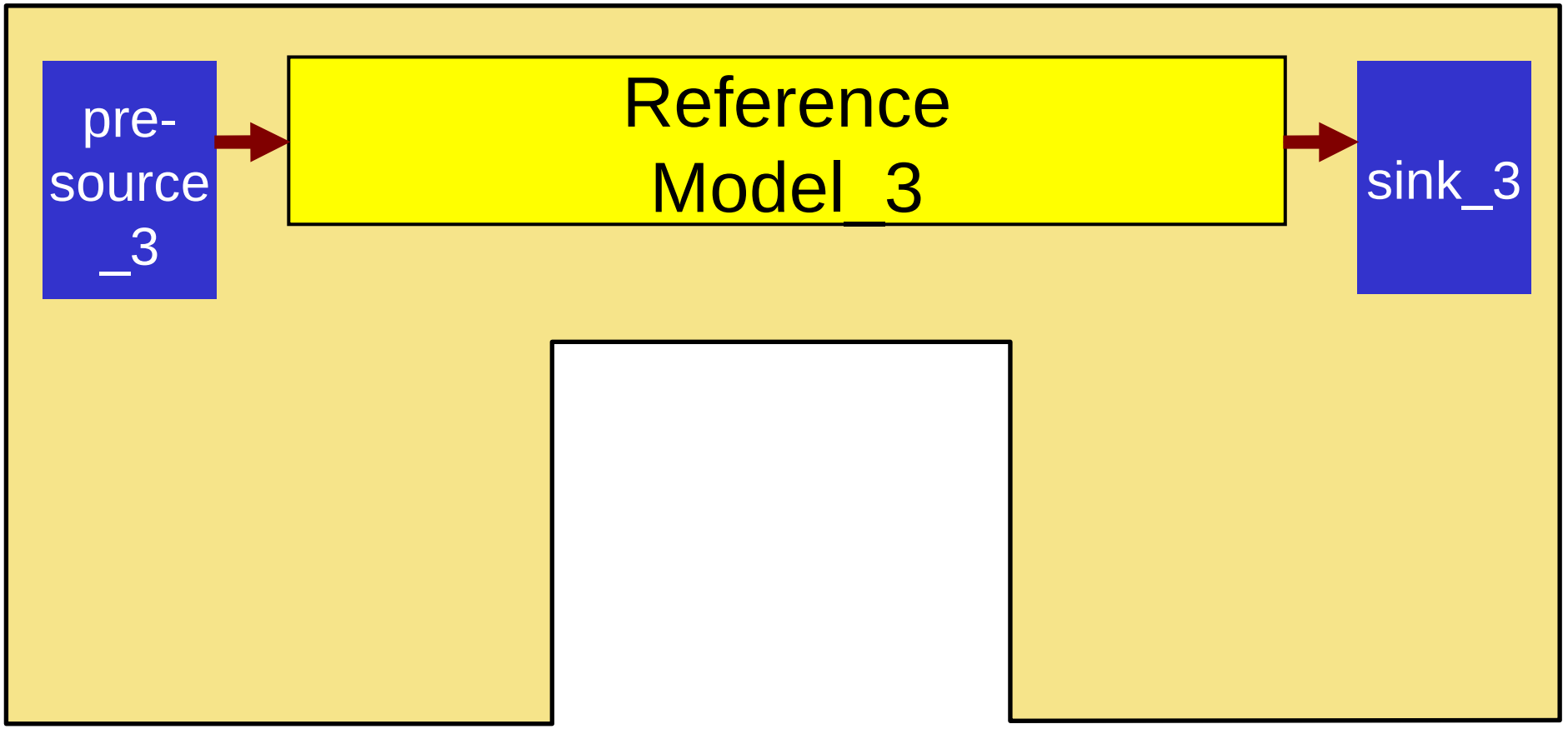
Passo 2: Hierarchical Refmod Decomposition

- 2.2 Single Hierarchical Refmods



Passo 2: Hierarchical Refmod Decomposition

- 2.2 Single Hierarchical Refmods



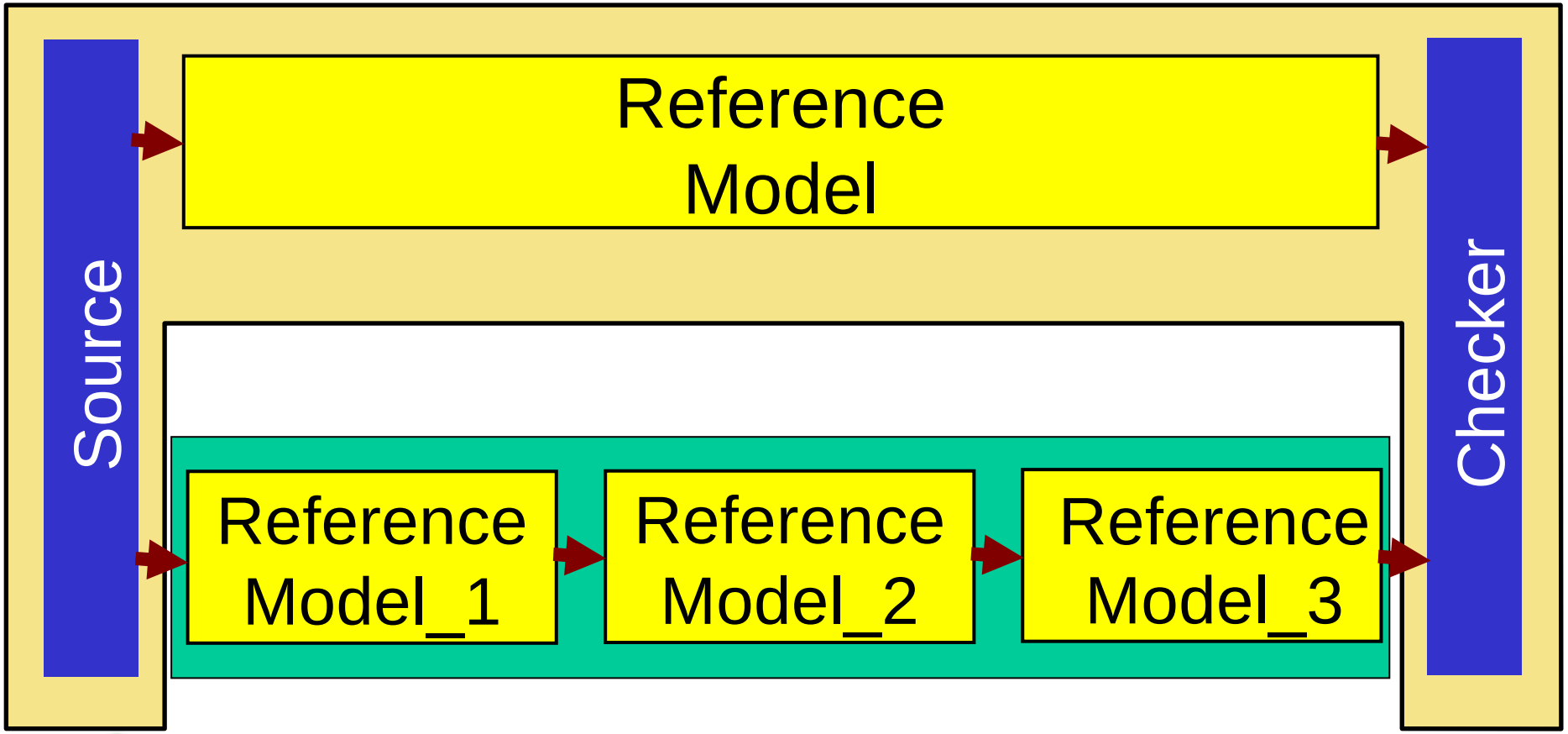
Passo 2: Hierarchical Refmod Decomposition

- 2.2 Single Hierarchical Refmods
  - Cada bloco do Reference Model deve ter suas entradas e saída testadas

channel
sinal

Passo 2: Hierarchical Refmod Decomposition

- 2.3 Hierarchical Refmods Verifications

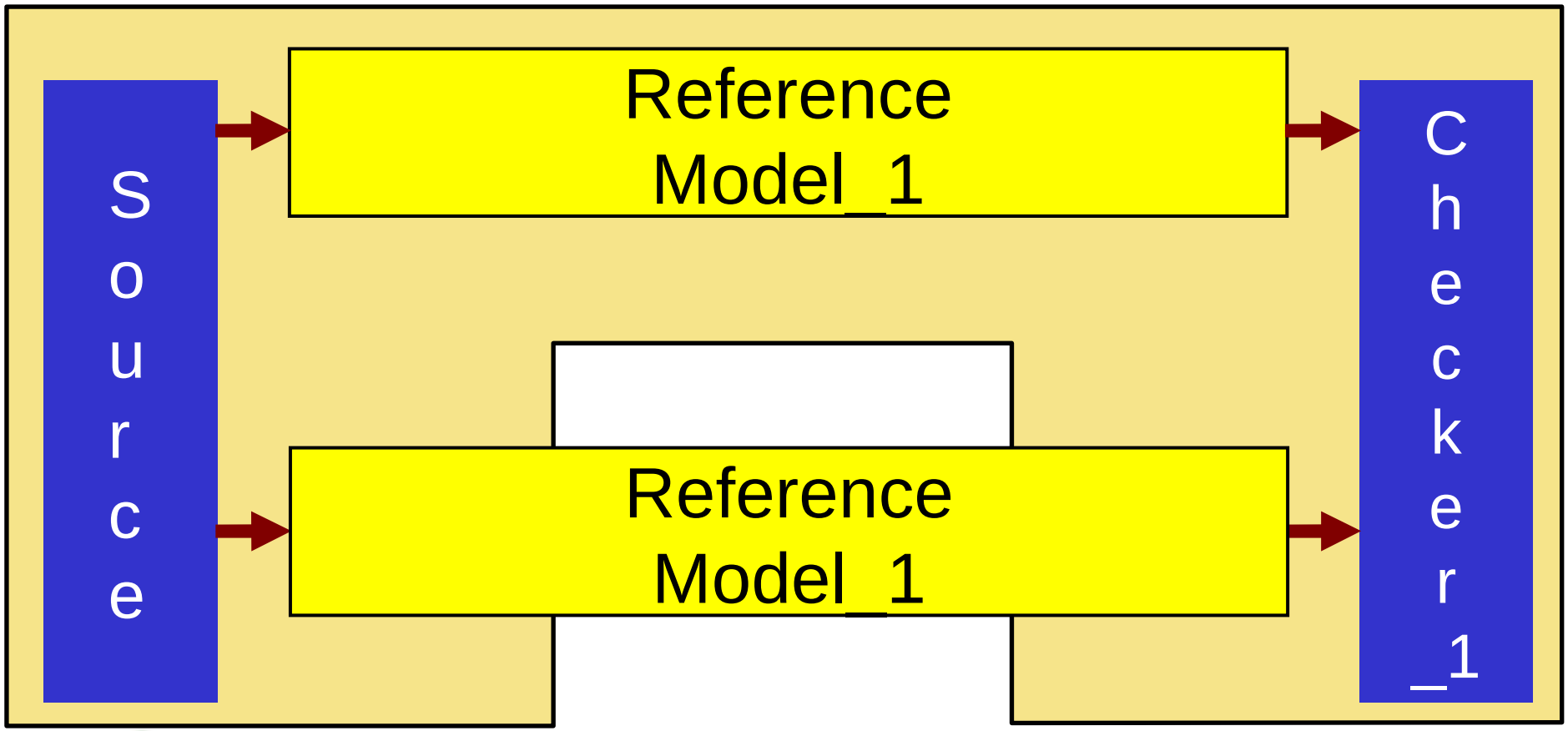


Passo 2: Hierarchical Refmod Decomposition

- 2.3 Hierarchical Refmods Verifications
  - A junção dos reference models hierárquicos deve ser comparada com o Modelo original para ver se ainda é equivalente.

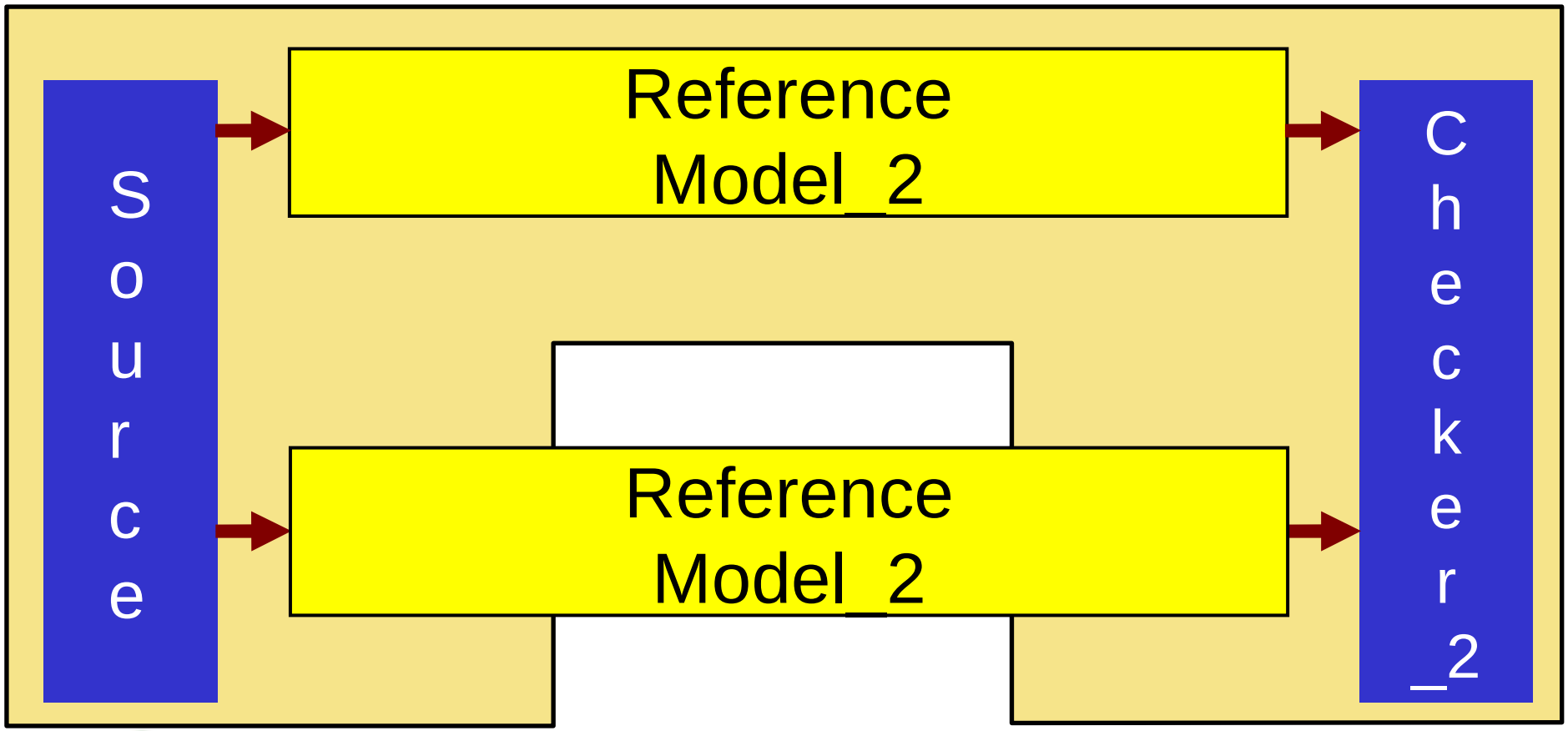
Passo 3: Hierarchical Testbench

- 3.1 Double Hierarchical Refmods



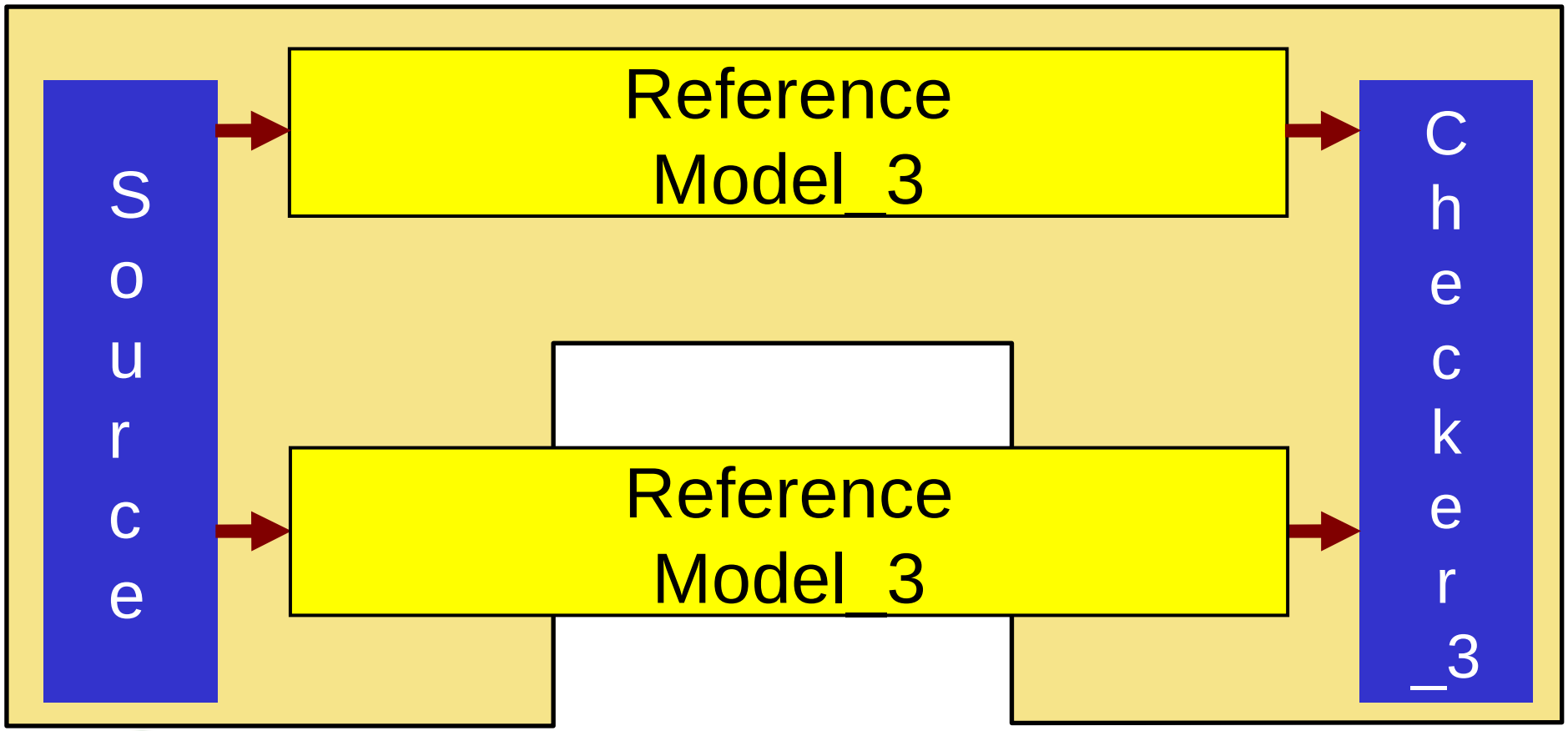
Passo 3: Hierarchical Testbench

- 3.1 Double Hierarchical Refmods



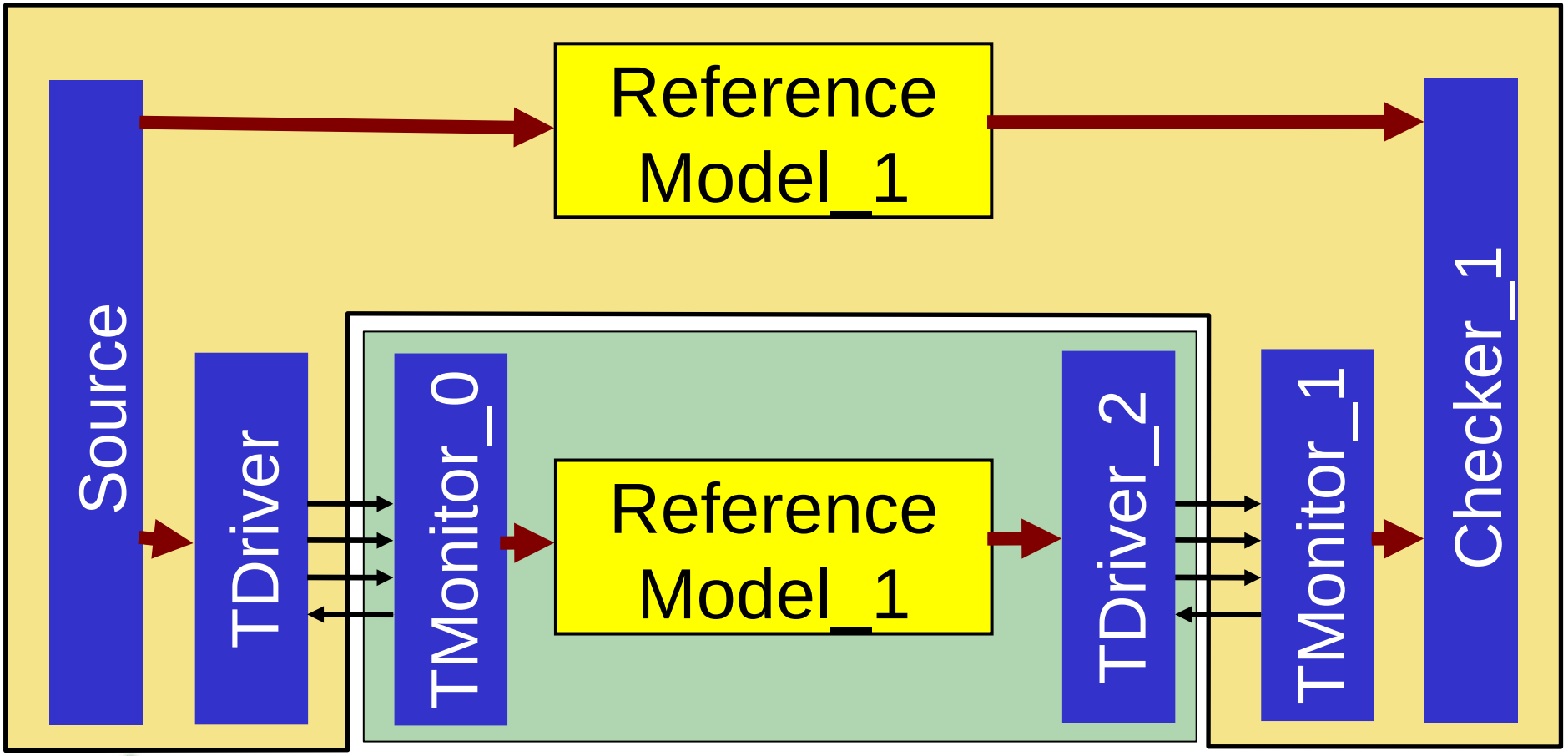
Passo 3: Hierarchical Testbench

- 3.1 Double Hierarchical Refmods



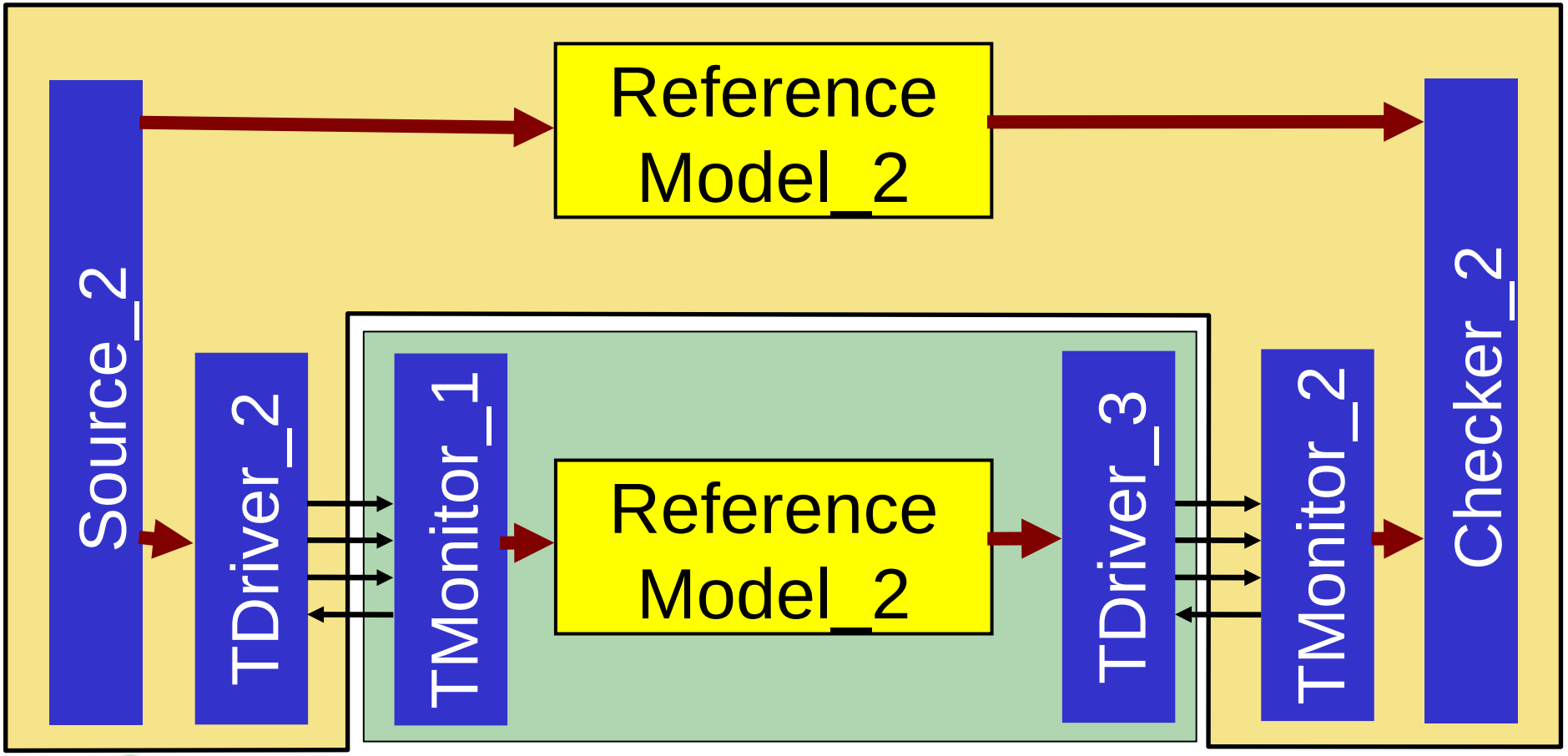
Passo 3: Hierarchical Testbench

- 3.2 Hierarchical DUV Emulation



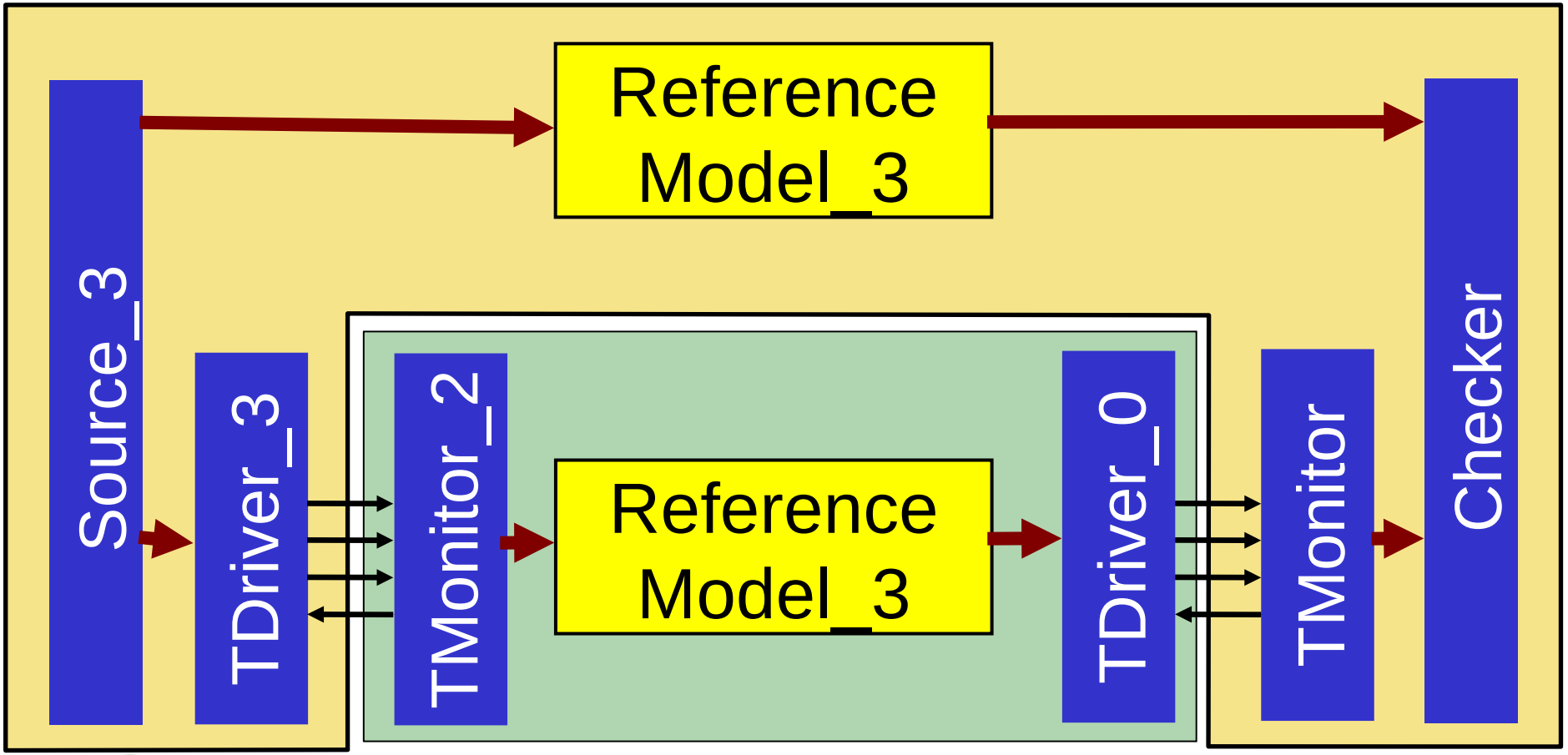
Passo 3: Hierarchical Testbench

- 3.2 Hierarchical DUV Emulation



Passo 3: Hierarchical Testbench

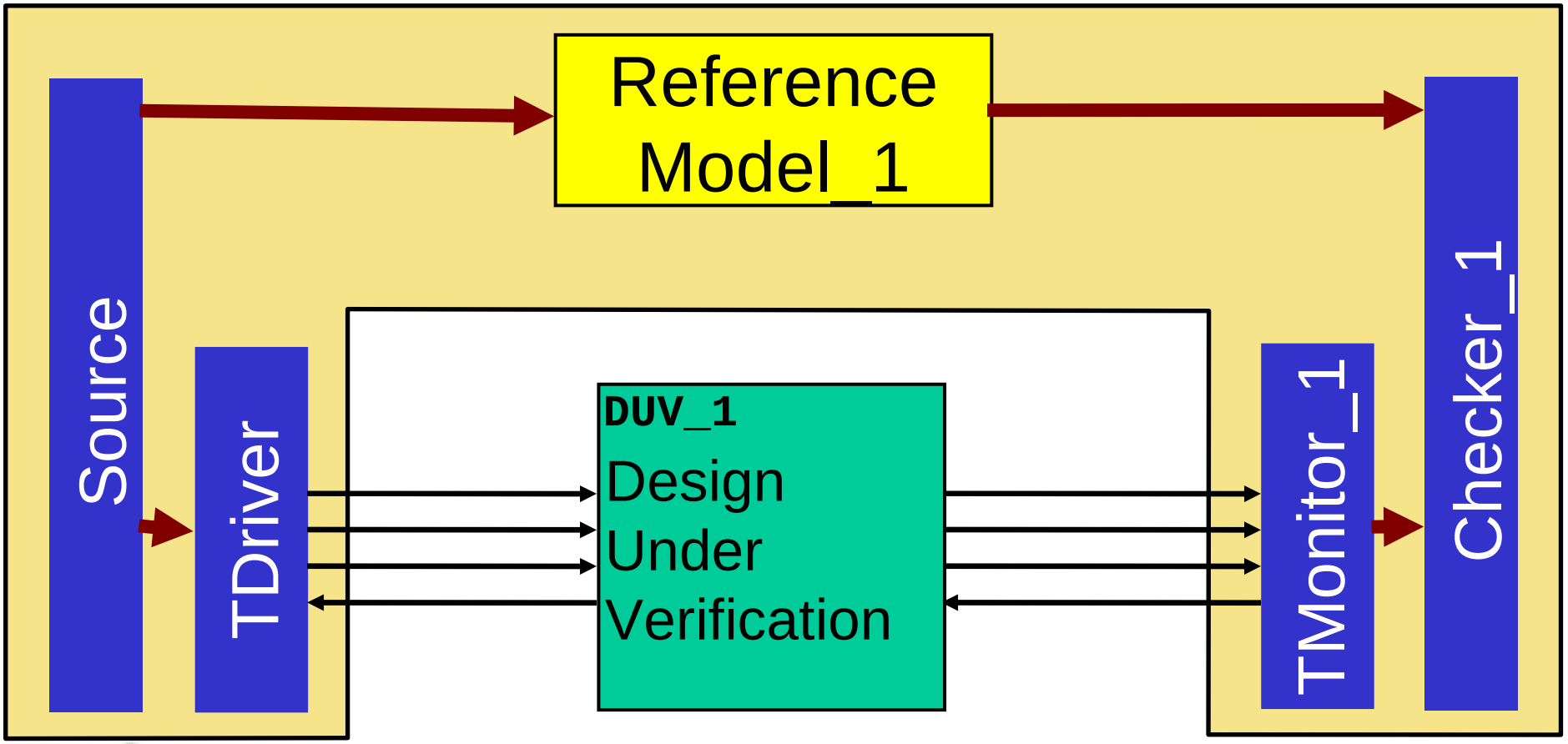
- 3.2 Hierarchical DUV Emulation



channel
sinal

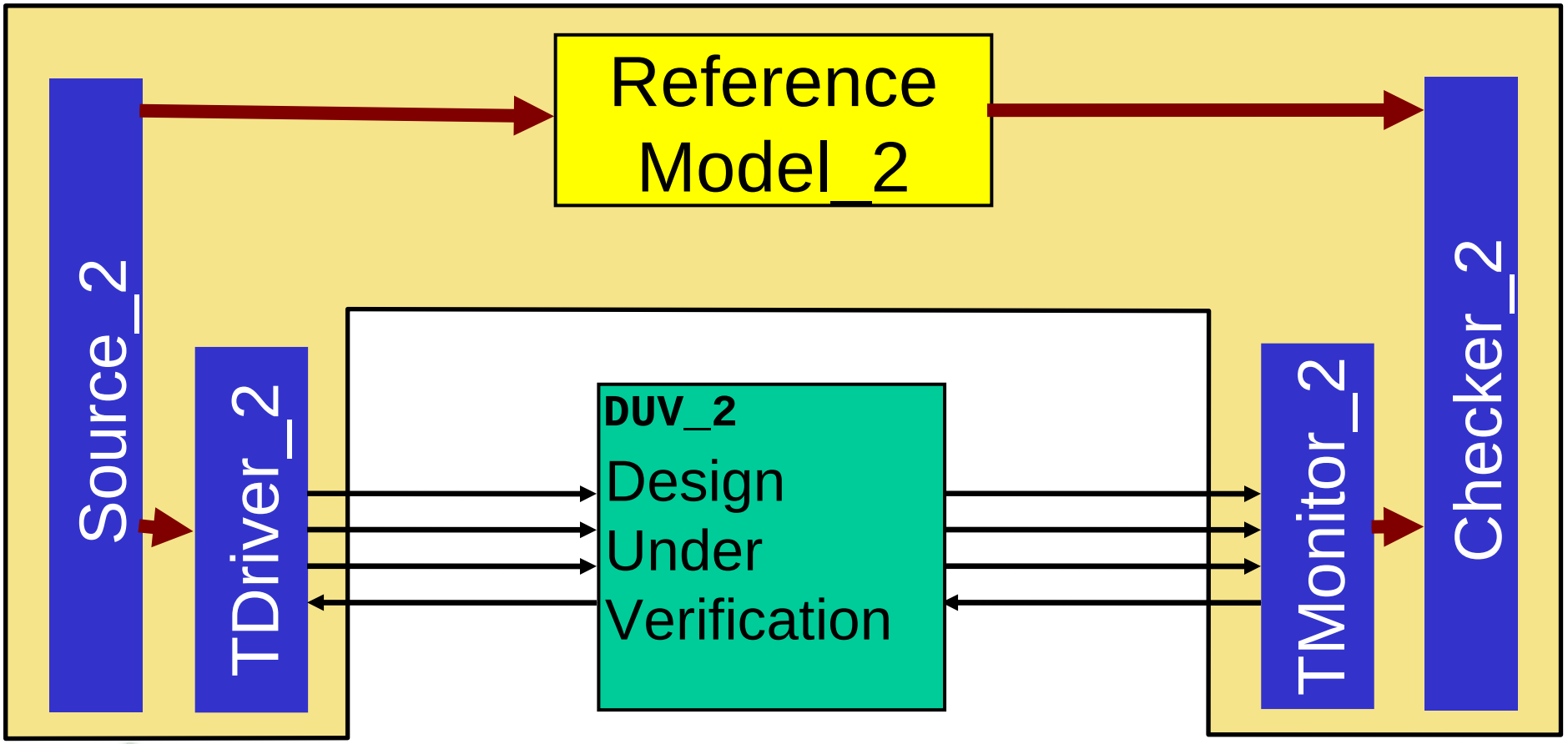
Passo 3: Hierarchical Testbench

- 3.3 Hierarchical DUV



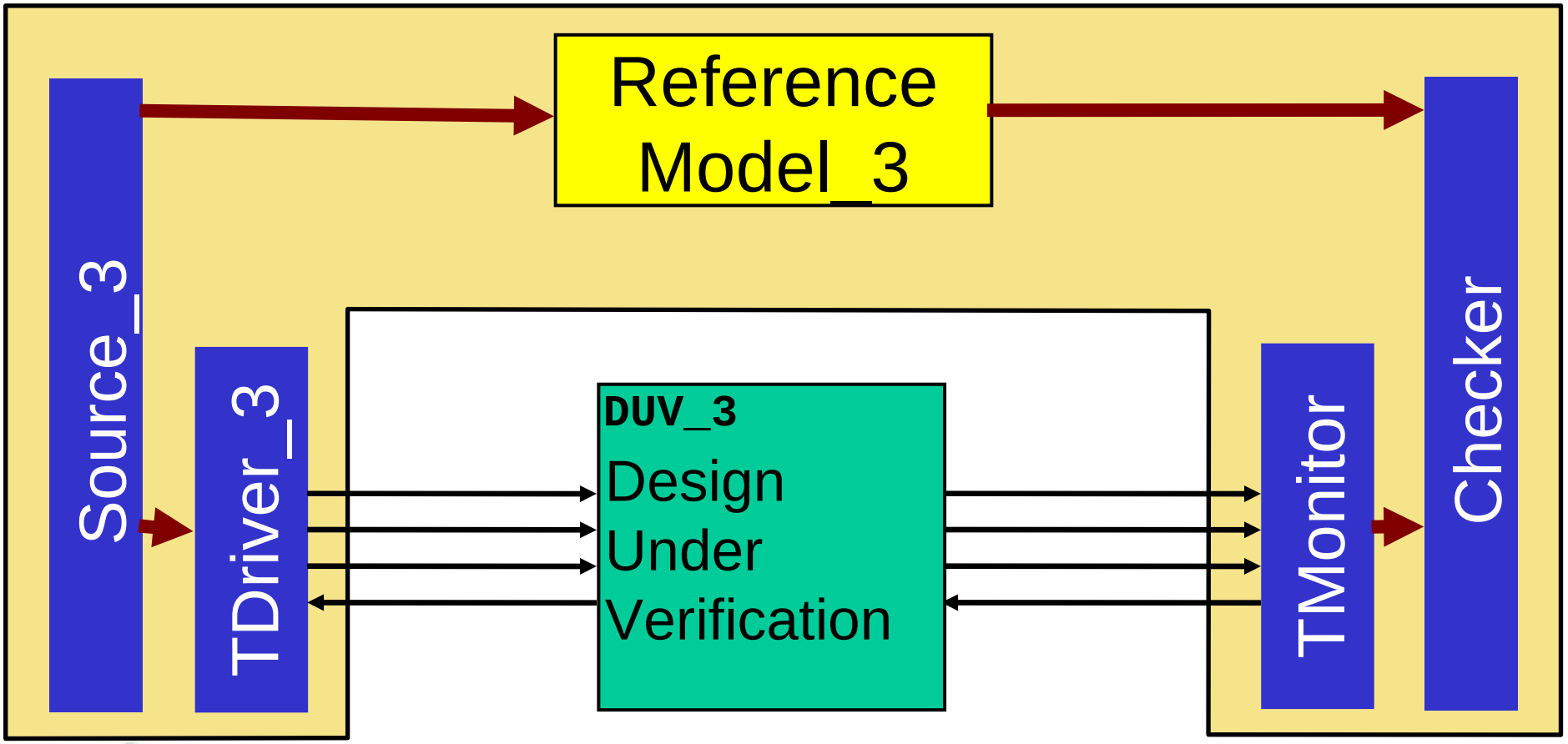
Passo 3: Hierarchical Testbench

- 3.3 Hierarchical DUV

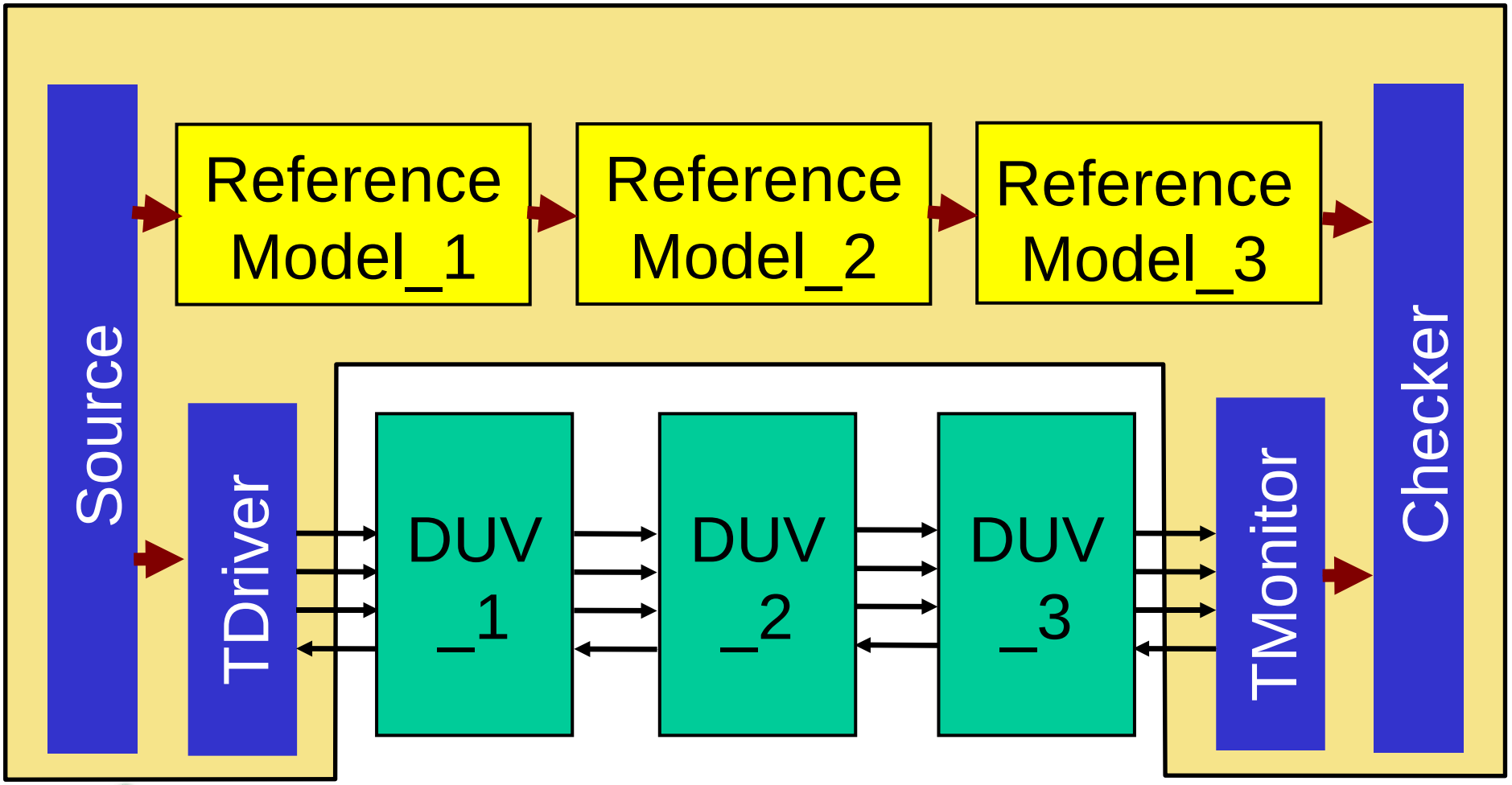


Passo 3: Hierarchical Testbench

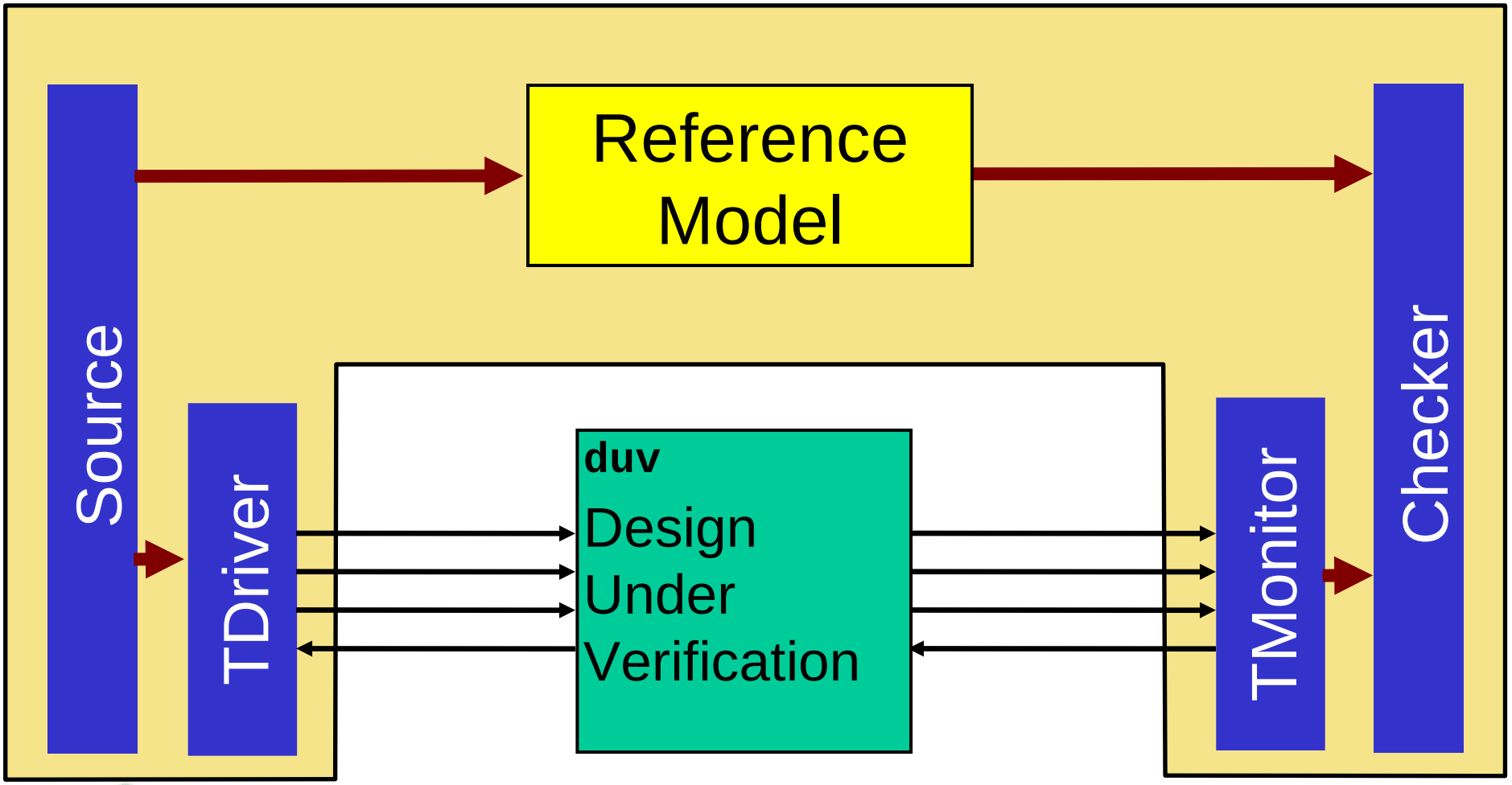
- 3.3 Hierarchical DUV



Passo 4: Full Testbench



Passo 4: Full Testbench



Metodologia VeriSC

- Testbench passa pela fase de depuração composta de passos, reduzindo a quantidade de erros.

- Muito reuso de código, diminuindo tempo de implementação do testbench.

A Arte da verificação

- Estou exercitando todos os possíveis cenários de entrada?
- Como vou saber se ocorreu um erro?

Ideal x Real

- Gerar todas as possíveis combinações de entrada tem custo inviável para unidades maiores.
- Testar muitos unidades pequenas também tem custo inviável.
- É necessário fazer uma escolha.

Tipos de Estimulos

- Compliance Testing
  - Verificar situações mencionadas na especificação
- Corner Case
  - Verificar situações críticas (extremas) do projeto
- Real Code
  - Utilizar estímulos reais da aplicação
- Random
  - Cria situações “inusitadas”
  - Cobertura tipicamente melhor do que os outros tipos porque pode gerar cenários que seriam esquecidos.

A Arte da verificação

- Forma mais óbvia: inspeção visual de formas de onda:
  - muito usado,
  - impossível de reaproveitar,
  - passível a mal interpretação (erro humano),
  - impraticável para muitas transações.
- **Use a visualização de formas de onda somente para depuração.**

A Arte da verificação

- Sniffers
  - Possuem interface dedicada para monitorar e extrair estatísticas dos sinais de um módulo.
  - Muito usado na depuração de sinais.

A Arte da verificação

- Self Checking Testbench
  - TDriver e TMonitor implementados em código do simulador (SystemC);
  - Checker implementado em SystemC compara dados recebidos do TMonitor com dados esperados;

Implementação do checker

- A simulação de rodar quieto enquanto tudo estiver o.k. para não afogar mensagens de erro no meio de outras mensagens.
- Quando aparece um erro, devem ser fornecidas todas as informações disponíveis sobre o problema para facilitar a depuração.
- Inserir erro no DUV para ver se o Checker funciona.

Implementação do Reference Model

- Refmod implementado em código
  - SystemC, C++, C, ou Pascal, Java etc.
    - Checker compara diretamente saída do TMonitor com a saída do refmod.
    - Refmod fácil de escrever.
  - Basic, Matlab, etc.
    - Problema de comunicação com SystemC, pode usar IPC ou arquivos.
    - Refmod ainda mais fácil de escrever.

Implementação do Reference Model

- Refmod implementado em código (continuação)
  - possibilita teste pseudo-aleatório com número grande de amostras (muitos vetores de entrada),
  - deixar rodar a noite,
  - **dormir tranqüilo !**

Cobertura

- Ao acordar de manhã ainda está rodando...
- Quando vou parar ?
- Quando atingir a cobertura indicada no plano de verificação
- Tipos de cobertura:
  - de código
  - funcional

Cobertura de código

- Cobertura de linhas
  - mostra quantas vezes uma linhas de código (um comando) foi executada
  - também chamada cobertura de blocos ou cobertura de segmentos
- Cobertura de chaveamento
  - conta quantas vezes cada sinal mudou de nível lógico
- Cobertura de estados de FSM
  - conta quantas vezes se chegou em cada estado
  - quais transições entre estados foram simuladas

Cobertura de código

- Cobertura de eventos
  - verifica se todos os eventos na lista de sensibilidade de todos processos foram exercitadas
- Cobertura de desvios
  - resultado semelhante ao cobertura de blocos
- Cobertura de expressões
  - verifica se todas as combinações de expressões lógicas foram exercitadas
- Cobertura de caminhos
  - verifica passagem variada por seqüência de comandos if...else

Cobertura funcional

- É mais importante do que a cobertura de código.
- Observa sinais e transações durante a simulação.
- Conta ocorrências de determinadas situações, por exemplo: “quantas vezes o valor da saída fica entre 0 e 8”

Cobertura funcional

- Cobertura desejada é especificada no Plano de Verificação (“a tal situação deve ser verificada 100 vezes”)

- Quando a cobertura desejada é atingida a simulação é encerrada

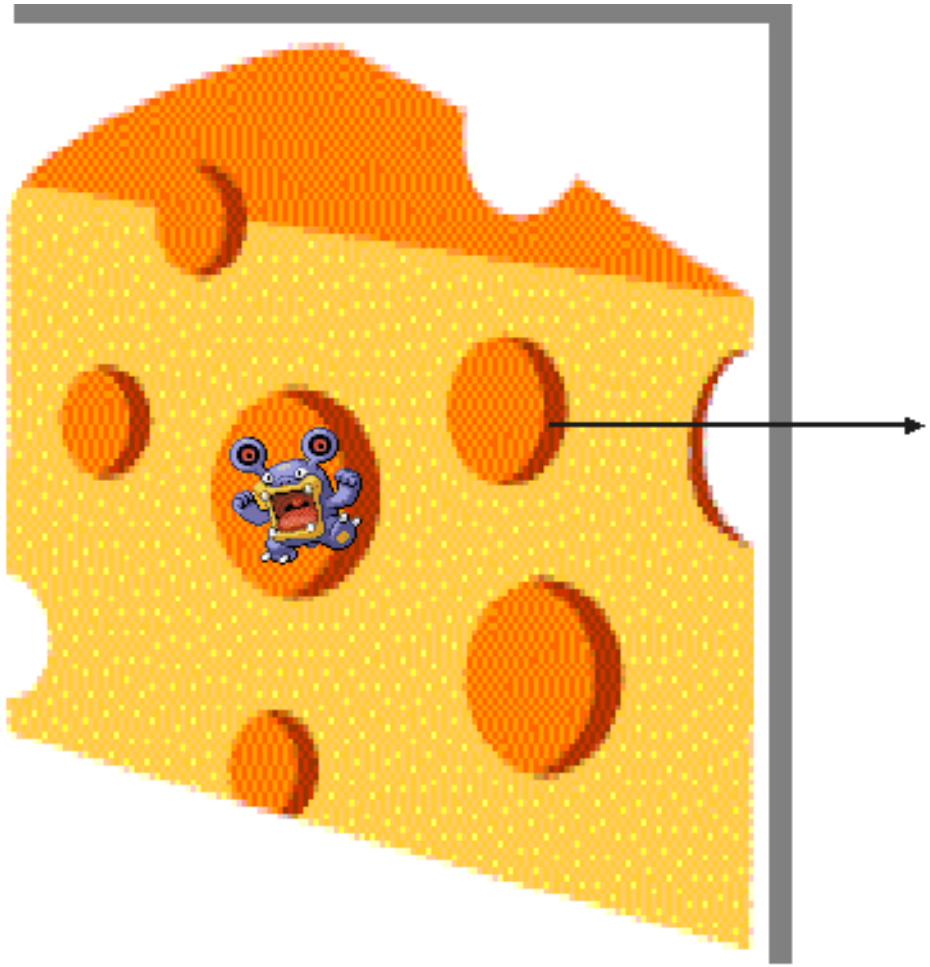
Cobertura funcional

“Cobertura é responsável por medir o progresso da verificação através de várias métricas pré-estabelecidas e ajudar o engenheiro a se localizar com relação ao término da verificação” [piziali2004].

Cobertura funcional

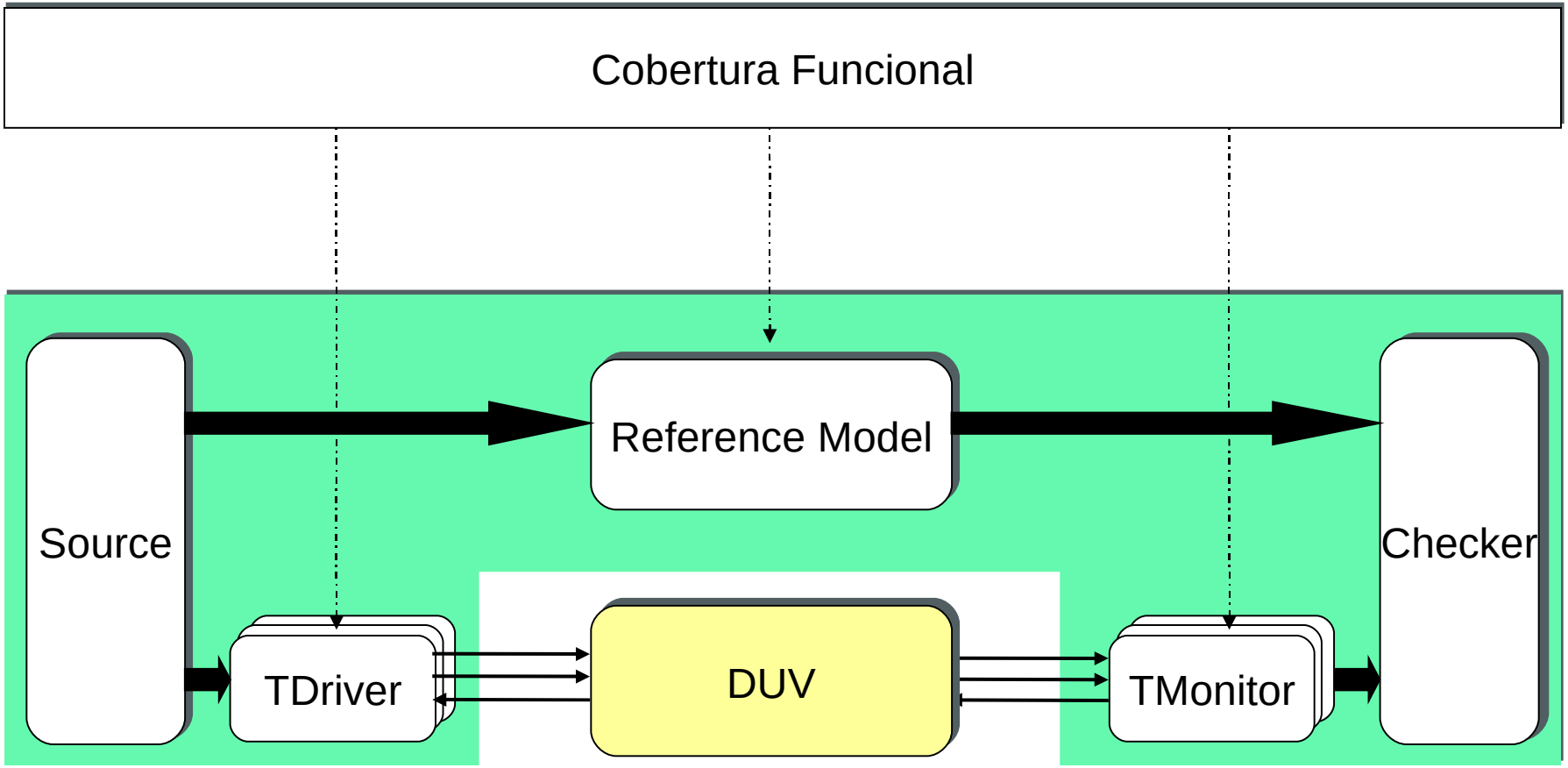
- Mede o progresso da simulação e reporta quais funcionalidades não foram exercitadas ou foram exercitadas mais de uma vez.
- Pode ajudar a inspecionar a qualidade da verificação e direcionar os estímulos de forma a alcançar as funcionalidades não cobertas (**buracos de cobertura**).

Cobertura funcional

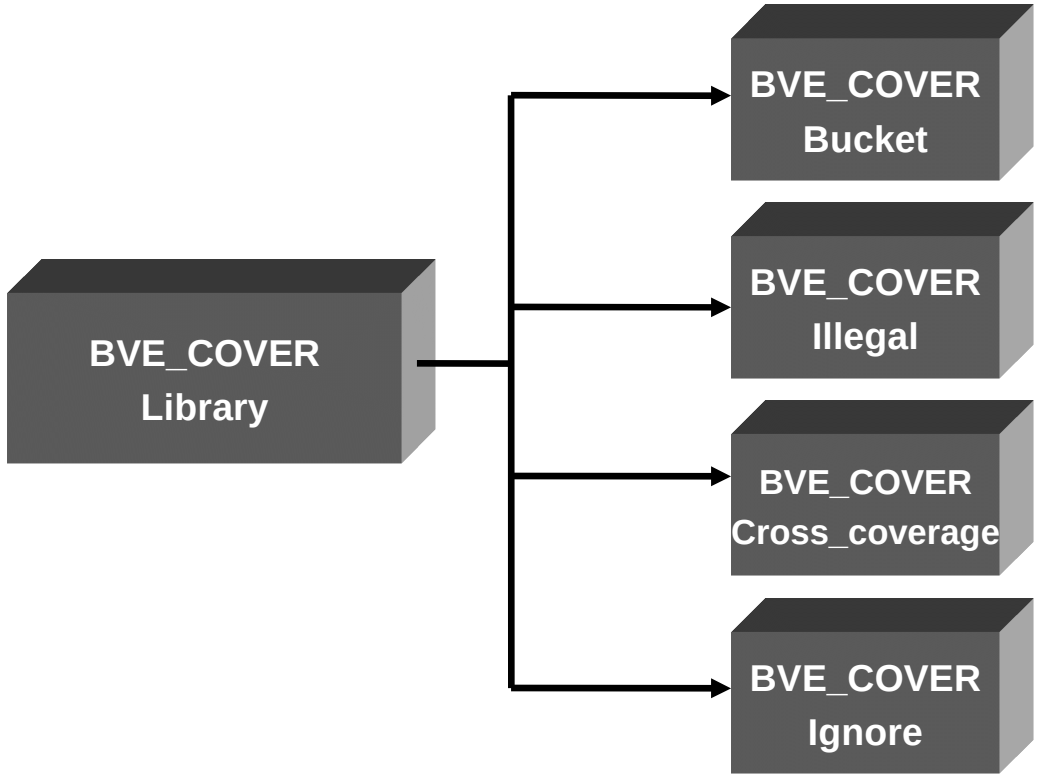


Buracos de Cobertura

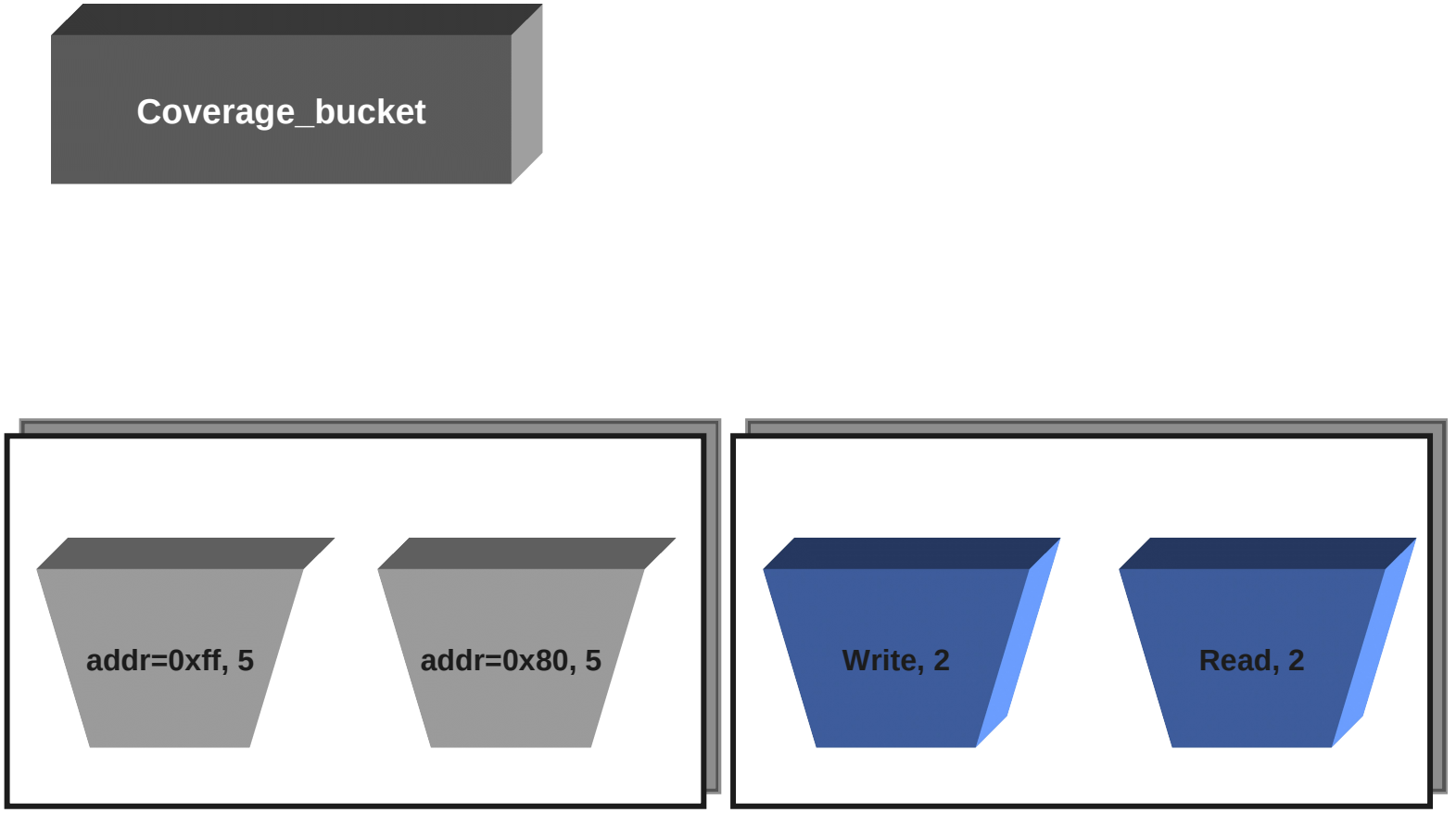
Onde Fazer Cobertura Funcional?



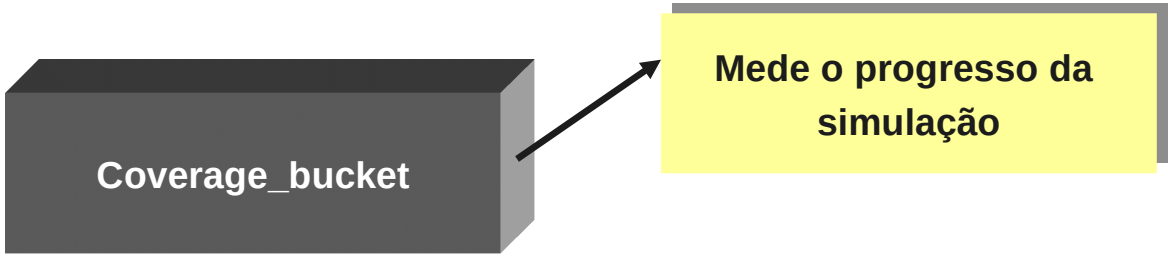
Biblioteca BVE_COVER

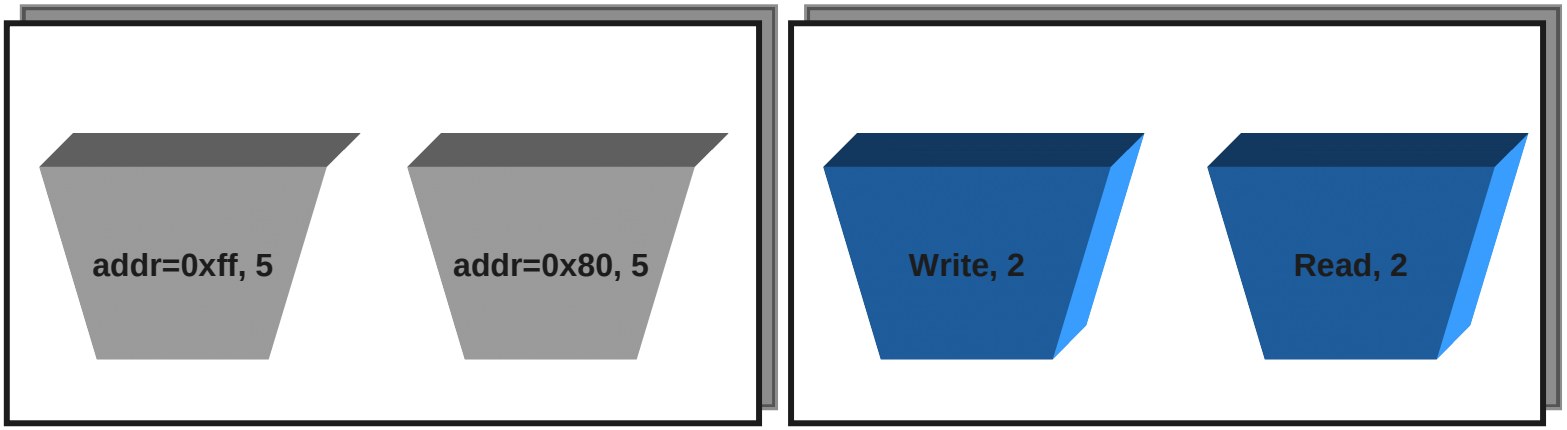


BVE_COVER Bucket

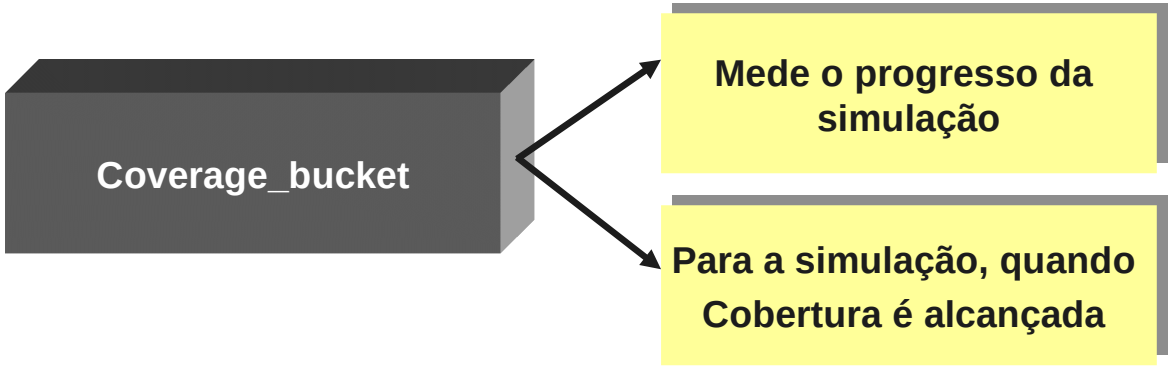


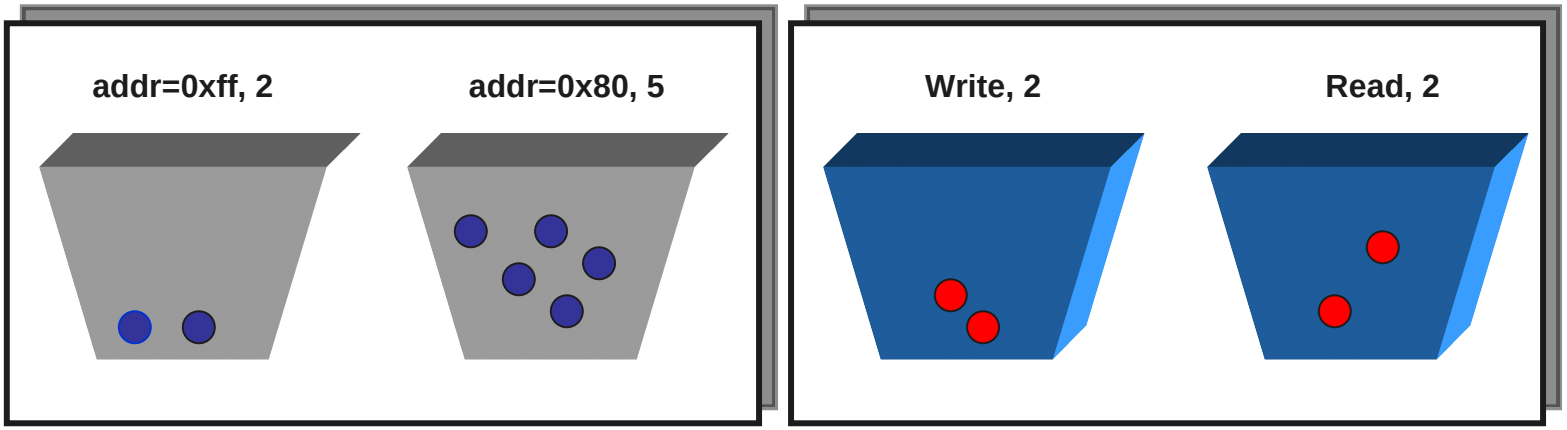
BVE_COVER Bucket



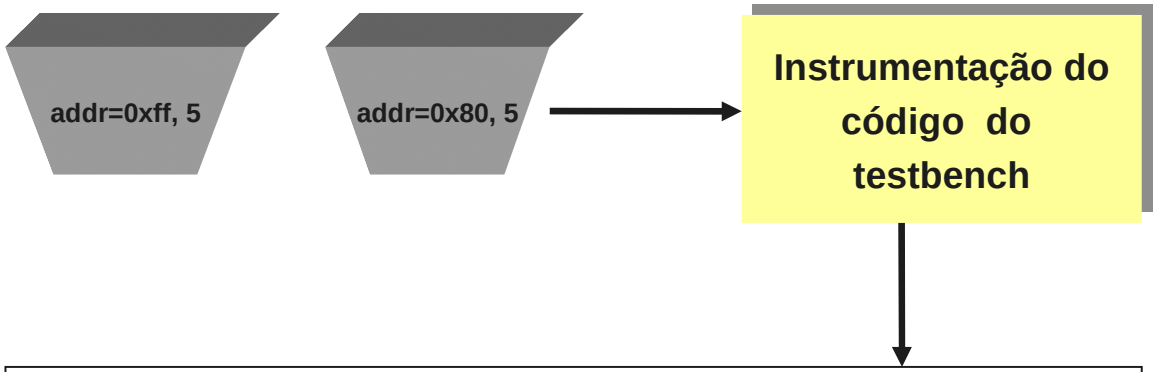


BVE_COVER Bucket



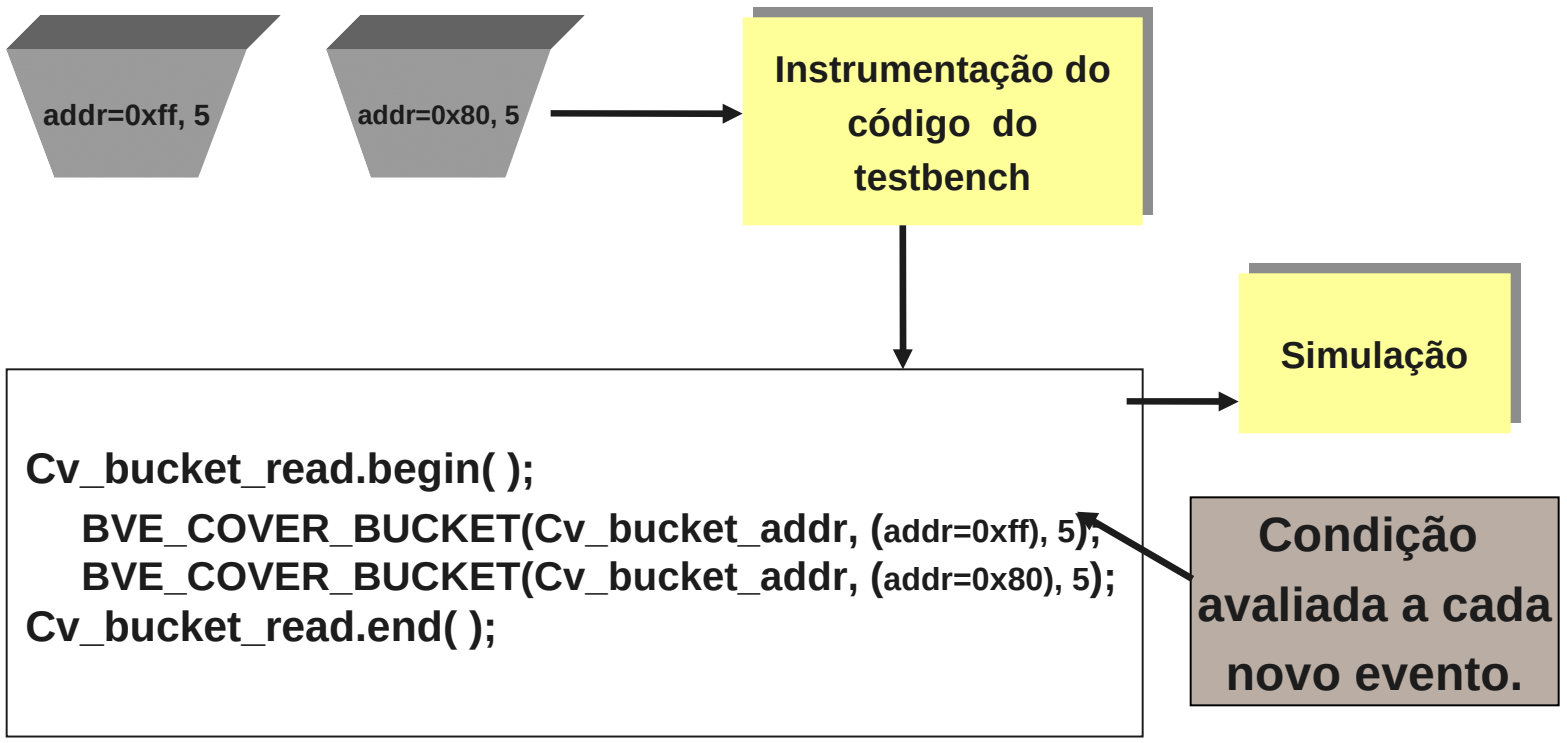


BVE_COVER Bucket

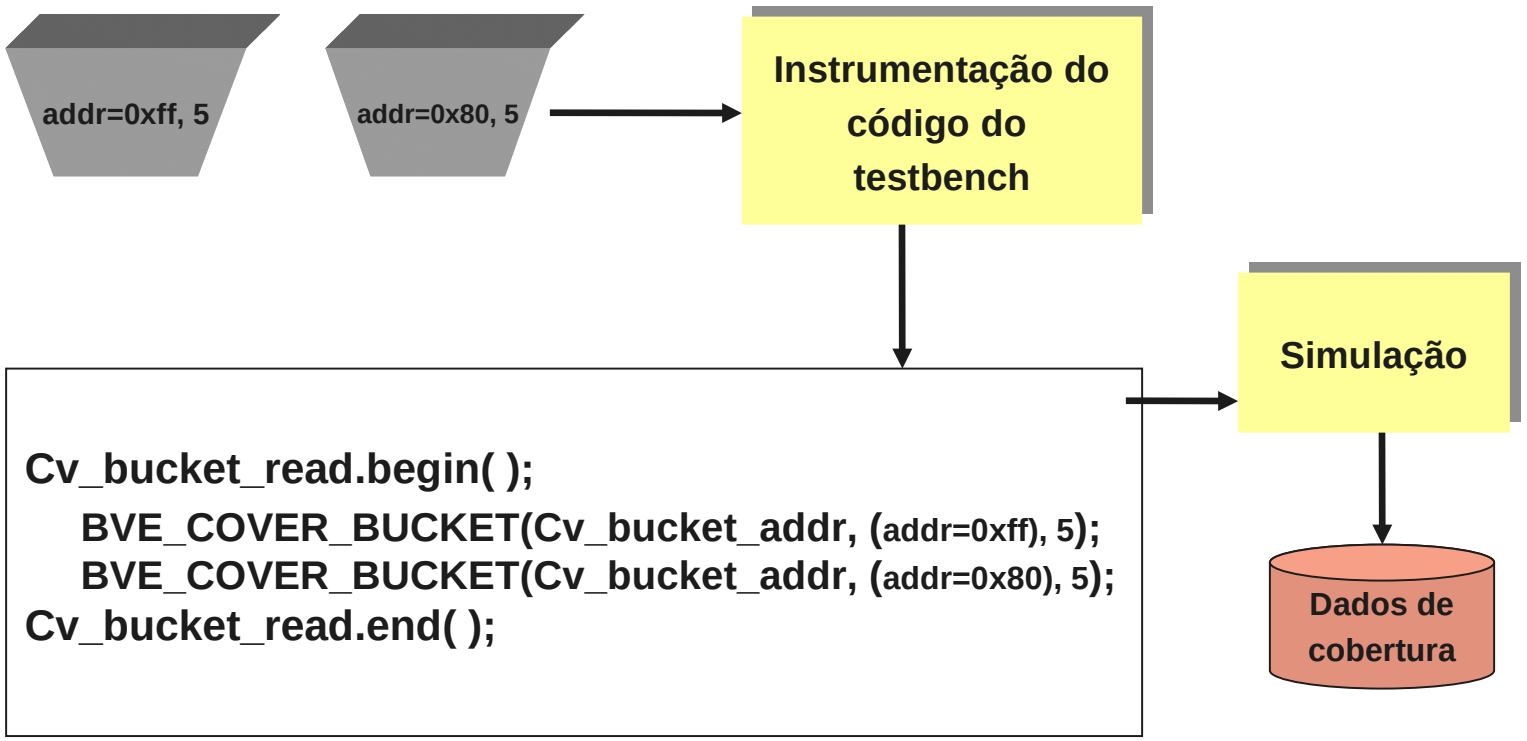



```
Cv_bucket_read.begin( );
    BVE_COVER_BUCKET(Cv_bucket_addr, (addr=0xff), 5);
    BVE_COVER_BUCKET(Cv_bucket_addr, (addr=0x80), 5);
Cv_bucket_read.end( );
```

BVE_COVER Bucket



BVE_COVER Bucket

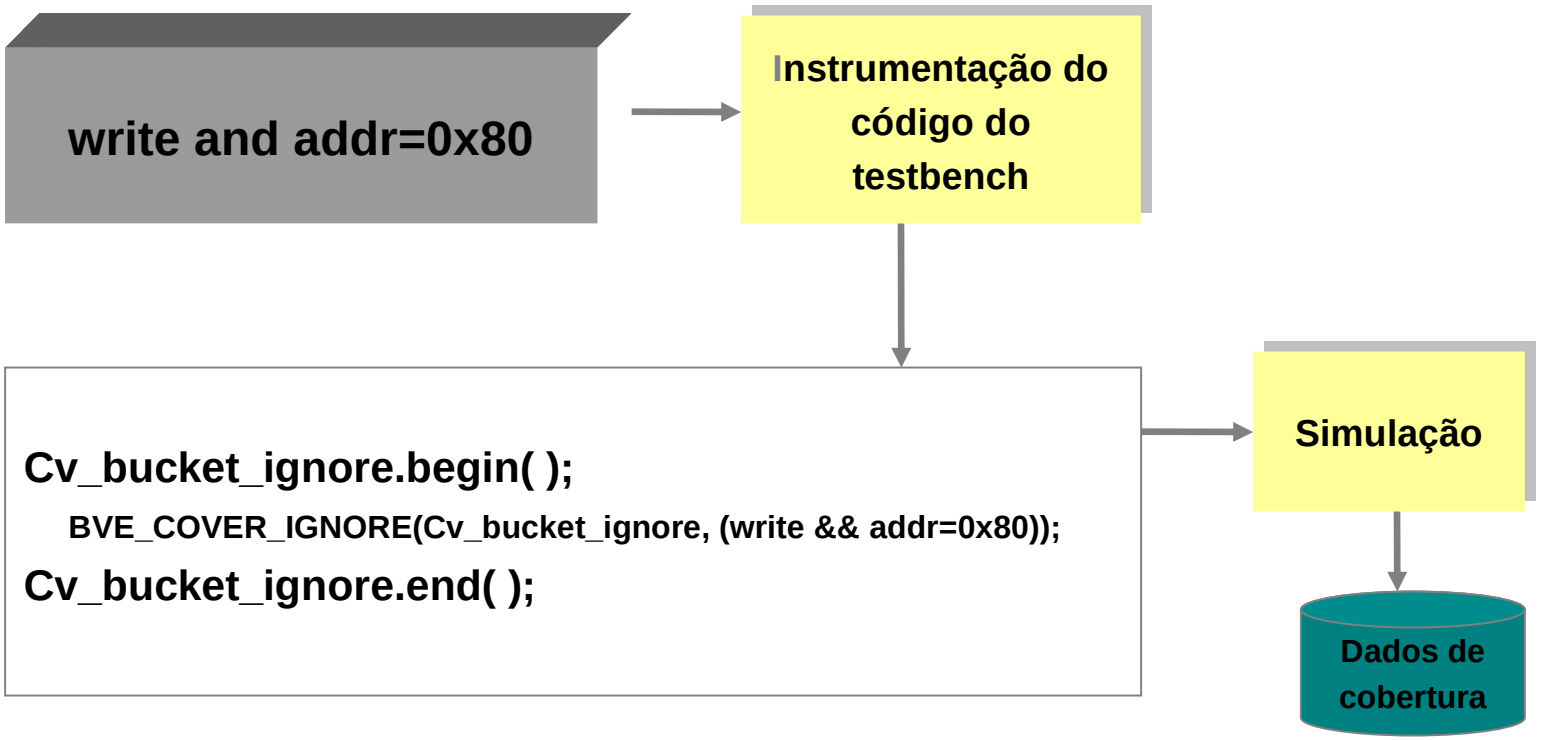


BVE-COVER Ignore

- Especifica as funcionalidades que podem ser ignoradas.

- Usado na identificação de buracos de cobertura.

BVE-COVER Ignore

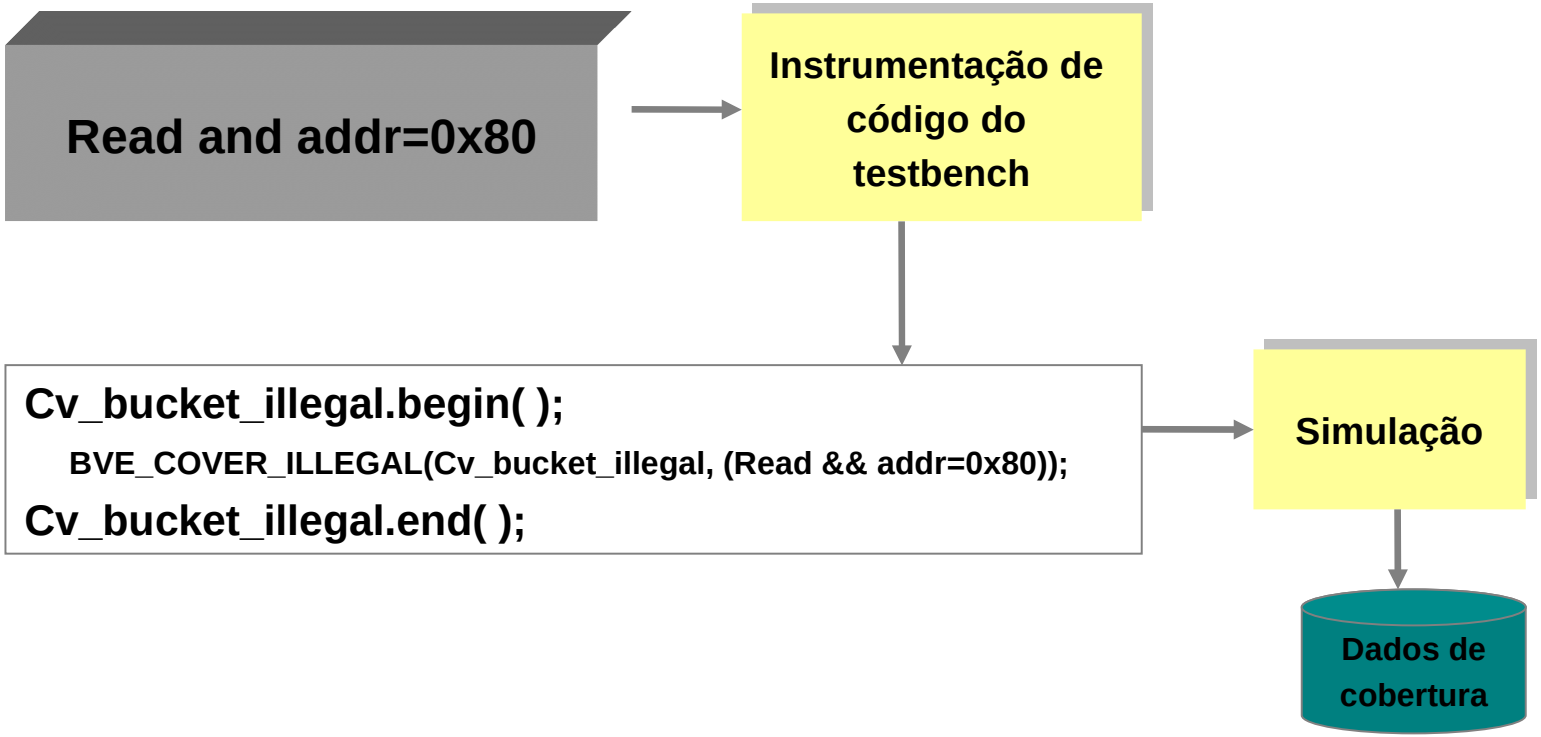


BVE-COVER Illegal

- Especifica as funcionalidades consideradas ilegais.

- Se funcionalidades ilegais são executadas a biblioteca mostra erros.

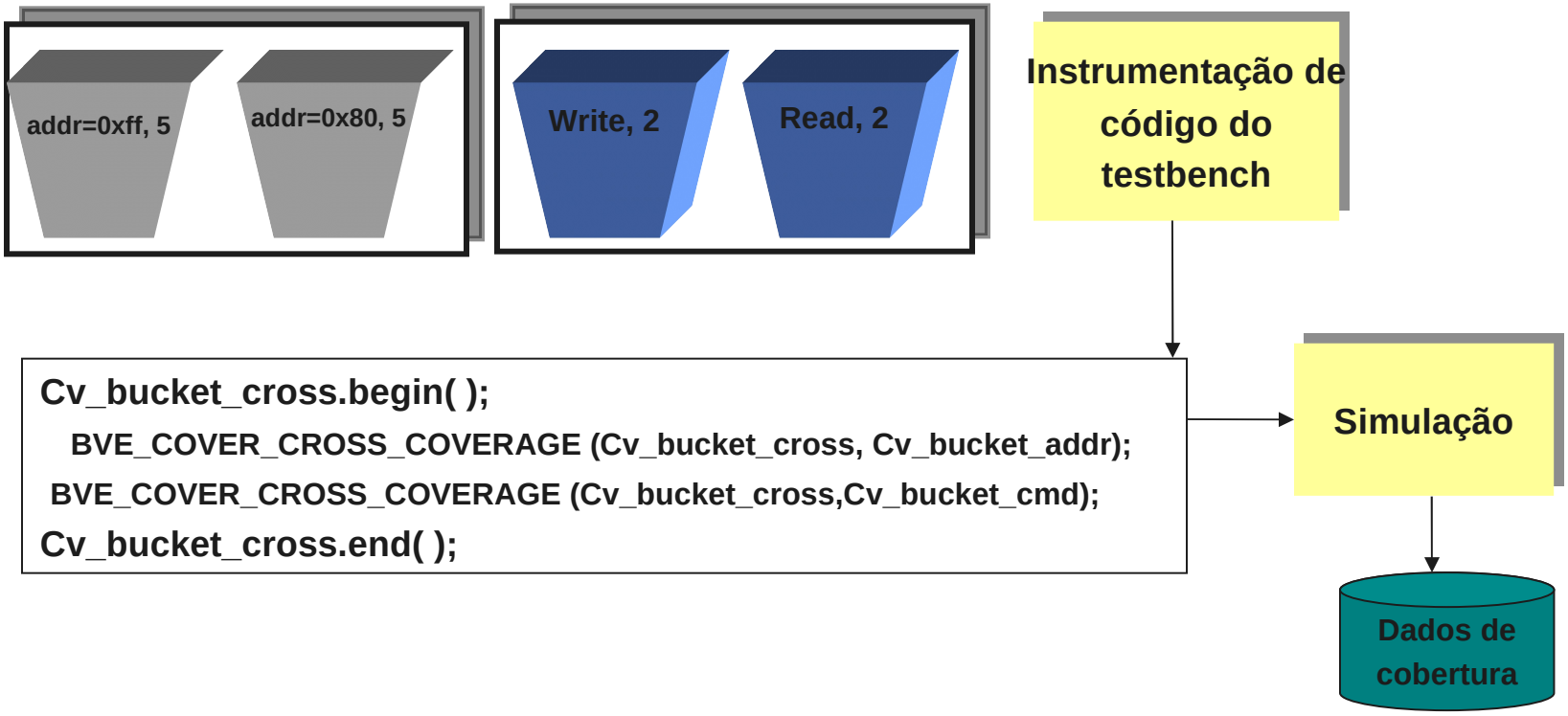
BVE-COVER Illegal



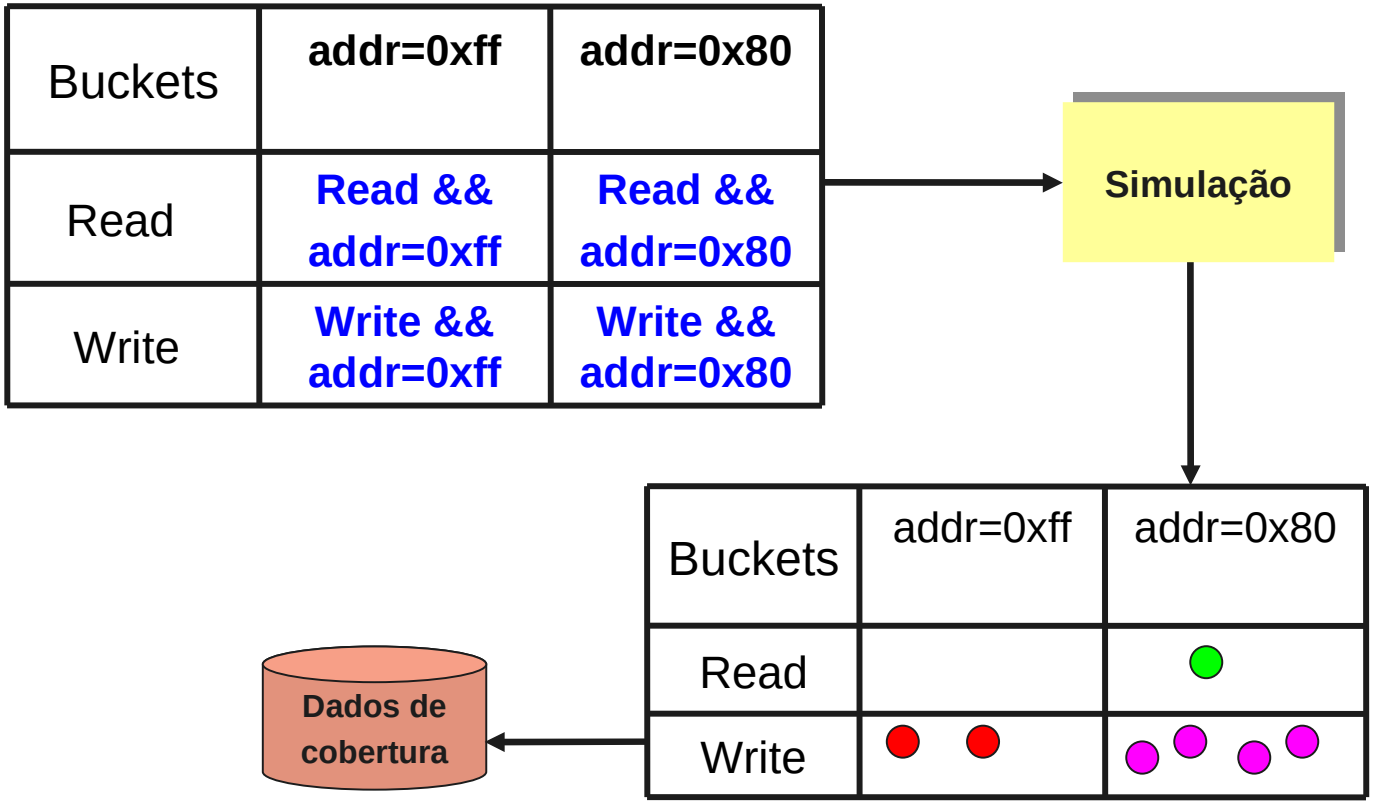
BVE-COVER Cross-coverage

- Cruza informações dos Buckets.
- Mostra se funcionalidades importantes são executadas simultaneamente.
- Importante para encontrar buracos de cobertura.

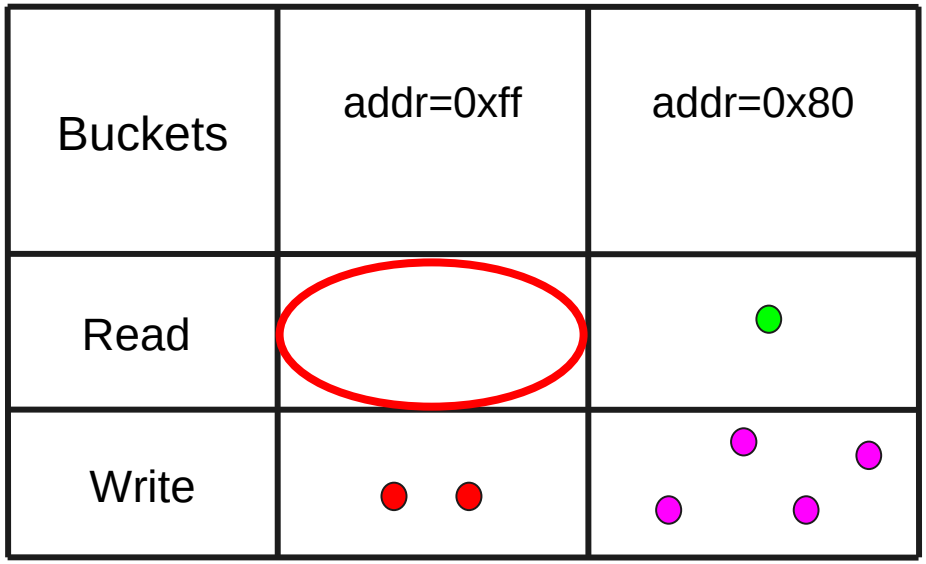
BVE-COVER Cross-coverage



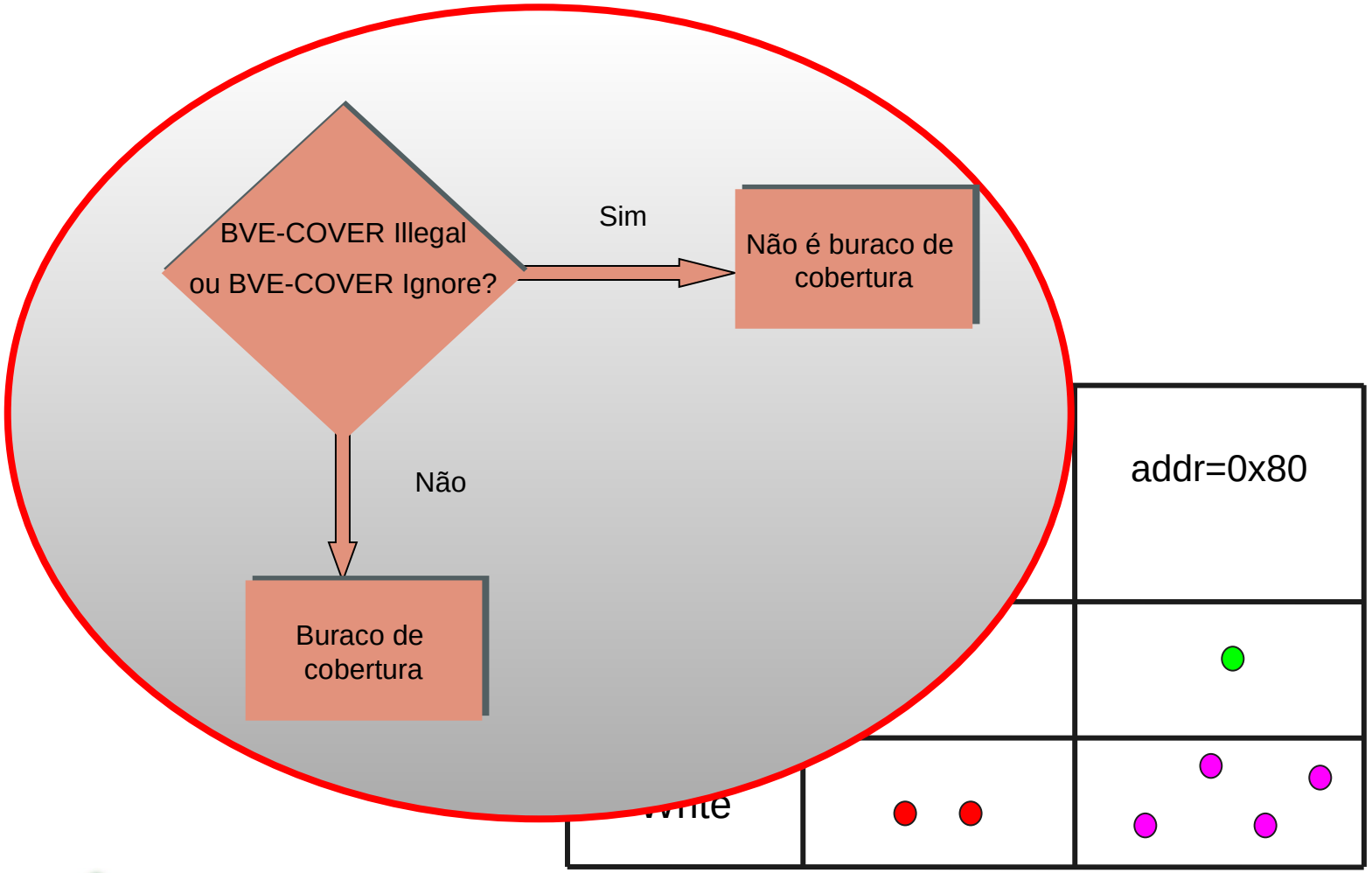
BVE-COVER Cross-coverage



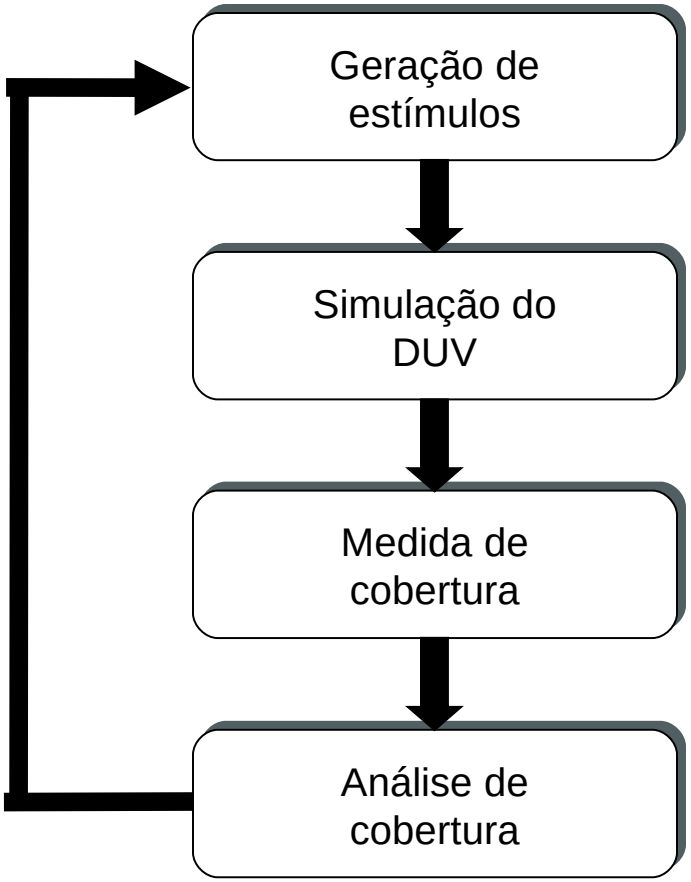
BVE-COVER Cross-coverage



BVE-COVER Cross-coverage

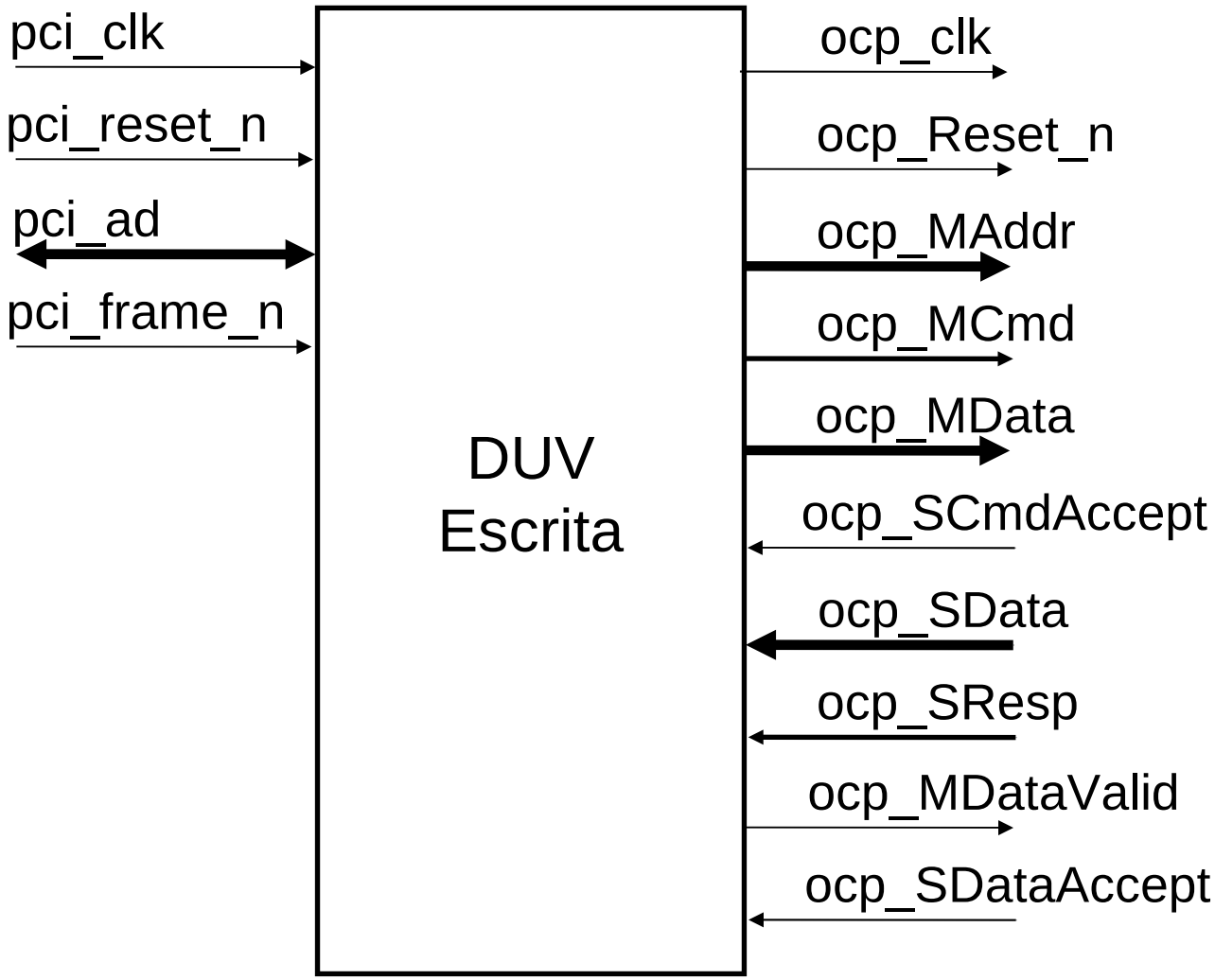


Cobertura Funcional

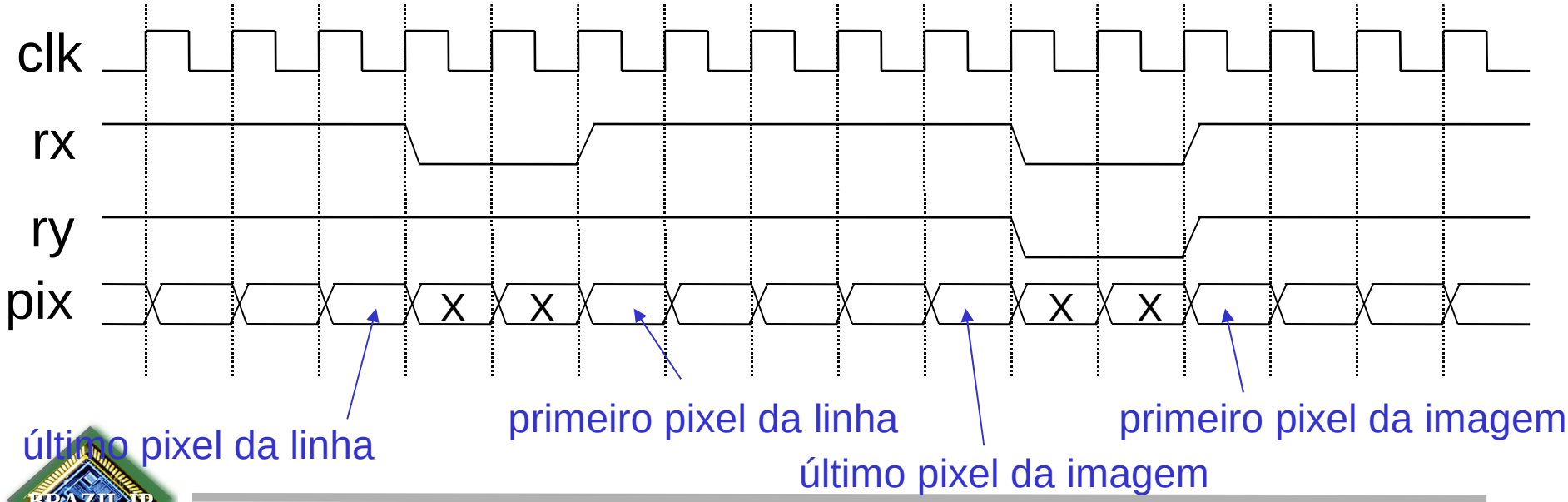


Exemplos

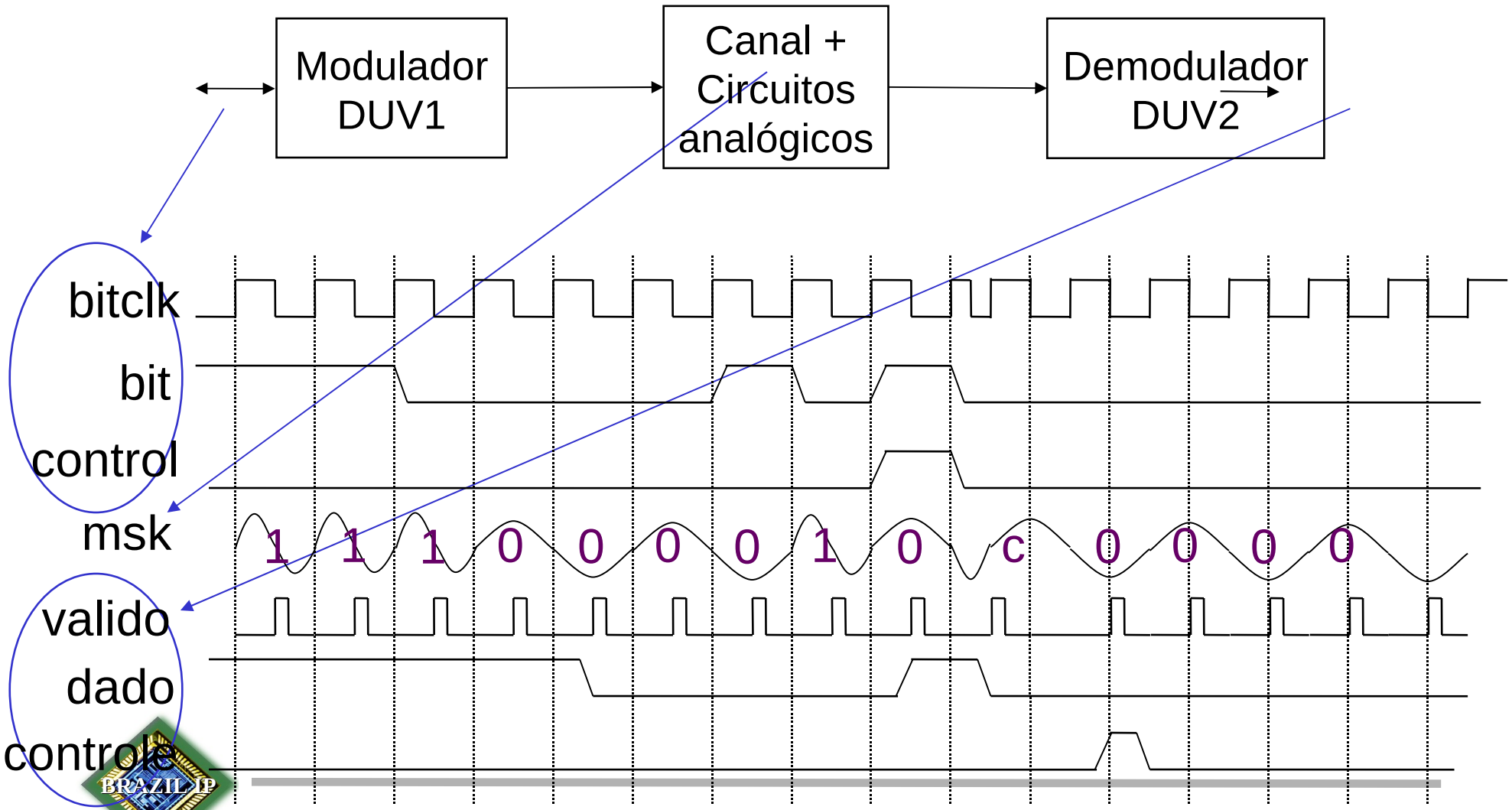
Exemplo PCI/OCP



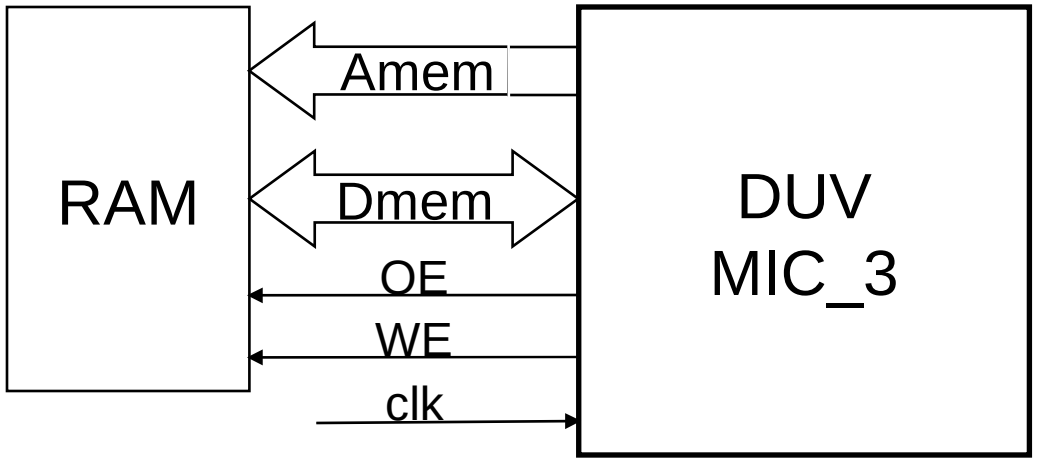
Exemplo Gradiente



Exemplo MSK

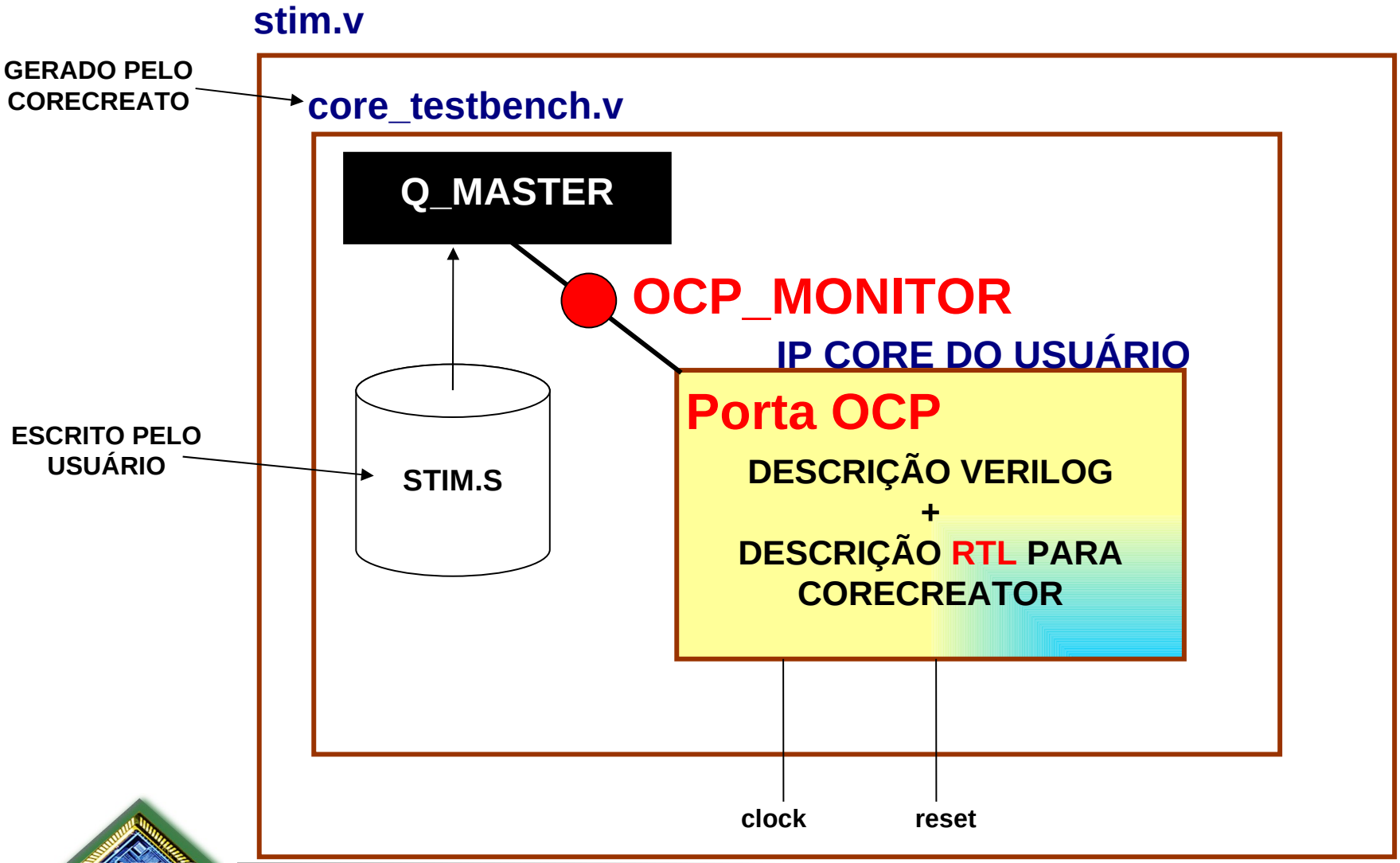

Modulador DUV1
Canal + Circuitos analógicos
Demodulador DUV2
bitclk
bit
control
msk
valido
dado
controle
1 1 1 0 0 0 0 1 0 c 0 0 0 0

Exemplo Java

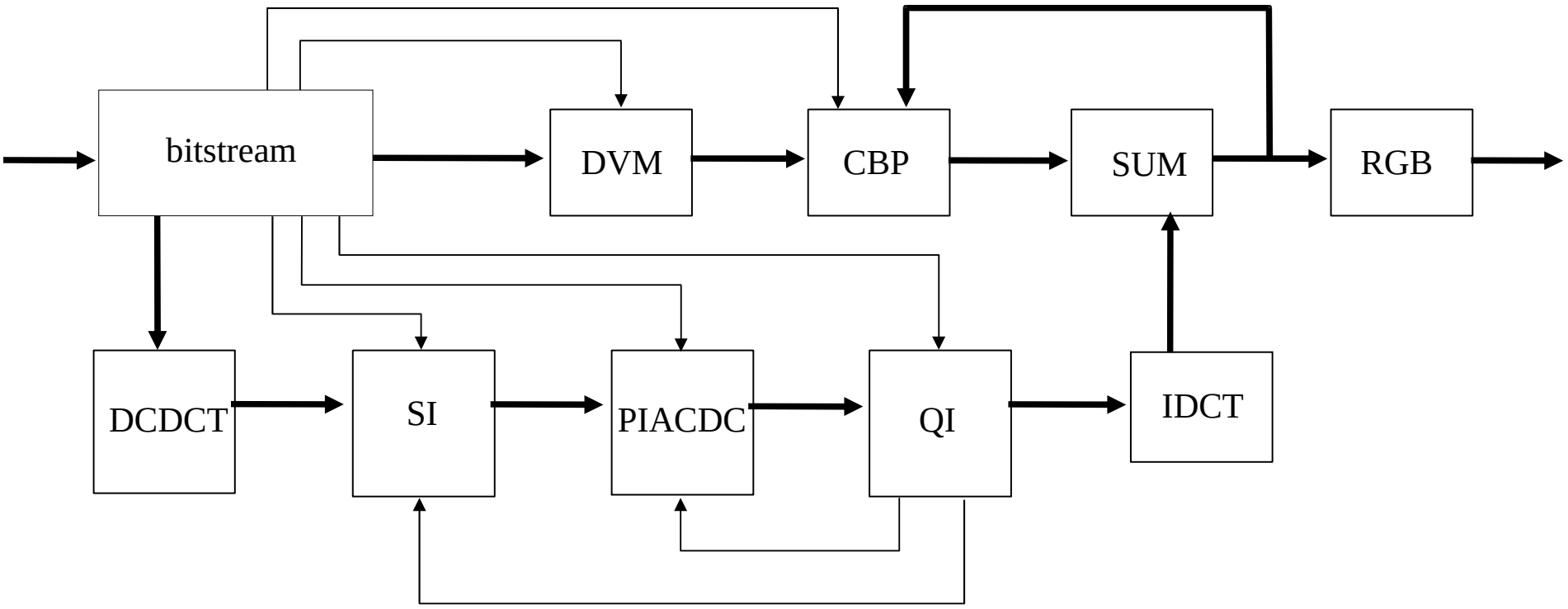


Byte-code	Mnemônico	Descrição
0x10	BIPUSH *byte*	Empilhar bytecode seguinte
0x60	IADD	Desempilhar duas palavras, somar, empilhar resultado
0x5F	SWAP	Trocar posição de duas palavras no topo da pilha
	. . .	

Exemplo CoreCreator



Exemplo MPEG4



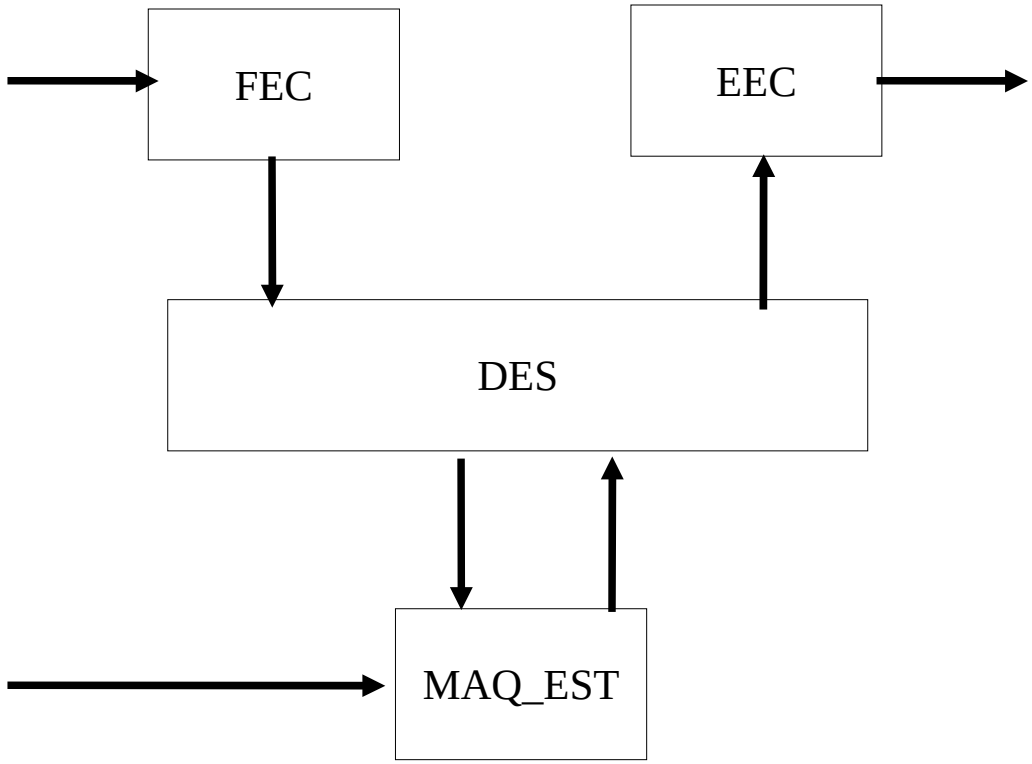
Conexão de dados

Conexão de configuração

Exemplo MPEG4

- Testbench depois do RTL, com ferramenta para templates.
- Pior problema: erramos em fazer um só testbench para PIACDC e QI
- Tempo entre Verificação dos sub-blocos o.k. e verificação do conjunto dos DUVs o.k.: 3 semanas
- Tempo entre Verificação o.k. e FPGA rodando: 3 dias

Exemplo Lacre Eletrônico



Exemplo Lacre Eletrônico

- Testbench antes do RTL, sem ferramenta para templates
- Pior problema: Verificação da junção dos sub-blocos feita depois já ter iniciado verificação de 2 blocos.
- Tempo entre Verificação dos sub-blocos o.k. e verificação do conjunto dos DUVs o.k.: 1 dia
- Tempo entre Verificação o.k. e FPGA rodando: 0