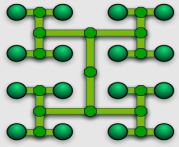
Universidade de Brasília

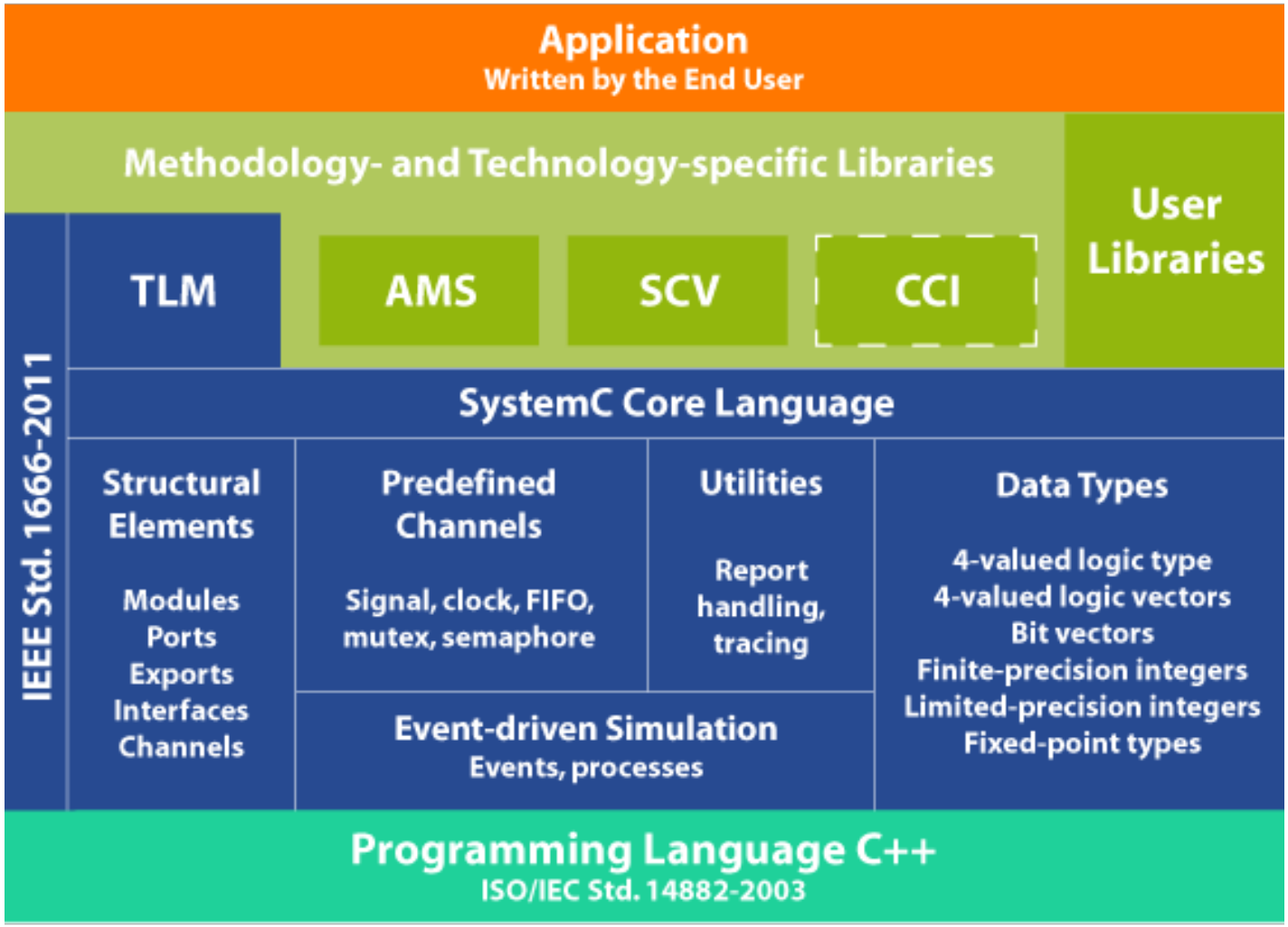
Fundamentos de SystemC

Ricardo Jacobi
Departamento de Ciência da Computação
Universidade de Brasília
jacobi@unb.br

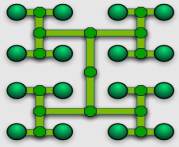


LAICO

Arquitetura da Linguagem



Universidade de Brasília



LAICO

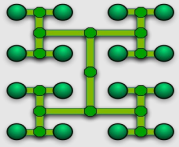
Histórico SystemC

- SystemC 0.9 (Set. 1999)
  - Introduz características de HDLs: concorrência, tempo, sinais...
  - Kernel de simulação
  - Aritmética de ponto fixo
  - Módulos
    - Hierarquia estrutural
  - Restringe-se a descrições RTL
- SystemC 1.0 (Abril 2000)
  - Primeira versão de uso geral

Histórico de SystemC

- SystemC 2.0 (Fev 2002)
  - Modelagem em nível de sistema
    - Maior suporte a modelagem em nível de transações, separação entre processamento e comunicação
  - Comunicação: canais, interfaces e portas
  - Eventos como elementos de controle de simulação
- TLM 1.0 (Abril 2005)
  - API para modelagem transacional
  - Conexões ponto a ponto
  - Baseado em troca de mensagens
  - Comunicação bloqueante e não-bloqueante

Universidade de Brasília

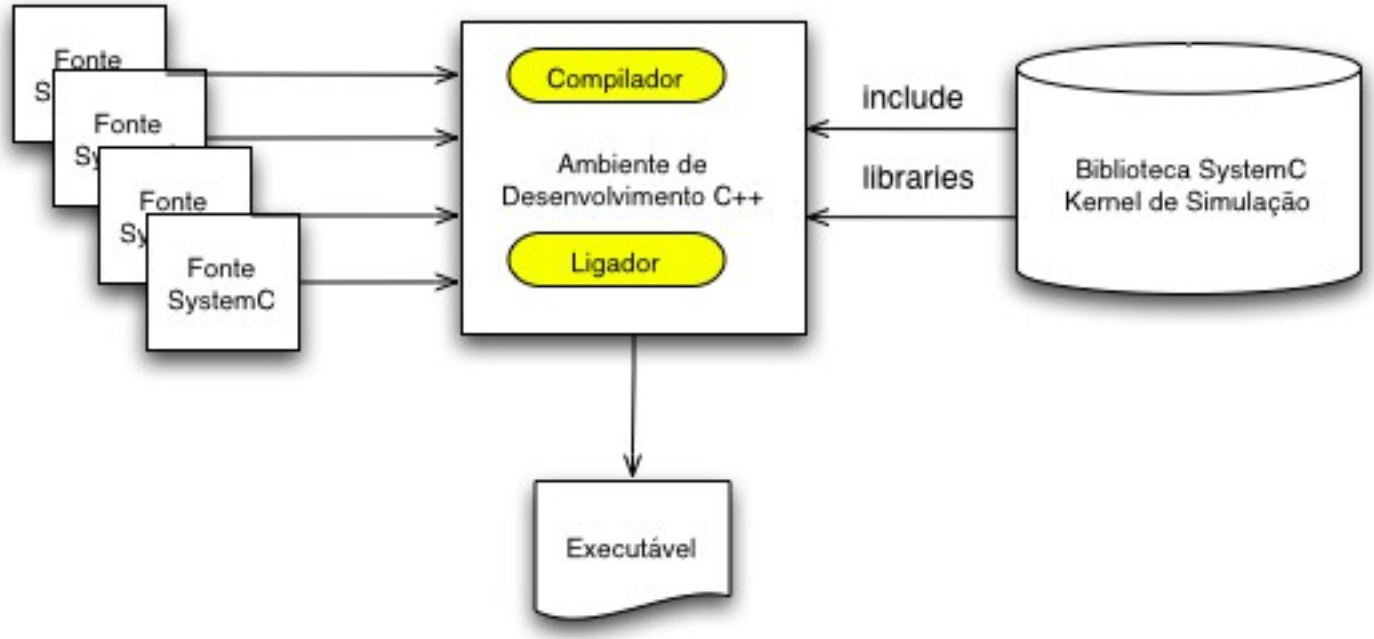


LAICO

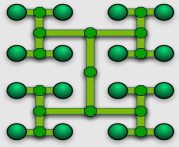
Histórico de SystemC...

- SystemC 2.1 (Set. 2005)
  - melhor suporte a modelagem transacional
  - melhor modularidade
  - bugs
- TLM 2.0 – (Jun. 2008)
  -
  - Baseado em troca de mensagens
- SystemC 2.3 ()
  -

Compilando SystemC



Universidade de Brasília

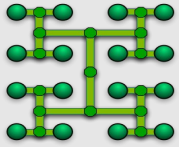


LAICO

Modelagem em SystemC

- Modelagem da temporização
- Tipos de dados em hardware
- Hierarquia e estrutura de módulos
- Gerenciamento de comunicação entre módulos concorrentes
- Modelagem da concorrência

Universidade de Brasília



LAICO

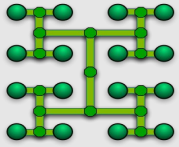
Temporização

- A resolução da medida de tempo é de 64 bits
- A classe **sc_time** é utilizada para representar o tempo na simulação, intervalos de tempo e atrasos
- Existem métodos que fornecem o tempo atual, calculam intervalos de tempo e atrasos
- O tempo é medido em unidades de tempo **sc_time_unit** (inteiro)

Temporização

- As unidades de tempo pré-definidas são:
  - SC_FS (femto segundo)
  - SC_PS (pico segundo)
  - SC_NS (nano segundo)
  - SC_US (micro segundo)
  - SC_MS (mili segundo)
  - SC_SEC (segundos)
- SC_ZERO_TIME é uma constante que representa o interval nulo de tempo
  - usado sempre que se deseja indicar tempo = 0
  - Ex: wait(SC_ZERO_TIME);

Universidade de Brasília



LAICO

Utilizando sc_time

- Criação de variáveis do tipo sc_time:

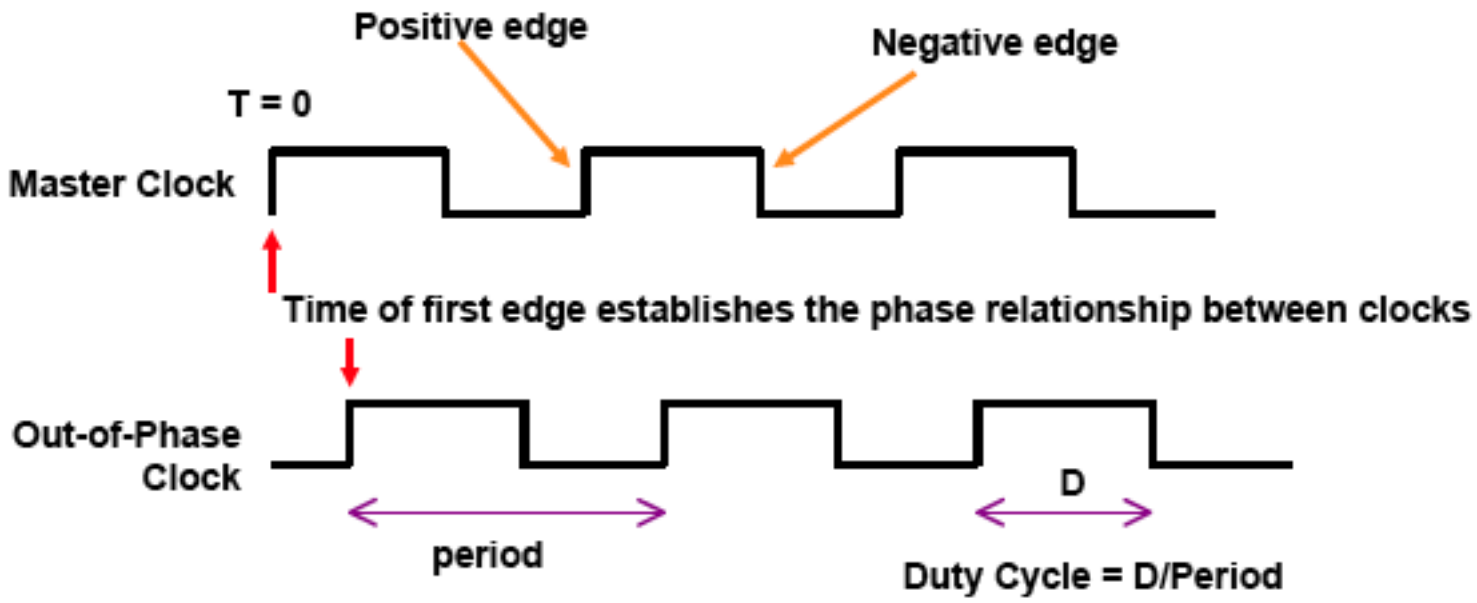
```
sc_time tempo;
sc_time t_Limite (100, SC_NS);
sc_time t_Periodo (5, SC_NS);
```

- Operações:

```
t_Intervalo = t_Atual – t_Evento;
if (t_Evento > t_Limite) ...
```

Clock

- SystemC suporta explicitamente um sinal de clock:
  - **Período**: duração do ciclo
  - **Duty cycle**: fração do período em que o relógio está em nível alto
  - **Borda positiva**: quando o valor do relógio transiciona de zero para um
  - **Borda negativa**: quando o valor do relógio transiciona de um para zero

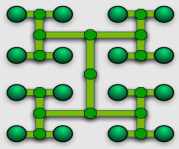


sc_clock

- **sc_clock** clock_name (“name”, **period**, **duty_cycle**, **start_time**, **positive_first**);
  - Cria um objeto do tipo relógio

Parâmetro	Descrição	Tipo	Default
name:	nome	char *	nenhum
period:	duração do ciclo	double	1
duty_cycle:	fração nível alto	double	0,5
start_time:	primeira borda	double	0
positive_first:	primeiro positiva	bool	true

Universidade de Brasília



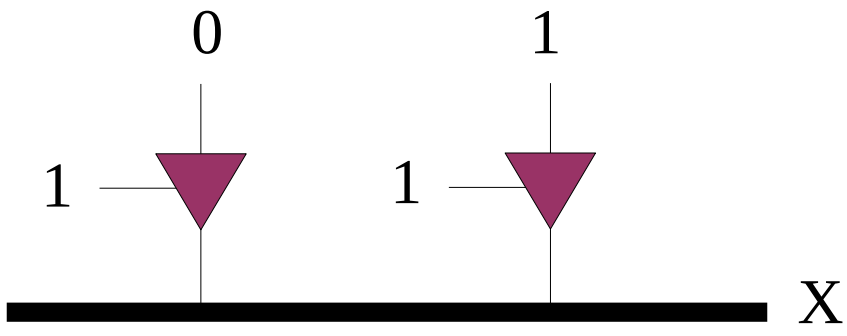
LAICO

Tipos de Dados

- SystemC oferece uma variedade de tipos de dados orientados ao hardware, com largura de palavra parametrizável:
  - Booleanos: **sc_bit e sc_bv<*n*>**
    - Podem valer 1 (verdadeiro) ou 0 (falso)
  - Multivalorados: **sc_logic e sc_lv<*n*>**
    - 4 valores: 1, 0, X, Z
  - Inteiros: **sc_int<*n*> e sc_uint<*n*>**
    - Inteiros de *n* bits com e sem sinal
  - Inteiros grandes: **sc_bigint<n> e sc_biguint<n>**
    - Inteiros de *n* bits, com n > 64, com e sem sinal

Lógica Multivalorada

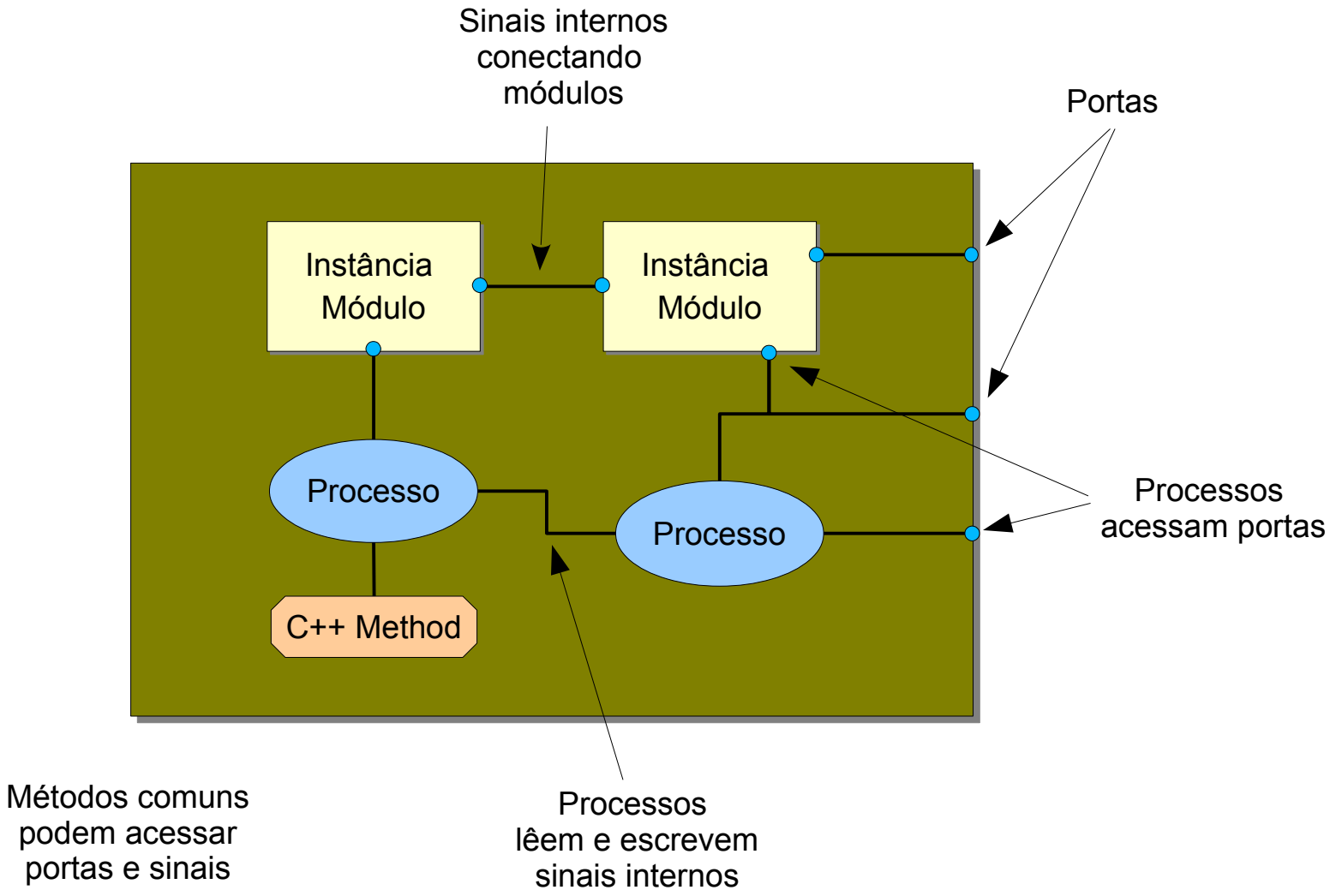
- Suporta valores não binários derivados do hw
- sc_logic é multivalorado. Valores aceitos:
  - 1 : valor lógico verdadeiro
  - 0 : valor lógico falso
  - Z : alta impedância (ou *tri-state*), indica um sinal que não está sendo acionado nem para 1 nem para 0
  - X : valor indeterminado. Utilizado quando o simulador não tem meio de calcular o valor associado ao sinal



Números Reais em Ponto Fixo

- SystemC provê uma *library* de ponto fixo:
  - sc_fix< >
  - sc_ufix< >
    - Ponto fixo com e sem sinal, parâmetros podem ser variáveis
  - sc_fixed<>
  - sc_ufixed<>
    - -ed: parâmetros constantes em tempo de compilação
  - sc_fix_fast var_name<>
  - sc_ufix_fast var_name<>
    - _fast: implementados com double, mais rápidos

Hierarquia


Sinais internos conectando módulos
Portas
Instância Módulo
Instância Módulo
Processo
Processo
C++ Method
Processos acessam portas
Métodos comuns podem acessar portas e sinais
Processos lêem e escrevem sinais internos

SC_MODULE

- SC_MODULE é uma classe que descreve componentes de hardware e serve de base para a construção de hierarquias
- Pode conter:
  - Sinais e variáveis
  - Processos
    - SC_METHOD: executa sem interrupção
    - SC_THREAD: pode ser interrompida no meio da execução
    - SC_CTHREAD: acionada pelo relógio
  - Outros módulos
  - Métodos C++

Processos

- Processos são métodos C++ registrados no kernel de simulação para serem gerenciados pelo simulador
  - Modelam componentes do hardware que executam de forma concorrente
- SC_METHOD:
  - Execução ocorre sem avanço no tempo de simulação
  - Invocado pelo simulador em função da ocorrência de eventos

Threads

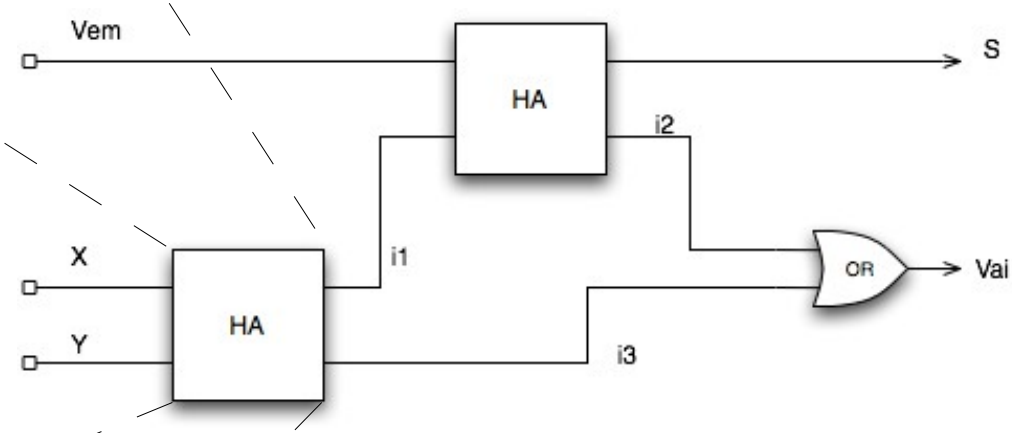
- SC_THREAD:
  - Invocado apenas uma vez pelo simulador
  - Ao encerrar execução, não é chamado de novo durante a simulação
  - Pode ser interrompida e suspensa a sua execução, ficando a espera de um evento de sincronização
  - Usualmente executa com um laço infinito, até o final da simulação
- SC_CTHREAD:
  - Thread sensível a eventos de relógio. A interrupção e retomada de execução dependem diretamente de eventos gerados por um relógio

Threads

- SC_THREAD:
  - Invocado apenas uma vez pelo simulador
  - Ao encerrar execução, não é chamado de novo durante a simulação
  - Pode ser interrompida e suspensa a sua execução, ficando a espera de um evento de sincronização
  - Usualmente executa com um laço infinito, até o final da simulação
- SC_CTHREAD:
  - Thread sensível a eventos de relógio. A interrupção e retomada de execução dependem diretamente de eventos gerados por um relógio

Exemplo: meio-somador

```
SC_MODULE (ha_sc) {
sc_in<bool> x, y;
sc_out<bool> s, v;

SC_CTOR(ha_sc) {
  SC_METHOD(proc);
  sensitive << x << y;
}
void proc(void) {
  s = x ^ y;
  v = x & y;
 }
};
```



Declarando módulos

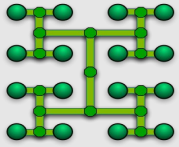
```
SC_MODULE (ha_sc) {
sc_in<bool> x, y;
sc_out<bool> s, v;

SC_CTOR(ha_sc) {
  SC_METHOD(proc);
  sensitive << x << y;
}
void proc(void) {
  s = x ^ y;
  v = x & y;
 }
};
```

Macro que declara o módulo

Nome do módulo

Nome do construtor do módulo: área que contém o código de iniciação dos componentes instanciados a partir do módulo

Universidade de Brasília



LAICO

Declarando módulos

```
SC_MODULE (ha_sc) {
  sc_in<bool> x, y;
  sc_out<bool> s, v;

  SC_CTOR(ha_sc) {
    SC_METHOD(proc);
    sensitive << x << y;
  }
  void proc(void) {
    s = x ^ y;
    v = x & y;
  }
};
```

Porta de entrada do módulo tipo de dado booleano

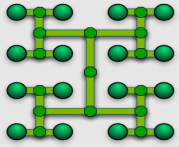
Portas de saída do módulo tipo de dado booleano

Declaração de processo para o kernel de simulação Nome do processo é **proc**

Declaração da lista de sinais aos quais o processo é sensível, ou seja, cuja alteração dispara a sua execução

Corpo do processo

Universidade de Brasília

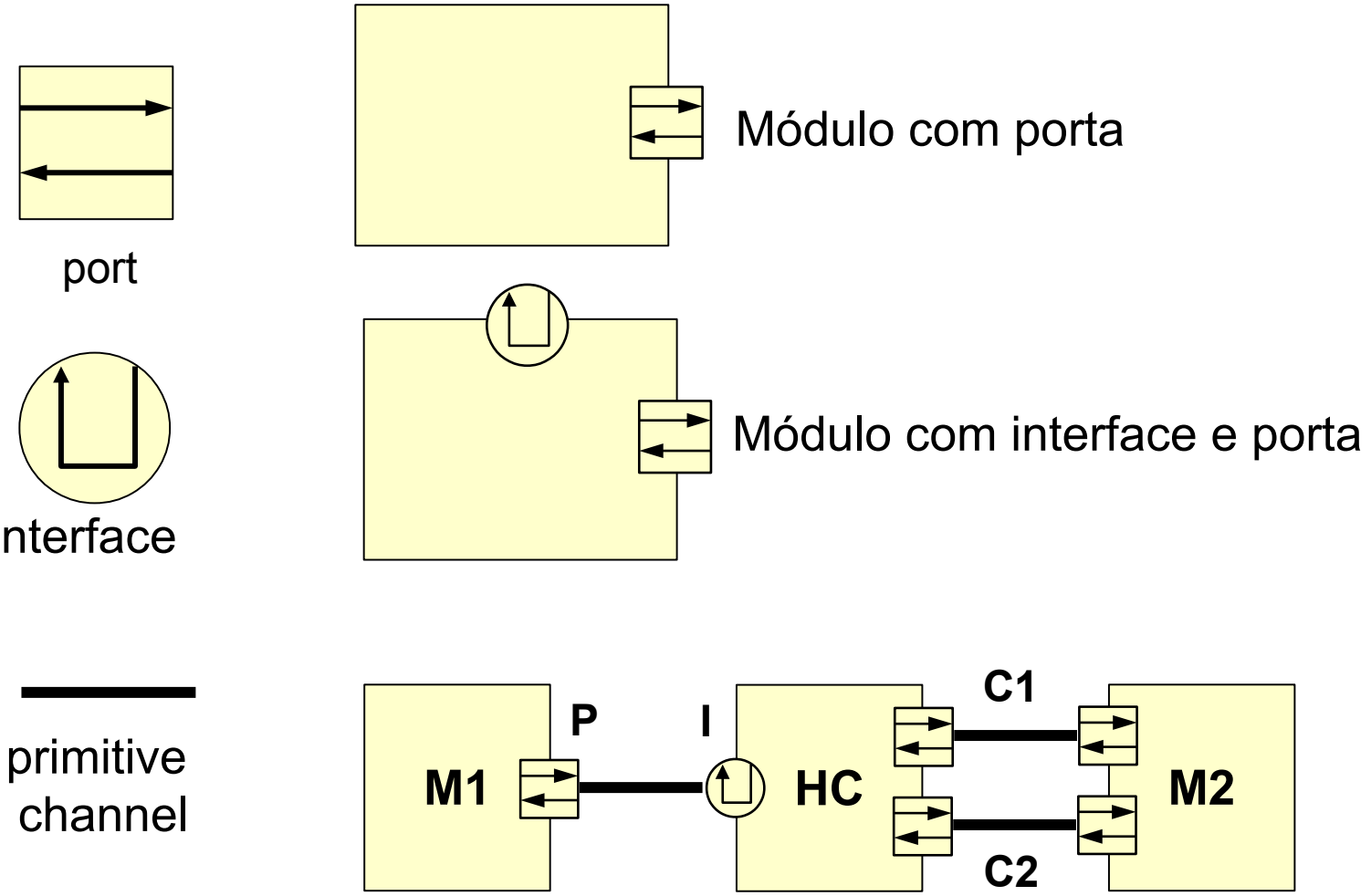


LAICO

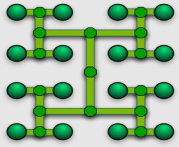
Interfaces, Portas e Canais

- *Ports* definem as entradas e saídas de módulos
- *Channels* são os meios de comunicação (objetos de SystemC) que interconectam *ports*.
- *Interfaces* definem os métodos que os canais devem implementar para viabilizar a comunicação entre módulos
  - SystemC adota o conceito de interface de Java: uma interface define apenas as declarações dos métodos, sem definir a sua implementação
  - Os canais *implementam interfaces*

Representação Gráfica



Universidade de Brasília

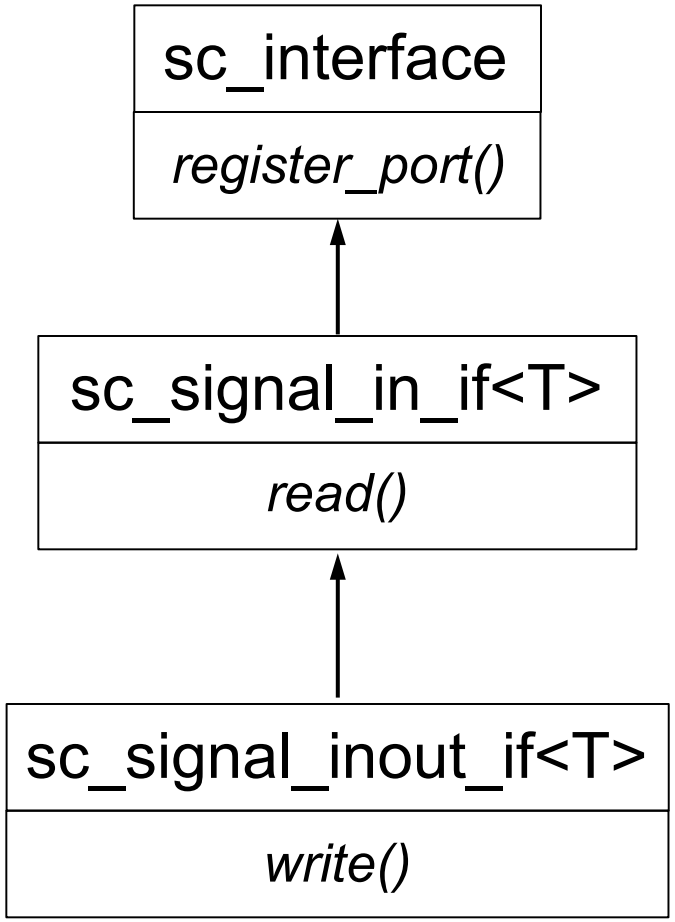


LAICO

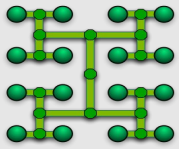
Interfaces

- Interfaces especificam quais métodos (operações) um canal deve implementar
  - Mais especificamente, a assinatura dos métodos: nome do método, parâmetros e valor retornado
  - Não especifica *como* as operações são implementadas nem define campos de dados
- Em SystemC, todas as interfaces são derivadas, direta ou indiretamente, da classe abstrata **sc_interface**

Hierarquia de Interfaces



Universidade de Brasília



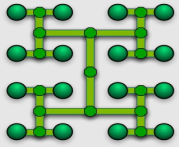
LAICO

Exemplo de Interface

```
// ----------------------------------------
// An example read interface: sc_read_if
// this interface provides a 'read' method
// ----------------------------------------
template <class T>
class sc_read_if
: virtual public sc_interface
{
 public:
 // interface methods
 virtual const T& read() const = 0;
};
```

Interface que só lê

Universidade de Brasília



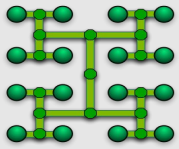
LAICO

Exemplo de Interface...

```
// ----------------------------------------
// An example write interface: sc_write_if
// this interface provides a 'write' method
// ----------------------------------------
template <class T>
class sc_write_if
: virtual public sc_interface
{
  public:
    // interface methods
    virtual void write( const T& ) = 0;
};
```

Interface que só escreve

Universidade de Brasília



LAICO

Portas

- Uma porta é um objeto através do qual um módulo pode acessar a interface de um canal
- Portas básicas:
  - sc_in<T> : porta de entrada, módulo lê dados externos
  - sc_out<T> : porta de saída, módulo escreve dados
  - sc_inout<T> : porta de entrada e saída
- sc_port_base é a classe abstrata raiz de todas as portas
- outras podem ser criadas

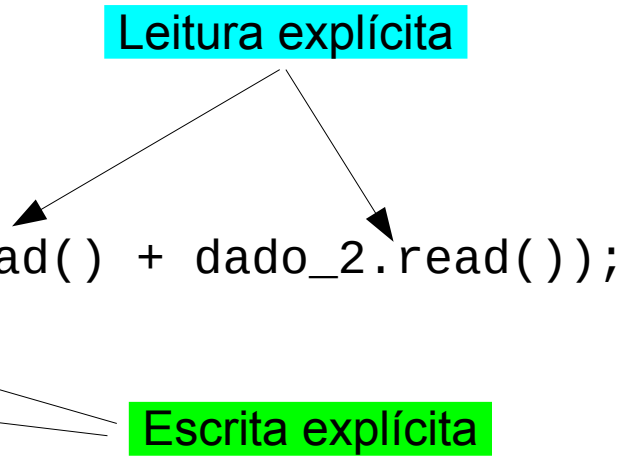
Portas...

- Portas e sinais podem ser do tipo:
  - Tipos primitivos de C++
    - int, float, double, bool...
  - Tipos definidos pelo SystemC
    - sc_int, sc_lv, sc_bigint...
  - Tipos definidos pelo usuário
    - Classes, estruturas, ...
- Sintaxe:
  - sc_in<*tipo_porta*> pi1, pi2..., pin ; // entrada
  - sc_out<*tipo_porta*> po1, po2, ..., pon; // saida
  - sc_inout<*tipo_porta*> pio1, pio2, ..., pion; // E/S

Leitura e Escrita

- Acesso a dados através de portas pode ser feito usando as funções read() e write()
- Compilador algumas vezes faz conversão automática
- Ex:

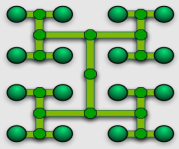
```
SC_MODULE (modulo) {
    sc_in<int> dado_1, dado_2;
    sc_in<bool> cond;
    sc_out<int> res;

    void soma_cond () {
        if (cond.read())
            res.write(dado_1.read() + dado_2.read());
        else
            res.write(0);
    ...
```



Canais

- Canais implementam os métodos que realizam a comunicação entre módulos e entre processos em um módulo
- Podem ser conexões ponto a ponto ou multiponto
- Uma flexibilidade desta abordagem é que pode-se trocar um canal por outro, desde que tenham a mesma assinatura
  - isso permite um processo de refinamentos sucessivos da intercomunicação entre módulos: pode-se trabalhar com protocolos abstratos no princípio, e ir detalhando o canal até sinais de hardware

Universidade de Brasília



LAICO

Canais Primitivos

- São canais que suportam o método de acesso *request-update*
- Request-update simula concorrência entre eventos de forma similar ao delta delay em VHDL
- Canais primitivos derivam direta ou indiretamente da classe sc_prim_channel, que implementa o método request_update() e define o método virtual update()

Exemplos de Canais Primitivos

- Sinal de hardware: **sc_signal<T>**
- Canal tipo fila: **sc_fifo<T>**
  - implementa as interfaces sc_fifo_in_if<T> e sc_fifo_out_if<T>
  - estes métodos podem ser bloqueantes ou não bloquenates
- Chave de exclusão mútua: **sc_mutex**, para delimitar regiões críticas de variáveis compartilhadas

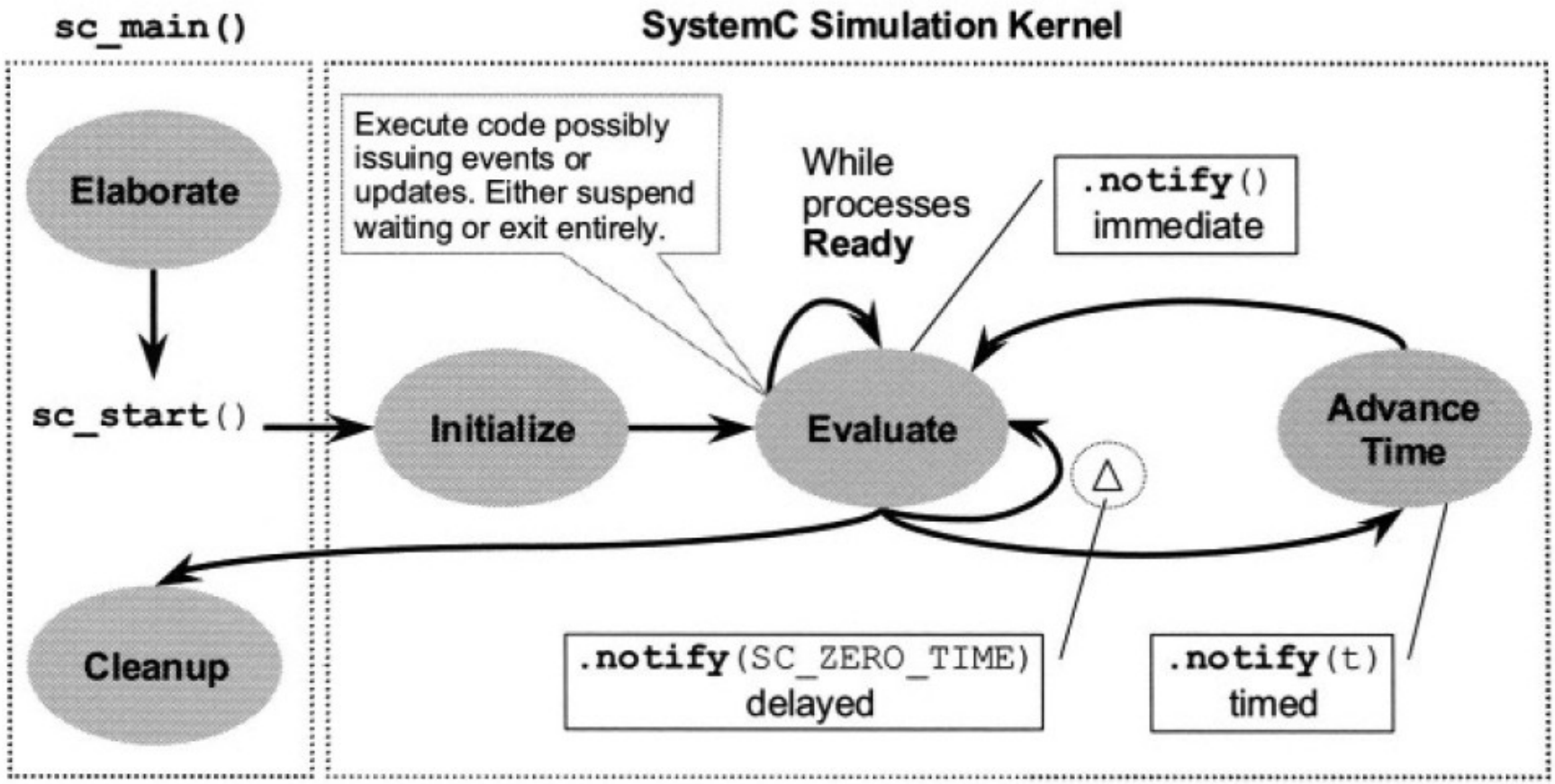
Canais Hierárquicos

- Canais primitivos não contém outras estruturas SystemC
- Canais hierárquicos, entretanto, podem conter outros módulos e processos
  - Ex: *network on chip*, NoC, é uma forma de comunicação intra pastilha que consiste em uma rede de roteadores que se comunicam via pacotes, com capacidade de endereçamento dos módulos IP

Eventos

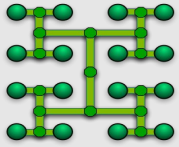
- Eventos em SystemC estão associados a objetos da classe `sc_event`
- Eventos acontecem em um instante de tempo, não tem duração nem valor associado a ele
- Eventos devem ser *observados* para que os seus efeitos sejam úteis
- São utilizados para representar condições que podem ocorrer no curso de uma simulação, controlando o disparo de processos

Simulação no SystemC



Do livro: SystemC from Ground-Up

Universidade de Brasília



LAICO

Simulação no SystemC

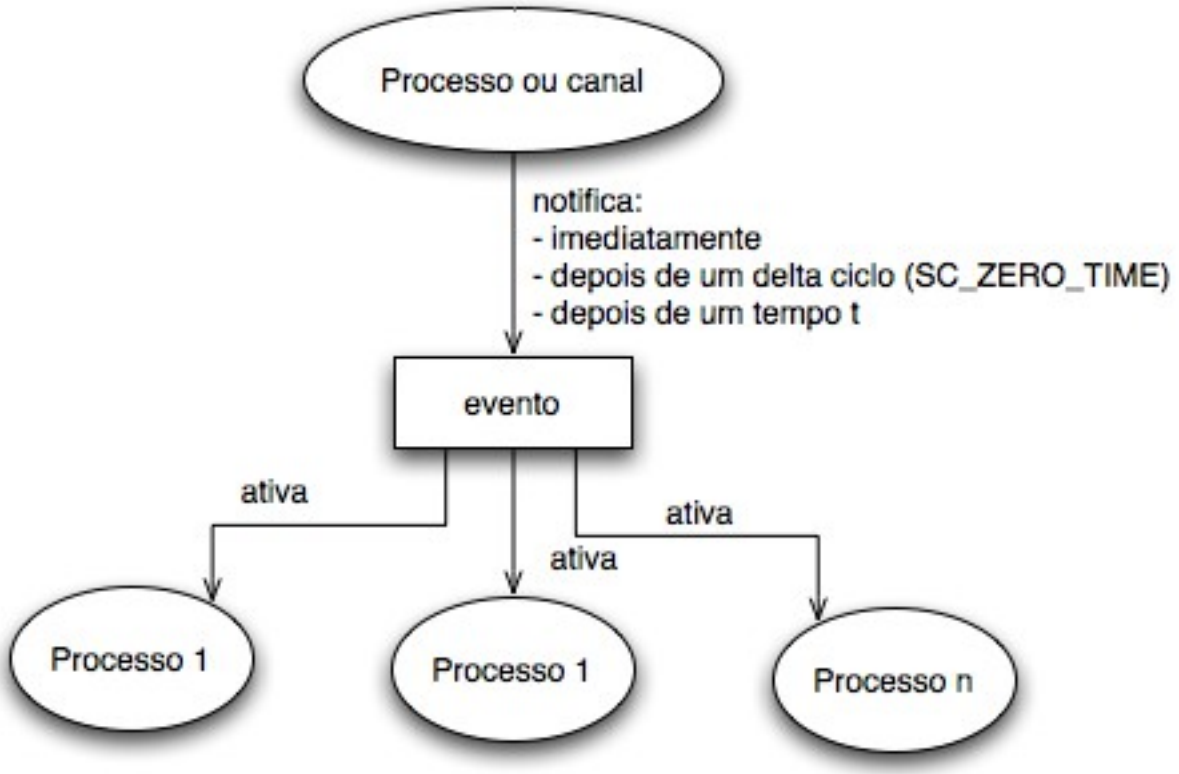
- Duas fases principais: elaboração e execução
- Elaboração:
  - Na elaboração, os módulos são instanciados e a hierarquia construída
  - Ocorre no módulo principal, antes de sc_start()
- Execução:
  - `sc_start()`é executado e chama o kernel de simulação
  - Durante a inicialização, os processos são executados em uma ordem aleatória
  - Durante a execução, processos são simulados quando os eventos relacionados são ativados

Simulação em SystemC

- Cada processo é executado até que termina (`return`) ou é suspenso (`wait`)
- Os processos suspensos ficam em uma lista de espera para executar
- Processos suspensos são ativados pela ocorrência de eventos
- Um objeto evento (`sc_event`) está associado a um módulo ou processo em SystemC
- O dono do objeto evento se encarrega de notificar a ocorrência de uma mudança
- O objeto evento mantém uma lista de processos sensíveis a ele

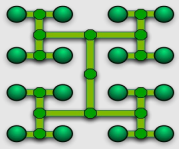
Notificação de Eventos

- Notificação de evento e disparo de processos



Momento da Ativação

- Notificação imediata:
  - Aciona os processos no mesmo instante, sem esperar pela avaliação dos outros processos segundo a filosofia *request-update*
- Notificação *delay* zero:
  - Processo espera pela fase de update antes de ser executado
- Notificação com atraso:
  - Escalona o evento em uma lista de eventos, para ativar os processos quando o tempo de simulação atingir o período especificado



Sensitividade Estática

- Determinada em tempo de compilação, se mantém durante toda a simulação
- Os sinais especificados ativam o processo quando da ocorrência de eventos
  - Ex:
    - SC_METHOD(m);
    - sensitive << a << b;

Sensitividade Dinâmica

- Uma thread pode se tornar momentaneamente sensível a outro evento, que não esteja em sua lista de sensitividade.
  - O comando wait() é utilizado para tornar uma thread sensível a um evento
    - wait(evento);
    - wait (e1 & e2 & e3); // conjunção de eventos
    - wait (e1 | e2 | e3); // disjunção de eventos
    - wait (100, SC_NS); // tempo
    - wait (100, SC_NS, e); // evento ou limite de tempo
  - A thread fica insensível a lista estática, até o evento ocorrer

Sentitividade em Métodos

- Um método pode ter sua sensitividade momentâneamente redirecionada a eventos
  - next_trigger(e) // mesmos parâmetros de wait()
    - muda temporariamente a sensitividade do método, sobreescrevendo a lista estática
    - não suspende o método no meio de sua execução, o método executa até o final, e depois