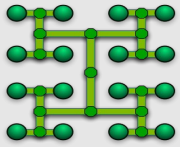
Universidade de Brasília

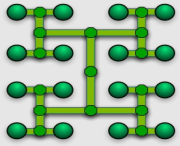
Introdução à Modelagem Transacional de Sistemas em Silício

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



LAICO

Universidade de Brasília



LAICO

Sumário

- Introdução
- Modelagem
- Templates
- SystemC
- Interfaces
- Portas
- Canais
- Processos

- Ref:
  - System Design with SystemC, T. Grötker et al. Kluwer Academics
  - SystemC: From de Ground Up David Black and Jack Donovan, Eklectic Ally
  - Thinking in C++, B. Eckels.

Introdução

- Este curso tem por objetivo oferecer uma introdução à modelagem de sistemas integrados monolíticos - ou sistemas em silício (*SoC – System on Chip*) - em nível transacional

- Introduz-se a linguagem SystemC como ferramenta de modelagem em níveis abstratos

Introdução

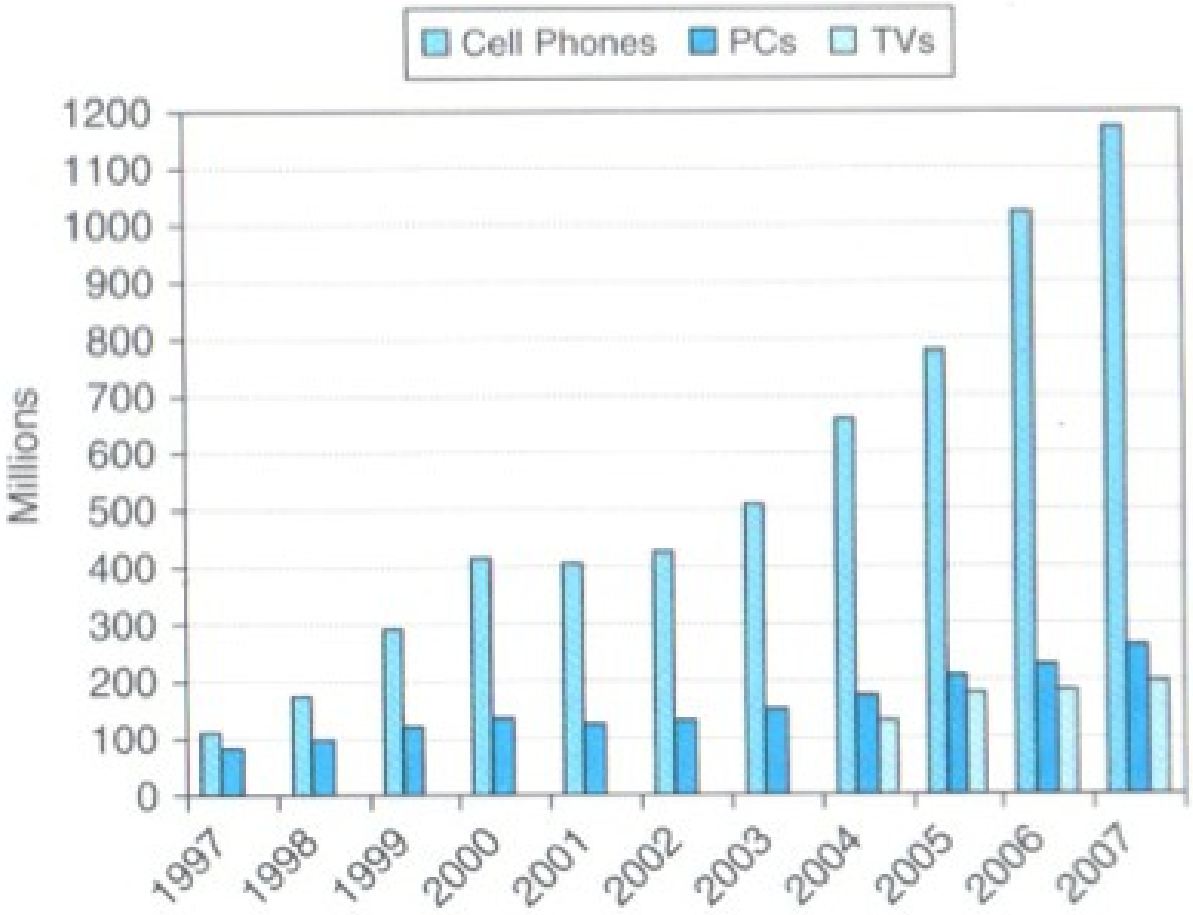
- Complexidade dos sistemas em silício:
  - Não apenas hardware, mas hardware e software
  - Diversos módulos interagindo através de uma estrutura de comunicação complexa
  - Sistemas envolvendo tipicamente:
    - Processadores de propósito geral
    - Processadores de sinais digitais (DSPs)
    - Memórias permanentes e voláteis
    - Controladores de E/S (USB, Firewire, Ethernet, ...)
    - Circuitos de dedicados (decodificadores, cifradores, etc)
    - Comunicação sem fio integrada
    - Conversores para sensores atuadores
    - ...

Sistemas Embarcados

- Ex: BMW série 7 possui 63 processadores embarcados.

- Veículos de última geração tem centenas de processadores interligados em rede

Sistemas Embarcados



Microprocessadores

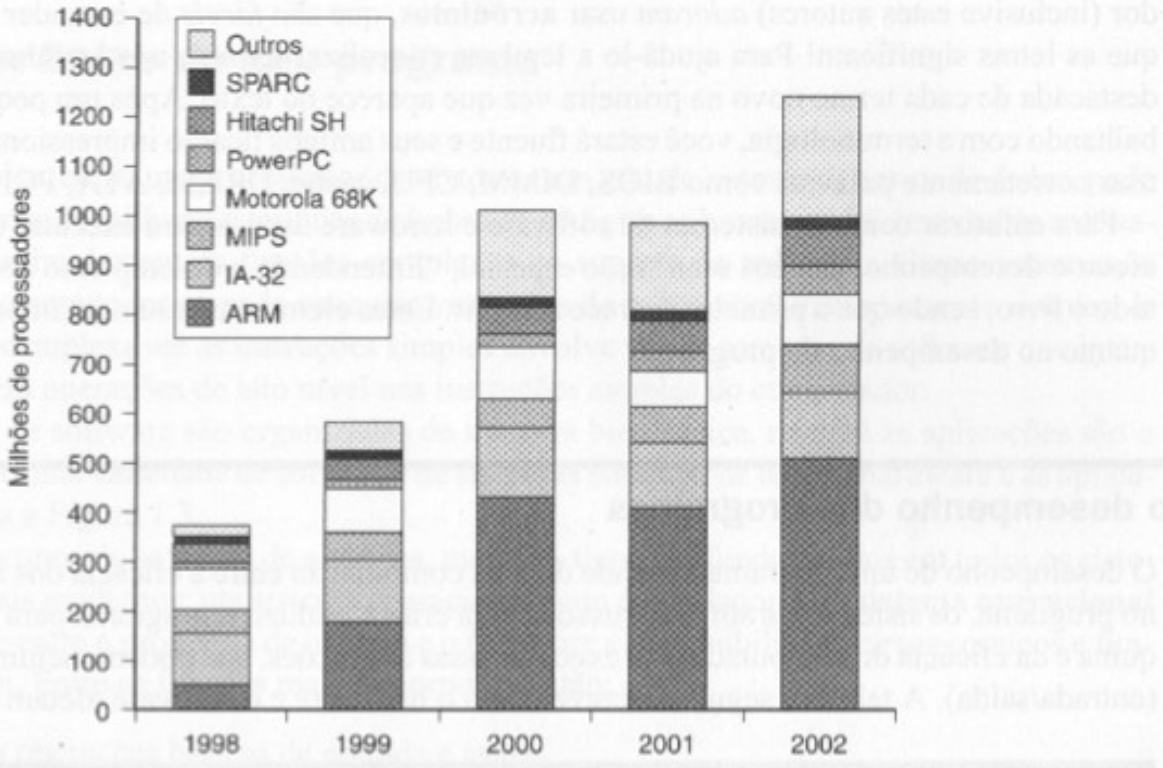
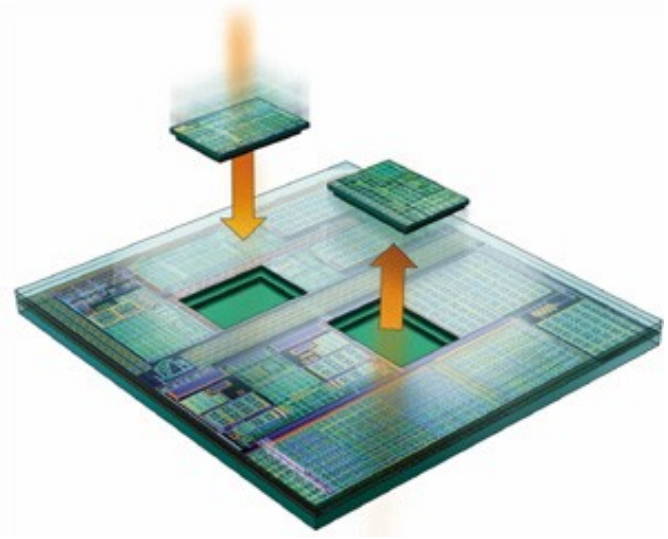


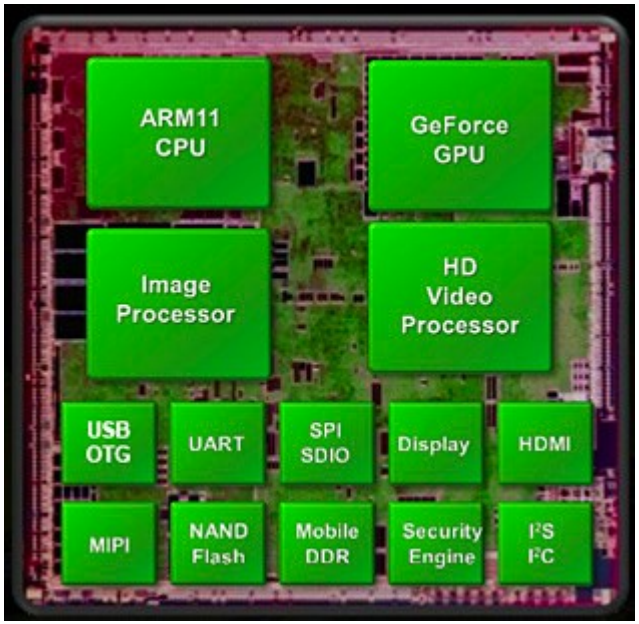
FIGURA 1.2 Vendas de microprocessadores entre 1998 e 2002 por arquitetura combinando todos os usos. A categoria "outros" se refere aos processadores específicos à aplicação ou às arquiteturas personalizadas. No caso do ARM, aproximadamente 80% das vendas são para telefones celulares, sendo que um núcleo do ARM é usado em conjunto com lógica específica da aplicação em um chip.

Sistemas em Silício



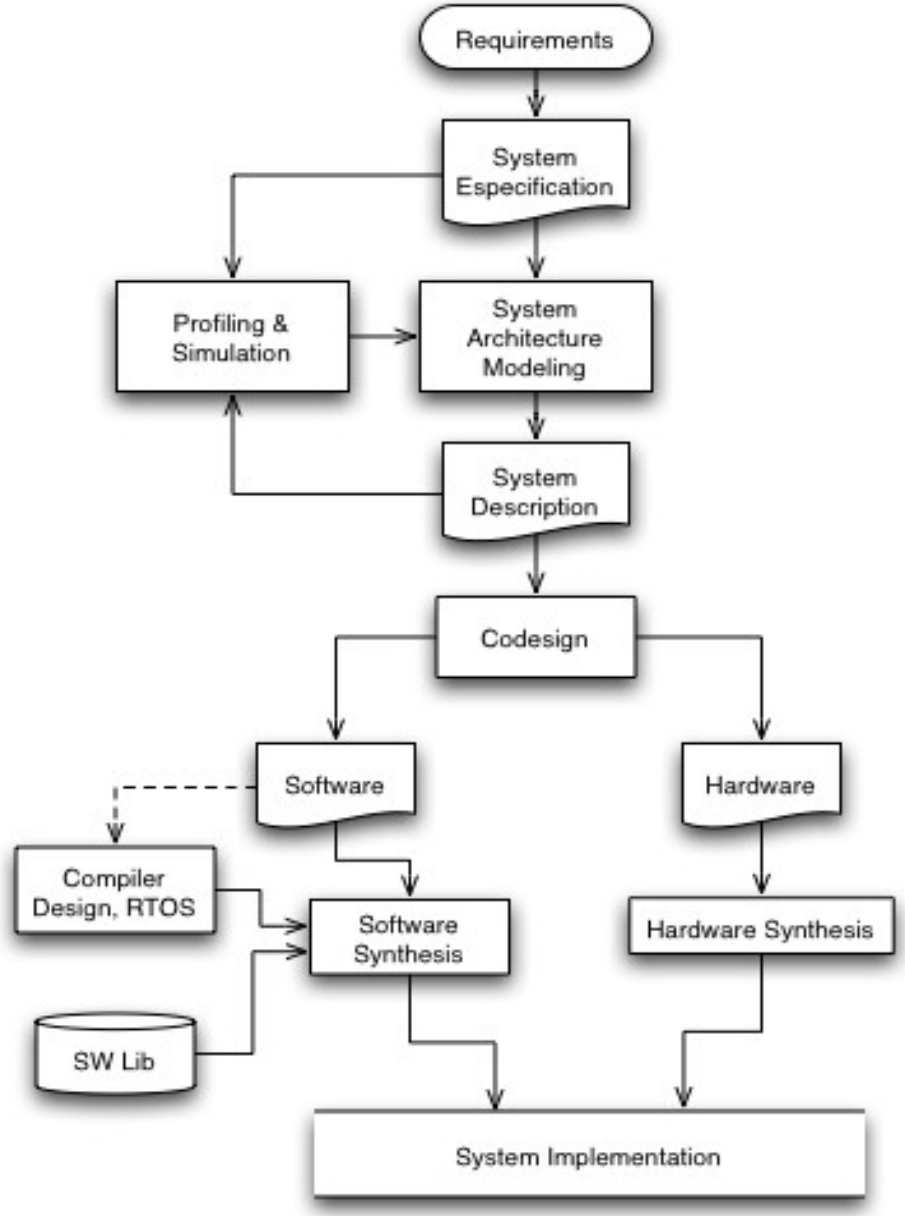
SoC GDS System-on-Chip Graphic Display Streamer

Dolphin Integration

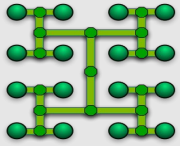


Tegra System on Chip for HandHeld

Fluxo de Síntese



Universidade de Brasília



LAICO

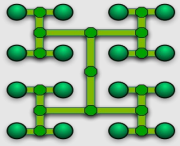
Modelagem e Simulação

- Independente da abordagem adotada, a modelagem e simulação do sistema em nível abstrato é uma etapa cada vez mais importante no desenvolvimento de um SoC
- Modelagem:
  - representação formal de aspectos relevantes do sistema, visando compreender seu funcionamento
- Simulação:
  - possibilidade de obter respostas do sistema modelado a estímulos fornecidos

Eixos de Modelagem

- O nível de abstração pode ser expresso em termos da acurácia dos eixos de modelagem
  - acurácia estrutural
    - o quanto o modelo aproxima-se da estrutura do sistema real
  - acurácia temporal
    - grau de aproximação da temporização do sistema real
  - acurácia funcional
    - o quanto o modelo reflete a funcionalidade do sistema real
  - acurácia da organização de dados
    - o quanto reflete a organização de dados do sistema real
  - acurácia do protocolo de comunicação
    - o grau em que o modelo reflete o protocolo de comunicação real utilizado na aplicação

Universidade de Brasília

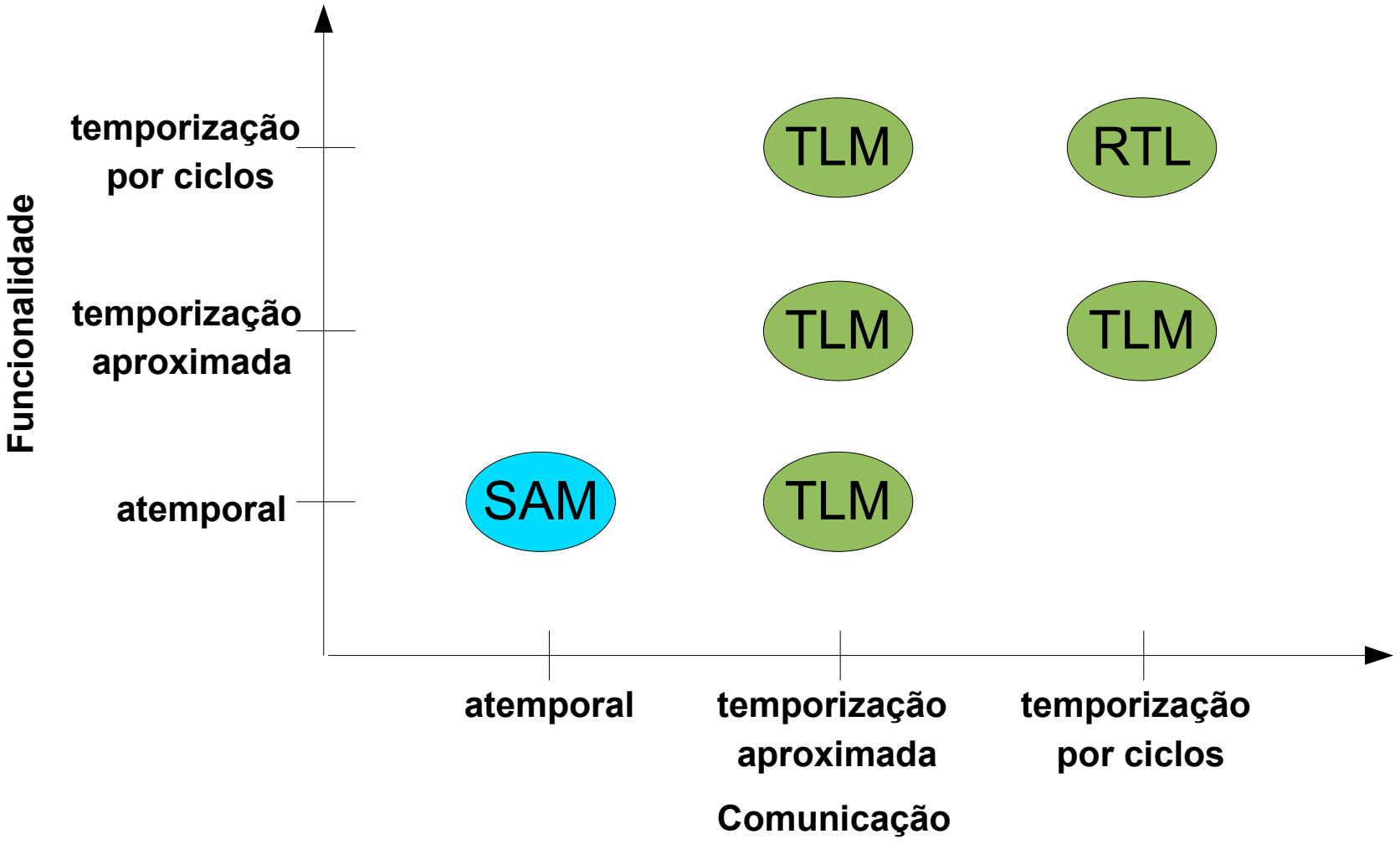


LAICO

Modelagem Transacional

- A modelagem em nível de transações (TLM – *Transaction Level Modeling*) é uma metodologia que está sendo proposta como alternativa para os primeiros passos do projeto de SoCs
- Não há uma definição precisa do que seja TLM
- Algumas classificações foram propostas na literatura, como Lukai e Gajski em CODES´03
- Um conceito básico em TLM é a separação entre computação x comunicação
- Uma *transação* é a transmissão de um conjunto de informações relevante ao problema

Classificação TLM



Modelagem em C++ ?

- Problemas com C++:
  - Reatividade
    - Módulos em hardware reagem a mudanças nas entradas, atualizando as saídas. O método, neste caso, deve ser chamado explícitamente para computar novas saídas
  - Temporização
    - Não existe noção de tempo no programa.
  - Concorrência
    - C++ não modela o paralelismo intrínseco do hardware
  - Tipos de dados
    - C++ não modela tipos de dados de hardware, bit-wise, multivalorados
  - Sinais, protocolos em hardware

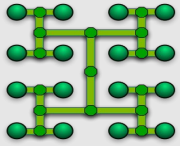
Histórico SystemC

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

Histórico de SystemC

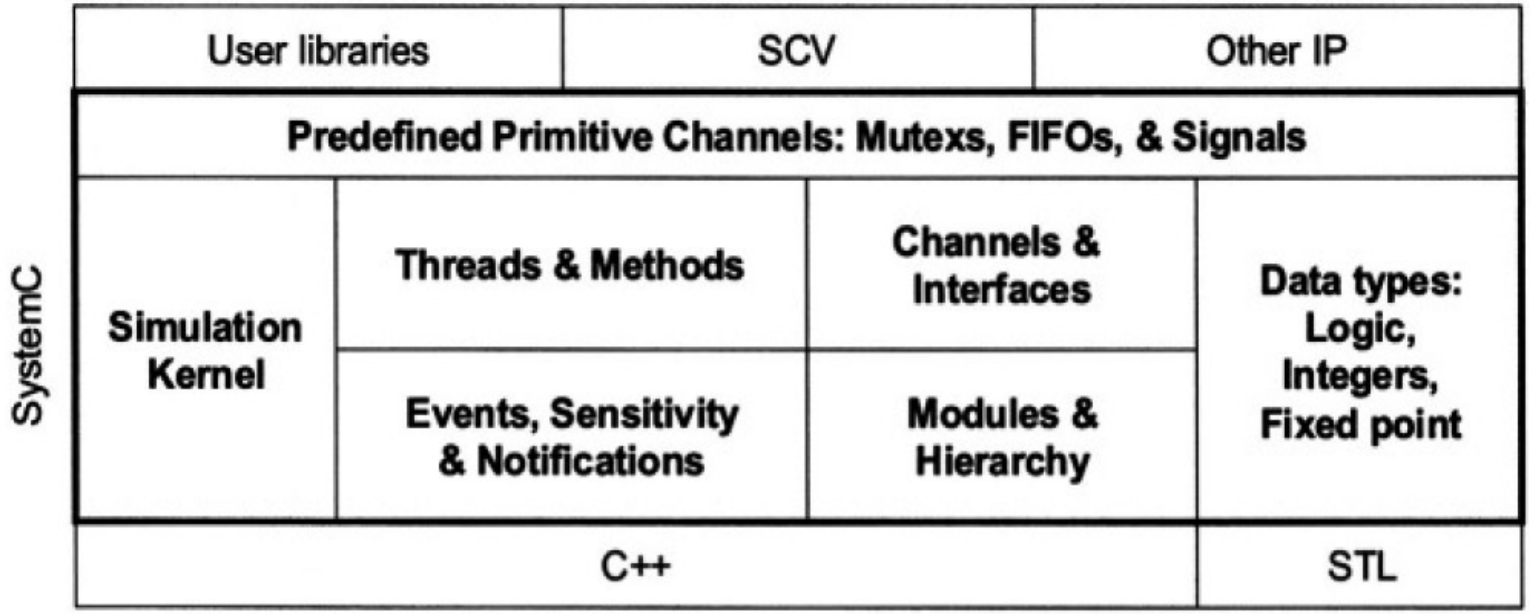
- SystemC 2.0 (Maio 2001)
  - Modelagem em nível de sistema
    - Maior suporte a modelagem em nível de transações, separação entre processamento e comunicação
  - Comunicação: canais, interfaces e portas
  - Eventos como elementos de controle de simulação
- SystemC 3.0 (...)
  - Modelagem de software
  - RTOS
  - Mixed-signal

Universidade de Brasília

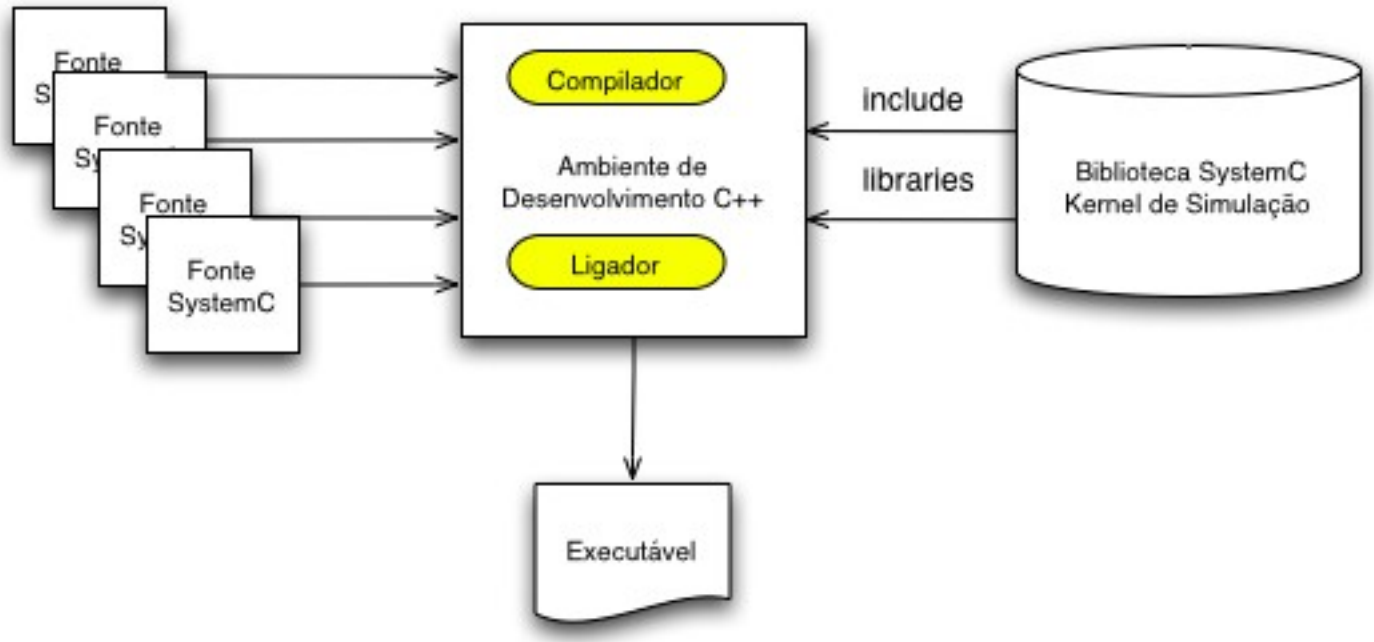


LAICO

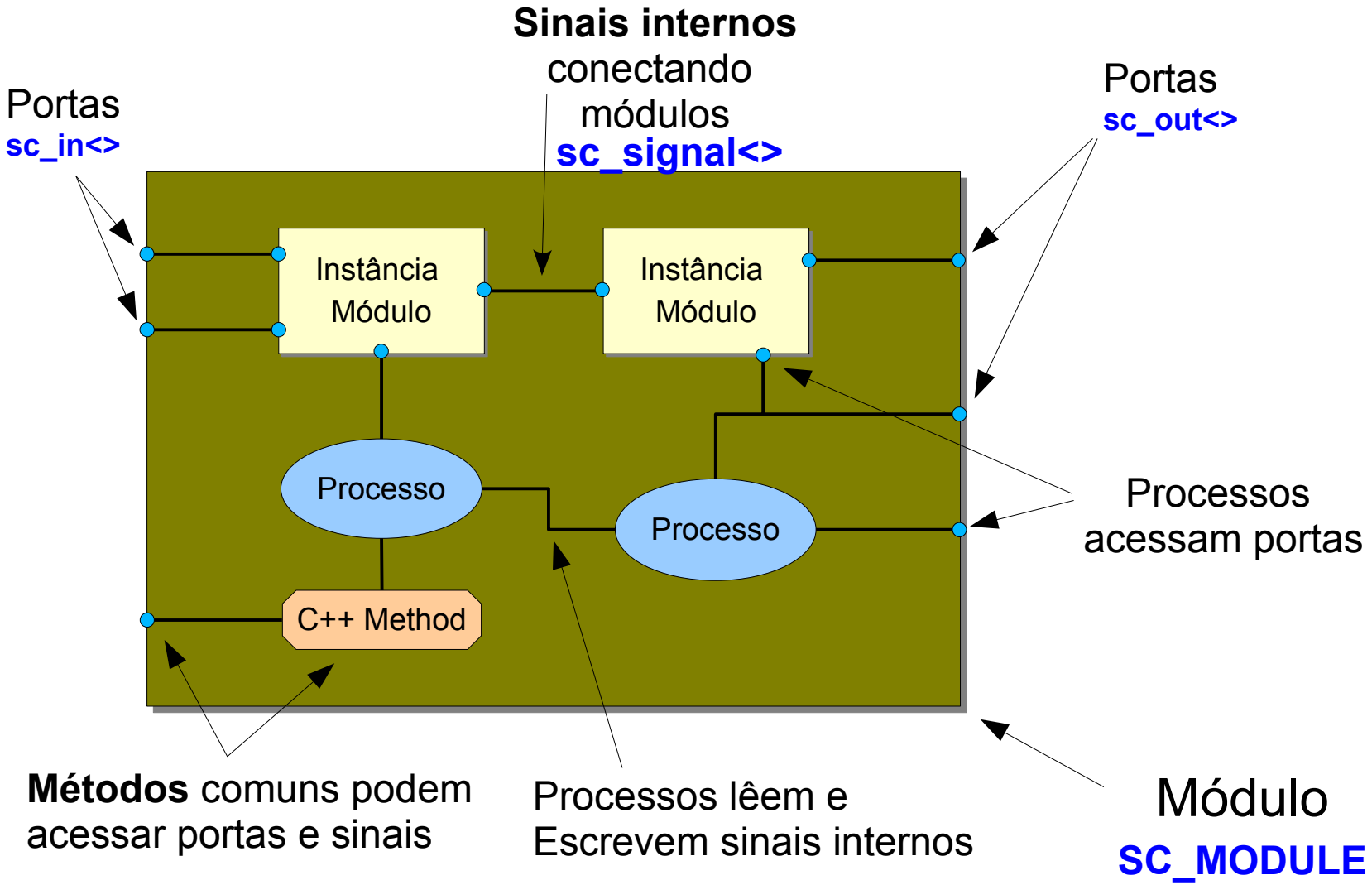
Estrutura do SystemC



Compilando SystemC



Descrição SystemC



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
  - Portas de entrada e saída
  - Processos
    - SC_METHOD
    - SC_THREAD
    - SC_CTHREAD
  - 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
  - Lista de sensitividade: define quais eventos disparam a execução do método (sinais)

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
    - wait (10, SC_NS) # espera 10 nanosegundos
    - wait (event) # espera pela ocorrência de um evento
  - Usualmente executa com um laço infinito, até o final da simulação

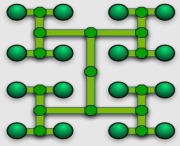
Portas e Sinais

- Portas e sinais modelam elementos de comunicação em hardware
- Os tipos de dados associados às portas e sinais podem ser:
  - Tipos de dados de C++
  - Tipos de dados do SystemC
  - Tipos de dados definidos pelo usuário

Portas e Sinais...

- Portas conectam os processos de um módulo com o meio externo
- Tipos de portas:
  - **sc_in<tipo_dado>** : porta de entrada
  - **sc_out<tipo_dado>** : porta de saída
  - **sc_inout<tipo_dado>** : porta bidirecional
- Sinais são definidos pela classe sc_signal:
  - **sc_signal<tipo_dado>**

Universidade de Brasília



LAICO

Leitura e Escrita

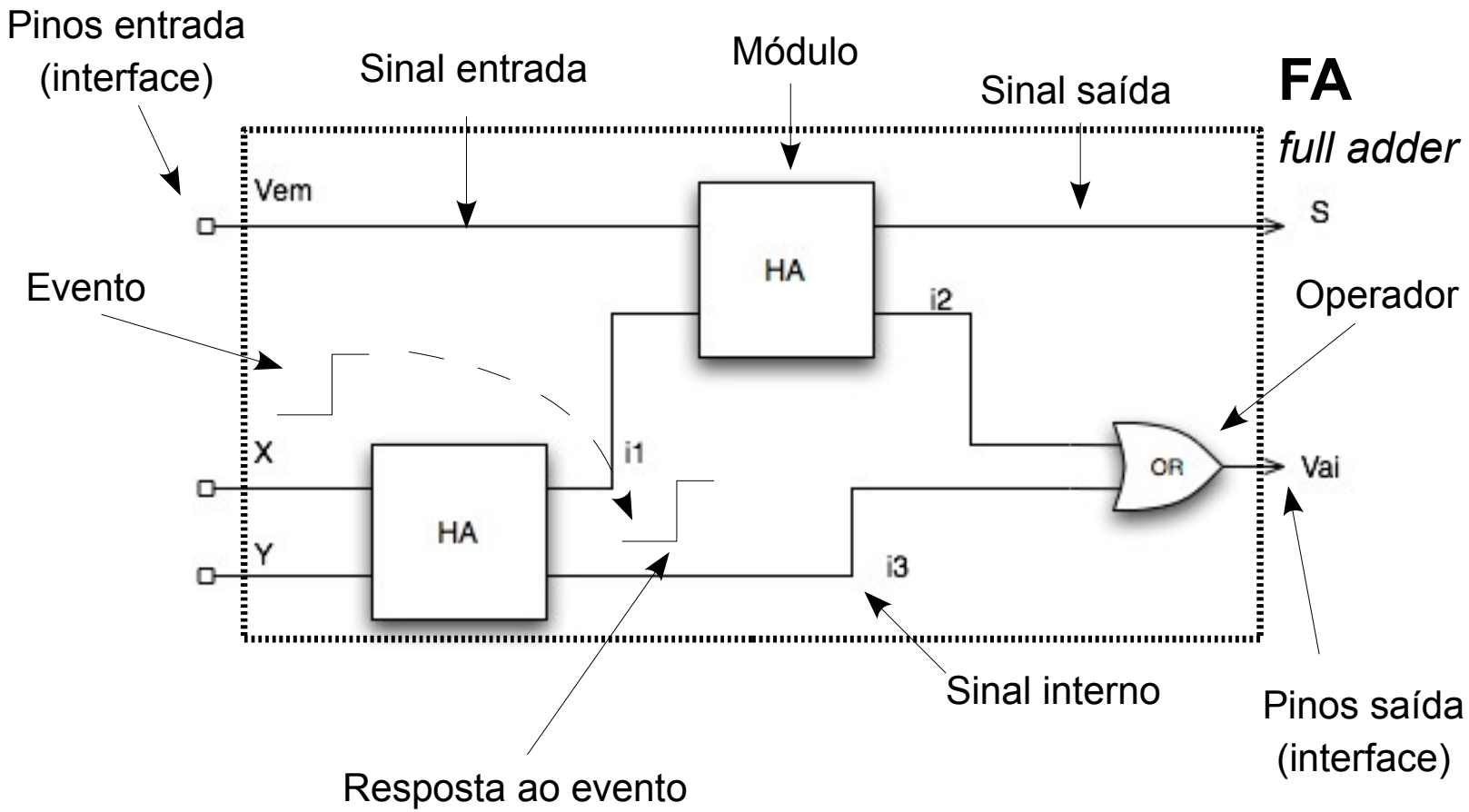
- Portas e sinais pode ser lidos invocando-se os métodos *read() e write(dado)*:

Exemplo:

```
SC_MODULE (fa) {
 sc_in<bool> x, y, vem;
 sc_out<bool> soma, vai;
 sc_signal<bool> s1, s2, s3;

 bool b = x.read();
 vai.write(false);
...
```

Exemplo FA SystemC



Exemplo FA SystemC

```
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

Exemplo FA SystemC

```
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

Exemplo FA SystemC

```
SC_MODULE (fa) {
  sc_in<bool> x, y, vem;
  sc_out<bool> soma, vai;
  ha_sc *h1, *h2;
  sc_signal<bool> s1, s2, s3;
  SC_CTOR (fa) {
    h1 = new ha_sc("h1");
    h2 = new ha_sc("h2");
    h1->x(x);  h1->y(y);   h1->s(s1);  h1->v(s2);
    h2->x(s1); h2->y(vem); h2->s(soma); h2->v(s3);
    SC_METHOD(proc);
    sensitive << s2 << s3;
  }
  void proc() {
    vai = s2 | s3;
  }
};
```

Ponteiros para módulos

Construtor do módulo

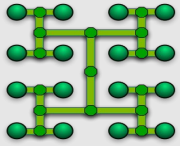
Instanciação de módulo

Conexão de sinal à porta

Declaração de método

Definição da lista de Sensitividade do método

Universidade de Brasília

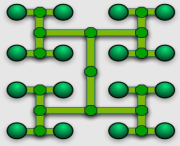


LAICO

Tipos de Dados

- Tipos primitivos do C++:
  - *long, int, short, char, unsigned long, unsigned int, unsigned short, unsigned char, float, double, long double* e *bool*
- Tipos SystemC:
  - Escalares: sc_bit, sc_logic
  - Inteiros: sc_int, sc_uint, sc_bigint, sc_biguint
  - Vetoriais: sc_bv, sc_lv
  - Ponto fixo: sc_fixed, sc_ufixed, sc_fix, sc_ufix

Universidade de Brasília



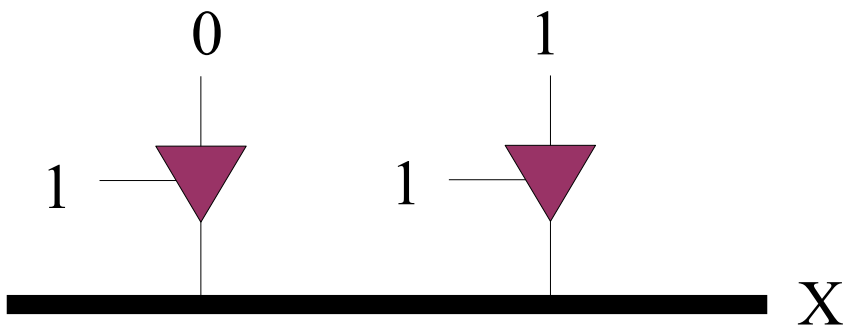
LAICO

Tipos de Dados

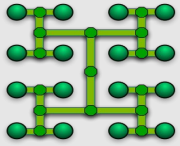
- Booleanos: **sc_bit e sc_bv<*n*>**
  - Podem valer 1 (verdadeiro) ou 0 (falso)
- Multivalorados: **sc_logic e sc_lv<*n*>**
  - 4 valores: 1, 0, X, Z
- Inteiros: **sc_int<*n*> e sc_uint<*n*>**
  - Inteiros de *n* bits com e sem sinal
- Inteiros grandes: **sc_bigint<n> e sc_biguint<n>**
  - Inteiros de *n* bits, com n > 64, com e sem sinal

Lógica Multivalorada

- 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



Universidade de Brasília



LAICO

Exemplos

```
sc_bit flag(SC_LOGIC_1); // melhor bool
sc_bv<5> positions = "01101";
sc_bv<6> mask = "100111";
sc_bv<5> active = positions & mask;// 00101
sc_bv<1> all
          = active.and_reduce(); // SC_LOGIC_0
positions.range (3,2) = "00"; // 00001
positions [2] = active[0] ^ flag;
sc_logic b('1');
sc_lv<7> data("zZXX011");
sc_lv<9> fio = "0101xxxZZ";
```

Operadores Lógicos-Aritméticos

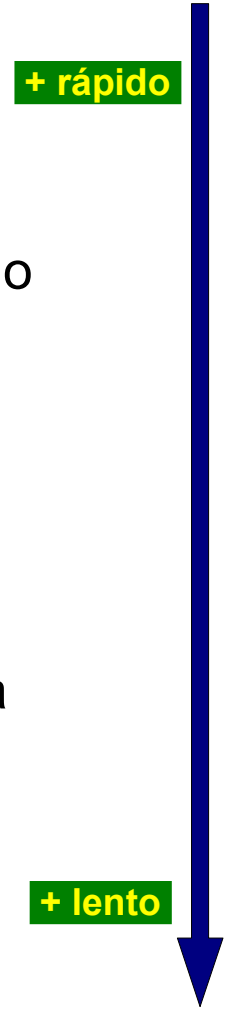
- Unários:
  - z é tratado como x
  - ~ negação bit por bit
  - ! negação lógica
  - - negativo
  - ++ auto-incremento
  - -- auto-decremento
- Binários
- Se um dos bits envolvidos for x ou z, todo o resultado é x.
  - * multiplicação
  - / divisão
  - % resto da divisão
  - + soma
  - - subtração

Operadores Lógico-Aritméticos

- Lógica de redução
  - and_reduce()
    - "e" lógico bit a bit
  - or_reduce()
    - "ou" lógico bit a bit
  - xor_reduce()
    - "ou-exclusivo" lógico bit a bit
- Lógicos binários
  - >> desloca direita
  - << desloca esquerda
    - Preenche com zeros
    - Negativo não permitido
  - Comparação:
    - == != < > <= >=
  - & "e" lógico
  - | "ou" lógico
  - ^ "ou-exclusivo" lógico

Que tipo de dados usar?

- Utilizar tipos nativos C++ sempre que possível
  - sc_int<> e sc_uint<>
    - Dados com menos de 64 bits, com tamanho arbitrário
    - Operações lógicas e aritméticas
  - sc_bit, sc_bv<>
    - Operações booleanas, quando não puder usar bool
  - sc_logic, sc_lv<>
    - Modelagem de alta impedância, lógica multivalorada
  - sc_bigint<>, sc_biguint<>
    - Dados com mais de 64 bits
  - sc_fix<>, sc_ufix<>
    - Ponto fixo



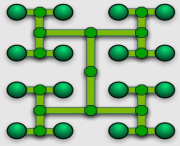
Temporização

- 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
- 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

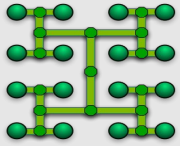
- Criação de variáveis do tipo sc_time:

```
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) ...
```

Universidade de Brasília

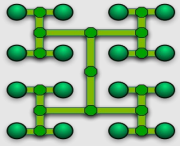


LAICO

Alguns Métodos

- **sc_time_stamp()** - retorna o tempo atual de simulação como um objeto **sc_time**
- **sc_simulation_time()** - retorna o tempo atual em double
- **sc_set_time_resolution**(double, sc_time_unit) – define o passo de tempo usado na simulação. Por padrão é SC_PS (picosegundos)

Universidade de Brasília



LAICO

Utilizando sc_start()

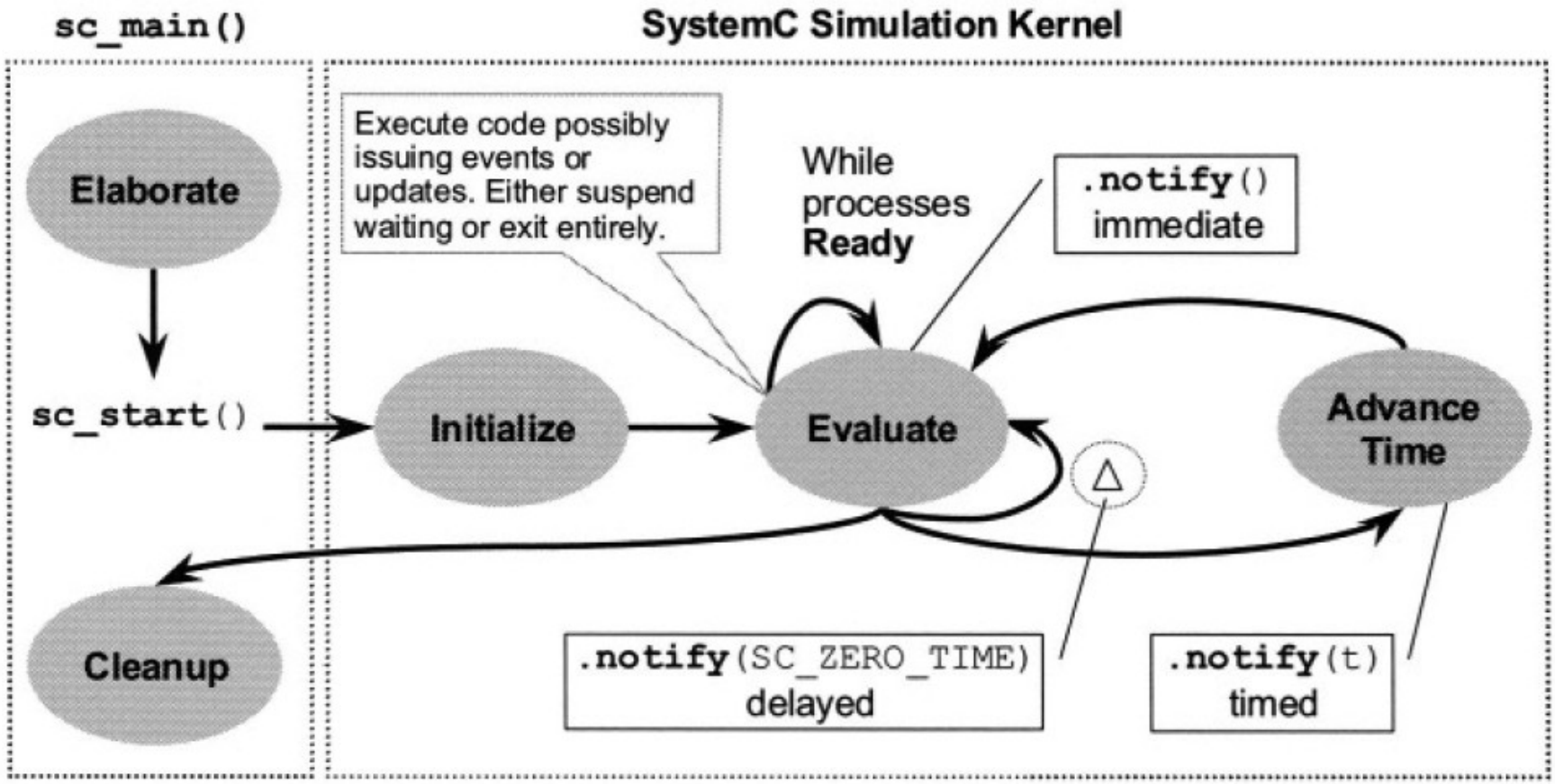
- sc_start() é o método que inicia a fase de simulação no SystemC
  - sc_start() executa para sempre
  - sc_start(const sc_time&) executa pelo período de tempo indicado
  - sc_start(double, sc_time_unit) tempo indicado por double x time_unit
  - Ex:
    - sc_start(100.0, SC_PS);

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

Simulação no SystemC

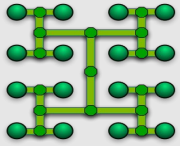


Do livro: SystemC from Ground-Up

Simulação no SystemC

- Na elaboração, os módulos são instanciados e a hierarquia construída
- `sc_start()` é executado e chama o kernel de simulação
- Durante a inicialização, os processos são colocados em uma lista de processos prontos para executar
- Durante execução os processos são retirados da lista e executados de forma aleatória

Universidade de Brasília

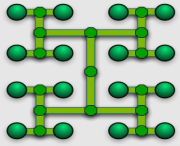


LAICO

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

Universidade de Brasília



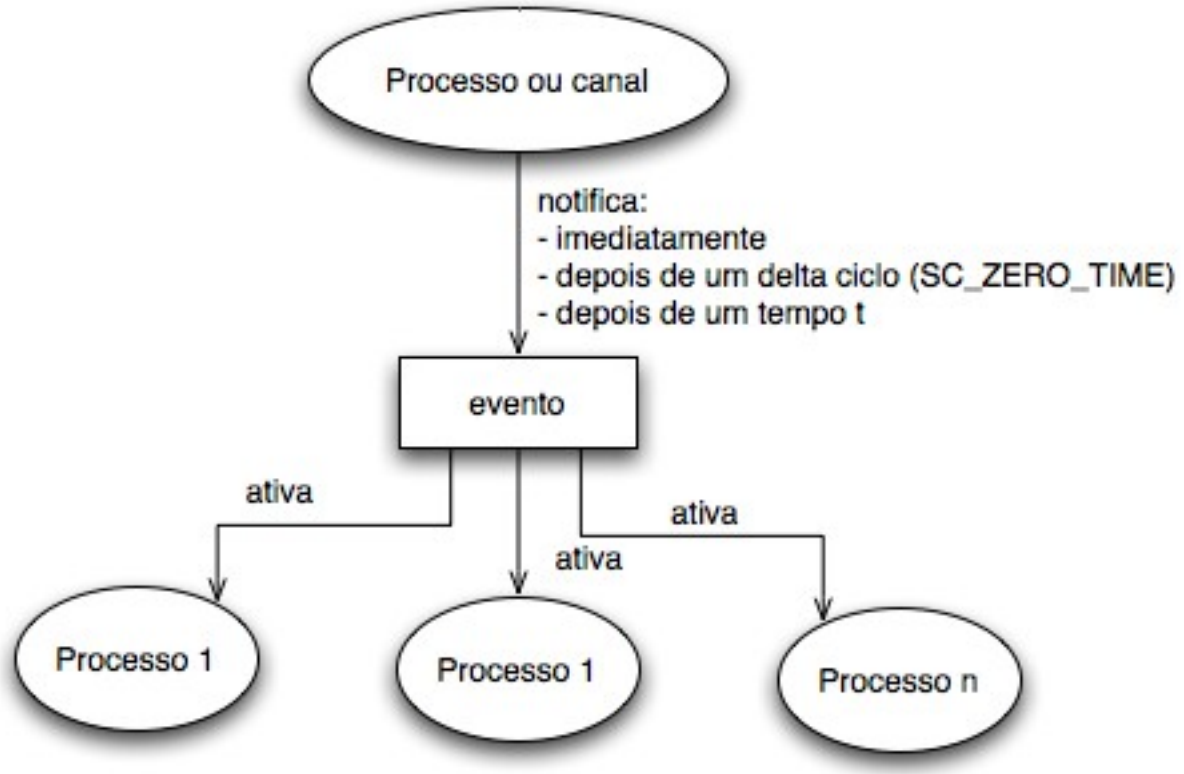
LAICO

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

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;

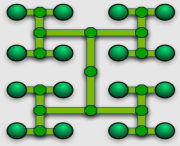
Sensitividade Dinâmica

- 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 Dinâmica: 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

Universidade de Brasília

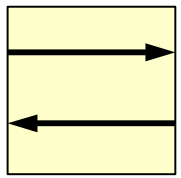


LAICO

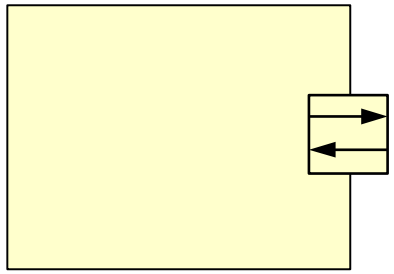
Interfaces, Portas e Canais

- *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*

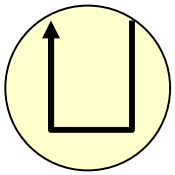
Representação Gráfica



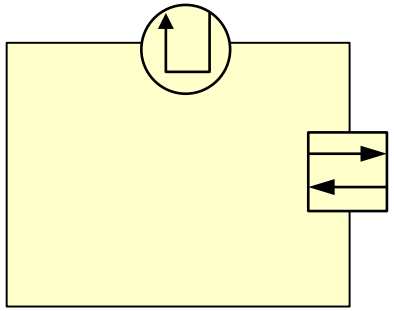
port



Módulo com porta

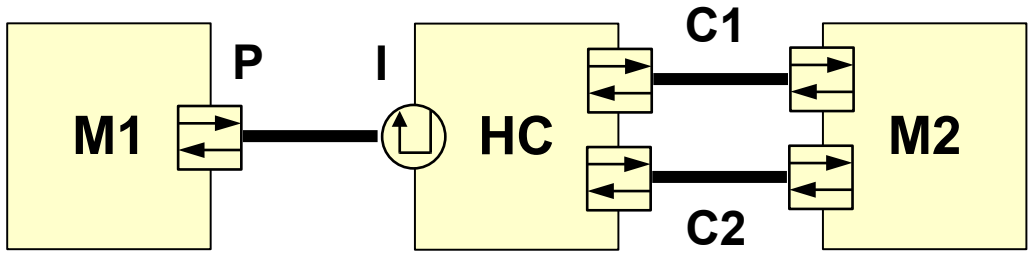


interface

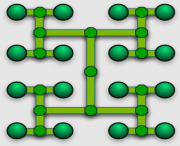


Módulo com interface e porta

primitive channel



Universidade de Brasília

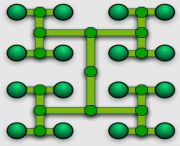


LAICO

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**

Universidade de Brasília

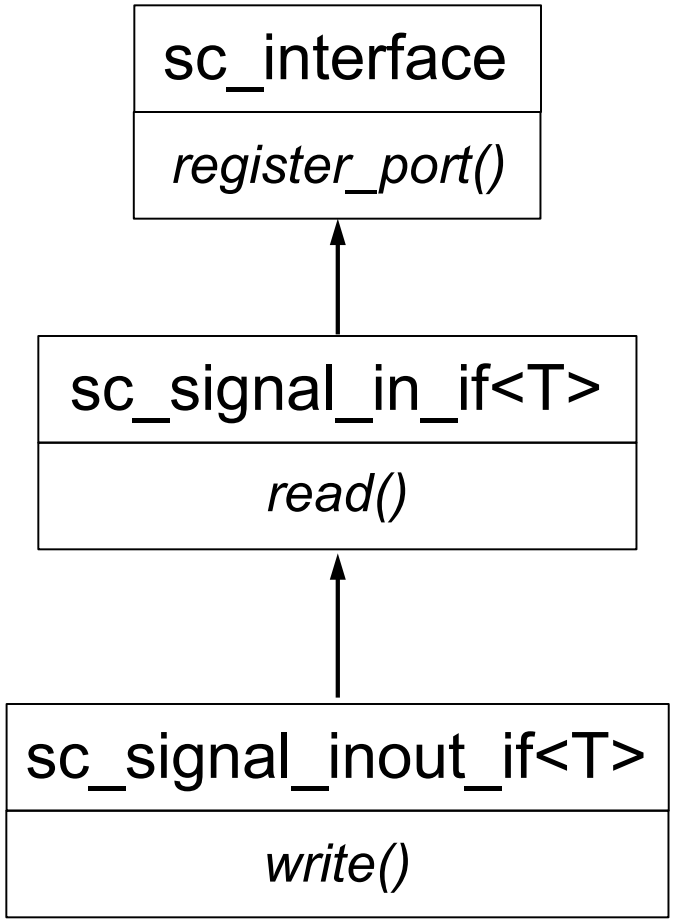


LAICO

Interfaces...

- **sc_signal_in_if<T>:** interface derivada diretamente de sc_interface, parametrizada pelo tipo T. Define o método read() que retorna uma referencia constante para T
- **sc_signal_inout_if<T>:** interface derivada diretamente de sc_signal_in_if<T>. Acrescenta o método write(T), que escreve um elemento tipo T no canal

Hierarquia de Interfaces

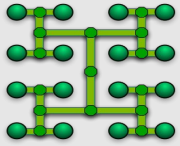


Exemplo de Interface

```
// ----------------------------------------
// 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

Portas

- Uma porta é um objeto através do qual um módulo pode acessar a interface de um canal
- Portas básicas:
  - sc_in<T> : chama método read() do canal
  - sc_out<T> : chama método write() do canal
  - sc_inout<T> : chama ambos
- sc_port_base é a classe abstrata raiz de todas as portas
- outras podem ser criadas

Portas...

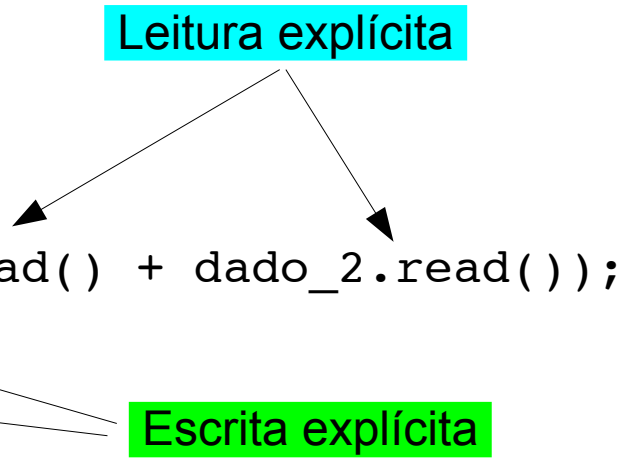
- Portas e sinais podem ser do tipo:
  - Tipos primitivos de C++
    - int, float, double, bool...
  - Tipos definidos pelo SystemC
    - sc_int, sc_lv, sc_bigint...
  - Tipos definidos pelo usuário
    - Classes, estruturas, ...
- Sintaxe:
  - sc_in<*tipo_porta*> pi1, pi2..., pin ; // entrada
  - sc_out<*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);
    ...
```



SC_FIFO

- As portas **sc_fifo_in<T>** e **sc_fifo_out<T>** são muito utilizadas na modelagem de alto nível
- Conectam-se a canais tipo **sc_fifo<T>**, que implementa uma fila de comunicação
- **sc_fifo_in<>** implementa a interface de comunicação **sc_fifo_in_if<>**
- **sc_fifo_out<>** implementa a interface de comunicação **sc_fifo_out_if<>**
- Interface define operações **read** e **write** bloqueantes

Template de Porta

- SystemC provê um template para criação de portas:

```
template<class IF, int N=1>
class sc_port : … //detalhes omitidos
{
   public:
      IF* operator->();
      // outras funções
};
```

Template de Porta...

- N representa o número de interfaces que podem ser conectadas à porta (default uma)
- O operador " -> " retorna um ponteiro para a interface a qual a porta está associada
- Assim:

```
p->read() indica que está sendo chamado
o método read() da interface a
qual p está associado
```

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

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