

Descrição de hardware em SystemC

Gustavo Girão
ggbsilva@inf.ufrgs.br

Baseado no mini-curso de Bruno Zatt (EMICRO 2009)

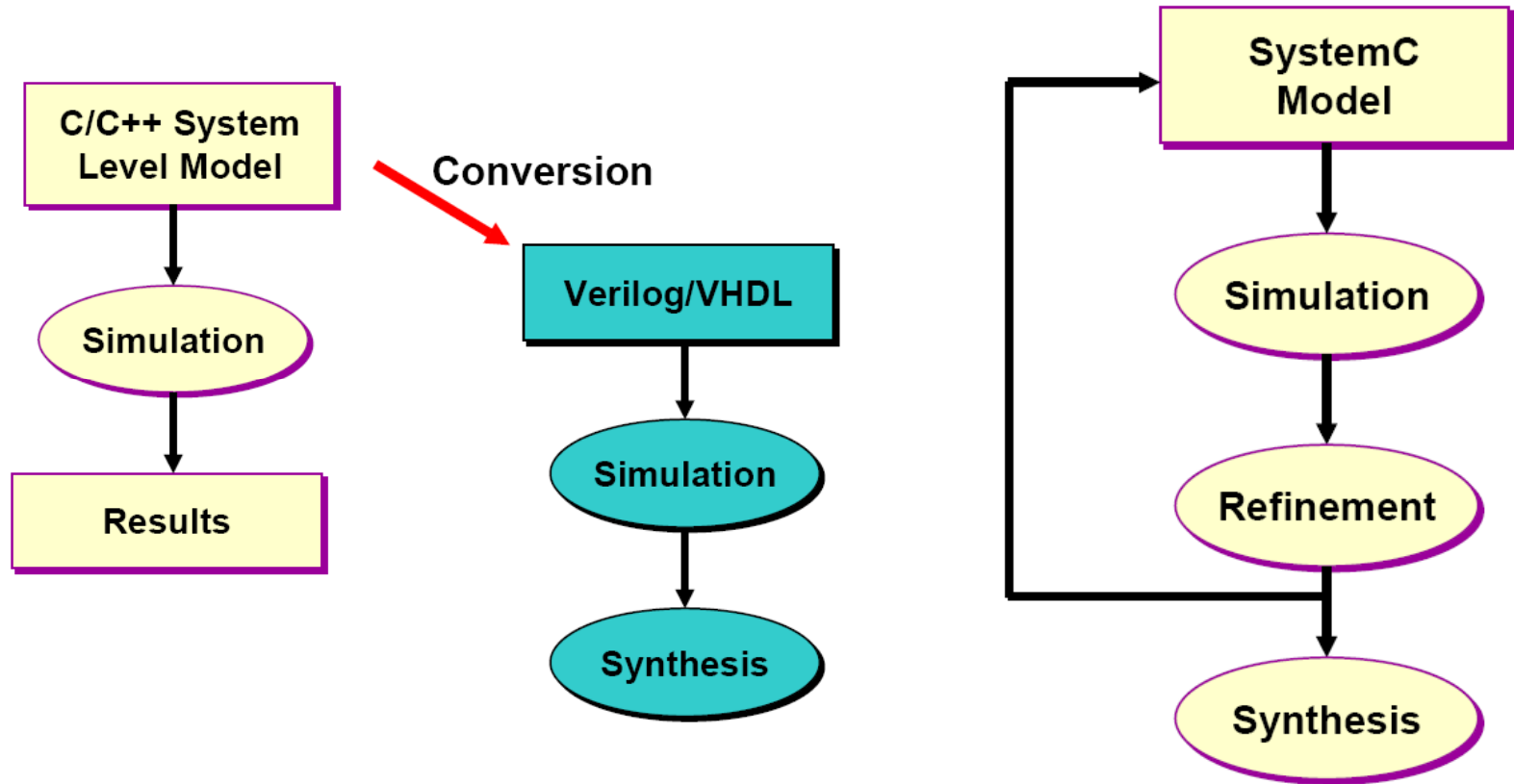
Sumário

- ▶ **Introdução**
 - ▶ O que é SystemC
 - ▶ Estruturas, primitivas e tipos de dados
 - ▶ Exemplo
- ▶ **Níveis de abstração**
 - ▶ Modelagem de Sistemas
 - ▶ Níveis de Abstração
- ▶ **Compilação**

SystemC

- ▶ SystemC NÃO é um linguagem
- ▶ Uma biblioteca de classes e macros para C++
- ▶ Primeira versão em 1999 pela OSCI (*Open SystemC Initiative*)
 - ▶ V 0.9 – 1999
 - ▶ V 1.0 – 2000
 - ▶ V 2.0 - 2001
- ▶ Conectar linguagem de descrição de sistema com HDL (C++ -> HDL)
- ▶ Eliminar erros de conversão C++ -> HDL
 - ▶ Refinamento e não conversão
- ▶ Especificação executável arquitetura e implementação
- ▶ Alta velocidade de simulação em níveis mais altos de abstração

C++/HDL vs. SystemC



Características do SystemC

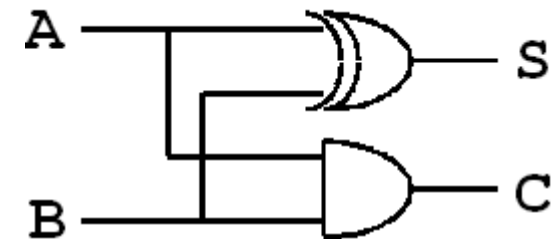
- ▶ Comunicação de HW
 - ▶ Sinais, protocolos..
- ▶ Noção de Tempo
- ▶ Concorrência
 - ▶ Módulos de HW operam em paralelo
- ▶ Reatividade
 - ▶ HW é reativo, responde a estímulos e está em constante interação com o sistema
- ▶ Tipos de dados específicos p/ HW
 - ▶ `sc_logic`, `sc_lv...`

Características do SystemC

Primitivas Pré-definidas: Fifos, Mutex, Sinalização			
Núcleo de Simulação	Threads e métodos	Interfaces e Channels	Tipos de dados: Lógicos Inteiros Ponto Fixo
	Sensitividade à eventos	Hierarquia de módulos	
C++			STL

Exemplo de SystemC

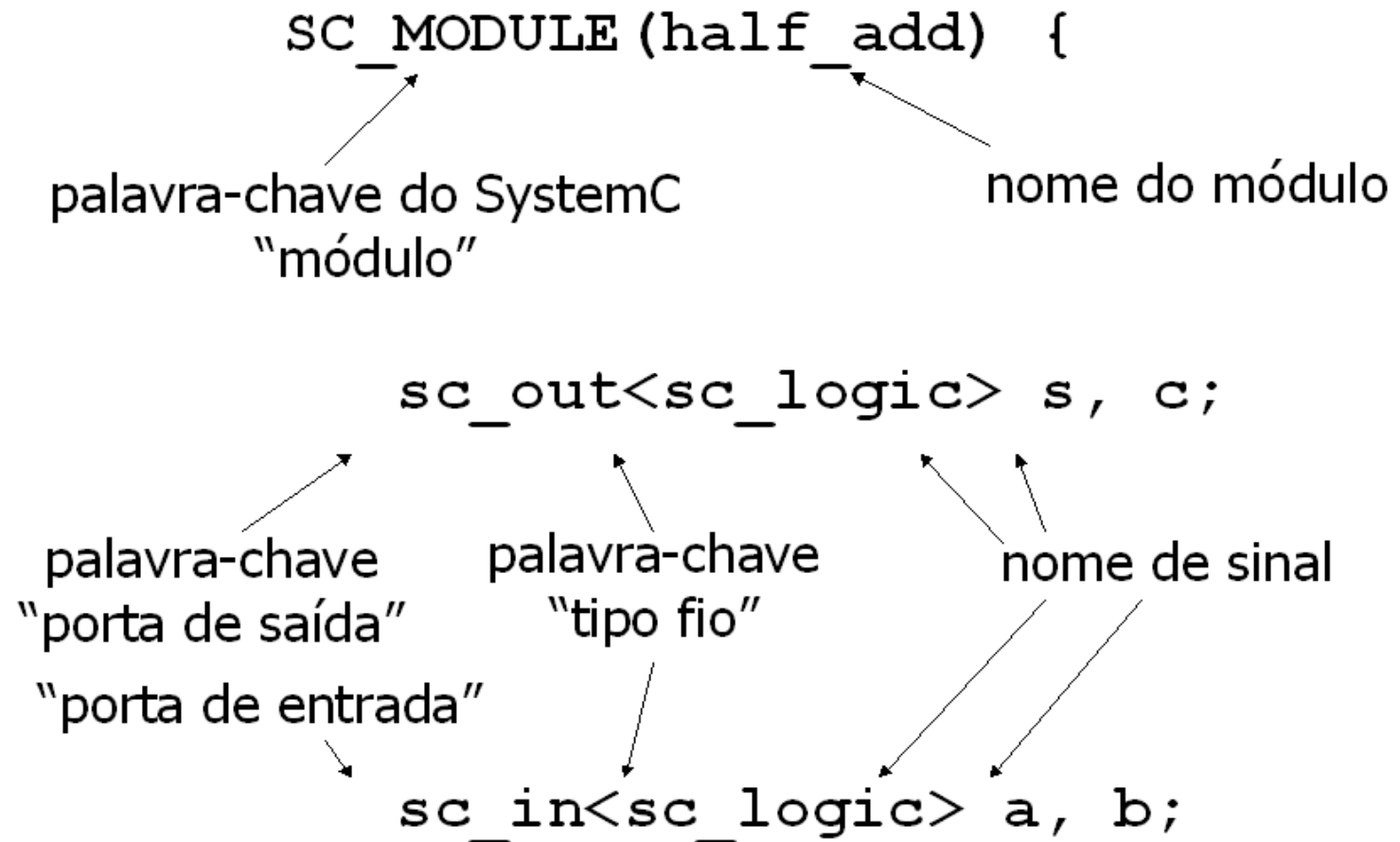
```
SC_MODULE(half_add) {  
    sc_out<sc_logic> s, c;  
    sc_in<sc_logic> a, b;  
  
    void compute () {  
        s = a ^ b;  
        c = a & b;  
    }  
  
    SC_CTOR(half_add) {  
        SC_METHOD(compute) ;  
        sensitive << a << b;  
    }  
};
```



Estrutura

- ▶ Declaração de módulo
- ▶ Declaração de sinal
- ▶ Instanciar um módulo
- ▶ Hierarquia de módulos

Declaração de Módulo



Declaração de Processos

palavra-chave do SystemC "função"

`void proc () {`

nome de um processo do módulo

`s = a ^ b;`

`c = a & b;`

`}`

atribuição de sinal

Declaração do Construtor

```
SC_CTOR(half_add) {  
    SC_METHOD(proc) ; sensitive << a << b;  
};
```

nome do módulo

nome de função do módulo

palavra-chave do SystemC
"reage a eventos em"

nome de sinal

Declaração de Sinal

`sc_signal<bool> sa, sb;`

palavra-chave

palavra-chave
"tipo booleano"

nome de sinal

Instanciação e Amarração

-Instanciação

```
half_add half_add_i ("half_add_i");
```

nome do módulo nome da instância apelido da instância

-Amarração (binding)

```
half_add_i.a(sa);
```

nome da instância nome da porta de entrada/saída da instância nome do sinal

Tipos de Dados

C++ built in data types may be used

`long, int, short, char, unsigned long,
unsigned int, unsigned short,
unsigned char, float, double, long double, and bool.`

SystemC provides other types that are needed.

Scalar types: `sc_bit, sc_logic`

Integer types: `sc_int, sc_uint, sc_bigint, sc_biguint`

Bit and logic vector types: `sc_bv, sc_lv`

Fixed point: `sc_fixed, sc_ufixed, sc_fix, sc_ufix`

Tipos de Dados

Fastest



Slowest

Important to use right data types in right place for best performance

- Use native C++ types as much as possible
- Use `sc_int` or `sc_uint`
 - Less than 64 bits wide
 - Two value logic
 - Boolean and arithmetic operations on integers
- Use `sc_bit`, `sc_bv`
 - Boolean operations.
- Use `sc_logic`, `sc_lv`
 - tri-state ports, signals & logic.
 - Convert to appropriate type for computation.
- Use `sc_bigint` or `sc_biguint`
 - More than 64 bits wide
- Use `sc_fixed`, `sc_fix` or `sc_ufixed`, `sc_ufix`
 - Fixed-point arithmetic
 - Convert to `sc_bv` for large number of boolean operations.

Operadores p/ Tipos de Dados

List of all operations SystemC allows on types										
Types	Bitwise	Arithmetic	Logical	Equality	Relational	Assignment	Auto-increment or decrement	Arithmetic if	Concatenation	Index
sc_bit	~ & ^			all		=. &=, =, ^=			yes	
sc_logic	~ & ^			all		=. &=, =, ^=			yes	
sc_int	all	all	all	all	all	all	all	yes	yes	yes
sc_uint	all	all	all	all	all	all	all	yes	yes	yes
sc_bigint	all	all	all	all	all	all	all	yes		yes
sc_biguint	all	all	all	all	all	all	all	yes		yes
sc_bv sc_lv	~ & ^			all		=. &=, =, ^=			yes	yes

Funções p/ Tipos de Dados

Summary of Methods and Functions on types								
Types	Methods					Functions		
Types	range	to_ signed	to_ unsigned	to_ string	to_ double	and_ reduce	or_ reduce	xor_ reduce
sc_int	Yes		Yes	Yes	Yes			
sc_uint	Yes	Yes		Yes	Yes			
sc_bv	Yes	Yes	Yes	Yes		Yes	Yes	Yes
sc_lv	Yes	Yes	Yes	Yes		Yes	Yes	Yes

- **and_reduce()** = e lógico bit a bit
- **or_reduce()** = ou lógico bit a bit
- **xor_reduce()** = ou exclusivo bit a bit

Usando Sinais

```
SC_MODULE ( module_name) {  
    // ports  
    sc_in<int> a;  
    sc_out<int> b;  
    int c;  
    void entry() {  
        b.write(10); // write 10 to the port  
        if (a.read() < 5) { // Read the port  
            c = a.read(); // Read the port  
            again  
        }  
    }  
    // rest of module
```

```
read(), write(), event(),  
posedge(), negedge()
```

```
sc_in<bool> day;  
bool bed;  
  
if (day.posedge())  
    bed = 0;  
else  
    if (day.negedge())  
        bed = 1;
```

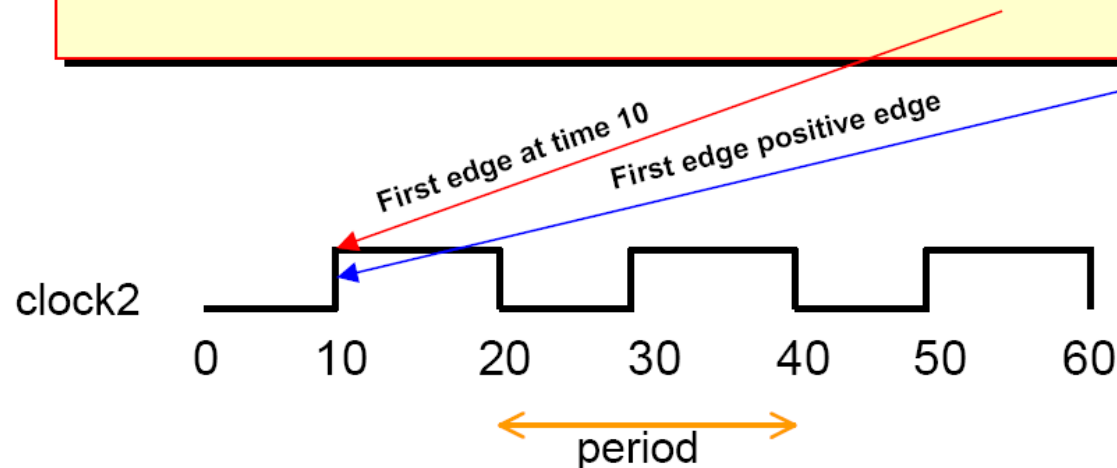
Sinais de Clock

```
sc_clock clock_name ("name", period, duty_cycle, start_time,  
positive_first );
```

name:	<i>name</i>	type: char *	default value: none
period:	<i>clock period</i>	type: double	default value: 1
duty_cycle:	<i>clock duty cycle</i>	type: double	default value: 0.5
start_time:	<i>time of first edge</i>	type: double	default value: 0
positive_first:	<i>first edge positive</i>	type: bool	default value: true

Example:

```
sc_clock clk2("clk2", 20.0, 0.5, 10, true);
```



Processos

- ▶ Um processo é a unidade funcional básica de SystemC
- ▶ Apenas os métodos registrados junto ao Kernel são tratados como processos SystemC
- ▶ SystemC tem dois tipos básicos de processos:
 - ▶ SC_METHOD
 - ▶ SC_THREAD

Processos

- ▶ **SC_METHOD:**

- ▶ não mantém um estado interno: quando ativado, executa do início ao fim e retornam o controle para o mecanismo de chamada
- ▶ processo mais rápido
- ▶ são recomendados para síntese

- ▶ **SC_THREAD:**

- ▶ pode ser suspenso pela chamada de wait() ou suas variantes
- ▶ pode suspender ela mesma e continuar a execução mais tarde do ponto onde parou
- ▶ Tem sua própria thread de operação
- ▶ mais lentos que SC_METHOD
- ▶ mais utilizados para simulação em níveis mais abstratos

SC_METHOD

```
void compute () {  
    s = a ^ b;  
    c = a & b;  
}
```

```
SC_CTOR(half_add) {  
    SC_METHOD(compute) ;  
    sensitive << a;  
    sensitive_pos << b;  
}
```

SC_THREAD

```
void stimgen::stim_proc()
{
    int tmp;
    while (true) {
        tmp = seed + 1;
        a1.write() = tmp;
        a2.write() = tmp + 5;
        seed = (seed + 19) % 123;
        wait( );
    }
}
```

Entidade Topo - SC_MAIN

```
int sc_main()
{
    sc_signal<bool> a,b,s,c;
    half_add half_inst("Adder");
    half_inst.a(a);
    half_inst.b(b);
    half_inst.c(c);
    half_inst.s(s);

    sc_start(500000, SC_NS);
    return 0;
}
```


Eventos

- ▶ Elemento básico de sincronização:

- ▶ Criação
- ▶ Notificação
- ▶ Sensitividade

```
sc_event my_event; // create an event called my_event
```

- ▶ Immediate

```
my_event.notify(); // immediate
```

- ▶ Delayed

```
my_event.notify(SC_ZERO_TIME); //delta cycle delay
```

- ▶ Sensitive to one event in list

```
wait(ev1 | ev2 | ...);
```

- ▶ Sensitive to all events in list

```
wait(ev1 & ev2 & ...);
```

- ▶ Wait for timed event

```
wait(100, SC_NS); // wait 100ns
```

```
wait(100, SC_MS, ev1 | ev2 );
```

```
// wait on events with timeout of 100ms;
```

Comunicação: interfaces, canais, portas

- ▶ São estruturas de comunicação entre módulos:
- ▶ **Interface:**
 - ▶ Define um conjunto de métodos
 - ▶ Não implementa os métodos
 - ▶ Ex: `read()`, `write()`
- ▶ **Canais:**
 - ▶ Implementa os métodos da interface
 - ▶ Container com funcionalidades de comunicação
 - ▶ Ex: `sc_signal<T>`
- ▶ **Portas**
 - ▶ Objeto pelo qual módulos podem acessar a interface de um canal
 - ▶ Definido como um tipo de interface
 - ▶ Ex: `sc_in<T>`, `sc_out<T>`, `sc_inout<T>`

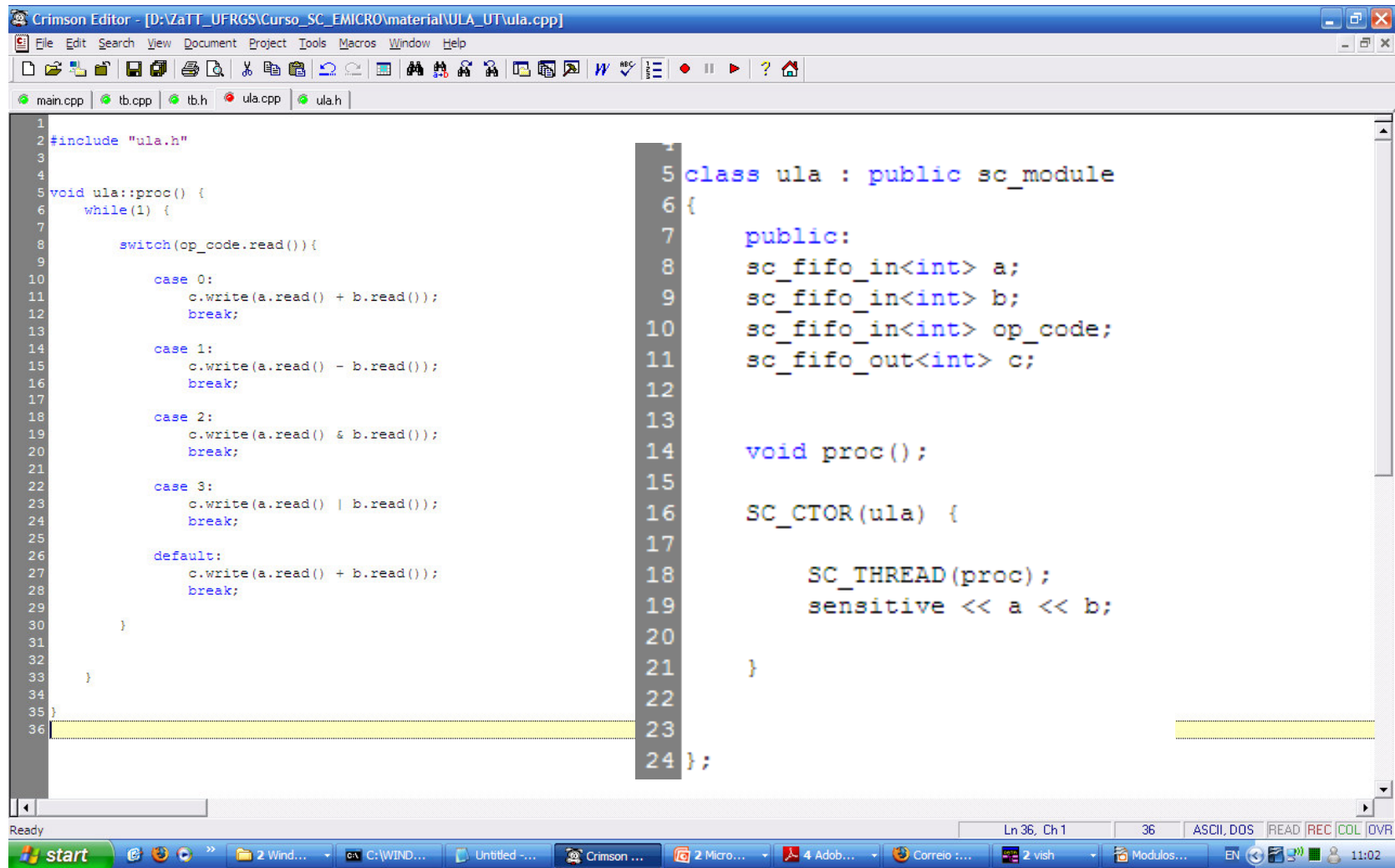
Modelos de Abstração do SystemC

- ▶ **UTF**
 - ▶ Untimed Functional Level
- ▶ **TF**
 - ▶ Timed Functional Level
- ▶ **BCA**
 - ▶ Bus Cycle Accurate Level
- ▶ **RT**
 - ▶ Register Transfer Level

UTF – Untimed Functional Level

- ▶ Especificação
 - ▶ Arquitetura
 - ▶ Componentes
- ▶ Modela comportamento algoritmo
- ▶ Utiliza uma forma “sequencial” do comportamento (sem tempo)
- ▶ Processo executa em tempo zero
- ▶ Comunicação por canais abstratos

UTF – Untimed Functional Level



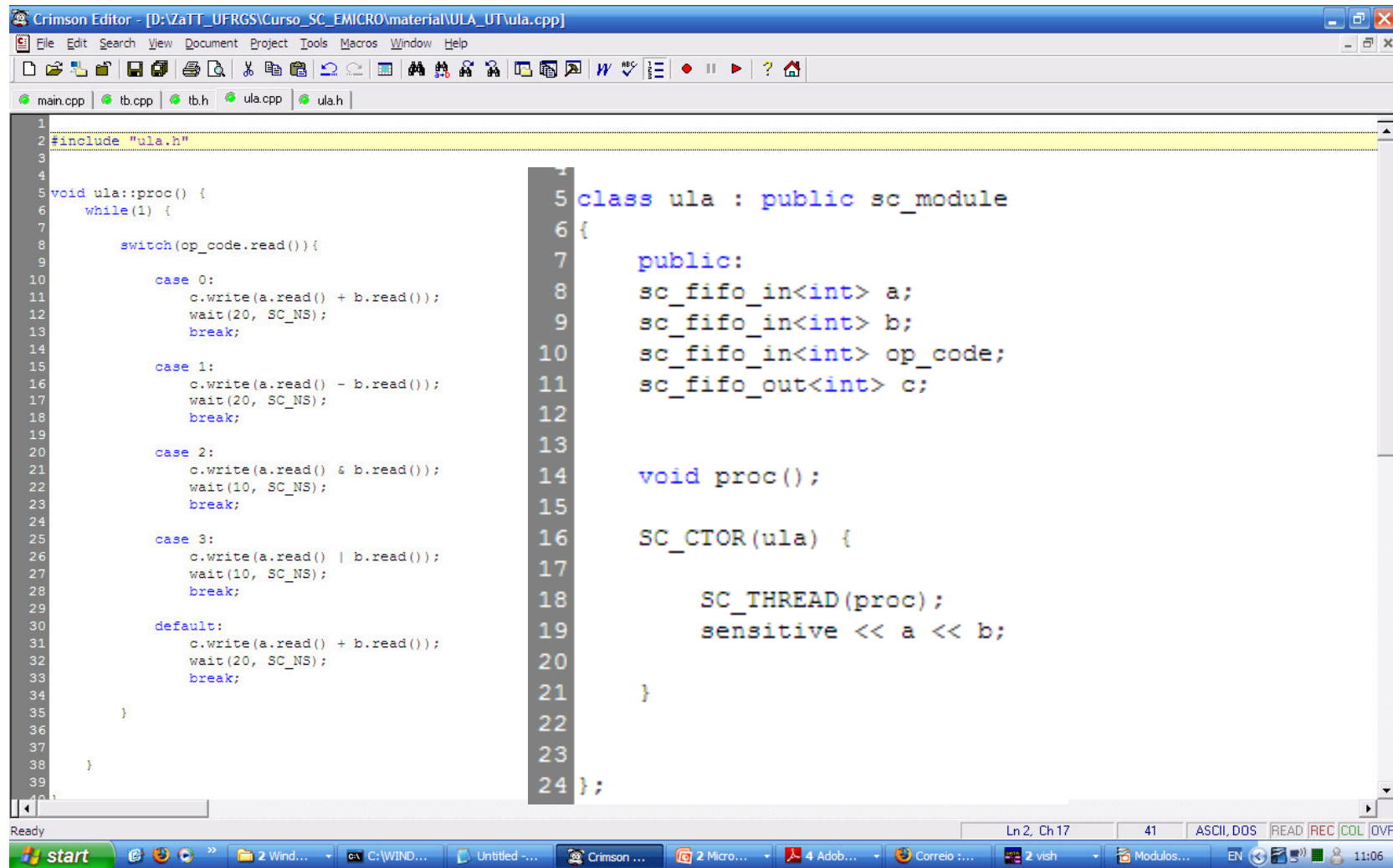
```
1 #include "ula.h"
2
3
4
5 void ula::proc() {
6     while(1) {
7
8         switch(op_code.read()) {
9
10             case 0:
11                 c.write(a.read() + b.read());
12                 break;
13
14             case 1:
15                 c.write(a.read() - b.read());
16                 break;
17
18             case 2:
19                 c.write(a.read() & b.read());
20                 break;
21
22             case 3:
23                 c.write(a.read() | b.read());
24                 break;
25
26             default:
27                 c.write(a.read() + b.read());
28                 break;
29
30         }
31     }
32 }
33
34
35
36
```

```
1
2
3
4
5 class ula : public sc_module
6 {
7     public:
8         sc_fifo_in<int> a;
9         sc_fifo_in<int> b;
10        sc_fifo_in<int> op_code;
11        sc_fifo_out<int> c;
12
13
14        void proc();
15
16        SC_CTOR(ula) {
17
18            SC_THREAD(proc);
19            sensitive << a << b;
20
21        }
22
23
24};
```

TF – Timed Functional Level

- ▶ Insere tempo ao modelo UTF
- ▶ Clock pode ser utilizado para ter noção de duração, não sincronização.

TF – Timed Functional Level



```
1
2#include "ula.h"
3
4
5void ula::proc() {
6    while(1) {
7
8        switch(op_code.read()) {
9
10           case 0:
11               c.write(a.read() + b.read());
12               wait(20, SC_NS);
13               break;
14
15           case 1:
16               c.write(a.read() - b.read());
17               wait(20, SC_NS);
18               break;
19
20           case 2:
21               c.write(a.read() & b.read());
22               wait(10, SC_NS);
23               break;
24
25           case 3:
26               c.write(a.read() | b.read());
27               wait(10, SC_NS);
28               break;
29
30           default:
31               c.write(a.read() + b.read());
32               wait(20, SC_NS);
33               break;
34
35       }
36
37   }
38
39 }
40
```

```
1
2
3
4
5class ula : public sc_module
6{
7    public:
8        sc_fifo_in<int> a;
9        sc_fifo_in<int> b;
10        sc_fifo_in<int> op_code;
11        sc_fifo_out<int> c;
12
13
14    void proc();
15
16    SC_CTOR(ula) {
17
18        SC_THREAD(proc);
19        sensitive << a << b;
20
21    }
22
23
24};
```

BCA – Bus Cycle Accurate Level

- ▶ Comportamento preciso
- ▶ Precisão de ciclo nas interfaces dos módulos
- ▶ Canais abstratos refinados
- ▶ Funcionalidade do sistema deve ser descrita utilizando UTF, TF ou BCA
- ▶ Latência modelada
- ▶ Internamente sem precisão de ciclo
- ▶ Clock utilizado para sincronização

BCA – Bus Cycle Accurate Level

```
1
2
3
4
5 class ula : public sc_module
6 {
7     public:
8         sc_in<int> a;
9         sc_in<int> b;
10        sc_in<int> op_code;
11        sc_out<int> c;
12        sc_in<bool> clock;
13
14        void proc();
15
16        SC_CTOR(ula) {
17
18            SC_THREAD(proc);
19            sensitive << a << b;
20
21        }
22
23
24 };
25
```

```
5 void ula::proc() {
6     while(1) {
7
8         switch(op_code.read()) {
9
10            case 0:
11                c.write(a.read() + b.read());
12                wait(20, SC_NS);
13                break;
14
15            case 1:
16                c.write(a.read() - b.read());
17                wait(20, SC_NS);
18                break;
19
20            case 2:
21                c.write(a.read() & b.read());
22                wait(10, SC_NS);
23                break;
24
25            case 3:
26                c.write(a.read() | b.read());
27                wait(10, SC_NS);
28                break;
29
30            default:
31                c.write(a.read() + b.read());
32                wait(20, SC_NS);
33                break;
34
35        }
36
37    }
38 }
39
```

RT – Register Transfer Level

- ▶ Descrição funcional completa do sistema
- ▶ Utilizada por ferramentas de síntese
- ▶ Completamente orientada pelo clock
- ▶ Precisão de ciclo tanto interna como externamente

RT – Register Transfer Level

```
5
6 SC_MODULE (nc_index) {
7     sc_in<sc_uint<4> > data;
8     sc_in<sc_uint<2> > index;
9     sc_out<bool> dout;
10
11     void proc();
12
13     SC_CTOR (nc_index) {
14         SC_METHOD (proc);
15         sensitive << data << index;
16     }
17 };
18
```

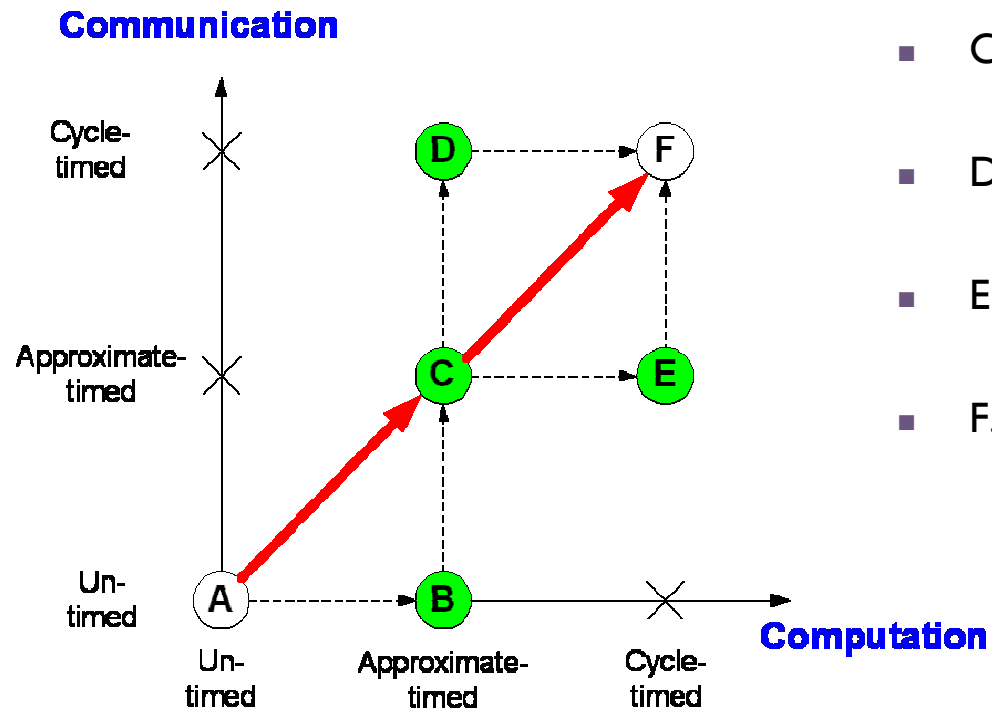
```
1
2
3
4
5 void nc_index::proc() {
6     sc_uint<DSIZE> dt;
7     sc_uint<ISIZE> it;
8
9     dt = data.read();
10    it = index.read();
11    dout = dt[it];
12 };
13
```

TLM – Transaction Level Modeling

- ▶ Separa computação de comunicação
- ▶ Comunica através de transações
- ▶ Em SystemC, transações são definidas como métodos declarados em interfaces e implementados nos canais de comunicação
- ▶ Abstrai-se o *handshaking* detalhado dos sinais, sincronizando-se as operações através de operações de E/S bloqueantes e não-bloqueantes

TLM - Níveis de Abstração

- A. Specification Model
- B. Component Assembly Model
- C. Bus-Arbitration Model
- D. Bus-functional Model
- E. Cycle Accurate computation Model
- F. Implementation Model



* CAI, L., GAJSKI, D. **Transaction Level Modeling: An Overview**

Descrição de hardware em SystemC

Gustavo Girão
ggbsilva@inf.ufrgs.br