Hardware / Software Codesign

SystemC

Florian Eibensteiner

Embedded Systems Design
FH Hagenberg ESD

2019

R 4319

SystemC

Standards:

- IEEE Std. 1666 Standard System C Language Reference Manual
- aktuell: SystemC 2.3.3 inklusive TLM (www.accellera.org)
- Erweiterung zu SystemC: SCV 2.0.1, SystemC-AMS 2.0, OVL 2.8.1, SystemC Synthesis 1.4.7

Bücher:

- T. Grötker, S. Liao, G. Martin, S. Swan: *System Design with SystemC*
- D. C. Black, J. Donovan: *SystemC: From The Ground up*
- F. Ghenassia: *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*

Inhalt

Basisdatentypen in SystemC

`sc_bit`	zweiwertiges Bit (0 und 1)
`sc_logic`	vierwertiges Bit (0, 1, Z, X)
`sc_int<n>`	Integertyp mit fester Bitlänge n ($n \leq 64$)
`sc_uint<n>`	Integertyp mit fester Bitlänge n ($n \leq 64$) ohne Vorzeichen
`sc_bigint<n>`	Integertyp mit fester Bitlänge n ($n \geq 64$)
`sc_biguint<n>`	Integertyp mit fester Bitlänge n ($n \geq 64$) ohne Vorzeichen
`sc_bv<n>`	Bitvektor der Länge n (zweiwertig)
`sc_lv<n>`	Bitvektor der Länge n (vierwertig)

Basisdatentypen in SystemC

`sc_fix<WL, IWL, [...]>`	Festkommatyp der Länge WL mit IWL Vorkommastellen
`sc_ufix<WL, IWL, [...]>`	Festkommatyp der Länge WL mit IWL Vorkommastellen ohne Vorzeichen
suffix `ed`	Parameter sind statisch (compile time)
suffix `_fast`	Genauigkeit auf 53 bits begrenzt (intern ein C++ double)

z.B.:

```
...
#define SC_INCLUDE_FX // Einbinden der noetigen Ressourcen
#include <systemc.h>
...
sc_fixed<5,3> x; //     -4,75 to 3,75 in steps of 0,25
sc_fixed_fast<7,1> y; // |0,984375| in steps of 0,015625
sc_fix z(13,9);
```

C++ Datentypen in SystemC

Weitere Typen:

- alle C++ Datentypen
- benutzerdefinierte Datentypen

Guideline

Immer Typen verwenden, die den nativen C++ Typen (int, bool, double) am ähnlichsten sind! $\rightarrow$ Simulationsgeschwindigkeit!

Module

- Grundlegende Bausteine in SystemC
- Vergleichbar mit Entity-Architecture in *VHDL* oder Modulen in *Verilog*
- Grundsätzlich eine C++ Klassendefinition

```
#include <systemc.h>

SC_MODULE(module_name){
 // Module Body
};
```

Definition:

```
#define SC_MODULE(module_name) \
    struct module_name : public sc_module
```

- Beachte, bei Definition als `struct`, sind Members standardmäßig *public*!

Module

Module Body:

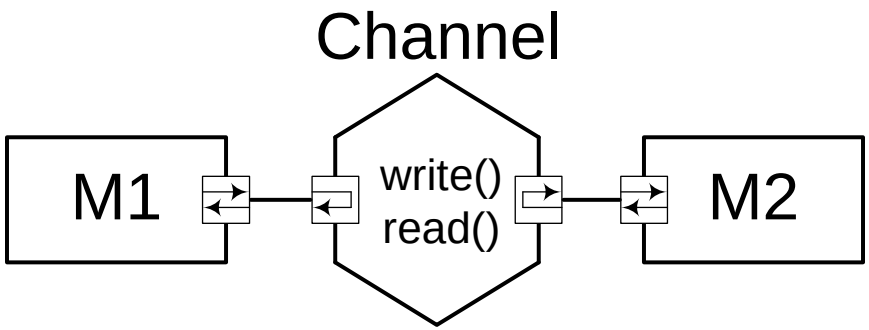
- Port definitions
- Signal definitions
- Variable definitions
- Modul-Instanzen (Sub-Module)
- Construtor bzw. Destructor
- Prozesse (eigentliche Funktionalität)
- Member-Funktionen

Ports

- Kommunikationsschnittstellen zwischen Modulen
- 3 elementare Ports (*"signal ports"*)

`sc_in<type>`	Eingang
`sc_out<type>`	Ausgang
`sc_inout<type>`	Bidirektional

- Komplexe Verbindungen mit `sc_port<channel>`
  - FIFOs
  - Busse
  - ...



Ports - Beispiel

Einfache und komplexe Ports:

```
#include <systemc.h>

SC_MODULE(GPIO){
    sc_in<bool> iClk;
    sc_in<bool> inResetAsync;
    sc_port<simple_bus_slave_if> ioSlavePort;
    sc_out<sc_int<8> > oData;
    ...
};
```

```
    ...
    sc_signal<bool> nrst;
    sc_clock clk("clk", 10, SC_NS, 0.5, 1, SC_NS);
    ...
    GPIO pio("gpio");
    pio.iClk(clk);
    pio.inResetAsync(nrst);
    ...
```

Signale und Variablen

Signale:

- „Leitungen“ zum Verbinde von Prozessen und Modulen
- `sc_signal<int> MySignal; //32 bit`
- Zugriff:

```
MySignal.write(8);
int i = MySignal.read(); // i != 8 -> signals have drivers!
```

Variablen:

- Verwenden als Speicher oder Register
- Definition innerhalb von Modulen
- `sc_uint<8> MyRegister; //8 bit`

Konstruktor

Konstruktor: `SC_CTOR(module_name)`

- Deklaration, Initialisierung und verbinden von Submodulen
- Definition und Registrieren (im SystemC Kernel) von Prozessen
- Festlegen der statischen Sensitivität

Alternativ Konstruktor: `SC_HAS_PROCESS(module_name)`

- Ermöglicht zusätzliche Parametrisierung des Konstruktors (mehr als Modulname)
- Aufteilung auf Header und Source Files
- Notwendig bei Hierarchien (ableiten von Klassen)

Konstruktor - Beispiele

Konstruktor: `SC_CTOR(module_name)`

```
...
SC_MODULE(module_name){
    ...
    SC_CTOR(module_name)
        : Initialization    //    Optional
    {
        // CTOR BODY
    }
    ...
};
```

Alternativ Konstruktor: `SC_HAS_PROCESS(module_name)`

```
...
SC_MODULE(module_name){
    ...
    SC_HAS_PROCESS(module_name);

    module_name(sc_module_name name [, other_args ...])
        : sc_module(name)
        [, other_init ...]{
        // CTOR BODY
    }
    ...
};
```

Prozesse

Es gibt 3 Arten von Prozessen:

- `SC_METHOD(process_name)`
  - Kann auf Signale sensitiv sein (statisch)
  - Wird bei Signaländerung komplett ausgeführt (kein `wait()`!)
- `SC_THREAD(process_name)`
  - Kann auf Signale sensitiv sein (statisch)
  - Abarbeitung kann durch `wait()` unterbrochen werden (dyn. Sensitivität)
  - Thread wird nur einmal aufgerufen und ausgeführt
  - Beschreibung von Zustandsmaschinen oder Testbenches
- `SC_CTHREAD(NAME_cthread, clock_name.edge())`
  - Sensitiv auf Flanken eines Clock-Signales
  - Unterbrechbar mit `wait()`
  - geeignet für Behavioral-Synthese

Inhalt

Cycle Accurate DFF – dff.h

```
#ifndef DFF_H
#define DFF_H

#include <systemc.h>

SC_MODULE(dff){
    // port definition
    sc_in<bool> iClk;
    sc_in<bool> inResetAsync;
    sc_in<bool> iData;
    sc_out<bool> oData;

    SC_HAS_PROCESS(dff);
    //CTOR
    dff(sc_module_name name);

    // process funktion
    void flipflop();
};

#endif
```

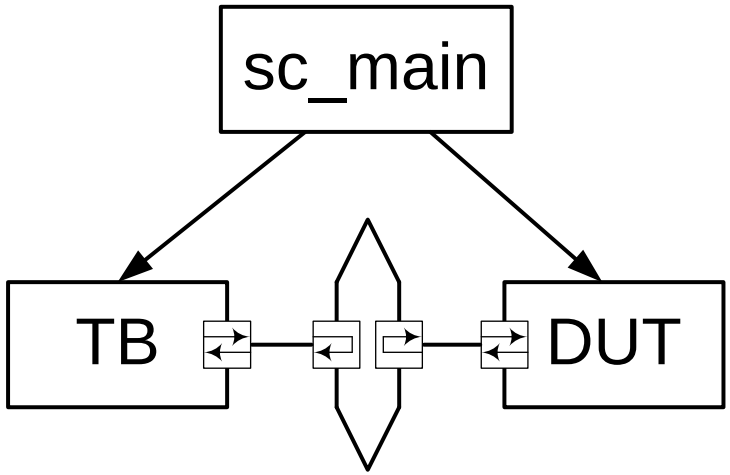
Cycle Accurate DFF – dff.cpp

```
#include "dff.h"

// CTOR - init sc_module name because we use SC_HAS_PROCESS
dff::dff(sc_module_name name)
    : sc_module(name){
    SC_METHOD(flipflop);
      // sensitivity list - pos clock edge and neg rst edge
      sensitive << iClk.pos() << inResetAsync.neg();
      // do not init module at simulation startup
      dont_initialize();
}

// Process
void dff::flipflop(){
    if(!inResetAsync->read()){
        oData->write(false);
    }else{
        oData->write(iData->read());
    }
}
```

Inhalt

Testbench

- Testbench als Modul implementieren
- Stimuli als `SC_THREAD` implementieren
- DUT außerhalb der Testbench
- Verbinden der Module und starten der Simulation in Hauptroutine



Cycle Accurate DFF – testbench.h

```
#ifndef TESTBENCH_H
#define TESTBENCH_H

#include <systemc.h>

SC_MODULE(testbench){
    // port definition
    sc_in<bool> iClk;
    sc_in<bool> iData;
    sc_out<bool> oData;
    sc_out<bool> onResetAsync;

    // process to generate reset
    void RstGen();

    // process to generate stimulus
    void Stimuli();

    // CTOR
    SC_CTOR(testbench){
        SC_THREAD(RstGen);

        SC_THREAD(Stimuli);
    }
};

#endif
```

Cycle Accurate DFF – testbench.cpp

```
#include "testbench.h"

void testbench::RstGen(){
    // set reset to true
    onResetAsync->write(true);
    // wait some time and generate reset
    wait(5, SC_NS);
    onResetAsync->write(false);
    wait(20, SC_NS);
    onResetAsync->write(true);
}

void testbench::Stimuli(){
    // wait for reset
    wait(onResetAsync.negedge_event());
    wait(onResetAsync.posedge_event());

    // set data to high
    oData->write(true);

    wait(iClk.negedge_event());
    // check data
    if(oData->read() && !iData->read()){
        std::cout << "data should be high!" << sc_time_stamp() << std::endl;
    }
    ...
}
```

Inhalt

Simulation

- Starten der Simulation mit `sc_start()`
- Initialisierung: jeder Process kommt in den Ready-Pool
- Simulation-Kernel kümmert sich um Delta-Zyklen, Signal updates Eventmanagement, Timeouts, usw.
- Kooperatives Scheduling $\rightarrow$ *"Concurrency"*
- Geschwindigkeit stark von Modellierungsebene und verwendeten Datentypen abhängig

Hauptroutine

```
int sc_main(int argc, char * argv[]){
    // signal declaration
    sc_clock clk("clock", 10, SC_NS);
    sc_signal<bool> nResetAsync;
    sc_signal<bool> iData;
    sc_signal<bool> oData;

    //instances
    dff myFF("MyFlipFlop");
    testbench testdff("TestDff");

    //port mapping
    myFF.iClk(clk);
    myFF.inResetAsync(nResetAsync);
    myFF.iData(iData);
    myFF.oData(oData);

    testdff.iClk(clk);
    testdff.onResetAsync(nResetAsync);
    testdff.oData(iData);
    testdff.iData(oData);


    // start simulation
    sc_start(500, SC_NS);

    return 0;
}
```

Trace File

- Erzeugen von Waveform-Dateien (VCD, ASCII, WIF, ISDB)
- Vor der Simulation muss Datei erzeugt und Signale eingehängt werden
- Viewer: z.B.: GTKWAVE, Wave VCD,

```
...
// trace signals
sc_trace_file * my_trace_file;
my_trace_file = sc_create_vcd_trace_file("dff_trace");

sc_trace(my_trace_file, myFF.iClk, "iClk");
sc_trace(my_trace_file, myFF.inResetAsync, "inResetAsync");
sc_trace(my_trace_file, myFF.iData, "iData");
sc_trace(my_trace_file, myFF.oData, "oData");

// start simulation
sc_start(500, SC_NS);

// close trace file
sc_close_vcd_trace_file(my_trace_file);
...
```

Inhalt

Installation von SystemC

- Download der Sourcen von `www.accellera.org` und entpacken
- Öffnen des Projekts im Ordner *mscv10* → Projekt gegebenenfalls konvertieren
- Erzeugen der Bibliothek *SystemC.lib*
- Beispiele sind im Ordner *example* enthalten → Testen der Bibliothek

- SystemC-2.3.3 im Elearning (noch nicht kompiliert)
- Ordner lokal am Rechner ablegen und Bibliothek erzeugen.
- Installationsanleitung siehe *SysC_inst_conf.txt* (Elearning) bzw. *INSTALL* (Ordner *systemC-2.3.3*)

Erstellen einer SystemC–Applikation

1. Start Visual Studio.Auf der Startseite wähle *New Project* und *Win32 Console Project*. Eingabe des Projektnamen und Auswahl eines entsprechenden Speicherplatzes → OK.
2. Auf der *Application Settings* Seite im *Win32 Application Wizard* sicherstellen, dass *Empty project* ausgewählt ist (alle anderen Häkchen deaktivieren). → *Finish* um den Wizard zu schließen.
3. Hinzufügen neuer/existierender C++ Files.
4. Anzeige der *Project's Properties* Seite durch Auswahl von 'Properties...' im Menü *Project*.

Erstellen einer SystemC–Applikation

6. Im C/C++ Tab, unter *Code Generation* setzen der *Runtime Library* auf *Multi-threaded Debug (/MTd)*
7. Im C/C++ Tab, unter *Command Line* hinzufügen des Option */vmg* im Feld *Additional Options:*
   - Used if a pointer to a member of a class is declared before the class is defined.
8. Im Linker Tab, unter *Input* Angabe der Bibliothek *systemc.lib* im Feld *Additional Dependencies*

Setzen der Bibliotheks- und Include-Pfade für alle Projekte (bis VS2017)

- Öffnen eines C++ Projektes
- View → Other Windows → Property Manager
- Im *Property Manager* gibt es in der Regel die Knoten *Debug* und *Release*, diese Expandieren
- Die Einträge *Microsoft.Cpp.Win32.user* auswählen und mit einem rechten Mausklick → *Properties* auswählen
- Im Feld *Common Porperties* → VC++ Directories folgende Einträge hinzufügen:
  - Include Directories:$(SYSTEMC)\..\src
  - Library Directories:$(SYSTEMC)\SystemC\Debug

  - Umgebungsvariable: zB.: SYSTEMC = C:\files\systemc-2.3.3\msvc10