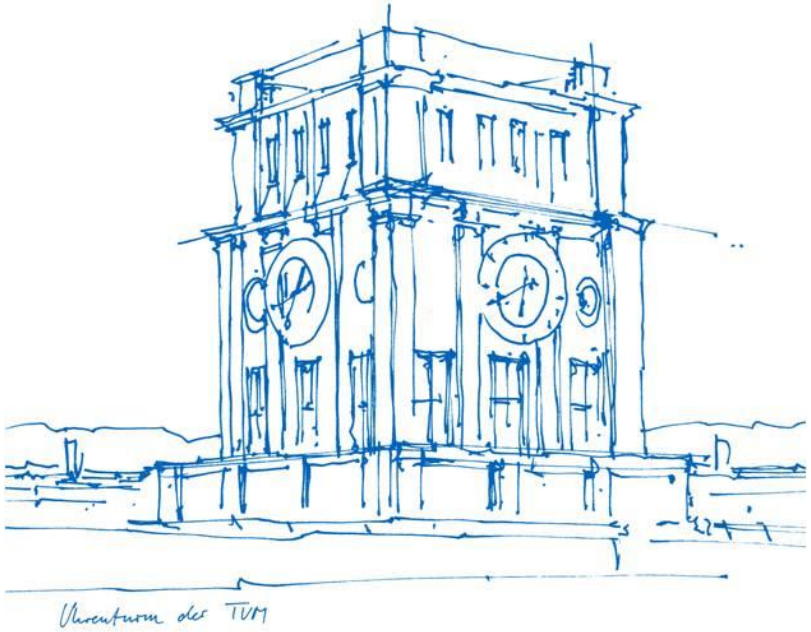
Grundlagenpraktikum: Rechnerarchitektur

SoSe 2025

~ *Danial Arbabi*
danial.arbabi@tum.de



Zulip-Gruppen

GRP 01: Montag 10:00

MI 03.13.10

#GRA/S - Tutorial-GRP-01

GRP 03: Montag 14:00

MI 01.06.20

#GRA/S - Tutorial-GRP-03

Tutoriums-Website

https://home.cit.tum.de/~arb

oder

https://arb.tum.sexy

Disclaimer:
Dies sind keine offiziellen Materialien, somit besteht keine Garantie auf Korrektheit und Vollständigkeit.
Falls euch Fehler auffallen, bitte gerne melden.

Umfrage zum Tutorium

https://forms.gle/ZdUkvGeqPc5sqk477

Viel Erfolg bei euren Projekten!

T 9.1 - SystemC und C

Anleitung zum Projektaufbau

- `main.c`:
  - Commandline Argumente parsen (mit Fehlerbehandlung) und Simulationsfunktion aus `modules.cpp` aufrufen
  - Alle Funktionen, die von SystemC/C++ aufgerufen werden sollen, deklarieren
    `extern <Rückgabetyp> <Funktionsname>(<Parameter>);`

T 9.1 - SystemC und C

Anleitung zum Projektaufbau

- `main.c`:
  - Commandline Argumente parsen (mit Fehlerbehandlung) und Simulationsfunktion aus `modules.cpp` aufrufen
  - Alle Funktionen, die von SystemC/C++ aufgerufen werden sollen, deklarieren
    `extern <Rückgabetyp> <Funktionsname>(<Parameter>);`
- `modules.hpp`:
  - Module definieren (wie wir es bereits kennen)
  - Alle Funktionen, die von C aus verwendet werden sollen, deklarieren:
    `extern "C" <Rückgabetyp> <Funktionsname>(<Parameter>);`

T 9.1 - SystemC und C

Anleitung zum Projektaufbau

- `main.c`:
  - Commandline Argumente parsen (mit Fehlerbehandlung) und Simulationsfunktion aus `modules.cpp` aufrufen
  - Alle Funktionen, die von SystemC/C++ aufgerufen werden sollen, deklarieren
    `extern <Rückgabetyp> <Funktionsname>(<Parameter>);`
- `modules.hpp`:
  - Module definieren (wie wir es bereits kennen)
  - Alle Funktionen, die von C aus verwendet werden sollen, deklarieren:
    `extern "C" <Rückgabetyp> <Funktionsname>(<Parameter>);`
- `modules.cpp`:
  - `sc_main()` mit Fehlercode return hinschreiben (wird nicht aufgerufen)
  - **Simulationsfunktion/Mainfunktion** des Projekts, die in `modules.hpp` deklariert wurde, definieren: Module initialisieren, Signale setzen, Tracefiles erstellen, Simulation starten, return des Ergebnisses, etc.

Commandline Parsing mit `getopt()`

```
int getopt(int argc, char *argv[], const char *optstring);
```

- `argc und argv aus der main-Funktion`
- `const char *optstring:` Alle unterstützten Optionen
  - `:` => Argument **benötigt**
  - `::` => Argument **optional**
  - <nichts> => **Kein** Argument
- Beispiele:
  - `hta:b:c::`
  - `-h` und `–t`
  - `a 1.0` oder `a1.0` und `b5c5` oder `c`

Commandline Parsing mit `getopt()`

```
int getopt(int argc, char *argv[], const char *optstring);
```

- Rückgabewert:
  - Nächster options-Character
  - `-1` bei Ende der Optionen
  - `?:` falls Fehle

Umwandlung von Strings zu Zahlen

- `strtol()`
- `strtoul()`
- `strtoull()`
- `strtof()`
- `strtod()`
- **KEIN** `atoi()` und `atof()` benutzen, da keine Fehlererkennung

Umwandlung von Strings zu Zahlen

`strtol()` und `strtod()`

```
long strtol(const char *ptr, char **endptr, int base),
double strtod(const char *ptr, char **endptr);
```

- `const char *ptr:` zu konvertierender String
- `char **endptr:` Pointer auf einen Pointer, der auf den fehlerhaften Teil des ptr Strings zeigen wird
- `int base`: Zahlenbasis

Umwandlung von Strings zu Zahlen

Beispiel:

```c
int a;  // Zu ändernde Variable
errno = 0;  // Errno für Fehlerbehandlung
char* endptr;   // Endptr für Fehlerbehandlung

a = strtol(currentOpt, &endptr, 10);

if (*endptr != '\0' || errno) {
    fprintf(stderr, "Invalid number\n");
    /* Weitere Fehlerbehandlung bzgl errno Wert*/
    return EXIT_FAILURE;
}
```

File-I/O

fopen

- Öffnen von Files:

```
FILE *fopen(const char *pathname, const char *mode);
```

- `const char *pathname:` Pfad der Datei
- `const char *mode:` Berechtigungen, mit denen die Datei geöffnet wird
  - `O_RDONLY` → « `r` »
  - `O_WRONLY` → « `w` »
  - `O_RDWR` → « `r+ / w+` »
  - … « `a / a+` »
- `fopen` *(man 3 fopen)* verwendet im Hintergrund Systemcall open *(man 2 open)*

`Rückgabe: Pointer auf FILE-Struktur`

File-I/O

fclose

- Schließen von Files:

  `int fclose(FILE *stream);`

- `FILE *stream:` Pointer auf Filestruktur

`Rückgabe: 0` wenn erfolgreich, sonst `EOF` und `errno` gesetzt

File-I/O

fread

- Lesen von Files:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- `void *ptr:` Speicherbereich, in dem Fileinhalt gespeichert werden soll
- `size_t size:` Größe der zu lesenden Items in Bytes
- `size_t nmemb:` Anzahl der zu lesenden Items
- `FILE *stream:` File, aus der gelesen werden soll

`Rückgabe:` Anzahl der gelesenen Items

File-I/O

fwrite

- Schreiben von Files:

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- `Const void *ptr:` Pointer auf einen konstanten Speicherbereich, aus dem gelesen wird
- `size_t size:` Größe der zu schreibenden Items in Bytes
- `size_t nmemb:` Anzahl der zu lesenden Items
- `FILE *stream:` File, die beschrieben wird

`Rückgabe:` Anzahl gelesener Items

File-I/O

fstat

- Erhalten von File Informationen:

  `int fstat(int fd, struct stat *statbuf);`

- `int fd:` Filedeskriptor (bekommt man mit `fileno(FILE*)`)
- `struct stat *statbuf:` Pointer auf `stat` Struktur, worin die Information geschrieben werden soll
- z.B. `statbuf->st_size` für die Dateigröße

`Rückgabe:` 0 wenn erfolgreich, ansonsten -1 (errno wird gesetzt)

Makefile

Was passiert in den beiden Zeilen?

```
main: main.c xor_cipher.c
    $(CC) $(CFLAGS) -o $@ $^
```

- Ein solches Konstrukt heißt „`Rule`“

  `target: prerequisite1 prerequisite2 ...` (prerequisite kann auch ein weiteres Target sein)

  `recipe1`

  `recipe2`

- Quelldateien: `main.c` und `xor_cipher.c`
- Shell-Befehle: `$(CC) $(CFLAGS) -o $@ $^` (kann z.B. auch “`echo Hallo`” sein)
- `Output-Name: main`

Makefile

Womit wird nun $@ und $^ ersetzt?

```
main: main.c xor_cipher.c
    $(CC) $(CFLAGS) -o $@ $^
```

- $@ = The file name of the target of the rule
- $^ = The names of all the prerequisites, with spaces between them

Makefile

Mehr Mals make? + PHONY

Mehrmals `make`:

- `make` verfolgt Dependencies auf Basis der Modfikationszeit
- Ist ein `target` neuer als alle `prerequisites`, muss es nicht erneut gebaut werden

PHONY:

- `.PHONY: <target>` (z.B. .PHONY: clean)
- Konflikt mit Datei mit selbem Namen wie `<target>` vermeiden, die nichts mit dem Target zu tun hat

Module in SystemC

Stichworte: Konstruktor, Sensitivity-Lists, behaviour

```
1  SC_MODULE(M1) {
2      sc_signal<bool> x;
3      sc_signal<bool> y;
4      sc_signal<bool> output;
5
6      SC_CTOR(M1) {
7          SC_THREAD(behaviour);
8          sensitive << x << y;
9      }
10
11     void behaviour() {
12         while (true) {
13             output = x.read() | (!x.read() & y.read());
14             wait();
15         }
16     }
17 };
```

Signale

Konstruktor

update / behaviour

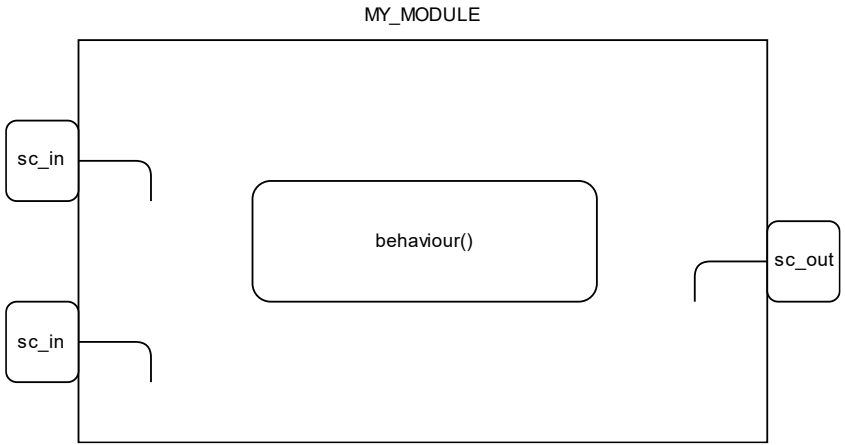
Module in SystemC

Stichworte: Ports (input, output)

```
1  SC_MODULE(MyModule) {
2      sc_in<bool> input;
3      sc_out<bool> output;
4
5      SC_CTOR(MyModule) {
6          SC_THREAD(behaviour);
7      }
8
9      void behaviour() {
10         while (true) {
11             output->write(!input->read());
12             wait();
13         }
14     }
15 };
```

- Lesen vom Input
- Schreiben auf den Output

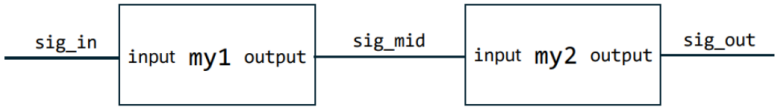
Ports in SystemC



- Innerhalb des Moduls:
  - Lesen und schreiben auf Ports ist möglich
- Außerhalb des Moduls (`sc_main` oder äußeres Modul):
  - Kabel an Port binden
  - Von Kabel lesen / auf Kabel schreiben

Module in SystemC

Kommunikation zwischen Modulen (Kabel verbinden)




```
MyModule my1("my1");
MyModule my2("my2");
sc_signal<bool> sig_in, sig_mid, sig_out;
// Die Tatsächlichen Signale werden außerhalb der Module
    erstellt.

my1.input.bind(sig_in);
my1.output.bind(sig_mid);
// port.bind(signal) weist einem Port ein Signal zu.

my2.input(sig_mid);
my2.output(sig_out);
// Alternative Schreibweise: port(signal).
```

- Schreiben / Lesen von Signalen
- Verbinden dieser Signale mit den Ports

Clocks in SystemC

- Best Practice: Wir erstellen eine Clock in `sc_main` und binden sie an Input Ports in Modulen.

```
1  SC_MODULE(MY_MODULE) {
2      sc_in<bool> clk;
3      SC_CTOR(MY_MODULE) { }
4  };
5
6  int sc_main(int argc, char* argv[]) {
7      sc_clock clk("clk", 2, SC_SEC);
8
9      MY_MODULE my_module("my_module");
10     my_module.clk(clk);
11
12     sc_start(10, SC_SEC);
13     return 0;
14 }
```

Clocks in SystemC

SC_CTHREAD

Nutzen der Clock mit SC_CTHREAD():

```
1 SC_MODULE(MY_MODULE) {
2     sc_in<bool> clk;
3
4     SC_CTOR(MY_MODULE) {
5         SC_CTHREAD(behaviour, clk.pos());
6     }
7
8     void behaviour() { ... }
9 };
```

Clocks in SystemC

SC_CTHREAD

- `SC_CTHREAD(behaviour, clk.pos())`
  - `sc_in.pos()`: Event für steigende Flanke.
  - `sc_in.neg()`: Event für sinkende Flanke.
  - `sc_in.value_changed()`: Event für jeden Flankenwechsel.

SC_CTHREAD beginnt mit
behaviour erst ab erster Flanke
→ wait() am Ende der while(true) Loop

```
1  ... // SC_CTHREAD(behaviour, clk.pos())
2  void behaviour() {
3      while(true) {
4          wait();
5          std::cout << sc_time_stamp() << std::endl;
6      }
7  }
8  // 2 s
9  // 4 s
10 // 6 s
11 // 8 s
```

Clocks in SystemC

SC_THREAD und SC_METHOD

SC_THREAD beginnt sofort
→ wait() am Anfang der while(true) Loop

- SC_METHOD und SC_THREAD bieten Möglichkeiten, die Prozesse bei steigender Flanke eines sc_in<bool> auszuführen.
- SC_METHOD:

```
1 void behaviour() {
2     std::cout << sc_time_stamp() << std::endl;
3     next_trigger(clk.posedge_event());
4 }
```

- SC_THREAD:

```
1 ...
2 SC_THREAD(behaviour);
3 sensitive << clk.pos();
```

Datenspeicherung in SystemC Modulen

```
1  SC_MODULE(MY_STORAGE_MODULE) {
2      sc_in<int> address;
3      sc_in<int> value;
4      sc_in<bool> clk;
5      int storage[256];
6
7      ...
8
9      void behaviour() {
10         while(true) {
11             wait();
12             storage[address->read()] = value->read();
13         }
14     }
15 };
```

Tracefile

P 8.1 Trace File

1. Erstellen der Tracefile (`Pfad bzw. Name vom User über Commandline übergeben`) und zuweisen auf eine Pointervariable `trace_file`

```
sc_trace_file* trace_file = sc_create_vcd_trace_file(argv[1]);
```

2. Beobachten einer Variablen `var` in `trace_file` und nenne sie beliebig (z.B. „var“)

```
sc_trace(trace_file, var, „var“);
```

3. Starten der Simulation für `time` Sekunden

```
sc_start(time, SC_SEC);
```

4. Schließen der `trace_file`

```
sc_close_vcd_trace_file(trace_file);
```

I/O-Analyse

T 8.1 IO Counter

```
1  SC_MODULE(MY_MODULE) {
2      int reads;
3      int writes;
4      ...
5
6      void behaviour() {
7          while (true) {
8              wait();
9              bool result = a->read() ^ b->read();
10             reads += 2;
11             out->write(result);
12             writes++;
13         }
14     }
15 };
```

T 9.1 - SystemC und C

Anleitung zum Projektaufbau

- `main.c`:
  - Commandline Argumente parsen (mit Fehlerbehandlung) und Simulationsfunktion aus `modules.cpp` aufrufen
  - Alle Funktionen, die von SystemC/C++ aufgerufen werden sollen, deklarieren
    `extern <Rückgabetyp> <Funktionsname>(<Parameter>);`
- `modules.hpp`:
  - Module definieren (wie wir es bereits kennen)
  - Alle Funktionen, die von C aus verwendet werden sollen, deklarieren:
    `extern "C" <Rückgabetyp> <Funktionsname>(<Parameter>);`
- `modules.cpp`:
  - `sc_main()` mit Fehlercode return hinschreiben (wird nicht aufgerufen)
  - **Simulationsfunktion/Mainfunktion** des Projekts, die in `modules.hpp` deklariert wurde, definieren: Module initialisieren, Signale setzen, Tracefiles erstellen, Simulation starten, return des Ergebnisses, etc.

Vorgehen bei der SystemC Programmierung

Implementierung eines Moduls

1. Erstellen einer <my_module.hpp> Datei (s. Tutorium)
2. Modul definieren
    1. `sc_in<type>` und `sc_out<type>`
    2. Member Hilfs-Variablen (normale ints, Arrays, Hashmaps, ...>
    3. Clock: `sc_in<bool>`
3. `void behaviour()` definieren
4. Konstruktor definieren
    1. Ggf. Parameterliste (s. Folie später mit Besonderheit bei eigenen Parametern)
    2. Ggf. Setzen von Startwerten von Hilfsvariablen
    3. Neuen Thread starten: `SC_THREAD(behaviour);`
    4. Sensitivity List: `sensitive << clk << …;`

Vorgehen bei der SystemC Programmierung

Starten der Simulation

1. Benötigte SystemC-Module jeweils in einer `.hpp` Headerdatei implementieren
2. `#include <my_module.hpp>` in `.cpp` Datei
3. Einen **Schaltplan** aufzeichnen: Module, Verbindungskabel, Ein-/Ausgabekabel, Clock **benennen**
4. In eigener Simulationsfunktion:
    1. Alle **Module initialisieren** (Konstruktoraufruf)
    2. Alle benötigten **Signale deklarieren**
    3. Alle Signale an die jeweiligen **Modulports binden** (jeder Port muss gebindet sein!)
    4. Inputwerte auf die **Signale schreiben**, die an die Modul-**Inputports** gebindet sind
    5. `sc_start(<time>, <unit>);`
    6. Outputwerte aus den **Signalen lesen**, die an die Modul-**Outputports** gebindet sind
    7. Ggf. wieder zu **Schritt 4**

Konstruktor mit Parametern

T 6.3 RISC-V Multiplexer

- Leeren `SC_CTOR()` Konstruktor hinschreiben
- Erster Parameter vom neuen Konstruktor ist `sc_module_name name`

```
SC_CTOR(MULTIPLEXER_BOOLEAN); // Leerer Konstruktor mit Semicolon

// RICHTIGER KONSTRUKTOR! BENUTZEN!
MULTIPLEXER_BOOLEAN(sc_module_name name, uint8_t fanIn, uint8_t fanOut) : sc_module(name), in(fanIn), out(fanOut) {
    SC_THREAD(behaviour);

    sensitive << select;

    for (size_t i = 0; i < in.size(); i++) {
        sensitive << in[i];
    }
}
```

Ports im Konstruktor

T 7.2 RISC-V Multiplexer

```
SC_MODULE(PC) {
  sc_in<bool> clk;
  sc_out<uint32_t> pc;
  uint32_t pcValue;

  SC_CTOR(PC) {
    pc->write(0x00001000); // GEHT NICHT
    pcValue = 0x00001000;
    SC_THREAD(behaviour);
    sensitive << clk.pos();
  }

  void behaviour() {
    pc->write(pcValue);
    while (true) {
      wait();
      pc->write(pcValue);
    }
  }
};
```

- Ports existieren im Konstruktor NICHT! (Laufzeitfehler)
- Stattdessen normale Hilfsvariable benutzen und diese in `behaviour()` auf den Port schreiben

behaviour() und wait() mit Clock

```
SC_MODULE(MY_MODULE) {
  sc_in<bool> clk;

  SC_CTOR(MY_MODULE) {
    SC_THREAD(behaviour);
    sensitive << clk.pos();
  }

  void behaviour() {
    while(true) {
      wait(); // wait() am Anfang
      /* ... */
    }
  }
};
```

```
SC_MODULE(MY_MODULE) {
  sc_in<bool> clk;

  SC_CTOR(MY_MODULE) {
    SC_CTHREAD(behaviour, clk.pos());
  }

  void behaviour() {
    while(true) {
      /* ... */
      wait(); // wait() am Ende
    }
  }
};
```

- `SC_THREAD()`: `wait()` am Anfang
- `SC_CTHREAD()`: `wait()` am Ende

wait(SC_ZERO_TIME)

T 7.1 D-Flip-Flop

```
SC_MODULE(D_FLIP_FLOP) {

  sc_in<bool> d;
  sc_in<bool> clk;
  sc_out<bool> q, q_bar;

  SC_CTOR(D_FLIP_FLOP) {
    SC_THREAD(update);
    sensitive << clk.pos();
  }

  void update() {
    while (true) {
      wait();
      q->write(d->read());
      wait(SC_ZERO_TIME);
      q_bar->write(!q->read());
    }
  }
};
```

- Wenn sicher gestellt werden soll, dass ein Signal propagiert ist, dann `wait(SC_ZERO_TIME)`
- Eventuell öfter hintereinander benötigt, falls Modultiefe groß ist

Umfrage zum Tutorium

https://kurzlinks.de/i513

Viel Erfolg bei euren Projekten!