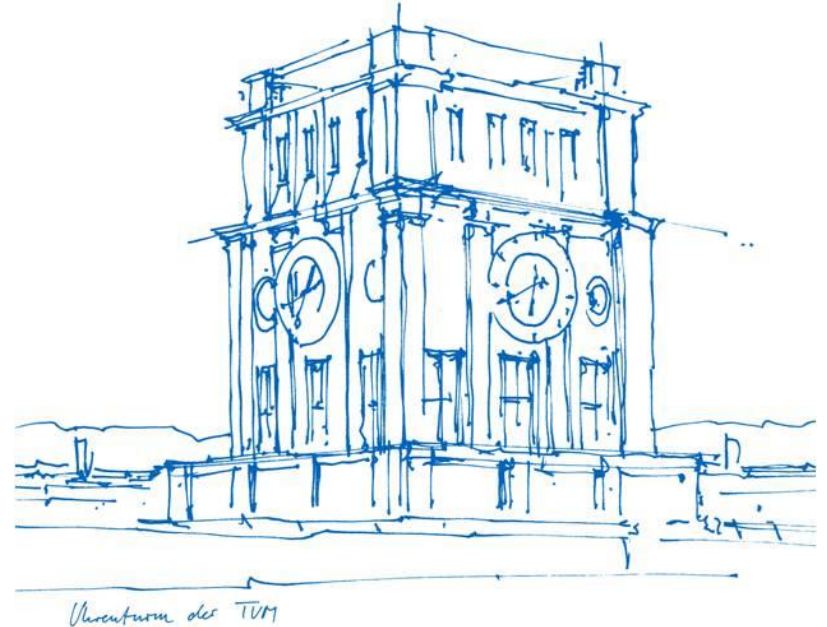


# Grundlagenpraktikum: Rechnerarchitektur

SoSe 2024

~ *Danial Arbabi*

[danial.arbabi@tum.de](mailto:danial.arbabi@tum.de)



# Zulip-Gruppen

Gruppe 29

FR 12:00



<https://zulip.in.tum.de/#narrow/stream/2276-GRA-Tutorium---Gruppe-29>

Gruppe 32

FR 15:00



<https://zulip.in.tum.de/#narrow/stream/2279-GRA-Tutorium---Gruppe-32>

# Tutoriums-Website



<https://home.in.tum.de/~arb/>

## Disclaimer:

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

# Organisatorisches

- Letzte Inhaltswoche
- Woche 9 Fragestunde
- Teamtreffen
- Praktikumsordnung (vllt. Später genaueres)
- Hausaufgaben und Übungen machen
- Fragen JETZT stellen

# Trace File

- ▶ *Trace*: Gibt an, zu welchem Zeitpunkt sich der Wert eines Signals ändert.
- ▶ *Trace File*: Sammlung von Traces verschiedener Signale.

```
1 int sc_main(int argc, char* argv[]) {  
2     sc_signal<bool> my_signal;  
3     sc_signal<int> my_count;  
4     ...  
5  
6     sc_trace_file* file = sc_create_vcd_trace_file("trace");  
7     sc_trace(file, my_signal, "my_signal");  
8     sc_trace(file, my_count, "my_count");  
9     sc_start(10, SC_SEC);  
10    sc_close_vcd_trace_file(file);  
11  
12    ...  
13 }
```

# Tracefile

## Erstellen

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 „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);
```

# Trace File

## Aufbau

- ▶ *Value Change Dump*: Format für Traces, häufig bei Systemdesign Tools.

```
1 $date // Erstellungsdatum des Traces
2     May 31, 2024      18:00:00
3 $end
4
5 $version // Version und Datum von SystemC
6 SystemC 2.3.4-Accellera --- May 11 2024 10:27:04
7 $end
8
9 $timescale // Zeiteinheit aller Zeitangaben im Trace
10     1 ps
11 $end
```

# Trace File

## Aufbau

- *Value Change Dump*: Format für Traces, häufig bei Systemdesign Tools.

```
1 $scope module SystemC $end
2 $var wire    1  aaaaa  my_signal      $end
3   // 1 Bit, Name "my_signal", ID "aaaaa"
4 $var wire   32  aaaab  my_count [31:0]  $end
5   // 32 Bit, Name "my_count", ID "aaaab"
6 $upscope $end
7 $enddefinitions  $end
8
9 $comment
10 All initial values are dumped below at time 0 sec = 0 timescale
    units.
11 $end
```



# Trace File

## Aufbau

- ▶ *Value Change Dump*: Format für Traces, häufig bei Systemdesign Tools.

```
1 $dumpvars           // Startwerte aller Signale
2 1aaaaa              // aaaaa (my_signal) = 1
3 b0 aaaab            // aaaab (my_count) = b0 = 0
4 $end
```

# Trace File

## Aufbau

- *Value Change Dump*: Format für Traces, häufig bei Systemdesign Tools.

```
1 #20000000000000 // Zeitpunkt 2 Sekunden
2 0aaaaa // aaaaa (my_signal) = 0
3 b1 aaaab // aaaab (my_count) = b1 = 1
```

```
1 #40000000000000 // Zeitpunkt 4 Sekunden
2 1aaaaa // aaaaa (my_signal) = 1
3 b10 aaaab // aaaab (my_count) = b10 = 2
```

```
1 #60000000000000 // Zeitpunkt 6 Sekunden
2 b11 aaaab // aaaab (my_count) = b11 = 3
3 // my_signal hat sich nicht verändert
```

# Trace File

## Bedenken

- ▶ Performance-Analyse durch Messung von Delays: genau genug?
- ▶ Es gibt weiterhin "real-world" Faktoren, die Performance beeinflussen können
  - ▶ *Signal I/O*: Jeder *read/write* benötigt gewisse Zeit.
  - ▶ *Taktfrequenz*: Taktfrequenzen von 1 ps sind selten möglich. Overhead kann uns zwingen, auf tiefere Frequenzen auszuweichen.
  - ▶ *Abstraktion & Rechenzeit*: Manche Operationen sind in C/C++ nur eine Zeile Code, aber im Schaltkreis sehr komplex. Kann viel Overhead bringen.
- ▶ SystemC bietet uns Möglichkeit zur Abstraktion von Systemdesign.
  - ▶ Hilft uns, Schaltkreise zu entwerfen.
  - ▶ Macht aber genaue Analysen schwieriger.

# Trace File

## I/O-Analyse

```
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};
```

# Trace File

## I/O-Analyse

- ▶ Resultate der Zählung bieten nützliche Statistiken:
  - ▶ Gesamtzahl I/O Operationen
  - ▶ I/O Operationen pro Sekunde
  - ▶ I/O Operationen pro Clock Zyklus
- ▶ Bieten Möglichkeit zum Vergleich verschiedener Lösungen
- ▶ Aber: Wie genau ist Resultat?
  - ▶ Read/Write dauert nicht unbedingt immer gleich lang.
  - ▶ Abstrahierte Teile eines Moduls können in "real-world" auch Reads/Writes beinhalten.
  - ▶ I/O Kosten hängen auch von Signaltyp ab: `sc_signal<bool>` benötigt weniger Writes/Reads als `sc_signal<int>`.
  - ▶ Umgekehrt: 10-mal so viele Writes/Reads bedeutet nicht gleich 10-mal langsamer. I/O kann auch parallel stattfinden.

# T8 - 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 <typ> <funktion>(<parameter>);`

### ■ modules.hpp:

- ☐ Module definieren (wie wir es bereits kennen)
- ☐ Alle Funktionen, die von C aus verwendet werden sollen, deklarieren:  
`extern "C" <typ> <funktion>(<parameter>);`

### ■ modules.cpp:

- ☐ `sc_main()` mit Fehlercode return hinschreiben (wird aber nicht aufgerufen)
- ☐ **Simulationsfunktion/Mainfunktion** des Projekts, die in `modules.hpp` deklariert wurde, definieren:  
Module initialisieren, Signale setze, Tracefiles erstellen, Simulation starten, return des Ergebnisses, etc.

# Umfrage zum Tutorium



<https://kurzelinks.de/em4f>