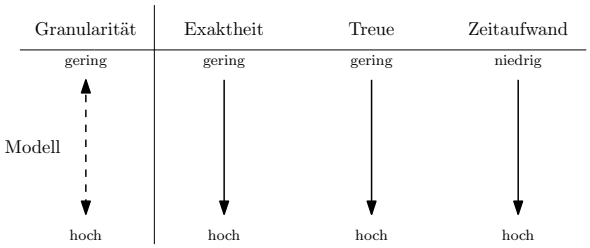
FH-OÖ Hagenberg/ESD
Hardware / Software Codesign, SS 2019
Florian Eibensteiner © 2019 (R 4698)

2. Übung: Different Levels of Abstraction

Name(n):

1 Abstraktion

Wie bereits aus der Vorlesung bekannt ist, spielt die Wahl der Abstraktionsebene bei der Modellierung von Systemen eine entscheidende Rolle. Auf der einen Seite soll ein Modell die Wirklichkeit möglichst gut beschreiben und auf der anderen Seite sollen Modelle auch eine möglichst schnelle Simulation ermöglichen. Darüber hinaus sollen Modelle Abschätzungen über das finale System erlauben. Leider schließen sich diese Paradigmen gegenseitig aus.



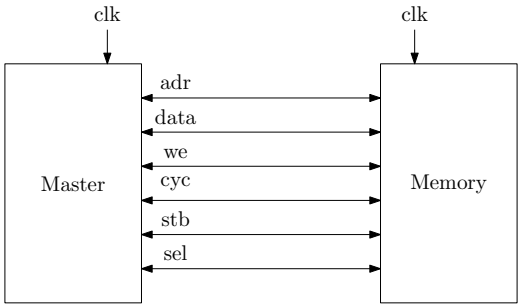
1.1 Gegenüberstellung von 3 Abstraktionsebenen

Sie sollen nun die 3 Abstraktionsebenen, *loosely timed* (LT), *approximately timed* (AT) und *cycle callable* (CC) gegenüberstellen und nach folgenden Punkten bewerten:

- Lines of Code (nur eigener Source Code),
- Ausführungszeit,
- Anzahl der benötigten Deltazyklen (*sc_delta_count()*),
- Abbildung der Wirklichkeit.

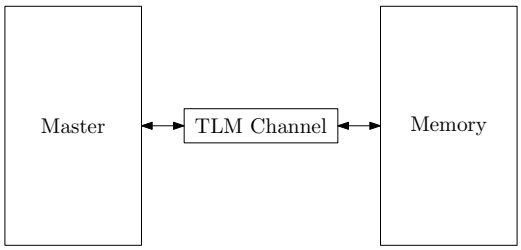
Als Beispiel soll ein System bestehend aus einem Master und einem Slave (Memory, 4096x32 Bit) implementiert werden, die über einen Bus (Wishbone) miteinander kommunizieren. Falls der Wishbone Bus und dessen Protokoll nicht bereits aus der Übung *AMV* bekannt sein, orientieren Sie sich an dessen Spezifikation [Ope10], wobei die *Data Tags* nicht implementiert werden müssen. Der Speicher soll bei jedem Zugriff eine zufällige Anzahl von Waitstates ausführen bevor er auf die Anfrage des Masters reagiert, wobei mindestens 1 und maximal 10 Waitstates ausgeführt werden sollen.

Erstellen Sie zuerst ein zyklengenaues Modell (CC) dieses Systems, indem sie zwei Module in SystemC implementieren und mit den entsprechenden Signalen verbinden. Dabei greift der Master auf den Speicher zu, indem er die entsprechenden Signale setzt und das Timing zyklengenau einhält. Das Slave-Modul arbeitet ebenso zyklengenau, d.h. die Eingabesignale werden bei einer steigenden Flanke des Taktsignals verarbeitet und die entsprechenden Ausgänge ebenso bei steigender Taktflanke gesetzt. Sie können sich an dem *BFM* des Wishbone Busses aus *AMV* orientieren und müssen nur einfache Lese- und Schreibzugriffe durchführen. Demnach implementiert der Master in einem Prozess die geforderten Testfälle und realisiert über die Funktionen `singleRead(...)` und `singleWrite(...)` den eigentlichen Buszugriff. Dadurch kann der Code für die Testfälle in den verschiedenen Abstraktionsmodellen wieder verwendet werden und es müssen nur die beiden Funktionen `singleRead(...)` und `singleWrite(...)`, entsprechend der Abstraktionsschicht, angepasst werden. Auf diese Weise können die einzelnen Modelle hinsichtlich *Lines of Code* leichter verglichen werden.

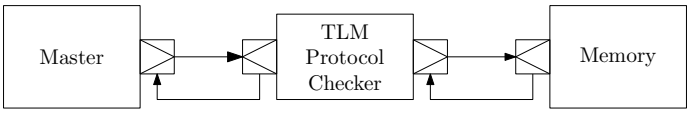


Für die Implementierung des LT-Modells verwenden Sie das TLM-Framework aus der Übung. Modellieren Sie dabei den Bus als TLM-Kanal und verwenden Sie für das LT-Modell die blockierende Funktion *b_transport()*. Es sollen folgende zwei Varianten implementiert werden:

- pro Zugriff soll eine zufällige Anzahl von Waitstates gewartet werden.
- pro Testfall soll die akkumulierte Anzahl der Waitstates gewartet werden.



Beim AT-Modell sollen ebenfalls TLM verwendet werden, wobei ausschließlich der *Forward-Path* (`nb_transport_fw(...)`) und der *Backward-Path* (`nb_transport_bw(...)`) für die Kommunikation eingesetzt werden sollen. Halten Sie sich dabei an das Beispiel aus der Übung und bilden Sie die Buszugriffe mit dem *TLM Base-Protocol* nach. Um die richtige Kommunikation zwischen Master und Slave sicherzustellen, soll zusätzlich der *TLM Protocol-Checker* von Doulos (www.doulos.com) eingebunden werden, der die Erfüllung des Basis-Protokolls prüft. In folgender Abbildung ist die Einbindung des Protocol-Checkers dargestellt.

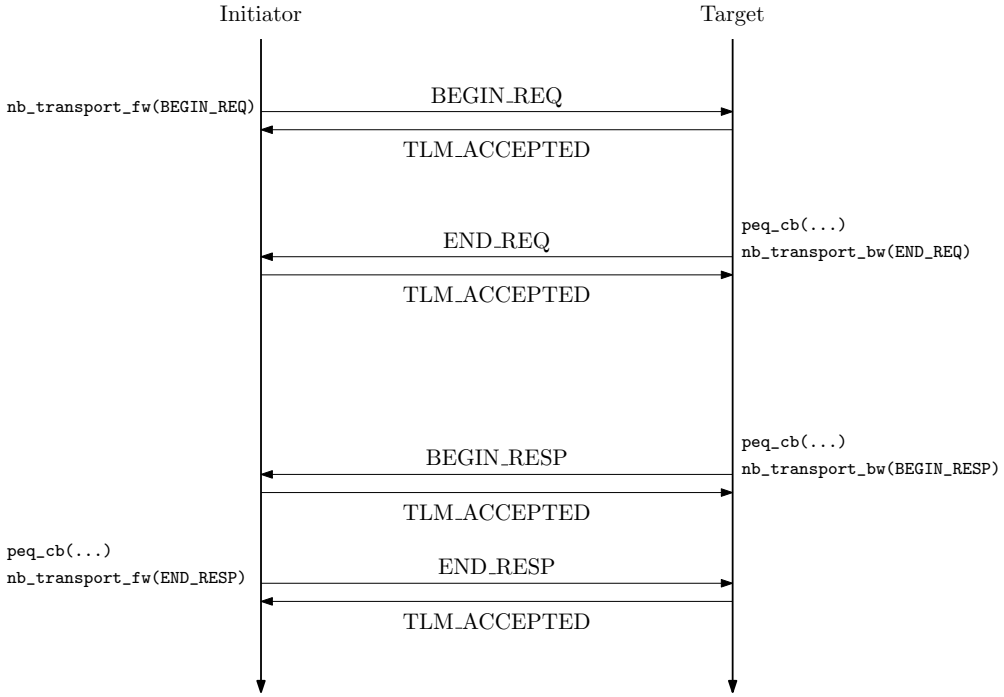


Sie finden den Code auf der Elearning-Plattform und können ihn ohne Veränderung wie folgt im Toplevel-Modul einbinden:

```
1  SC_MODULE(top){
2      Master * initiator;
3      Slave * memory;
4      tlm_utils::tlm2_base_protocol_checker<> * checker;
5
6      SC_CTOR(top){
7          initiator = new Master("initiator");
8          memory = new Slave("memory");
9          checker = new tlm_utils::tlm2_base_protocol_checker<>("chekcer");
10
11         initiator->initiator_socket.bind(checker->target_socket);
12         checker->initiator_socket.bind(memory->target_socket);
13     }
14 };
```

Da der Wishbone Bus keine überlappenden Zugriffe erlaubt, darf immer nur **eine** Transaktion aktiv sein und erst wenn diese vollständig abgearbeitet ist, kann die nächste Transaktion starten. Sie können dieses Verhalten nachbilden, indem Sie beispielsweise am Ende eines Zugriffs auf ein Event warten, welches gesetzt wird, wenn der Master die Phase *End_Resp* schickt.

Der Ablauf der Kommunikation soll wie in folgender Grafik dargestellt, ablaufen. Definieren Sie
□ selbständig wann bzw. wo explizit gewartet wird und begründen Sie ihre Entscheidung.



1.2 Testfälle

Um eine Aussage über die Ausführungszeit machen zu können, soll der Master folgende Zugriffe □ durchführen, wobei die Ergebnisse aller Testläufe in einer Tabelle zusammengefasst werden sollen:

- sequentielles Schreiben aller Speicherstellen,
- sequentielles Lesen aller Speicherstellen (Verifikation mittels Assertion),
- 10^6 Zugriffe auf zufällige Adressen, wobei zufällige Daten geschrieben und gelesen werden (Verifikation mittels Assertion).

Literatur

[Ope10] OpenCores Organization. *WISHBONE System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores, Rev B.4*, September 2010. http://cdn.opencores.org/downloads/wbspec_b4.pdf.