FH-OÖ Hagenberg/ESD
Hardware / Software Codesign, SS 2020
Florian Eibensteiner © 2020 (R 4921)

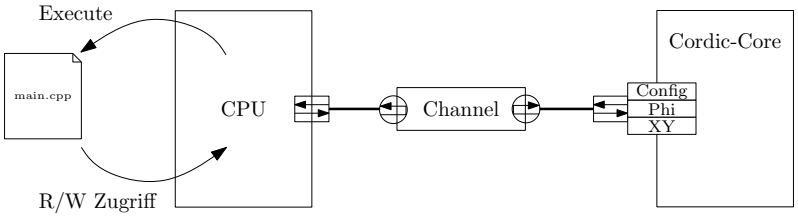
3. Übung: Virtual Prototyping

Name(n): Punkte: (max. 24)

1 Systemmodellierung

Das Modell aus der 1. Übung soll nun um ein Verhaltensmodell einer CPU erweitert werden. Dies ermöglicht es, das gesamte System – bestehend aus CPU und Cordic-Core – zu simulieren und parallel zur Hardware auch die Firmware zu entwickeln.

Wie in der letzten Übung ersichtlich wurde, wirkt sich der Grad der Abstraktion sehr stark auf die Simulationszeit aus. Damit das Verhaltensmodell als Basis für die Modellierung eines SoCs verwendet werden kann, soll die Kommunikation zwischen Peripherie und Prozessor möglichst abstrakt auf Transaktionsebene stattfinden. In unten stehender Abbildung ist das Konzept des Systemmodells dargestellt und entspricht im Wesentlichen dem LT-Modell aus der 2. Übung.

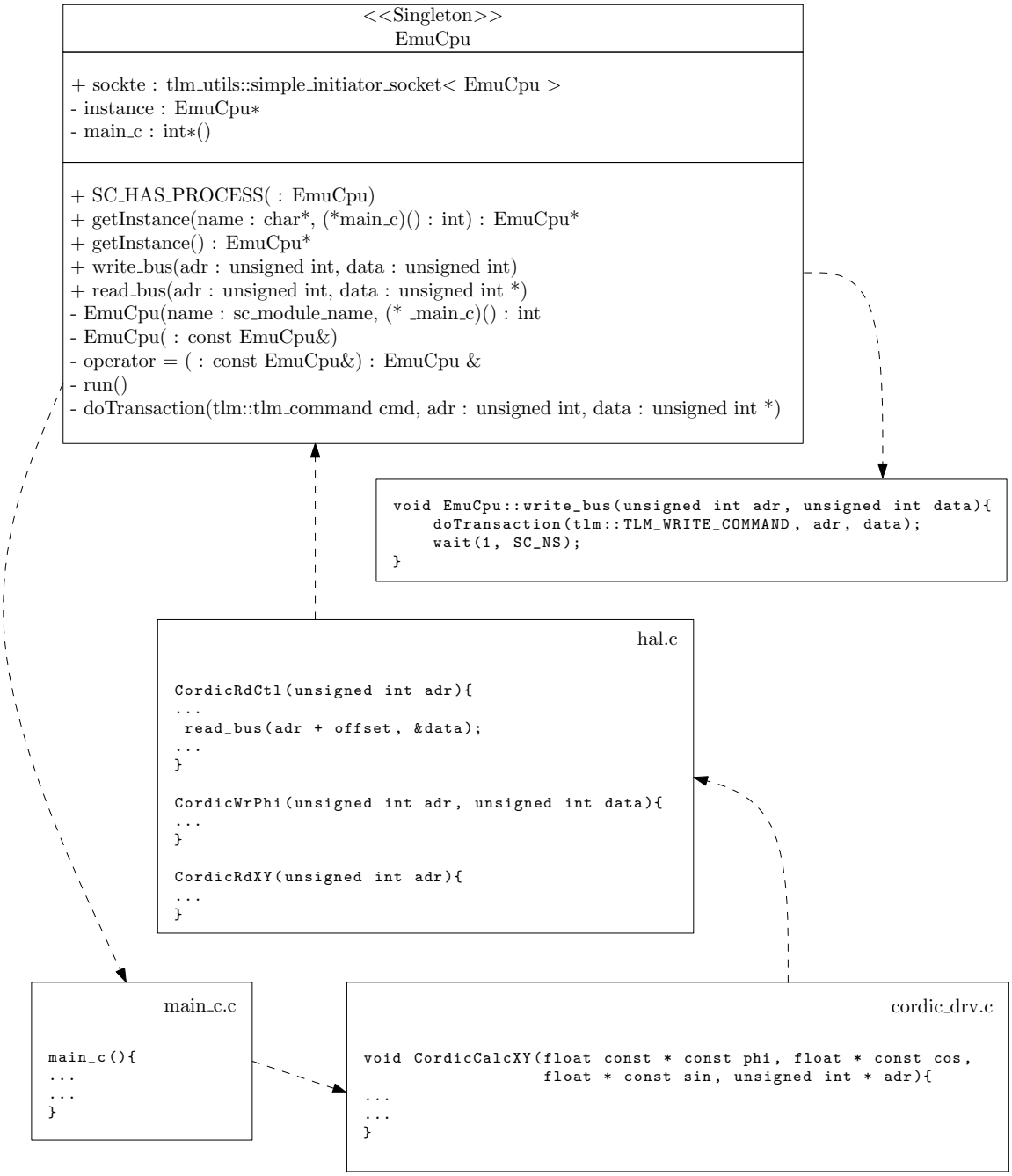


1.1 CPU

Die CPU soll als abstraktes SystemC-Modul implementiert werden. Es handelt sich dabei nicht um einen ISS (Instruction Set Simulator) für eine bestimmte Zielplattform, vielmehr wird die Firmware innerhalb eines SystemC-Threads ausgeführt. Diese Ebene der Abstraktion erlaubt zwar keine genaue Performance-Analyse der Software, ermöglicht allerdings eine sehr schnelle Simulation des gesamten Systems. Die Kommunikation zwischen dem Prozessor und der Peripherie soll mittels TLM-Kanälen (*LT-Modell*) abgewickelt werden, wobei die CPU als *Initiator* und der Cordic-Core als *Target* fungieren und die Daten als *generic Payload* übertragen werden.

Für die Entwicklung der Firmware gibt es nun zwei Möglichkeiten: entweder man implementiert die Firmware direkt als Funktion im SystemC-Modul die als Thread läuft. Hier muss allerdings die Applikation bei der Überführung auf die Zielplattform wieder aus dem SystemC-Modul herausgelöst werden.

Eine andere Möglichkeit ist, über einen Funktionszeiger direkt auf die Funktion *main()* zu zugreifen, welche extern implementiert, **keinen** SystemC-Code enthält und später direkt übernommen werden kann. Da diese Funktion außerhalb der Klasse ausgeführt wird, kann nun aber nicht mehr direkt auf die Funktionen (*read/write*) der CPU zugegriffen werden. Hier kann Abhilfe geschaffen werden indem die CPU als *Singleton* implementiert wird, wodurch eine Instanz der Klasse geholt und dadurch auf die R/W–Funktionen zugegriffen werden kann. Nachfolgende Abbildung zeigt den Aufbau der Klasse *EmuCpu*, sowie das Zusammenwirken der Applikation, des Treibers und der CPU.



Damit die Firmware und der Treiber später problemlos portiert werden können, darf in diesen Softwareteilen **kein** SystemC-Code enthalten sein.

1.2 Peripherie

Damit der Cordic-Core von der CPU angesprochen werden kann, muss er durch eine *Target*-Klasse gekapselt werden. Die Übertragung der Daten soll blockierend implementiert werden, wobei bei jeder Transaktion eine *generic Payload* transferiert wird. Die Decodierung der Adresse soll nach folgender Tabelle erfolgen, wobei der Zugriffsmode der Register beachtet werden muss:

Name	Adresse (Offset)	Mode	Beschreibung
Config und Status	0x00	R	Rdy-Flag = Config[0]
Phi	0x04	W	Eingabewinkel: Phi[21:0]
XY	0x08	R	Ergebnis: Y[31:16], X[15:0]

Die Berechnung wird gestartet, sobald ein neuer Wert in das Register *Phi* geschrieben wird. Das *Start*-Flag des Cordic-Cores wird beim Schreiben des Registers erzeugt und gesetzt. Nach Beendigung der Berechnung wird das *Rdy*-Flag gesetzt und aus dem Register *XY* kann das Ergebnis gelesen werden.

Sobald das Start-Flag in der Adresslogik erzeugt wird, muss das Rdy-Flag zurückgesetzt werden, dies sollte vom Cordic-Core aus Übung 1 bereits erledigt werden. Wird das Rdy-Flag nach Beendigung der Berechnung gesetzt, wird das Start-Flag wieder gelöscht und das Rdy-Flag bleibt gesetzt bis die nächste Berechnung beginnt.

Orientieren Sie sich bei der Implementierung des Peripherals an dem Beispiel (LT-Modell) aus der 2. Übung und stellen Sie sicher, dass die Parameter der *generic Payload* in der Funktion *b_transport* entsprechend überprüft werden. Eine DMI- und Debug-Schnittstelle muss auch hier **nicht** implementiert werden.

1.3 Firmware

Damit die Firmware später leicht in die Zielplattform integriert werden kann, soll sie nicht direkt in der CPU als SystemC-Thread, sondern via Funktionszeiger eingebunden werden. Dabei soll die Software so entwickelt werden, dass sie möglichst unverändert übernommen werden kann, was unter Umständen problematisch werden kann (z.B. Hardwarezugriffe, Target-spezifische Funktionen).

Die Applikation soll in einer Schleife *Sin* und *Cos* für alle Winkel von 0° bis 360° mit einer Schrittweite von 0.0125 berechnen, wobei als Datentyp *float* verwendet werden soll. Für die Berechnung soll folgende Funktion im Treiber aufgerufen werden:

```
1 void CordicCalcXY(float const * const phi, float * const cos,
2                   float * const sin, unsigned int * adr){
3     ...
4     ...
5 }
```

Da der Cordic-Core in Hardware nicht mit *floats* umgehen kann, muss der Winkel Phi entsprechend in eine **Fix-Point-Zahl** konvertiert werden bevor dieser in das Register geschrieben wird.

Die Adresse, die beim Aufruf übergeben wird, ist die Basisadresse des Cordic-Cores – diese Schnittstelle ist für die spätere Integration in die FPGA-Plattform notwendig. Außerdem soll die Firmware die Funktionsweise des Cordic-Cores **nicht** verifizieren, dies sollte bereits in Übung 1 gemacht worden sein.

1.4 Treiber

Der Treiber übernimmt die Eingabewinkel und gibt sie über die Funktionen, die der HAL (Hardware Abstraction Layer) zur Verfügung stellt, an die Hardware weiter. Da der Cordic-Core nur Winkel zwischen 0° und 90° berechnen kann, müssen alle Winkel zuvor in den 1. Quadranten transformiert und nach der Berechnung wieder zurückgerechnet werden. Beachten Sie, dass das Vorzeichen erhalten bleiben muss.

Sobald durch Schreiben des Eingabewinkels die Berechnung gestartet wurde, pollt der Treiber das Rdy-Flag und wartet so auf das Ende der Berechnung.

Damit der Treiber später leicht in die Zielplattform integriert werden kann, muss er entsprechend in einem eigenen Modul (*.h, *.c) gekapselt werden.

Der Zugriff auf die Hardware muss durch einen geeigneten HAL (ebenfalls ein eigenes Modul) abstrahiert werden, da sich die Zugriffe bei der Integration auf die Zielplattform mit Sicherheit ändern. Implementieren Sie für jeden Zugriff eine eigene Funktion, der die Basisadresse und das entsprechende Datum übergeben werden. Eine mögliche Realisierung könnte wie folgt aussehen:

```
1  extern void write_bus(unsigned int adr, unsigned int data);
2  extern void read_bus(unsigned int adr, unsigned int * data);
3
4  unsigned int CordicRdCtl(unsigned int adr){
5      ...
6      read_bus(adr + OFFSET_CTL, &data);
7      ...
8  }
9
10 void CordicWrPhi(unsigned int adr, unsigned int data){
11     ...
12 }
13
14 unsigned int CordicRdXY(unsigned int adr){
15     ...
16 }
```

Der explizite Zugriff erfolgt indem auf die Funktionen *write_bus* bzw. *read_bus* der CPU zugegriffen wird. Damit von C auf Funktionen einer C++ Klasse zugegriffen werden kann, müssen diese **außerhalb** der Klasse entsprechend deklariert werden (emucpu.h):

```
extern "C"{
void read_bus(unsigned int adr, unsigned int * data);
void write_bus(unsigned int adr, unsigned int data);
};
```

und auch definiert werden. Der Zugriff auf die Member-Funktionen der Klasse erfolgt dann in diesen C++ Funktionen wie folgt:

```
/* Function to read from given adr.
 * Called by firmware and encapsulating cpu access.
 */
void read_bus(unsigned int adr, unsigned int *data){
    EmuCpu::getInstance()->read_bus(adr, data);
}

/* Function to write to given adr.
 * Called by firmware and encapsulating cpu access.
 */
void write_bus(unsigned int adr, unsigned int data){
    EmuCpu::getInstance()->write_bus(adr, data);
}
```

1.5 Abgabe und Hinweise

Die Abgabe soll folgendes beinhalten:

- kompilierbares und ausführbares VS-Projekt und
- ein Trace-File welches Start- und Rdy-Flag visualisiert.

Instanzieren Sie den Cordic-Core als Sub-Modul in der Target-Klasse, um ihn bei Bedarf einfach durch ein Modell auf anderer Abstraktionsstufe ersetzen zu können, bzw. Modelle parallel simulieren zu können.

Beachten Sie dass in einer späteren Übung zur Toolchain der Zielplattform gewechselt wird und dort ggf. Versionen des GCC eingesetzt werden, die Probleme mit C++11/14/17 aufweisen können.

Beenden Sie die Simulation indem Sie am Ende der Funktion *run* in der EmuCPU die Funktion *sc_stop()* aufrufen. Dadurch kann die Simulation beendet werden, auch wenn später getaktete Module verwendet werden.