FH-OÖ Hagenberg/ESD
Hardware / Software Codesign, SS 2020
Florian Eibensteiner © 2020 (R 4969)

5. Übung: Virtual Prototyping – Get HLS ready!

OBERÖSTERREICH

Name(n):

1 Refine it – synthetisierbare Hardware Beschreibung

Das Modell aus der 3. Übung ermöglichte es, das gesamte System zu simulieren und gleichermaßen Software als auch Hardware zu entwickeln und zu verifizieren. In dieser Übung soll nun in einem weiteren Schritt die Hardware bis hin zur zyklengenauen Implementierung verfeinert werden.

Als Zielplattform soll der aus der 4. Übung bereits bekannte Xilinx FPGA *XC7Z007S* [Xil18] eingesetzt werden. Die Synthese der Cordic IP soll später mittels *Vivado HLx* [Xil19] direkt aus der Beschreibung C++ erfolgen. *Vivado HLx* unterstützt hier entweder die Verwendung von SystemC [Xil19, S.372ff] oder C++ in Verbindung mit Xilinx HLS C Libraries [Xil19, S.201ff].

Wir werden vorerst bei SystemC bleiben und dazu die bestehende Beschreibung aus der 3. Übung entsprechend verfeinert.

1.1 Cordic - Refinement in SystemC

In der ersten Übung wurde der Cordic-Algorithmus, angelehnt an die Implementierung in MATLAB, als SystemC-Modell implementiert, wobei weder das Timing noch die interne Darstellung der Daten berücksichtigt wurde.

Nun soll das Modell weiter verfeinert werden und damit auf RTL-Ebene gebracht werden. Erweitern Sie zuerst das Modell um eine Takt- und eine Resetleitung, damit der Cordic-Core zyklengenau arbeiten kann. Dabei soll in jedem Takt ein Iterationsschritt berechnet werden. Wie bisher, soll die Berechnung starten sobald das Start-Flag gesetzt wird. Beachten Sie, dass das Modell bei jeder Taktflanke (Sensitivity-List) arbeiten soll und die Berechnung durch das Start-Signal lediglich angestoßen wird. Wenn die Berechnung abgeschlossen wurde, soll das *Rdy*-Flag gesetzt werden.

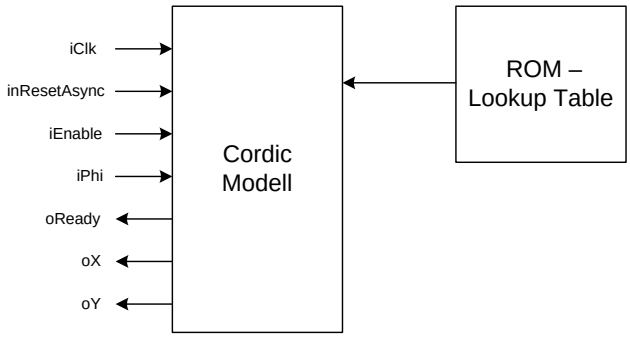
Als nächstes soll die noch immer enthaltene $atan()$ Funktion ersetzt werden, da diese später in Hardware ebenfalls nicht verfügbar ist. Diese Funktion kann durch eine Lookup-Tabelle ersetzt werden, welche die benötigten Werte enthält und beispielsweise mit MATLAB vorberechnet werden kann. Definieren Sie das Format der Einträge so, dass die geforderte Genauigkeit ($\pm 2^{-15}$ bzw. $\epsilon \leq 2^{-14}$ bei 16 Iterationen) erreicht werden kann. Gehen Sie davon aus, dass für die Berechnung der Winkelfunktionen immer 16 Iterationen durchgeführt werden.

Da der Cordic-Algorithmus eine verstärkende Wirkung hat, muss der Faktor K_i berücksichtigt werden. Überlegen Sie mit Rücksicht auf eine spätere Implementierung in Hardware, an welcher

Stelle – in Software oder in Hardware – der Faktor multipliziert werden sollte. Begründen Sie Ihre
□ Entscheidung.

Für die Darstellung der Zahlen soll innerhalb des Moduls Festkomma-Arithmetik verwendet werden. Sie können dabei auf den in SystemC vorhandenen Datentyp zurückgreifen. Wählen Sie ein sinnvolles Format, damit weiterhin die geforderte Genauigkeit erreicht wird. **Anmerkung**: Vivado HLS erwartet bei Schiebeoperationen mit Fix-Point Datentypen einen *Integer* für die Anzahl der zuschiebenden Bits.

In unten stehender Abbildung ist die Schnittstelle des Cordic-Cores nach den einzelnen Verfeinerungsschritten als Blockdiagramm dargestellt.



1.2 High-Level Synthese – SystemC Coding Styles

Das Synthese-Werkzeug Vivado HLS von Xilinx unterstützt die direkte Übersetzung von SystemC nach *VHDL* oder *Verilog*, sprich das Werkzeug generiert eine synthetisierbare RTL-Beschreibung in einer HDL. Allerdings kann für die HLS nur ein synthesefähiges Subset aus SystemC verwendet werden. So dürfen beispielsweise keine SystemC-Module in anderen SystemC-Modulen definiert werden und es werden auch keine *SC_THREAD*s unterstützt – näheres finden Sie in [Xil19, S.372ff] und [Syn09].

Bei der Verwendung von *SC_CTHREAD*s muss im Konstruktor neben dem Takt auch die Reset-Bedingung (Anmerkung: in IEEE 1666 nur synchrone Resets spezifizierbar [Syn09, S.49]) definiert werden, z.B.:

```
1  ...
2  sc_in<bool> clk;
3  sc_in<bool> nrst;
4  // Constructor
5  SC_CTOR(MyCordic){
6    // Process Registration
7    SC_CTHREAD(calc_cordic, clk.pos());
8    reset_signal_is(nrst, false); // low-active reset
9  }
10 ...
```

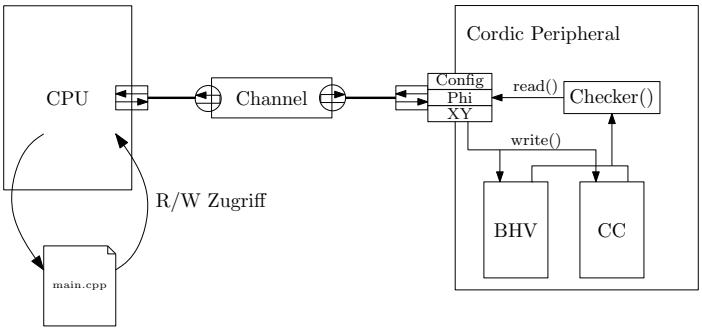
In der Funktion selbst muss zwischen Reset- und zyklischem Teil unterschieden werden, z.B.:

```
1  void MyCordic::calc_cordic(){
2    // implement reset behavior here
3    // -> must be exec in a single cycle!
4    wait(); // wait for clk -> implies end of reset part
5
6    while(true){
7      // implement cyclic part here
8      wait();
9    }
10 }
```

1.3 Verifikation – Golden Model

Damit Fehler, die im Zuge der Verfeinerung entstehen schnell gefunden werden können, soll ein sogenanntes *Golden Model* (GM) verwendet werden. Ein GM ist eine sehr abstrakte Umsetzung des eigentlichen Problems und dient während der gesamten Entwicklungsphase als Referenzimplementierung. In unserem Fall kann die Implementierung aus Übung 1 als GM verwendet werden. Es wäre auch denkbar, die trigonometrischen Funktionen aus `math.h` als GM zu verwenden.

Wie in der folgenden Abbildung dargestellt, werden beide Modelle im Peripheral (*TLM Target*) instantiiert, wobei die Daten parallel in **beide** Modelle geschrieben werden. Wenn das Ergebnis der Berechnung von der CPU gelesen wird, muss vorher durch eine *Checker*-**Funktion** überprüft werden ob die Ergebnisse beider Modelle übereinstimmen. Im Fehlerfall kann der *Checker* eine *Assertion* werfen oder eine entsprechende Ausgabe machen. Darum muss **keine** explizite Testbench für das zyklengenaue Modell geschrieben werden.



Da das Modul Takt und Reset benötigt, müssen diese Signale in der *Target*-Klasse erzeugt werden. Wählen Sie hierbei sinnvolle Werte für die Periode des Taktes und die Dauer des Resets – idealerweise stehen diese in **keinem** ganzzahligen Verhältnis.

Das *Start*-Flag soll wie in Übung 3 automatisch generiert werden wenn ein neuer Eingabewinkel geschrieben wird.

Nachdem das restliche System **ohne** Zeitbasis arbeitet und keine Notiz vom Takt des verfeinerten Cordic-Cores nimmt, müssen die einzelnen Module an den entsprechenden Stellen **synchronisiert**

werden. Zum einem darf die CPU erst dann mit der Exekution der Firmware beginnen wenn alle Peripherals mit dem **Reset** fertig sind. Zum anderen können die Ergebnisse erst gelesen werden wenn beide Modelle, untimed **und** getaktet, mit der Berechnung fertig sind. Überlegen Sie sich wo bzw. wie die Synchronisation eingebaut werden kann und Begründen Sie Ihre Umsetzung. □

2 Abgabe und Hinweise

Die Abgabe soll folgendes beinhalten:

- kompilierbares und ausführbares VS-Projekt und
- ein Trace-File welches Tackt, Reset, Start- und Rdy-Flag, Phi, Cos und Sin visualisiert.

Instanzieren Sie das BHV- und das CC-Modell des Cordic-Cores als Sub-Modul in der Target-Klasse, um diese 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 gezielt nach Abarbeitung der Firmware beendet werden, auch wenn getaktete Module im System vorhanden.

Literatur

[Syn09] Synthesis Working Group of Open SystemC Initiative. *SystemC Synthesizable Subset – 1.3 Draft*, June 2009.

[Xil18] Xilinx. *Zynq-7000 SoC Technical Reference Manual*, July 2018.

[Xil19] Xilinx. *Vivado Design Suite User Guide – High-Level Synthesis*, July 2019.