FH-OÖ Hagenberg/ESD
Hardware / Software Codesign, SS 2020
Florian Eibensteiner © 2020 (R 4997)

7. Übung: High Level Synthesis – What comes out?

Name(n):

Im nächsten Schritt soll der Codic-Core mittels High-Level-Synthese (HLS) auf RTL-Ebene gebracht werden, um in weiterer Folge in ein FPGA-Design integriert werden zu können. Dabei soll durch das HLS-Werkzeug Vivado HLS auch eine AXI4-Lite Schnittstelle erzeugt werden. damit die IP an das Zynq7 Prozessorsystem angebunden werden kann. Aus Gründen der Einfachheit wird in dieser Übung nicht der gesamte VP in die Umgebung von Xilinx migriert sondern lediglich der synthesefähige Cordic-Core.

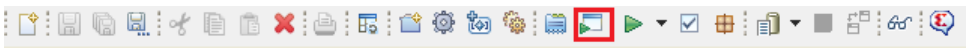
Nähere Informationen zur Synthese von SystemC in Vivado HLS finden Sie in [Xil19b] ab Seite 372, sowie allgemeine Information und Beispiele in [Xil19a] (Source-Files zu den Beispielen befinden sich im Elearning im Abschnitt zur 7. Übung). Beispielprojekte zur Synthese von SystemC mit Vivado HLS befinden sich auch bei den von Xilinx zur Verfügung gestellten Beispielen, welche sich im Installationspfad von Vivado befinden (unter `C:\Xilinx\Vivado\2019.1\examples\coding`, beispielsweise die Projekte `sc_combo_method` oder `sc_sequ_cthread`) – Pfad entsprechend an Ihre Installation anpassen.

1 Aus SystemC wird RTL

Erstellen Sie ein neues Projekt in *Vivado HlS* und integrieren Sie den synthesefähigen Cordic Core aus Übung 5. Als Testbench kann jene aus der 1. Übung verwendet werden, wobei hier kein Trace-File erstellt werden muss. Gehen Sie bei der Projekterstellung wie folgt vor:

- Eingabe des Projektnamen und Speicherorts
- Hinzufügen der zu synthetisierenden Sourcen und Definition der *Top Function* (bei SystemC hier den Namen des Moduls eingeben, nicht den Funktionsnamen der vorgeschlagen wird). Es reicht hier das cpp-File mit der Implementierung der zu synthetisieren Funktion anzugeben.
- Hinzufügen der Testbench-Files (inkl. `main.cpp` mit der Funktion `sc_main(...)`)
- Einstellen des Taktes und des Ziel-FPGAs

Wenn das Projekt erstellt wurde starten Sie zunächst eine Simulation, um zu testen ob das Design fehlerfrei funktioniert.



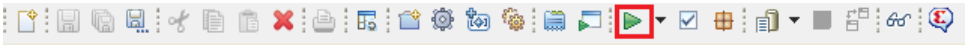
Teile im Code die für Debug-Zwecke eingebunden werden, wie beispielsweise Ausgaben, können

mittels Präprozessoranweisung von der Synthese ausgeschlossen werden (siehe [Xil19b, S.375].

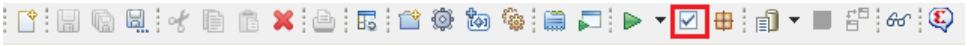
Listing 1: Verwendung des Makros __SYNTHESIS__ zur Einbindung von Code der nicht synthetisiert werden soll.

```
1 void adder::Sumup(){
2 sum.write(a.read() + b.read());
3 // debug prints are not included for synthesis
4 #ifndef __SYNTHESIS__
5 cout << "Sum is " << a << " + " << b << " = " << a+b << " @"
6      << sc_time_stamp() << endl;
7 #endif
8 }
```

Nachdem das Design in einer Simulation getestet wurde kann die Synthese gestartet werden. Im Report der bei der Synthese generiert wird kann der Ressourcenbedarf und das Timing eingesehen werden.



Nach erfolgreich abgeschlossener Synthese kann nun der generierte RTL-Code mittel Co-Simulation erneut verifiziert werden. Vivado HLS generiert hierzu automatisch einen Wrapper in SystemC der zum einen die verwendeten Datentypen auf jene umlegt die von der Synthese verwendet werden. Zum anderen wird der generierte RTL-Code eingebunden. Zur Verifikation wird die gleiche Testbench verwendet wie schon zuvor bei der Simulation des SystemC-Moduls. Näheres zur Co-Simulation finden Sie in [Xil19b, S.181ff] und in [Xil19a, Ch.8, S.162].



1.1 Hinweise für die HLS mit SystemC

Der Design-Flow in Vivado HLS weist für SystemC leider einige Inkonsistenzen auf die für den Entwurfsablauf und die spätere Integration in den IP-Katalog von Vivado sowie Verwendung in SDK wichtig sind.

Das **Header-File** in dem das Top-Modul definiert ist sollte **gleich benannt** werden wie das **Top-Modul** selbst. Bei der Synthese und der Co-Simulation von C++ und RTL-Code werden einige Wrapper-Files generiert die nach dem Modul benannt werden, dadurch kommt es dann zu Fehlern bei der Inkludierung des Moduls in der Testbench. Für einen Addierer würde das beispielsweise wie folgt aussehen.

Listing 2: Definition des Moduls *adder* in der Datei *adder.h*.

```
1 #ifndef ADDER_H_
2 #define ADDER_H_
3
4 #include<systemc.h>
5
```

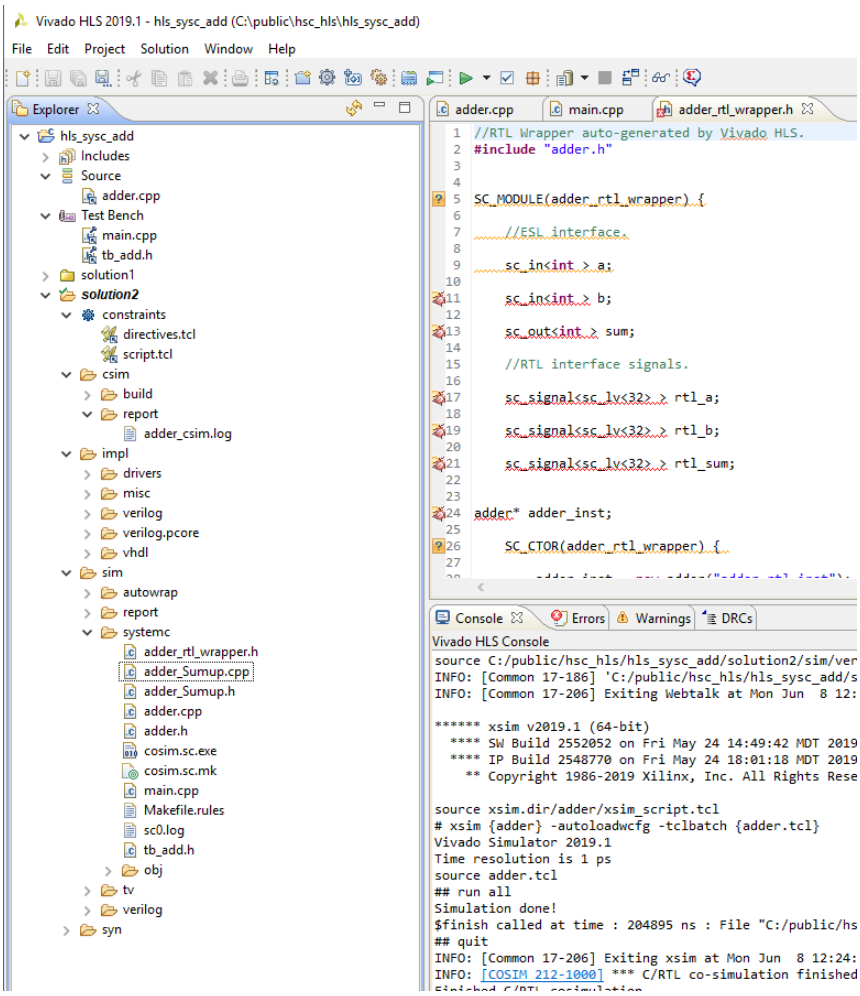
```
6  SC_MODULE(adder){
7      sc_in<int> a;
8      sc_in<int> b;
9      sc_out<int> sum;
10
11     SC_CTOR(adder){
12         SC_METHOD(Sumup);
13         sensitive << a << b;
14     }
15
16     void Sumup();
17 };
18 #endif
```

Bei der Synthese werden nicht nur die internen Datentypen sondern auch jene der Ports auf Datentypen umgebaut die in Hardware abbildbar sind, im Fall des Moduls *Adder* z. B. von `sc_in<int>` auf `sc_in< sc_lv<32> >`. Damit für die Co-Simulation weiterhin die **bestehende Testbench** verwendet werden kann, generiert Vivado HLS entsprechende **Wrapper-Files** die in `main.cpp` eingebunden werden müssen. Um die selbe Code-Basis für Simulation und Co-Simulation verwenden zu können, empfiehlt sich auch hier der Einsatz entsprechender **Präprozessoranweisungen**.

Listing 3: Einbinden der SystemC-Warpper mittels Makro __RTL_SIMULATION__.

```
1 // Settings to operate with Vivado HLS C/RTL simulation wrappers
2 #ifdef __RTL_SIMULATION__
3 #include "adder_rtl_wrapper.h"
4 #define adder adder_rtl_wrapper
5 #else
6 #include "adder.h"
7 #endif
8 ...
```

Die generierten Wrapper-Files für die Co-Simulation liegen in aktiven *Solution* im Ordner `\sim\systemc\`, wie in untenstehnder Grafik dargestellt.



1.2 Schnittstellen-Synthese

Leider wird für SystemC die automatische Schnittstellen-Synthese nicht unterstützt. Das Verhalten und die Funktionsweise von I/O Ports müssen vollständig im Code beschrieben werden. Ausnahmen bilden lediglich Block-RAM Schnittstellen und AXI4 Stream, Lite und Master Interfaces [Xil19b, S.89].

Das von uns benötigte AXI4-Lite Interface kann wie folgt mittels Direktiven erzeugt werden [Xil19b, S.92ff]. Die Definition der Ports erfolgt wie gewöhnlich im Header-File.

```
1 SC_MODULE(Cordic_Sysc){
2
3   // port definition
4   sc_in<bool> iClk;
5   sc_in<bool> inResetAsync;
6   sc_in<bool> iStart;
7   sc_in<cordic_in> iPhi;
8
9   sc_out<bool> oRdy;
```

```
10    sc_out<cordic_out> oX;
11    sc_out<cordic_out> oY;
12
13    //CTOR
14    SC_CTOR(Cordic_Sysc){
15      SC_CTHREAD(ComputeCordic, iClk.pos());
16        reset_signal_is(inResetAsync, false);
17    }
18  };
```

Mittels der Direktive *Resource* kann in der Funktion, in der die Ports verwendet werden, ein AXI4-Lite Interface erzeugt werden. Für obige Moduldefinition würden die Direktiven wie folgt aussehen, um die Ports *iStart*, *iPhi*, *oX*, *oY* und *oRdy* zu einem AXI-Bus zusammenzufassen.

```
1  void Cordic_Sysc::ComputeCordic(){
2  #pragma HLS RESOURCE variable=iStart core=AXI4LiteS metadata="-bus_bundle cordic_if"
3  #pragma HLS RESOURCE variable=iPhi core=AXI4LiteS metadata="-bus_bundle cordic_if"
4  #pragma HLS RESOURCE variable=oX core=AXI4LiteS metadata="-bus_bundle cordic_if"
5  #pragma HLS RESOURCE variable=oY core=AXI4LiteS metadata="-bus_bundle cordic_if"
6  #pragma HLS RESOURCE variable=oRdy core=AXI4LiteS metadata="-bus_bundle cordic_if"
7  ...
```

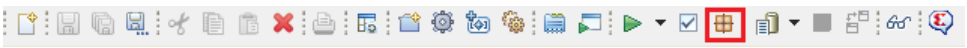
Alle hier angegebenen Ports werden zu eigenen Registern auf die via AXI-Bus zugegriffen werden kann. Verglichen mit der bisherigen Umsetzung am VP ergeben sich nun folgende Unterschiede:

- Das Start-Flag wird nicht wie angenommen in der Adresslogik erzeugt, sodern muss seperat gesetzt werden.
- Es wird kein Interrupt-Port erzeugt.
- oX und oY werden als getrennte Register ausgeführt und müssen einzeln ausgelesen werden.

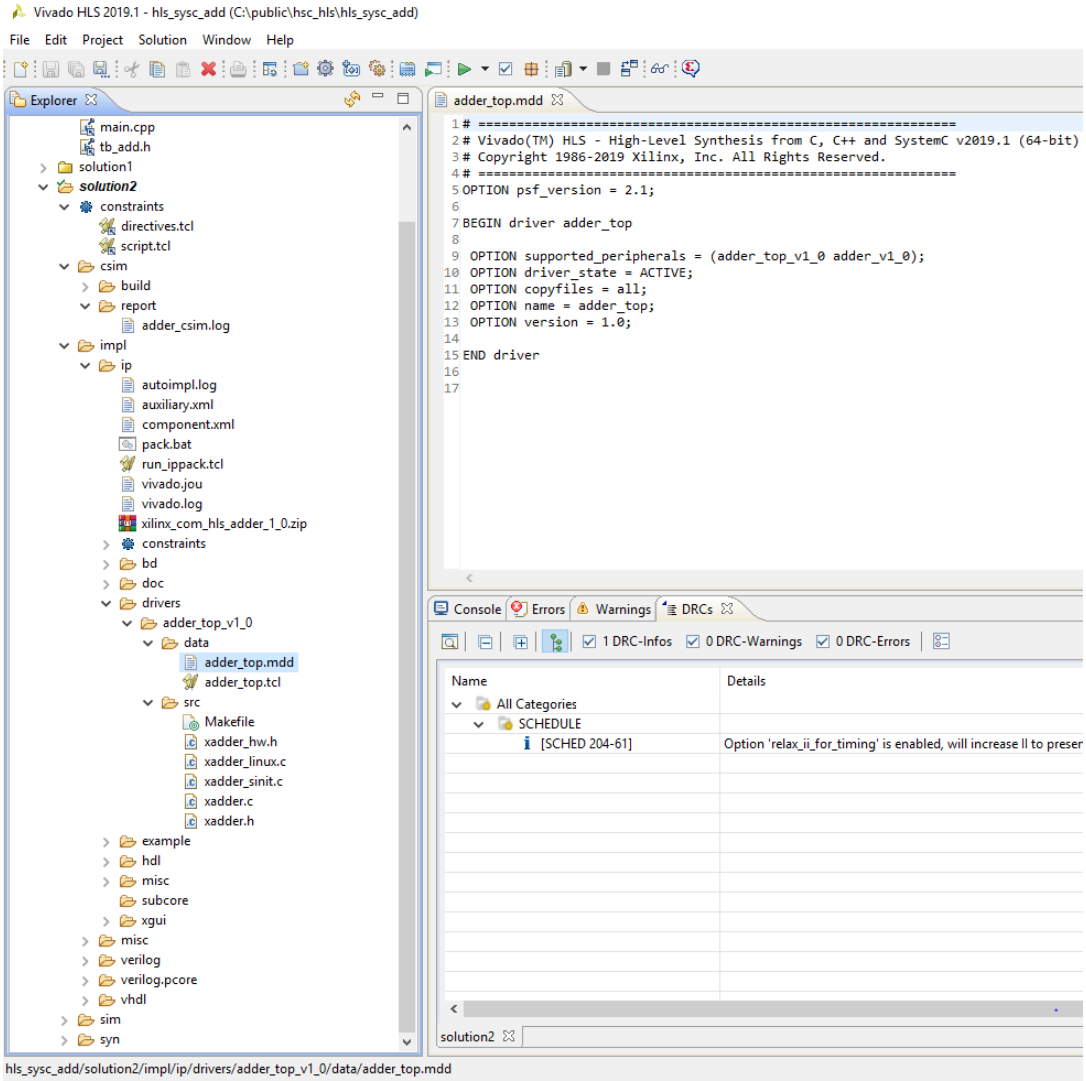
Das führt leider dazu, dass die Firmware final entsprechend angepasst werden muss.

Bei der Verwendung von SystemC werden nach der Synthese im Report immer noch die Standard-Ports angezeigt und keine AXI-Ports – diese werden erste beim Export hinzugefügt [Xil19b, S.93].

Exportieren Sie nun das Design als IP im Format *IP Catalog*, um die Treiber und Schnittstellen für die weitere Verwendung in *Vivado* zu erzeugen (siehe [Xil19b, S.196ff]).

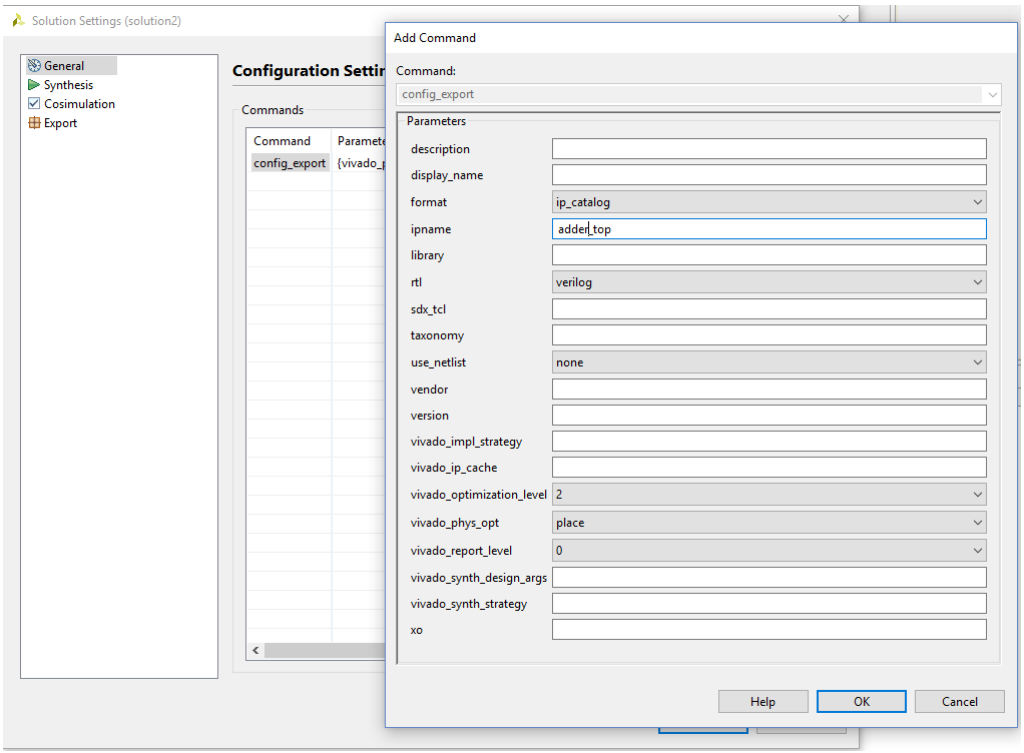


Beim Export werden neben den für die Integration in ein Block-Design nötigen RTL- Sourcen auch die C-Treiberdateien für den Zugriff über die AXI4-Slave-Schnittstelle erzeugt. Dabei wird in der aktiven *Solution* folgende Ordnerstruktur generiert: `impl/ip/`. Der Ordner `ip` bzw. das Zip-Archiv können dann in weiter Folge in *Vivado* bzw. *Xilinx SDK* als Repository eingbunden werden (dazu mehr in der nächsten Übung)



Wie in obiger Abbildung ersichtlich ist, wird beim Export eines SystemC Moduls bei der Treiberinformation das Postfix *top* angehängt. Bei der Softwareentwicklung im Xilinx SDK wird bei der Erzeugung eins Board-Support-Packages der Treiber nur dann eingebunden wenn auch der eingebundene Hardware-Block dieses Postfix trägt. Dies kann sichergestellt werden indem in den Einstellungen für die aktive *Solution* der Name der IP enstprechend eingestellt wird. Gehen Sie dabei wir folgt vor:

- Öffnen Sie die Eintellungen für die Solution über die Menüleiste *Solution* → *Solution Settings*. . .
- Unter *General* sollte das Commando *config_export* bereits existieren, klicken Sie darauf und anschließend auf *Edit*. . .. Falls das Commando noch nicht vorhanden ist, klicken Sie auf *Add*. . . und wählen Sie als Commando *config_export* aus.
- Tragen Sie im Feld *ipname* den Namen <*sc_modul_name*>_*top* ein. In unten stehender Grafik wurde dies am Beispiel des Moduls *Adder* gemacht.



1.3 Bonusziele

Durch die Implementierung folgender Features können Bonuspunkte erreicht werden.

- Hinzufügen einer Methode zur Erzeugung des Start-Flags sobald ein neuer Winkel Phi geschrieben wurde.
- Zusammenfassen der Ergebnisse X und Y auf einen 32 Bit Vektor, damit dieser durch einen Zugriff gelesen werden können.

2 Abgabe

Die Abgabe soll folgendes beinhalten:

- kompilierbares und synthetisierbares Vivado HLS Projekt

In Vivado HLS können Projekte einfach archiviert werden. Gehen Sie dazu unter *File* → *Archive Project...*

Literatur

[Xil19a] Xilinx. *Vivado Design Suite Tutorial – High-Level Synthesis*, May 2019.

[Xil19b] Xilinx. *Vivado Design Suite User Guide – High-Level Synthesis*, July 2019.