

# Vivado HLS - Labs

Prof. Dr.-Ing. Frank Kesel  
Hochschule Pforzheim

29. September 2017

## Inhaltsverzeichnis

<b>1</b>	<b>Vorbemerkungen</b>	<b>2</b>
<b>2</b>	<b>Lab1: Einführung</b>	<b>2</b>
2.1	Anlegen eines Projekts . . . . .	2
2.2	Validierung des C-Codes . . . . .	5
2.3	Synthese des C-Codes . . . . .	6
2.4	RTL Verifikation: C/RTL Cosimulation . . . . .	8
2.5	Erzeugung des IP Cores: Export RTL . . . . .	9
<b>3</b>	<b>Lab2: Verwendung von TCL-Skripten</b>	<b>11</b>
<b>4</b>	<b>Lab3: I/O Interfaces und Optimierung des Durchsatzes</b>	<b>13</b>
4.1	Spezifikation von I/O Interfaces . . . . .	13
4.2	Optimierung des Durchsatzes . . . . .	16
<b>5</b>	<b>Lab4: Interface Synthese</b>	<b>18</b>
5.1	Verwendung von RAM-Ports . . . . .	18
5.2	Verwendung von Dual-Port-RAM und FIFO . . . . .	19
5.3	Partitionierung der Felder . . . . .	20
5.4	AXI-Stream und AXI-Lite Interface . . . . .	21
<b>6</b>	<b>Lab5: Design Optimierung</b>	<b>25</b>
6.1	Analyse des Default-Designs . . . . .	25
6.2	Pipelining der inneren Schleife . . . . .	26
6.3	Pipelining der <code>col</code> -Schleife . . . . .	29
6.4	Reshaping der Felder . . . . .	30

## 1 Vorbemerkungen

Die nachfolgenden Labs dienen der Einarbeitung in die High-Level Synthese mit Vivado HLS. Um die Übungen durchzuführen benötigen Sie Vivado HLS Version 2016.4. Dieses Werkzeug ist in der „Vivado Design Suite - HLx Editions“ enthalten, welches von der Xilinx Homepage heruntergeladen werden kann. Die Anleitung geht davon aus, dass Vivado HLS unter Windows benutzt wird.

Entpacken Sie die Quellen aus der zu diesen Labs gehörigen Datei „hls\_labs.zip“ in ein Arbeitsverzeichnis auf Ihrem Computer. Die nachfolgenden Pfadangaben beziehen sich auf das Wurzelverzeichnis „hls\_labs“ der entpackten Verzeichnisstruktur.

## 2 Lab1: Einführung

Im ersten Lab werden wir die Vivado HLS Entwicklungsumgebung anhand eines Beispiels kennenlernen. Bei diesem Beispiel handelt es sich um ein einfaches FIR-Filter.

### 2.1 Anlegen eines Projekts

- Starten Sie Vivado HLS. Sie finden es normalerweise unter: Xilinx Design Tools > Vivado 2016.4 > Vivado HLS > Vivado HLS 2016.4
- Als erstes sehen Sie die „Welcome Page“ von Vivado HLS, siehe Abbildung 1. Wählen Sie `Create New Project`, um ein neues Projekt anzulegen.

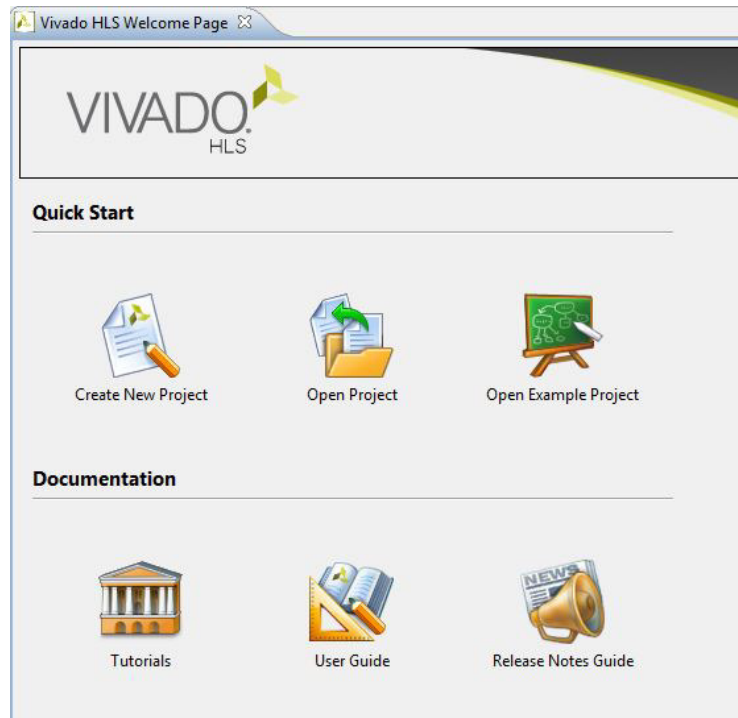


Abbildung 1: Vivado HLS Startfenster (Ausschnitt)

- Geben Sie den Projektnamen und den Speicherort wie in Abbildung 2 gezeigt an. Im lab1-Ordner wird nun ein Unterverzeichnis lab1\_proj angelegt. Drücken Sie auf Next.

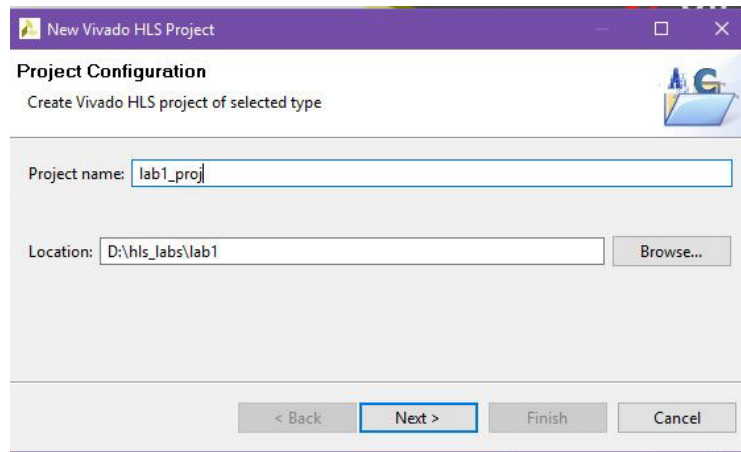


Abbildung 2: Project Configuration

- Im nächsten Fenster (Abbildung 3) werden die Quelldateien des Designs angegeben. Drücken Sie hierzu auf **Add Files** und wählen im Verzeichnis `src` die Datei `fir.c` aus. Mit **Browse** wählen Sie die „Toplevel-Funktion“ des Designs aus, dies ist im Beispiel die Funktion `fir`. Wenn das Design aus weiteren C/C++-Quelldateien besteht, so müssen an dieser Stelle alle Quelldateien hinzugefügt werden. Header-Datei, die im gleichen Verzeichnis liegen, werden automatisch hinzugefügt. Drücken Sie auf **Next**.

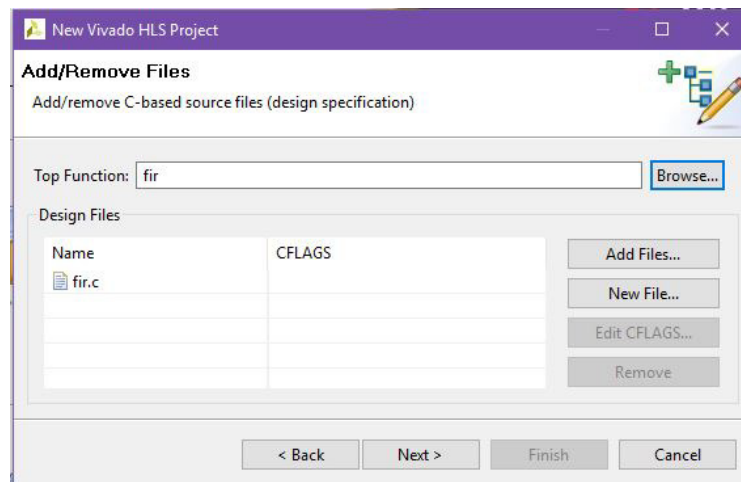


Abbildung 3: Project Design Files

- Im nächsten Fenster (Abbildung 4) werden die Dateien für die Testbench hinzugefügt. Dies sind C-Quelldateien (`fir_test.c`) und auch Dateien, die in der Testbench (`out.gold.dat`) benutzt werden. Wenn Sie hier nicht alle benötigten Dateien angeben, so kann die Simulation später nicht korrekt ausgeführt werden. Drücken Sie auf **Next**.
- Im nächsten Fenster müssen die Technologieangaben für die erste „Solution“ angelegt werden. Belassen Sie den voreingestellten Namen der Solution und die Taktperiode von 10 ns. Die „Uncertainty“ wird von der Taktperiode subtrahiert und ist eine zusätzliche Marge für eine eventuelle spätere Vergrößerung der Verzögerungszeiten bei der Logiksynthese und bei Place&Route. Wenn Sie nichts angeben, dann ist die Marge 12,5% von der angegebenen Taktperiode. Bei **Part Selection** wählen Sie den Ziel-FPGA-Baustein aus, verwenden Sie hier den in Abbildung 5 angegebenen FPGA. Drücken Sie nun auf **Finish**.

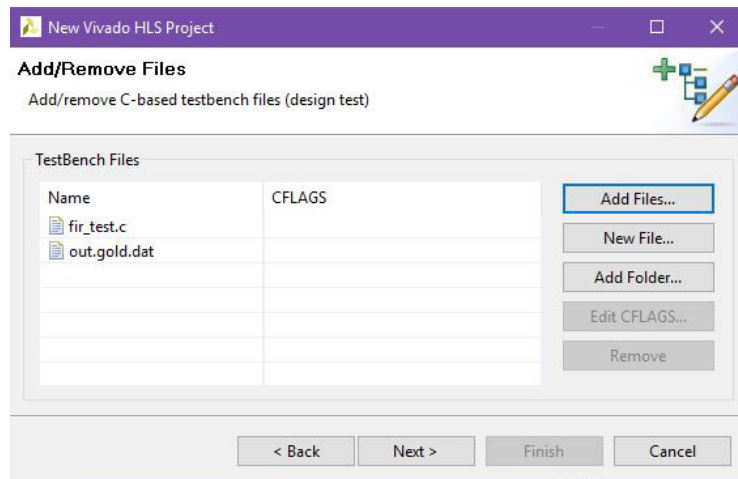


Abbildung 4: Project Testbench Files

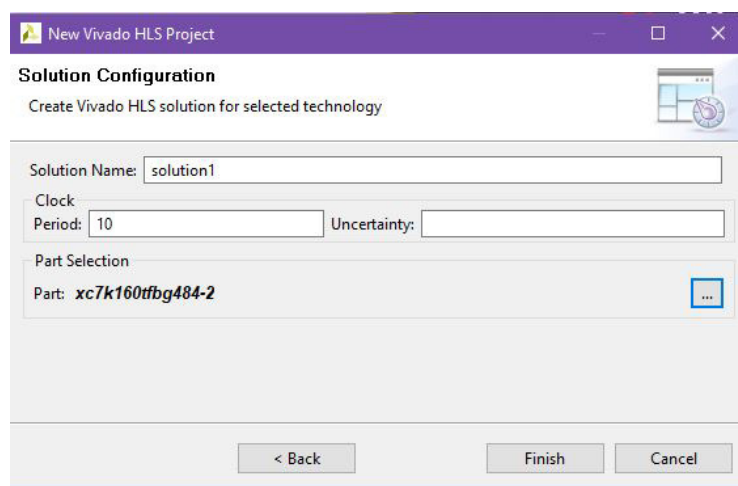


Abbildung 5: Einstellungen für die Solution

- Ein Vivado HLS Projekt ist hierarchisch aufgebaut. Zu einem Projekt gehören zunächst alle Quelldateien für das Design, also der in Hardware umzusetzende Code. Dann gehören auch alle Quelldateien für die Testbench dazu, inklusive der Dateien, die die Testbench einliest. Die Testbench wird nur für die Simulation benötigt. Ein Projekt kann aus mehreren Solutions bestehen, wobei für jede Solution eine unterschiedliche Zieltechnologie oder unterschiedliche Direktiven angegeben werden können. Alle Solutions verwenden die gleichen Quelldateien. Einstellungen für das Projekt oder für die einzelnen Solutions können auch später noch geändert werden.
- Bevor wir fortfahren soll hier die in Abbildung 6 gezeigte graphische Oberfläche (GUI) von Vivado HLS besprochen werden. Da die GUI auf Eclipse aufbaut, werden Sie sich schnell zurecht finden, wenn Sie mit der Eclipse-Oberfläche vertraut sind. Über die Toolbar können die meisten der nachfolgenden Operationen gestartet werden (Simulation, Synthese, Cosimulation, Export). Wenn Sie mit der Maus über die Toolbar-Buttons fahren, sehen Sie welche Operation ausgeführt wird. Die Perspectives sind aus Eclipse bekannt und ermöglichen ein schnelles Umschalten der Gesamtansicht, je nach Aufgabenstellung (Debug, Synthesis, Analysis). Der Project Explorer ermöglicht Ihnen die Navigation durch den Quellcode und durch die Ergebnisse der Operationen in den einzelnen Solutions. Im Information Pane werden die Dateien geöffnet (Quelldateien, Reports). Im Auxiliary Pane

erscheinen Kontext-bezogene Informationen, je nach geöffneter Datei im Information Pane. Hier werden beispielsweise später die für die HLS wichtigen Direktiven angezeigt. In der Console werden Meldungen angezeigt, während die einzelnen Operationen ausgeführt werden. Es kann sich zum Beispiel durchaus lohnen, die Meldungen, welche die Synthese-Operation ausgibt, genauer zu studieren.

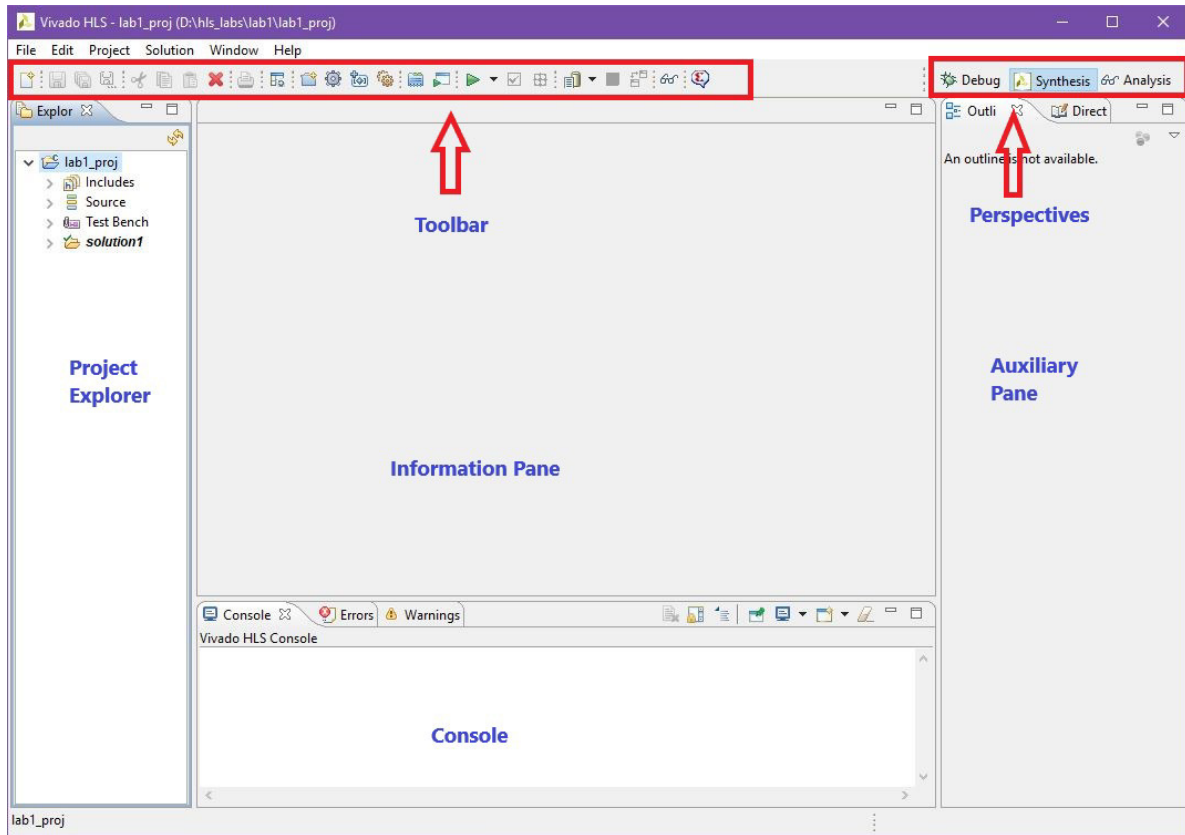


Abbildung 6: Vivado HLS Graphical User Interface

## 2.2 Validierung des C-Codes

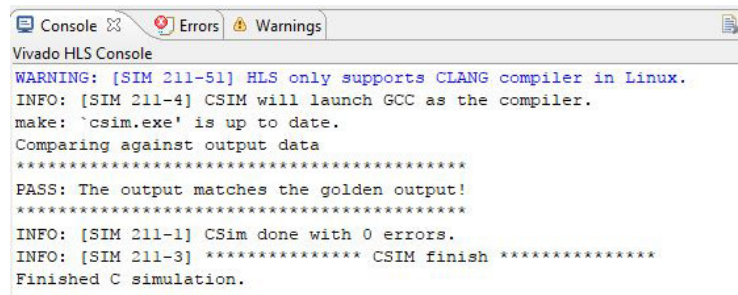
Der erste Schritt in einem HLS-Projekt ist es, den C-Code zu schreiben und zu validieren. Dies wird als `C Validation` oder `C Simulation` bezeichnet. Für die `C Simulation` wird eine Testbench benötigt. Expandieren Sie im Project Explorer den Eintrag `Test Bench` und öffnen Sie die Datei `fir_test.c`.

Studieren Sie nun den Code der Testbench. In der `main()`-Funktion wird die Funktion `fir` aufgerufen, welche das Design darstellt und woraus später der IP Core wird. Diese Testbench ist „self-checking“ und bedeutet, dass die Ergebnisse unserer Design-Funktion mit erwarteten Werten verglichen werden, welche in der Datei `out.gold.dat` gespeichert sind. In diesem Fall ist es so, dass die Ergebnisse des Designs in der Datei `out.dat` zunächst gespeichert werden und dann am Ende durch Aufruf der Systemfunktion `diff` mit den erwarteten Werten verglichen werden. Diese erwarteten Werte kann man sich beispielsweise aus einer Matlab-Simulation erzeugen, das Matlab-Modell wäre dann das so genannte „Golden Device“. Genauso kann man aus einer Matlab-Simulation auch die Eingangswerte erzeugen und aus einer Datei einlesen.

Wichtig hierbei ist, dass diese Testbench auch für die spätere Cosimulation des synthetisierten HDL-RTL-Codes verwendet wird. Daher ist es erforderlich, dass die Testbench, also die `main()`-Funktion, bei erfolgreichem Test eine 0 zurück gibt und im Fehlerfall eine 1 (oder ein anderer Wert ungleich 0). Dieser Rückgabewert wird dann in der Cosimulation verwendet, um einen erfolgreichen Abschluss der Cosimulation zu erkennen (`=0`). Da

man die Cosimulation in der Regel auch benutzt, um den RTL-Code des IP-Cores vor Einbau in das System zu verifizieren, ist das Entwickeln einer „self-checking“ Testbench zwingend.

- Drücken Sie nun in der Toolbar den Knopf Run C Simulation. Dies können Sie auch über das Menü erreichen: Project > Run C Simulation
- In der Dialog-Box drücken Sie auf OK.



```

Vivado HLS Console
WARNING: [SIM 211-51] HLS only supports CLANG compiler in Linux.
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
make: `csim.exe' is up to date.
Comparing against output data
*****
PASS: The output matches the golden output!
*****
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***** CSIM finish *****
Finished C simulation.
  
```

Abbildung 7: Ergebnis der C Simulation

Sie sollten nun ein Ergebnis wie in Abbildung 7 in der Console sehen. Im Rahmen der Labs sind die C-Codes für das Design und die Testbenches in der Regel vorgegeben. Bei einem späteren Projekt werden Sie diese vermutlich selbst entwickeln müssen. Sie haben in Vivado HLS einen Debugger zur Verfügung, der genauso funktioniert wie der Debugger, wenn Sie mit Eclipse C/C++-Software entwickeln. Um das Programm mit dem Debugger auszuführen, müssten Sie in der Dialog-Box den Haken bei Launch Debugger setzen, dann öffnet sich die Debug-Perspektive. Wir werden im Rahmen der Labs nicht auf die Arbeit mit dem Debugger weiter eingehen. Es ist selbstverständlich auch möglich, den C-Code für Design und Testbench zunächst in einer normalen Eclipse-Umgebung zu entwickeln und dann in Vivado HLS zu importieren.

## 2.3 Synthese des C-Codes

In diesem Schritt werden wir nun die eigentliche High-Level Synthese des C-Codes durchführen und den Code in eine RTL-Architektur umsetzen, welche dann später als VHDL-Code exportiert werden kann.

- Drücken Sie in der Toolbar auf den Knopf Run C Synthesis (oder: Solution > Run C Synthesis > Active Solution).
- Nach Abschluss der Synthese öffnet sich automatisch der Synthesis Report im Information Pane. Im Auxiliary Pane können Sie im Reiter Outline durch den Report navigieren. Wählen Sie Performance Estimates aus, wie in Abbildung 8 gezeigt und expandieren Sie unter Detail den Loop.

Die Ziel-Taktperiode wurde auf 10 ns gesetzt, abzüglich der Uncertainty-Marge verbleiben damit 8,75 ns als verfügbare Taktperiode. Die erreichte Taktperiode wird mit 8,43 ns angegeben, so dass die Vorgabe also erreicht werden kann. Hierbei ist zu bemerken, dass sich die minimale Taktperiodendauer später nach der Logiksynthese und nach Place&Route noch ändern kann. Der hier angegebene Wert ist nur eine Schätzung.

Was für die Analyse des Designs noch wichtig ist, ist die Latenz des Designs, welches Sie dem Abschnitt Latency entnehmen können und 78 Taktzyklen beträgt. Das Interval (= Initiation Interval II) ist die

**Performance Estimates**

**Timing (ns)**

**Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.43	1.25

**Latency (clock cycles)**

**Summary**

Latency	Interval
min	max
78	79

**Detail**

**Instance**

**Loop**

Loop Name	min	max	Iteration	Latency	Initiation	Interval	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	77	77	7	7	-	-	-	-	11	no

Abbildung 8: Performance Estimates

Anzahl der Taktzyklen bis zum Anlegen des nächsten Eingangswertes. Dieser Wert beträgt 79 Taktzyklen und daraus lässt sich entnehmen, dass das Design nicht mit Pipelining arbeitet.

In den Details kann man noch sehen, wodurch die Latenzzeiten entstehen. Öffnen Sie bitte hierzu auch den Quellcode des Designs (`fir.c` unter *Source*) und machen Sie sich dessen Funktionsweise klar. Die Schleife `Shift_Accum_Loop` wird 11 mal iteriert, dies wird als „Trip Count“ bezeichnet. Die Latenz einer Iteration beträgt 7 Taktzyklen, so dass insgesamt 77 Taktzyklen für die Abarbeitung der Schleife benötigt werden. Ferner wird ein Takt für den Eintritt in die Schleife benötigt und ein Takt für den Austritt aus der Schleife, was gleichzeitig auch der letzte Taktzyklus für die Abarbeitung des gesamten Designs ist.

**Utilization Estimates**

**Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	39
FIFO	-	-	-	-
Instance	-	4	0	0
Memory	0	-	64	6
Multiplexer	-	-	-	114
Register	-	-	179	-
<b>Total</b>	<b>0</b>	<b>4</b>	<b>243</b>	<b>159</b>
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

**Detail**

**Instance**

Instance	Module	BRAM_18K	DSP48E	FF	LUT
fir_mul_32s_32s_3bkb_U0	fir_mul_32s_32s_3bkb	0	4	0	0
<b>Total</b>		<b>0</b>	<b>4</b>	<b>0</b>	<b>0</b>

**DSP48**

**Memory**

Memory	Module	BRAM_18K	FF	LUT	Words	Bits	Banks	W*Bits*Banks
shift_reg_U	fir_shift_reg	0	64	6	11	32	1	352
<b>Total</b>		<b>0</b>	<b>64</b>	<b>6</b>	<b>11</b>	<b>32</b>	<b>1</b>	<b>352</b>

Abbildung 9: Utilization Estimates

- Wählen Sie als nächstes im Synthesis Report die *Utilization Estimates* aus, wie in Abbildung 9 gezeigt. In der Übersicht kann man die Gesamtzahl der benötigten Ressourcen sehen. Unter Detail kann man dann etwas genauer sehen, wofür die Ressourcen benötigt werden. Es werden 4 DSP-Blöcke für den Sub-Block (Instance) `fir_mul_32s_32s_3bkb_U0` verwendet. Dieser wird für die Implementierung der Multiplikation im C-Code benötigt. Vier DSPs sind notwendig, weil es sich um einen 32-Bit



Integer-Datentyp handelt. Der Datenspeicher `shift_reg[N]` des C-Codes wird mit Flipflops und LUTs implementiert (Abschnitt `Memory`).

- Sehen wir uns abschließend noch den Abschnitt `Interface` im Synthesis Report an. Da der IP Core getaktet wird gibt es einen Takt-Port `ap_clk` und einen Reset-Port `ap_rst`. Bei den Ports `ap_start`, `ap_done`, `ap_idle` und `ap_ready` handelt es sich um „Block Level“ Control-Ports, welche zur Steuerung des IP Cores benötigt werden. Der Port `y` implementiert den Ausgang des IP-Cores und damit das Argument `y` des C-Codes mit einer Bitbreite von 32. Über `y_ap_vld` wird die Gültigkeit der Daten signalisiert. Der Port `x` implementiert das Argument `x` des C-Codes und ist ebenfalls 32 Bit breit. Für das Argument `c[N]`, welches die Koeffizienten des FIR-Filters darstellt, wird ein Speicher-Interface implementiert, so dass an den IP-Core später ein Block-RAM angeschlossen werden kann, in welchem die Koeffizienten gespeichert sind.

Interface					
Summary					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
<code>ap_clk</code>	in	1	<code>ap_ctrl_hs</code>	<code>fir</code>	return value
<code>ap_rst</code>	in	1	<code>ap_ctrl_hs</code>	<code>fir</code>	return value
<code>ap_start</code>	in	1	<code>ap_ctrl_hs</code>	<code>fir</code>	return value
<code>ap_done</code>	out	1	<code>ap_ctrl_hs</code>	<code>fir</code>	return value
<code>ap_idle</code>	out	1	<code>ap_ctrl_hs</code>	<code>fir</code>	return value
<code>ap_ready</code>	out	1	<code>ap_ctrl_hs</code>	<code>fir</code>	return value
<code>y</code>	out	32	<code>ap_vld</code>	<code>y</code>	pointer
<code>y_ap_vld</code>	out	1	<code>ap_vld</code>	<code>y</code>	pointer
<code>c_address0</code>	out	4	<code>ap_memory</code>	<code>c</code>	array
<code>c_ce0</code>	out	1	<code>ap_memory</code>	<code>c</code>	array
<code>c_q0</code>	in	32	<code>ap_memory</code>	<code>c</code>	array
<code>x</code>	in	32	<code>ap_none</code>	<code>x</code>	scalar

Abbildung 10: Interface

## 2.4 RTL Verifikation: C/RTL Cosimulation

Der nächste Schritt nach der Synthese besteht darin, den durch die Synthese generierten VHDL- oder Verilog-Code zu verifizieren.

- Drücken Sie auf den Knopf `Run C/RTL Cosimulation` in der Toolbar (oder: `Solution > Run C/RTL Cosimulation`).
- Wählen Sie in der Cosimulation-Box (Abbildung 11) unter `RTL Selection`: `VHDL` aus und unter `Dump Trace`: `port` aus. Mit diesen Einstellungen werden für die Ports Traces aufgezeichnet, welche danach im Vivado Wave Viewer betrachtet werden können.
- Nach Abschluss der Simulation öffnet sich der Cosimulation Report. Hier sollten Sie die Werte für Latenz und Intervall sehen, die denen aus dem Synthese Report entsprechen sollten. Ferner sollten Sie in der Console die PASS-Meldung der Testbench sehen, welche den erfolgreichen Abschluss der Cosimulation zeigt.
- Während der Cosimulation passiert folgendes: Die C Testbench erzeugt die Eingangs-Stimuli für das VHDL-RTL-Design. Das VHDL-Design wird dann mit dem Vivado Simulator simuliert. Die Ausgangs-Vektoren des Designs werden wieder in der C Testbench eingelesen und dort mit den erwarteten Werten



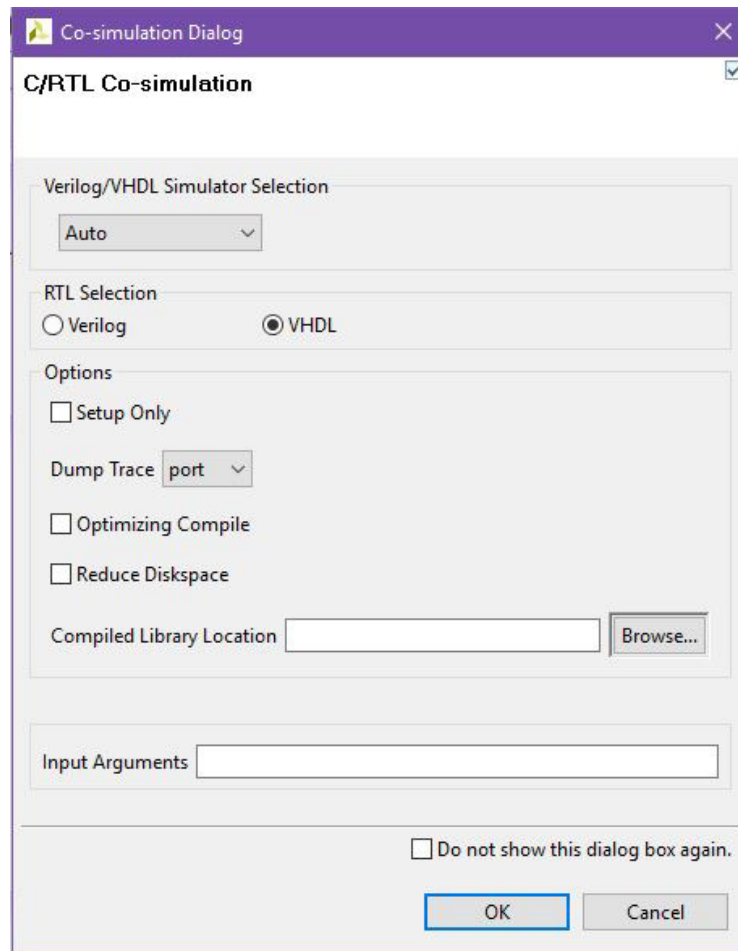


Abbildung 11: Cosimulation

verglichen, wie in der C Simulation. Es ist ausschließlich der von Ihnen codierte Vergleich in der Testbench und der Rückgabewert 0 der `main()`-Funktion, der zu der von der Cosimulation ausgegebenen Meldung führt:

```
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
```

- Drücken Sie nun noch den Knopf `Open Wave Viewer` in der Toolbar (oder: `Solution > Open Wave Viewer`). Der Vivado Wave Viewer wird gestartet und die Traces der Inputs und Outputs des Designs werden geladen. Sehen Sie sich die Traces an und messen Sie die Zeit zwischen zwei Pulsen des Signals `ap_done`, wie in Abbildung 12 zu sehen. Die Zeitdauer sollte 790 ns betragen und damit der Latenz des Designs von 79 Taktzyklen entsprechen.

## 2.5 Erzeugung des IP Cores: Export RTL

Der letzte Schritt im Entwurfsablauf ist der Export des Designs als IP Core. Es gibt verschiedene Möglichkeiten den IP Core zu exportieren. Wenn Sie mit Vivado den IP Core in einem System integrieren möchten, dann ist die Erzeugung eines Packages für den Vivado IP Katalog die richtige Lösung.

- Drücken Sie den Knopf `Export RTL` in der Toolbar (oder: `Solution > Export RTL`). In der Box sollte als Format `IP Catalog` voreingestellt sein. Wenn Sie auf `Configuration` drücken, dann

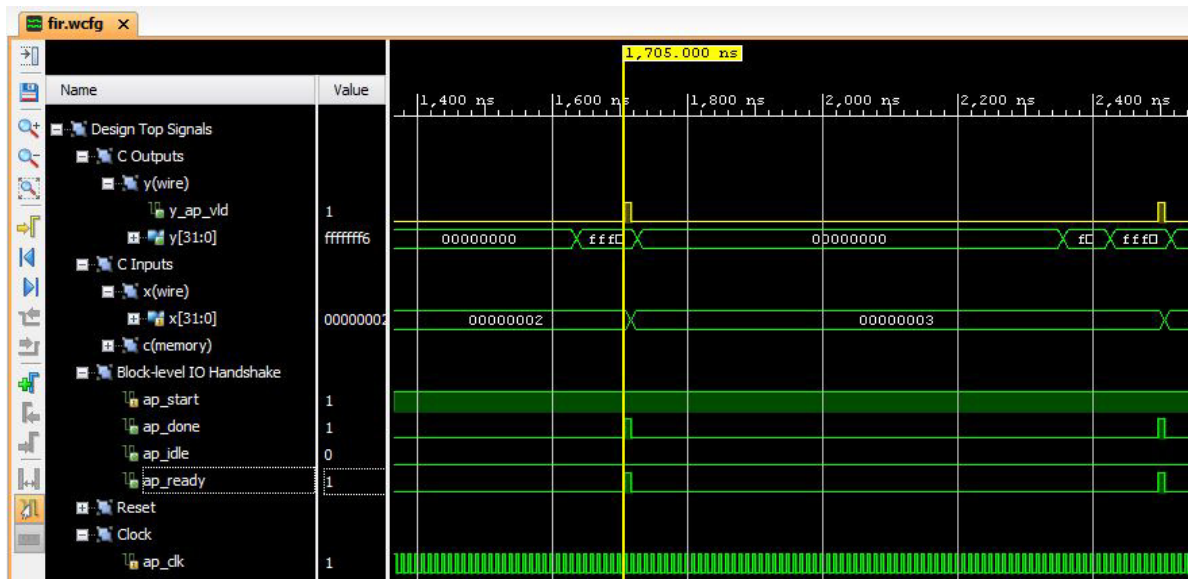


Abbildung 12: Vivado Waveform Viewer

könnten Sie noch zusätzliche Informationen für die spätere Anzeige des IP Cores im Vivado IP Katalog vorgeben. Unter dem Eintrag `Evaluate Generated RTL` könnte man einen Logiksyntheselauf oder ein Place&Route-Lauf starten. Dies ist allerdings an dieser Stelle nicht erforderlich. Drücken Sie auf OK.

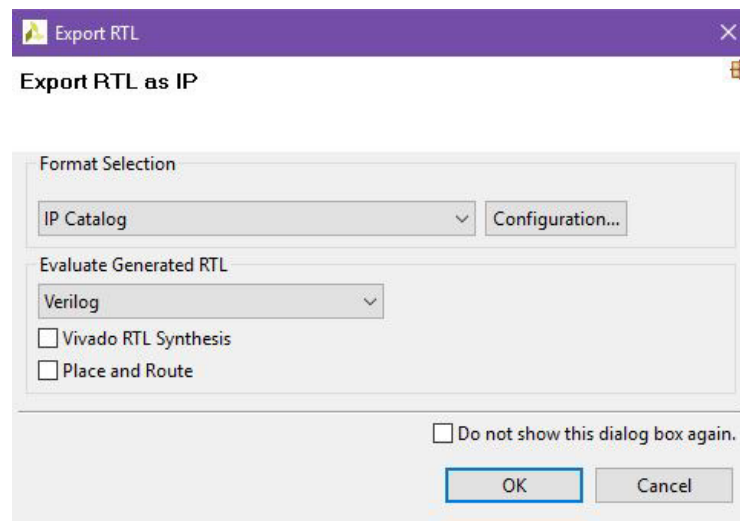


Abbildung 13: Export RTL

- Wenn Sie nun im Project Explorer `solution1 > impl > ip` aufklappen, dann sehen Sie die Dateien, die zum IP Core gehören. Beispielsweise können Sie dort auch die generierten Verilog- und VHDL-Quellen sehen, die dann beim Einbau des IP Cores in einem Systemdesign verwendet werden und dort per Logiksynthese in die Zieltechnologie umgesetzt werden. In Vivado können Sie dann das Verzeichnis `impl` zum IP Katalog hinzufügen und dann den IP Core in Ihr Systemdesign mit Hilfe des IP Integrators einbauen. Dies ist allerdings nicht Bestandteil dieser Labs.

Lassen Sie nun das Projekt in Vivado HLS geöffnet und machen Sie mit dem nächsten Lab 2 weiter.

### 3 Lab2: Verwendung von TCL-Skripten

Vivado HLS bietet die Möglichkeit, die im vorangegangenen Lab ausgeführten Schritte auch durch ein TCL-Skript ausführen zu lassen. Die TCL-Skripten werden bei Verwendung der GUI automatisch erstellt und können als Vorlage benutzt werden.

- Im Vivado HLS Projekt aus Lab1 öffnen Sie in der solution1 unter constraints das Skript `script.tcl`, wie in Abbildung 14 gezeigt. Sie sehen darin die TCL-Kommandos, die den Schritten entsprechen, die wir in Lab1 durchgeführt haben. Es empfiehlt sich gerade für das Aufsetzen des Projekts, also die Kommandos bis Zeile 13, ein TCL-Skript zu benutzen. Die Kommandos sollten selbst-erklärend sein und sind im User Guide zu Vivado HLS erläutert ([1]). Das Skript `directives.tcl` ist noch leer, da wir in Lab1 keine Direktiven verwendet haben.

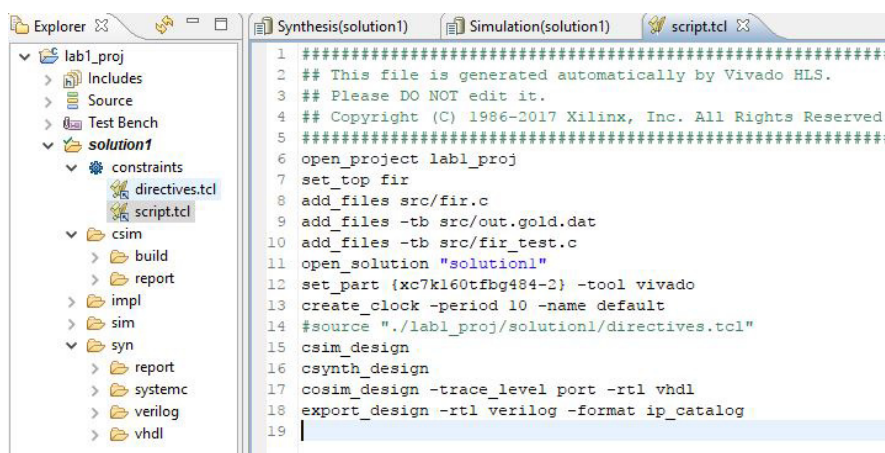


Abbildung 14: TCL Script

- Kopieren Sie nun mit Hilfe eines Datei-Explorers die Datei `\hls_labs\lab1\lab1_proj\solution1\script.tcl` in das Verzeichnis `\hls_labs\lab2` und öffnen Sie die Datei in einem Texteditor.

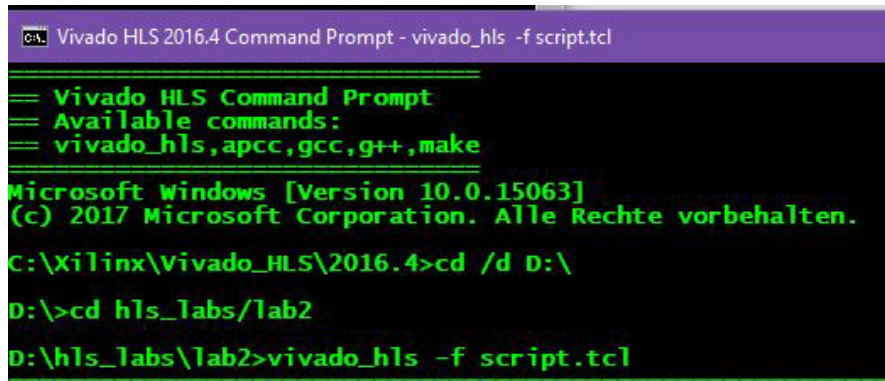
Listing 1: TCL Script

```
open_project -reset lab2_proj
set_top fir
add_files src/fir.c
add_files -tb src/out.gold.dat
add_files -tb src/fir_test.c
open_solution -reset "solution1"
set_part {xc7k160tfg484-2} -tool vivado
create_clock -period 10 -name default
#source "../lab1_proj/solution1/directives.tcl"
csim_design
csynth_design
cosim_design -trace_level port -rtl vhdl
export_design -rtl verilog -format ip_catalog
exit
```

Fügen Sie bei `open_project` und `open_solution` die `reset`-Option hinzu, ändern Sie den Namen des Projektes auf `lab2_proj` und fügen Sie am Ende den `exit`-Befehl hinzu, wie in Listing 1

gezeigt. Die `reset`-Option dient dazu, das Projekt oder die Solution bei einem erneuten Aufruf des Skripts zurückzusetzen und Daten zu überschreiben.

- Öffnen Sie nun den Vivado HLS Command Prompt: Xilinx Design Tools > Vivado 2016.4 > Vivado HLS > Vivado HLS 2016.4 Command Prompt



```
C:\> Vivado HLS 2016.4 Command Prompt - vivado_hls -f script.tcl

== Vivado HLS Command Prompt
== Available commands:
== vivado_hls, apcc, gcc, g++, make

Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. Alle Rechte vorbehalten.
C:\Xilinx\Vivado_HLS\2016.4>cd /d D:\
D:\>cd hls_labs/lab2
D:\hls_labs\lab2>vivado_hls -f script.tcl
```

Abbildung 15: Vivado HLS Command Prompt

- Als erstes müssen Sie das Verzeichnis auf Ihr Arbeitsverzeichnis setzen (`hls_labs/lab2`). Hierzu müssen Sie ggf. die Festplatte wechseln, die Kommandos dazu können Sie Abbildung 15 entnehmen. Führen Sie dann das Skript mit `vivado_hls -f script.tcl` aus. Sie sehen nun sämtliche Konsolenausgaben im Command Prompt Fenster. Warten Sie bis alle Kommandos abgearbeitet wurden.
- In der Vivado HLS GUI können Sie nun, sofern das Lab1 noch geöffnet ist, mit `File > Open Project` das mit dem TCL-Skript erzeugte Projekt öffnen. Klicken Sie in der Box auf `OK` und wählen Sie das Projektverzeichnis `hls_labs/lab2/lab2_proj` aus. Das Projekt wird nun geöffnet und Sie sollten die gleichen Ergebnisse und Dateien wie in Lab1 haben. Schließen Sie das Command Prompt Fenster und die GUI.

## 4 Lab3: I/O Interfaces und Optimierung des Durchsatzes

In diesem Lab werden wir für das Design aus Lab1 die I/O Interfaces explizit spezifizieren. Man könnte sich natürlich auf die Default-Einstellungen für die Interfaces verlassen, so wie wir es in Lab1 getan haben. Da der IP Core später in einem System eingebaut wird, sind die Default-Einstellungen aber möglicherweise nicht passend. Daher ist es gute Praxis, die Interfaces durch Direktiven explizit zu spezifizieren und sich nicht auf die Default-Einstellungen von Vivado HLS zu verlassen.

Weiterhin werden wir dann in diesem Lab die Performance des IP Cores genauer analysieren und optimieren.

### 4.1 Spezifikation von I/O Interfaces

- Öffnen Sie das Command Prompt Fenster und verzweigen Sie in das Verzeichnis `hls_labs/lab3`
- Führen Sie das TCL-Skript mit `vivado_hls -f run_hls.tcl` aus.
- Wenn das Skript beendet ist, dann können Sie vom Command Prompt aus die GUI starten und das Projekt öffnen mit: `vivado_hls -p lab3_proj`  
Ebenso können Sie die Vivado GUI wie in Lab1 starten und dann dort mit `Open Project` nach dem Projekt suchen (`hls_labs/lab3/lab3_proj`)
- In Lab1 wurden für die I/O-Ports Default-Einstellungen benutzt. In diesem Lab werden wir nun die I/O-Ports mit Hilfe von Direktiven genauer spezifizieren. Hierzu werden wir zunächst eine neue Solution anlegen und damit die Ergebnisse der ersten Solution zum Vergleich behalten können.
- Drücken Sie den Knopf `New Solution` in der Toolbar (oder: `Project > New Solution`). Belassen Sie in der Box die Voreinstellungen und drücken Sie `Finish`. Sie müssten nun im Project Explorer eine neue Solution `solution2` haben, diese ist fett markiert und damit aktiv. Sie könnten später auch eine ältere Solution wieder mit `Project > Set Active Solution` aktivieren.
- Öffnen Sie nun im Project Explorer unter `Source` die Datei `fir.c`. Wählen Sie im Auxiliary Pane den `Directive`-Tab aus, wie in Abbildung 16 gezeigt. Hier sind nun alle Objekte aus dem Quellcode gezeigt, auf die Direktiven angewendet werden können.

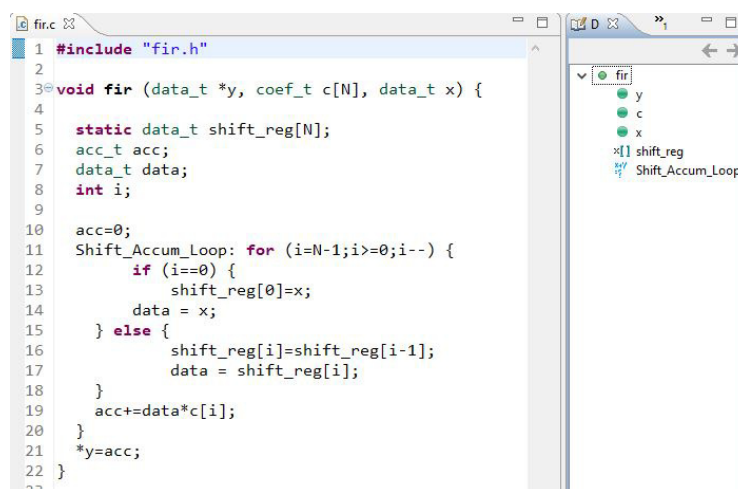


Abbildung 16: Quellcode und Direktiven

- Wählen Sie das Argument `c` aus, drücken Sie die rechte Maustaste und wählen `Insert Directive`. In der Box wählen Sie im Drop-Down-Menü ganz oben `RESOURCE` aus. Klicken Sie in das `core`-Feld und wählen Sie `RAM_1P_BRAM`. Wählen Sie als `Destination`: `Source File` und dann `OK`. Die Direktive wird nun als `Pragma` im Code eingebettet. Eine andere Möglichkeit wäre es, die Direktive in eine Datei zu schreiben. Da sich die Interfaces zwischen verschiedenen Solutions in der Regel nicht ändern, ist es sinnvoll diese Direktiven im Code einzubetten. Mit der Direktive spezifizieren wir, dass für Port `c` ein externes Single-Port-RAM (BlockRAM) später verwendet wird. Die Synthese wird nun dafür sorgen, dass ein entsprechendes Interface dafür erzeugt wird.
- Wählen Sie als nächstes das Argument `x` aus und wählen wieder `Insert Directive`. Wählen Sie hier nun `INTERFACE` und im Feld `mode` den Wert `ap_vld`. Wählen Sie unter `Destination` wieder `Source File`.
- Für das Argument `y` gehen Sie genauso wie für das Argument `x` vor und setzen `INTERFACE` ebenfalls auf `ap_vld`. Sie sollten nun die drei Direktiven im Quellcode haben, wie in Abbildung 17 gezeigt. Speichern Sie den Quellcode (Toolbar-Knopf `Save`). Falls die Direktiven nicht korrekt sind, wählen Sie die Direktive erneut aus und wählen über die rechte Maustaste `Modify Directive`. Prinzipiell ist es auch möglich, die Direktive im Quellcode zu ändern, hierzu müssen Sie aber die exakte Syntax kennen.

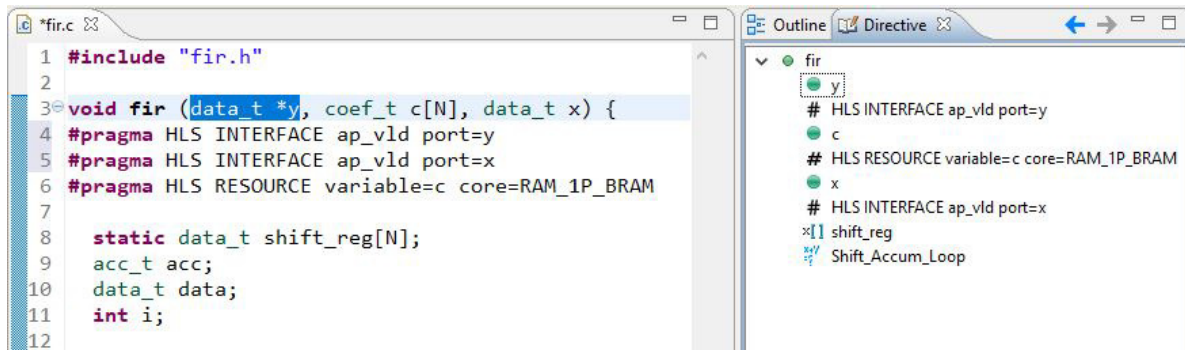


Abbildung 17: I/O Direktiven für das Beispiel

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_clk	in	1	ap_ctrl_hs	fir	return value	
ap_rst	in	1	ap_ctrl_hs	fir	return value	
ap_start	in	1	ap_ctrl_hs	fir	return value	
ap_done	out	1	ap_ctrl_hs	fir	return value	
ap_idle	out	1	ap_ctrl_hs	fir	return value	
ap_ready	out	1	ap_ctrl_hs	fir	return value	
y	out	32	ap_vld	y	pointer	
y_ap_vld	out	1	ap_vld	y	pointer	
c_address0	out	4	ap_memory	c	array	
c_ce0	out	1	ap_memory	c	array	
c_q0	in	32	ap_memory	c	array	
x	in	32	ap_vld	x	scalar	
x_ap_vld	in	1	ap_vld	x	scalar	

Abbildung 18: Interface des Beispiels

- Starten Sie nun einen Syntheselauf durch `Run C Synthesis`. Wenn Sie im Synthese Report nun das Interface ansehen, sollte dies wie in Abbildung 18 aussehen: Für den Port `c` wurde ein Memory-Interface implementiert und für die Ports `x` und `y` wurde jeweils ein Valid-Signal hinzugefügt.



- Bevor wir das Design optimieren können, müssen wir es zunächst analysieren. Öffnen Sie die *Analysis Perspective* (rechts oben). Mit der *Analysis Perspective* kann die Performance und der Ressourcenbedarf untersucht werden. Das kleine Fenster links oben dient der Navigation, im kleinen Fenster darunter findet man dann eine Übersicht über die Performance und die Ressourcen. Im Information Pane finden Sie für die Analyse der Performance eine Darstellung wie in Abbildung 19, mit der wir uns in der Folge etwas näher auseinandersetzen möchten. Klappen Sie hierzu *Shift\_Accum\_Loop* auf, so dass Sie die Ansicht wie Abbildung 19 haben.

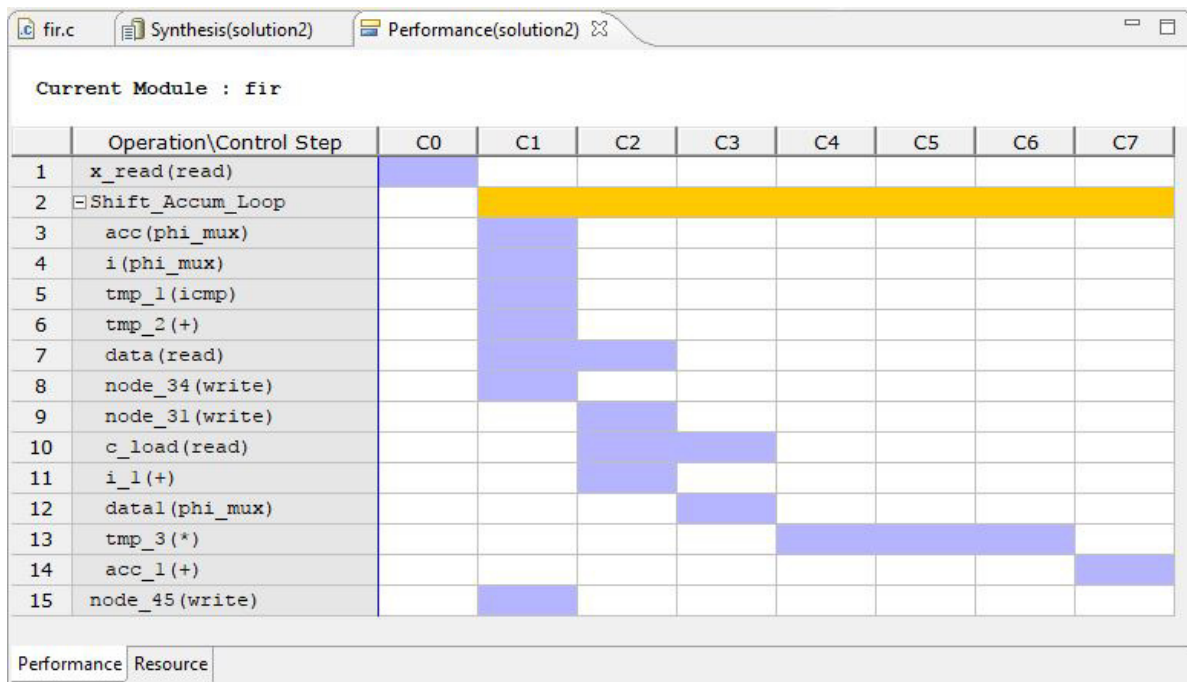


Abbildung 19: Performance Fenster

- In der linken Spalte von Abbildung 19 sind die Operationen in diesem Modul zu sehen (das Modul ist hier das Design *fir*), welche im RTL-Code vorhanden sind. Wenn Sie mit der rechten Maustaste auf den zugehörigen Balken im Diagramm klicken, dann erhalten Sie die Kreuzreferenz zum C-Quellcode. In der obersten Zeile (Control Step) sind die Zustände zu sehen, die das „Scheduling“ darstellen und damit zeigen, wann welche Operation ausgeführt wird. Dies entspricht im Wesentlichen auch den Zuständen einer internen FSM („Finite State Machine“) und damit letztlich auch den Taktzyklen.
- Das Modul/Design startet im Zyklus C0, hier wird der Eingangs-Port *x* gelesen. Schleifen werden durch einen orangenen Balken dargestellt, den man aufklappen kann und dann werden die Operationen innerhalb der Schleife gezeigt. Wir wir schon in Lab1 gesehen haben, benötigt die Schleife 7 Taktzyklen für die Abarbeitung. Da die Schleife 11 mal iteriert wird, wird der orangene Schleifen-Balken 11 mal ausgeführt, so dass wir insgesamt 77 Taktzyklen benötigen. Im ersten Zyklus der Schleife wird das Abbruchkriterium der Schleife geprüft. Erkennbar ist auch das Lesen des Koeffizienten *c[i]* aus dem BlockRAM, was zwei Taktzyklen C2/C3 benötigt (Zeile 10, *c\_load*). In den Zyklen C4 bis C6 erfolgt die Multiplikation des Koeffizienten mit dem Datum *data* (Zeile 13). Es sind drei Taktzyklen hierfür erforderlich, da der Multiplizierer des DSP-Blocks im Pipelining betrieben wird. Im letzten Takt C7 erfolgt die Akkumulation (Variable *acc* im C-Code, Zeile 14). Nachdem alle 11 Schleifendurchläufe erfolgt sind, wird die Schleife im Zustand C1 (Prüfung des Abbruchkriteriums) verlassen und der Ausgangs-Port *y* beschrieben (Zeile 15), dies ist der letzte Taktzyklus in der Abarbeitung des Moduls.



## 4.2 Optimierung des Durchsatzes

- Wir können nun die Ausführungszeit des Designs optimieren, indem wir die Schleife entrollen. Legen Sie hierzu eine neue Solution an: `Project > New Solution`. Belassen Sie alle Einstellungen in der Box. Sie könnten an dieser Stelle entscheiden, ob Sie die Direktiven aus der aktuellen Solution übernehmen möchten.
- Öffnen Sie wieder den Quellcode `fir.c` und den Directive-Tab im Auxiliary Pane. Wählen Sie die Schleife mit der rechten Maustaste aus und wählen Sie `Insert Directive` aus und dann die Direktive `UNROLL`. Als Destination wählen Sie hier nun `Directive File` aus. Wenn man optimiert, will man in der Regel mehrere Lösungen mit unterschiedlichen Direktiven ausprobieren. Daher ist es sinnvoller, solche Direktiven eher in der Datei abzuspeichern als im Quellcode, da Quellcode-Direktiven in jeder Solution angewendet werden. Die Direktive wurde in der Datei `directives.tcl` abgespeichert. Sie finden diese unter `solution3 > constraints` im Project Explorer.
- Führen Sie nun wieder die Synthese durch (`Run C Synthesis`).
- Nach Abschluss der Synthese können Sie die Ergebnisse von `solution2` und `solution3` vergleichen: `Project > Compare Reports`  
Wählen Sie nur die Solutions 2 und 3 für den Vergleich aus.

### Performance Estimates

Timing (ns)

Clock		solution3	solution2
ap_clk	Target	10.00	10.00
	Estimated	8.43	8.43

Latency (clock cycles)

Latency	min	solution3	solution2
	max	15	78
	max	15	78
Interval	min	16	79
	max	16	79

### Utilization Estimates

	solution3	solution2
BRAM_18K	0	0
DSP48E	44	4
FF	977	276
LUT	254	194

Abbildung 20: Vergleich der Solutions

- Im Vergleich in Abbildung 20 zeigt sich, dass wir für Solution 3 nur 16 Taktschritte für die Ausführung benötigen. Da die Schleife entrollt wird, wird der Schleifenkörper 11 mal implementiert, daher werden auch 44 DSPs für die 11 Multiplizierer benötigt.
- Untersuchen Sie nun, wie die 16 Taktzyklen für Solution 3 zustande kommen, indem Sie wieder die Analysis-Perspective öffnen. Auf den ersten Blick überraschend ist vielleicht die Tatsache, dass nicht alle 11 Multiplikationen parallel ausgeführt werden können. Der Flaschenhals ist hier das Lesen der Koeffizienten `c[i]` aus dem externen BlockRAM. Da dieses nur einen Port aufweist, können wir mit jeder Taktflanke nur einen Koeffizienten lesen (`c_load` bis `c_load_10`). Im Diagramm werden die Zyklen C0 bis C11 benötigt, um alle 11 Koeffizienten auslesen zu können. Dazwischen finden die Multiplikationen und ein Teil der Akkumulationen statt. Die letzte Multiplikation kann dann erst in den Zyklen C12 bis C14 stattfinden. Im letzten Zyklus finden die restlichen Akkumulationen statt und das Schreiben des

Ergebnisses. Dieses Beispiel zeigt, dass die Parallelisierung von Schleifen zum Einen einen hohen Ressourcenaufwand bedeuten kann und zum Anderen möglicherweise durch Speicher-Flaschenhälse nicht sehr effektiv ist.

## 5 Lab4: Interface Synthese

In Lab1 hatten wir uns die Interfaces angesehen, die per Default erzeugt werden. Das Block-Level I/O Protokoll erzeugt dabei die Signale `ap_start`, `ap_done`, `ap_idle` und `ap_ready`, welche der Steuerung des IP Cores dienen. Dieses Protokoll nennt sich `ap_ctrl_hs`, wobei `hs` für „Handshake“ steht. Es gibt noch die Möglichkeit kein Protokoll zu verwenden (`ap_none`) oder das Protokoll `ap_ctrl_chain`, worauf wir hier nicht näher eingehen. Das gebräuchteste Protokoll dürfte sicher `ap_ctrl_hs` sein, so dass wir in der Regel für das Block-Level-Protokoll den Default benutzen und dieses nicht explizit per Direktive spezifizieren müssen.

Bei den Port-Level I/O Protokollen hatten wir in Lab3 schon das Protokoll `ap_vld` kennengelernt, was einem Eingangs- oder Ausgangs-Port einen zusätzlichen „Valid“-Port zuordnet. In diesem Lab möchten wir anhand eines weiteren Beispiels noch etwas genauer zeigen, wie die Interfaces die Optimierung eines IP Cores beeinflussen.

### 5.1 Verwendung von RAM-Ports

- Öffnen Sie das Command Prompt Fenster und verzweigen Sie in das Verzeichnis `hls_labs/lab4`
- Führen Sie das TCL-Skript mit `vivado_hls -f run_hls.tcl` aus. Öffnen Sie anschließend das Projekt in der Vivado GUI.
- Machen Sie sich die Funktion des Designs klar, indem Sie die Quellcodes für das Design und die Testbench studieren. Führen Sie dann die C Simulation durch.
- Führen Sie nun einen Syntheselauf durch. Wenn Sie sich die Interfaces im Synthese Report ansehen, können Sie erkennen, dass die beiden Array-Argumente in zwei Speicherschnittstellen vom Typ `ap_memory` umgesetzt wurden. Hierbei wird wieder unterstellt, dass zwei externe BlockRAMs später angeschlossen werden, wie schon in den Labs 1 und 3 diskutiert wurde. Beachten Sie die Breite der Daten-Ports von 16 Bit, die sich aus dem Datentyp ergibt. Die Breite der Adress-Ports ergibt sich aus der Größe der Felder. Für `d_o` gibt es einen zusätzlichen „Write Enable“-Port, da die Daten geschrieben werden. Der Default ist hier die Annahme, dass ein Single-Port-RAM angeschlossen wird. In den folgenden Schritten werden wir Dual-Port-RAMs und FIFOs als Alternative verwenden.

Interface					
Summary					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
<code>ap_clk</code>	in	1	<code>ap_ctrl_hs</code>	accumulator	return value
<code>ap_rst</code>	in	1	<code>ap_ctrl_hs</code>	accumulator	return value
<code>ap_start</code>	in	1	<code>ap_ctrl_hs</code>	accumulator	return value
<code>ap_done</code>	out	1	<code>ap_ctrl_hs</code>	accumulator	return value
<code>ap_idle</code>	out	1	<code>ap_ctrl_hs</code>	accumulator	return value
<code>ap_ready</code>	out	1	<code>ap_ctrl_hs</code>	accumulator	return value
<code>d_o_address0</code>	out	5	<code>ap_memory</code>	<code>d_o</code>	array
<code>d_o_ce0</code>	out	1	<code>ap_memory</code>	<code>d_o</code>	array
<code>d_o_we0</code>	out	1	<code>ap_memory</code>	<code>d_o</code>	array
<code>d_o_d0</code>	out	16	<code>ap_memory</code>	<code>d_o</code>	array
<code>d_i_address0</code>	out	5	<code>ap_memory</code>	<code>d_i</code>	array
<code>d_i_ce0</code>	out	1	<code>ap_memory</code>	<code>d_i</code>	array
<code>d_i_q0</code>	in	16	<code>ap_memory</code>	<code>d_i</code>	array

Abbildung 21: Interfaces

## 5.2 Verwendung von Dual-Port-RAM und FIFO

Wie wir schon in Lab3 gesehen haben, kann die Speicherschnittstelle ein Flaschenhals sein, wenn das Design durch Entrollen einer Schleife optimiert werden soll. In der ersten Solution in diesem Lab hatten wir allerdings die Schleife nicht entrollt, so dass ein Single-Port-RAM ausreichend war. Vivado HLS ist in der Lage, diese Zusammenhänge zu analysieren und ggf. eine Dual-Port-Speicherschnittstelle einzubauen, falls dies notwendig wird. Auch das Gegenteil ist möglich: Wenn Sie eine Dual-Port-Schnittstelle per Direktive spezifizieren, diese aber gar keinen Vorteil bietet, weil Sie im Fall der nicht-entrollten Schleife pro Taktschritt nur ein Datum lesen oder schreiben, so kann Vivado HLS die Direktive überschreiben und ein Single-Port-Interface einbauen. Um die Verwendung von mehreren Ports zu demonstrieren, müssen wir also zunächst die Schleife entrollen.

- Legen Sie eine neue Solution `solution2` an (Project > New Solution). Belassen Sie alle Voreinstellungen der Box.
- Setzen Sie die Direktive `UNROLL` für die Schleife `L1` aus dem Quellcode `accumulator.c`, analog zur Vorgehensweise in Lab3. Speichern Sie alle Direktiven in diesem Lab in der Datei `ab` (Destination > Directive File).
- Spezifizieren Sie ein Dual-Port-RAM für das Argument `d_i`: Wählen Sie bei der Direktive `RESOURCE` im Feld `core` den Wert `RAM_2P_BRAM` aus.

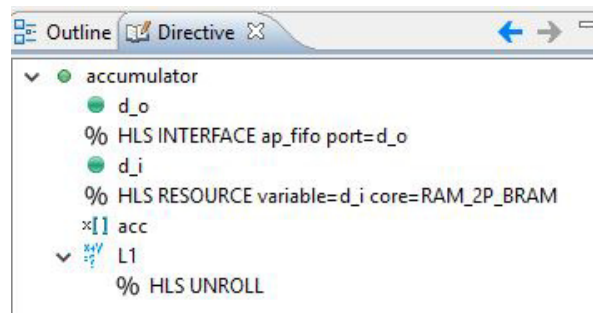


Abbildung 22: Direktiven

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_clk	in	1	ap_ctrl_hs	accumulator	return value	
ap_rst	in	1	ap_ctrl_hs	accumulator	return value	
ap_start	in	1	ap_ctrl_hs	accumulator	return value	
ap_done	out	1	ap_ctrl_hs	accumulator	return value	
ap_idle	out	1	ap_ctrl_hs	accumulator	return value	
ap_ready	out	1	ap_ctrl_hs	accumulator	return value	
d_o_din	out	16	ap_fifo	d_o	pointer	
d_o_full_n	in	1	ap_fifo	d_o	pointer	
d_o_write	out	1	ap_fifo	d_o	pointer	
d_i_address0	out	5	ap_memory	d_i	array	
d_i_ce0	out	1	ap_memory	d_i	array	
d_i_q0	in	16	ap_memory	d_i	array	
d_i_address1	out	5	ap_memory	d_i	array	
d_i_ce1	out	1	ap_memory	d_i	array	
d_i_q1	in	16	ap_memory	d_i	array	

Abbildung 23: Interface

- Spezifizieren Sie für das Argument `d_o` einen FIFO-Port: Wählen Sie bei der Direktive `INTERFACE` im Feld `mode` den Wert `ap_fifo` aus. Die Direktiven sollten nun wie in Abbildung 22 aussehen.
- Starten Sie einen Syntheselauf. In Abbildung 23 können Sie die erzeugten Interfaces erkennen. Es ist ersichtlich, dass für `d_i` nun zwei Speicher-Interfaces vorhanden sind. Hier kann man nun außen ein Dual-Port-RAM anschließen und an den FIFO-Port ein entsprechendes FIFO.
- Wenn Sie die beiden Solutions vergleichen (`Project > Compare Reports`), dann werden Sie sehen, dass die Latenz von 129 Takten auf 33 reduziert werden konnte.
- Sehen Sie sich die Performance Analyse in der Analysis Perspective an. Es ist erkennbar, dass für `d_i` tatsächlich zwei Speicherzugriffe pro Taktdurchgeführt werden. Allerdings ist nun die FIFO-Schnittstelle der Flaschenhals, da pro Taktschritt nur ein Datum geschrieben werden kann (Dies sind die Schreiboperationen `node_xxx(write)`). Im nächsten Abschnitt werden wir dieses Problem noch lösen.

### 5.3 Partitionierung der Felder

Um einen höheren Datendurchsatz zu erreichen und damit die Performance beim Entrollen der Schleife zu erhöhen, können wir die Argument-Felder partitionieren. Dabei entstehen dann mehrere I/O Interfaces.

- Legen Sie eine neue Solution `solution3` an. Kopieren Sie dabei die Direktiven aus `solution2`.
- Öffnen Sie wieder den Quellcode des Designs und wählen Sie im Directives-Tab das Argument `d_o` mit der rechten Maustaste aus, um eine weitere Direktive hinzuzufügen. Wählen Sie `ARRAY_PARTITION` aus und setzen Sie das Feld `type` auf `block`. Tragen Sie im Feld `factor` den Wert 4 ein.
- Wiederholen Sie dies für das Argument `d_i`, tragen Sie aber als Faktor den Wert 2 ein. Abbildung 24 zeigt, wie die Direktiven nun aussehen sollten.

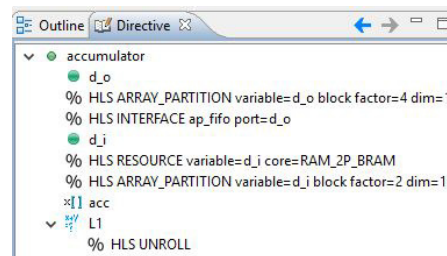


Abbildung 24: Direktiven

- Starten Sie einen Syntheselauf. Sie sollten nun die Interfaces wie in Abbildung 25 erhalten. Nun sind 4 FIFO-Interfaces vorhanden und 4 BlockRAM-Interfaces. Letztere bestehen im Grunde aus zwei Dual-Port-Interfaces, daher wurde bei der Partitionierung der Faktor 2 eingesetzt.
- Vergleichen Sie wieder die Performance aller drei bisher synthetisierten Solutions. `solution3` ist die Lösung mit der höchsten Performance, wobei der Ressourcenaufwand gegenüber `solution2` sogar gesunken ist. Allerdings müssen wir nun am IP Core zwei DP-RAMs und vier FIFOs anschließen. Sehen Sie sich auch die Performance Analysis wieder dazu an.
- Für den Port `d_i` haben wir hier eine `RESOURCE`-Direktive benutzt statt einer `INTERFACE`-Direktive. Mit einer `RESOURCE`-Direktive kann vorgegeben werden, welche Bibliothekselemente zur Umsetzung benutzt werden sollen. Man könnte damit beispielsweise auch die Umsetzung von lokalen Feld-Variablen

Interface					
Summary					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	accumulator	return value
ap_rst	in	1	ap_ctrl_hs	accumulator	return value
ap_start	in	1	ap_ctrl_hs	accumulator	return value
ap_done	out	1	ap_ctrl_hs	accumulator	return value
ap_idle	out	1	ap_ctrl_hs	accumulator	return value
ap_ready	out	1	ap_ctrl_hs	accumulator	return value
d_o_0_din	out	16	ap_fifo	d_o_0	pointer
d_o_0_full_n	in	1	ap_fifo	d_o_0	pointer
d_o_0_write	out	1	ap_fifo	d_o_0	pointer
d_o_1_din	out	16	ap_fifo	d_o_1	pointer
d_o_1_full_n	in	1	ap_fifo	d_o_1	pointer
d_o_1_write	out	1	ap_fifo	d_o_1	pointer
d_o_2_din	out	16	ap_fifo	d_o_2	pointer
d_o_2_full_n	in	1	ap_fifo	d_o_2	pointer
d_o_2_write	out	1	ap_fifo	d_o_2	pointer
d_o_3_din	out	16	ap_fifo	d_o_3	pointer
d_o_3_full_n	in	1	ap_fifo	d_o_3	pointer
d_o_3_write	out	1	ap_fifo	d_o_3	pointer
d_i_0_address0	out	4	ap_memory	d_i_0	array
d_i_0_ce0	out	1	ap_memory	d_i_0	array
d_i_0_q0	in	16	ap_memory	d_i_0	array
d_i_0_address1	out	4	ap_memory	d_i_0	array
d_i_0_ce1	out	1	ap_memory	d_i_0	array
d_i_0_q1	in	16	ap_memory	d_i_0	array
d_i_1_address0	out	4	ap_memory	d_i_1	array
d_i_1_ce0	out	1	ap_memory	d_i_1	array
d_i_1_q0	in	16	ap_memory	d_i_1	array
d_i_1_address1	out	4	ap_memory	d_i_1	array
d_i_1_ce1	out	1	ap_memory	d_i_1	array
d_i_1_q1	in	16	ap_memory	d_i_1	array

Abbildung 25: Interface

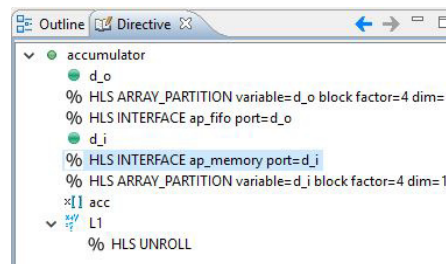


Abbildung 26: Alternative Direktiven

in BlockRAMs oder LUT-RAMs steuern. Da wir die RESOURCE-Direktive auf ein Funktionsargument anwenden, wird damit spezifiziert, wie ein externes Feld umgesetzt werden soll und damit indirekt auch die Interface-Implementierung. Man könnte die gleiche Umsetzung der Interfaces auch mit den Direktiven aus Abbildung 26 erreichen, indem man für `d_i` die INTERFACE-Direktive verwendet und den Typ auf `ap_memory` setzt. Dann müsste man bei der Partitionierung des Feldes allerdings den Wert 4 vorgeben. Dies führt dann zum gleichen Interface wie in Abbildung 25.

## 5.4 AXI-Stream und AXI-Lite Interface

Im letzten Teil dieses Labs werden wir aus den Argumenten AXI-Stream-Interfaces machen, so dass wir den IP Core in einem System mit anderen Komponenten verschalten können, welche ebenfalls AXI-Stream-Interfaces haben. Ferner werden wir für den IP Core auch ein AXI-Lite-Interface implementieren, so dass wir den IP Core von einem Prozessorsystem aus kontrollieren können.

- Legen Sie eine neue Solution `solution4` an. Dieses Mal entfernen Sie bitte den Haken aus Copy



`directives` . . . , da wir die alten Direktiven nicht benötigen. Öffnen Sie wieder den Quellcode, so dass die Direktiven spezifiziert werden können.

- Führen Sie diesen Schritt für beide Argument `d_o` und `d_i` aus: Wählen Sie das Argument aus und wählen Sie `Insert Directive`. Wählen Sie `INTERFACE` und für `mode` den Wert `axis`.
- Wählen Sie die Schleife `L1` aus und wählen Sie `Insert Directive`. Wählen Sie `PIPELINE` aus. Lassen Sie alle Felder leer. Wir werden die Schleife im Pipelining betreiben, um einen höheren Durchsatz zu erreichen.
- Wählen Sie die Toplevel-Funktion `accumulator` aus und mit der rechten Maustaste `Insert Directives`. Geben Sie die Direktive `INTERFACE` an und wählen Sie `s_axilite` aus. Alle anderen Felder bleiben leer. Damit werden nun die Block-Level-Ports nicht mehr als Ports herausgeführt, sondern sind als entsprechende Bits in einem Steuerregister des IP-Cores vorhanden. Das Register kann über die AXI-Lite-Schnittstelle per Memory-Mapped-I/O erreicht werden. Die Direktiven sollten nun wie in Abbildung 27 sein.

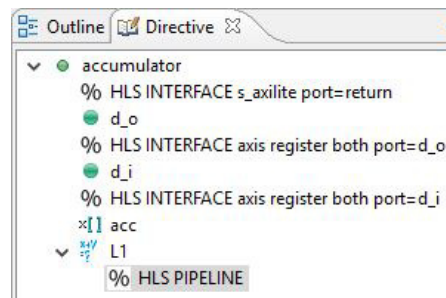


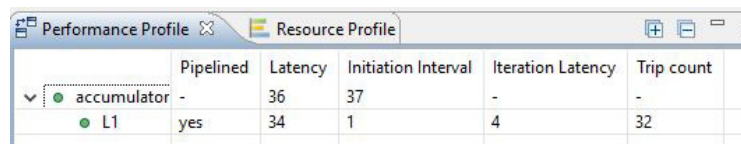
Abbildung 27: Direktiven

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
s_axi_AXILiteS_AWVALID	in	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_AWREADY	out	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_AWADDR	in	4	s_axi	AXILiteS	return void	
s_axi_AXILiteS_WVALID	in	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_WREADY	out	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_WDATA	in	32	s_axi	AXILiteS	return void	
s_axi_AXILiteS_WSTRB	in	4	s_axi	AXILiteS	return void	
s_axi_AXILiteS_ARVALID	in	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_ARREADY	out	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_ARADDR	in	4	s_axi	AXILiteS	return void	
s_axi_AXILiteS_RVALID	out	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_RREADY	in	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_RDATA	out	32	s_axi	AXILiteS	return void	
s_axi_AXILiteS_RRESP	out	2	s_axi	AXILiteS	return void	
s_axi_AXILiteS_BVALID	out	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_BREADY	in	1	s_axi	AXILiteS	return void	
s_axi_AXILiteS_BRESP	out	2	s_axi	AXILiteS	return void	
ap_clk	in	1	ap_ctrl_hs	accumulator	return value	
ap_rst_n	in	1	ap_ctrl_hs	accumulator	return value	
interrupt	out	1	ap_ctrl_hs	accumulator	return value	
d_o_TDATA	out	16	axis	d_o	pointer	
d_o_TVALID	out	1	axis	d_o	pointer	
d_o_TREADY	in	1	axis	d_o	pointer	
d_i_TDATA	in	16	axis	d_i	pointer	
d_i_TVALID	in	1	axis	d_i	pointer	
d_i_TREADY	out	1	axis	d_i	pointer	

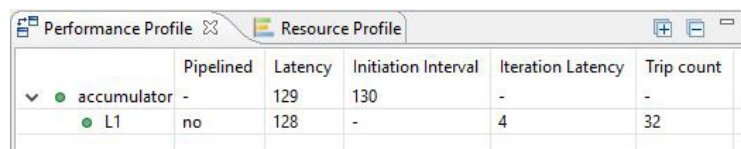
Abbildung 28: Interfaces



- Starten Sie einen Syntheselauf. Die Interfaces sollten nun wie in Abbildung 28 aussehen. Im oberen Teil sehen Sie die AXI-Lite-Schnittstelle (`s_axi_xxx`). Nach Clock und Reset ist ein Interrupt-Port zu sehen: Der IP Core kann nun am Ende des Aufrufs einen Interrupt generieren. Danach kommen die beiden AXI-Stream-Interfaces für `d_i` und `d_o`. Diese bestehen jeweils aus den 16-Bit breiten Daten und den Handshake-Ports `TVALID` und `TREADY`.
- Öffnen Sie wieder die Analysis Perspective. Im kleinen Fenster links unten sollten Sie die Ergebnisse nach Abbildung 29 sehen. Die Schleife wird nun im Pipelining betrieben, zu erkennen am Eintrag `yes` in der Spalte `Pipelined`. Vergleichen Sie dies mit dem Performance Profil für `solution1` in Abbildung 30. Durch das Pipelining kann nun die nächste Iteration einen Takt nach der vorhergehenden Iteration gestartet werden und wir erreichen einen erheblich höheren Durchsatz, verglichen mit `solution1` ohne Pipelining. Wenn Sie nun wieder alle vier Solutions miteinander vergleichen, so werden Sie sehen, dass `solution4` einen Durchsatz hat, der fast so gut ist, wie `solution2`, jedoch einen erheblich geringeren Ressourcenaufwand benötigt. Ein Vorteil gegenüber `solution3` ist, dass wir jeweils nur ein Interface benötigen und damit auch den äußeren Aufwand senken. Das Pipelining von Schleifen stellt eine sehr effektive Maßnahme dar, da der Durchsatz bei moderatem Ressourcenmehraufwand erheblich erhöht werden kann.



	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
accumulator	-	36	37	-	-
L1	yes	34	1	4	32

Abbildung 29: Performance Profile `solution4`


	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
accumulator	-	129	130	-	-
L1	no	128	-	4	32

Abbildung 30: Performance Profile `solution1`

- Starten Sie zum Abschluss die Cosimulation: Drücken Sie `Run C/RTL Cosimulation` in der Toolbar (oder: `Solution > Run C/RTL Cosimulation`). Wählen Sie in der Cosimulation-Box (Abbildung 11) unter `RTL Selection: VHDL` aus und unter `Dump Trace: port` aus. Mit diesen Einstellungen werden für die Ports Traces aufgezeichnet, welche danach im Vivado Wave Viewer betrachtet werden können.
- Drücken Sie nun noch den Knopf `Open Wave Viewer` in der Toolbar (oder: `Solution > Open Wave Viewer`). Der Vivado Wave Viewer wird gestartet und die Traces der Inputs und Outputs des Designs werden geladen. In der Wave View können Sie dann sehen, wie die Eingangsdaten über das AXI-Stream-Interface gelesen werden und die Ausgangsdaten über das zweite AXI-Stream-Interface geschrieben werden. In Abbildung 31 kann man sehen, dass die Daten tatsächlich ohne Pause „gestreamt“ werden und wir somit den maximalen Durchsatz erhalten.

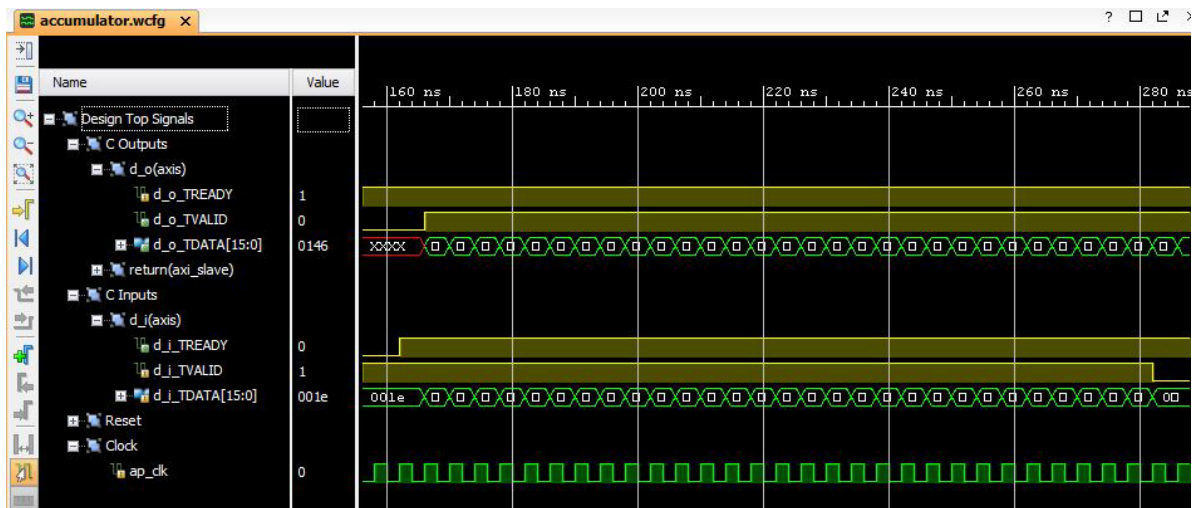


Abbildung 31: Co-Simulation

## 6 Lab5: Design Optimierung

In diesem Lab werden wir uns die Optimierung eines Designs genauer ansehen und hier insbesondere auf die Optimierung der Schleifen eingehen, da dies ein häufiger Fall darstellt. Als Beispiel nehmen wir eine Matrix-Multiplikation, welche mit drei geschachtelten Schleifen implementiert ist. Nach einer Analyse des Default-Designs, welches also ohne Vorgabe von speziellen Direktiven erzeugt wird, werden wir durch Anwendung der PIPELINE-Direktive das Design optimieren, wobei dann damit auch eine Entrollung der Schleifen verbunden ist. Es empfiehlt sich, bei der Optimierung eines Designs mit geschachtelten Schleifen, so wie in diesem Lab vorzugehen: Man fängt in der inneren Schleife an, die PIPELINE-Direktive anzuwenden und generiert dann mehrere Solutions, indem man sukzessive die PIPELINE-Direktive auf die nächste Schleifenebene anwendet, wobei dann die darunterliegende Schleifenebene entrollt wird. Dabei wird man dann die Performance sukzessive steigern können, wobei der Ressourcenaufwand ebenfalls ansteigt.

### 6.1 Analyse des Default-Designs

- Öffnen Sie das Command Prompt Fenster und verzweigen Sie in das Verzeichnis `hls_labs/lab5`.
- Führen Sie das TCL-Skript mit `vivado_hls -f run_hls.tcl` aus. Öffnen Sie anschließend das Projekt in der Vivado GUI.
- Machen Sie sich die Funktion des Designs klar, indem Sie die Quellcodes für das Design und die Testbench studieren. Führen Sie dann die C Simulation durch.
- Führen Sie dann wieder einen Syntheselauf durch. Überprüfen Sie zunächst die erzeugten Interfaces im Synthese Report. Neben den Block-Level Ports werden für die Matrizen `a`, `b` und `res` jeweils wieder ein Speicher-Interface vom Typ `ap_memory` erzeugt (vgl. Lab4).

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	12.00	8.18	1.50

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
79	79	80	80	none

Detail

Instance

Loop

Loop Name	Latency		Iteration	Latency	Initiation Interval		Trip Count	Pipelined
	min	max			achieved	target		
- Row	78	78		26	-	-	3	no
+ Col	24	24		8	-	-	3	no
++ Product	6	6		2	-	-	3	no

Abbildung 32: Performance

- Sehen Sie sich im Synthese Report nun die Performance an, wie in Abbildung 32 gezeigt. Für eine genauere Analyse können Sie auch wieder in der Analysis Perspective das Scheduling Diagramm ansehen. Die Schleife `Product` benötigt zwei Taktzyklen für eine Iteration. Im Scheduling Diagramm kann man sehen, dass jeweils ein Element der Matrix `a` und `b` aus dem Speicher geholt werden muss, die Werte werden multipliziert und dann akkumuliert. Der Speicherzugriff führt dazu, dass wir zwei Taktzyklen benötigen. Da die Schleife `Product` einen Trip Count von 3 hat, ergibt sich eine Latenz von 6 Taktzyklen. Die Schleife `Col` ruft die Schleife `Product` auf und für Eintritt in und den Austritt aus der

Schleife `Product` wird jeweils ein zusätzlicher Takt benötigt, so dass eine Iteration der Schleife `Col` 8 Taktzyklen und die gesamte Abarbeitung 24 Taktzyklen benötigt. Für die äußere Schleife `Row` gilt das Gleiche: 2 zusätzliche Takte für Eintritt und Austritt, so dass eine Iteration 26 Takte benötigt und die gesamte Abarbeitung 78 Takte. Mit einem Takt für den Eintritt in die Schleife `Row` und einem Takt für Austritt und Abschluss benötigt der IP Core also insgesamt 80 Takte für die Abarbeitung.

## 6.2 Pipelining der inneren Schleife

- Legen Sie eine neue Solution `solution2` an (`Project > New Solution`). Belassen Sie alle Voreinstellungen der Box.
- Öffnen Sie den Quellcode `matmul.cpp`, um Direktiven über den Directive Tab im Auxiliary Pane angeben zu können. Wählen Sie die Schleife `Product` aus, dann `Insert Directive` und wählen Sie dann die Direktive `PIPELINE` aus. Bei den Options tragen Sie nichts ein. Wenn Sie für den Wert `II` (= Initiation Interval) nichts eintragen, ist die Default-Zielvorgabe `II=1`. Dies bedeutet, dass die nächste Schleifeniteration mit dem nächsten Takt starten soll.
- Führen Sie wieder einen Syntheselauf durch. Nach Abschluss der Synthese sollten Sie im Synthese Report die Performance aus Abbildung 33 sehen.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	12.00	8.18	1.50

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
82	82	83	83	none

Detail

Instance

Loop

Loop Name	Latency		Iteration	Latency	Initiation Interval		Trip Count	Pipelined
	min	max			achieved	target		
- Row_Col	81	81	9		-	-	9	no
+ Product	6	6	3		2	1	3	yes

Abbildung 33: Performance

- Bei der Optimierung von Designs kann man aus den Ausgaben auf der Console nützliche Informationen entnehmen. Wenn Sie die Konsolenausgaben durchsuchen, werden Sie beispielsweise folgende Meldung finden:

```
INFO: [XFORM 203-541] Flattening a loop nest 'Row'...
```

Es ist so, dass Vivado HLS bei Anwendung der `PIPELINE`-Direktive automatisch auch ein so genanntes „Loop Flattening“ durchführt. Da der Eintritt und Austritt in und aus Schleifen jeweils Taktzyklen benötigen, kann die HLS Schleifen ausflachen. In unserem Design wird ja in der `Row`- und `Col`-Schleife jeweils dreimal iteriert und damit die 9 Werte für die Ergebnis-Matrix berechnet. In der generierten Hardware wird dies zu einer Schleife zusammengefasst, in welcher neunmal iteriert wird. Für das Ausflachen von Schleifen ohne dass die `PIPELINE`-Direktive verwendet wird, gibt es auch eine spezielle Direktive (siehe [1]). Die `Product`-Schleife kann übrigens nicht in die `Col`-Schleife ausgeflacht werden, da in dieser Schleife das Argument `res[i][j] = 0` gesetzt wird. Die `Col`-Schleife ist damit keine „perfekte Schleife“. Auch hierzu existiert einer Warnung auf der Console, näheres hierzu kann auch [1] entnommen werden.

- Wenn man sich die Performance in Abbildung 33 ansieht, dann kann man erkennen, dass sich diese gegenüber `solution1` verschlechtert hat. Auf der Console findet man auch hierzu eine interessante Meldung:

```
WARNING: [SCHED 204-68] Unable to enforce a carried dependence constraint
(II = 1, distance = 1, offset = 1) between 'store' operation
(src/matmul.cpp:18) of variable 'tmp_3', src/matmul.cpp:18 on array
'res' and 'load' operation ('res_load', src/matmul.cpp:18) on array 'res'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1,
Final II: 2, Depth: 3.
```

Das Problem wird als „carried dependence“ bezeichnet und resultiert aus der Tatsache, dass die Schleife `Product` im Pipelining betrieben wird. Zur Analyse können Sie auch wieder das Scheduling Diagramm in der Analysis Perspective öffnen, was Abbildung 34 zeigt.

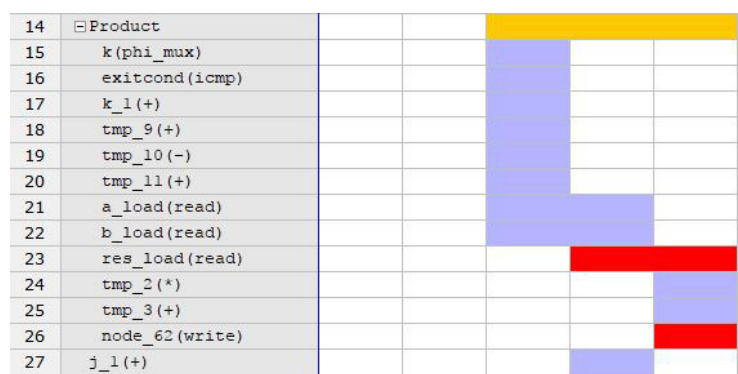


Abbildung 34: Pipelining-Problem

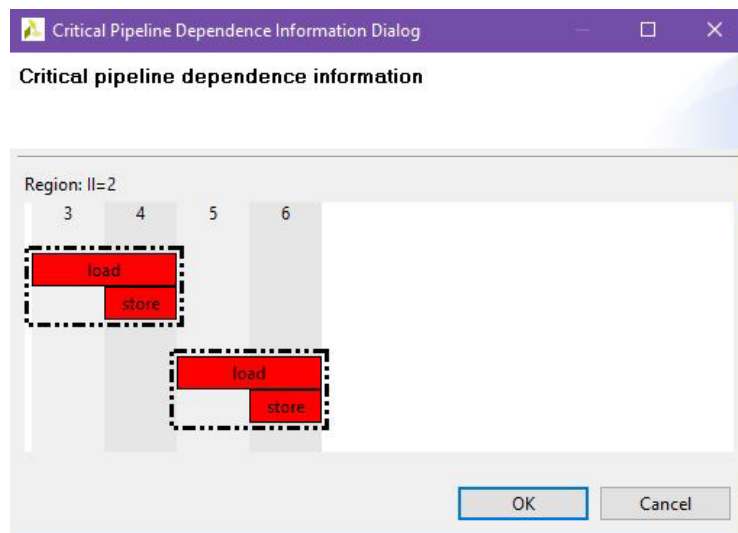


Abbildung 35: Critical Pipeline Dependence

Im Diagramm in Abbildung 34 und aus dem Quellcode ist ersichtlich, dass das Feld `res[i][j]` während einer Iteration der Schleife sowohl gelesen als auch geschrieben werden muss (in Zeile 23 und 26 im Diagramm rot markiert). Werden nun die Schleifeniterationen mit einem `II=1` durch das Pipelining in ihrer Ausführung verschränkt, dann bedeutet das, dass das externe BlockRAM, welches `res` implementiert, im gleichen Taktzyklus sowohl geschrieben als auch gelesen werden muss, was aber nicht

möglich ist. Daher wird  $II=2$  gesetzt und damit die nächste Iteration der Schleife erst nach zwei Taktzyklen gestartet. Wenn Sie auf den roten Balken in Abbildung 34 doppelt klicken, dann erhalten Sie die „Critical Pipeline Dependence“ aus Abbildung 35, die zwei aufeinander folgende Iterationen zeigt und das Problem verdeutlicht: In der ersten Iteration wird das BlockRAM beim Übergang von Zyklus 3 nach 4 gelesen und im Zyklus 4 geschrieben. Folglich kann die nächste Iteration erst in Zyklus 5 starten, da bei einem Start in Zyklus 4 das BlockRAM gleichzeitig gelesen und geschrieben werden müsste. Damit gewinnt man aber im vorliegenden Beispiel nichts gegenüber der ursprünglichen Lösung.

- Die Lösung des Problems ist im Grunde einfach: Wir müssen es vermeiden, dass in der Product-Schleife das Argument `res` gelesen und geschrieben wird. Wir können das Problem durch eine lokale Variable lösen, wie in Abbildung 36 gezeigt. Führen Sie die Änderungen in Ihrem Code durch und starten Sie einen erneuten Syntheselauf.

```

8 void matmul(din_t a[A_ROWS][A_COLS], din_t b[B_ROWS][B_COLS],
9             dout_t res[A_ROWS][B_COLS]){
10
11     uint16_t temp=0;
12
13     // Iterate over the rows of the a matrix
14     Row: for(int i = 0; i < A_ROWS; i++) {
15         // Iterate over the columns of the b matrix
16         Col: for(int j = 0; j < B_COLS; j++) {
17             temp = 0;
18             // Inner product of row vector a and column vector b
19             Product: for(int k = 0; k < A_COLS; k++) {
20                 temp += a[i][k] * b[k][j];
21             }
22             res[i][j] = temp;
23         }
24     }
25 }
26

```

Abbildung 36: Code Optimierung

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	12.00	8.18	1.50

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
30	30	31	31	none

Detail

+ Instance

- Loop

Loop Name	Latency		Iteration Latency	Initiation Interval	Trip Count	Pipelined	
	min	max		achieved			target
- Row_Col_Product	28	28	3	1	1	27	yes

Abbildung 37: Performance des optimierten Codes

- Sehen Sie sich nach dem Syntheselauf die Ausgaben in der Console an und analysieren Sie die Performance (siehe Abbildung 37). Sie müssten jetzt feststellen können, dass nun ein  $II=1$  möglich ist und sich die Performance auf eine Latenz von 30 Taktzyklen verbessert hat. Es ist nun übrigens auch möglich, die



Product-Schleife in die Col-Schleife auszuflachen, da  $\text{res}[i][j] = 0$  nicht mehr nötig ist. Diese kleine Änderung des Codes war also sehr effektiv!

### 6.3 Pipelining der Col-Schleife

- Legen Sie eine neue Solution `solution3` an (Project > New Solution). Belassen Sie alle Voreinstellungen der Box, entfernen Sie aber den Haken bei Copy directives ..., wir werden die Direktive nicht übernehmen.
- Öffnen Sie den Quellcode `matmul.cpp`, um Direktiven über den Directive Tab im Auxiliary Pane angeben zu können. Wählen Sie nun die Col-Schleife aus und geben Sie die PIPELINE-Direktive für diese Schleife an.

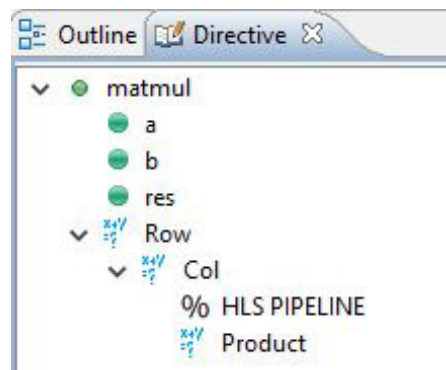


Abbildung 38: PIPELINE-Direktive für Schleife Col

- Führen Sie wieder einen Syntheselauf durch. Sie müssten nun eine Performance wie in Abbildung 39 haben. Die innere Schleife `Product` wurde durch das Pipelining der Schleife `Col` vollständig entrollt. Die `Row`- und `Col`-Schleife wurden ausgeflacht, so dass wir hier 9 Iterationen haben. Allerdings konnte die Vorgabe für  $II=1$  für das Pipelining nicht erreicht werden.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	12.00	8.74	1.50

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
22	22	23	23	none

Detail

Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation achieved	Interval target	Trip Count	Pipelined
	min	max					
- Row Col	20	20	5	2	1	9	yes

Abbildung 39: Performance

- Zur Analyse des Problems durchsuchen Sie bitte wieder die Console-Ausgaben. Sie müssten dort folgendes entdecken:



WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('a\_load\_2', src/matmul.cpp:20) on array 'a' due to limited memory ports.

Sehen Sie sich auch das Interface im Synthese Report (siehe Abbildung 40) an und das Scheduling Diagramm in der Analysis Perspective. Wie man dem Interface entnehmen kann, werden für die beiden Ports a und b jeweils zwei Speicher-Interfaces eingebaut, so dass man hier jeweils ein Dual-Port-RAM anschließen kann. Damit kann man für a und b jeweils zwei Werte pro Taktschritt lesen.

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_clk	in	1	ap_ctrl_hs	matmul	return value	
ap_rst	in	1	ap_ctrl_hs	matmul	return value	
ap_start	in	1	ap_ctrl_hs	matmul	return value	
ap_done	out	1	ap_ctrl_hs	matmul	return value	
ap_idle	out	1	ap_ctrl_hs	matmul	return value	
ap_ready	out	1	ap_ctrl_hs	matmul	return value	
a_address0	out	4	ap_memory	a	array	
a_ce0	out	1	ap_memory	a	array	
a_q0	in	8	ap_memory	a	array	
a_address1	out	4	ap_memory	a	array	
a_ce1	out	1	ap_memory	a	array	
a_q1	in	8	ap_memory	a	array	
b_address0	out	4	ap_memory	b	array	
b_ce0	out	1	ap_memory	b	array	
b_q0	in	8	ap_memory	b	array	
b_address1	out	4	ap_memory	b	array	
b_ce1	out	1	ap_memory	b	array	
b_q1	in	8	ap_memory	b	array	
res_address0	out	4	ap_memory	res	array	
res_ce0	out	1	ap_memory	res	array	
res_we0	out	1	ap_memory	res	array	
res_d0	out	16	ap_memory	res	array	

Abbildung 40: Interface

Durch das Entrollen der Schleife `Product` wäre es allerdings notwendig, dass wir jeweils drei Werte pro Taktschritt lesen können, wenn wir maximale Performance anstreben. Daher werden nun zwei Taktschritte pro Iteration für den Speicherzugriff notwendig und die nächste Schleifeniteration kann erst zu jedem zweiten Taktschritt starten, also  $II=2$ . Das Problem existiert übrigens für beide Ports a und b, allerdings führt der Konflikt für a schon dazu, dass das Scheduling mit einem  $II=2$  implementiert wird, so dass das Problem für b ebenfalls gelöst ist. Um also einen höheren Durchsatz durch  $II=1$  zu erreichen, benötigen wir eine größere Speicherbandbreite, was wir im nächsten Abschnitt vornehmen werden.

## 6.4 Reshaping der Felder

- Legen Sie eine neue Solution `solution4` an (Project > New Solution). Kopieren Sie dabei die Direktiven aus der vorherigen Solution, so dass die `PIPELINE`-Direktive erhalten bleibt.
- Um eine höhere Speicherbandbreite zu erreichen und damit das Problem der vorherigen Solution zu lösen, möchten wir in dieser Solution ein „Reshaping“ der Felder a und b vornehmen. Hierzu muss folgendes bedacht werden: Da die Schleife `Product` entrollt wird und diese über `k` iteriert, müssen wir für das Feld a drei Elemente aus der zweiten Dimension (Spalten) und für das Feld b drei Elemente aus der ersten Dimension (Zeilen) gleichzeitig holen. Dies ist wichtig für die Entscheidung in welcher Dimension das Reshaping durchzuführen ist.
- Wählen Sie zunächst a aus, und wählen Sie die Direktive `ARRAY_RESHAPE` aus. In den Options wählen

Sie als type den Wert `complete` und für die Dimension tragen Sie 2 ein. Führen Sie das Gleiche für `b` durch, aber setzen Sie die Dimension auf 1. Sie sollten die Direktiven nun wie in Abbildung 41 haben.

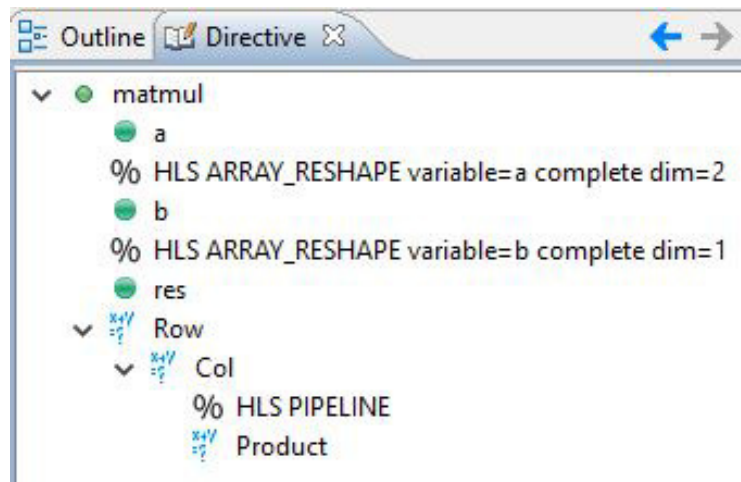


Abbildung 41: Direktiven

- Starten Sie wieder die Synthese. Sie sollten dann eine Performance wie in Abbildung 42 erreichen. Sie werden auf der Console noch eine Warnung erhalten, dass die Zielvorgabe für die Taktperiode nicht erreicht werden kann, unter Angabe des kritischen Pfads. Ob das dann tatsächlich ein Problem ist, müsste man sich nach Place&Route nochmals ansehen. Wir erreichen nun tatsächlich ein  $II=1$  im Pipelining und damit eine Schleifen-Latenz von 10 Takten und eine Gesamtlatenz von 12 Takten. Wenn Sie alle vier Solutions wieder vergleichen, dann ist `solution4` diejenige mit der höchsten Performance. Im Vergleich zu `solution1` und `solution2` benötigen wir aber 3 DSP-Blöcke, da in jeder Iteration ja drei Multiplikationen parallel auszuführen sind.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	12.00	11.13	1.50

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
12	12	13	13	none

Detail

Instance

Loop

Loop Name	Latency	Initiation Interval	Trip Count	Pipelined
- Row Col	min max	achieved target		
	10 10	1 1	9	yes

Abbildung 42: Performance

- Sehen Sie sich abschließend noch das Interface an (Abbildung 43). Für `a` und `b` ist nun nur noch ein Port vorhanden, so dass hier jeweils ein Single-Port RAM anzuschließen wäre. Allerdings ist nun die Bitbreite von 8 Bit auf 24 Bit angewachsen, so dass man pro Speicherzugriff 3 Elemente der Matrix holen kann. Beim Beschreiben der RAMs von außen muss nun aber darauf geachtet werden, dass die Matrix-Elemente in der richtigen Reihenfolge gespeichert werden!

Interface					
Summary					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matmul	return value
ap_rst	in	1	ap_ctrl_hs	matmul	return value
ap_start	in	1	ap_ctrl_hs	matmul	return value
ap_done	out	1	ap_ctrl_hs	matmul	return value
ap_idle	out	1	ap_ctrl_hs	matmul	return value
ap_ready	out	1	ap_ctrl_hs	matmul	return value
a_address0	out	2	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	24	ap_memory	a	array
b_address0	out	2	ap_memory	b	array
b_ce0	out	1	ap_memory	b	array
b_q0	in	24	ap_memory	b	array
res_address0	out	4	ap_memory	res	array
res_ce0	out	1	ap_memory	res	array
res_we0	out	1	ap_memory	res	array
res_d0	out	16	ap_memory	res	array

Abbildung 43: Interface

- Für eine noch höhere Performance könnte man nun die ganze Funktion mit einer PIPELINE-Direktive versehen. Dabei würden dann alle drei Schleifenebenen vollständig entrollt und der Ressourcenaufwand würde weiter ansteigen, insbesondere für die DSP-Blöcke. Da man in der Regel komplexere Funktionen als das vorliegende Beispiel hat, dürfte ein Pipelining einer ganzen Funktion und damit Entrollung sämtlicher Schleifen in der Funktion nur selten wirklich Sinn machen, da der Hardware-Aufwand sehr groß werden kann. Es empfiehlt sich also wie in diesem Lab vorzugehen, Schleifen von innen nach außen mit Pipelining zu versehen, dabei tiefer liegende Schleifen zu entrollen und dabei auch auf eine entsprechende Speicherbandbreite zu achten, um den Hardware-Aufwand durch Entrollen nicht wirkungslos werden zu lassen.

## Literatur

- [1] Xilinx, *Vivado Design Suite User Guide High-Level Synthesis*, 2016. UG902 (v2016.3) October 5, 2016.