

Hardware-Design Praktikum

Anhang

WS 2018/19

Software-Tutorial

Dieser Abschnitt beschreibt die Installation und die Benutzung der Software, die während des Praktikums verwendet wird. Alle Programme sind auf den Rechnern des FPGA-Labors (BH003) installiert und können von den angemeldeten Studentengruppen entsprechend genutzt werden. Ein Teil der Programme findet sich auch im SVN (s. 1).

Die meisten Programme sind als Freeware erhältlich. Hier wird allerdings nur die Installation und Benutzung unter Windows beschrieben. Benutzer anderer Betriebssysteme konsultieren bitte die Webseiten der Hersteller.

TortoiseSVN <http://tortoisesvn.tigris.org>

Über dieses *Version Control System* können die benötigten Daten (Bibliotheken, Testbenches, Skripte, ...) bezogen, sowie eigene Dateien abgegeben werden. Der HaDesXI-Workspace wird im Abschnitt 1 näher erläutert.

GHDL <http://ghdl.free.fr>

Mit diesem VHDL-Simulator werden die beschriebenen Hardware-Komponenten analysiert und getestet. Die Aufrufsyntax dieses Kommandozeilentools wird durch die drei Skripte `lib.bat`, `build.bat` und `run.bat` vereinfacht. Diese können wiederum aus einem Texteditor (z.B. PSPad) heraus aufgerufen werden. Näheres dazu finden Sie in den Abschnitten 2, 3 und 4.

Installationshinweise:

- Für dieses Praktikum wird die Version 0.26 von GHDL empfohlen. Laden Sie sich den Installer dieser Version (`ghdl-installer-0.26.exe`) von der Webseite <http://ghdl.free.fr/download.html> herunter.
- Installieren Sie GHDL durch Ausführen des Installers.
- Kontrollieren Sie, ob der absolute Pfad zum Unterordner `bin` des GHDL Installationsverzeichnis zum Suchpfad hinzugefügt wurde (Umgebungsvariable `PATH`). Falls nicht, fügen Sie diesen Pfad komplett zur Umgebungsvariablen hinzu.

GTKWave <http://gtkwave.sourceforge.net>

Dient zum Betrachten der Signalverläufe (*Waveform*) einer Testbench. Näheres zu diesem Tool und seiner Benutzung finden Sie im Abschnitt 5.

Installationshinweise:

- Laden Sie sich die von uns erstellte, 11 MB große, selbst-extrahierende ZIP-Datei des GTKWave von der WueCampus-Seite des Hardwarepraktikums herunter. **Achtung:** Für die 64-Bit Version des Betriebssystems Windows 7 gibt es eine speziell gezippte Version des GTKWave!
- Entpacken Sie diese in Ihr GHDL-Verzeichnis.
- Kontrollieren Sie, ob der absolute Pfad zum Unterordner `bin` des GTKWave Verzeichnisses zum Suchpfad hinzugefügt wurde (Umgebungsvariable `PATH`). Falls nicht, fügen Sie diesen Pfad zur Umgebungsvariablen hinzu.

PSPad <http://www.pspad.com>

Grundsätzlich können Sie die Programmieraufgaben mit dem Texteditor Ihrer Wahl bearbeiten. Wir empfehlen aber diesen Texteditor, da er bereits Syntaxhighlighter für VHDL besitzt. Die notwendigen Dateien für das Einbinden der GHDL-Skripte sowie für die Entwicklung der Assembler-Programme direkt aus PSPad werden zur Verfügung gestellt.

Installationshinweise:

- Laden Sie sich die von uns erstellte, 6 MB große, selbst-extrahierende ZIP-Datei des PSPad von der WueCampus-Seite des Hardwarepraktikums herunter.
- Entpacken Sie diese in ein Verzeichnis Ihrer Wahl.

Xilinx WebPack <http://www.xilinx.com/support/download.htm>

Diese Software dient der Synthese des Hardware-Designs und der Hardware-Programmierung des FPGAs. Nach kurzer Registrierung ist sie bei der angegebenen Webseite kostenlos herunterladbar. Näheres dazu erfahren Sie in den Abschnitten ?? und ??.

HaDes Object Assembler / HaDes-Linker

Diese Kommandozeilentools dienen dazu, Assembler-Quellcode (.has) in Objekt-Code (.ho) zu übersetzen und anschließend zu Binärdateien (.hix) zu binden. Die Aufrufsyntax wird durch das Skript `compile.bat` vereinfacht, welches wiederum aus einem Texteditor heraus aufgerufen werden kann, siehe Abschnitt 6.

Installationshinweise:

- Die benötigten Dateien (`hoasm.exe`, `hlink.exe`) sind bereits in `HaDes_assembler\bin` enthalten.
- Das Einbinden von `compile.bat` sowie der Assembler-Syntax in PSPad erfolgte bereits, wenn Sie die Installationshinweise zu PSPad beachtet haben.

HaDes-Emulator

Mit diesem Simulator können Sie Ihre Assembler-Programme testen und debuggen. Auf dem FPGA lassen sich die Gründe für unerwartetes Programmverhalten nur sehr schwer lokalisieren. Mit dem Emulator kann die Befehlsverarbeitung und der Inhalt der Register und Datenspeicher schrittweise nachvollzogen werden (siehe Abschnitt 7). Für die Echtzeitsimulation der Programme ist der Emulator aber nur bedingt zu gebrauchen.

Installationshinweise: Der Emulator ist unter `HaDes_assembler\bin\emu.exe` zu finden.

1 Workspace





Das gesamte Arbeitsverzeichnis (*Workspace*) für das Praktikum wird auf einem SVN-Server verwaltet, der unter

`svn://balin.informatik.uni-wuerzburg.de/hadesXI-18`


erreichbar ist. Benutzername und Passwort für den Zugriff wird den Studentengruppen nach der Anmeldung zugewiesen.

Zunächst muss das Arbeitsverzeichnis in einen lokalen Ordner (z.B. `HaDes`) ausgecheckt werden (📁 SVN Checkout im Kontextmenu des lokalen Ordners). Dabei wird folgende Ordnerstruktur angelegt:


HaDes	Workspace
_assembler	Ressourcen für die Assembler-Programmierung
bootloader	Bootloader für die Synthese
inc	Hilfsfunktionen, die direkt eingebunden werden können
tests	CPU-Tests
_bin	Assembler, Linker, Emulator, SimpleTerminal
_doc	Praktikumsanleitung
_ghdl	GHDL Ressourcen
_lib	VHDL-Bibliotheken
unisim	Quellen (nur für die Simulation benötigt)
work	Arbeitsbibliothek
xbus-common	XBus-Quellen für Simulation und Synthese
xbus-sim	Simulationsversion des XBus
xbus-synth	Syntheseversion des XBus
_pspad	Ressourcen für PSPad
_testbench	Testbenches
<group>	Ordner der Praktikumsgruppe
assembler	Assembler-Programm
design	Beschreibungen der Hardware-Module
synthese	Ergebnisse der Hardware-Synthese
hugo44.ucf	Datei mit Konfigurationsparametern für die Synthese
hugo44.xise	Zum Starten des ISE Project Navigators mit Voreinstellungen für die HaDesXI
testbench	Testbenches, die von der Praktikumsgruppe selbst anzufertigen sind

 build.bat	Skript zum Kompilieren eines Moduls
 compile.bat	Skript zum Kompilieren des Assembler-Programms
 lib.bat	Skript zum Erstellen und Löschen von Bibliotheken
 run.bat	Skript zum Ausführen von Testbenches

Während des Praktikums werden im wesentlichen folgende Aktionen wichtig sein:

 **SVN Commit (Hochladen lokal veränderter Dateien)** Jede Praktikumsgruppe besitzt Schreibrechte ausschließlich für ihren Ordner (`<group>`). Beim Hochladen veränderter Dateien dieses Verzeichnisses werden die früheren Versionen nicht überschrieben, so dass im Falle gravierender Probleme jederzeit auch ältere Versionen des Workspace wiederhergestellt werden können.

Sind Fristen für die Abgabe von Teilaufgaben gesetzt, so wird ausschließlich die letzte Version jeder Gruppe vor Ablauf der jeweiligen Frist gewertet. Die Abgabe von (Teil-)Aufgaben erfolgt also ausschließlich per SVN Commit.

Einige Dateien außerhalb des `<group>`-Verzeichnisses (speziell die Bibliotheken) werden während des Praktikums von den Teilnehmern verändert. Diese Veränderungen können und sollen aber nicht auf den SVN-Server übertragen werden. Es empfiehlt sich, solche Dateien zu ignorieren ( Add to ignore list).

 **SVN Update (Herunterladen serverseitig veränderter Dateien)** Jede Praktikumsgruppe besitzt Leserechte auf alle oben angegebenen Ordner des Workspace. Mit einem SVN Update werden alle lokalen Dateien auf die aktuellste serverseitige Version gebracht. In der Regel werden Dateien serverseitig geändert, wenn ein anderes Gruppenmitglied seinen lokalen Gruppenordner hochgeladen hatte. Es können aber auch allgemeine Ressourcen durch die Praktikumsbetreuer geändert oder hinzugefügt werden.

Um Konflikte (Datei lokal **und** serverseitig verändert) zu vermeiden, empfiehlt es sich, grundsätzlich ein Update auszuführen, bevor man mit der Arbeit am lokalen Workspace beginnt.

Für alle weiteren SVN-Optionen, wie etwa das Einpflegen neuer Dateien oder das Wiederherstellen älterer Versionen informieren Sie sich bitte auf den Webseiten von TortoiseSVN.

2 Konzept der VHDL-Bibliotheken

Mit VHDL werden Entities, Architectures und Packages in `.vhd` Dateien beschrieben. Um diese Komponenten auch in anderen Dateien (z.B. zum strukturierten Aufbau von Hardwarekomponenten) zu verwenden, müssen sie zunächst in Bibliotheken (*Libraries*) gesammelt werden. Diese Bibliotheken werden durch `*-obj93.cf` Dateien im `HaDes\lib` Verzeichnis repräsentiert und enthalten im Wesentlichen Links auf die entsprechenden Quelldateien der Komponenten.

Wir verwenden während des Praktikums folgende Bibliotheken:

unisim Quellen nur für die Simulation benötigt Enthält Deklarationen, die nur für die Simulation benötigt werden.

work Arbeitsbibliothek Enthält alle selbst beschriebenen Hardware-Komponenten, sowie ein bereitgestelltes Package `hades-components`, das einige nützliche Komponenten und Datentypen deklariert. Dieses Paket sollte daher in alle selbst erstellten Dateien mittels `use work.hadescomponents.all;` eingebunden werden. Zusätzlich enthält diese Bibliothek noch die Komponenten `hades_addsub`, `hades_compare`, `hades_mul`, `hades_ram_dp` und `hades_shift`, die Sie an gegebener Stelle benutzen können.

xbus_common XBus Quellen für Simulation und Synthese Enthält Beschreibungen von Komponenten und Packages, die bei Simulation und Synthese benötigt werden.

xbus_sim Simulationsversion des XBus Enthält den XBus mit XDMemory für die Simulation.

xbus_synth Syntheseversion des XBus Enthält den synthetisierbaren XBus mit Komponenten wie beispielsweise XConsole, XPS2, XVGA, XLCD und XDMemory.

Nur die Bibliothek `work` kann direkt verwendet werden. Alle anderen müssen zuvor mittels `library <name der bibliothek>;` deklariert werden.

Für den Umgang mit den VHDL-Bibliotheken steht Ihnen das Skript `HaDes\lib.bat` zur Verfügung. Es erwartet ein bis zwei (selbsterklärende) Argumente:

1. [obligatorisch] Name einer Bibliothek oder `all`
2. [optional] `build` oder `clean`

Ohne das zweite Argument wird der Inhalt der Bibliothek(en) angezeigt.


Da auf dem SVN-Server zwar die Quelldateien für die Bibliotheken, nicht aber die Bibliotheken selbst verwaltet werden, müssen sie zunächst (einmalig) erstellt werden:

```

Console: HaDes\
lib all build
== unisim <= unisim_VCOMP ==
== unisim <= unisim_VPKG ==
== xbus_common <= xtoolbox ==
== xbus_common <= xtimerxt ==
== xbus_sim <= xmemory_sim ==
== xbus_sim <= xbus_sim ==
== xbus_synth <= xddr2_mig ==
== xbus_synth <= xddr2_package ==
== xbus_synth <= xddr2_arbiter ==
== xbus_synth <= xddr2_ctrl ==
== xbus_synth <= xddr2_cache ==
== xbus_synth <= xmemory_dcache ==
== xbus_synth <= xvga_rcache ==
== xbus_synth <= xvga_wcache ==
== xbus_synth <= xvga_out ==
== xbus_synth <= xvga ==
== xbus_synth <= xconsole ==
== xbus_synth <= xlcd ==
== xbus_synth <= xps2 ==
== xbus_synth <= xuart ==
== xbus_synth <= xbus_syn ==
== work <= hadescomponents ==
== work <= hades_addsub ==
== work <= hades_compare ==
== work <= hades_mul ==
== work <= hades_shift ==
== work <= hades_ram_dp ==

```

Auftretende Warnungen wie `universal integer bound must be numeric literal or attribute` oder `package "xddr2" does not require a body` können ignoriert werden.

Wurde das Skript in PSPad eingebunden, so kann es auch aus dem Texteditor über  Mit externem Programm öffnen (z.B. im Menü `Datei`) ausgeführt werden, solange eine `.vhd` Datei aus dem `<group>`-Verzeichnis geöffnet ist. Die Argumente werden dabei als Benutzer-Parameter angegeben. Die Ausgabe erfolgt daraufhin in einem separaten Log-Fenster.

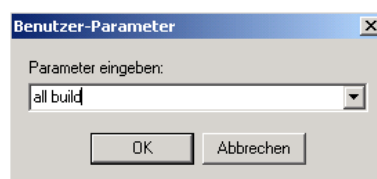


Abbildung 1: PSPad Benutzerparameter eingeben

3 Kompilieren eines Hardwaremoduls in die work-Library

Für jedes Hardwaremodul der HaDesXI-CPU, das von den Praktikumssteilnehmern beschrieben werden soll, existiert eine Datei `HaDes\<group>\design\<module>.vhd`, welche die Entity und Architecture des Moduls enthält. Der Name der Entity muss mit dem Namen der Quelldatei übereinstimmen.

Um ein solches Modul in einer Testbench oder in einem anderen Modul zu verwenden, muss es zunächst in die Bibliothek `work` kompiliert werden. Dazu steht Ihnen das Skript `HaDes\build.bat` zur Verfügung. Es erwartet zwei Argumente:


1. [obligatorisch] Name der Praktikumsgruppe
2. [obligatorisch] Name des Moduls

Hängt das Modul von anderen Modulen ab (wie etwa `datapath` von `alu`), dann werden zunächst die benötigten Module übersetzt. Soll beispielsweise das Modul `datapath` der Gruppe `muLoe` kompiliert werden, so wird folgender Aufruf ausgeführt:

```

Console: HaDes\
build muLoe datapath
== work <= muLoe.alu ==
== work <= muLoe.datapath ==

```

Wurde das Skript in PSPad eingebunden, so kann es auch aus dem Texteditor über  **Aktuelle Datei in einem externen Programm öffnen** oder per **(F9)** ausgeführt werden, wenn die Quelldatei des Moduls geöffnet ist. Die Ausgabe erfolgt in einem separaten Log-Fenster.

4 Ausführen einer Testbench

Da beim Kompilieren eines Hardwaremoduls nur die syntaktische Korrektheit seiner Beschreibung getestet wird, muss seine funktionale Korrektheit in einem zweiten Schritt durch Ausführung einer Testbench überprüft werden. Diese instanziiert das Modul, versorgt seine Eingangspins mit bestimmten Signalverläufen, vergleicht die Signalverläufe der Ausgabepins mit den erwarteten Werten und meldet ggf. Abweichungen.

Erst wenn die Testbench ohne Fehlermeldungen durchläuft (Warnungen können stets ignoriert werden), kann das Modul als korrekt beschrieben angesehen werden. Dabei behalte man aber stets im Hinterkopf:

„Durch einen Test kann man die Anwesenheit, nicht aber die Abwesenheit von Fehlern nachweisen.“

E. W. Dijkstra

Die Quelldatei der Testbench eines Moduls heißt stets `<module>_tb.vhd` und liegt normalerweise im Verzeichnis `HaDes\testbench`. Einige Testbenches sollen aber von den Praktikumssteilnehmern selbst angefertigt oder ergänzt werden. Diese werden dann unter `HaDes\<group>\testbench` abgelegt. Dabei ist folgendes zu beachten:

- Der Name der Entity der Testbench muss mit dem Dateinamen der Testbench übereinstimmen.
- Alle Signale der Testbench müssen so heißen, wie die Ports des zu testenden Moduls, mit denen Sie verbunden sind. Dies ist notwendig, um die mitgelieferten Waveformeneinstellungen richtig laden zu können.
- Die benötigte Laufzeit der Testbench muss (sofern nicht bereits vorgegeben) im Kopfkomentar der Datei eingetragen werden.

Zum Ausführen einer Testbench steht Ihnen das Skript `HaDes\run.bat` zur Verfügung. Es erwartet zwei bis drei Argumente:

1. [obligatorisch] Name der Praktikumsgruppe
2. [obligatorisch] Name des zu testenden Moduls oder `all`
3. [optional] `show` (näheres dazu in Abschnitt 5)

Soll beispielsweise das Modul `PMEMORY` der Gruppe `muLoe` getestet werden, so wird folgender Aufruf ausgeführt:

```

Console: HaDes\
run muLoe pmemory
== work <= _testbench\pmemory_tb ==
== run pmemory_tb for 550ns ==
_testbench\pmemory_tb.vhd:132:8:@526ns:<report note>: !!!TEST DONE !!!
ghdl:info: simulation stopped by --stop-time

```

Hinweis: Die Warnung component instance 'uut' is not bound erhält man, wenn das zu testende Modul noch nicht in die work-Bibliothek kompiliert wurde.

Wurde das Skript in PSPad eingebunden, so kann es auch aus dem Texteditor über (**Alt+F9**) ausgeführt werden, wenn die Quelldatei des zu testenden Moduls geöffnet ist. Die Ausgabe erfolgt in einem separaten Log-Fenster.

Eine Besonderheit stellt die Testbench der CPU dar. Diese kontrolliert das Verhalten von CPU und XBus beim Ausführen eines Programms, das in PMEMORY geladen wird. Es gibt fünf verschiedene solcher Testprogramme: HaDes_testbench\cpu_tb<i>.hex mit $1 \leq i \leq 5$. Beim Ausführen eines dieser Tests muss deshalb zusätzlich die Nummer des Programms angegeben werden, das geladen werden soll:

```

Console: HaDes\
run muLoe cpu 3
== work <= _testbench\cpu_tb ==
== run cpu_tb for 1800ns ==
_testbench\cpu_tb.vhd:115:16:@100ns:(report note):
_testbench\cpu_tb.vhd:116:16:@100ns:(report note): == CPU Test 3 ==
_testbench\cpu_tb.vhd:117:16:@100ns:(report note):
Überprüfen Sie die in _testbench\cpu_tb3.mif spezifizierten Bedingungen in der Waveform!
ghdl:info: simulation stopped by --stop-time

```

Beim Starten des Skripts aus PSPad benutzen Sie **Compilieren (Strg+F9)** und geben die Nummer als Benutzer-Parameter an. Der Aufruf der Waveform kann auch durch die Eingabe der Nummer mit dem Zusatz show erfolgen. Die Korrektheit der Ausführung wird dabei nicht direkt von der Testbench überprüft. Stattdessen müssen die Laufzeitbedingungen, die in der dazugehörigen Quelldatei des Testprogramms angegeben sind (HaDes_testbench\cpu_tb<i>.mif), von Hand in der Waveform (siehe Abschnitt 5) des Testdurchlaufs kontrolliert werden.

5 Beobachten einer Waveform

Die Ursachen von Fehlermeldungen beim Durchlauf einer Testbench sind oft nicht ohne Weiteres ersichtlich. Oftmals hilft es aber, sich die Signalverläufe des Tests anzuschauen, um zu erkennen, welche Eingabesignale anliegen und wie das getestete Modul darauf reagiert.

Dazu verwenden Sie einfach den finalen Parameter show für das Skript HaDes\run.bat:

```

Console: HaDes\
run muLoe pmemory show
== work <= _testbench\pmemory_tb ==
== run pmemory_tb for 550ns ==
_testbench\pmemory_tb.vhd:132:8:@526ns:<report note>: !!!TEST DONE !!!
ghdl:info: simulation stopped by --stop-time
== show waveform ==

```

Daraufhin werden beim Ausführen der Testbench alle Signalverläufe in eine .ghw Datei gedumpt und danach mittels GTKWave angezeigt:

Diese Anzeige besteht im Wesentlichen aus zwei Teilen: Links ist der *Signal Search Tree (SST)* dargestellt, der alle verfügbaren Signale hierarchisch abbildet. Im SST markierte Signale können mittels Append oder Insert in den rechten Anzeigebereich (*Signals / Waves*) kopiert werden. Dort können dann über das Menu Edit die Anzeigeeinstellungen (ewta das Datenformat) angepasst werden. Wollen Sie beispielsweise das Signal OPC aus der INDEC-Testbench mit Alias-Bezeichnungen anzeigen, dann gehen Sie wie folgt vor:

- Selektion von top→indec_tb→opc im SST
- Selektion von []opc[4:0] in Signals unter SST
- Append oder Insert
- Selektion von opc[4:0] in Signals im rechten Anzeigebereich
- Edit→Alias Highlighted Trace
- Eingabe von OPC (dabei wird nur der Signalname, nicht aber die angezeigten Werte verändert)

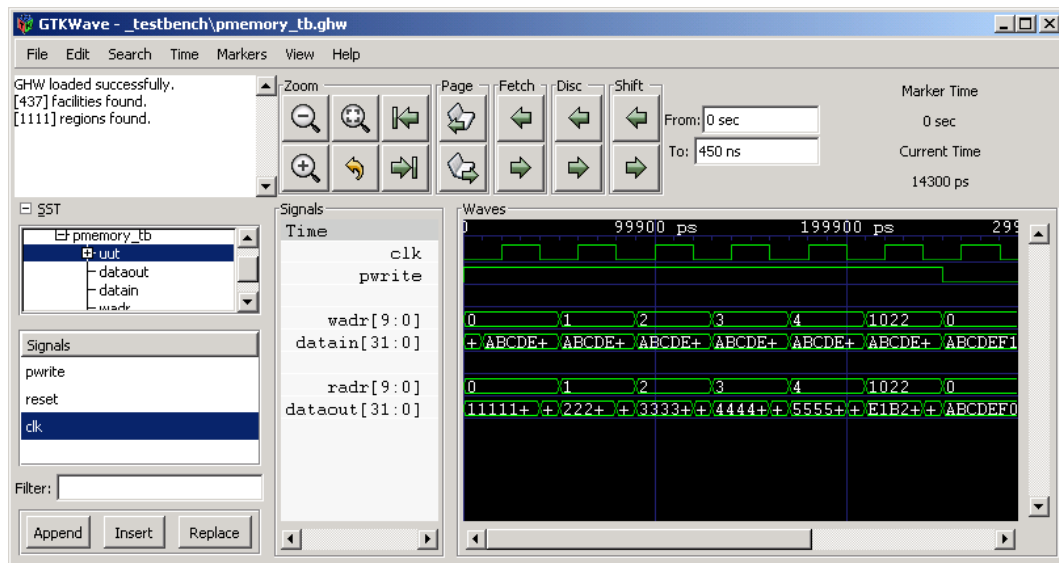



Abbildung 2: GTKWave

- Selektion von OPC in Signals im rechten Anzeigebereich
- Edit→Data Format→Binary
- Edit→Data Format→Translate Filter File→Enable and Select
- Add Filter to List
- Auswahl von _testbench\opcodes.txt und Bestätigen

Näheres zur Verwendung des Waveviewers entnehmen Sie bitte der Webseite des Herstellers.

Bei der Suche nach den Ursachen für das Fehlverhalten eines Hardwaremoduls wird man häufig die Waveform untersuchen, die Beschreibung des Moduls abändern und danach die Ausführung der Testbench sowie die Anzeige der Waveform wiederholen. Da es aber einige Zeit dauern kann, alle interessanten Signale anzuzeigen und zu formatieren, sollte man nicht vergessen, diese Einstellungen über **File→Write Save File (Strg+W)** zu speichern.

Haben Sie Alias-Dateien wie oben beschrieben verwendet, dann sollten Sie nun die erstellte .sav-Datei mit einem Texteditor öffnen und den absoluten Pfad zur Alias-Datei durch einen relativen Pfad (beispielsweise _testbench\opcodes.txt) ersetzen. Dadurch kann die WaveForm mit den Aliasbezeichnungen auch auf anderen Rechnern (etwa von den Praktikumsbetreuern) betrachtet werden.

Um die Anzeige der Waveform beim Ausführen der Testbench aus PSPad heraus zu aktivieren, benutzen Sie (**Shift+F9**), wenn es sich nicht um einen CPU-Test handelt. Ansonsten geben Sie neben der Nummer des CPU-Tests noch das Argument show in den Benutzerparameter ein, nachdem Sie den CPU-Test über  **Compilieren (Strg+F9)** gestartet haben.

6 Übersetzen des Assemblerprogramms

Alle Assembler-Quelldateien werden unter HaDes_init enthalten. Diese bildet die Hauptdatei, aus der das Hex-Programm generiert wird. Alle anderen Quelldateien müssen über `@inc` in die Hauptdatei eingebunden werden. Zusätzlich können jederzeit alle Quelldateien aus HaDes\

Zum Übersetzen des Assemblerprogramms steht Ihnen das Skript HaDes\compile.bat zur Verfügung. Es erwartet als einziges Argument den Namen der Praktikumsgruppe:


```

Console: HaDes\
compile muLoe draw.has
**** Working on file: muLoe\assembler\draw.has ****

```


Bei fehlerfreiem Durchlauf wird das Programm unter HaDes\<group>\assembler im *HaDes Internal Exchange* Format (hix) abgelegt. Dieses Format besteht aus einer Abfolge von 32-Bit Wörtern mit folgender Bedeutung:

Wortindex	Bedeutung
0	Instruktionsanzahl l
1	Erste Instruktion
...	...
l	l -te Instruktion
$l + 1$	Instruktionschecksumme (32 Bit signed)
$l + 2$	Anzahl der Datenwörter d
$l + 3$	Erstes Datenwort
...	...
$l + 2 + d$	d -tes Datenwort
$l + 2 + d + 1$	Datenchecksumme (32 Bit signed)

Wurde das Skript in PSPad eingebunden, so kann es über  **Compilieren (Strg+F9)** gestartet werden, solange der Assembler-Quellcode geöffnet ist.

7 Debuggen eines Assemblerprogramms

Hat man gerade kein FPGA-Board zur Hand oder wird man aus dem Verhalten seines Boards nicht schlau, dann hilft oftmals nur noch der Emulator weiter, der unter HaDes\assembler\bin\emu.exe zu finden ist.

Nach dem Öffnen des .hix Programms stehen fünf Fenster zur Verfügung, die im Menü **View** ausgewählt werden können:

- CPU** Zeigt Inhalt von HAREGS (oben), PMEMORY (links) und XDMEMORY (rechts) an
- XConsole** LCD, Taster, Switches, Drehregler und LEDs
- XVGA** Ausgabe des Graphikspeichers
- XUART** Terminal zur Dateneingabe (einzelne ASCII-Zeichen oder ganze Dateien)
- XPs2** Emuliert Maus oder Tastatur (je nach Einstellung im Menü **Extras**)

Wurden die richtigen Fenster ausgewählt und platziert sowie ggf. Breakpoints (!) an den entscheidenden Programmstellen gesetzt, dann kann zwischen drei Simulationsgeschwindigkeiten gewählt werden:

- ▶ Einzelschritt mit Update des CPU-Fensters
- ▶ Schnell mit Update des CPU-Fensters
- ▶▶ Sehr schnell ohne Update des CPU-Fensters

Dabei bleibt anzumerken, dass auch die schnellste Simulation bei Weitem nicht die tatsächliche Ausführungsgeschwindigkeit auf dem FPGA erreicht. Gerade Zeit-sensible Programme lassen sich daher mit dem Emulator kaum nachvollziehen.

Befehlssatz HaDesXI-Übersicht

Im folgenden seien $w, a, b \in \{0, \dots, 7\}$ Register der HaDesXI-CPU, wobei 0 dem Null-Register entspricht. Konstanten werden durch $\#k$ dargestellt, wobei ihre Länge (Bitzahl) zuvor vereinbart wird. Bei den Annotationen stehen $_s$ für vorzeichenbehaftete, $_u$ für vorzeichenlose, $_{32}$ für auf 32 Bit erweiterte Konstanten, und $_{(x\dots 0)}$ für die jeweils verwendeten Bits.

Zuerst wollen wir die ALU-Befehle einführen. Bei den Immediate-Varianten wird der 16-Bit-Immediate-Operand auf 32 Bit erweitert, wobei dies bei arithmetischen Befehlen vorzeichenbehaftet und bei bitweise logischen Befehlen vorzeichenlos erfolgt.

Befehl	Semantik
NOP	$PC++$
ADD w, a, b	$Reg[w] := Reg[a] + Reg[b]$
SUB w, a, b	$Reg[w] := Reg[a] - Reg[b]$
MUL w, a, b	$Reg[w] := Reg[a] \times Reg[b]$
AND w, a, b	$Reg[w] := Reg[a] \text{ and } Reg[b]$
OR w, a, b	$Reg[w] := Reg[a] \text{ or } Reg[b]$
XOR w, a, b	$Reg[w] := Reg[a] \text{ xor } Reg[b]$
XNOR w, a, b	$Reg[w] := Reg[a] \text{ xnor } Reg[b]$
SHL w, a, b	$Reg[w] := Reg[a] \text{ shl } Reg[b]_{(4\dots 0)}$
CSHL w, a, b	$Reg[w] := Reg[a] \text{ cshl } Reg[b]_{(4\dots 0)}$
SHR w, a, b	$Reg[w] := Reg[a] \text{ shr } Reg[b]_{(4\dots 0)}$
CSHR w, a, b	$Reg[w] := Reg[a] \text{ cshr } Reg[b]_{(4\dots 0)}$
ADDI $w, a, \#k$	$Reg[w] := Reg[a] + \#k_{s32}$
SUBI $w, a, \#k$	$Reg[w] := Reg[a] - \#k_{s32}$
MULI $w, a, \#k$	$Reg[w] := Reg[a] \times \#k_{s32}$
ANDI $w, a, \#k$	$Reg[w] := Reg[a] \text{ and } \#k_{u32}$
ORI $w, a, \#k$	$Reg[w] := Reg[a] \text{ or } \#k_{u32}$
XORI $w, a, \#k$	$Reg[w] := Reg[a] \text{ xor } \#k_{u32}$
XNORI $w, a, \#k$	$Reg[w] := Reg[a] \text{ xnor } \#k_{u32}$
SHLI $w, a, \#k$	$Reg[w] := Reg[a] \text{ shl } \#k_{u32(4\dots 0)}$
CSHLI $w, a, \#k$	$Reg[w] := Reg[a] \text{ cshl } \#k_{u32(4\dots 0)}$
SHRI $w, a, \#k$	$Reg[w] := Reg[a] \text{ shr } \#k_{u32(4\dots 0)}$
CSHRI $w, a, \#k$	$Reg[w] := Reg[a] \text{ cshr } \#k_{u32(4\dots 0)}$
GETOV w	$Reg[w] := \text{aktuelles Overflow-Bit}$
SETOV b	$\text{Overflow-Bit} := Reg[b]_{(0)}$
SETOVI $\#k$	$\text{Overflow-Bit} := \#k_{(0)}$

Die nächste Einheit bilden die Set–Condition–Befehle, die ebenfalls als zweiten Quell–Operanden ein Register oder einen 16–Bit–Immediate–Operanden besitzen können. In letzterer Variante wird dieser vorzeichenbehaftet auf 32 Bit erweitert.

Befehl	Semantik
SEQ w, a, b	$Reg[w] := (Reg[a] = Reg[b]) ? 1 : 0$
SNE w, a, b	$Reg[w] := (Reg[a] \neq Reg[b]) ? 1 : 0$
SLT w, a, b	$Reg[w] := (Reg[a] < Reg[b]) ? 1 : 0$
SGT w, a, b	$Reg[w] := (Reg[a] > Reg[b]) ? 1 : 0$
SLE w, a, b	$Reg[w] := (Reg[a] \leq Reg[b]) ? 1 : 0$
SGE w, a, b	$Reg[w] := (Reg[a] \geq Reg[b]) ? 1 : 0$
SEQI $w, a, \#k$	$Reg[w] := (Reg[a] = \#k_{s32}) ? 1 : 0$
SNEI $w, a, \#k$	$Reg[w] := (Reg[a] \neq \#k_{s32}) ? 1 : 0$
SLTI $w, a, \#k$	$Reg[w] := (Reg[a] < \#k_{s32}) ? 1 : 0$
SGTI $w, a, \#k$	$Reg[w] := (Reg[a] > \#k_{s32}) ? 1 : 0$
SLEI $w, a, \#k$	$Reg[w] := (Reg[a] \leq \#k_{s32}) ? 1 : 0$
SGEI $w, a, \#k$	$Reg[w] := (Reg[a] \geq \#k_{s32}) ? 1 : 0$

Folgende bedingte und unbedingte Sprünge stehen zur Verfügung. Das Sprungziel ist eine 12–Bit–Konstante, die die Differenz zum PC+1 angibt. Das OV–Flag bezieht sich auf das Overflow–Ergebnis des letzten Overflow–generierenden ALU–Befehls (ADD, SUB, SHL, SHR) oder eines SETOV–Befehls.

Achtung: LOAD- und STORE-Befehle verwenden zur Adressberechnung den ADD-Opcode der ALU, d.h. auch sie setzen bzw. löschen das OV-Flag!

Befehl	Semantik
BEQZ a, #k	$PC := (!Reg[a]) ? ((PC + 1 + \#k_s) \bmod 4096) : (PC++)$
BNEZ a, #k	$PC := (Reg[a]) ? ((PC + 1 + \#k_s) \bmod 4096) : (PC++)$
BOV #k	$PC := (OV) ? ((PC + 1 + \#k_s) \bmod 4096) : (PC++)$
JMP #k	BEQZ 0, #k
JAL w, #k	$Reg[w] := PC + 1; PC := ((PC + 1 + \#k_s) \bmod 4096)$
JREG a	$PC := Reg[a]$

Folgende Befehle sprechen den Speicher an. Dabei ist $\#k$ eine 16-Bit-Konstante, die vorzeichenbehaftet auf 32 Bit erweitert wird. Ist der Adresswert negativ oder größer als $2^{18} - 1$ (Größe des Datenspeichers) so wird ein XMEMINTR-Interrupt ausgelöst.

LOAD wirkt stets auf den externen Datenspeicher XMEMORY, wohingegen STORE bis zum Aufruf eines DPMA-Befehles (*Disable Program Memory Access*) auf den internen Programmspeicher PMEMORY wirkt und erst dann den XMEMORY anspricht. Der Programmspeicherzugriff kann durch den EPMA-Befehl (*Enable Program Memory Access*) wieder ermöglicht werden. Der Programmspeicher PMEMORY wird bei negativen oder zu großen Adresswerten als Ringspeicher angesehen (*Wrapping*).

Befehl	Semantik
LOAD w, a, #k	$Reg[w] := XMEMORY[Reg[a] + \#k_{s32}]$
STORE b, a, #k	$XMEMORY[Reg[a] + \#k_{s32}] := Reg[b]$ (nach DPMA) $PMEMORY[Reg[a] + \#k_{s32}] := Reg[b]$ (vor DPMA)
DPMA	Beenden des PMEMORY-Schreibzugriffs
EPMA	Einschalten des PMEMORY-Schreibzugriffs

Die folgenden Befehle stellen Zugriffsmöglichkeiten auf externe Komponenten (Peripherie) bereit. Die 16-Bit-Konstante gibt die Port-Adresse der Komponente am XBus an.

Befehl	Semantik
IN w, #k	$Reg[w] := IOReg[\#k_u]$
OUT a, #k	$IOReg[\#k_u] := Reg[a]$

Die HaDesXI-CPU verfügt über vier Interrupt-Ebenen. Diese werden durch Interrupt-Service-Routinen (ISR) im Programmspeicher verarbeitet. Für jede Ebene der Interrupts kann die Einsprungsadresse der ISR zu jedem Zeitpunkt durch den SISA-Befehl (*Set Interrupt Service Address*) neu gesetzt werden. Dabei ist $\#k$ eine 12-Bit-Konstante und entspricht - analog zu den Sprungbefehlen - der Differenz der Sprungadresse zum aktuellen PC+1.

Mit dem SWI-Befehl wird ein Software-Interrupt ausgelöst. Dabei werden zwei Argumente gespeichert, eine 16-Bit-Konstante und ein Registerwert. Diese Argumente können später im Interrupthandler mithilfe des GETSWI-Befehls wieder ausgelesen werden.

Die DEI-/ENI-Befehle sperren bzw. erlauben die Interrupt-Ausführung neuer Interrupts, aber nicht deren Annahme (d.h. das Puffern der Ereignisse, wobei pro Level maximal ein Event gepuffert werden kann!)

Befehl	Semantik
ENI	Enable Interrupt
DEI	Disable Interrupt
SISA #0, #k	$Software-IntrHandlerAdr := (PC + 1 + \#k_s) \bmod 4096$
SISA #1, #k	$XPeripherie-IntrHandlerAdr := (PC + 1 + \#k_s) \bmod 4096$
SISA #2, #k	$XNA-IntrHandlerAdr := (PC + 1 + \#k_s) \bmod 4096$
SISA #3, #k	$XMEM-IntrHandlerAdr := (PC + 1 + \#k_s) \bmod 4096$
RETI	Rückkehr zum letzten Interruptlevel
SWI #k, a	Software Interrupt auslösen mit den Argumenten $\#k$ und $Reg[a]$
GETSWI w, #0	$Reg[w] := \#k$ des letzten Software Interrupts
GETSWI w, #1	$Reg[w] := Reg[a]$ des letzten Software Interrupts

Zuletzt gibt es noch einige Befehle, die nicht notwendig sind, aber das Leben des Programmierers erleichtern:

Befehl	Semantik
LDI $w, \#k$	ADDI $w, 0, \#k$ (Load Signed)
LDUI $w, \#k$	ORI $w, 0, \#k$ (Load Unsigned)
MOV a, w	ADDI $w, a, \#0$ (Move Register)
INC a	ADDI $a, a, \#1$ (Increment)
DEC a	SUBI $a, a, \#1$ (Decrement)