VHDL Wishbone Intercon Generator Geschrieben in Python, generiert VHDL

HARALD HECKMANN

Studiengang Angewandte Informatik, Hochschule RheinMain
Wintersemester 2015/2016
Projekt bei Prof. Dr. Steffen Reith

VHDL Wishbone Intercon Generator
Geschrieben in Python, generiert VHDL

Zusammenfassung

Dieses Dokument beschreibt Einleitend Wishbone. Anschließend wird die Projektstruktur und die Funktion der einzelnen Komponenten beschrieben

Inhaltsverzeichnis

1 Einleitung

1.1 Wishbone

Wishbone ist ein Opensource Bussystem für System-on-Chip (SoC) Systeme. Wishbone gibt ein Regelwerk vor, das bei der Entwicklung von IP-Cores berücksichtigt werden kann, sodass alle Module, die diese Regeln berücksichtigen, in der Lage sind bei geeigneter Verschaltung miteinander zu kommunizieren. Die Teilnehmer eines Wishbone Bussystems werden in Master und Slave Kompontenten unterteilt. Lediglich ein Master initiiert einen Datenaustausch und gibt dabei vor, ob es ein lesender oder schreibender Zugriff ist und an welcher Adresse die Daten abgelegt oder ausgelesen werden. Die Verschaltungseinheit, welche im weiteren Verlauf dieses Dokumentes mit dem Begriff „Intercon"referenziert wird, verschaltet bei einer Anfrage des Masters mit Hilfe der vom Master angegebenen Adresse dessen Leitungen mit den Leitungen des für die Adresse zuständigen Slaves.

1.2 Aufgaben des Wishbone Intercon

- Adressdekodierer - den für die angegebene Adresse zuständigen Slave wählen
- (Nur bei Systemen mit mehreren Mastern) Ein Arbiter, der vom Benutzer definiert wird
- Verbindung der Komponenten, sodass
  - variable Adress- und Datenbusbreiten berücksichtigt werden
  - byte- und Wordadressierung berücksichtigt werden
  - bei unterschiedlicher Endianess eine Konvertierung durchgeführt wird

1.3 Wishbone Beispiele

Beispiel 1: Übungsszenario

(Master) Button-Controller
(Slave1) LED-Controller
(Slave2) RGB-LED-Controller

Beispiel 2: Reales Szenario

(Master) MCU
(Slave1) VGA-Controller
(Slave2) SRAM-Controller

2 Das Projekt

Ziel des Projektes ist es, einen VHDL Code Generator zu erstellen, der Informationen aus einer Konfigurationsdatei ausliest und aus diesen Informationen einen geeigneten Wishbone Intercon in VHDL (Hardwarebeschreibungssprache) generiert. Der Intercon soll in der Lage sein, einen Master und beliebig viele Slaves mit einer „Point-to-Point Verbindung" (1 Master, 1 Slave) oder einem „Shared Bus" (1 Master, n Slaves) zu verschalten.

2.1 Projektaufbau

Folgende Ordnerstruktur findet sich im Projektverzeichnis wieder:

- ./
  - Beinhaltet die Python-Datei main.py, welche die Programmhauptroutine enthält.
- cfg/
  - Beinhaltet eine Konfigurationsdatei, aus der die Informationen für den Intercon entnommen werden
- documentation/
  - Beinhaltet diese Dokumentation
- libs/
  - Beinhaltet Python-Dateien, die dazu verwendet werden einen Wishbone Bus in Objekten zu abstrahieren als auch eine Konfigurationsdatei einzulesen und eine VHDL-Datei zu generieren
- literature/
  - Beinhaltet Literatur, die im laufe dieses Projektes verwendet wurde
- vhdl/
  - Beinhaltet Templatedateien (dazu später mehr) und die generierten VHDL-Dateien

2.2 Aufgaben der einzelnen Module

libs/wb_component.py
Diese Datei beinhaltet folgende Klassen:

- WishboneComponent
- WishboneComponent:WishboneMaster
- WishboneComponent:WishboneSlave

Die Klassen dienen dazu Informationen über Wishbone Master und Slaves in Objekten zu abstrahieren. Generelle Informationen, die Master als auch Slaves benötigen, werden in Objekten der Klasse WishboneComponent abgespeichert. Alle Informationen, die Master Komponenten benötigen, werden zudem in Objekten der Klasse WishboneMaster gespeichert, welche von WishboneComponent erbt. Das gleiche gilt für Slave Komponenten, deren Informationen in Objekten der Klasse WishboneSlave abgespeichert werden. Diese Objekte sind leicht erweiterbar und können in anderen Projekten verwendet werden.

libs/wb_intercon.py
Diese Datei beinhaltet die Klasse WishboneIntercon. In Objekten der Klasse WishboneIntercon sind folgende Informationen hinterlegt:

- Generelle Informationen über den Wishbone Intercon
- Ein WishboneMaster Objekt
- Beliebig viele (unterschiedliche) WishboneSlave Objekte

Das WishboneIntercon Objekt ist leicht erweiterbar und kann in anderen Projekten verwendet werden.

libs/wb_file_manager.py
Diese Datei beinhaltet die Klasse WishboneFileManager. Es ist die Klasse, die das einlesen der Konfigurationsdatei übernimmt, diese Informationen in WishboneMaster, WishboneSlave und WishboneIntercon Objekte einspeißt, und letztendlich mit dem erzeugten WishboneIntercon Objekt, welches wie bereits erwähnt ein WishboneMaster und beliebig viele WishboneSlave Objekte enthält (also alles was man braucht), einen Wishbone VHDL Intercon generieren kann. Abgesehen davon bietet Sie auch eine Funktion, um die Informationen aus den erstellen Objekten sauber auf einer Konsole auszugeben.

libs/main.py
Diese Datei beinhaltet die Hauptroutine, welche alle Funktionen in der richtigen Reihenfolge aufruft und zudem eine Berechnungsdauer und den Ausgangstatus auf einer Konsole ausgibt. Es ist zwingend erforderlich, das Programm mit Python 3.0 oder neuer auszuführen.
Befehl in der Konsole: python3 main.py

2.3 Aufbau der Konfigurationsdatei

Das Pythonprogramm entnimmt alle Informationen über den Intercon, sowie den Master und Slave Komponenten, einer Konfigurationsdatei namens „wishbone.ini“. Die Konfigurationsdatei kann in drei Bereiche aufgeteilt werden:

1. General - enthält generelle Informationen über den Wishbone Intercon
2. Master - enthält Informationen über den Wishbone Master
3. Slave(n) - enthält Informationen über den Wishbone Slave mit der nummer „n“

Bereich: General

```
# ---------------------- General configuration ----------------------
# List of keywords for general configuration
# tga_bits = decimal value
# tgc_bits = decimal value
# tgd_bits = decimal value
# data_bus_width = decimal value (size of databus in intercon, bits)
# address_bus_with = decimal value (size of addressbus in intercon, bits)

[GENERAL]
name = the_intercon
# use 0 bits if you do not need these signals
tga_bits = 3
tgc_bits = 3
tgd_bits = 3
data_bus_width = 32
address_bus_width = 32
```

Die generellen Informationen geben, abgesehen vom Namen, an, wie Breit (in Bits) die Leitungen zwischen Master und Slave Komponenten im Intercon sind. Die Leitungen im Intercon können durchaus breiter sein als die des Masters oder der Slaves.

Bereich: Master

```
# ---------------------- (single) Master component ----------------------


# List of keywords for master modules and possible values:
# name = string (will be used as signal prefix)
# data_bus_width = decimal value
# address_bus_width = decimal value
# endianess = big/little
# data_flow = r/w/rw
# err = true/false
# rty = true/false
# tga = true/false
# tgc = true/false
# tgd = true/false

[MASTER]
# name
name = btn_ctrl
# bus sizes and byte ordering
data_bus_width = 32
address_bus_width = 32
endianess = big

# write / read access
data_flow = w

# additional signals
err = true
rty = true
tga = true
tgc = true
tgd = true
```

Der Master lässt sich hinsichtlich

- seines Namens
- der Breite des Daten- und Adressbusses (in Bits)
- der Position des most-significant Bits (endianess)
- der Lese- und Schreibberechtigung auf den Datenbus
- zusätzlicher Signale (err, rty, tga, tgc, tgd)

konfigurieren.

Bereich: Slave

```
# ---------------------- Slave components ----------------------


# List of keywords for slave modules and possible values:
# base_address = hexadecimal value
# address_size = hexadecimal value
# addressing_granularity = byte/word
# word_size = decimal value
# address_bus_high = decimal value
# address_bus_low = decimal value (lowest bit = 0)
# name = string (will be used as signal prefix)
# data_bus_width = decimal value
# endianess = big/little
# data_flow = r/w/rw
# err = true/false
# rty = true/false
# tga = true/false
# tgc = true/false
# tgd = true/false

# name slaves section SLAVEn, where n E N+
[SLAVE1]
# name
name = led_ctrl
# bus sizes and byte ordering
data_bus_width = 32
endianess = little
# partial address decoding
address_bus_high = 31
address_bus_low = 0

# base address and address size
base_address = 0x00000000
address_size = 0x00100000

# address bus granularity and word size
addressing_granularity = byte
word_size = 32

# write / read access
data_flow = r

# additional signals
err = true
rty = true
tga = false
tgc = false
tgd = false
```

Slaves lassen sich hinsichtlich

- ihres Namens

- der Datenbusbreite (in Bits)
- der Position des most-significant Bits (endianess)
- des höchstwertistgen und des niederwertigsten relevanten Bit der Adressleitung
- der Adressierungspräzision
- der Lese- und Schreibberechtigung auf den Datenbus
- der zusätzlichen Signale (err, rty, tga, tgc, tgd)

konfigurieren.

2.4 Arbeitsweise des Generators

Der Generator verwendet ein WishboneIntercon Objekt (aus wb_intercon.py), dass wie bereits bekannt alle notwendigen Informationen aus der Konfigurationsdatei enthält. Anschließend wird eine Templatedatei geöffnet, die den gesamten VHDL-Code und Platzhalter enthält. Einige Platzhalter werden immer durch Text ersetzt, da diese Zeilen im Template bei jeder Konfiguration im generierten VHDL Code enthalten sein werden. Nicht der gesamte VHDL Code kann so generiert werden, da es VHDL Code gibt der nur bei bestimmten Konfigurationen erzeugt wird (Beispielsweise 0-6 zusätzliche Signale) oder sich bei unterschiedlichen Konfigurationen unterscheidet (Beispielsweise Verschaltung der Datenleitung mit gleicher und unterschiedlicher Endianess). Der Code, der von der Struktur her nicht immer gleich bleibt, wird in Python erstellt und nachträglich hinzugefügt.

2.4.1 VHDL Templates

template_intercon.tmpl
Diese Templatedatei enthält die gesamte VHDL Codestruktur mit Platzhaltern. In ihr sind die Portdefinitionen des Masters, ein Platzhalter für alle Portdefinitionen der Slaves, die Signalerstellung- und Zuweisung für feste als auch optionale Signale und der strukturelle Programmfluss mit Platzhaltern enthalten. Die Platzhalter für optionale Funktionalität werden im Pythoncode ersetzt, da der Aufbau erst zur Laufzeit des Programs (nach einlesen der Konfigurationsdatei) bekannt ist. Der Platzhalter „%slaves%"in der Portdefinition wird durch die Portdefinition aller Slaves ersetzt. Da erst zur Laufzeit bekannt ist, wieviele Slaves es geben wird, kann die Portdefinition für die Slaves erst zur Laufzeit des Programms erstellt werden. Für jeden Slave wird eine Kopie des Inhalts aus der Datei „template_slave.tmpl"modifiziert und anschließend der Portdefinition beigefügt.

template_slave.tmpl
Diese Templatedatei ist entstanden, da es eine beliebige Anzahl an Slaves geben kann und die Portdefinition beliebig oft benötigt wird. Sie enthält lediglich die Portdefinition der Leitungen die zwingend vorhanden sein müssen und einen Platzhalter für die optionalen Leitungen.

2.5 Programmfluss

Alle Module mit Assoiziationspfeilen zeichen, module im nächsten kapitel erklären (AKTIVITÄTSDIAGRAMM?)
Dateipfade angeben
WICHTIG: Programm main.py muss mit python3 ausgeführt werden (NICHT python)

2.6 Fehlende Features

2.6.1 Im Generator nicht Berücksichtigt

Einige Konfigurationsmöglichkeiten werden mit in die Objekte (WishboneMaster, WishboneSlave, WishboneIntercon) übernommen, allerdings nicht vom Generator berücksichtigt. Dazu zählen:

Für Slaves:

- addressing_granularity und word_size
  Beschreibt, in welchem Abstand Adressen angesprochen werden können.

  Beispiel (Adressraum: 0x0 - 0x100):

  addressing_granularity = byte, Adressen 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, etc. können angesprochen werden

  addressing_granularity = word und word_size = 32, Adressen können in log2(32)=4er Schritten angesprochen werden: 0x0, 0x4, 0x8, etc.

Für Master und Slaves:

- data_flow
  Gibt an, ob eine Komponente auf den Datenbus nur lesen, nur schreiben oder lesen und schreiben darf

2.6.2 Nachträglich entfernt

- datatransfer kann die Werte single, burst und rmw (read, modify, write) annehmen. Dieser Wert wurde auf Grund eines Missverständnisses zu Begin des Projektes anschließend im Verlauf des Projektes wieder entfernt, da das Verhalten des Intercon sich nicht dadurch beinflusst, ob die Daten einzelnt, mit bursts oder mittels rmw übertragen werden

2.7 Probleme / Schwierigkeiten

Ohne jemals etwas über die Arbeitsweise von Bussystemen gehört zu haben, ist es sehr Schwierig zu verstehen, was in einem Wishbone Bussystem genau geschieht. Das offizielle Wishbone Referenzdokument „Wishbone B4" war im laufe des Projektes die Hauptreferenzquelle. An der Stelle ist es wichtig zu erwähnen, dass der Autor des Dokuments hervorragende Arbeit geleistet hat. Schon Bald wurde klar, was in einem Wishbone Bussystem geschieht und wie die Komponenten miteinander kommunizieren, allerdings war eine lang anhaltende Schwierigkeit zu differenzen, welche Aufgaben in den Wishbone Modulen und welche Aufgaben in dem Wishbone Intercon durchgeführt werden. Hierbei hat sehr geholfen, sich bestehende Intercons anzuschauen und bereits bestehende Intercon Generatoren und deren generierten Code zu analyisieren.
Ein weitere Schwierigkeit ist es, mit einem Programm, das VHDL Quelltext generieren soll, einen rein kombinatorischen Schaltkreis ohne Latches zu erstellen. Da sich der generierte VHDL Code je nach Konfiguration verändert, verändern sich auch die Signalzuweisungen, die man zu beachten hat um den Schaltkreis kombinatorisch zu gestalten. Es ist in diesem Projekt nicht gelungen, einen Generator zu schreiben, der in allen Fällen einen VHDL Code generiert, welcher einen rein kombinatorischen Schaltkreis beschreibt.

3 Schlusswort

War die Vorgehensweise rückblickend gut?
Die Vorgehensweise, in Python einen Parser und Generator (in dieser Reihenfolge) zu schreiben, ist weder eine besonders hervorragende, noch eine besonders schlechte Idee. Der Zusatzaufwand, der erforderlich ist um einen Parser und einen Generator zu schreiben, ist nicht außer acht zu lassen. Für diese Zwecke ist Python jedoch geeignet. Der Parser und der Generator machen den kleineren Teil aus, der größte Teil ist die Erstellung und Validierung der Objekte, bevor diese von der Generatorprozedur verwendet werden. Wäre das Projekt größer und der zu generierende Code von weiteren Bedingungen abhängig (= größeres Regelwerk für den Generator), so wird der Generator in Python unüberschaubar und nicht wartbar. Für einen Generator wie in diesem Pythonprojekt, der nur ein sehr kleines Regelwerk benötigt, wird es schon langsam unüberschaubar. Ohne Wissen über Compilerbau ist diese Vorgehensweise jedoch ein gutes Training, einen Teil der Probleme, die Generatoren zu behandeln haben, genauer kennen zu lernen. Eine Alternative ist, bereits bestehende Code Generatoren zu verwenden, was einen wesentlichen Anteil and Programmcode spart und das Projekt leichter wartbar macht. Um einen solchen Generator genannt zu haben, „GSL Universal Code Generator "bietet die Möglichkeit „Für alle Sprachen und alle Zwecke "(Zitat aus der Dokumentation des Generators) Code zu generieren.

was lief gut?

was lief schlecht?

Bekannte Bugs

Abschließend muss erwähnt werden, dass das Wissen das bei der Erstellung eines solchen Projektes über Parser, Generatoren, VHDL und vor allen Dingen Bussystemen, von großem Wert ist. Für das weitere Hardwarenahe Arbeiten ist das Wissen über Bussysteme von großer Bedeutung und beinahe unvermeidbar. Für das weitere Arbeiten mit FPGAs ist das Wissen über die Umsetzung von VHDL-Strukturen in Hardware von großer Wichtigkeit. Und Falls jemals wieder Code generiert werden muss, so ist nun klar, das nächstes mal ein Generatortool dafür verwendet wird oder ein spezielles Generatortool, erst nach Aneignung einigen Wissens über Compilerbau, entwickelt wird.