Rechnerarchitektur
Sommersemester 24

ertis embedded real-time systems ulm university institute

# Laborprojekt – Versuch 7

Dieser Versuch beinhaltet die Umsetzung des Datenspeichers in der Memory-Phase als idealisierte Komponente. Idealisiert bedeutet, dass der Speicher mit allen nötigen Daten direkt eingebunden wird. In einem komplexeren System würde an dieser Stelle ein Cache mit Anbindung an die nächste Speicherstufe stehen, um die Zeiten und Kosten des Systems zu reduzieren. Sie werden dazu in diesem Versuch die Load- und Store-Befehle umsetzen, die das Speichern und Laden, also den Austausch von Daten zwischen Registerfile und Datenspeicher, ermöglichen.

## Ziele des 6. Versuchs

- Anpassung der aktuellen Architekturen und Komponenten (Decoder, Konstanten)
- Konzeptionierung, Umsetzung und Validierung des Datenspeichers mit integriertem Load/Store-Controller
- Erweiterung der RISC-V-Umsetzung um die Load/Store-Befehle

## Vorbereitung

### Spezifikation RISC-V (L/S-Befehle)

Eignen Sie sich den Ablauf, die Phasen und die Ansteuerung für die Ausführung von Load- und Store-Befehlen basierend auf der Vorlesung und der Referenz (RV32I Base Integer Instruction Set, Version 2.0) an. Eignen Sie sich ein Grundverständnis so an, dass Sie die Befehlscodierung und die Ansteuerung mithilfe der Referenzkarten für jede Phase wiedergeben und erklären können. Erarbeiten Sie sich ein Konzept zur Umsetzung der RISC-V-Befehle zum Speichern und Laden. Sie sollten in der Lage sein, die Notwendigkeit von Cache in Bezug zum Pipelining und den Unterschied zwischen Caches und idealisierten Speichern erklären zu können und die verbundene Problematik aufzeigen zu können.

### GHDL-Standards

Lesen Sie sich in die GHDL-Optionen ein, insbesondere wie Sie Standards der Sprache einstellen. Ab sofort werden alle Komponenten und Testbenches nur noch mit VHDL 2008 genutzt.

# Aufgaben

## Aufgabe 1: Anpassung der Architektur

Erarbeiten Sie sich die Änderungen in den zur Verfügung gestellten Entity. Passen Sie Ihr System gemäß den folgenden Spezifikationen an und überprüfen Sie die Funktionsweise des Systems mit der Testbench des vorherigen Versuchs. Bisher haben wir für die Ansteuerung des Registerfiles zum Schreiben einzig das negierte Steuersignal *IS_BRANCH* genutzt. Dies ersetzen wir nun durch das Signal *REG_WRITE*. Damit verhindern wir bei einem Store-Befehl, dass etwas zurückgeschrieben wird (ist ja kein Branch). Zusätzlich erweitern wir unser Steuerwort um die Signale *IS_JUMP* für die Sprungbefehle, *MEM_READ* für die Load-Befehle, *MEM_WRITE* für die Store-Befehle und *MEM_CTR* zur Auswahl und Steuerung, was geladen oder gespeichert werden soll (siehe Befehle).

- Erstellen Sie eine Kopie Ihres Systems als *riubs_only_RISC_V_tb.vhdl* und passen Sie ggf. die Signale in dieser und der Entity Decoder an.
- Erweitern SIe Ihren Sign-Extender um die das S-Immediate.
- Testen Sie Ihre kompletten Anpassungen und Erweiterungen abschließend mit der zur Verfügung gestellten Entity *riu_only_RISC_V_tb* aus Versuch 6 vollständig.

## Aufgabe 2: Datenspeicher

Für die Umsetzung des Datenspeichers stellen wir Ihnen die Entity *data_memory* zur Verfügung. Diese verfügt über Ports, die zum Lesen und Schreiben der Daten auf dem Speicher sowie dem Steuereingang zur Befehls-abhängigen Selektion und Erweiterung der Datenwörter (Halbwörter oder Bytes) dienen. Diese Erweiterung ist normalerweise vor- und nachgelagert, kann aber durch die Idealisierung hier kompakter verwendet werden.

Erstellen Sie gemäß der RISC-V-Referenz eine Testbench, die diesen Datenspeicher stichprobenartig testet. Dazu würde das System mit einem Debug-Port ausgestattet, der Ihnen jederzeit den Zugriff auf den kompletten Speicherinhalt ermöglicht. Binden Sie anschließend den Datenspeicher in Ihr System ein und testen Sie dieses mit der Testbench aus Versuch 6, auch hier sollte es zu keinen Problemen kommen.

- Erstellen Sie die Testbench *data_memory_tb.vhdl* und testen Sie den Datenspeicher.
- Erweitern Sie Ihr System um den Datenspeicher in der Memory-Phase, inklusive aller Pipelines, die noch fehlen.
- Passen Sie Ihr System nun so an, dass der *pi_writedata* mit dem zweiten Operanden des Registerfiles in der MEM-Phase, *pi_adr* mit dem Ergebnis der ALU in der MEM-Phase, den Steuersignalen in der MEM-Phase sowie dem Signal *s_readdataMEM*, welches den Ausgang *po_readdata* mit einem Register und in der WB-Phase mit dem vierten Eingang des WB-MUX verbindet.
- Testen Sie abschließend Ihr Testsystem mit der Testbench *riub_only_RISC_V_tb.vhdl* erfolgreich.

## Aufgabe 3: L/S-Befehle

Erweitern Sie nun das System, so dass Sie auf den Speicher über Load- und Store-Befehle zugreifen können.

- Erweitern Sie Ihren Decoder um die Load- und Store-Befehle. Achten Sie darauf, dass Sie die Signale *I_IMM_SEL* und *MEM_WRITE* für die Store-Befehle auf 1 setzen und für die Lade-Befehle *I_IMM_SEL*, *MEM_READ* und *REG_WRITE* jeweils auf 1 und *WB_SEL* auf 11 (den bisher ungenutzten vierten Eingang des WB_MUX). Vergessen Sie abschließend nicht das Signal *MEM_CTR* gemäß der Befehle zu setzen (funct3).
- Erweitern SIe Ihr System um einen Ausgangsport *po_debugdatamemory* über dem Sie der Testbench den kompletten Datenspeicher zur Verfügung stellen.
- Testen Sie abschließend Ihr Testsystem mit der Testbench *riubs_only_RISC_V_tb.vhdl* erfolgreich.

Gegebenenfalls funktioniert Ihr System nicht richtig, das könnte neben jedweden anderen Fehlern ein Problem mit dem Flushing sein. Grundsätzlich müssen wir beim Flushen die Daten aller drei Phasen vor der Memory-Phase auf Null setzten ( NOP). Ein weiteres Problem könnte sein, dass Sie falsch flushen, überprüfen Sie dazu die VCD und achten Sie darauf, dass Sie taktsynchron flushen. Erweitern Sie Ihre Steuerung zum Flushen so, wenn das noch nicht geschehen ist, dass Sie auch bei allen Sprüngen Ihr System entsprechend flushen (vergessen Sie JAL und JALR nicht).

## Aufgabe 4: Abgabe

Laden Sie die erstellte Ordnerstruktur mit den dazugehörigen Dateien als Zip-Datei unter dem Namen

*Name_Vorname_Versuch7.zip* in Moodle hoch.