Rechnerarchitektur
Sommersemester 24

ertis embedded real-time systems
ulm university institute

Laborprojekt – Versuch 2

In Ihrem letzten Versuch haben Sie die generischen Komponenten ALU, 2-1-Multiplexer und Pipeline-Register erstellt. In diesem Versuch erweitern wir die Komponenten um ein Registerfile für den RISC-V (ähnlich dem der MIPS aus GdTi), ein Befehls-Decoder (beschränkt auf die Registerbefehle) und eine Sign-Extension. Die Sign-Extension wird ausgelegt auf alle Befehlsarten und soll die möglichen Immediates zur Verwendung in der RISC-V-Architektur auf 32-Bit gemäß Spezifikation erweitern.

Ziele des 1. Versuchs

- Konzeptionierung, Umsetzung und Validierung eines generischen Registerfile
- Konzeptionierung, Umsetzung und Validierung eines Befehls-Decoders für R-Befehle
- Konzeptionierung, Umsetzung und Validierung einer RISC-V-Sign-Extension

Vorbereitung

Registerfile

Eignen Sie sich die Spezifikation des Registerfiles der RISC-V-Spezifikation (Handbuch) so an, dass Sie die grundlegenden Eigenschaften und Funktion wiedergeben können. Erarbeiten Sie sich ein Konzept zur Umsetzung.

Spezifikation RISC-V (R/I-Befehle)

Lesen Sie sich in die Register- und Immediate-Befehle in der Referenz (RV32I Base Integer Instruction Set, Version 2.0) ein und eignen Sie sich ein Grundverständnis der Befehlsstruktur so an, dass Sie die Befehlscodierung mithilfe der Referenzkarten wiedergeben und erklären können. Erarbeiten Sie sich ein Konzept zur Umsetzung.

GHDL-Standards

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

Zusatzdateien

Erarbeiten Sie sich die Inhalte der Ihnen zur Verfügung gestellten Zusatzdateien so, dass Sie diese anhand des Codes erklären können.

Aufgaben

Aufgabe 1: Registerfile mit generischer Datenbreite

Ein Registerfile ist eine wichtige Komponente in Prozessoren wie dem RISC-V, die zur Zwischenspeicherung von Daten verwendet wird. In einem RISC-V-Prozessor besteht das Registerfile aus einer Sammlung von Registern, die in der Regel eine Datenbreite von 32 Bits haben und die mit einer 5 Bits breiten Adresse angesteuert werden. Diese Register werden verwendet, um temporäre Daten während der Ausführung von Instruktionen zu speichern.

Das Verhalten des Registerfiles besteht hauptsächlich aus dem Lesen und Schreiben von Daten. In einem Takt-Zyklus kann der Prozessor Daten aus den zwei Registern lesen, um sie für eine Operation zu verwenden, und in einem Register das Ergebnis einer Operation in ein anderes Register zu schreiben. Dies ermöglicht es dem Prozessor, Daten zwischen verschiedenen Teilen eines Programms zu übertragen und Zwischenergebnisse zu speichern. Das Registerfile soll über eine generische Datenbreite *word_width* (Standardwert: *WORD_WIDTH*), Adressbreite *adr_width* (Standardwert: *REG_ADR_WIDTH*) und Anzahl von Registern *reg_amount* (Standardwert: *2**REG_ADR_WIDTH*) verfügen. Das Zero-Register darf nicht überschrieben werden und soll immer 0 zurückgeben.

Für die Abbildung des Registerfiles sollen sie in diesem Versuch sogenannte *types* nutzen:

```
1       type reg_array is array (0 to 3) of std_logic_vector(7 downto 0);
2       signal regs : reg_array := (others => (others => '0'));  -- Array
```

Das Code-Beispiel zeigt die definition des *types* reg_array, welches aus 4 Datenwörtern mit 8 Bit Datenbreite besteht und als Signal genutzt wird.

Unter der angepassten Verwendung dieses *types* kann das Registerfile bei einem Reset auf einen festen Standardwert gesetzt werden (in der Regel alles auf null). Bei einer steigenen Flanke des Taktsignals werden die beiden Registeradressen (*pi_readRegAddr1*, *pi_readRegAddr2*)

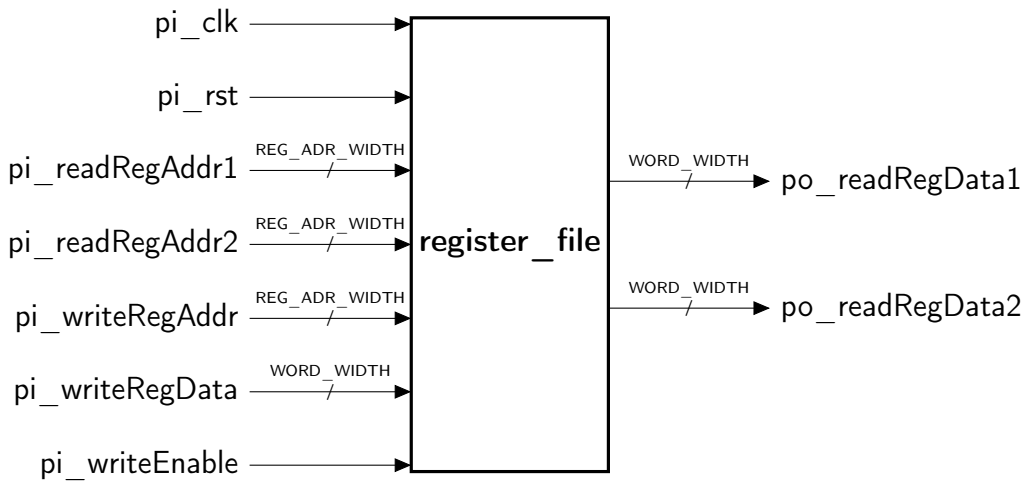

Abbildung 1: register_file

gelesen und auf *pi_readRegData1* und *pi_readRegData2* ausgegeben. Wenn bei der steigenden Flanke zusätzlich das *pi_writeEnable*-Signal auf $'1'$ gesetzt ist, werden die Daten des Dateneingang *pi_writeRegData* auf die Adresse *pi_writeRegAddr* geschrieben.

- Erstellen Sie den Unterordner *<Projektordner>/Komponenten/Registerfile*.
- Erstellen Sie die Entity register_file in der Datei *register_file.vhdl* und setzen Sie das beschriebene Verhalten um.
- TestenSie Ihre Umsetzung mit der Testbench-Entity *register_file_tb* in der Datei *register_file_tb.vhdl* im Unterordner *<Projektordner>/Testbenches/register_file*, welche das register_file für die Datenbreite $5, 6, 8, 16, 32$ testet.
- Erstellen Sie ein Bash-Skript im Ordner *<Projektordner>/Testbenches/register_file*, mit dem Sie die Simulation der Register-File-Testbench inklusive der Analyse und Elaboration automatisch und erfolgreich ausführen können.

Abnahme *Funktion des Registerfiles*, Programmierkonventionen, RISC-V Spezifikation des Befehlssatzes und generische Parameter.

Aufgabe 2: Befehls-Decoder für R-Befehle

Ein Decoder ist eine wichtige Komponente in Prozessoren wie dem RISC-V, die zur Zerlegung des Befehls in Abhängigkeit der Art des Befehls zerlegt. In diesem Versuch beschränkt sich der Decoder auf die R-Befehle und wird sukzessive erweitert. Das Verhalten des Decoders besteht hauptsächlich aus zwei Phasen, dem Erkennen der Art des Befehls und der zerlegten

Ausgabe der Befehls-spezifischen Daten.

Für die Abbildung des Decoders sollen sie in diesem Versuch den sogenannte *record* nutzen. In der Datei *Type_packages.vhdl* ist dazu ein das *controlword* als *record* definiert und beinhaltet alle Befehlsteile für alle Befehle und die Konstante *control_word_init* die den *record* mit Nullen initialisiert.

Bei dem *record* ist für diesen Versuch nur die *ALU_OP* und *I_IMM_SEL* von Interesse. Die *ALU_OP* setzt sich hier aus zwei Teilen des Befehls zusammen der sogenannten **funct7** und **funct3**, wobei von der **funct7** nur das zweite Bit genommen wird um anzuzeigen ob es sich um eine Subtraktion handel oder nicht und entspricht damit den deklarierten der 4-Bit langen Werten der ALU-Operationen im *Constant_package*.

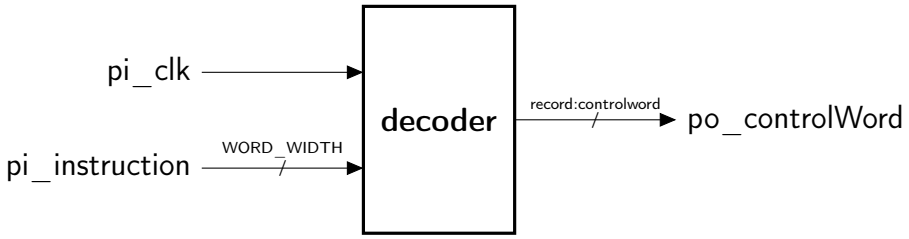

Abbildung 2: decoder

Bei einer steigenden Flanke des Taktsignals *pi_clk* soll aus dem Eingang *pi_instruction* mit 32-Bit Länge auf die Ausgabe *po_controlWord* in Form des eingeführten *controlword* ausgegeben werden. Die Umsetzung soll aus einem Takt-sensitiven Prozess bestehen der in der ersten Phase über *cases* die Art des Befehls identifiziert. In diesem Versuch ist dies beschränkt auf die R-Befehle mit *ADD_OP_INS* und alle anderen. Wenn der OP-Code der Konstanten *ADD_OP_INS* entspricht, soll der Variable *v_insFormat* das *rFormat* aus dem *Type_packages.vhdl* zugeordnet werden.

In einer zweiten Case-Abfrage des Formats im selben Prozess sollen nun die Befehls-Daten gemäß Spezifikation auf den Ausgang *po_controlWord* gesetzt werden.

- Erstellen Sie den Unterordner *<Projektordner>/Komponenten/Decoder*.
- Erstellen Sie die Entity decoder in der Datei *decoder.vhdl* und setzen Sie das beschriebene Verhalten um.
- Überprüfen Sie ihre Umsetzung mit der Testbench *decoder_tb* in der Datei *decoder_tb.vhdl* im Unterordner *<Projektordner>/Testbenches/Decoder* erfolgreich.
- Erstellen Sie ein Bash-Skript im Ordner *<Projektordner>/Testbenches/Decoder*, mit dem Sie die Simulation der Decoder-Testbench inklusive der Analyse und Elaboration automatisch und erfolgreich ausführen können.

Abnahme *Funktion des Decoders*, Programmierkonventionen, RISC-V Spezifikation der Registerbefehle.

Aufgabe 3: Sign-Extension

In RISC-V ist die Sign-Extension wichtig, da die Befehlssätze normalerweise auf eine feste Breite von 32 Bits begrenzt ist, aber die Operanden, mit denen sie arbeiten, möglicherweise kürzer sind. Wenn also ein Wert, der in einem Befehl verwendet wird, kürzer als die Breite des Befehls ist, muss er auf die volle Breite des Befehls erweitert werden, damit das Vorzeichen korrekt behandelt wird.

Die Sign-Extension in RISC-V erfolgt grundsätzlich wie folgt:

Zunächst wird der Wert aus dem Speicher oder aus einem Register geladen. Anschließend wird das Vorzeichenbit des Werts auf die volle Breite des Zielregisters oder der Zieloperation erweitert. Durch die Sign-Extension wird sichergestellt, dass das Vorzeichen des ursprünglichen Werts korrekt beibehalten wird, wenn er auf eine größere Bitbreite erweitert wird. Das bedeutet, dass ein positiver Wert immer noch positiv ist und ein negativer Wert immer noch negativ ist, nachdem er erweitert wurde.

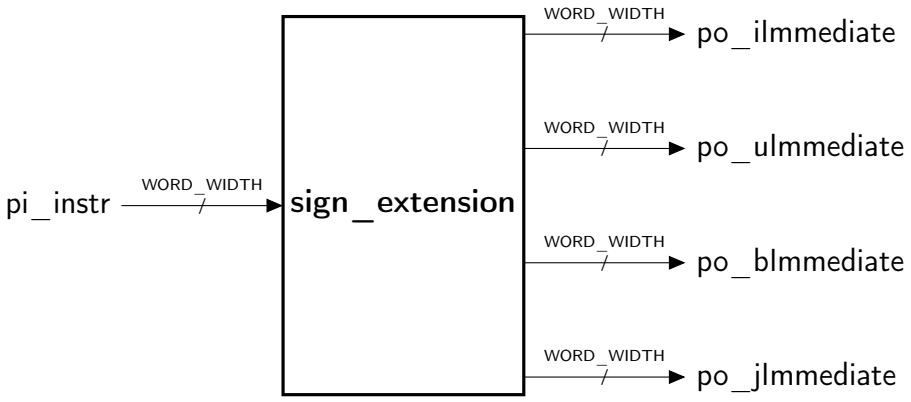

Abbildung 3: sign_extension

Die von Ihnen umzusetzende Sign-Extension soll über einen Eingang in Breite der Befehle *pi_instr* verfügen und alle vier möglichen Immediate-Werte (*po_iImmediate*, *po_uImmediate*, *po_bImmediate*, *po_jImmediate*) gemäß Spezfikation des RISC-V ausgeben.

- Erstellen Sie den Unterordner *<Projektordner>/Komponenten/SignExtender*.
- Erstellen Sie die Entity decoder in der Datei *sign_extender.vhdl* und setzen Sie das beschriebene Verhalten um.
- Überprüfen Sie ihre Umsetzung mit der Testbench *sign_extender_tb* in der Datei *decoder_tb.vhdl* im Unterordner *<Projektordner>/Testbenches/SignExtender* erfolgreich.

- Erstellen Sie ein Bash-Skript im Ordner *<Projektordner>/Testbenches/SignExtender*, mit dem Sie die Simulation der Sign-Extender-Testbench inklusive der Analyse und Elaboration automatisch und erfolgreich ausführen können.

Abnahme *Funktion der Sign-Extension*, Programmierkonventionen, RISC-V Spezifikation der Registerbefehle.

Aufgabe 4: Abgabe

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

Name_Vorname_Versuch2.zip in Moodle hoch.