4. Blatt	Fachgebiet Architektur eingebetteter Systeme **Rechnerorganisation Praktikum**	AES Embedded Systems Architecture WS2022/23

Ausgabe: 20. November 2023

Abgaben

Theorie	entfällt
Praxis	26. November 2023
Rücksprache	27./28. November 2023

Aufgabe 1: Sign-Extension (2 Punkte)

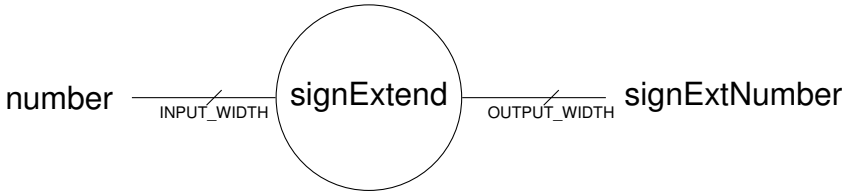

Abbildung 1: Entity signExtend

Name	Typ	in / out	Beschreibung
INPUT_WIDTH	generic	integer	Breite des Eingangs
OUTPUT_WIDTH	generic	integer	Breite des Ausgangs
number	in	signed(INPUT_WIDTH-1 downto 0)	zu erweiternde Zahl
signExtNumber	out	signed(OUTPUT_WIDTH-1 downto 0)	erweiterte Zahl

Das Ziel dieses Praktikums ist die Implementierung eines MIPS-kompatiblen Prozessors. Diese MIPS-CPU verarbeitet Befehle, welche zum Teil Zahlen beinhalten. Da die CPU eine 32-Bit-Architektur ist, sind Daten (und auch Befehle) 32 Bit breit. Somit müssen Zahlen, welche aus den Befehlen extrahiert werden (sogenannte *Immediates*), notwendigerweise kleiner als 32 Bit sein. In unserer CPU sind dies 16 Bit im Gegensatz als die zur Weiterverarbeitung benötigten 32 Bit.
Insofern müssen wir diese Zahlen um 16 Bit "strecken", d.h. nach vorne hin erweitern. Die Zahl soll jedoch nur anders dargestellt werden, der (interpretierte) Zahlenwert soll erhalten bleiben. Bedenken Sie, dass alle Zahlendarstellungen in der MIPS-CPU Zweierkomplement-Darstellungen sind!

1. Implementieren Sie zur gegebenen `entity signExtend` die `architecture behavioral` in der Datei `signExtend.vhd`.
   Verwenden Sie dazu keine Funktionen aus der Library `numeric_std`.
2. Testen und verifizieren Sie Ihr Design mithilfe der vorgegebenen Testbench, indem Sie im Aufgabenordner das Kommando `make clean all` ausführen.

Aufgabe 2: Links-Shifter (2 Punkte)

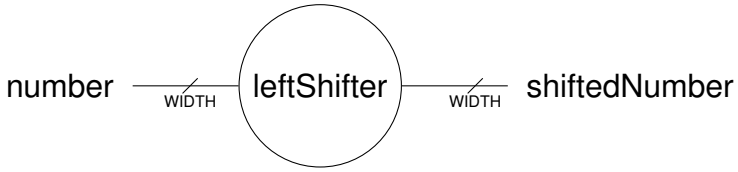

Abbildung 2: Entity leftShifter

Name	Typ	in / out	Beschreibung
WIDTH	generic	integer	Breite des Eingangs
SHIFT_AMOUNT	generic	integer	Weite des Shifts (Verschiebung in Bit)
number	in	std_logic_vector(WIDTH-1 downto 0)	zu shiftende Zahl
shiftedNumber	out	std_logic_vector(WIDTH-1 downto 0)	geshiftete Zahl

Da die MIPS-CPU eine 32-Bit-Architektur ist, werden aus dem Speicher immer 4 Byte geladen, um ein 32-Bit-Wort zu bilden. Daher werden die zwei niederwertigsten Bits einer Byte-Adresse nicht benötigt (sie sind immer `"00"`) Aus Optimierungsgründen werden diese Bits daher in manchen Situationen weggelassen. Wir müssen nun eine Schaltung entwerfen, welche diese Wort-Adresse mithilfe einer Schiebeoperation nach links zurück in eine Byte-Adresse umwandelt.

1. Implementieren Sie zur gegebenen `entity leftShifter` die `architecture behavioral` in der vorgegebenen Datei `leftShifter.vhd`. Verwenden Sie dazu keine Funktionen aus der Library `numeric_std`.
2. Testen und verifizieren Sie Ihr Design mithilfe der vorgegebenen Testbench, indem Sie im Aufgabenordner das Kommando `make clean all` ausführen.

Aufgabe 3: 32-Bit-Multiplizierer (6 Punkte)

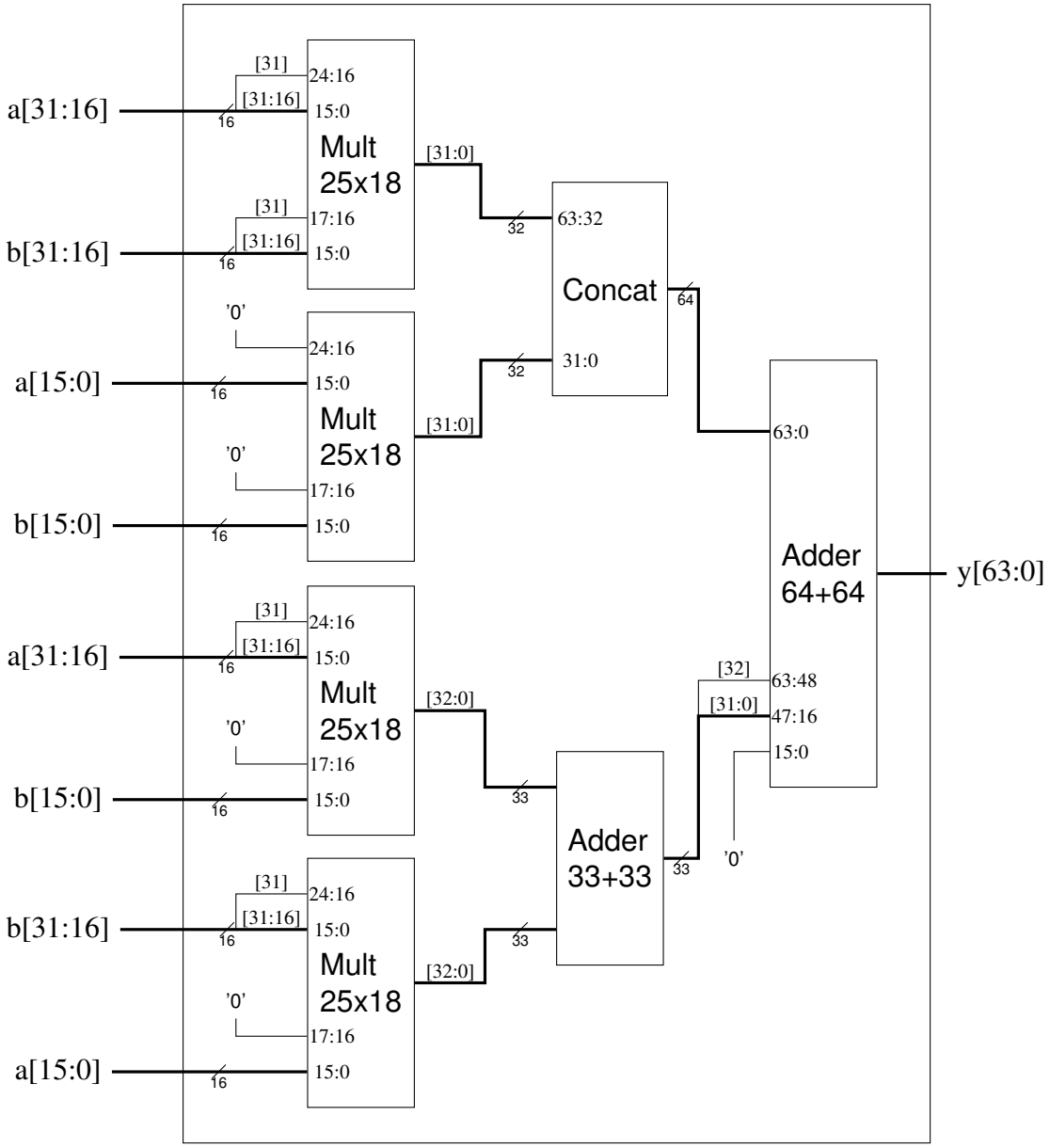

Abbildung 3: RTL 32 Bit-Multiplizierer

Für unsere 32 Bit-Architektur wird ein Multiplizierer benötigt, welcher zwei vorzeichenbehaftete 32 Bit-Operanden multipliziert. Das Ergebnis ist 64 Bit breit.
Zur Realisierung des Multiplizierers stehen vorgegebene Multiplizierer bereit, bei dem ein Operandeneingang eine Breite von 25 Bit und der andere eine Breite von 18 Bit aufweist. Diese sollen nun wie in Abbildung 3 gezeigt verschaltet werden, um den benötigten 32 Bit Addierer zu erhalten.

Zur Realisierung der Funktionalität werden vier 16x16-Multiplikationen benötigt. Die Multiplizierer berechnen das Produkt der unteren Hälfte der beiden Eingangsbits, der oberen Hälften der beiden Eingangsbits und der Kombination von oberen und unteren Eingangsbits. Die untere Hälfte der Eingabe (Bits 0 bis 15) kann immer als vorzeichenlos betrachtet werden, während die oberen Bits (16 bis 31) als vorzeichenbehaftet betrachtet werden müssen. Da die Eingänge des vorgegebene Multiplizierers breiter sind und der Multiplizierernur mit vorzeichenbehafteten Zahlen arbeitet, wird entsprechend eine

Vorzeichenerweiterung durchgeführt. Die sich ergebenen Partialsummen werden im weiteren Verlauf auf möglichst effiziente Weise addiert. Da jeweils nicht die gesamte Breite der 25x18-Multiplizierer genutzt wird, können auch Teile der Ausgänge verworfen werden.

Das Ergebnis der Multiplikation der beiden oberen und unteren Eingangsbits kann konkateniert werden, sodass aus den beiden auf 32 Bit gekürzten Vektoren ein 64 Bit-Vektor entsteht. Die 32 Bit des Ergebnis der Multiplikation der oberen Hälften (vorzeichenbehaftet) werden der oberen Hälfte des konkatenierten Vektors zugewiesen. Die unteren Bits des konkatenierten Vektors (nicht vorzeichenbehaftet) entsprechen dem Ergebnis der Multiplikation der unteren Hälften.
Die kombinierten Produkte werden (vorzeichenbehaftet) werden addiert. Das Ergebnis dieser Addition wird verwendet um einen zweiten 64 Bit Vektor zu erzeugen, dessen Bits 47 bis 16 aus den unteren 32 Bit des Additionsergebnisses bestehen. Die verbleibenden unteren 16 Bits werden auf den Wert `'0'` gesetzt und alle Bits im Bereich 63 bis 48 werden auf das höchstwertigste Bit des Additonsergebnis gesetzt (Bit 32). Abschließend werden die beiden 64 Bit Vektoren addiert um das Endergebnis zu erhalten.

Beachten Sie bei der Implementierung den richtigen Umgang mit vorzeichenlosen (`unsigned`) und vorzeichenbehafteten (`signed`) Zahlen.
Nutzen Sie keine `integer` für Berechnungen, da ihr Wertebereich für viele der Operationen nicht ausreichend ist!

1. Implementieren Sie die in Abbildung 3 gezeigte Funktionalität in der `architecture structural` in der Datei `mult32x32.vhd`.
2. Testen und verifizieren Sie Ihr Design mithilfe der vorgegebenen Testbench, indem Sie im Aufgabenverzeichnis das Kommando `make clean all` aufrufen.

Literatur

[1] ROrgPr Team. Rorgpr Übersicht. `https://rorgpr.gitlab-pages.tu-berlin.de/material/`.