Aufgabenblatt 1

Einführung VHDL/Implementierung und Validierung einer ALU

1 Einfacher Zähler mit variabler Schrittweite

Allgemein

Der erste Versuchsteil befasst sich mit den Grundlagen der VHDL Syntax und -Sematik und der Inbetriebnahme des HDL-Simulators. Dazu ist ist die VHDL-Umsetzung eines einfachen Zählers mit variabler Schrittweite (Offset) betrachtet. Hierzu finden Sie in TEAMs entsprechende Vorlagen für das Design (`increment.vhd`) und eine zugehörige Testbench. Das Design umfasst einen einfachen Zähler, der jeden Takt um den extern vorgebbaren Wert `added` erhöht wird.

Aufgabenstellung

1. Schauen Sie sich die VHDL-Dateien an und versuchen Sie die Funktionsweise nachzuvollziehen.
2. Öffnen Sie den Simulator (installiert auf den Laborrechnern) und versuchen Sie beide Dateien zu kompilieren und anschließend die Simulation zu starten.
3. In die `increment.vhd` haben sich ein paar (vier) Fehler eingeschlichen. Versuchen Sie die Fehler zu beheben und notieren Sie sich Fehler und Ihre Lösung. Wiederholen Sie im Anschluss Schritt 2. Sobald keine Fehler mehr während der Kompilierung auftreten können Sie die Simulation starten und sich die Ergebnisse als Waveform anzeigen lassen
4. Zur Abnahme des ersten Versuchsteils müssen Sie den Aufbau beider VHDL-Dateien erklären können und das Design muss vollständig fehlerfreies Verhalten aufweisen.

2 Implementierung ALU

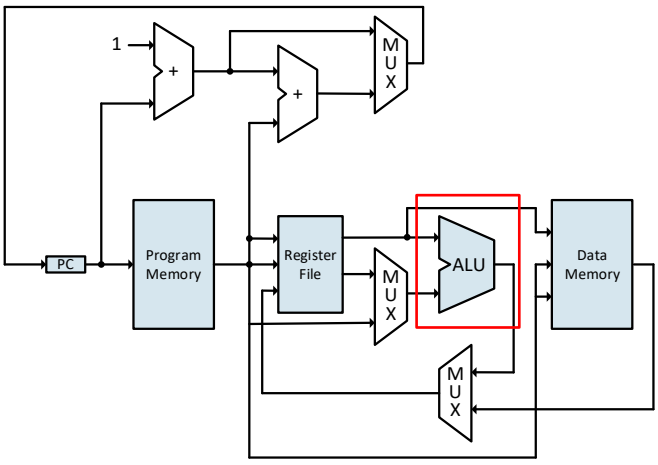

Abb.1: *Grundlegender Aufbau einer RISC-Architektur*

Allgemein

In diesem Versuchsteil wird die HDL-Umsetzung einer arithmetisch, logischen Einheit (ALU, siehe rot umrandetes Element in Abb. 1) betrachtet. Der Funktionsumfang der ALU soll die Berechnung von 1-Operand- bzw. 2-Operand-Befehlen umfassen. Folgende Operationen sind dabei zu berücksichtigen:

Tab.1: *Operationssatz der ALU*

Operation		***OpCode (binär)***	***Funktion***
Addition	(ADD)	000	`<dst> := <src0> + <src1>`
Subtraktion	(SUB)	001	`<dst> := <src0> - <src1>`
Binärer Linksshift	(SHL)	010	`<dst> := <src0> << 1`
Binärer Rechtsshift	(SHR)	011	`<dst> := <src0> >> 1`
Logisches UND	(AND)	100	`<dst> := <src0> AND <src1>`
Logisches ODER	(OR)	101	`<dst> := <src0> OR <src1>`
Logisches XOR	(XOR)	110	`<dst> := <src0> XOR <src1>`
Logisches NICHT	(NOT)	111	`<dst> := NOT <src0>`

Die Datenpfadbreite beträgt Eingangs- wie Ausgangs-seitig 16-bit. Die Eingangsoperanden `<src0>`, `<src1>` werden dabei direkt verarbeitet, d.h. ohne Zwischenspeicherung in Registern; Das Resultat `<dst>` soll in einem Register zwischengespeichert werden.

Neben der Ergebnisausgabe sollen zudem Statusbits (flags) ausgegeben werden. Hierfür soll ein Steuerregister SREG implementiert werden, welches die folgenden 4 Informationen speichert:

Bits

3	2	1	0
Zero	***Negative***	***Carry***	***Overflow***[1]

Die einzelnen Flags lassen sich dabei wie folgt berechnen (bekannt auch aus GT/RMP):

Tab.2: *Berechnung der Flags*

Flag	***Condition***
Zero	`'1'`: wenn alle Bits in `<dst> == '0'` `'0'`: sonst
Negative	`'1'`: wenn das oberste Bit von `<dst> == '1'` ist `'0'`: sonst
Carry	c_8: bei Addition/Subtraktion (siehe Abb. 2) a_7: bei Shift Left (siehe Abb. 2) a_0: bei Shift Right (siehe Abb. 2)
Overflow	c_8 XOR c_7-1 (siehe Abb. 2)

[1] ***Hinweis:*** *Overflow kann in diesem Versuch vernachlässigt werden*

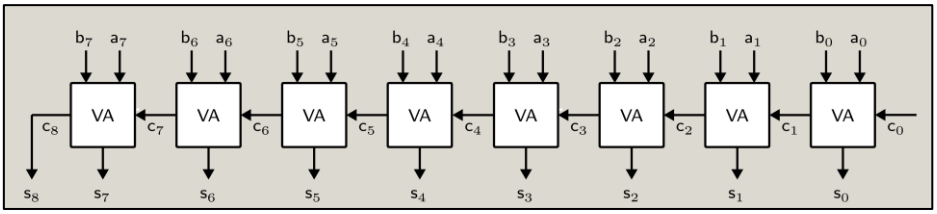

Abb. 2: *Beispiel eines 8-Bit Ripple Carry Addierers (n=8), mit a (<src0>) und b (<src1>) als Eingangsoperanden, s (<dst>) als Ausgangsoperand und c_i als Carrybit an der i-ten stelle*

Aufgabenstellung

Die ALU soll als synthetisierbare VHDL-Beschreibung umgesetzt werden und die Funktionalität mittels Testbench validiert werden. Hierzu finden Sie entsprechende Vorlagen in TEAMs (`alu.vhd` bzw. `alu_tb.vhd` mit `stimuli.txt`). Gehen Sie dabei wie folgt vor:

1. Schauen Sie sich die VHDL-Dateien an und versuchen Sie die Funktionsweise nachzuvollziehen.
2. Vervollständigen Sie die `alu.vhd`, so dass die ALU zunächst nur eine Addition (mit Register am Ausgang) ausführt.
3. Testen Sie Ihre Implementierung durch Verwendung der Testbench und der `stimuli.txt`, aus der sowohl die Daten für „Stimuli“ und „Response Control“ befinden. Erweitern Sie dazu auch die `stimuli.txt`, so dass jeder Befehl einmal getestet wird. Hinweis: Damit keine *assertions* für das Status Register auftreten, können Sie diesen Befehl zunächst auskommentieren.
4. Implementieren und Testen Sie die vollständige Funktionalität aus Tab. 1.
5. Implementieren und Testen Sie nun das Status Register mit `Negative-`, `Zero-` und `Carry-`Flag. Erweitern Sie dazu die `stimuli.txt`, so dass das Auftreten jedes Flags einmal getestet wird. Für das Carry diskutieren Sie am besten vorab (mit uns), wie die Funktionalität möglichst hardwareeffizient implementiert werden kann.

Zusatzaufgabe *(optional)*

6. Erweitern Sie das Design, so dass eine beliebige Datenwortbreite `DWIDTH` eingestellt werden kann. Der Parameter kann über `generic` in der `entity` an die ALU übergeben werden