Implementierung eines Softcore-Mikroprozessor für FPGAs

Gruppe 2

Jannik Graef, 3392032, st161399@stud.uni-stuttgart.de
Tobias Weinschenk, 3404690, st161650@stud.uni-stuttgart.de
Jochen Benzenhöfer, 3456431, st166313@stud.uni-stuttgart.de
Alexander Bunz, 3456583, st166212@stud.uni-stuttgart.de
Omar Al Kadri, 3456978, st166418@stud.uni-stuttgart.de
Simon Naß, 3460883, st166318@stud.uni-stuttgart.de
Jonas Unterweger, 3464025, st167417@stud.uni-stuttgart.de

4. April 2022

Der im Rahmen dieses Projektes entstandene VHDL-Code und alle weiteren Ressourcen sind auf Github unter (`https://github.com/graefjk/RO2`) verfügbar.

Inhaltsverzeichnis

1 Motivation

Geschrieben von: Tobias Weinschenk
Dieses Projekt entsteht im Rahmen der Vorlesung Rechnerorganisation 2 an der Universität Stuttgart. Ziel der Vorlesung ist die Grundlagen des Entwurfs digitaler Schaltungen und Hardwarebeschreibungssprachen kennenzulernen. Hierzu soll in Gruppenarbeit ein 8-bit Mikroprozessor in der Beschreibungssprache VHDL erstellt werden. Dieser soll im Optimalfall korrekt auf einem „field-programmable gate array“ (FPGA) ausgeführt werden. Hierfür wird uns ein Xilinx Zybo Z7-10 für Testzwecke zur Verfügung gestellt. Die Entwicklung wird durch die kostenlose Version der Xilinx Vivado 2020.2 Entwiclkungsumgebung gestützt. Zudem sollen einfache Programme in Maschinensprache für den Mikroprozessor entwickelt werden, um dessen Funktionsfähigkeit zu zeigen.

2 Anforderungsschreibung

Realisieren Sie einen Prozessor unter Verwendung der Hardware-Modellierungssprache VHDL für den folgenden Befehlssatz:

- Mnemonic: label: instruction operand1, operand2 ; comment
- 18-bit instruction width
- register names are represented by ‘sx’ or ‘sy’
- x : Register s0 to sF
- y : Register s0 to sF
- Constant values are represented by ‘aaa’, ‘kk’, ‘ss’, ‘p’ and ‘pp’
- aaa : 12-bit address 000 to FFF
- kk : 8-bit constant 00 to FF
- pp : 8-bit port ID 00 to FF
- p : 4-bit port ID 0 to F
- ss : 8-bit scratch pad location 00 to FF

Mit den Befehlen:

- ADD sX,kk
- ADD sX,sY
- ADDCY sX,kk
- ADDCY sX,sY
- AND sX,kk
- AND sX,sY
- COMPARE sX,kk
- COMPARE sX,sY
- INPUT sX,(sY)
- INPUT sX,pp
- JUMP aaa
- JUMP C aaa
- JUMP NC aaa
- JUMP NZ aaa
- JUMP Z aaa
- LOAD sX,kk
- LOAD sX,sY
- OR sX,kk
- OR sX,sY
- OUTPUT sX,(sY)
- OUTPUT sX,pp
- RL sX
- RR sX
- SL0 sX
- SL1 sX
- SLA sX
- SLX sX
- SR0 sX
- SR1 sX
- SRA sX
- SRX sX
- SUB sX,kk
- SUB sX,sY

- SUBCY sX,kk
- SUBCY sX,sY
- TEST sX,kk
- TEST sX,sY
- XOR sX,kk
- XOR sX,sY

2.1 Erweiterungen

Zusätzlich zu den geforderten Befehlen haben wir uns entschlossen einen Scratchpad-RAM, sowie einen CALL/RETURN-Stack zur Erstellung von Subroutinen zu implementieren und hierfür die folgenden Befehle ergänzt:

- CALL aaa
- CALLC aaa
- CALLNC aaa
- CALLNZ aaa
- CALLZ aaa
- RETURN
- RETURNC
- RETURNNC
- RETURNZ
- RETURNNZ
- FETCH sX, ss
- FETCH sX, (sY)
- STORE sX, ss
- STORE sX, (sY)

Die weitere Spezifikation dieses Befehlssatz zeigen wir in Kapitel 3.3 auf.

3 Architekturbeschreibung

3.1 Blockdiagramm

Hauptverantwortlich: Simon Naß.

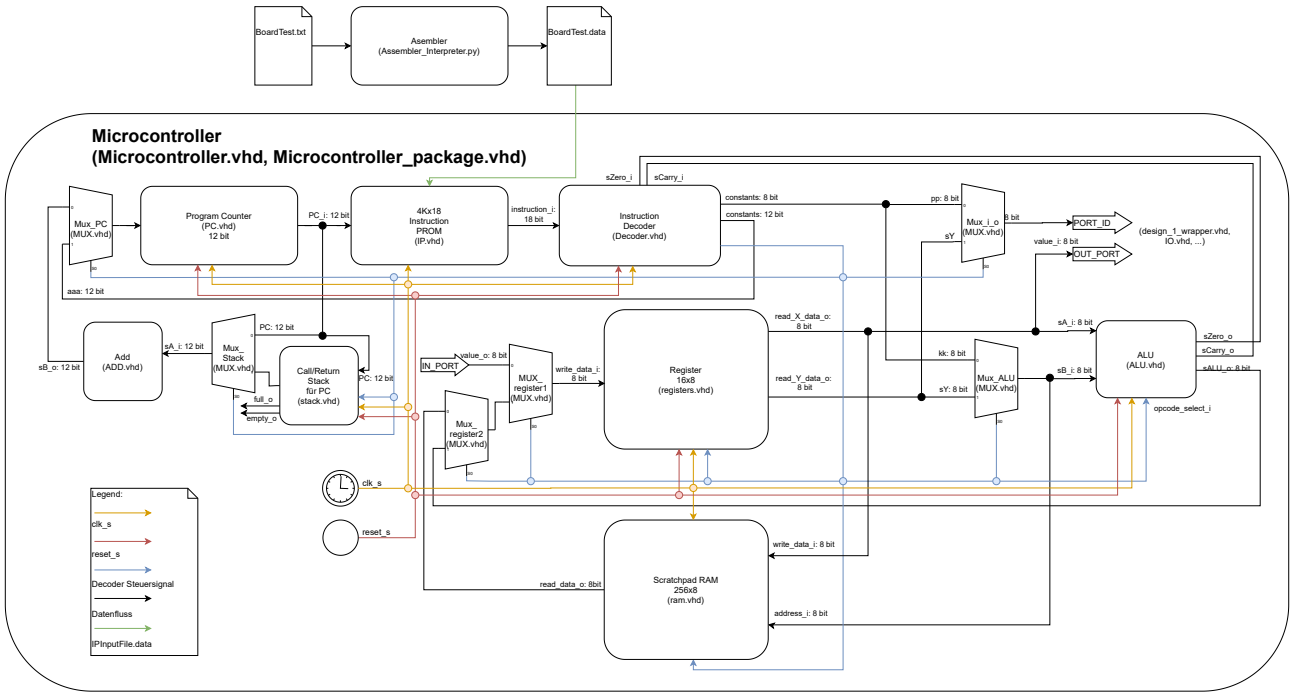


Abbildung 1: Top-Level Blockdiagramm des Mikroprozessors

Die Abbildung zeigt die einzelnen Funktionsblöcke des Mikroprozessors, deren Verkabelung und Interaktionen sowie die Verknüpfung zum Assembler. Schwarze Pfade zeigen hierbei einen Datenfluss zwischen den Blöcken an, während die blauen Pfade Steuerbefehle des Instruction Decoders anzeigen. Alle Blöcke die eine Clock benötigen sind an die gelbe Leitung angeschlossen und alles was eine Reset-Funktion enthält an die rote Leitung.

3.2 Blockbeschreibungen

Im folgenden werden die einzelnen Funktionsblöcke des Mikroprozessors mit deren Ein-/Ausgänge, sowie deren Aufgabe innerhalb des Mikroprozessors beschrieben.

3.2.1 Arithmetic Logic Unit (ALU)

Hauptverantwortlich: Alexander Bunz, Omar Al Kadri, Simon Naß
Die ALU führt Operationen auf zwei Eingabedaten aus wenn das enable_i Signal auf 1 gesetzt ist. Die Operationen umfassen Arithmetische, Logische Verknüpfungen und Shift Operationen. Die Eingabedaten sind entweder Konstante Werte vom Instruction Decoder (kk) oder aus den Registern (sX,sY) geladene Werte. Sie werden an den Eingängen sA_i und sB_i angelegt. Der opcode_select_i Eingang (vom Instruction Decoder), entscheidet darüber, welche Operation auf den Eingabedaten ausgeführt wird. Das Ergebnis der ausgeführten Operation wird an sALU_o angelegt. Die sZERO_o und sCARRY_o ausgänge werden je nach Operation, entsprechend der unten aufgeführten Tabelle (3.3), gesetzt.

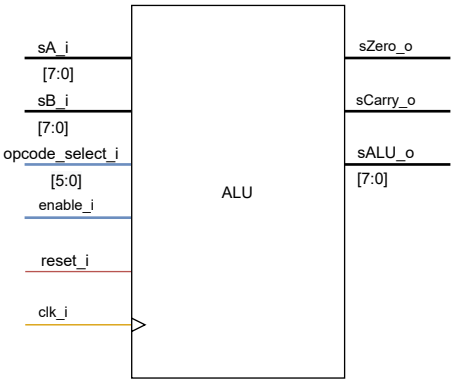


Abbildung 2: Blockdiagramm der ALU

3.2.2 Instruction Decoder

Hauptverantwortlich: Omar Al Kadri
Unterstützt von: Alexander Bunz, Simon Naß
Der Instruction Decoder (ID) dekodiert die aus dem Instruktionsspeicher geladenen Befehle und sendet entsprechende Steuersignale an die einzelnen Komponenten. Die Instruktionen, die vom ID verarbeitet werden, liegen am Eingang instruction_i in form von 18 bit an. Je nach Instruktion, werden die Werte an den Ausgänge des ID entsprechend gesetzt. An sRegister_X_adresse_o und sRegister_Y_adresse_o werden die in den Instruktionen vorhandenen Register Adressen ausgegeben und ans Register weitergeleitet. Enthält die Instruktion konstanten, so werden diese entsprechen an die Ausgänge constant_kk_o (8 bit) und const_aaa_o (12 bit) angelegt. Die mux_... Ausgänge leiten entsprechend der eingegangenen Instruktion Steuerbits an verschiedene Multiplexer weiter. Der sALU_select_o Ausgang gibt den Opcode der Instruction an die ALU weiter. Die ...write_or_read_o Ausgänge signalisieren, ob ein Wert geschrieben oder gelesen werden soll. Die sRegister_write_enable_o, sALU_enable_o, sPC_enable_o, sStack_enable_o, sRAM_write_or_read_o und sRAM_enable_o Ausgänge, steuern den zugriff auf Register, ALU, Stack und RAM.

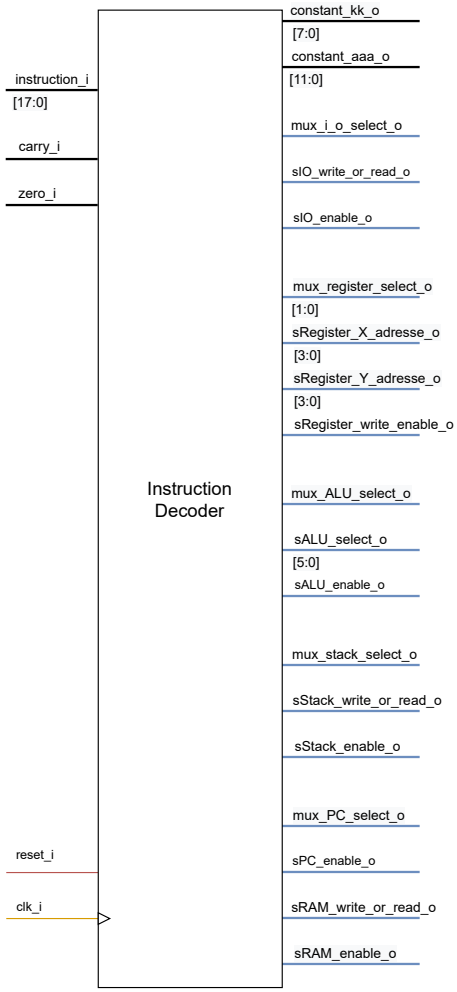


Abbildung 3: Blockdiagramm des Instruction Decoders

3.2.3 Program Counter

Hauptverantwortlich: Jonas Unterweger
Unterstützt von: Simon Naß
Der Program Counter zeigt mit den Outputbits pc_o auf die Position im Instruktionsspeicher, an welcher der aktuelle Befehl steht, der in den Instruktions Decoder geladen werden soll um ausgeführt zu werden. Bei jeder rising edge am clk_i Input werden, falls das enable_i Bit auf 1 gesetzt ist, die momentanen pc_i Bits auf die pc_o Bits übertragen, und damit der nächste Befehl geladen.
Durch ein setzten des reset_i Bits auf 1 kann der Program Counter asynchron auf den ersten Befehl zurückgesetzt werden.

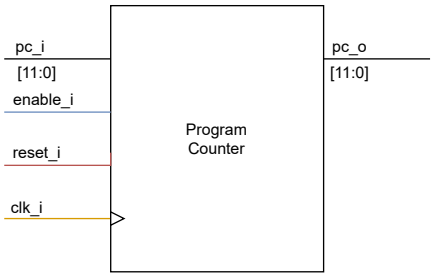


Abbildung 4: Blockdiagramm des Program Counters

3.2.4 Scratchpad RAM

Hauptverantwortlich: Jannik Graef
Unterstützt von: Tobias Weinschenk
Ein zusätzlicher Speicher um Werte zu Speichern. Das write_or_read_i bit bestimmt ob gelesen oder geschrieben wird. Es wird entweder aus dem RAM die Daten an der Adresse address_i auf read_data_o gelesen oder es wird write_data_i auf die Adresse address_i im RAM geschrieben. address_i kommt entweder aus dem Register oder vom Instruction Decoder. write_data_i und read_data_o sind beide mit dem Register verbunden. Es wird nur aus dem RAM gelesen oder in den RAM geschrieben wenn enable_i auf 1 gesetzt ist.

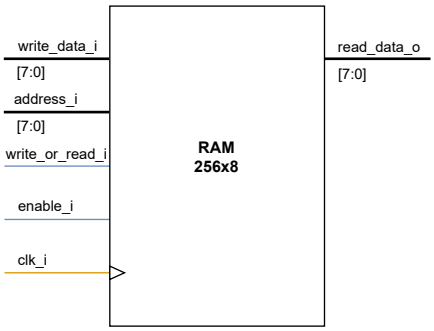


Abbildung 5: Blockdiagramm des Scratchpad RAMs

3.2.5 Register

Hauptverantwortlich: Tobias Weinschenk
Unterstützt von: Jannik Graef
Speichert 8bit Werte. Ergebnisse der ALU werden meist hierhin geschrieben. Das Register hat einen 8-bit breiten Eingang write_data_i um Daten anzunehmen, welche gespeichert werden sollen. Hierzu gibt es noch einen 4-bit breiten Eingang write_address_i um das Register auszuwählen und das write_enable_i bit um das schreiben zu aktivieren. Um aus dem Register zu lesen gibt es zwei Eingänge read_X_address_i und read_Y_address_i um die Register auszuwählen und zwei Ausgänge read_X_data_o read_Y_data_o auf welchen die Daten ausgegeben werden. So können zwei Werte aus dem Register gelesen werden. Über den reset_i Eingang lässt sich das gesamte Register auf 0-Einträge zurücksetzen.

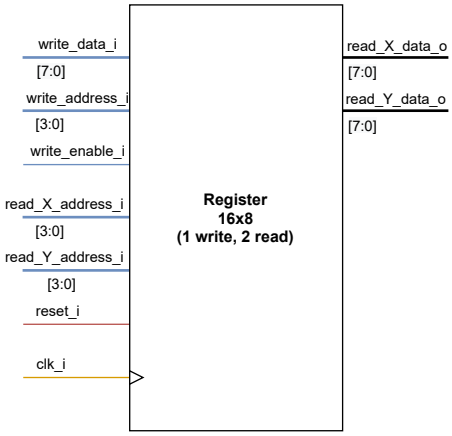


Abbildung 6: Blockdiagramm des Registers

3.2.6 Instruction Memory

Hauptverantwortlich: Jochen Benzenhöfer
Ein Speicher in dem die Instruktionen als 18 Bit Werte gespeichert werden. Der Input(pc_i) beschreibt die Stelle im Speicher, die derzeit ausgeführt wird und deshalb am Ausgang instruction_o für den Instruction Decoder bereitliegt. Ändert sich der Input pc_i ändert sich der Ausgang instruction_o mit der nächsten steigenden Taktflanke von clk_i.

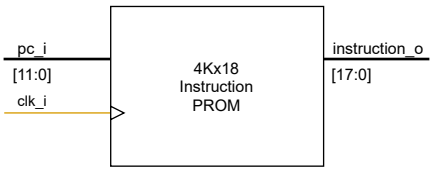


Abbildung 7: Blockdiagramm des Instruction PROM

3.2.7 CALL/RETURN Stack

Hauptverantwortlich: Tobias Weinschenk
Ein Stackspeicher, der die Position des Program Counters vor einem Unterfunktionsaufruf speichert. Dabei werden die bei sPC_i anliegenden Bits auf den Stack gespeichert, wenn write_or_read_i auf 0 gesetzt ist, enable_i auf 1 gesetzt ist und bei clk_i ein rising edge vorliegt. Ist dagegen write_or_read_i auf 1, werden bei der nächsten rising edge von clk_i die oben auf dem Stack liegenden Bits auf den sStack_o Output übertragen und vom Stack gelöscht. Ist enable_i auf 0, so werden weder Daten auf den Stack gelegt, noch welche von ihm genommen.
Durch ein setzten von reset_i auf 1 lässt sich der Stack asynchron resetten und damit leeren.
Über die Ausgänge full_o und empty_o zeigt der Stack, für full_o = 1, dass er voll ist (somit werden weitere Schreibvorgänge verworfen) oder mit empty_o = 1, dass er leer ist (somit werden weitere Lesevorgänge nicht ausgeführt).

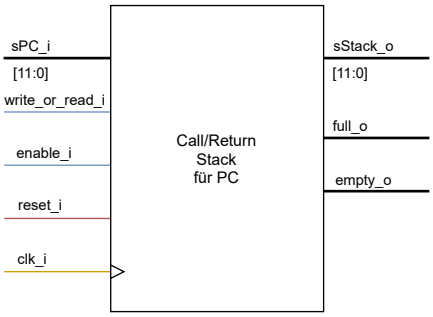


Abbildung 8: Blockdiagramm des CALL/RETURN Stack

3.2.8 Input/Output

Hauptverantwortlich: Jochen Benzenhöfer
Die nach außen zu sendenden Signale(erhalten durch value_i) werden am Port mit der Nummer gegeben durch port_id_i ausgegeben, falls in_out_i und enable_i den Wert 1 enthält. Falls enable_i den Wert 1 und in_out_i 0 enthält, wird der derzeit anliegende Wert beim Port der port_id_i in value_o gespeichert.

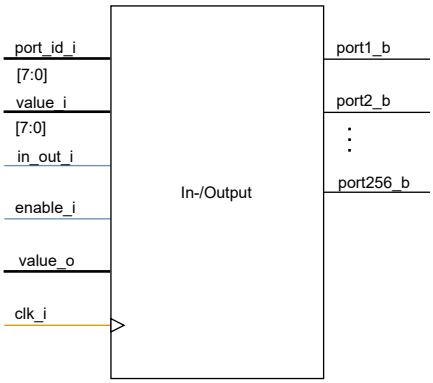


Abbildung 9: Blockdiagramm des Input/Output-Moduls

Port	Beschreibung	Funktion
50-53	Pmod-Eingang	In dem jeweiligen Vektor stellen die Positionen jeweils einen Pin im Anschluss dar. Die verschiedenen Anschlüsse können durch die Portnummer angesprochen werden. Der Wert wird nur aktualisiert, wenn der jeweilige Port zuvor auf empfangen gesetzt wurde.
54	Pmod-In-Out Schalter	Schaltet die Pmodgeräte als Eingang oder Ausgang. 0 Eingabe, 1 Ausgabe. Defaultwert: 0000
55	Resetschalterauswahl auslesen	Defaultwert: 10000000; Gibt aus, welche Knöpfe/Schalter aktuell als reset benutzt werden können.
56-59	alter Pmod-Eingang	In dem jeweiligen Vektor stellen die Positionen jeweils einen Pin im Anschluss dar. Die verschiedenen Anschlüsse können durch die Portnummer angesprochen werden.
60, 61	Gibt die alte Belegung der Knöpfe und Schalter als Vektor aus	Schalter(0 bis 3) in Ausgabe(0 bis 3), Knöpfe(0 bis 3) in Ausgabe(4 bis 7). 61 letzte Belegung, 60 vorletzte Belegung.
62	Gibt die aktuelle Belegung der Knöpfe und Schalter als Vektor aus	Schalter(0 bis 3) in Ausgabe(0 bis 3), Knöpfe(0 bis 3) in Ausgabe(4 bis 7).
63-255	Port-Speicherzugriff	Unbenutzte Ports, welche als Speicher genutzt werden können- die Ausgabeports werden direkt auf die jeweiligen Eingabeports abgebildet. Liest den Speicher assoziiert mit dem Port aus und gibt den Inhalt aus.

Tabelle 1: Portbelegung der Eingabeports

Port	Beschreibung	Funktion
50-53	Pmod-Ausgang	In dem jeweiligen Vektor stellen die Positionen jeweils einen Pin im Anschluss dar. Die verschiedenen Anschlüsse können durch die Portnummer angesprochen werden. Ein Senden ist nur möglich, wenn der Port auf Senden gesetzt wurde.
54	Pmod-In-Out Schalter Ausgabe	Gibt aus, ob die Pmodgeräte als Eingang oder Ausgang genutzt werden. 0 Eingabe, 1 Ausgabe. Defaultwert: 0000
55	Resetschalterauswahl	Defaultwert: 10000000; Eingangswert xnor aktuellerWert = neuerWert, 1 bedeutet, das wenn dieser Knopf/ Schalter den Wert 1 sendet, ein reset signal gesendet wird.
56 - 62	Ausgabe für LEDs	Port(56 bis 59) für LED 0 bis 4, Port(60 bis 62) für RGB von LED 5. Die Eingabe stellt linear die Helligkeit der LED dar. 0 für aus, 255 für maximale Helligkeit.
63-255	Port-Speicherschreiben	Unbenutzte Ports, welche als Speicher genutzt werden können- die Ausgabeports werden direkt auf die jeweiligen Eingabeports abgebildet. Schreibt in den Speicher assoziiert mit dem Port Eingabe.

Tabelle 2: Portbelegung der Ausgabeports

3.3 Instruktionen

Instruktion	Beschreibung	Funktion
ADD sX, kk	Addiert zum Register sX das Literal kk hinzu.	sX ← sX + kk
ADD sX, sY	Addiert zum Register sX den Inhalt aus Register sY hinzu.	sX ← sX + sY
ADDCY sX, kk	Addiert zum Register sX das Literal kk mit Carry-Bit hinzu.	sX ← sX+kk+CARRY
ADDCY sX, sY	Addiert zum Register sX den Inhalt aus Register sY mit Carry-Bit hinzu.	sX ← sX+sY+CARRY
AND sX, kk	Bitweises UND von Register sX mit dem Literal kk.	sX ← sX AND kk
AND sX, sY	Bitweises UND von Register sX mit Register sY.	sX ← sX AND sY
CALL aaa	Bedingungsloser Aufruf der Unterfunktion an der Adresse aaa.	TOS ← PC, PC ← aaa
CALLC aaa	Falls CARRY-Bit gesetzt, Aufruf der Unterfunktion an der Adresse aaa.	if CARRY =1 TOS ← PC, PC ← aaa
CALLNC aaa	Falls CARRY-Bit nicht gesetzt, Aufruf der Unterfunktion an der Adresse aaa.	if CARRY =0 TOS ← PC, PC ← aaa
CALLNZ aaa	Falls ZERO-Bit nicht gesetzt, Aufruf der Unterfunktion an der Adresse aaa.	if ZERO=0 TOS ← PC, PC ← aaa
CALLZ aaa	Falls ZERO-Bit gesetzt, Aufruf der Unterfunktion an der Adresse aaa.	if ZERO=1TOS ← PC, PC ← aaa
COMPARE sX, kk	Vergleicht Register sX mit dem Literal kk. Setzt das CARRY und ZERO flag wie angegeben, Register bleiben dabei unverändert.	if sX = kk ZERO ← 1, if sX < kk CARRY ← 1
COMPARE sX, sY	Vergleicht Register sX mit dem Register sY. Setzt das CARRY und ZERO flag wie angegeben, Register bleiben dabei unverändert.	sX = sY ZERO ← 1, if sX < sY CARRY ← 1
FETCH sX, (sY)	Lese scratchpad RAM von der in Register sY gespeicherten Adresse in Register sX	sX ← RAM[(sY)]
FETCH sX, ss	Lese scretchpad RAM von Adresse ss in Register sX	sX ← RAM[ss]
INPUT sX, (sY)	Lese Wert des Input-Port, welcher vom Register sY spezifiziert wird, in das Register sX.	PORT_ID ← sY, sX ← IN_PORT
INPUT sX, pp	Lese Wert des Input-Port, welcher von pp spezifiziert wird, in das Register sX.	PORT_ID ← pp, sX ← IN_PORT
JUMP aaa	Bedingungsloser Sprung nach aaa.	pc ← aaa
JUMPC aaa	Falls das CARRY-Bit gesetzt ist, springe zu aaa.	if CARRY=1 pc ← aaa
JUMPNC aaa	Falls das CARRY-Bit nicht gesetzt ist, springe zu aaa.	if CARRY=0 pc ← aaa
JUMPNZ aaa	Falls das ZERO-Bit nicht gesetzt ist, springe zu aaa.	if ZERO=0 pc ← aaa
JUMPZ aaa	Falls das ZERO-Bit gesetzt ist, springe zu aaa.	if ZERO=1 pc ← aaa
LOAD sX, kk	Lade das Literal kk in das Register sX.	sX ← kk
LOAD sX, sY	Lade den Inhalt des Registers sY in das Register sX.	sX ← sY
OR sX, kk	Bitweise OR von Register sX mit literal kk.	sX ← sX OR kk
OR sX, sY	Bitweise OR von Register sX mit Register sY.	sX ← sX OR sY

Tabelle 3: Beschreibung und Funktion der Instruktionen

Tabelle 3: Beschreibung und Funktion der Instruktionen

Instruktion	Beschreibung	Funktion
OUTPUT sX, (sY)	Schreibe Register sX zum in sY gespeicherten output Port.	PORT_ID ← sY, OUT_PORT ← sX
OUTPUT sX, pp	Schreibe Register sX zu output Port pp.	PORT_ID ← pp, OUT_PORT ← sX
RETURN	Bedingungslose Rückkehr von der Unterfunktion.	PC ← TOS+1
RETURNC	Falls Carry-Bit gesetzt, Rückkehr von der Unterfunktion.	If CARRY=1, PC ← TOS+1
RETURNNC	Falls Carry-Bit nicht gesetzt, Rückkehr von der Unterfunktion.	If CARRY=0, PC ← TOS+1
RETURNZ	Falls Zero-Bit gesetzt, Rückkehr von der Unterfunktion.	If ZERO=1, PC ← TOS+1
RETURNNZ	Falls Zero-Bit nicht gesetzt, Rückkehr von der Unterfunktion.	If ZERO=0, PC ← TOS+1
RL sX	Rotiert Register sX einen Schritt nach links.	sX ← sX[6:0],sX[7], CARRY ← sX[7]
RR sX	Rotiert Register sX einen Schritt nach rechts.	sX ← sX[0],sX[7:1],CARRY ← sX[0]
SL0 sX	Schiebe Register sX links, mit 0 aufgefüllt.	sX ← sX[6:0],0, CARRY ← sX[7]
SL1 sX	Schiebe Register sX links, mit 1 aufgefüllt.	sX ← sX[6:0],1, CARRY ← sX[7]
SLA sX	Schiebe Register sX links durch alle Bits, inklusive Carry.	sX ← sX[6:0],CARRY, CARRY ⇐ sX[7]
SLX sX	Schiebe Register sX links. Bit sX[0] unbetroffen.	sX ← sX[6:0],sX[0], CARRY ← sX[7]
SR0 sX	Schiebe Register sX rechts, mit 0 aufgefüllt.	sX ← 0,sX[7:1], CARRY ← sX[0]
SR1 sX	Schiebe Register sX rechts, mit 1 aufgefüllt.	sX ← 1,sX[7:1], CARRY ← sX[0]
SRA sX	Schiebe Register sX durch alle bits, inklusive Carry.	sX ← CARRY,sX[7:1], CARRY ← sX[0]
SRX sX	Arithmetisches rechtsschieben von Register sX mit Vorzeichenerweiterung.	sX ← s[7],sX[7:1], CARRY ← sX[0]
STORE sX, (sY)	Schreibe Register sX in scratchpad RAM an der in Register sY gespeicherten Adresse.	RAM[(sY)] ← sX
STORE sX, ss	Schreibe Register sX an Adresse ss in scratchpad RAM.	RAM[ss] ← sX
SUB sX, kk	Subtrahiere literal kk von Register sX.	sX ← sX-kk
SUB sX, sY	Subtrahiere Register sY von Register sX.	sX ← sX-sY
SUBCY sX, kk	Subtrahiere literal kk von Register sX mit Carry.	sX ← sX-kk-CARRY
SUBCY sX, sY	Subtrahiere Register sY von Register sX mit Carry.	sX ← sX-sY-CARRY
TEST sX, kk	Teste Bits in Register sX gegen literal kk. Update Carry und Zero flags. Register bleiben unverändert.	if(sX AND kk)=0 ZERO ← 1, CARRY ← odd parity of (sX AND kk)
TEST sX, sY	Teste Bits in Register sX gegen Bits in Register sY. Update Carry und Zero flags. Register bleiben unverändert.	if(sX AND sY)=0 ZERO ← 1, CARRY ← odd parity of (sX AND sY)
XOR sX, kk	Bitweise XOR von Register sX mit literal kk.	sX ← sX XOR kk
XOR sX, sY	Bitweise XOR von Register sX mit Register sY.	sX ← sX XOR sY

Tabelle 3: Beschreibung und Funktion der Instruktionen

Instruktion	ZERO	CARRY	opcode
ADD sX, kk	if (sX + kk = (0 or 256)): ZERO = 1, else: ZERO = 0	if ((sX + kk) > 255): CARRY = 1, else: CARRY = 0	000001
ADD sX, sY	if (sX + sY = (0 or 256)): ZERO = 1, else: ZERO = 0	if ((sX + sY) > 255): CARRY = 1, else: CARRY = 0	000000
ADDCY sX, kk	if (sX + kk + CARRY = (0 or 256)): ZERO = 1, else: ZERO = 0	if ((sX + kk + CARRY) > 255): CAR- RY = 1, else: CARRY = 0	000011
ADDCY sX, sY	if (sX + sY + CARRY = (0 or 256)): ZERO = 1, else: ZERO = 0	if ((sX + sY + CARRY) > 255): CAR- RY = 1, else: CARRY = 0	000010
AND sX, kk	if (sX AND kk = 0): ZERO = 1, else: ZERO = 0	0	001001
AND sX, sY	if (sX AND sY= 0): ZERO = 1, else: ZERO = 0	0	001000
CALL aaa	-	-	100001
CALLC aaa	-	-	100010
CALLNC aaa	-	-	100011
CALLNZ aaa	-	-	100100
CALLZ aaa	-	-	100101
COMPARE sX, kk	if (sX = kk): ZERO = 1, else: ZERO = 0	if (kk > sX): CARRY = 1, else: CARRY = 0	011011
COMPARE sX, sY	if (sX = sY): ZERO = 1, else: ZERO = 0	if (sY > sX): CARRY = 1, else: CARRY = 0	011010
FETCH sX, (sY)	-	-	010010
FETCH sX, ss	-	-	010011
INPUT sX, (sY)	-	-	010110
INPUT sX, pp	-	-	010111
JUMP aaa	-	-	100110
JUMPC aaa	-	-	100111
JUMPNC aaa	-	-	101000
JUMPNZ aaa	-	-	101001
JUMPZ aaa	-	-	101010
LOAD sX, kk	-	-	001111
LOAD sX, sY	-	-	001110
OR sX, kk	if (sX OR kk = 0): ZERO = 1, else: ZERO = 0	0	001011
OR sX, sY	if (sX OR sY= 0): ZERO = 1, else: ZERO = 0	0	001010
OUTPUT sX, (sY)	-	-	010100
OUTPUT sX, pp	-	-	010101
RETURN	-	-	101011
RETURNC	-	-	101100
RETURNNC	-	-	101101
RETURNZ	-	-	101110
RETURNNZ	-	-	101111
RL sX	if (rl(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110000
RR sX	if (rr(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	110001
SL0 sX	if (sl0(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110010
SL1 sX	if (sl1(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110011
SLA sX	if (sla(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110100
SLX sX	if (slx(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110101

Tabelle 4: Opcode Definition, sowie Carry + Zero-Flag Verhalten der Instruktionen

Tabelle 4: Opcode Definition, sowie Carry + Zero-Flag Verhalten der Instruktionen

Instruktion	ZERO	CARRY	opcode
SR0 sX	if (sr0(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	110111
SR1 sX	if (sr1(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	110111
SRA sX	if (sra(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	111000
SRX sX	if (srx(sX) = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	111001
STORE sX, (sY)	-	-	010000
STORE sX, ss	-	-	010001
SUB sX, kk	if ((sX - kk) = 0): ZERO = 1, else: ZERO = 0	if ((sX - kk) < 0): CARRY = 1, else: CARRY = 0	000101
SUB sX, sY	if ((sX - sY) = 0): ZERO = 1, else: ZERO = 0	if ((sX - sY) < 0): CARRY = 1, else: CARRY = 0	000100
SUBCY sX, kk	if ((sX - kk- CARRY) = (0 or -256): ZERO = 1, else: ZERO = 0	if ((sX - kk - CARRY) < 0): CARRY = 1, else: CARRY = 0	000111
SUBCY sX, sY	if ((sX - sY - CARRY) = (0 or -256): ZERO = 1, else: ZERO = 0	if ((sX - sY - CARRY) < 0): CARRY = 1, else: CARRY = 0	000110
TEST sX, kk	if (sX AND kk = 0): ZERO = 1, else: ZERO = 0	if (parityTest[1](sX And kk) = 1): CAR- RY = 1, else: CARRY = 0	011001
TEST sX, sY	if (sX AND sY = 0): ZERO = 1, else: ZERO = 0	if (parityTest[1](sX And sY) = 1): CAR- RY = 1, else: CARRY = 0	011000
XOR sX, kk	if (sX XOR kk = 0): ZERO = 1, else: ZERO = 0	0	001101
XOR sX, sY	if (sX XOR sY = 0): ZERO = 1, else: ZERO = 0	0	001100

Tabelle 4: Opcode Definition, sowie Carry + Zero-Flag Verhalten der Instruktionen

[1]parityTest() testet, ob die Eingabe eine gerade oder ungerade Anzahl an '1'bit besitzt.

4 Entwicklungsprozess

4.1 ALU

Hauptverantwortlich: Alexander Bunz, Omar Al Kadri, Simon Naß.

In der ALU wurde zu beginn die Shiftbefehle, umgesetzt diese haben wie bereits hier schon in ersten Tests ersichtlich, keine Fehler.
Als nächstes wurden Logikbefehle implementiert. Aufgrund von unklarer Dokumentation des Zero und Cary Signals, traten hier Unterschiede zwischen dem Testergebnis und dem erwarteten Ergebnis auf. Nach Anpassung im ALU Code traten keine weiteren Fehler bezüglich Logikbefehlen auf. Es ist jedoch anzumerken, dass die Dokumentation diesbezüglich nach den ausführlichen externen Tests zur deutlicheren Klarstellung überarbeitet wurde.
Bei den arithmetischen Befehlen wurden Zweifel an der fehlerfreien Funktionsweise auf dem Bord geäußert, wodurch sie auf unterschiedlichste Weise zusätzlichen Prüfungen unterzogen wurden. Es konnten jedoch keine Nachteile dieser Implementierungsweise festgestellt werden. Bei den ausführlichen externen Tests trat ein Fehler bei der Operation SUBCY auf. Hier konnte durch einen Overflow das CARRY nicht richtig ermittelt werden.

```
when operation_SUBCY | operation_SUBCY_kk=>
  result_s <= to_stdulogicvector(to_stdlogicvector(sA_i) - to_stdlogicvector(sB_i) -
  ↪  carry_s);
  if (to_stdlogicvector(sA_i) < (to_stdlogicvector(sB_i) + carry_s)) then
    carry_s <= '1';
  else
    carry_s <= '0';
  end if;
```

Vorheriger Code zeigt die Umsetzung des fehlerhaften Bereich vor Problembehebung in der ALU. Die Korrektur ist im folgenden Codeausschnit dargestellt.

```
when operation_SUBCY | operation_SUBCY_kk=>
  result_s <= to_stdulogicvector(to_stdlogicvector(sA_i) - to_stdlogicvector(sB_i) -
  ↪  carry_s);
  if (('1'&to_stdlogicvector(sA_i))-carry_s) < ('1'&to_stdlogicvector(sB_i))  then
    carry_s <= '1';
  else
    carry_s <= '0';
  end if;
```

4.2 Instruction Decoder

Hauptverantwortlich: Omar Al Kadri.
Unterstützt von: Alexander Bunz und Simon Naß.

Der Instruction Decoder wurde zunächst vollständig innerhalb eines Prozesses implementiert, um je nach Befehl entsprechende Steuersignale abhängig vom Takt zu setzen. Dies erwies sich allerdings aus zweierlei Hinsicht als unvorteilhaft. Zum Einen wurde der Decoder durch viele switch cases sehr groß und unübersichtlich, zum anderen konnten die Enable Signale für die anderen Komponenten nicht in einer bestimmten zeitlichen Abfolge gesetzt werden.
Deshalb haben wir den Decoder so umstrukturiert, dass die einzelnen Enable Signale über eine State Machine gesteuert werden, sodass jeder Block einen Takt Zeit für seine Aufgaben hat. Dadurch wird gewährleistet, dass jeder Block genügend Zeit für seine Berechnungen hat und die Signale für andere Blöcke stabil anliegen. Damit gibt der Decoder auch die Zeit für die Abarbeitung einzelner Befehle vor. Jumps, Calls und Return Befehle benötigen vier Takt Zyklen und alle anderen Befehle benötigen sechs Takt Zyklen.
Das eigentliche Decodieren der Befehle wurde aus dem Prozess heraus verlagert und deutlich vereinfacht. Das Decodieren der Befehle geschieht nun unabhängig vom Takt, dem Decoder wird allerdings von der eigenen State Machine auch ein Takt Zeit dafür gegeben.
Beim Decoder wurden Instruktionen, die es eigentlich nicht gibt (wie z.B. ”1111111111111111”)

zugelassen, da diese eigentlich durch das Assembler Programm gar nicht erst erzeugt werden können. Da solche Befehle dadurch allerdings unvorhergesehenen Einfluss auf ein Programm haben können und wir sicherstellen wollten, dass auch Maschinencode der nicht von unserem Assembler Programm erzeugt wird auf dem Prozessor laufen kann, haben wir uns am Ende noch dazu entschieden den Decoder so anzupassen, dass unbekannte Instruktionen einfach ignoriert werden.
Zuletzt hatten wir noch versucht einen Takt einzusparen, indem wir die Zustände der State Machine von dem Program Counter und dem Instruction PROM zu einem zusammenfassen und den ROM asynchron lesen. Dadurch konnte der Instruction PROM allerdings nicht mehr den Block-RAM des Boards verwenden, weshalb wir dies rückgängig gemacht haben.

4.3 Program Counter

Hauptverantwortlich: Jonas Unterweger.
Unterstützt von: Simon Naß.

Der Program Counter durchlief keine Änderungen, allerdings entstand durch seinen „enable_i“ Eingang in Zusammenarbeit mit dem Decoder eine allgemein wichtige Diskussion welche Blöcke ebenfalls ein „enable_i“ benötigen und wie der Decoder diese steuern soll.
Die relativ späte Implementierung der ADD-Komponente erwies sich als entsprechend einfach. Hier hat man nämlich auf die bereits in der ALU verwendete und getestete Variante gesetzt. Man hat sich jedoch von zwei auf einen Eingang um-entschieden, da man immer um demselben konstanten Wert ('1') inkrementieren wollte. Die ausführlichen externen Tests zeigten keine auffälligen oder fehlerhafte Verhalten. Das Anlegen eines undefinierten Signals führt natürlich in der Simulation zu einer Ungewissheit des Ausgangs, jedoch wird durch andere Komponenten sichergestellt, dass ein solches Signal hier niemals anliegen kann.

4.4 Scratchpad RAM

Hauptverantwortlich: Jannik Graef.
Unterstützt von: Tobias Weinschenk.

Die erste Idee war es den Inhalt des RAM als einzelnen, großen std_ulogic_vector der Größe 2048 bit zu implementieren und immer Zugriff auf 8-bit Abschnitte zu gewähren.

```
signal data  : std_ulogic_vector(2047 downto 0);
...
if (clk_i'event and clk_i = '1' and  enable_i = '1') then
  address := to_integer(unsigned(address_i));
  if(write_or_read_i = '0') then
    read_data_o <= data((address+1)*ram_width_g-1 downto address*ram_width_g);
  else
    data((address+1)*ram_width_g-1 downto address*ram_width_g) <= write_data_i;
  end if;
end if;
```

Es erwies sich jedoch als praktischer die Daten in einem eigenen Type zu speichern. Der definierte Type besteht aus einem 256 Elemente langem Array aus u_logic_vetor der Größe 8:

```
type ram_type is array (2 ** ram_select_size_g -1 downto 0) of std_ulogic_vector(ram_width_g
↪  -1 downto 0);
signal ram_s : ram_type := (others => (others => '0'));
...
if (rising_edge(clk_i) and  enable_i = '1') then
                if(write_or_read_i = '1') then
                    read_data_o <= ram_s(to_integer(unsigned(address_i)));
                else
                    ram_s(to_integer(unsigned(address_i))) <= write_data_i;
                end if;
            end if;
```

Wie im gezeigten Code schon ersichtlich wurde der RAM zudem mit Generics versehen, um die Größe schnell anpassen zu können. Hierzu gibt es zwei Generics um die Größe zu steuern: „ram_witdh_g “ gibt die Wortbreite des RAMs an, „ram_select_size_g “ gibt die Anzahl an Addressierungs-bits an. Das Array bekommt dann die Größe $2^{ram\_select\_size\_g}$.
Zudem wurde das Attribut „ram_style “ hinzugefügt, welches Vivado anweist einen bestimmten RAM-Typ zu synthetisieren. Zur Auswahl stehen, „registers “(Flip-Flops), „distributed “(Look-Up-Table RAM), sowie „block “(Block-RAM). Auch dieses Attribut kann per Generic schnell verändert werden. Am sinnvollsten scheint uns aber „distributed “, da Block-RAM nur in 32kB Blöcken verfügbar ist und somit mit den 2kB RAM kaum sinnvoll ausgelastet wird und auf der anderen Seite werden mit „registers “ unnötig viele Flip-Flops verbraucht werden. Der LUT-RAM scheint daher angebracht. Im Testbench haben sich bei dieser Architektur keine Fehler gezeigt.

4.5 Register

Hauptverantwortlich: Tobias Weinschenk.
Unterstützt von: Jannik Graef.

Da das Register in Zusammenarbeit mit dem RAM entstand, wurden hier die selben Überlegungen wie beim RAM angestellt. Auch hier war die Überlegung zunächst den gesamten Inhalt in einem Signal der Größe 128 bit zu speichern, welche schnell durch den eignen Type ersetzt wurde.

```
 type register_type is array (2 ** register_select_size_g -1 downto 0) of
 ↪  std_ulogic_vector(register_width_g -1 downto 0);
signal register_s : register_type := (others => (others => '0'));
...
if (reset_i = '1') then
                    register_s <= (others => (others => '0'));
                    read_X_data_o <= (others => '0');
                    read_Y_data_o <= (others => '0');
                elsif (rising_edge(clk_i)) then
                    read_X_data_o <= register_s(to_integer(unsigned(read_X_address_i)));
                    read_Y_data_o <= register_s(to_integer(unsigned(read_Y_address_i)));
                    if(write_enable_i = '1') then
                        register_s(to_integer(unsigned(write_address_i))) <= write_data_i;
                    end if;
                end if;
```

Wie im Code ersichtlich, ist der Prozess des Registers, aufgrund der anderen Anforderungen, etwas anders aufgebaut. Zunächst, gibt es beim Register einen asynchronen Reset, welcher auch die Inhalte des Registers zurücksetzt. Außerdem gibt es zwei Ausgabeports, welche gleichzeitig gelesen werden können. Aber auch wie beim RAM wurde die Größe über die Generics „register_widt_g “ (Wort-Breite) und „register_select_size_g “ (Anzahl Addressierungs-bits) gesetzt. Durch den vorhandenen asynchronen Reset, viel die Überlegung, welcher Speichertyp verwendet werden soll hier allerdings weg, da hierdurch nur noch Flip-Flops verwendet werden können. Was aber auch sehr gut zum Aufgabenbereich und der Größe des Moduls passt. Auch hier haben sich in ausführlichen externen Tests keine Fehler gezeigt.

4.6 Instruction Memory

Hauptverantwortlich: Jochen Benzenhöfer.

Die erste Idee, das Programm für den Prozessor auf dem NOR-Speicher des Boards oder einer SD-Karte abzulegen und von dort aufzurufen klingt im ersten Moment sehr verlockend. Nach etwas Recherche stellte sich jedoch heraus, dass diese Speichermethoden nicht wirklich trivial sind. So lässt der NOR-Speicher nur über das PS (Processing System) auslesen und die SD-Karte liefert nur sehr langsam Daten.
Zur Umsetzung müsste somit ein RAM-Speicher als Zwischenspeicher erstellt werden. Deshalb wurde der Speicher komplett durch BRAM umgesetzt. Dieser ist auf dem Board ausreichend verfügbar, muss

jedoch auch bei der Synthese initialisiert werden. Grundsätzlich wäre es auch möglich den PROM durch DRAM zu implementieren, durch die große Größe schien dies jedoch nicht wirklich praktikabel. Die Befürchtung, dass die Verzögerung durch Nutzung des BRAMs zu hoch sein könnte konnte in der Praxis nicht bestätigt werden, weshalb kein Cache zur Unterstützung vollständig implementiert wurde.
Die ausführlichen externen Tests ergaben, dass keine Fehler durch die Implementierung des Instruction Memorys mit dem BRAM auftreten. Kritisch ist jedoch der Inhalt der in das Instruction Memorys geladen werden kann. Hier sind sowohl im Assembler als auch im Instruction Decoder Maßnahmen vorhanden um einen fehlerhaften Kontrollfluss und unbekannte Instruktionen abzufangen.

4.7 Call/Return Stack

Hauptverantwortlich: Tobias Weinschenk.

Bei der Entwicklung des Stacks erwiesen sich die gewonnenen Kenntnisse über die unterschiedlichen Speicherarten und die Definition eines eigenen Types aus der Entwicklung des RAMs als sehr nützlich und wurden direkt umgesetzt. Da der Stack eine Tiefe von 128 besitzt und 12 Bits pro Eintrag Speichern soll, haben wir uns aufgrund der Speichermenge von 1,5 KiB auch hier für den LUT-RAM entschieden. Aber auch beim Stack lässt sich der verwendete RAM-Typ, sowie die Wort-Breite als auch die Tiefe per Generics schnell ändern.
Die Funktionsweise des Stacks wurde komplett in einem Prozess unter der Verwendung von Variablen für den Stack-Pointer, dem Empty-Bit und dem Full-Bit umgesetzt. Allerdings sind beim ersten Test zwei Probleme aufgefallen. Das erste Problem war der Stack-Pointer, welcher die untere Ebene im Stack nie beschrieben hat. Dies kam dadurch zustande, dass der Stack-Pointer auf 0 initialisiert wird und vor jedem Schreiben um eins inkrementiert wird. Dadurch konnte die Ebene 0 nie beschreiben werden. Dies konnte durch eine Verschiebung bei der Definition des Arrays behoben werden. Somit wurde aus der Zeile:

```
type stack_type is array (stack_depth_g -1 downto 0) of std_ulogic_vector(
↪  instruction_address_g -1 downto 0);
```

folgende Zeile:

```
type stack_type is array (stack_depth_g downto 1) of std_ulogic_vector(
↪  instruction_address_g -1 downto 0);
```

Beim zweiten Problem handelte es sich um den asynchronen Reset, da dieser zunächst auch den gesamten Inhalt des Stacks zurücksetzen sollte. Dies ist allerdings beim LUT-RAM, sowie Block-Ram nicht möglich. Daher haben wir uns entschieden, beim asynchronen Reset den Inhalt des Stacks nicht zu löschen, da es bereits ausreicht, den Stack-Pointer, sowie den Ausgang, das Empty-Flag und das Full-Flag zurückzusetzen. Ein Lesen bevor etwas geschrieben wird, ist aufgrund der Logik innerhalb des Prozesses nicht möglich.
Weiterhin gab es noch Diskussionen über die Implementierung per Variablen. Es stellte sich die Frage, ob diese Implementierung sich auch nach der Synthese korrekt verhält. Hierbei gab es aber erneut Probleme. Der erstellte Testbench verwendet wie der Stack selbst den Datentyp std_ulogic, allerdings wandelt Vivado den Stack bei der Synthese in den Datentyp std_logic um, weshalb Vivado eine Fehlermeldung bezüglich der nicht übereinstimmenden Datentypen erzeugte. Dies konnte durch umschreiben des Testbenches auf den Datentyp std_logic behoben werden. Hier zeigte sich nun auch, dass die Implementierung mit Variablen für den Stack-Pointer, sowie die Flags sich auch nach der Synthese verhält wie erwartet.

Ausführlichen externen Tests des Stacks ergaben eine fehlerfreie normaldurchlauf. Gerade das auslesen auf leerem Stack und schreiben auf vollem Stack sind interesante Randfälle. Betrachtet man das verhalten hier im Stack als alleinstehenden Block, so tut dieser das bestmögliche mit seinen „full_o“und „empty_o“Ausgangssignalen, sowie internen Stack verhalten. Allerdings wird zur derzeitigen Lage keine Fehlerbehandlung dieser Randfälle auf dem Board durchgeführt. Dies erfordert also eigene Aufmerksamkeit bei dem Programmieren der auf dem Board laufenden Programme.
Der vom Stack ausgegebene Resetwert war zwar in sich logisch wurde jedoch auf „111111111111“abgeändert um im Gesamtkonzept mit dem Resetwert des Program Counters übereinzustimmen.

4.8 Input/Output

Hauptverantwortlich: Jochen Benzenhöfer.

Der einfachste Weg den Block umzusetzen wäre, die Daten durch die gegebene Port_id zu den Prozessorpins zu multiplexen, jedoch fallen durch dieses System viele Probleme an.

1. Die Port_id ist eine 8-Bit Zahl, wodurch bis zu 256 verschiedene Ports adressiert werden könnten, der Prozessor hat jedoch nur 230 Pins, welche jedoch nicht alle durch den FPGA-Teil direkt erreichbar sind. Somit musste ein Verhalten des Blocks bei unbenutzter Port_id definiert werden, oder die unbenutzten Ports anderweitig verwendet werden.

2. Der Prozessor geht davon aus, dass jeder Port bidirektional verfügbar ist. Dies ist jedoch nicht für jeden Pin der Fall, wodurch verschiedene Pins zu einem Port verbunden werden müssen, was 1. verstärkt. Eine weitere Lösung wäre, die ungenutzten unidirektionalen Ports anderweitig zu nutzen. Dies zerstört jedoch die einfache Multiplexerstruktur. Dieses Problem sorgte für weitere Probleme, da manche Pins (z.B. die HDMI-Pins) nicht durch einen inout Bus nutzbar sind. Es müssen somit mehrere Ein/Ausgangsvektoren zu den Pins verbunden werden.

3. Jeder Pin kann nur mit einem Bit gleichzeitig benutzt werden. Die Eingabe- und Ausgabevektoren haben jedoch eine Größe von 8Bit. Gelöst werden kann dieses Problem entweder durch das nutzen mehrerer Pins gleichzeitig oder durch Serielle Nutzung der Pins. Die Parallele Nutzung ist oftmals nicht besonders sinnvoll, da hierdurch häufig verschiedene Geräte gleichzeitig benutzt werden. Bei der seriellen Nutzung muss die vom jeweiligen Protokoll genutzte Clock beachtet werden, was die Komplexität stark erhöht.

4. Die verschiedenen Protokolle der Geräte benötigen synchron Daten auf verschiedenen Kanälen oder bestimmte Bitfolgen oder bestimmte Verschlüsselungen. Mit diesen Protokollen lässt sich mit einem Primitiven Multiplexer in Kombination mit unserem Prozessorinterface nur sehr schwer oder nicht kommunizieren.

5. Ein- und Ausgaben müssen aktiv auf Veränderungen überprüft werden bzw. Verändert werden. Um z.B. eine LED nicht mit voller Helligkeit leuchten zu lassen muss diese an und ausgeschaltet werden. Dieses Problem kann durch Zwischenspeichern der Ein- und Ausgaben behoben werden. Ein weiterer Vorteil der Speicher ist, dass asynchrone Ein- und Ausgaben korrekt empfangen und gesendet werden können.

Aufgrund dieser Probleme wurde das Portsytem verändert: Jeder Port hat eine Funktion angepasst an das jeweils verwendete Protokoll. Hierdurch können auch komplexere Protokolle genutzt werden. Die Ordnung der Portfunktionen ist basierend auf den Pins, so folgt auf den Teil, welcher über EMIO erreicht wird der MIO Teil, bevor die durch den FPGA erreichbaren Pins folgen. Durch Speichern der Werte von Pins und Ports lassen sich zudem Werte auch passiv und asynchron ein- und ausgeben. Im aktuellen Entwicklungsstand ungenutzte Pins lassen sich als Speicher nutzen.

4.9 Top Level

Hauptverantwortlich: Tobias Weinschenk, Simon Naß.
Unterstützt von: Alexander Bunz, Jochen Benzenhöfer.

Zunächst wurden alle Komponenten zusammengesetzt und über Signale und Mux-Blöcke verbunden. Währenddessen wurden einige fälschlicherweise auf std_logik gesetzte Vektoren auf std_ulogik korrigiert.
Nachdem alle Komponenten zusammengesetzt und alle zuvor aufgetretenen Fehler in Vivado behoben wurden, sind bei den ersten Tests mit kleineren Programmen einige Fehler in der Top Level Datei und einzelnen Blöcken aufgefallen. Diese waren aber nur sehr kleine Fehler, wie etwa vertauschte if-else Verzweigungen oder falsch gemappte Block Verbindungen. Dementsprechend waren diese schnell behoben.
Als für die Implementierung verschiedene Assemblerprogramme geschrieben wurden, ist ein seltsames

Verhalten der Implementierung bei einigen FileName.data Dateien aufgetreten. Bei eingeschalteter Optimierung entfernt die Implementierung einige Blocke wie z.B. den RAM, wenn dieser nicht von den im FileName.data verwendeten Instruktionen benötigt wird. Es ist bekannt woher das Problem wahrscheinlich stammt. Gelöst wurde es jedoch, indem FileName.data wenn nötig alle Instruktionen enthält um jeden Block zu verwenden.
Als für den Test auf dem Board ein passendes Programm, welches alle Befehle, sowie LEDs und Schalter/Knöpfe verwendet, geschrieben wurde, ist noch ein kleiner Fehler im Input/Output aufgefallen. Als dieser auch behoben wurde, funktionierte das Programm auf dem Board ohne Probleme.
Abschliesend noch eine ausführlichere Variante des Top-Level Blockdiagramms um genauer die Zuordnung von VHDL-Code zu den einzelnen Blöcken herzustellen (zugehörige VHDL Datei in Klammern bei Blocknamen angegeben). Der IO-Block ist weiterhin abstrakter dargestellt. Zudem sind hier die Entscheidungen was letztendlich resetet werden kann und was ohne die Clock funktioniert gut abgebildet.

4.10 Assembler

Hauptverantwortlich: Alexander Bunz.

Anfangs sollte ein Python-Programm entstehen, welches eine Text Datei als Eingabe bekommt (in welcher Assembler für unseren Mikroprozessor steht), diese Textdatei liest und daraus Maschinencode erzeugt. Aus dieser anfänglichen Idee wurde dann schnell ein Programm, in welchem man direkt Assembler schreiben konnte, Textdateien speichern und laden, sowie den Maschinencode ausgeben und in die Zwischenablage kopieren lassen konnte. Dabei werden die obigen Befehle unterstützt. Komma, Semikolon, Klammern, sowie Groß-/Kleinschreibung und leere Zeilen werden dabei ignoriert, sind also Optional. Weiterhin war geplant, dass das Programm den Assembler Code direkt ausführt, um die Korrektheit zu prüfen. Dieser Plan wurde letztendlich wegen zu geringem Aufwand-Nutzen-Verhältnis verworfen. Letztendlich wurde dann noch die Möglichkeit implementiert für Jump und Call Befehle ein Label zu benutzen, welches vor den entsprechenden Befehl, zu dem gesprungen werden soll, geschrieben wird. Dafür musste zuvor noch die Zeile angegeben werden, in welcher der Befehl im Instruction PROM letztendlich liegt, dies ist allerdings auch weiterhin möglich. Deswiteren wurde noch ein Problem gefunden, bei dem Tabs als Befehle angesehen wurden und nicht wie Kommentare oder leere Zeilen ignoriert wurden. Das hat zu Problemen mit den Sprüngen zu Labeln geführt. Der Fehler war jedoch simpel zu beheben.

5 Beispiel Assembler

5.1 Fibonacci

Geschrieben von Alexander Bunz.

```
//Assembler for fibonacci sequence.
//Starting from (0, 1) and running for 10 steps.
LOAD x0, 0
LOAD x1, 1
LOAD x2, 10 //amount of steps
LOAD x3, 0 //counter

//Check counter
COMPARE x3, x2
JUMPC 7 //Jump to Fib-loop
JUMP 14 //Jump to End

//Fib-loop:
LOAD x4, 0
ADD x4, x0
ADD x4, x1
LOAD x0, x1
LOAD x1, x4
ADD x3, 1
JUMP 4

//End:
LOAD x0, x0
```

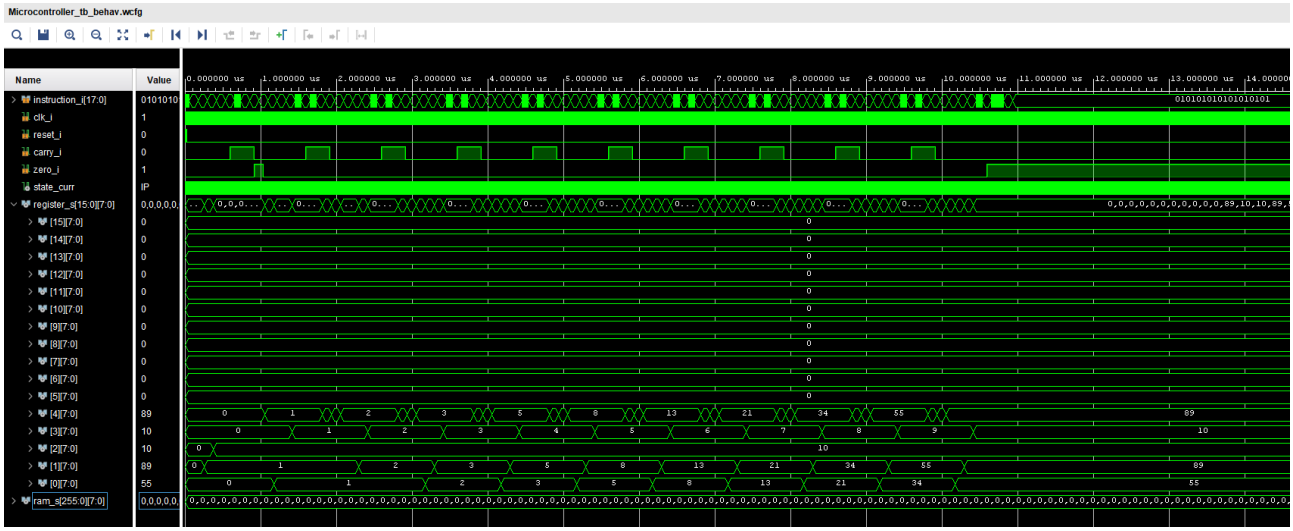


Abbildung 10: Fibonacci

5.2 Potenzen

Geschrieben von Alexander Bunz.

//Assembler to calculate x^y
LOAD x0, 3 //x value
LOAD x1, 5 //y value

COMPARE x0, 2
JUMPZ 15 //Jump to shift block if x = 2.
LOAD x4, x0

//multiply:
COMPARE x1, 1
JUMPZ 20 //Jump to end.
SUB x1, 1
LOAD x2, x0
LOAD x3, x4
//add:
COMPARE x3, 1
JUMPZ 5 //Repeat multiply block.
ADD x0, x2
SUB x3, 1
JUMP 10 //Repeat add block.

//Shift:
COMPARE x1, 1
JUMPZ 20 //Jump to end.
SL0 x0
SUB x1, 1
JUMP 15 //Repeat Shift block.

//End:
LOAD x5, x0 //Save result in x5.

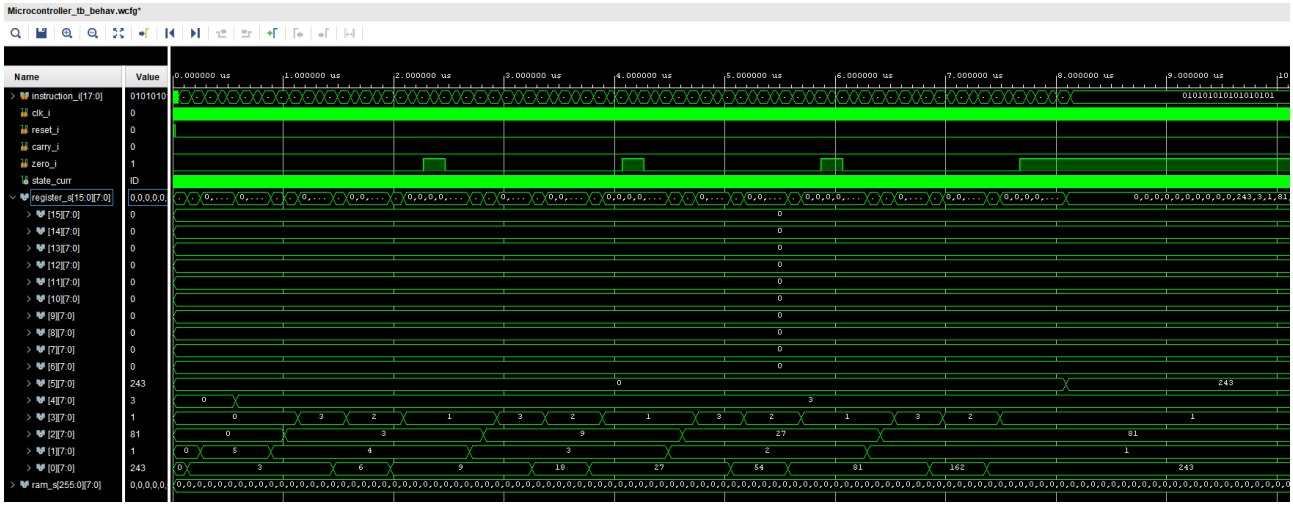


Abbildung 11: Potenzen

5.3 Primfaktorzerlegung

Geschrieben von Jannik Graef.

```
//Assembler to do a prime factorisation
//132=11*12=2*2*3*11
LOAD x0, 132 //number to be factorised

COMPARE x0, 2
JUMPC end //terminate if x0 less then 2
//as x0 has only itself as factor

//find all primes up to 255 and save them in RAM
LOAD x4, 2
//x6 is x4 / 2. so that only half as many
//numbers need to be tested
LOAD x6, 1
LOAD x5, 0 //RAM address to write
//new primes to

Label1
LOAD x2, 1
Label2
COMPARE x6, x2
//all numbers up to x4 / 2 have been checked
//and no divider has been found, x4 is prime
JUMPZ prime
ADD x2, 1
LOAD x1, x4
CALL divider //divide x4 and x2
COMPARE x1, 0 // x2 divides x4,
//x4 is not a prime
JUMPZ noPrime
JUMP Label2

prime
STORE x4, x5
ADD x5, 1

noPrime
ADD x4, 1
LOAD x6, x4
SR0 x6 // x4 / 2 rounded down
JUMPNZ Label1
//Prime finding complete

//begin factorisation
LOAD x4, 0
LOAD x5, 255

Label3
FETCH x2, x4 //save prime in x2
ADD x4, 1
LOAD x1, x0
CALL divider // calculate x1 / x2
COMPARE x1, 0 //if x1 mod x2 = 0 then x2 is a factor of x4
JUMPZ addFactor
JUMP Label3

addFactor
STORE x2, x5
SUB x5, 1
LOAD x4, 0
LOAD x0, x3
COMPARE x0, 1
JUMPZ end
JUMP Label3

divider // calculates x1 / x2 saves divident in
//x3 and rest in x1
LOAD x3, 0 //rest
LoopStart
SUB x1, x2
JUMPC LoopEnd
ADD x3, 1
JUMP LoopStart
LoopEnd
ADD x1, x2
RETURN

end
```

Abbildung 12: Faktoren

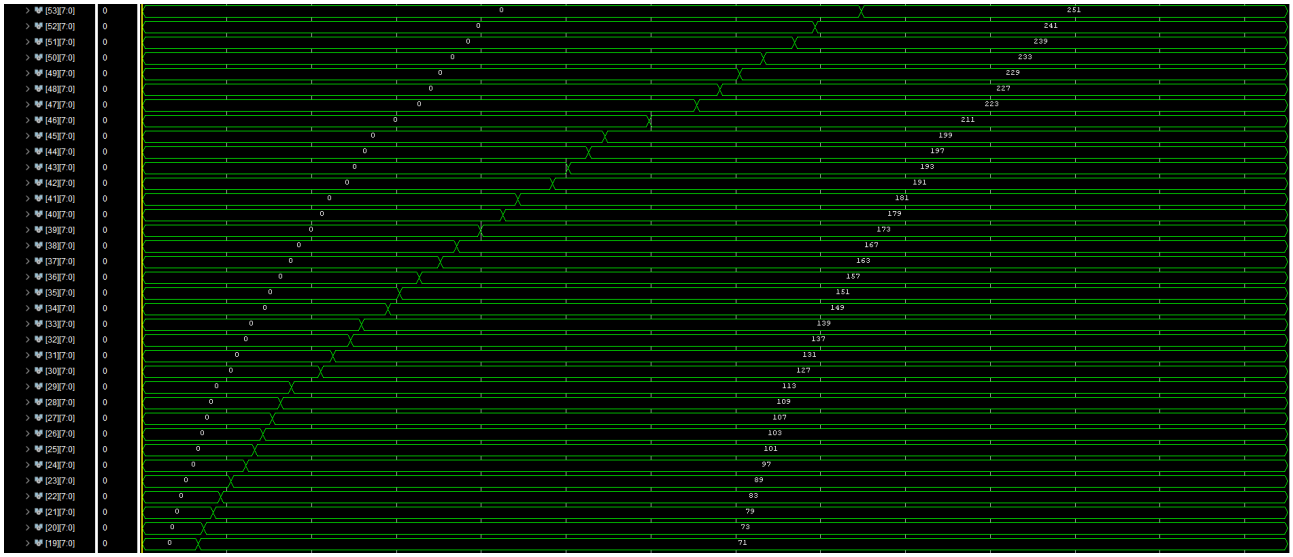


Abbildung 13: Primzahlen 1

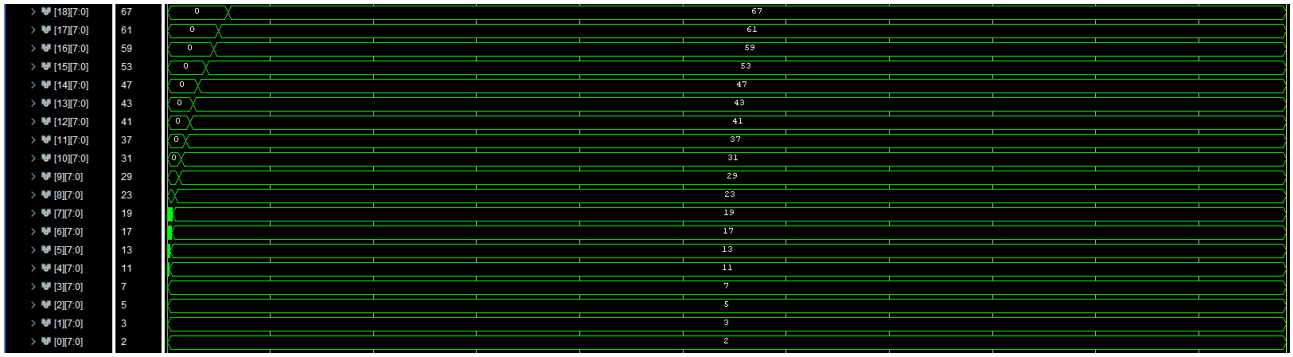
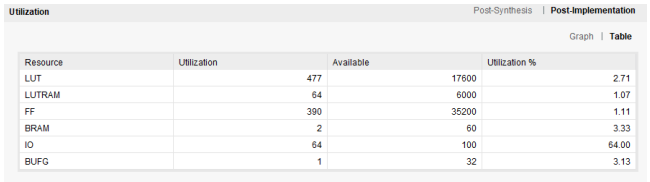


Abbildung 14: Primzahlen 2

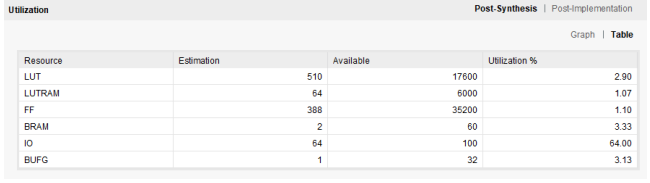
Zahlreiche weitere Assembler Programme lassen sich auf GitHub unter `https://github.com/graefjk/RO2/tree/main/Assembler` finden.

6 Interessante Zahlen



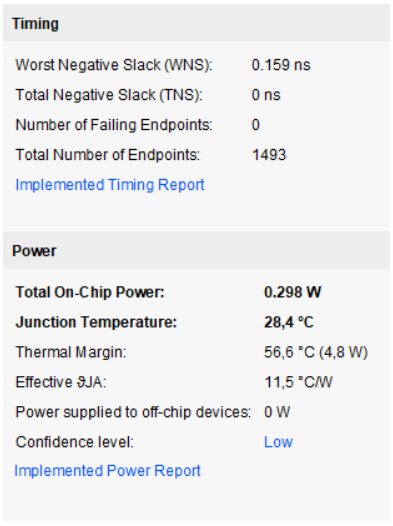
Resource	Utilization	Available	Utilization %
LUT	477	17600	2.71
LUTRAM	64	6000	1.07
FF	390	35200	1.11
BRAM	2	60	3.33
IO	64	100	64.00
BUFG	1	32	3.13

Abbildung 15: Vivado Implementation



Resource	Estimation	Available	Utilization %
LUT	510	17600	2.90
LUTRAM	64	6000	1.07
FF	388	35200	1.10
BRAM	2	60	3.33
IO	64	100	64.00
BUFG	1	32	3.13

Abbildung 16: Vivado Synthesis



Worst Negative Slack (WNS):	0.159 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	1493

Implemented Timing Report

Power

Total On-Chip Power:	**0.298 W**
Junction Temperature:	**28,4 °C**
Thermal Margin:	56,6 °C (4,8 W)
Effective ϑJA:	11,5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

Abbildung 17: Vivado Timing and Power

Name	Wert	Bemerkung
IP größe	4096x18 Bit	
RAM größe	256x8 Bit	
clk fequenz	125 MHz	8 ns clk cycle
clk Zyklen pro Befehl	6	4 bei Jumps/Calls/Returns

Tabelle 5: Interessante Zahlen

7 Aufgabenaufteilung

Nr.	Arbeitspaket	JG	TW	JB	AB	OAK	SN	JU
10	Instruktionen	*			*			
20	Blockdiagram			*			*	
30	Register		*					
40	RAM	*						
50	PC							*
51	Instruction Memory			*				
52	Stack		*					
60	I/O			*				
71	ALU Aritmetik					*		
72	ALU Logik						*	
73	ALU Shift				*			
74	ALU Verifikation 16 einfache Testfälle						*	
80	Decoder					*		
90	Assembler				*			
100	Test Register			*				
110	Test RAM							*
120	Test PC						*	
121	Test Instruction Memory						*	
122	Test Stack						*	
130	Test IO on board				*			
140	Test ALU	*						
150	Test Decoder		*					
160	TOP Level conections						*	
161	TOP Level package		*					
162	TOP Level board				*			
162	Präsentation				*			*

Tabelle 6: Aufgabenaufteilung

JG: Jannik Graef, **TW**: Tobias Weinschenk, **JB**: Jochen Benzenhoefer,
AB: Alexander Bunz, **OAK**: Omar Al Kadri,
SN: Simon Naß, **JU**: Jonas Unterweger.

Literaturverzeichnis

[1] Xilinx Vivado Synthesis Guide, https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_2/ug901-vivado-synthesis.pdf Zuletzt besucht am 18 Februar 2022.

[2] Xilinx Vivado Memory Guide, https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf Zuletzt besucht am 18 Februar 2022.

[3] Xilinx Picoblaze User Guide, https://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf Zuletzt besucht am 18 Februar 2022.

[4] Zybo Z7 Reference Manual, https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual Zuletzt besucht am 18 Februar 2022.

[5] Zybo Z7 Constraints, https://github.com/Digilent/digilent-xdc/blob/master/Zybo-Z7-Master.xdc Zuletzt besucht am 18 Februar 2022.

[6] Implementation of a Stack in VHDL, https://vhdlguru.blogspot.com/2011/01/implementation-of-stack-in-vhdl.html Zuletzt besucht am 18 Februar 2022.

[7] Example LIFO Memory, https://riptutorial.com/vhdl/example/32049/lifo Zuletzt besucht am 18 Februar 2022.

Abbildungsverzeichnis

Tabellenverzeichnis