

Architekturbeschreibung: Mikrokontroller für FPGA-Board

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

26. November 2021

1 Blockdiagram

Blockdiagram.pdf

2 Blockbeschreibungen

2.1 ALU

ALU_beschreibung.pdf


Die ALU führt Operationen auf zwei Eingabedaten aus. 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, gesetzt.

2.2 Instruction Decoder

ID_beschreibung.pdf

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, sowie der `or_PC_o` Ausgang, leiten entsprechend der eingegangenen Instruktion Steuerbits an verschiedene Multiplexer weiter. Der `sALU_select_o` Ausgang gibt den Opcode der Instruction an die ALU weiter. Der `sRam_write_or_read_o` Ausgang signalisiert dem Speicher, ob ein Wert gelesen oder geschrieben werden soll. Die `sRegister_write_enable_o`, `sStack_enable_o`, `sRAM_write_or_read_o` und `sRAM_enable_o` Ausgänge, steuern den zugriff auf Register, Stack und RAM.

2.3 Program Counter



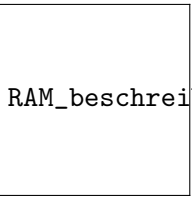
ProgramCounter_beschreibung (2).pdf

(2).pdf

Der Program Counter zeigt mit den Outputbits `pc_o` auf die Position im Instruktionsspeicher, an welcher der aktuelle Befehl steht, der in den Instruktionen 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 Setzen des `reset_i` Bits auf 1 kann der Program Counter asynchron auf den ersten Befehl zurückgesetzt werden.


2.4 scratchpad RAM



RAM_beschreibung.pdf

Ein zusätzlicher Speicher um Werte zu speichern. Das `write_or_read` bit bestimmt ob gelesen oder geschrieben wird. Es wird entweder von `address_i` auf `read_data` gelesen oder es wird `write_data` auf `address_i` geschrieben. `address_i` kommt entweder aus dem Register oder vom Instruktionen Decoder. `write_data` und `read_data` sind beide mit dem Register verbunden. Es wird nur in den RAM geschrieben wenn `enable` auf 1 gesetzt ist.

2.5 Register



Register_beschreibung.pdf

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` und `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.

2.6 Instruction Memory

IP_beschreibung.pdf

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`.

2.7 CALL/RETURN Stack

Stack_beschreibung.pdf

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 Setzen von `reset_i` auf 1 lässt sich der Stack asynchron resettet und damit leeren.

2.8 Input/Output

IO_beschreibung.pdf

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.

3 Instruktionen

	Instruktion	Beschreibung	Funktion
	ADD sX, kk	Addiert zum Register sX das Literal kk hinzu.	$sX \leftarrow sX + kk$
	ADD sX, sY	Addiert zum Register sX den Inhalt aus Register sY hinzu.	$sX \leftarrow sX + sY$
	ADDCY sX, kk	Addiert zum Register sX das Literal kk mit Carry-Bit hinzu.	$sX \leftarrow sX + kk + CARRY$
	ADDCY sX, sY	Addiert zum Register sX den Inhalt aus Register sY mit Carry-Bit hinzu.	$sX \leftarrow sX + sY + CARRY$
	AND sX, kk	Bitweises UND von Register sX mit dem Literal kk.	$sX \leftarrow sX \text{ AND } kk$
	AND sX, sY	Bitweises UND von Register sX mit Register sY.	$sX \leftarrow sX \text{ AND } sY$
	CALL aaa	Bedingungsloser Aufruf der Unterfunktion an der Adresse aaa.	$TOS \leftarrow PC, PC \leftarrow aaa$
	CALLC aaa	Falls CARRY-Bit gesetzt, Aufruf der Unterfunktion an der Adresse aaa.	if CARRY = 1 $TOS \leftarrow PC, PC \leftarrow aaa$
	CALLNC aaa	Falls CARRY-Bit nicht gesetzt, Aufruf der Unterfunktion an der Adresse aaa.	if CARRY = 0 $TOS \leftarrow PC, PC \leftarrow aaa$
	CALLNZ aaa	Falls ZERO-Bit nicht gesetzt, Aufruf der Unterfunktion an der Adresse aaa.	if ZERO = 0 $TOS \leftarrow PC, PC \leftarrow aaa$
	CALLZ aaa	Falls ZERO-Bit gesetzt, Aufruf der Unterfunktion an der Adresse aaa.	if ZERO = 1 $TOS \leftarrow PC, PC \leftarrow aaa$
	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 $\leftarrow 1$, if $sX < kk$ CARRY $\leftarrow 1$
	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 $\leftarrow 1$, if $sX < sY$ CARRY $\leftarrow 1$

	Instruktion	Beschreibung	Funktion
FETCH sX, (sY)	Lese scratchpad RAM von der in Register sY gespeicherten Adresse in Register sX	$sX \leftarrow \text{RAM}[(sY)]$	
FETCH sX, ss	Lese scratchpad RAM von Adresse ss in Register sX	$sX \leftarrow \text{RAM}[ss]$	
INPUT sX, (sY)	Lese Wert des Input-Port, welcher vom Register sY spezifiziert wird, in das Register sX.	$\text{PORT_ID} \leftarrow sY, sX \leftarrow \text{IN_PORT}$	
INPUT sX, pp	Lese Wert des Input-Port, welcher von pp spezifiziert wird, in das Register sX.	$\text{PORT_ID} \leftarrow pp, sX \leftarrow \text{IN_PORT}$	
JUMP aaa	Bedingungsloser Sprung nach aaa.	$pc \leftarrow aaa$	
JUMPC aaa	Falls das CARRY-Bit gesetzt ist, springe zu aaa.	if CARRY=1 $pc \leftarrow aaa$	
JUMPNC aaa	Falls das CARRY-Bit nicht gesetzt ist, springe zu aaa.	if CARRY=0 $pc \leftarrow aaa$	
JUMPNZ aaa	Falls das ZERO-Bit nicht gesetzt ist, springe zu aaa.	if ZERO=0 $pc \leftarrow aaa$	
JUMPZ aaa	Falls das ZERO-Bit gesetzt ist, springe zu aaa.	if ZERO=1 $pc \leftarrow aaa$	
LOAD sX, kk	Lade das Literal kk in das Register sX.	$sX \leftarrow kk$	
LOAD sX, sY	Lade den Inhalt des Registers sY in das Register sX.	$sX \leftarrow sY$	
OR sX, kk	Bitweise OR von Register sX mit literal kk.	$sX \leftarrow sX \text{ OR } kk$	
OR sX, sY	Bitweise OR von Register sX mit Register sY.	$sX \leftarrow sX \text{ OR } sY$	
OUTPUT sX, (sY)	Schreibe Register sX zum in sY gespeicherten output Port.	$\text{PORT_ID} \leftarrow sY, \text{OUT_PORT} \leftarrow sX$	
OUTPUT sX, pp	Schreibe Register sX zu output Port pp.	$\text{PORT_ID} \leftarrow pp, \text{OUT_PORT} \leftarrow sX$	
RETURN	Bedingungslose Rückkehr von der Unterfunktion.	$PC \leftarrow \text{TOS}+1$	
RETURNC	Falls Carry-Bit gesetzt, Rückkehr von der Unterfunktion.	If CARRY=1, $PC \leftarrow \text{TOS}+1$	
RETURNNC	Falls Carry-Bit nicht gesetzt, Rückkehr von der Unterfunktion.	If CARRY=0, $PC \leftarrow \text{TOS}+1$	

	Instruktion	Beschreibung	Funktion
RETURNZ	Falls Zero-Bit gesetzt, Rückkehr von der Unterfunktion.	If ZERO=1, PC \leftarrow TOS+1	
RETURNNZ	Falls Zero-Bit nicht gesetzt, Rückkehr von der Unterfunktion.	If ZERO=0, PC \leftarrow TOS+1	
RL sX	Rotiert Register sX einen Schritt nach links.	sX \leftarrow sX[6:0],sX[7], CARRY \leftarrow sX[7]	
RR sX	Rotiert Register sX einen Schritt nach rechts.	sX \leftarrow sX[0],sX[7:1],CARRY \leftarrow sX[0]	
SL0 sX	Schiebe Register sX links, mit 0 aufgefüllt.	sX \leftarrow sX[6:1],0, CARRY \leftarrow sX[7]	
SL1 sX	Schiebe Register sX links, mit 1 aufgefüllt.	sX \leftarrow sX[6:1],1, CARRY \leftarrow sX[7]	
SLA sX	Schiebe Register sX links durch alle Bits, inklusive Carry.	sX \leftarrow sX[6:1],CARRY, CARRY \leftarrow sX[7]	
SLX sX	Schiebe Register sX links. Bit sX[0] unbeeinträchtigt.	sX \leftarrow sX[6:1],sX[0], CARRY \leftarrow sX[7]	
SR0 sX	Schiebe Register sX rechts, mit 0 aufgefüllt.	sX \leftarrow 0,sX[7:1], CARRY \leftarrow sX[0]	
SR1 sX	Schiebe Register sX rechts, mit 1 aufgefüllt.	sX \leftarrow 1,sX[7:1], CARRY \leftarrow sX[0]	
SRA sX	Schiebe Register sX durch alle bits, inklusive Carry.	sX \leftarrow CARRY,sX[7:1], CARRY \leftarrow sX[0]	
SRX sX	Arithmetisches rechts-schieben von Register sX mit Vorzeichenerweiterung.	sX \leftarrow s[7],sX[7:1], CARRY \leftarrow sX[0]	
STORE sX, (sY)	Schreibe Register sX in scratchpad RAM an der in Register sY gespeicherten Adresse.	RAM[(sY)] \leftarrow sX	
STORE sX, ss	Schreibe Register sX an Adresse ss in scratchpad RAM.	RAM[ss] \leftarrow sX	
SUB sX, kk	Subtrahiere literal kk von Register sX.	sX \leftarrow sX-kk	
SUB sX, sY	Subtrahiere Register sY von Register sX.	sX \leftarrow sX-sY	
SUBCY sX, kk	Subtrahiere literal kk von Register sX mit Carry.	sX \leftarrow sX-kk-CARRY	
SUBCY sX, sY	Subtrahiere Register sY von Register sX mit Carry.	sX \leftarrow sX-sY-CARRY	

	Instruktion	Beschreibung	Funktion
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 \leftarrow 1, CARRY \leftarrow 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 \leftarrow 1, CARRY \leftarrow odd parity of (sX AND sY)	
XOR sX, kk	Bitweise XOR von Register sX mit literal kk.	sX \leftarrow sX XOR kk	
XOR sX, sY	Bitweise XOR von Register sX mit Register sY.	sX \leftarrow sX XOR sY	

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): CARRY = 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): CARRY = 1, else: CARRY = 0	000010
AND sX, kk	if (sX = 0): ZERO = 1, else: ZERO = 0	0	001001
AND sX, sY	if (sX = 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 = 0): ZERO = 1, else: ZERO = 0	0	001011
OR sX, sY	if (sX = 0): ZERO = 1, else: ZERO = 0	0	001010
OUTPUT sX, (sY)	-	-	010100
OUTPUT sX, pp	-	-	010101
RETURN	-	-	111010
RETURNC	-	-	111011
RETURNNC	-	-	111100
RETURNZ	-	-	111101
RETURNNZ	-	-	111110
RL sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110000
RR sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	110001

Instruktion	ZERO	CARRY	opcode
SL0 sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110010
SL1 sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110011
SLA sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110100
SLX sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[7]	110101
SR0 sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	110111
SR1 sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	110111
SRA sX	if (sX = 0): ZERO = 1, else: ZERO = 0	CARRY = sX[0]	111000
SRX sX	if (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): CAR- RY = 1, else: CARRY = 0	000110
TEST sX, kk	if (AND_TEST = 0): ZERO = 1, else: ZERO = 0	if (XOR_TEST = 1): CARRY = 1, else: CARRY = 0	011001
TEST sX, sY	if (AND_TEST = 0): ZERO = 1, else: ZERO = 0	if (XOR_TEST = 1): CARRY = 1, else: CARRY = 0	011000
XOR sX, kk	if (sX = 0): ZERO = 1, else: ZERO = 0	0	001101
XOR sX, sY	if (sX = 0): ZERO = 1, else: ZERO = 0	0	001100

4 Aufgabenaufteilung

	Nr.	Arbeitspaket	JG	TW	JB	AB	OAK	SN	JU
10	Instruktionen	*				*			
20	Blockdiagramm				*			*	
30	Register		*						
40	RAM	*							
50	PC								*
51	Instruction Memory				*				
60	I/O				*				
71	ALU Aritmetik						*		
72	ALU Logik							*	
73	ALU Shift					*			
74	ALU Verifikation 10 einfache Testfälle							*	
80	Decoder						*		
90	Assembler					*			
100	Test Register				*				
110	Test RAM								*
120	Test PC							*	
130	Test I/O					*			
140	Test ALU	*							
150	Test Decoder		*						