

9 Aufbau und Arbeitsweise eines Rechners

Die Arbeitsweise eines Rechners als Black-Box betrachtet, also ohne in den Rechner hinein zu schauen, lässt sich durch Abbildung 34 beschreiben.

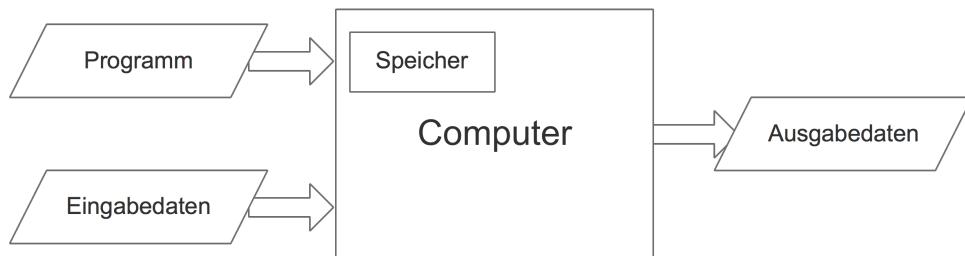


Abbildung 34: Rechner als Black Box

Der Rechner (Computer) nimmt Eingabedaten entgegen, verarbeitet diese und gibt die Ergebnisse als Ausgabedaten aus.

Das, was der Rechner machen soll, wird ihm in Form eines Programms, das in einem Speicher gehalten wird, „beigebracht“.

Das Programm besteht aus hintereinander auszuführenden Befehlen. Die einzelnen hintereinander auszuführenden Befehle sind auch hintereinander im Speicher abgelegt.

Im Gegensatz zu einer einfachen Kaufhauskasse oder einem Taschenrechner, die immer nur eine Operation auf einem oder zwei vorher eingegebene Werten ausführen können (danach muss man neu eingeben), können hier mehrere Operationen hintereinander ausgeführt werden, die vorher so durch das Programm geplant wurden.

9.1 Das Von-Neumann-Prinzip

Betrachten man den Rechner als White-Box, schaut man also hinein, so sieht man die in Abbildung 35 dargestellten Funktionseinheiten.

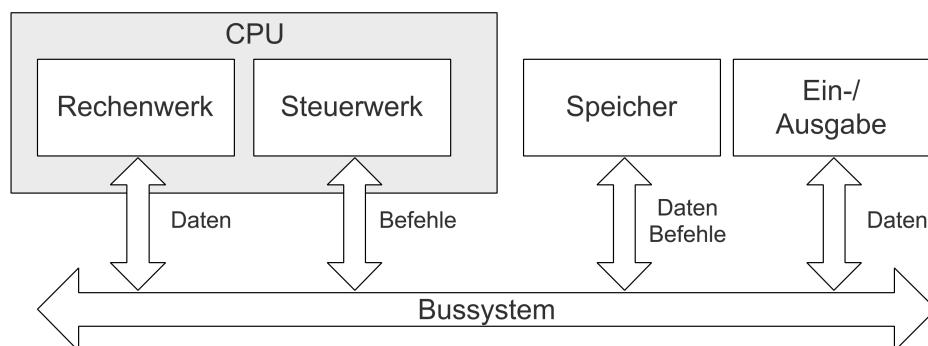


Abbildung 35: Funktionseinheiten in einem Rechner

Diese einfache aber geniale Prinzip eines Rechners wird nach seinem „Erfinder“ **Von-Neumann-Architektur** genannt.

- Der Speicher enthält sowohl das Programm als auch die zur Verarbeitung eingelesenen Eingabedaten sowie die berechneten Ausgabedaten.
- Das Rechenwerk führt die Rechnungen durch.
- Das Steuerwerk sorgt für den koordinierten Ablauf aller Aktivitäten des Rechners.
- Die Funktionseinheiten sind zwecks Datentransfer über Busse (durchaus mehr als einer) miteinander verbunden.

- Das Ein-/Ausgabewerk nimmt die Eingabedaten entgegen (z.B. über Tastatur) und gibt die Ausgabedaten aus (z.B. über Grafikkarte und Monitor)
- Das Rechenwerk und das Steuerwerk sind typischerweise in einem Chip untergebracht. Dieser Chip wird **CPU** (Central Processing Unit) oder auch **Prozessor** genannt.

Heute existierende Rechner sind sicherlich komplexer als die Von-Neumann-Architektur, dennoch können alle Rechner als eine Erweiterung dieses Prinzips betrachtet werden.

Bei einer (historisch gesehen) ersten Erweiterung dieses Prinzips hat der Rechner nicht einen, sondern zwei Speicher. Ein Speicher ist für die Daten, der andere für den Programm-Code (das Programm) reserviert. Beide Speicher sind über getrennte Busse gleichzeitig ansprechbar, wodurch eine Geschwindigkeitssteigerung erzielt wird. Dieses erweiterte Prinzip wird **Harvard-Architektur** genannt.

Im folgenden werden zunächst die einzelnen Funktionseinheiten näher betrachtet, darauf aufbauend wird dann das „Zusammenspiel“ der Funktionseinheiten anhand einer Simulation eines einfachen Rechners erläutert.

9.1.1 Rechenwerk

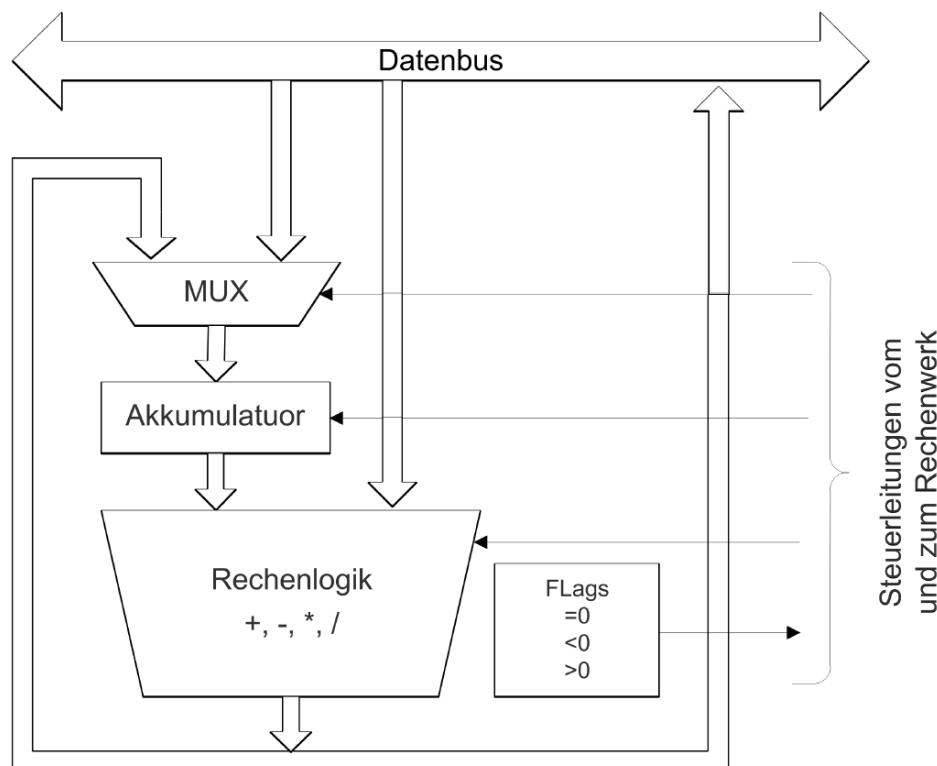


Abbildung 36: Rechenwerk

Rechenlogik: Schaltnetz, das aus einer logischen Verknüpfung der beiden Operanden das Ergebnis „berechnet“. Über die Steuersignale wird die Rechenlogik¹³ so eingestellt, dass eine von mehreren möglichen Verknüpfungen (oder Rechnungen) durchgeführt werden soll. Typische einfache Verknüpfungen sind: Volladdierer, Komplement, UND/ODER/NICHT (s. auch Nandgame Level ALU)

Akkumulator: Register, das einen der beiden Operanden zwischenspeichert und bei Bedarf das Ergebnis der Berechnung übernimmt.

¹³auch ALU: Arithmetic Logical Unit genannt

Multiplexer: Über ein Steuersignal (Der Selektor des Multiplexers, Kap. 4.11.1) kann bestimmt werden, ob der Akkumulator das Ergebnis der Rechenlogik speichert, oder einen Operanden vom Datenbus aufnimmt.

Flags: Die Flags geben Auskunft über das Ergebnis der Rechenlogik. Man kann sich die Bildung der Flags durchaus auch als einen Teil der Rechenlogik vorstellen. Die Flags werden dem Steuerwerk zur Auswertung zur Verfügung gestellt. Z.B. für bedingte Anweisungen oder Verzweigungen, z.B. bei der Auswertung von if ($a > b$) then {...}.

9.1.2 Steuerwerk

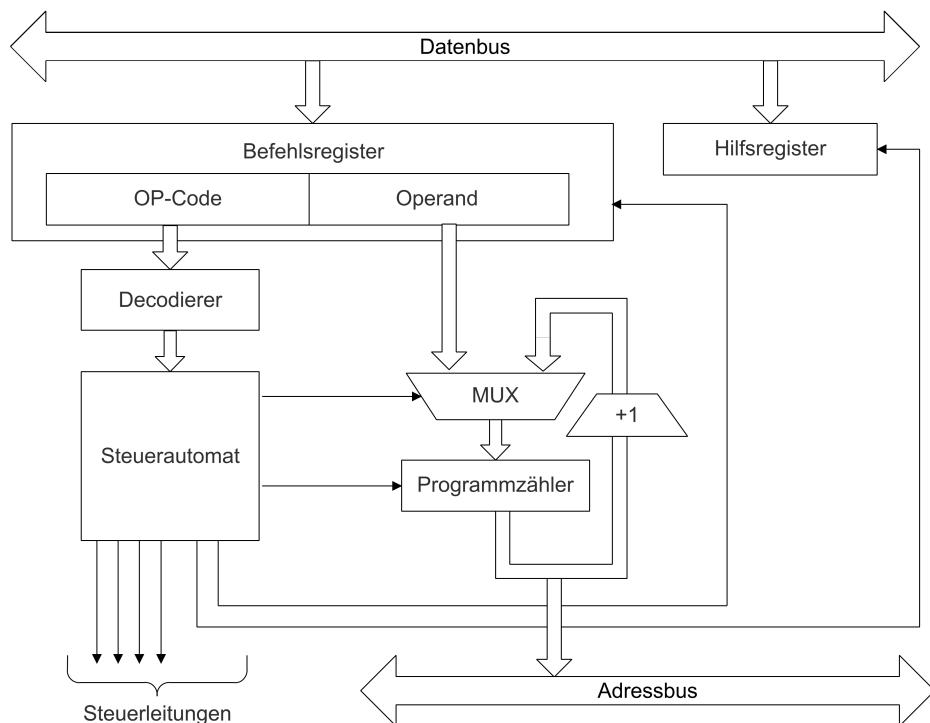


Abbildung 37: Steuerwerk

Das in Abbildung 37 dargestellte Steuerwerk besteht aus:

Befehlsregister: Register, das einen aus dem Speicher ausgelesenen Befehl zwischenspeichert. Ein Befehl besteht immer aus einem **Operator (OP-Code)**, der sagt „was zu tun ist“ und einem **Operand**, der besagt „womit etwas zu tun ist“.

Programmzähler (engl. Program Counter PC): Register, das die Speicheradresse des nächsten auszuführenden Befehls enthält. Belegen alle möglichen Befehle nur genau eine Speicherzelle so kann der Programmzähler nach jedem Befehl genau um eins erhöht werden.¹⁴

Hilfsregister: Register, das für die Berechnung von Speicheradressen herangezogen wird.

Decodierer: Schaltnetz, das aus dem Operationsteil ermittelt welcher Befehl ausgeführt werden soll.

¹⁴Ausgenomme sind Sprungbefehle, da wird der PC mit einem neuen Wert geladen

Steuerautomat: Automat, der abhängig vom durchzuführenden Befehl (abhängig von den vom Decoder erzeugten Signalen) eine Abfolge von Steuersignalen erzeugt. Über die Steuersignale veranlasst der Steuerautomat also die Abarbeitung eines Befehls.

Man kann sagen, dass der Automat zur Abarbeitung eines Befehls eine Sequenz von mehreren Unterbefehlen (Mikrobefehlen) abarbeitet.

9.1.3 Speicher

Im Rahmen des „Rechnerbaus“ wird der Begriff Speicher oftmals synonym zum Begriff „wahlfreier Schreib-Lese-Speicher“ oder auch „Random Access Memory (RAM)“ verwendet.

Der schematische Aufbau eines RAMs ist in Abbildung 38 dargestellt. Das RAM hat 16 Speicherzellen zu je 1 Bit.

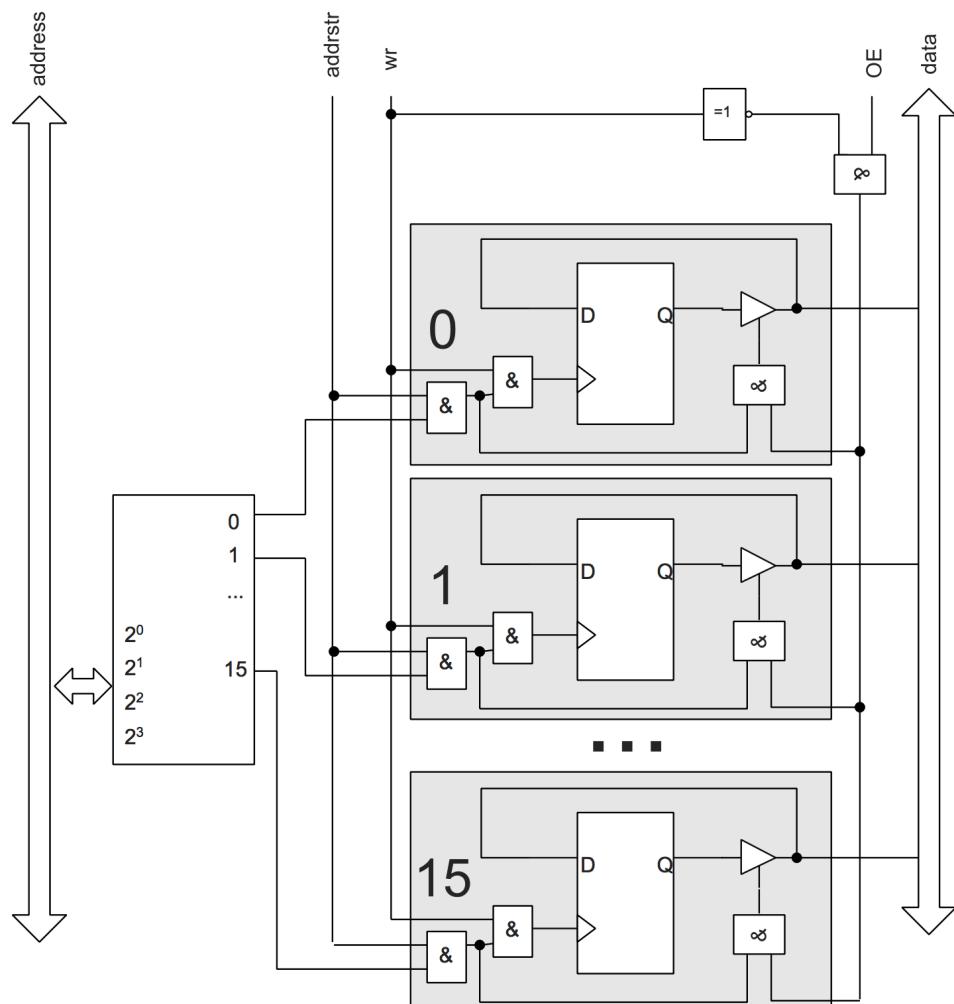


Abbildung 38: Ein 16 Bit Speicher

Adressierung: Die Adressierung der Speicherzellen erfolgt durch Anlegen der Adresse (in Abbildung 38 4 Bit).

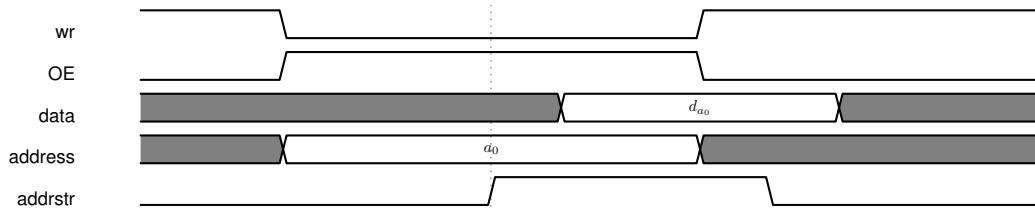
Die Adressbits werden mit der positiven Flanke des Steuersignals Address-Strobe (addrstr) ausgewertet.

Über den 1-aus-16 Decoder (s. Kapitel 4.11.3) wird die adressierte Speicherzelle selektiert und angesprochen.

Ist das Steuersignal Write (wr) aktiv ($wr=1$) so wird ins RAM geschrieben. Ist Write inaktiv ($wr=0$) so wird vom RAM gelesen.

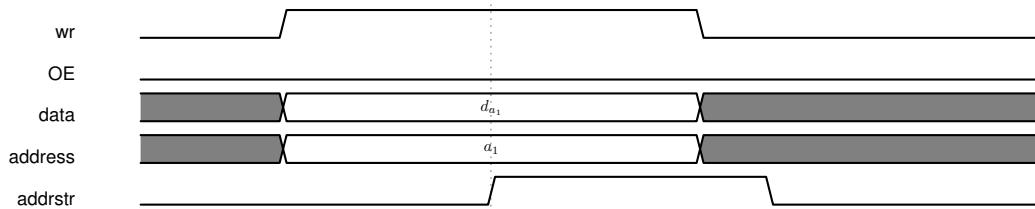
Bei vielen Speicherbausteinen können die Datenausgänge auch von außerhalb des Speichers durch ein Output-Enable-Signal (OE=0) in den Tri-State gebracht werden.

Lesen vom RAM Beim Lesen wird der Inhalt der selektierten (gewählten = adressierten) Speicherzelle an den Datenleitungen ausgegeben. Hierbei muss der folgende Zeitablauf (Timing) eingehalten werden:



1. Eine gewisse Zeit (z.B. 10 ns) vor der steigenden Flanke von **addrstr** müssen die Adresse (**address**) und die Steuersignale gültig sein (**wr=0, OE=1**).
2. Eine gewisse Zeit (z.B. 5 ns) nach der steigenden Flanke von **addrstr** werden die Daten an **data** ausgegeben
3. Die Adresse und die Steuersignal (**wr, OE**) müssen auch noch eine gewisse Zeit (z.B. 10ns) nach der steigenden Flanke **addrstr** anliegen.

Schreiben ins RAM Beim Schreiben werden die an den Datenleitungen anliegenden Daten in die selektierte Speicherzelle übernommen. Hierbei muss der folgende Zeitablauf (Timing) eingehalten werden:



1. Eine gewisse Zeit (z.B. 10 ns) vor der steigenden Flanke von **addrstr** müssen die Daten (**data**), die Adresse (**address**) und die Steuersignale gültig sein (**wr=1, OE=0**).
2. Mit der steigenden Flanke von **addrstr** werden die Daten in das adressierte Speicherregister übernommen.
3. Die Daten, die Adresse und die Steuersignal (**wr, OE**) müssen auch noch eine gewisse Zeit (z.B. 10 ns) nach der steigenden Flanke **addrstr** anliegen.

9.1.4 Ablauf einer Programmabarbeitung

Zur Abarbeitung der verschiedenen Assemblerbefehle (z.B. LDA, LDM, LDI, JA des später noch diskutierten SVNR) durchläuft der Steuerautomat die in Abbildung 39 gezeigten Zustände.

Der Ablauf kann in die drei Phasen Fetch, Decode und Execute aufgeteilt werden.

In der **Fetch**-Phase wird der Befehl bestehend aus OP-Code und Operand aus dem Speicher geholt. Dafür sind mehrere Schritte nacheinander notwendig, der Datenpfad wird mit Multiplexern so eingestellt, dass der Speicherausgang in das Befehlsregister führt und die Steuersignale müssen in zeitlichen Abständen erzeugt werden.

In der **Decode**-Phase wird der OP-Code-Teil (beim SVNR die ersten 8 Bit) des Befehls analysiert und der Automat verzweigt zu verschiedenen Abläufen je nach Befehl (0x18 = LDA, 0x11 = LDM, 0x12 = LDI, 0x48 = JA)

Die **Execute**-Phase wird für jeden der vier in Abbildung 39 gezeigten Befehlsausführungen mit unterschiedlich vielen Mikrobefehlen ausgeführt.

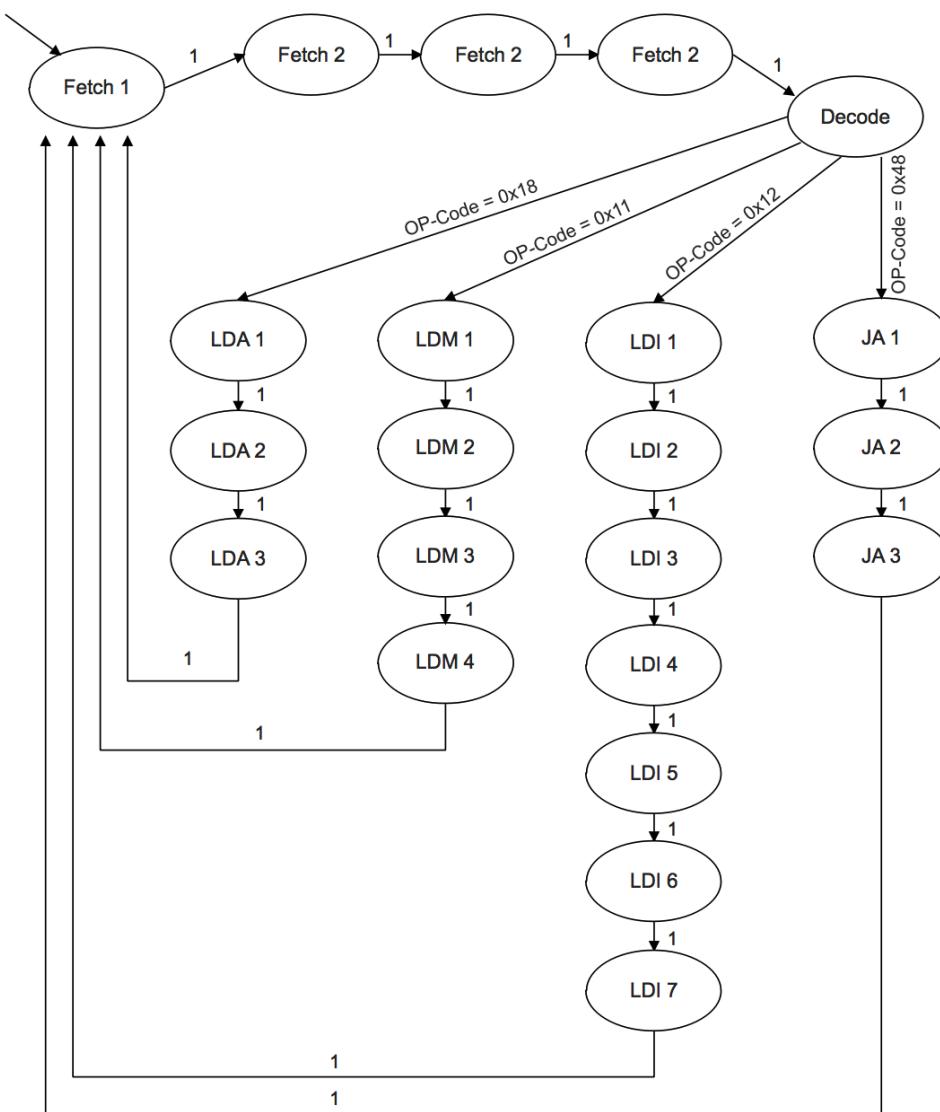


Abbildung 39: Fetch, Decode und Execute am Beispiel von 4 ausgewählten Assemblerbefehlen

9.2 Ein simpler/simulierbarer Von-Neumann-Rechner (SVNR)

Die bisher vorgestellten Sachverhalte sind ohne detaillierte Kenntnis des Zusammenspiels der einzelnen Komponenten nicht leicht zu verstehen.

Zum Verständnis wird deshalb eine animierte Simulation eines einfachen Von-Neumann-Rechners herangezogen.

Abbildung 40 zeigt den Aufbau des SVNR-Simulators

Der Aufbau kann unterteilt werden in Speicher, Rechenwerk und Steuerwerk

Die Simulation zeigt das Zusammenspiel der einzelnen Komponenten im Rahmen der Abarbeitung eines im Speicher befindlichen Programms.

Die Speicher- und Registerinhalte werden hexadezimal dargestellt. Sowohl die Speicher- als auch die Registerinhalte sind per Mausclick editierbar. D.h. der Rechner kann programmiert werden.

Das Rechenwerk (ALU) erzeugt die Flags für =0, <0 und >0, die neben der Rechenlogik dargestellt werden. Negative Zahlen werden im Zweierkomplement verarbeitet.

Die Datenwege sind untereinander verbunden und werden im Einzelschrittbetrieb hervorgehoben. Die Darstellung des Steuerwerks ist vereinfacht. Der Decodierer, der Steuerautomat und die Steuersignalleitungen sind nicht dargestellt. Man muss sich also den Steuerautomaten als „im Hintergrund arbeitend“ vorstellen.

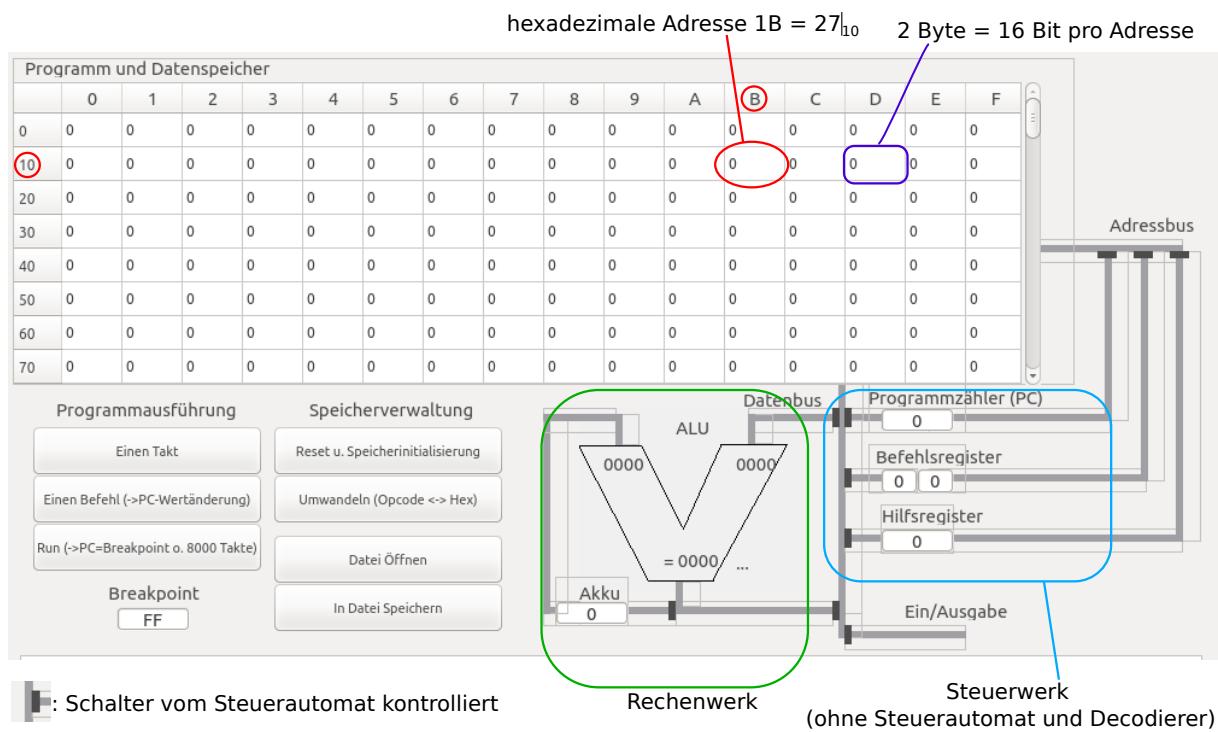


Abbildung 40: GUI des SVNR Simulators

Der Steuerautomat bestimmt unter anderem wie die Datenwege geschaltet sind. Er definiert somit zu jedem Zeitpunkt den Datenfluss.

Die über die Steuersignale geschalteten Datenwege sowie die adressierten Speicherzellen werden im Betrieb farbig hervorgehoben.

9.2.1 Befehlssatz des SVNR

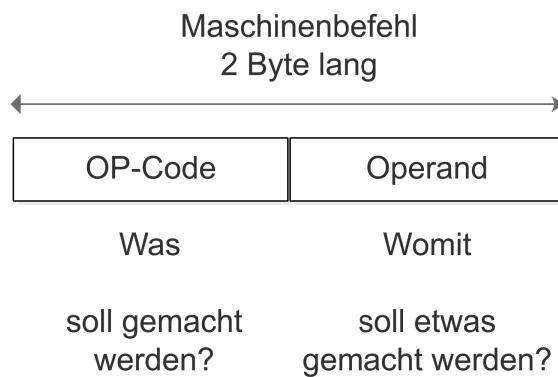


Abbildung 41: Aufteilung der Maschinenbefehle

Alle Maschinenbefehle des SVNR sind 2 Byte lang also hat der SVNR einen Maschinenbefehlssatz fester Länge (Das ist nicht bei jeder CPU so).

Das linke Byte (die höherwertigen 8 Bit) kennzeichnet die durchzuführende Operation. Dieser Teil des Befehls wird **Operationscode** (OP-Code) genannt.

Das rechte Byte kennzeichnet den Operanden (z.B. eine Adresse oder ein Wert). Der SVNR hat stets nur einen Operanden. Beim SVNR ist der zweite Operand immer der Akkumulator. Man sagt: „Der Rechner ist eine **Ein-Adress-Maschine** oder auch **Akkumulator-Maschine**.“ Es gibt auch Rechner, bei denen beide Operanden explizit angegeben werden können. Diese werden dann als **Zwei-Adress-Maschine** bezeichnet.

Damit die Befehle für den Menschen besser lesbar sind, legt man für die Visualisierung der OP-Codes

Hex	Mnemo	Mnemo Herleitung	Beschreibung
Datentransportbefehle:			
10xx	No Op.	no operation	keine Operation
11xx	LDM A	load accu from memory	Akku = (Inhalt von Adresse A)
12xx	LDI A	load accu from memory indirectly	Akku = (Inhalt von Adresse, die durch den Inhalt von Adresse A bestimmt wird)
18xx	LDA W	load accu absolutely	Akku = Wert W
21xx	STI A	store accu in memory indirectly	kopiere Akku an den Inhalt von der Adresse, die durch den Inhalt von Adresse A bestimmt wird
28xx	STM A	store accu in memory	kopiere Akku an den Inhalt von Adresse A
Arithmetische, logische und Schiebe- Befehle:			
30xx	ADD A	add memory to accu	Akku = Akku + (Inhalt von Adresse A)
31xx	SUB A	subtract memory from accu	Akku = Akku - (Inhalt von Adresse A)
34xx	AND A	logical AND accu with memory	Akku = Akku <i>logisch bitweise UND-verknüpft</i> mit (Inhalt von Adresse A)
35xx	OR A	logical OR accu with memory	Akku = Akku <i>logisch bitweise ODER-verknüpft</i> mit (Inhalt von Adresse A)
36xx	NOT	logical negation	Akku = Akku <i>logisch bitweise negiert</i>
37xx	XOR A	logical XOR accu with memory	Akku = Akku <i>logisch bitweise XOR-verknüpft</i> mit (Inhalt von Adresse A)
38xx	INC	increment accu	Akku = Akku + 1
39xx	DEC	decrement accu	Akku = Akku - 1
3Cxx	LEFT W	shift left accu	Akku W-mal bitweise nach links schieben, rechts mit Nullen auffüllen
3Dxx	RIGHT W	shift right accu	Akku W-mal bitweise nach rechts schieben, links mit Nullen auffüllen
Steuerungsbefehle:			
41xx	JM A	jump memory	Springe zu Adresse, die durch den Inhalt von Adresse A bestimmt wird
48xx	JA A	jump absolutely	Springe zu Adresse A
51xx	JZM A	jump zero memory	Springe zu Adresse, die durch den Inhalt von A bestimmt wird, wenn ALU-Flag = 0
52xx	JNM A	jump nonzero memory	Springe zu Adresse, die durch den Inhalt von A bestimmt wird, wenn ALU-Flag != 0
53xx	JLM A	jump less or equal memory	Springe zu Adresse, die durch den Inhalt von A bestimmt wird, wenn ALU-Flag <= 0
58xx	JZA A	jump zero absolutely	Springe zu Adresse A, wenn ALU-Flag = 0
59xx	JNA A	jump nonzero absolutely	Springe zu Adresse A, wenn ALU-Flag != 0
5Ax	JLA A	jump less or equal absolutely	Springe zu Adresse A, wenn ALU-Flag <= 0
Ein/Ausgabe Befehle:			
61xx	IN A		Inhalt von Adresse A = eingelesener Wert von einem externen Gerät (hier Zufallswerte)
71xx	OUT A		Inhalt von Adresse A an das externe Gerät ausgeben

Tabelle 7: Befehlssatz des SVNR

eine kurze aber aussagekräftige Zeichenfolge (**Mnemo**, Gedächtnisstütze) für jeden OP-Code fest. Die Menge aller möglichen Operationen, die ein Rechner ausführen kann, wird **Befehlssatz** genannt. Der Befehlssatz des SVNR ist in Tabelle 7 aufgeführt.

Opcodes ohne Operand sind INC, DEC, NOT und NoOP. Da ein Befehl aber *immer* aus Opcode und Operand besteht, muss ein Operand angegeben werden, der von der CPU jedoch nicht ausgewertet wird, z.B: INC 00 oder DEC 0f¹⁵.

Beispiele zu ausgewählten Befehlen:

LDA:	Adresse	Inhalt	Bedeutung
	0020	18ff	LDA 0xff

Nach Ausführung des Befehls ab Adresse 0x0020 enthält der Akku den Wert 0x00ff Man beachte, dass Werte größer als 0xff so nicht geladen werden können, da der Operand nur 1 Byte enthalten kann.

LDM:	Adresse	Inhalt	Bedeutung
	0020	114a	LDM 0x004a

	004a	10bb	An Adresse 0x4a steht 0x10bb (Daten)

Wird der Code ab Adresse 0x20 ausgeführt, wird der Inhalt der Adresse 0x4a in den Akku kopiert. Nach Ausführung des Befehls enthält der Akku den Wert 0x10bb.

Dies entspricht einer Variablen in C/Java, z.B.:

```
uint16_t var = 0x10bb; // var liegt an Adresse 0x004a
```

Der Speicher an Adresse 0x004a wird zum Speicherort für die Variable var.

¹⁵Achtung: Wird nur DEC im Textformat des SimVNR angegeben, interpretiert der Simulator das als Hexadezimalzahl: 0x0dec und nicht als das Mnemonic

Adresse	Inhalt	Bedeutung
0020	12ab	LDI 0xab
...	...	An Adresse 0x3a steht 0xfed0 (Daten)
003a	fed0	
003b	1111	
003c	0af3	
...	...	
00ab	003a	

LDI:
Wird der Code ab Adresse 0x20 ausgeführt, wird der Inhalt der Adresse 0x00ab ins Hilfsregister kopiert. Das ist 0x003a. Danach wird das Hilfsregister zum Adressieren genutzt und der Inhalt der Adresse 0x003a gelesen und in den Akku kopiert. Nach Ausführung des Befehls enthält der Akku den Wert 0xfed0.

Die entspricht einer lesenden Pointer-Operation in C

```
uint16_t var[3] = {0xfed0,    // var[0] liegt an Adresse 0x003a
                   0x1111,    // var[1] liegt an Adresse 0x003b
                   0x0af3};   // var[2] liegt an Adresse 0x003c
uint16_t *p = var;          // Einen Pointer anlegen,
                           // der auf den Anfang des Felds zeigt
                           // p wird an Adresse 0x00ab gespeichert
                           // und mit 0x003a initialisiert
uint16_t a = *p;           // entspricht LDI 0xab Befehl
                           // a enthält den ersten Wert des Felds
```

Der Speicher an Adresse 0x003a ist der Beginn des Felds für 3 Variablen. In Adresse 0x00ab wird der Pointer abgelegt, der auf den Feldanfang zeigt. Möchte man z.B. das 2. Element des Felds lesen, erhöht man den Pointer um 1 und ruft erneut LDI auf.

Adresse	Inhalt	Bedeutung
Akku	1234	Akku ist keine Adresse
0020	284b	
...	...	
004b		

STM:
Der Akku enthält 0x1234. Nach Ausführung des Befehls ab Adresse 0x0020 enthält die Speicherstelle an Adresse 0x004b den Wert des Akkus

Dies entspricht dem Schreiben auf eine Variablen in C/Java, z.B.:

```
uint16_t var;           // var wird an Adresse 0x004b angelegt
var = 0x1234;           // ein neuer Wert wird zugewiesen (STM)
```

Der Speicher an Adresse 0x004a wird zum Speicherort für die Variable var.

Adresse	Inhalt	Bedeutung
Akku	2345	Akku ist keine Adresse
0020	2133	
...	...	
0033	21ac	
...	...	
21ac	0000	
21ad	0000	
21ae	0000	

STI:
Der Akku enthält 0x2345. Wird der Code ab Adresse 0x20 ausgeführt, wird der Inhalt der Adresse 0x0033 ins Hilfsregister kopiert. Das ist 0x21ac. Danach wird das Hilfsregister zum Adressieren genutzt und der Akku an den Inhalt der Adresse 0x21ac geschrieben. Nach Ausführung des Befehls enthält die Speicherstelle 0x21ac den Wert 0x2345.

Die entspricht einer schreibenden Pointer-Operation in C

```
uint16_t array[3] = {0x0000, // array[0] liegt an Adresse 0x21ac
                     0x0000, // array[1] liegt an Adresse 0x21ad
                     0x0000}; // array[2] liegt an Adresse 0x21ae
uint16_t *p = array; // Einen Pointer anlegen,
                     // der auf den Anfang des Felds zeigt
                     // p wird an Adresse 0x0033 gespeichert
                     // und mit 0x21ac initialisiert
uint16_t *p = 0x2345 // entspricht STI 0x33 Befehl, wenn
                     // der akku 0x2345 enthält
```

Der Speicher an Adresse 0x21ac ist der Beginn des Felds für 3 Variablen. In Adresse 0x0033 wird der Pointer abgelegt, der auf den Feldanfang zeigt. Möchte man z.B. das 2. Element des Felds beschreiben, erhöht man den Pointer um 1 und ruft erneut STI auf.

	Adresse	Inhalt	Bedeutung
ADD:	Akku	000a	Akku ist keine Adresse
	0020	30b0	ADD 0xb0

	00b0	1000	An Adresse 0xb0 steht 0x1000 (Daten)

Nach Ausführung des Codes an Adresse 0x20 steht im Akku 0x100a ($0x1000 + 0x000a$).

Die entspricht folgenden Operationen in C/Java:

```
uint16_t sum = 0x1000; // sum wird an Adresse 0x00b0 angelegt
sum = sum + 0x000a; // Eine Addition liest den Wert, addiert und
                     // schreibt die Summe dann zurück
```

	Adresse	Inhalt	Bedeutung
AND:	Akku	000a	Akku ist keine Adresse
	0020	347f	AND 0x7f

	007f	fff8	An Adresse 0x7f steht 0xffff8 (Daten)

Bitweises UND von

0000 0000 0000 1010 (0x000a) und

1111 1111 1111 1000 (0xffff8) ergibt

0000 0000 0000 1000 = 0x0008. Nach Ausführung des Codes an Adresse 0x0020 steht im Akku 0x0008.

Die entspricht folgenden Operationen in C/Java:

```
uint16_t x = 0xffff8; // x wird an Adresse 0x007f angelegt
x = x & 0x000a; // &, |, ~, ^, und, oder, nicht, xor
```

Das Ergebnis der logische, bitweisen Verknüpfung wird an die Speicherstelle zurückgeschrieben. Die Symbole für die logischen Operatoren in C/Java sind in der Tabelle in Kapitel 3.3.4 aufgeführt.

9.3 Übungsaufgaben zum SVNR

9.3.1 Einfache Maschinenprogramme

Was machen die folgenden Befehlssequenzen?

	Adresse	Inhalt	Mnemo
a)	0000	1874	
	0001	3030	
	0002	2821	

	Adresse	Inhalt	Mnemo
b)	0000	1112	
	0001	3013	
	0002	2821	

	Adresse	Inhalt	Mnemo
c)	0000	1800	LDA 00
	0001	2830	STM 30
	0002	1130	LDM 30
	0003	3801	INC
	0004	2830	STM 30
	0005	4802	JA 02

9.3.2 Zusatzübungen

- A.13: Lesen von Assemblerprogrammen zum SVNR
- A.14: Programmierung im Assembler-Code des SVNR
- A.15: Realisierung der Multiplikation in einem Rechenwerk
- A.16: Assemblerprogramm zur Feldanalyse

9.4 Der SVNR als IP-Core-Implementierung

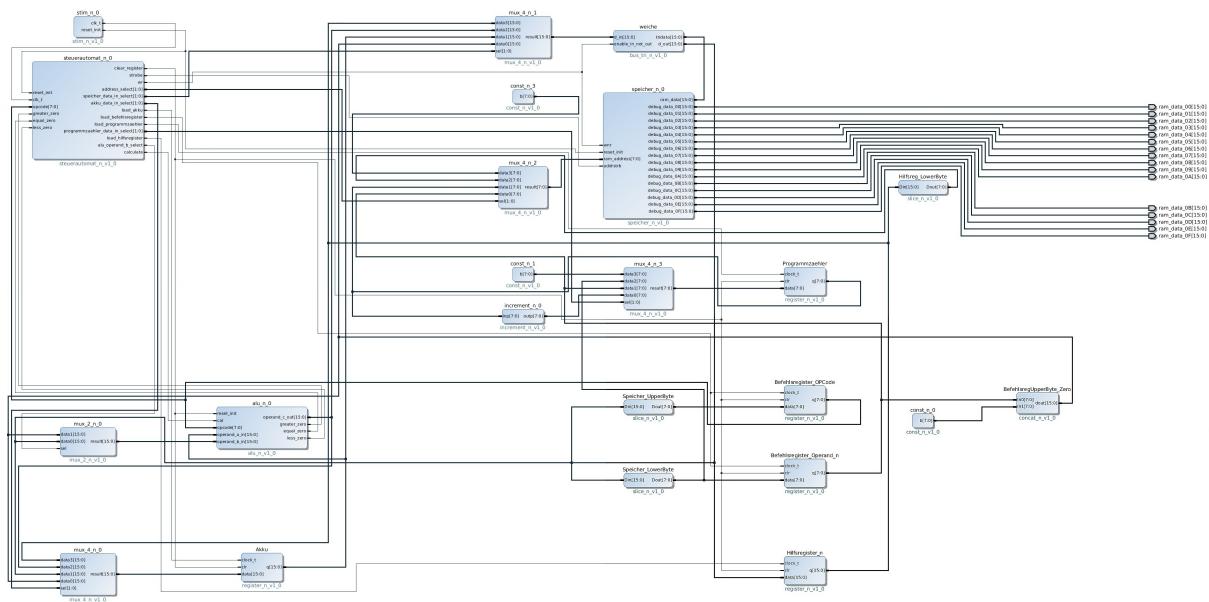


Abbildung 42: Komplettes SVNR Blockschaltbild

Den SVNR gib es nicht nur als Simulation, sondern dieser steht auch als „echte“ Implementierung zur Verfügung. Hier wird der SVNR mittels Schaltplan und VHDL beschrieben.

Die Gesamtdarstellung mit allen Komponenten finden Sie in Abbildung 42. Die einzelnen Komponenten werden auf den folgenden Seiten näher beschrieben. Jede Komponente wird dort zusammen mit den entsprechenden „steuerbaren Datenpfaden“ vorgestellt. Für das Lesen des Schaltplans gilt die folgende Regel: Eingänge stehen links, Ausgänge rechts an den Symbolen.

Signalbündel mit einer Anzahl von $n+1$ Signalen, wie man sie in VHDL mit (`n downto 0`) definiert, werden hier als `[n:0]` dargestellt.

Zuvor werden die häufig verwendeten Symbole kurz erläutert.

9.4.1 Verwendete Symbole

Zur Implementierung der Datenpfade und der Register kommen die folgenden aufgeführten Symbole intensiv zum Einsatz.

Multiplexer Die steuerbaren Datenpfade werden mittels Multiplexer realisiert.

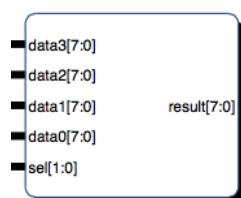


Abbildung 43: Multiplexer

Der hier dargestellte Multiplexer hat

- vier 8 Bit breite Dateneingänge (z.B. `datax[7:0]`),
 - einen 2 Bit breiten Steuereingang (`sel[1:0]`),
 - einen 8 Bit breiten Datenausgang (`result[7:0]`)

Der Multiplexer schaltet, abhängig von dem am Steuereingang anliegenden Bitmuster, einen der vier Dateneingänge auf den Datenausgang. Dies geschieht wie folgt:

sel[1]	sel[0]	result
0	0	data0
0	1	data1
1	0	data2
1	1	data3

Multiplexer mit nur zwei Dateneingängen verfügen nur über einen 1 Bit breiten Steuereingang.

Register Zur Implementierung der verschiedenen Register des SVNR kommt das folgende Symbol zum Einsatz.

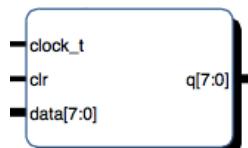


Abbildung 44: Register

Hierdurch wird ein 8 Bit breites Register definiert, das auf Basis von D-Flip-Flops aufgebaut ist. Mit einer steigenden Flanke an `clock_t` werden die an `data[7:0]` anliegenden Eingabedaten im Register gespeichert. Die gespeicherten Daten liegen eine gewisse Zeit später (z.B. nach 3 ns) an den Datenausgängen `q[7:0]` an und können von dort gelesen werden.

Die ins Register zu schreibenden Daten müssen eine gewisse Zeit (z.B. 3 ns) vor und nach der steigenden Flanke von `clock_t` gültig sein.

Mit dem Clear-Signal `clr` kann das Register (bzw. alle seine D-Flip-Flops) asynchron, also unabhängig vom Takt (`clock_t`) auf einen festen Wert (typischerweise 0) gesetzt werden.

9.4.2 Speicher mit Multiplexer

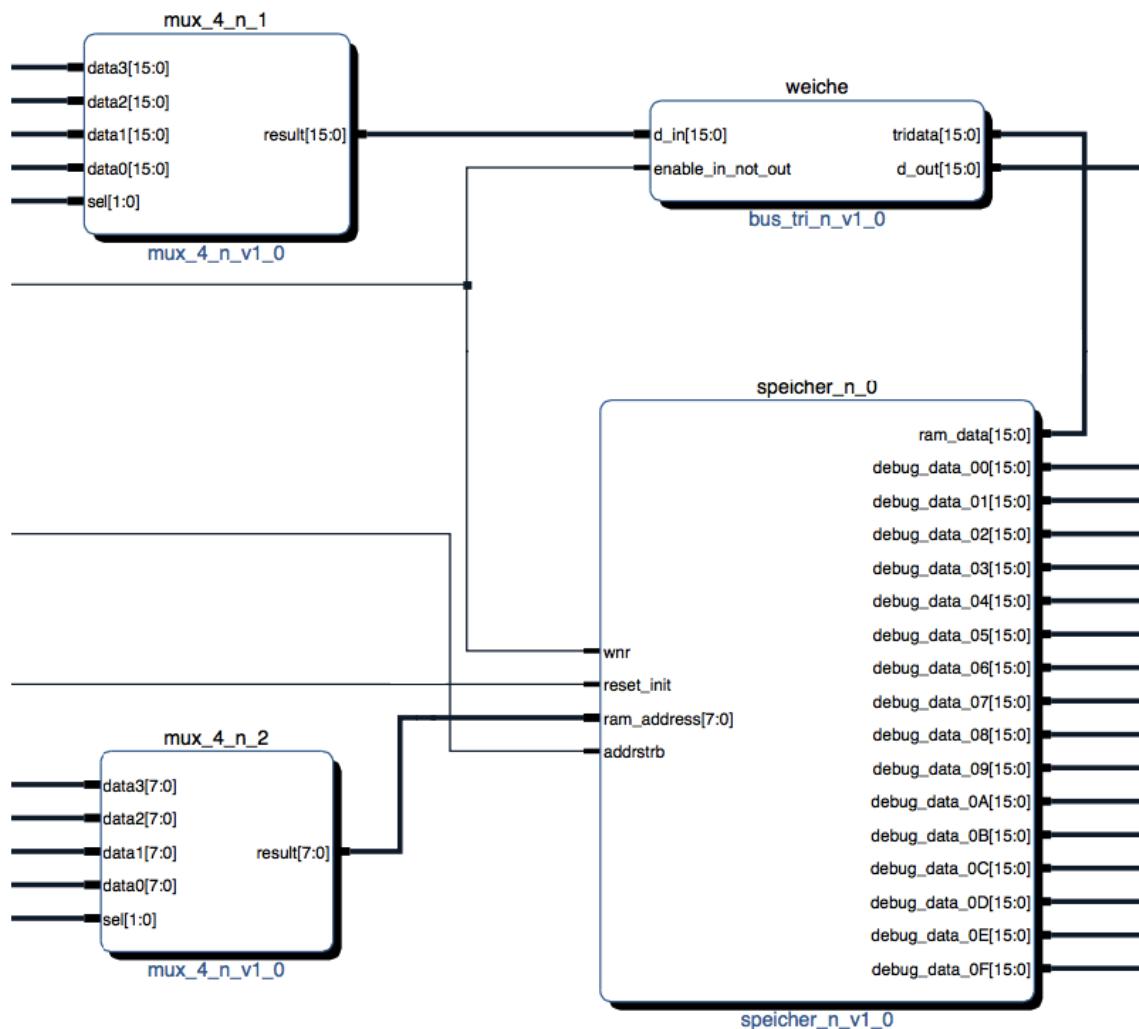


Abbildung 45: Speicher mit Multiplexer

Der Speicher **speicher_n_0** verfügt über die schon beschriebene Schnittstelle (**addrstrb**, **wr**, **ram_address[15:0]**, **ram_data[15:0]**) und muss mit dem ebenfalls schon beschriebenen Timing genutzt werden. Siehe Kapitel 9.1.3.

Der Datenein- bzw. -ausgang (**ram_data[15:0]**) wird nicht direkt angesteuert, sondern über eine Weiche die eine bidirektionale Richtungssteuerung vornimmt. Mit dem Signal **wr** das am Eingang **enable_in_not_out** der Weiche anliegt, wird über interne Tri-State-Buffer bestimmt, welcher dieser beiden Busse (**d_in[15:0]** oder **d_out[15:0]**) über **tridata[15:0]** auf **ram_data[15:0]** geschaltet wird. Hierdurch wird vermieden, dass der Ausgang **ram_data[15:0]** des Speichers gegen den Ausgang **result[15:0]** des Multiplexers **mux_4_n_1** arbeitet.

Von welcher Komponente der Speicher die Eingabedaten (**d_in[15:0]**) empfängt, wird durch **sel[1:0]** mittels des Multiplexers **mux_4_n_1** bestimmt. Von welcher Komponenten der Speicher die Adressen (**ram_address[7:0]**) erhält wird durch **sel[1:0]** über den Multiplexer **mux_4_n_2** definiert.

9.4.3 Programmzähler mit Multiplexer und Addierer

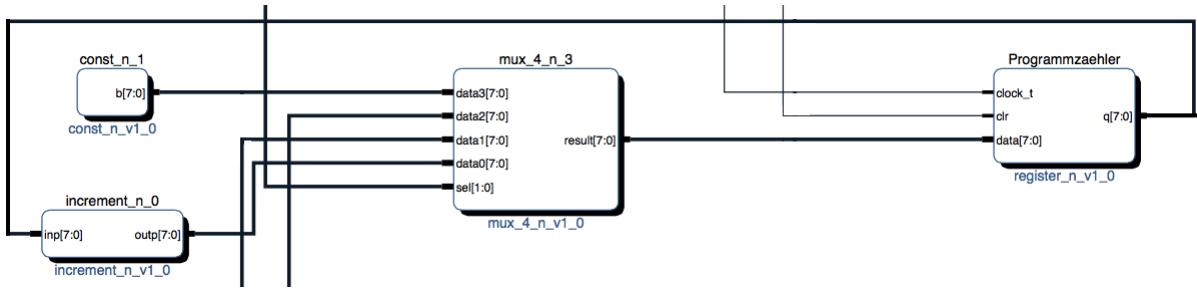


Abbildung 46: Programmzähler mit Multiplexer und Addierer

Der eigentliche Programmzähler ist ein 8 Bit breites Register, das auf Basis des schon beschriebenen Symbols aufgebaut ist.

Von welcher Komponente das Register die Eingabedaten $data[7:0]$ empfängt, wird durch $sel[1:0]$ mittels des Multiplexers **mux_4_n_3** bestimmt.

Der Addierer **increment_n_0** addiert den aktuellen Inhalt des Programmzählers ($q[7:0]$ bzw. $inp[7:0]$) mit dem konstanten Wert 1 und stellt das Ergebnis am Multiplexer-Eingang $data0[7:0]$ zur Verfügung.

Die Komponente **const_n_1** erzeugt einen einstellbaren zeitlich konstanten Bitvektor $b[7:0]$, der typischerweise den Wert 0 hat.

Was muss getan werden, damit der um 1 erhöhte Wert wieder in den Programmzähler gelangt?

9.4.4 Befehlsregister und Hilfsregister

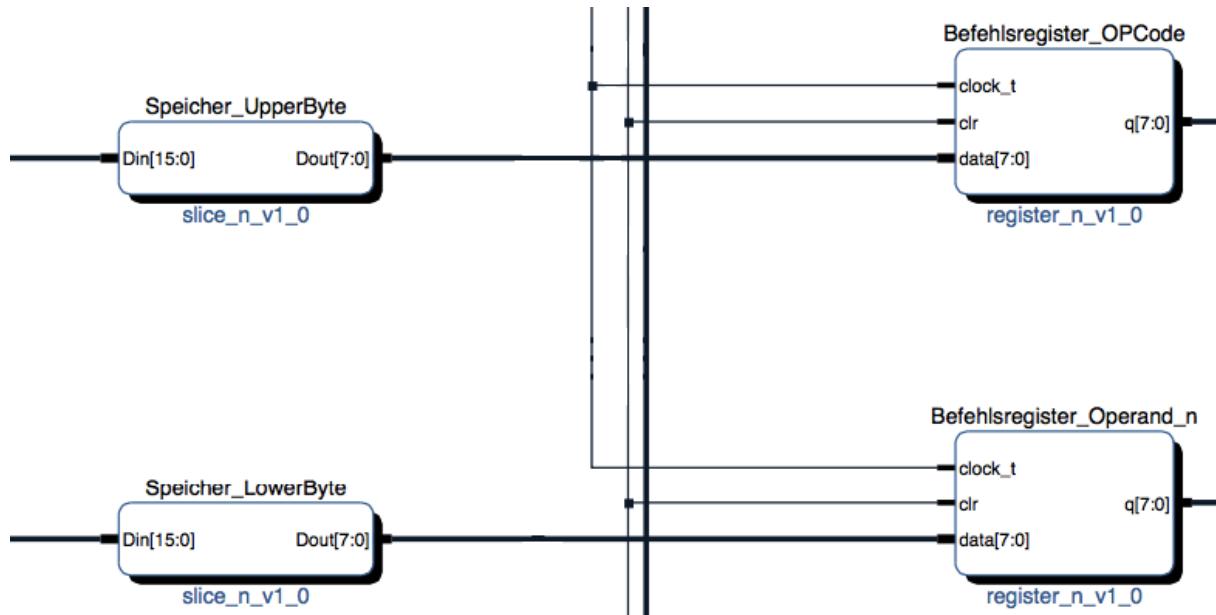


Abbildung 47: Befehlsregister und Hilfsregister

Das 16 Bit breite Befehlsregister ist auf Basis von zwei 8 Bit breiten Teil-Register (entsprechende Symbole) aufgebaut. Das erste Teil-Register enthält die oberen 8 Bit (den OP-Code), das zweite Register die unteren 8 Bit (den Operanden).

Das Befehlsregister und auch das Hilfsregister erhalten ihre Eingabe-Daten ausschließlich vom Speicher. Deshalb ist hier kein Multiplexer vorgeschaltet dafür aber zwei Module (Speicher_UpperByte, Speicher_LowerByte) die aus den von dem Speicher kommenden 16-Bit-Daten das obere und das untere Byte selektieren. Die in das Befehlsregister zu schreibenden Daten müssen somit an deren Eingänge Din[15:0] angelegt werden.

Die von den beiden Teil-Registern ausgegebenen Daten q[7:0] stellen den Inhalt des Befehlsregister dar.

Das Hilfsregister (Hilfsregister_n) ist auf Basis eines 16 Bit breiten entsprechenden Symbols aufgebaut.

9.4.5 Akkumulator und Multiplexer

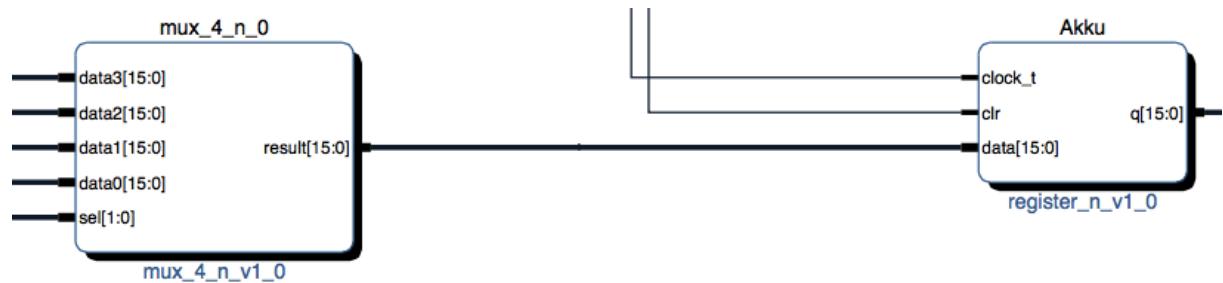


Abbildung 48: Akkumulator und Multiplexer

Der eigentliche Akkumulator ist ein 16 Bit breites Register, das auf Basis des schon beschriebenen Symbols aufgebaut ist.

Von welcher Komponente das Register die Eingabedaten data[15..0] empfängt, wird durch sel[1:0] mittels des Multiplexers mux_4_n_0 bestimmt.

9.4.6 Rechenwerk (ALU) und Multiplexer

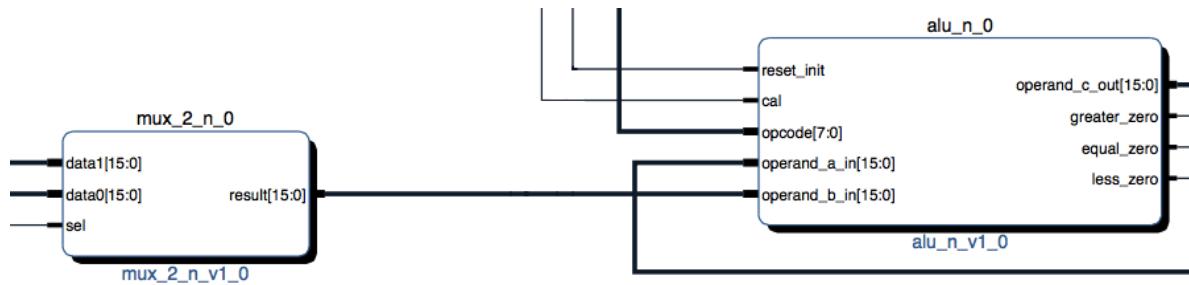


Abbildung 49: Rechenwerk (ALU) und Multiplexer

Die ALU (Arithmetic Logic Unit) oder auch Rechenwerk genannt, berechnet aus den beiden Eingabe-Werten A (operand_a_in[15..0]) und B (operand_b_in[15..0]) den Ausgabewert C (operand_c_out[15..0]). Der Operand A ist immer der Inhalt des Akkumulators, der Operand B kann mittels sel über den Multiplexer mux_2_n_0 ausgewählt werden.

Welche Operation durchgeführt wird, wird durch opcode[7..0] bestimmt. Hierbei wird der OP-Code des entsprechenden Maschinenbefehls direkt ausgewertet. Welche Operation bei welchem OP-Code durchgeführt wird, ist also der Tabelle zum Befehlssatz des SVNR zu entnehmen. Beispiel: Hat opcode[7..0] den hexadezimalen Wert 0x30 (0x33), so wird eine Addition (Subtraktion) durchgeführt, d.h. es gilt $C = A+B$ ($C = A-B$).

Die drei Ausgabesignale (Flags) greater_zero, equal_zero und less_zero geben an, ob das Ergebnis der Operation größer, gleich oder kleiner Null ist.

Das Ergebnis der Operation und auch die drei Flags werden nicht sofort aktualisiert, sondern erst mit einer steigenden Flanke am Eingangs-Signal cal. Der Opcode und die beiden Operanden müssen eine gewisse Zeit (z.B. 10 ns) vor und nach der steigenden Flanke von cal gültig sein. Mit dem Eingangs-Signal reset_init werden alle Ausgaben (d.h. operand_c_out[15..0], greater_zero, equal_zero und less_zero) unabhängig von cal auf 0 gesetzt.

9.4.7 Steuerwerk

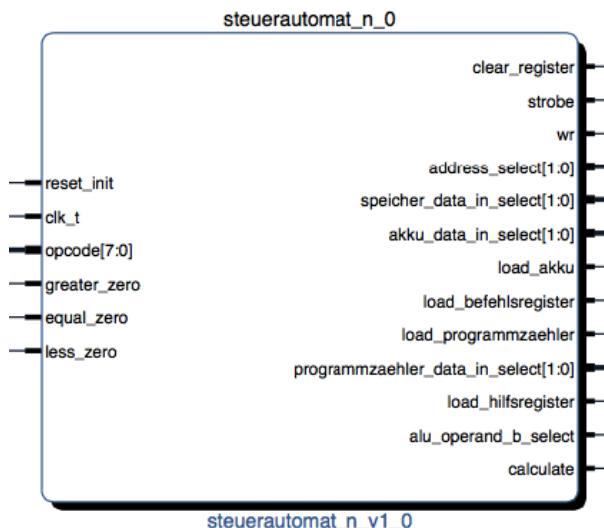


Abbildung 50: Steuerwerk

Das Steuerwerk steuerautomat_n_0 generiert abhängig vom Opcode im Befehlsregister (bzw. Eingangsvektor opcode[7:0]) und den Flags der ALU (greater_zero, equal_zero, less_zero) die rechts am Symbol aufgeführten Ausgabesignale. Dies sind

- die Steuerleitungen der verschiedenen Multiplexer
- die Signale zur Ansteuerung des Speichers und der ALU
- die Signale zum Laden der verschiedenen Register
- ein Signal zum asynchronen Löschen aller Registerinhalte

Das Steuerwerk ist als synchroner Automat realisiert, der mit clk_t getaktet und mit reset_init asynchron in einen Grundzustand versetzt wird. Diese beiden Signale sind die einzigen Eingabesignale des ganzen SVNR.

9.4.8 Hilfskomponenten

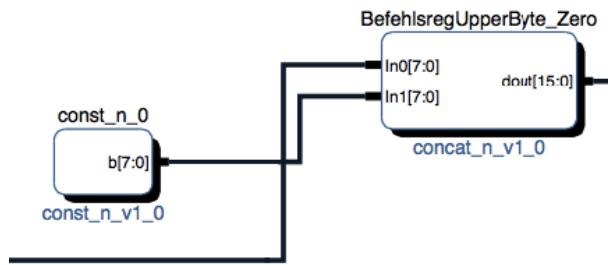


Abbildung 51: BefehlsregUpperByte_Zero

Mit BefehlsregUpperByte_Zero und const_n_0 wird ein neuer 16 Bit breiter Bus erzeugt (dout[15:0]), dessen unteren 8 Bit den Wert des Operanden im Befehlsregister enthalten, dessen oberen 8 Bit allerdings fest auf Null gesetzt sind.

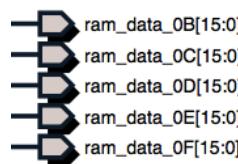
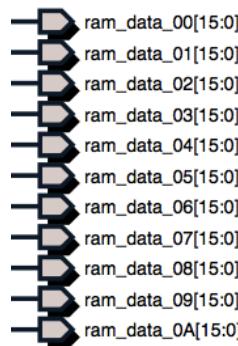


Abbildung 52: Ausgabe der Speicherinhalte zum debuggen

Wie der Name schon angibt, dienen alle hier aufgeführten Ausgabeports dem Debuggen, d.h. deren zeitlicher Verlauf kann nach einer Simulation im Waveform-Editor betrachtet werden und stellt die Speicherinhalte der Speicherzellen mit den Adressen 0x00 bis 0x0F dar.