Hardware Modellierung VL

Spezifikation

Calculator

Gruppen Nr.: 18

Harald Glanzer, Matr. Nr.: 0727156 harald.glanzer@gmail.com
Stefan Simhandl, Matr. Nr.: 0725240 e0725240@student.tuwien.ac.at

Wien, am 30. April 2010

Inhaltsverzeichnis

1 Einleitung - funktionale Beschreibung

Im Rahmen der Laborübung soll ein simpler Calculator entworfen und auf einem FPGA Entwicklerboard implementiert werden.
Bezüglich des Funktionsumfanges gibt es die Forderung, dass der Rechner die mathematischen Operationen Addition, Subtraktion, Multiplikation und Division sowie die damit verbundenen Rechenregeln (Punktrechnung vor Strichrechnung, etc.) beherrschen muss. Weiters sollen die letzten 50 Rechnungen gespeichert werden und die Möglichkeit bestehen, diese über die RS232-Schnittstelle des Boards an einen PC zu senden.
Das Userinterface besteht dabei aus einem PS/2 Keboard zur Eingabe der Rechenoperationen sowie einem Monitor welcher über die VGA Schnittstelle des Boards angesteuert wird. Die LVA-Leitung stellt dabei die Module für den VGA-Treiber sowie für die PS/2 - Schnittstelle zur Verfügung.

Der Rechner

2 Anforderungen im Detail

- Keine Division durch Null(Fehlermeldung)
- Kein Overflow der Variablen(Fehlermeldung)
- Keine Leerzeichen innerhalb einer Zahl(Fehlermeldung)
- Maximale Eingabelänge der Berechnung = 70 Zeichen. Wird diese Anzahl erreicht fährt der Calculator mit der Berechnung der bisher eingegebenen Zeile fort, wenn sich dadurch Syntaxfehler ergeben erfolgt eine Fehlermeldung
- Zwischen 2 Operatoren ist immer nur ein Operand erlaubt. Ausnahme ist das Zeichen für eine negative Zahl '-'
- Der erlaubte Zeichensatz ist auf die Ziffern 0-9, +, -, *, /, <ENTER>, <WHITESPACE> sowie <BACKSPACE> - alle anderen Zeichen werden schon bei der Eingabe abgefangen(ignoriert)
- Keine Leerzeilen(Fehlermeldung)
- Keine alleinstehenden Operatoren(Fehlermeldung)
- Keine alleinstehenden Operanden(Fehlermeldung)

2.1 Aritmetische Ausdrücke

Es sollen folgende Arithmetische Ausdrücke unterstützt werden:

- Addition: ERGEGBNIS = ZAHL1 + ZAHL2
- Subtraktion: ERGEGBNIS = ZAHL1 - ZAHL2
- Multiplikation: ERGEGBNIS = ZAHL1 * ZAHL2
- Division: ERGEGBNIS = ZAHL1 / ZAHL2

2.2 Rechenregeln

Die einzige zu beachtende Rechenregel ist die Regel **Punktrechnung vor Strichrechnung**. Klammerungen sind nicht gefordert und werden folglich auch nicht implementiert.

2.3 Datentypen

Als Datentyp soll 'Signed Long' verwendet werden - damit ergibt sich ein Wertebereich von -2^{31} bis $2^{31} - 1$. Overflows bei der Eingabe werden abgefangen, während Overflows welche sich während der Berechnung des eingegebenen Ausdrucks ergeben **NICHT** abgefangen werden.

2.4 Userinerface

Als Userinterface steht einerseits ein Standardkeyboard zur Verfügung, welches mittels PS/2 - Schnittstelle mit dem FPGA-Board verbunden ist. Die Textausgabe erfolgt auf einem Monitor, welcher mittels VGA-Schnittstelle vom Entwicklungsboard angesteuert wird. Weiters besteht die Möglichkeit einen Datentransfer zu triggern. Dabei sollen die letzten 50 Berechnungen, welche eingegeben wurden, per serieller Schnittstelle an den PC übertragen werden. Dieser Datentransfer soll entweder per Pushbutton oder nach Empfang eines speziellen Zeichens per RS232 angestossen werden. Wir haben uns entschlossen als RS232 - Trigger das Zeichen 's', ASCII-Code 0x73, zu verwenden.

2.4.1 Dateneingabe

Wie gesagt wird hierfür ein PS2 verwendet. Als Interface zw. Keyboardhardware und FPGA-Board wird ein Modul von der LVA-Leitung zur Verfügung gestellt. Dieses Modul liefert einerseits den 8Bit-Wert 'data', welcher einen Scancode darstellt, und andererseits ein Steuersignal 'new_data' welches anzeigt ob ein neuer Scancode zum Auslesen bereit ist. Abbildung 1 zeigt an wann ein neues, gültiges 8Bit - Datenwort gelesen werden darf.

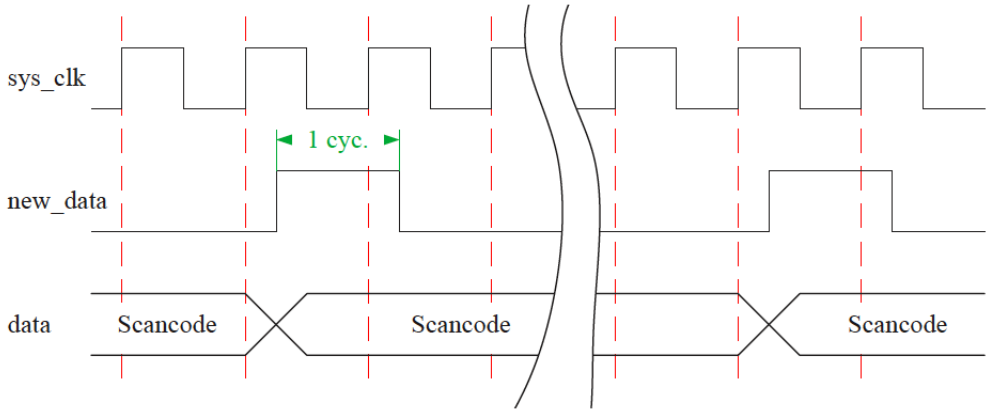

Abbildung 1: PS/2 Schnittstelle Timingdiagramm

Es muss also ein HIGH-Pegel auf new_data anliegen, dann kann bei der nächsten steigenden Flanke der Systemclock ein neuer Scancode gelesen werden. Die Scancodetabelle für eine Standard - PS/2 - Tastatur kann unter

http://www.computer-engineering.org/ps2keyboard/scancodes2.html

abgerufen werden. Für unsere Anwendung ist natürlich nur ein kleiner Teil davon von Interesse. Wir haben uns entschlossen als Eingabetasten nur die entsprechenden Numpad-Tasten zuzulassen, und zusätzlich noch die Leertaste, welche ja auch eine gültige Eingabe darstellt. Die erlaubten Eingaben plus Scancodes für MAKE und BREAK - Events können aus Tabelle 1 entnommen werden:

Prinzipiell müsste also die Logik, welche die ankommenden Scancodes auswertet und auf die den gedrückten Tasten entsprechenden ASCII-Werte mappt zuerst auf einen WAIT-Code warten und dürfte erst bei Erhalt des zugehörigen BREAK-Codes das entsprechende Zeichen abspeichern. Über die Zeitdifferenz zwischen WAIT - und BREAK-Code und einer frei zu wählenden Tastenwiederholungsrate könnte die Anzahl der zu speichernden Zeichen bestimmt werden. Damit würde die zugehörige State-Machine allerdings komplexer als notwendig werden. Wir haben uns deshalb dazu entschlossen, nur die BREAK-Code-Sequenzen der gewünschten Zeichen zu

Zeichen	Wait-Code	Break-Code
KP /	E0,4A	E0,F0,4A
KP EN	E0,5A	E0,F0,5A
KP *	7C	F0,7C
KP -	7B	F0,7B
KP +	79	F0,79
SPACE	29	F0,29
BKSP	66	F0,66
KP 0	70	F0,70
KP 1	69	F0,69
KP 2	72	F0,72
KP 3	7A	F0,7A
KP 4	6B	F0,6B
KP 5	73	F0,73
KP 6	74	F0,74
KP 7	6C	F0,6C
KP 8	75	F0,75
KP 9	7D	F0,7D

Tabelle 1: erlaubte Eingaben plus Scancodes für MAKE und BREAK

detektieren, und alle anderen ankommenden Codes zu ignorieren. Daraus ergibt sich natürlich 1. dass das Zeichen erst erkannt wird wenn die entsprechende Taste wieder losgelassen wird, und 2. dass - egal wie lange die Taste gedrückt wurde, immer nur ein einzelnes Zeichen erkannt wird. Sobald ein neues Zeichen erkannt wurde muss dieses natürlich in einem entsprechenden Buffer abgespeichert werden. Sobald ein <ENTER> - Zeichen erkannt wurde oder die maximale Eingabelänge erreicht wurde wird die Eingabe abgeschlossen.

Abbildung5 zeigt die Funktionalität.

2.4.2 Datenausgabe

Die Ausgabe erfolgt per VGA, wobei dies von 2 verschiedenen Modulen realisiert wird: die eingegebenen Zeichen werden vom Scancode-Modul geschrieben - ist die Eingabe abgeschlossen(<ENTER> bzw. 70 Zeichen eingegeben) wird mittels ASCII-Char 'für <NEWLINE> in eine neue Zeile gewechselt und die Kontrolle an das Parser/Calculator-Modul übergeben. Dieses bestimmt das Ergebnis der aktuellen Eingabe und schreibt diese(bzw. eine Fehlermeldung bei falscher Syntax) an die aktuelle Position im VGA-Buffer. Danach wird wieder mittels <NEWLINE> in die nächste Zeile gewechselt und das Scancode-Modul übernimmt wieder die Kontrolle.

Folgende VGA-Commandos werden benötigt:

- COMMAND_SET_CURSOR_COLUMN
- COMMAND_SET_CURSOR_LINE
- COMMAND_SET_CURSOR_COLOR
- COMMAND_SET_CURSOR_STATE
- COMMAND_SET_CHAR

2.5 Speichern von Rechnungen

Es werden nur gültige Eingabelines gespeichert. Dazu wird nach erfolgter Eingabe vom Parser/Calculator-Modul das Ergebnis berechnet. Danach wird die eingegebene Zeile($Line\_buffer$, Index 0 bis $Line\_ptr$), gefolgt vom Delimiter '=' und dem Ergebnis in die aktuelle Position eines Ringbuffers mit der Grösse 50 x 81 Zeichen geschrieben Die Elementgrösse 81 des Arrays ergibt sich aus den max. 70 einzugegebenen Zeichen, dem Delimiter '=' und dem Ergebnis mit max. 10 Zeichen(2^{31}).

2.6 RS232 Schnittstelle

Es muss sowohl Empfang als auch Senden von Datenbytes per Software-UART realisiert werden. Dazu muss ein Baudrate-Generator implementiert werden. Die Systemclock liefert ein Rechtecksignal mit f=33.33MHz - diese Frequenz muss geteilt werden, um eine gültige Baudrate erreichen zu können.

RS232 ist ein Standard für eine serielle Schnittstelle, wobei im Standard Timing, Spannungspegel, Leitungen und Stecker definiert sind. Die Daten werden Bitseriell und asyncron übertragen, was in Abbildung 2 dargestellt ist. Allerdings ist zu beachten dass RS-232 mit Signalpegeln im Bereich von +3 bis +15 V zur Darstellung einer logischen 0 (SPACE) und -3 bis -15 V zur Darstellung einer logischen 1 arbeitet.

Laut Angabe ist gefordert, die letzten 50 Berechnungen per RS232 versenden zu können, und dieser Sendevorgang soll auch per seriell empfangenen Zeichen angestossen werden können. Daraus folgt dass wir sowohl eine Sende- als auch eine Empfangsroutine für RS232 implementieren müssen. Als Triggerzeichen haben wir uns auf das Zeichen 's' geeinigt. Als Baudrate verwenden wir 38400 (Fehler aufgrund sys_clk am geringsten), als Format 8N1(8 Datenbits, No Paritiy, 1 Stopbit).

Der Teiler berechnet sich mittels $divisior = \frac{Sys\_clk}{Baudrate}$ - Tabelle 2 zeigt die Einstellmöglichkeiten.

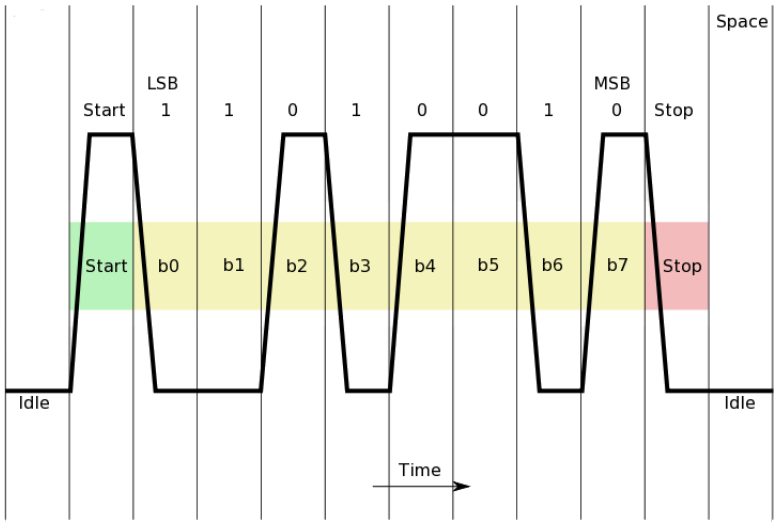

Abbildung 2: Datenframe RS232

Baudrate	Teiler
2400	13887.5
4800	6943.75
9600	3471.875
19200	1735.9375
38400	867.96875
57500	578.6458
115200	289.32

Tabelle 2: mögliche Baudrates

3 Detailierte Design Beschreibung der Module

3.1 Main

Das Programm setzt sich aus dem Scancode-Handler-Modul, dem RS232-Modul(Senden und Empfangen), einem Ringbuffer Modul, einem Line Buffer Modul, Memory Modul sowie dem Parser/Calculator-Modul zusammen, siehe Abbildung 4. Die beiden Puffer instanzieren dabei das Modul Memory.

Beim Einschalten wird ein Reset/Init sämmtlicher verwendeter Speicher durchgeführt. Danach beginnt die Programmausführung mit dem Scancode-Modul, parallel dazu schaltet das RS232-Modul in Empfangsmodus - diese 2 Module werden laufen also parallel ab.

Die Main ist die zentrale Steuereinheit des gesammten Programms. Sie akti-

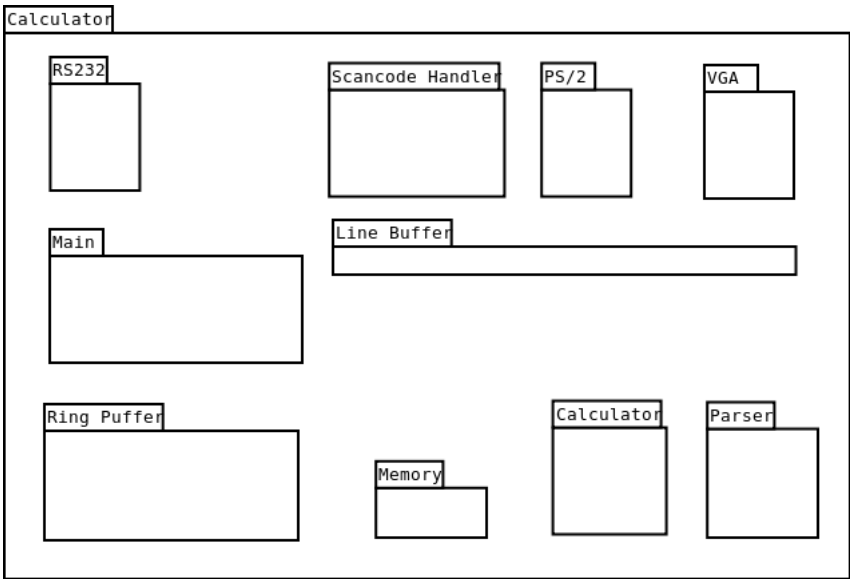

Abbildung 3: Modularer Aufbau der Applikation

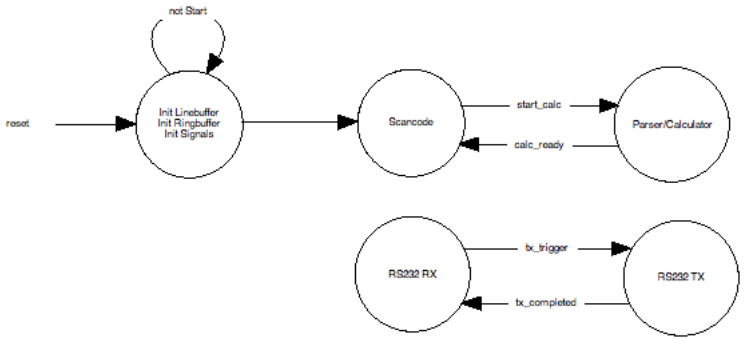

Abbildung 4: Benötigte Programmmodule

viert bzw. deaktiviert (Signal 'enable') den Line Buffer sowie das Calculator Modul. Der Zugriff auf die beiden Speicherbereiche (Lien Buffer und Ringbuffer) sowie die Kommunikation mit dem RS232 Modul wird ebenfalls vom Main Modul geregelt.

3.2 Scancode Handler

In Abbildung5 ist die Funktionalität des Scancode-Moduls abgebildet:

Zuerst werden die verwendeten Puffer und Signale resettet/initialisiert. Danach wird in 'READY' auf eine steigende Flanke von $data\_new$ gewartet. Wenn diese auftritt wird überprüft ob das empfangene Byte hexadezimal gleich F0 oder E0 ist. Ist es gleich F0 wird sofort in den Zwischenzustand 'WAIT_NEXT_DATA' ge-

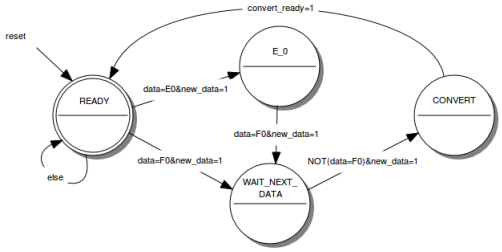

Abbildung 5: Statemachine für das Scancode Handler Modul

wechselt. Da wir die Breakcodes der Zeichen auswerten, ist im Zustand 'E_0' das nächste eingelesene Byte, hexadezimal immer F0. Dieses wird bei der nächsten steigenden Flanke von $data\_new$ einfach ignoriert und ebenfalls in den Zustand 'WAIT_NEXT_DATA' gewechselt. Diesen Zustand haben wir eingefürht da es möglich ist, dass das nächste Zeichen vom PS2 Modul noch nicht zur Verfügung steht obwohl bereits die steigende Flanke von $data\_new$ erkannt wird. Nur wenn das eingelesene Byte nicht F0 ist, wird in den Zustand 'CONVERT' gewechselt und das entsprechende Ascii Zeichen am Ausgang bereit gestellt sowie der Port new_ascii gesetzt. Dadurch wird signalisiert, dass ein neues Zeichen eingelesen wurde und zur weiterverarbeitung bereit steht.

3.3 Parser (Scancode Handler Puffer einlesen & interpretieren)

Abbildung 7 zeigt eine Statemachine für den Parser, in Abbildung 6 sind die Ein- und Ausgänge des Moduls dargestellt (Entity). Der Parser wird vom Calculatormodul verwendet um den mathematischen Ausdruck welchen der Scancodehandler in einem Puffer (70 Zeichen Puffer) in Form von Ascii Zeichen zur Verfügung stellt auszuwerten und zu interpretieren.
Im Folgenden wird die Funktionalität der Parser Statemachine sowie die Abläufe inerhalb der einzelnen States genauer beschrieben.

- State: READY (Initial- und Endzustand)
  Vom Calculatormodul wird mittels dem Signal read_next_num&op der nächste/erste Operand und Operator angefordert.
  Die Statemachine bleibt im READY Zustand solange keine neue Anforderung vom Calculatormodul kommt.

- State: CHECK UNSIGNED
  Sobald vom Calculatormodul eine neue Zeichenauswertung angefordert wird wechselt die Statemachine in den Zustand CHECK UNSIGNED. In diesem Zustand wird überprüft ob der nächste Operand vorzeichenbehaftet ist. Der Ablauf in diesem State kann wie folgt beschrieben werden.

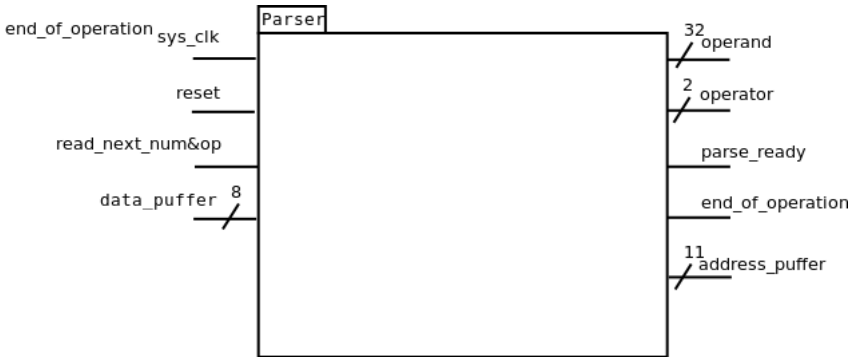

Abbildung 6: Entity des Parsers

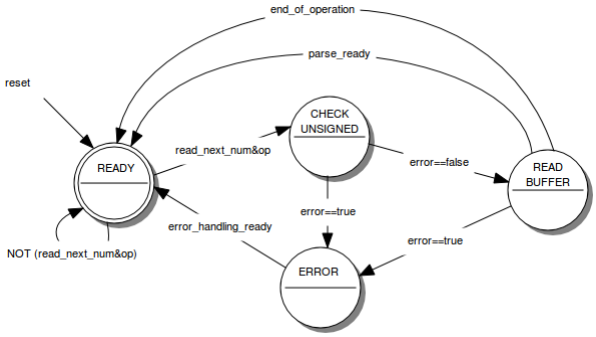

Abbildung 7: Statemachine für das Modul Parser

Zuerst wird das nächste Zeichen geprüft, ist dieses Zeichen eine Ziffer von 0 bis 9 wird der Puffer Positionspointer nicht incrementiert und sofort in den Zustand READ BUFFER gewechsetl.
Ist das nächste Zeichen kein Minus und auch keine Ziffer von 0 bis 9, so liegt ein ungültiger mathematischer Ausdruck vor und es wird in den Zustand ERROR gewechselt.
Ist das nächste Zeichen ein Minus, wird der Positionspointer um eins incrementiert und das nächste Zeichen überprüft. Wenn dieses Zeichen nun eine Ziffer von 0 bis 9 ist, wird das Signal 'leading_sign' gesetzt welches anzeigt, dass der nächste Operand vorzeichenbehaftet und in den Zustand READ BUFFER gewechselt.
Ansonst wird in den Zustand ERROR gewechselt.

- State: ERROR
  In diesem Zustand wird eine Next Line Command an das VGA Modul geschickt, danach die Fehlermeldung 'ungültiger mathematischer Ausdruck' in den VGA Puffer geschrieben und nochmal ein Next Line Command gesendet. Danach wird vom Parser das Calculatormodul resetet und die Statemachine

kehrt wieder in den Zustand READY zurück.

- State: READ BUFFER
  In diesem Zustand wird zuerst der aktuelle Wert des Pufferpointers (der Werd des Puffer Pointers wird vom Parser intern gespeichert un bei einem Reset mit 0 initialisiert) in einer Variablen (im folgenden pos_var genannt) zwischengespeichert. Anschließend wird mit Hilfe dieser Variablen der Puffer so weit durchlaufen bis ein Leerzeichen bzw. ein Operator auftritt oder das Pufferende erreicht wird. Aus der Differenz vom aktullen Wert der Variablen pos_var und dem Puffer Pointer Wert kann ermittelt werden, ob es sich um einen gültigen Operanden handelt (zu viele Stellen => zu großer nummerischer Wert). Diese Überprüfung wird immer ausgeführt nachdem das Ende eines Operanden erkannt wurde. Ist der Operand zu groß wird in den ERROR Zustand gewechselt.

  - Tritt ein Leerzeichen oder mehrere Leerzeichen hintereinander auf, muss überprüft werden ob das nachfolgende Zeichen eine Ziffer oder ein Operator ist. Handelt es sich um eine Ziffer so wird in den ERROR State gewechselt, handelt es sich um einen Operator wird dieser zwischengespeichert sowie die bisher eingelesenen Ziffer umgewandelt und ebenfalls zwischengespeichert. Falls das Signal 'leading_sign gesetzt ist muss der umgewandelte Wert mittels Zweierkomplement vor der Speicherung in seine negative Darstellung gebracht werden. Danach kann 'leading_sign' wieder deaktiviert werden.
    Um welchen Operator es sich handelt wird mittles 2 Bit unterschieden. Die Zuteilung der Bitkombinationen zu den Operatoren is in Tabelle 3.3 dargestell.
  - Tritt beim Pufferdurchlauf ein Operator auf, so hat man das Ende des nächsten Operanden ermittelt. Der Operator kann sofort zwischengespeichert werden. Handelt es sich um einen gültigen Operanden werden die Zifferen in einen Integer Wert umgewandelt und dieser Wert zwischengespeichert. Falls das Signal 'leading_sign gesetzt ist muss der umgewandelte Wert mittels Zweierkomplement vor der Speicherung in seine negative Darstellung gebracht werden. Danach kann 'leading_sign' wieder deaktiviert werden.
  - Erreicht man beim Pufferdurchlauf das Ende des Puffers (pos_var >=70) werden die bisher eingelesenen Ziffern umgewandelt und zwischengespeichert. Falls das Signal 'leading_sign gesetzt ist muss der umgewandelte Wert mittels Zweierkomplement vor der Speicherung in seine negative Darstellung gebracht werden. Danach kann 'leading_sign' wieder deaktiviert werden.

Mit dem Signal 'parse_ready' kehrt der Parser in den READY State zurück, zusätzlich wird einerseits das Signal 'end_of_operation' aktiviert und somit der Calculator über das Ende der Rechnung informiert, andererseits muss der Puffer Pointer zurückgesetzt werden (um bei einer neuen Rechnung wieder von Anfang an lesen zu können) und der Puffer mit Leerzeichen gefüllt werden (damit auch beim nicht Ausnutzen des gesammten Puffers das Ende einer Rechnung erkannt wird).

Nach erfolgreicher Zwischenspeicherung von Oerator und Operand kann in den Zustand READY gewechselt werden.

Operator	Bit-Code
+	00
-	01
	10
/	11

Tabelle 3: Zuweisung der Bitkombinationen zu den Operatoren

3.4 Calculator

Die Aufgabe des Calculatermoduls ist es mittels des Modules Parser immer einen Operanden und einen Operator einzulesen und diese richtig zu verarbeiten. Das Calculatormodul verwendet dazu 4 Speicherbereiche (wie in Abbildung 8 dargestellt). Damit wird je ein Operator und ein Operand für die Punktrechnung sowie für die Strichrechnung gespeichert.
Für die Operanden ist dabei ein 64 Bit Puffer vorgesehen, für die Operatoren ein 2 Bit Puffer.
Das Funktionsprinzip wird im folgenden an der State Machine (siehe Abbildung 9) erläutert, in Abbildung 10 sind die Ein- und Ausgänge des Moduls dargestellt (Entity).

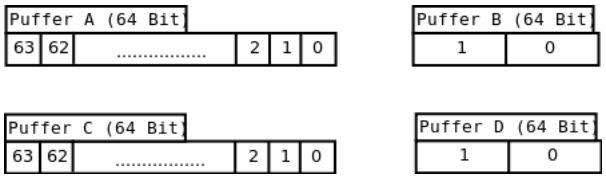

Abbildung 8: Puffer die das Calculator Modul verwendet

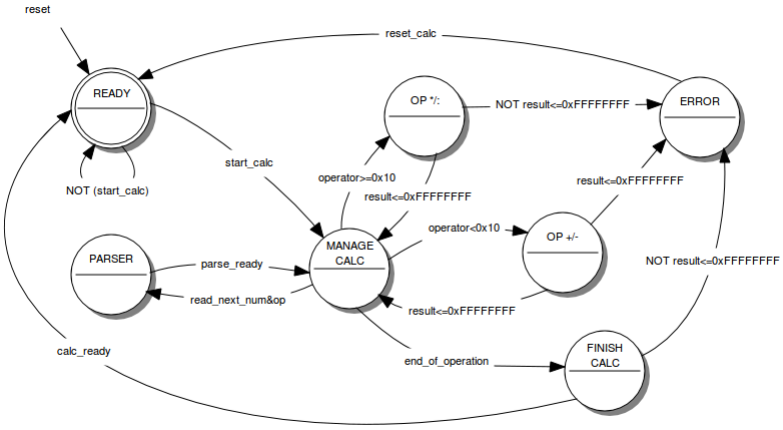

Abbildung 9: Statemachine für das Modul Calculator

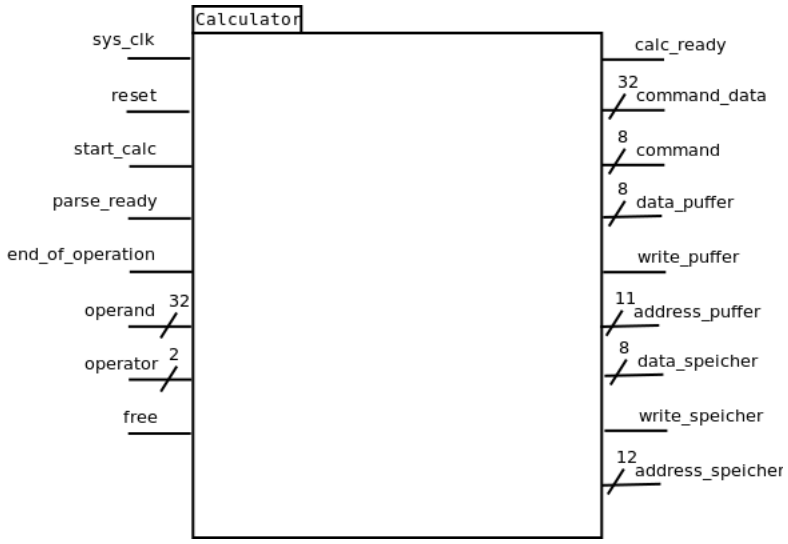

Abbildung 10: Entity des Calculatormodules

- State: READY
  Der Anfangs und Endzustand des Moduls ist der READY State. Das Modul verharrt in diesem Zustand solange nicht durch das Signal 'start_calc' eine neue Berechnung angestoßen wird. Es kehrt wieder in diesem Zustand zurück, wenn eine Berechnung abgeschlossen ist (letzter Operand verarbeitet ... wird durch Signal 'clac_ready' angezeit) oder es z.B. aufgrund eines Fehlers resetet wird.

- State: MANGAGE CALC
  Wird eine neue Berechnung gestartet wechselt das Calculatormodul in den MANAGE CALC Zustand. Dies ist der zentrale Zustand für die gesammte

Berechnung. Aus diesem Zustand heraus werden nächster Operator und Operand angeforder (Anforderung: Signal 'read_next_num&op', OP und NUM vorhanden: Signal 'parse_ready'), sowie überprüft ob es sich beim gerade eingelesenen Operator um einen Punktrechnungs- oder einen Strichrechnungsoperator handelt. Je nach Operatortyp wird dann entweder in den Zustand 'OP_+/-' oder in den Zustand 'OP_*/:' gewechselt.
Ist zusätzlich das Signal 'end_of_operation' aktiv wird in den Zustand FINISH CALC gewechselt und die Berechung abgeschlossen.

- State: PARSER
  Wenn das Modul im Zustand MANAGE CALC ist kann es mit Hilfe des Parsers einen neuen Operator und Operanden anfordern (Signal 'read_next_num&op). Das Modul wechselt in den State PARSE und verharrt dort bis die Anforderung abgearbeitet ist. Durch das Signal 'parse_ready' wird angezeigt, dass ein neuer Operand sowie ein (optionaler) neuer Operator zur Verarbeitung bereit stehen, es wird wieder in den MANAGE CALC State gewechselt.

- State: OP_*/:
  In diesem State muss zuerst überprüft werden, ob bei einem vorliegende Divisionsoperator eine 0 als Operand eingelesen wurde (Division durch 0). Ist dies der Fall wird sofort in den ERROR State gewechselt. Ist dies nicht der Fall kann zwischen 2 möglichen Szenarien unterschieden werden. Die Pufferbezeichnungen A, B, C und D beziehen sich auf Abbildung 8.

  1. Falls noch keine Operation vorgemerkt ist, wird der Operand in einem 64 Bit Puffer (Puffer A) gespeichert und er Operator in einem 2 Bit Puffer (Puffer B).

  2. Ist jedoch bereits eine Operation vorgemerkt, wird der vorgemerkte Operand (Puffer A), abhängig vom vorgemerkten Operator (Puffer B), mit dem aktuellen Operanden verarbeite (Berechnung wird durchgeführt) und in Puffer A gespeichert.
     Danach wird überprüft ob der Wert im 64 Bit Puffer (Puffer A) kleiner oder gleich 0xFFFFFFFF. Ist dies der Fall, ist das Ergebnis mit einer 32 Bit Variablen darstellbar und der Aktuelle Operator kann in Puffer B gespeichert werden.
     Ist dies nicht der Fall, wird sofort in den ERROR State gewechselt.

- State: OP_+/-
  In diesem State muss zwischen 4 möglichen Startszenarien unterschieden werden. Die Pufferbezeichnungen A, B, C und D beziehen sich auf Abbildung 8.

1. Es ist weder eine Punktrechnung, noch eine Strichrechnung gespeichert (alle 4 Puffer sind 0)
   In diesem Fall wir der Operand in Puffer C gespeicher und der Operator in Puffer D.
2. Es ist keine Punktrechnung gespeichert jedoch bereits eine Strichrechnung (Puffer A und Puffer B sind 0, Puffer C und Puffer D sind ungleich 0)
   In diesem Fall wird genau wie im vorigen Szenarion vorgegangen.
3. Es ist bereits eine Punktrechnung gespeichert, jedoch keine Strichrechnung (Puffer A und Puffer B sind ungleich 0, Puffer C und Puffer D sind 0)
   Bei diesem Szenario muss zuerst überprüft werden ob bei einem vorliegende Divisionsoperator eine 0 als Operand eingelesen wurde. Ist dies der Fall, wird sofort in den ERROR State gewechselt. Andernfalls wird der vorgemerkte Operand (Puffer A), abhängig vom vorgemerkten Operator (Puffer B), mit dem aktuellen Operanden verarbeite (Berechnung wird durchgeführt) und in Puffer C gespeichert.
   Danach wird überprüft ob der Wert im 64 Bit Puffer (Puffer C) kleiner oder gleich 0xFFFFFFFF. Ist dies der Fall, ist das Ergebnis mit einer 32 Bit Variablen darstellbar und der Aktuelle Operator kann in Puffer D gespeichert werden.
   Puffer A und Puffer B müssen gelöscht werden (logische Und- Verknüpfung mit 0)
4. Es ist sowohl eine Punktrechnung als auch eine Strichrechnung vorgemerkt (Alle 4 Puffer sind ungleich 0)
   Hier muss zuerst wieder überprüft werden, ob eine Division durch 0 vorliegt. Ist in Puffer B ein Divisionsoperator gespeichert und der aktuelle Operand gleich 0 so wird sofort in den ERROR State gewechselt. Andernfalls wird der vorgemerkte Operand (Puffer A) abhängig vom vorgemerkten Operator (Puffer B), mit dem aktuellen Operanden verarbeite (Berechnung wird durchgeführt) und in Puffer A zwischengespeichert. An dieser Stelle muss das Ergebnis wieder dahingehend überprüft werden ob es noch kleiner 0xFFFFFFFF ist und entweder die Berechnung fortgesetzt oder in den ERROR State gewechselt werden. Danach wird der Operand von Puffer A abhängig vom Operator von Puffer D mit dem Operanden von Puffer C verarbeitet und in Puffer C gespeichert.
   Nun wird wieder überprüft ob der Wert im 64 Bit Puffer (Puffer C) kleiner oder gleich 0xFFFFFFFF. Ist dies der Fall, ist das Ergebnis mit einer 32 Bit Variablen darstellbar und der Aktuelle Operator kann in Puffer D gespeichert werden.
   Weiters müssen Puffer A und Puffer B gelöscht werden (logische Und-

verknüpfung mit 0).

- State: FINISH_CALC
  Falls das Signal 'end_of_operation' aktiv ist, wird vom Zustand MANAGE CALC in den Zustand FINISH CALC gewechselt.
  In diesem Zustand wird zuerst überprüft ob eine Punktrechnung vorgemerkt ist (Puffer B ungleich 0). Ist dies der Fall, wird auf eine mögliche Division durch 0 geprüft und falls nötig in den ERROR State gewechselt. Wird nicht in den ERROR State gewechselt muss die Punktrechnung durchgeführt werden und falls auch eine Strichrechnung vorgemerkt ist, diese auch abgearbeitet werden.
  Sind alle Berechnungen abgeschlossen wird das Ergebnis wieder in Asciizeichen umgewandelt und zeichenweise in den VGA Puffer (warten bis VGA Modul frei... Signal 'free', in den VGA Puffer schreiben ... Signal 'command_data' und 'command', siehe Abbildung 10) sowie der gesamte $Line\_buffer$ in den Ringspeicher ('write' Signal, 'addres' Signal und Datenbus 'data', siehe Abbildung 10) geschrieben. Das Signal 'calc_ready' wird aktiviert und das Calculatormodul kehrt wieder in den Zustand READY zurück.

- State: ERROR
  In diesem Zustand wird eine Next Line Command an das VGA Modul geschickt, danach die Fehlermeldung 'Bufferoverflow' in den VGA Puffer geschrieben und nochmal ein Next Line Command gesendet. Danach wird das Calculatormodul resetet und die Statemachine kehrt wieder in den Zustand READY zurück.

3.4.1 Addition

Die Addier-Funktionalität wird mittels Addierer realisiert, hier muss von unserer seite her kein spezieller Aufwand betrieben werden. Wenn in VHDL eine Anweisung der Form ERG = Z1 + Z2 vorkommt erzeugt der Compiler eine entspr. Addierschaltung mit n Bit, wobei hier n der Bitbreite von Z1 und Z2 entspricht.

3.4.2 Subtraktion

Hier gilt das gleich wie bei der Addition, nur dass hier natürlich eine Subtrahierschaltung anstatt einer Addierschaltung erzeugt wird.

3.4.3 Multiplikation

Siehe oben.

3.4.4 Division

Die Divion bildet einen Sonderfall unter den geforderten Rechenoperationen. Prinzipiell könnte diese Rechenoperation - wie bei den 3 vorhergehenden - einfach über die Anweisung ERG = Z1 / Z2 abgebildet werden. Die resultierende Schaltung wäre aufgrund der relativ hohen Bitbreite allerdings sehr komplex, wodurch der kritische Pfad sehr lange werden würde, damit verbunden würde diese Schaltung dann einen entsprechend hohen Delay verursachen. Die Dividier-Operation wird also von einer eigens implementierten Algorithmus realisiert.

3.5 Ringpuffer

Im Ringpuffer (siehe Abbildung 11) werden die letzten 50 gültigen Rechenoperationen inklusive Ergebnisse gespeichert. Zum schreiben auf den Puffer muss das 'write_speicher' Signal aktiviert werden. Zusätzlich muss am Address Port (Signal'address_speicher') die gewünschte Adresse sowie am Dateneingang (Signalbus 'input_speicher') das zu speichernde Byte angelegt werden.

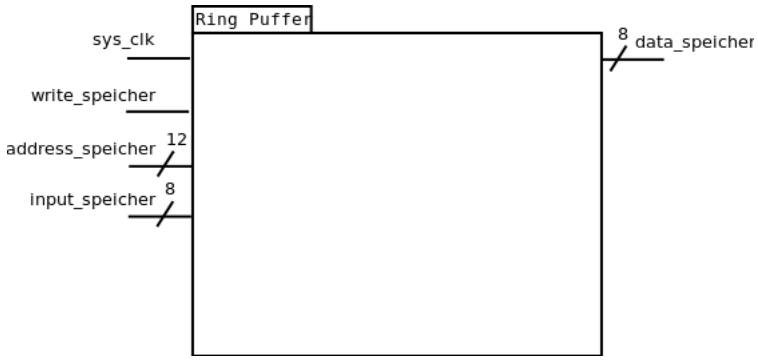

Abbildung 11: Entity Ringpuffer

3.6 Line Buffer

Der Line Buffer ist dafür zuständig die aktuell eingegebene Rechnung zwischen zu speichern. Dafür wird das Memory Modul (für die eigentliche Zwischenspeicherung) sowie das PS2 Modul und das Scancode Handler Modul (für die Eigabe der Zeichen) verwendet. Eine weitere Aufgabe des Line Buffers ist es, die eingelesenen Zeichen mit Hilfe des VGA Moduls am Bildschirm darzustellen. Die dafür benötigte Statemachine ist in Abbildung 12 dargestellt.

Um die Zeichen ENTER und BACKSPACE richtig darzustellen, müssen dem VGA Modul mehr als nur ein Zeichen übergeben werden (ENTER: linefeed und

nextline, BKSP: Cursorposition um eins verrigern, Leerzeichen schreiben und Cursorposition wieder um eins verringern). Wenn für das Schreiben eines Zeichens mehr als ein VGA Befehl benötigt wird braucht man einen Zwischenzustand (WAIT_STATE), da das VGA Modul eine gewisse Zeit braucht um ein Zeichen auf dem Bildschirm abzubilden. Da bei einem Reset der VGA Speicher nicht gelöscht wird, haben wir zusätzlich noch den Zustand CLEAR_SCREEN eingeführ. In diesem Zustand wird 30 mal ein nextline Befehl dem VGA Modul übergeben um bei einem Reset den Bildschirm zu leeren. Danach wird in den eigentlichen Startzustand CHECK_ASCII gewechselt. Die Ein- und Ausgänge des Moduls sind in Abbildung 13 zu finden. Mit Dem Signal 'read_ready' signalisiert der Line Buffer das eine Eingabe abgeschlossen ist und zur weiterverarbeitung (Berechnung) bereit steht. Durch den Eingangsport 'enable' kann der Linebuffer abgeschaltet werden. Dies ist wichtig, da während der Auswertung des einglesenen Ausdrucks dieser nicht verändert werden darf. Erst wenn die Berechnung abgeschlossen ist, darf der Line Buffer wieder Zeichen einlesen. Sobald der Port 'enable' auf 1 gesetzt wird, wird der Speicher mit Leerzeichen überschrieben und der Linebuffer kehrt in den Zustand 'CHECK_ASCII' zurück.

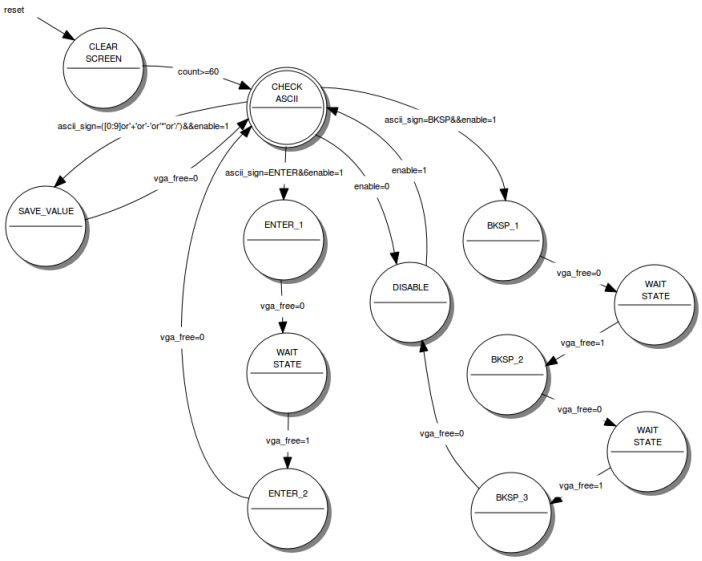

Abbildung 12: State Machine Line Buffer

3.7 Externe Schnittstellen

Hardware spezifisches zu den Schnittstellen, + Logische Implementierung

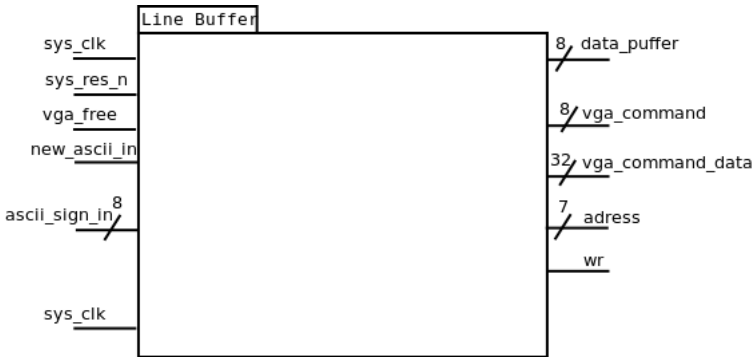

Abbildung 13: Entity Line Buffer

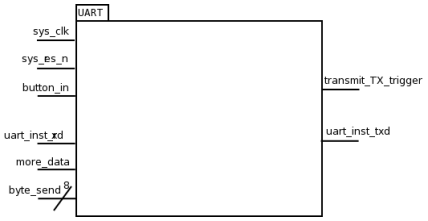

Abbildung 14: Entity der UART Hauptinstanz

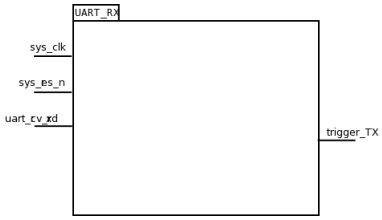

Abbildung 15: Entity der UART-Empfangsinsanz

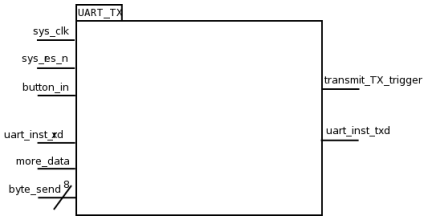

Abbildung 16: Entity der UART-Sendeinstanz

3.7.1 RS232-Schnittstelle

In Abbildung 17 ist die Statemachine des RS232 Moduls zu sehen, Abbildung 14 zeigt Ein- und Ausgänge der Hauptinstanz. Die UART-Funktionalität ist modular aufgebaut, zusätzlich zur UART-Hauptinstanz gibt es die Unterinstanzen

UART_RX 15 und UART_TX 16 . Die Funktionalität wird im Folgenden beschrieben.
Wenn das Board neu gestartet wurde werden zuerst die Buffer für Sende/Empfangsroutinen initialisiert. Danach muss die Empfangsroutine aktiviert werden. Dazu muss ein Baudrategenerator gestartet werden, welcher, sobald ein START-Bit erkannt wurde(wechsel von IDLE/negativer Spannungspegl = logische HIGH auf positiven Pegel), immer möglichst genau in der Mitte der zu empfangenden Bits den aktuellen logischen Pegel auf der RX-Leitung einliest und in einen Buffer speichert(LSB wird immer als erstes gesendet). Danach muss noch ein gültiges STOP-Bit gelesen werden(negativer Spannungspegel). Jetzt muss noch geprüft werden ob dieses eingelesene Byte mit dem Triggerzeichen(bei uns 's') übereinstimmt. Falls ja wird die Main-Instanz per Signal verständigt dass das Triggerzeichen empfangen wurde. Gleiches gilt wenn das Signal für den 'pushbutton' aktiviert wird(Taste muss entprellt werden). Während eines Sendevorgangs können keine Zeichen eingelesen werden, diese Routine 'blockiert' also. Wenn also ein Sendevorgang angestossen wurde versorgt die Main-Instanz das UART-Modul(und in weiterer Folge das UART-TX-Modul) mit den einzelnen Bytes aus der Memory-Instanz(wo die fertigen Berechnungen gespeichert werden). Dazu kopiert zuerst die Main-Instanz das 1. zu sendende Byte in 'byte_out' und setzt das Steuersignal 'more_data' auf HIGH(dieses bleibt HIGH solange Bytes zu senden sind). UART_TX setzt nun das Signal 'tx_busy' und schreibt das Byte auf die Serielle Schnittstelle, sobald die Leitung wieder auf IDLE gesetzt wurde wird 'tx_busy' auf LOW gesetzt. Dies ist das Zeichen für die Main-Instanz, das nächste Zeichen aus dem Speicher zu holen usw.

Eine Zeile wird mittels Newline 0x0A und Carriage Return 0x0D abgeschlossen, diese Sonderzeichen sind nicht im Speicher sondern müssen von der Main-Instanz extra eingefügt werden. Weiters haben wir beschlossen, sobald ein Sendevorgang angestossen wurde, immer den gesamten Ringbuffer(also alle 50 Zeilen zu je max. 80 Zeichen) zu senden, egal wieviele Berechnungen bisher gespeichert wurden.

3.7.2 PS/2-Schnittstelle

Es wird ein Modul von der LVA-Leitung zur Verfügung gestellt welche die Kommunikation Keyboard-Controller / FPGA-Board übernimmt (Abbildung 18 zeigt die Entity es Moduls). Unsere Aufgabe ist es die ankommenden Scancodes auszulesen und auf die entsprechenden ASCII - Codes zu mappen. Siehe Kapitel 'Scancode - Handler'.

3.7.3 VGA-Schnittstelle

Wie gesagt wird ein Modul zur Ansteuerung eines VGA-Monitors von der LVA-Leitung zur Verfügung gestellt (Abbildung 19 zeigt die Entity es Moduls). Dieses

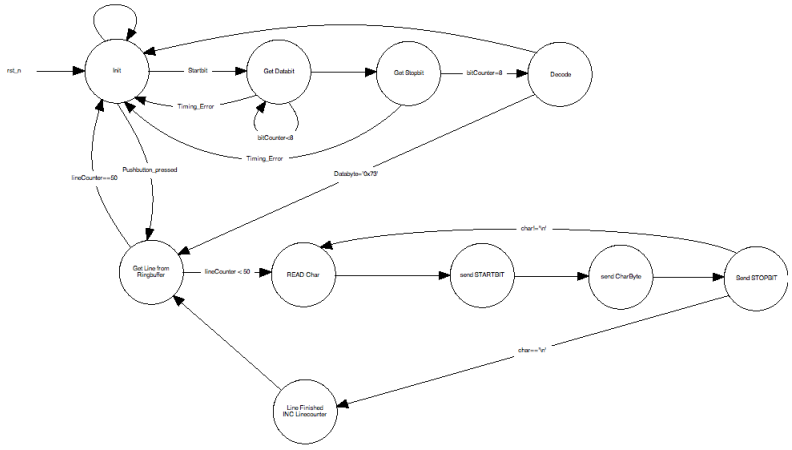

Abbildung 17: Statemachine für Sende- und Empfangsroutine

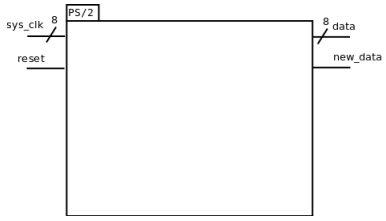

Abbildung 18: Entity des PS/2 Moduls

Modul unterstützt Standard-VGA-Modus, also eine 'Auslösung' von 80 Zeichen pro Zeile, und 30 derartige Zeilen. Als Zeichensatz wird die 'Western European Codepage CP850' unterstützt, womit alle gängigen ASCII-Zeichen dargestellt werden können.

Hintergrundfarbe, Textfarbe und Cursorform können frei gewählt werden, wobei eine Farbauslösung laut Standard-VGA von 8Bit unterstützt wird. Um einen möglichst hohen Kontrast und damit beste Lesbarkeit zu erzielen verwenden wir weissen Text auf schwarzem Hintergrund. Cursorfarbe ist weiss blinkend. Die Textausgabe startet in der 1. Zeile, 5. Zeichen, mit einem blinkenden Cursor. Eingegebene Zeichen werden in einen Buffer geschrieben, und dieser Buffer wird laufend von unserem sync_ps2tovga - Modul mittels dem Kommando 'COMMAND_SET_CHARACTER' in den VGA-Speicher geschrieben. Sobald die Eingabe abgeschlossen ist(entweder mit Eingabe von 'ENTER' oder nach der Erreichen der laut Angabe maximal 70 eingelesenen Zeichen) springt der Cursor in die nächste Zeile und gibt ab x-Position 10 das Ergebnis oder, im Falle einer falschen Eingabesyntax, den Text **'Fehlerhafte Eingabe'** aus. Um den Cursor zu positionieren können in x-Richtung Leerzeichen und in y-Richtung 'CL/LF' verwendet werden, es gibt aber auch spezielle Kommandos um den Cursor an eine beliebige Stelle setzen zu können.

Siehe auch Kapitel 'Scancode - Handler' bzw. 'Calculator'.

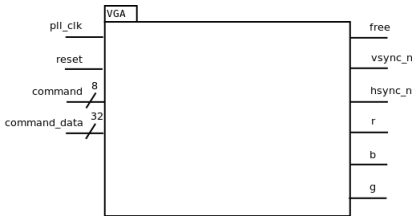

Abbildung 19: Entity des VGA Moduls

4 Testfälle

- Eingabe von nicht definierten Zeichen - müssen ignoriert werden
- Eingabe von zu grossen Zahlen als Operator($z > 2^{31}$) - Fehlermeldung
- Eingabe von Leerzeichen innerhalb von Zahlen - Fehlermeldung
- Eingabe von mehreren Operatoren zwischen 2 Zahlen(Ausnahme: negativ-Zeichen - Fehlermeldung
- Division durch Null - Fehlermeldung
- Eingabe von zu vielen Zeichen - Abbruch der Eingabe nach 70 Zeichen, Berechnung dieses Ausdrucks
- Triggern eines UART_TX per UART_RX
- Triggern eines UART_TX per Pushbutton